JPA
java persistence api

Helder da Rocha

JAVAEE7

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

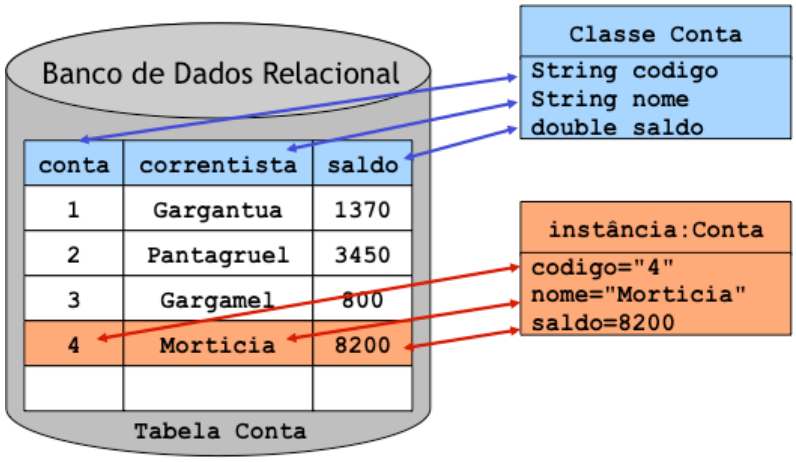
Capítulo 4: Java Persistence API (JPA)

Conteúdo

1 Introdução

JPA – Java Persistence API é uma especificação do Java EE que define um mapeamento entre uma estrutura relacional e um modelo de objetos em Java. JPA pode ser também usado de forma *standalone*, em aplicações Java SE, utilizando um provedor de persistência independente, mas é um componente obrigatório e nativo de qualquer servidor Java EE. O Java EE 7 adota a especificação JPA 2.1 (cuja especificação foi criada através do JSR 338).

JPA permite a criação de objetos persistentes, que retém seu estado além do tempo em que estão na memória. Oferece uma camada de persistência que permite pesquisar, inserir, remover e atualizar objetos, além de um mecanismo de mapeamento objeto-relacional (ORM) declarativo. Mapeamento objeto-relacional em aplicações JPA consiste na declaração de mapeamentos entre classes e tabelas, e atributos e colunas em tempo de desenvolvimento, e da sincronização de instancias e registros durante o tempo de execução.



1.1 Unidade de persistência e persistence.xml

Para configurar JPA em uma aplicação é necessário que essa aplicação tenha acesso, através do seu *Classpath*, aos seguintes componentes (geralmente empacotados em JARs):

- Módulo contendo a *API* do JPA (as classes, interfaces, métodos, anotações do pacote *javax.persistence*)
- Um provedor de persistência JPA que implementa a especificação (ex: Hibernate, EclipseLink)
- Drivers necessários para configuração do acesso aos datasources usados (e pools de conexão, se houver)

Além disso, a aplicação deverá incluir um arquivo *persistence.xml* contendo a configuração da camada de persistência.

Dependendo do provedor de persistência usado, pode haver necessidade de incluir outros JARs e arquivos de configuração. Um arquivo *orm.xml* pode também ser incluído para configuração de mapeamento (como alternativa, ou para sobrepor os mapeamentos declarados nas próprias classes através de anotações).

O *contexto de persistência*, usado para sincronizar as instâncias persistentes com o banco de dados é chamado de *Entity Manager*. Ele controla o ciclo de vida das instâncias. O Entity Manager pode ser instanciado pela própria aplicação (comum em aplicações JPA standalone) ou obtido através de JNDI ou injeção de dependências (padrão em ambientes Java EE).

O conjunto de entity managers e as entidades que eles gerenciam configura uma *unidade de persistência* (*Persistence Unit*). Essa configuração é declarada no arquivo *persistence.xml* e referenciada nas classes que *usam* os objetos persistentes, como DAOs, fachadas, session beans, servlets, etc.

O arquivo *persistence.xml* deve estar no Classpath acessível pelas classes Java usadas na aplicação em */META-INF/persistence.xml*, e possui a seguinte configuração *mínima*:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="MyPU" transaction-type="JTA">   </persistence-unit>

</persistence>
```

Este exemplo é utilizável em um servidor Java EE que tenha sido configurado para oferecer um datasource default (*java:comp/DefaultDataSource*). Normalmente aplicações usam datasources selecionadas de forma explícita. Este segundo exemplo é típico para uma aplicação JPA simples em um servidor Java EE 7:

```
<persistence version="2.1" …>
    <persistence-unit name="com.empresa.biblioteca.PU" transaction-type="JTA">

        <jta-data-source>jdbc/Biblioteca</jta-data-source>
        <class>br.empresa.biblioteca.entity.Livro</class>
        <properties>
            <property name="eclipselink.deploy-on-startup" value="true" />
        </properties>

     </persistence-unit>
</persistence>
```

Ele declara o *tipo de transações utilizada* (JTA – gerenciada pelo container), o nome JNDI de um datasource específico (que precisa ser previamente configurado no servidor de aplicações), uma entidade mapeada, e uma propriedade de configuração do provedor usado (tipicamente, e principalmente quando usa-se um provedor que não é nativo do servidor, há várias outras propriedades específicas do provedor de persistência que precisam ser configuradas).

Neste segundo exemplo temos um *persistence.xml* usado para acessar os mesmos dados através de uma camada de persistência configurada *localmente* (transações gerenciadas pela aplicação), usando provedor *EclipseLink* e banco *PostgreSQL*:

```
<persistence version="2.1"…>
    <persistence-unit name="com.empresa.biblioteca.PU"
                      transaction-type="RESOURCE_LOCAL">

      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
       <class>br.com.argonavis.javaee7.jpa.intro.Livro</class>

       <properties>
            <property name="javax.persistence.jdbc.driver"
                  value="org.postgresql.Driver" />
            <property name="javax.persistence.jdbc.url"
                  value="jdbc:postgresql://localhost:5432/postgres" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="admin" />
            <property name="eclipselink.ddl-generation"
                  value="drop-and-create-tables" />
            <property name="eclipselink.deploy-on-startup" value="true" />
       </properties>
    </persistence-unit>

</persistence>
```

Cada provedor de persistência poderá requerer propriedades específicas. Há quatro propriedades *padrão* para informar os dados de conexão a um banco de dados (*driver*, *url*, *user*, *password*). Essa configuração é mais comum em ambientes de testes, que rodam em Java SE.

A configuração do *persistence.xml* também depende do provedor de persistência usado. Este outro *persistence.xml* configura uma unidade de persistência com transações locais usando um provedor *Hibernate* acessando localmente um banco de dados *MySQL*. Diferentemente do exemplo anterior, este passa as propriedades de configuração do banco via propriedades do Hibernate. O resultado é o mesmo.

```
<persistence …>
    <persistence-unit name="LojaVirtual" transaction-type="RESOURCE_LOCAL">
```

```
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>lojavirtual.Produto</class>

        <properties>
           <property name="hibernate.dialect"
                     value="org.hibernate.dialect.MySQLDialect"/>
            <property name="hibernate.connection.driver_class"
                      value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password" value=""/>
            <property name="hibernate.connection.url"
                      value="jdbc:mysql://localhost:3306/test"/>
         </properties>
    </persistence-unit>
</persistence>
```

1.2 Entidades e mapeamento

A arquitetura do JPA baseia-se no mapeamento de tabelas a classes, colunas a atributos. O objetivo do mapeamento é a construção de uma *hierarquia de entidades*.

Entidade (*Entity*) é o nome dado a um objeto persistente que em JPA é representado por um JavaBean (ou POJO) mapeado a uma tabela (um JavaBean/POJO é basicamente uma classe Java com atributos privativos acessíveis via métodos get/set e que contém um construtor default sem argumentos.) Uma entidade JPA pode ser mapeada a uma tabela de duas formas:

- Através de elementos XML no arquivo *orm.xml*
- Através de anotações antes de declarações de classes, métodos, atributos, construtores.

Instruções de mapeamento podem ser incluídas associando classes a tabelas, atributos a colunas, etc. A listagem abaixo ilustra um exemplo de mapeamento declarado em *orm.xml*. Existem defaults para praticamente tudo. Os únicos tags obrigatórios são <*entity*> e <*id*>, e os atributos *name*. Pode-se incluir um nível maior de detalhamento (ex: tipos de dados, constraints) ou menor (ex: omitir tags com nomes das tabela e colunas, se forem iguais).

```
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm/orm_2_1.xsd"
    version="2.1">
    <entity name="Livro" class="br.com.argonavis.Livro" access="FIELD">
        <table name="LIVRO"/>
        <attributes>
            <id name="id">
                <column name="ID"/>
                <generated-value strategy="TABLE" generator="LIVRO_SEQ"/>
            </id>
            <basic name="titulo">
                <column name="TITULO"/>
```

```
            </basic>
            <basic name="paginas">
                <column name="PAGINAS"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```

Não é necessário usar o arquivo *orm.xml* nem mapeamentos em XML para criar entidades. Os mapeamentos podem ser declarados através de anotações nas próprias classes Java que representam os objetos persistentes. Esse é o procedimento mais comum e recomendado. Como o mapeamento em XML tem *precedência* sobre o mapeamento realizado via anotações, ele geralmente só é usado quando é necessário sobrepor ou refinar o mapeamento default em uma aplicação existente.

Para declarar uma entidade com anotações JPA é necessário no mínimo anotar a classe com *@Entity* e anotar pelo menos um atributo (ou método) com *@Id*, indicando sua chave primária.

```
@Entity
public class Livro implements Serializable {

    @Id
    private Long id;
    private String titulo;
    private int paginas;

    public Long getId() {
            return id;
    }

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

    public String getTitulo() {
            return this.titulo;
    }

    public void setTitulo(String titulo) {
            this.titulo = titulo;
    }
    public int getPaginas() {
            return this.paginas;
    }

    public void setPaginas(int paginas) {
            this.paginas = paginas;
    }
```

```
}
```

Por default, todos os atributos da classe, expostos via métodos get/set serão mapeados a colunas de mesmo nome. Ou seja, no exemplo acima seriam mapeadas as colunas: *ID*, *TITULO* e *PAGINAS*. O nome da classe será usado por default como nome da tabela. Se o nome da tabela e o nome da classe coincidir, e se coincidirem os nomes e das colunas com os nomes dos atributos, e seus tipos forem compatíveis, é possível que o mapeamento seja suficiente, e não haja a necessidade de configuração adicional.

O valor e o tipo usado no ID é importante para a operação do mecanismo de persistência transitiva no JPA, portanto, é comum que seja um valor sequencial gerado automaticamente. A estratégia de geração da seqüencia, que muitas vezes depende do banco usado, também pode ser declarada em XML ou com anotações.

Se houver necessidade adicional de configuração, é possível informar nomes de colunas, tabelas, tipos, limites, relacionamentos, estratégias e diversos outros mecanismos para viabilizar o mapeamento usando anotações como *@Table*, *@Column*, *@CollectionTable*, *@JoinTable*, *@JoinColumn*, etc. No exemplo abaixo foram usadas anotações para mapear a classe e seus atributos a tabelas e colunas com nomes diferentes:

```
@Entity
@Table(name="BOOK")
public class Livro implements Serializable {

     @Id
     private Long id;

    @Column(name="TITLE")
     private String titulo;

    @Column(name="PAGES")
     private int paginas;

     …
}
```

1.3 Configuração do ambiente

E critico para o uso do JPA que o ambiente de desenvolvimento e implantação esteja configurado corretamente. Nas seções a seguir descreveremos os detalhes de algumas configurações através de exemplos completos que incluem operações de persistência com um objeto simples.

1.3.1 Configuração local (standalone)

A configuração *standalone* pode ser usada em projetos Java EE para rodar testes unitários sobre as instâncias sem a necessidade de realizar *deploy* no servidor. Pode também ser usada em ferramentas e aplicações que rodam fora do servidor em comunicação direta com o banco de dados.

É preciso incluir explicitamente no *Classpath* os JARs da API, do provedor JPA usado e do driver do banco de dados (e pool de conexões se houver). O exemplo abaixo ilustra a configuração das dependências *Maven* para baixar os JARs para um acesso JPA usando *EclipseLink 2.5* e banco de dados *PostgreSQL*:

```
<dependencies>
        <dependency>
                <groupId>org.eclipse.persistence</groupId>
                <artifactId>javax.persistence</artifactId>
                <version>2.1.0</version>
        </dependency>

        <dependency>
                <groupId>org.eclipse.persistence</groupId>
                <artifactId>eclipselink</artifactId>
                <version>2.5.0</version>
        </dependency>

        <dependency>
                <groupId>org.postgresql</groupId>
                <artifactId>postgresql</artifactId>
                <version>9.4.1208</version>
                <scope>runtime</scope>
        </dependency>
</dependencies>
```

Pode-se usar outros mecanismos (IDEs, Ant, etc.) para incluir essas dependências. O importante é que a primeiro (API) esteja disponível em tempo de desenvolvimento e execução, e todas estejam no *Classpath* em tempo de execução.

No modo standalone, o arquivo *persistence.xml* precisa declarar a configuração JDBC para que a camada de persistência possa se comunicar com o banco. Como as transações serão controladas pela aplicação e serão locais, a unidade de persistência declara o tipo de transações como *RESOURCE_LOCAL*:

```
<persistence version="2.1"…>
    <persistence-unit       name="tutorial-jpa"
                            transaction-type="RESOURCE_LOCAL">

      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

```
        <class>br.com.argonavis.javaee7.jpa.intro.Livro</class>
        <properties>
            …
           <property name="eclipselink.ddl-generation"
                 value="drop-and-create-tables" />
          <property name="eclipselink.deploy-on-startup" value="true" />
        </properties>

    </persistence-unit>
</persistence>
```

Dependendo do provedor usado, poderá ser necessário informar propriedades adicionais, dialetos, opções, etc. Neste exemplo incluímos uma propriedade do EclipseLink que irá causar a geração automática de esquemas a partir das instâncias (útil em tempo de desenvolvimento e ambiente de testes). Essa geração será genérica e baseada em vários defaults (para ter um controle maior é necessário incluir dados adicionais no mapeamento via anotações ou XML).

As classes que irão instanciar o *EntityManager* podem obtê-lo via consulta JNDI global a um servidor, se houver um container cliente configurado para tal, ouy instancia-lo diretamente. No exemplo abaixo, a configuração foi feita localmente, instanciando diretamente uma fábrica local para obter o *EntityManager* da unidade de persistência especificada.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class LivroTest {
    public static void main(String[] args) {

        EntityManagerFactory factory =
           Persistence.createEntityManagerFactory("tutorial-jpa");
        EntityManager em = factory.createEntityManager();

        ...
    }
}
```

Observe que o parâmetro passado ao método de fabrica do *EntityManagerFactory* corresponde ao nome da unidade de persistência exatamente como foi declarada no *persistence.xml*.

A entidade a ser sincronizada com a tabela é instanciada e configurada normalmente em Java puro.

```
public class LivroTest {
    public static void main(String[] args) {
        ...
       Livro livro = new Livro();
```

```
        livro.setTitulo("Meu Livro");
        livro.setPaginas(100);
        ...
    }
}
```

Como é um objeto novo será usado o método persist() para sincronizar o objeto com o banco. Dependendo de como foi configurada a persistência transitiva, a criação do objeto novo poderá gerar um ID numérico que será a chave primária do registro. Nesse tipo de configuração também poderia ser usado o método merge(), que distingue objetos novos (inserções) de antigos (atualizações) através do ID. Tanto merge() como persist() precisam ser chamados dentro de um contexto transacional, e como as transações são locais, é preciso obtê-las do EntityManager:

```
public class LivroTest {
    public static void main(String[] args) {
        ...
        try {
            em.getTransaction().begin();

            em.persist(livro); // irá criar um registro novo no commit()

            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
        } finally {
            em.close();
        }
    }
}
```

O método *getTransaction()* retorna uma *EntityTransaction*, que possui os métodos *begin(), commit()* e *rollback()*.

1.3.2 Configuração em ambiente Java EE

Em Java EE o suporte a JPA é *nativo*, e a configuração do banco de dados deve ser preferencialmente realizada através de um datasource *previamente configurado no servidor*, e que possa ser acessado dentro de um contexto JTA. Portanto, as dependências em um projeto Maven não devem ser incluídas no componente (JAR) que será implantado (devem ser marcadas como *provided*):

```
<dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
```

```
            <scope>provided</scope>
        </dependency>
...
</dependencies>
```

O arquivo *persistence.xml* declara estratégia transacional JTA (gerenciada pelo container) e em vez de informar dados para acesso ao driver do banco, informa-se o caminho JNDI para o datasource que corresponde ao banco usado.

```
<persistence version="2.1" …>
    <persistence-unit name="tutorial-jpa" transaction-type="JTA">
        <jta-data-source>jdbc/TutorialJPA</jta-data-source>
        <class>br.com.argonavis.javaee7.jpa.intro.ejb.Livro</class>
    </persistence-unit>
</persistence>
```

Dependendo do provedor usado, poderá ser necessário informar propriedades adicionais, dialetos, opções, etc. Neste exemplo *não* estamos usando a geração automática de esquemas, assumindo que as tabelas já existem e estão mapeadas corretamente.

Em Java EE o *EntityManager* poderá ser obtido via JNDI ou via injeção de dependências através de *@PersistenceContext* (ou ainda via CDI, se previamente configurado). Neste exemplo de acesso em servidor Java EE através de um *WebServlet*, identificamos a unidade de persistência usada e injetamos também um *UserTransaction*, necessário para delimitar o contexto transacional para cada operação de alteração (não e permitido usar o *EntityTransaction* – ou qualquer outro tipo de transação gerenciada pela aplicação – pois declaramos o uso de transações gerenciadas pelo container).

```
@WebServlet("/LivroTestServlet")
public class LivroTestServlet extends HttpServlet {

    @PersistenceContext(unitName="tutorial-jpa")
    EntityManager em;

    @Resource
    private UserTransaction ut; // Transação JTA

    protected void doGet(HttpServletRequest request,
     HttpServletResponse response) throws ServletException, IOException {

     Writer out = response.getWriter();
     try {
        ut.begin();

       List<Livro> livros =
          em.createQuery("select livro from Livro livro")
            .getResultList();
```

```
        for(Livro livro : livros) {
              out.write("<p>"+livro.getTitulo()+ "</p>");
        }

          ut.commit();

       } catch (…) {…} // rollback e finalização

}
```

Aqui construímos um query usando a linguagem JPQL para selecionar *todas* as entidades do tipo *Livro*. Assumindo que o exemplo anterior (*standalone*) foi executado, devemos ter um objeto persistente no banco. A query irá retornar a lista (contendo um elemento) que será exibida na página retornada pelo servlet, ao acessar *http://servidor:porta/aplicacao/LivroTestServlet* (onde *servidor*, *porta* e *aplicacao* são respectivamente o nome do servidor, número da porta, e o nome da *webapp* instalada)

Neste outro exemplo fizemos o acesso através de um EJB, um *Session Bean*, cujos métodos são *transacionais por default*. Neste caso não é necessário (nem permitido) escrever nenhum código de delimitação para transações, já que ela será iniciada antes do método começar, e será cometida após o término do método, provocando um *rollback* se houver exceção (as políticas transacionais podem posteriormente ser configuradas via anotações do EJB.) Se houver apenas uma unidade de persistência, o nome dela não precisa ser informado ao injetar o *@PersistenceContext*.

```
@Stateless
public class LivroDAOSessionBean {

     @PersistenceContext
     EntityManager em;

    public Livro findByID(Long id) {
        Query query = em.createNamedQuery("selectById");
        query.setParameter("id", id);
        return (Livro)query.getSingleResult();
    }

    public List<Livro> findAll() {
        return (List<Livro>)em.createNamedQuery("selectAll").getResultList();
    }

    public void delete(Livro livro) {
     em.remove(livro);
    }

    public void update(Livro livro) {
     em.merge(livro);
```

```
    }

    public Livro insert(Livro livro) {
     return em.merge(livro);
    }

}
```

No exemplo acima, o JPQL foi declarado na classe que define a entidade usando *@NamedQuery*. Essa forma permite que os queries sejam referenciados pelo nome:

```
@Entity
@NamedQueries({
   @NamedQuery(name="selectAll",  query="SELECT livro FROM Livro livro"),
   @NamedQuery(name="selectById", query="SELECT livro FROM Livro livro WHERE livro.id=:id")
})
public class Livro { ... }
```

Por fim, neste último exemplo a seguir, usamos um POJO comum com CDI. Diferentemente do EJB, o uso de CDI não garante métodos transacionais por default. Como o POJO roda dentro do servidor Java EE, podemos usar o *UserTransaction* (como no exemplo com *WebServlet*) para fornecer contexto transacional para as operações de persistência, ou usar a anotação CDI *@Transactional* antes de cada método. No servidor Java EE isto garante que o método será protegido por transação JTA e terá comportamento igual ao EJB.

```
@Named
public class LivroDAOManagedBean {

    @Inject
    EntityManager em;

    public Livro findByID(Long id) {
        TypedQuery<Livro> query =
              em.createNamedQuery("selectById", Livro.class);
        query.setParameter("id", id);
        return query.getSingleResult();
    }

    public List<Livro> findAll() {
        return em.createNamedQuery("selectAll", Livro.class)
                 .getResultList();
    }

    @Transactional
        public void delete(Livro livro) {
        em.remove(livro);
    }

    @Transactional
        public void update(Livro livro) {
```

```
        em.merge(livro);
    }

    @Transactional
        public Livro insert(Livro livro) {
        return em.merge(livro);
    }
}
```

No exemplo acima usamos um *TypedQuery* em vez de *Query* para construir a consulta. *TypedQuery* é recomendado pois evita o uso de *cast* ao declarar o tipo dos objetos retornados.

Experimente rodar todos esses exemplos. Se necessário adapte-os para as configurações do seu ambiente. E importante rodar exemplos simples para garantir que o JPA esteja bem configurado. A configuração é uma das principais fontes de problemas em sistemas JPA e é bem mais fácil solucioná-los antes de construir aplicações mais complexas.

Consulte também os JavaDocs e a API para conhecer outras classes e anotações não abordadas aqui. O objetivo deste material não é ser completo e abrangente mas proporcionar um roteiro prático de introdução ao JPA.

2 Persistência

2.1 Operações CRUD

Operações CRUD (Create, Retrieve, Update, Delete – Criar, Recuperar, Atualizar, Remover) são as principais tarefas realizadas pela camada de persistência. Podem envolver uma única entidade ou uma rede de entidades interligadas via relacionamentos. JPA oferece várias maneiras de realizar operações CRUD:

- Através dos métodos de persistência transitiva: *persist*, *merge*, *delete*, *find*
- Através do *Java Persistence Query Language* (*JPQL*)
- Através da API *Criteria*
- Através de SQL nativo

Neste curso trataremos apenas das três primeiras formas.

A recuperação de uma entidade, o "R" de "Retrieve" do acrônimo CRUD, pode ser feita conhecendo-se sua chave primária através de find:

```
Livro livro = em.find(Livro.class, 1234);
```

Normalmente a recuperação envolve uma pesquisa no estado da entidade, então pode-se usar JPQL através de um Query (ou TypedQuery):

```
TypedQuery query =
    em.createQuery("select m from Livro m where m.id=:id", Livro.class);
query.setParameter("id", 1234);
Livro livro = query.getSingleResult();
```

Criteria é um query que é construído dinamicamente, através da construção de relacionamentos entre objetos. É ideal para pesquisas que são construídas em tempo de execução:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Livro> criteria = builder.createQuery(Livro.class);
Root<Livro> queryRoot = criteria.from(Livro.class);
criteria.where(builder.equal(queryRoot.get(Livro_.id), 1234));
TypedQuery<Livro> q = em.createQuery(criteria);
Livro livro = q.getSingleResult();
```

Para usar a API desta forma, uma classe *Livro_class* foi gerada (veja: Criteria Metamodel API) automaticamente pelas ferramentas da IDE (ou plug-in Maven).

Operações que envolvem atualização, "Create", "Update" e "Delete", geralmente são executadas através dos métodos de persistência transitiva persist, merge e delete. Por exemplo, para alterar o título do livro dentro de um contexto transacional pode-se usar:

```
Livro livro = em.find(Livro.class, 1234);
livro.setTitulo("Novo título");
em.merge(livro);
```

A mesma alteração pode ser realizada via JPQL:

```
Query query = em.createQuery("UPDATE Livro m"
    + "SET m.titulo = :titulo WHERE m.id = :id");
query.setParameter("titulo", "Novo título");
query.setParameter("id", "1234");
query.executeUpdate();
```

Ou Criteria:

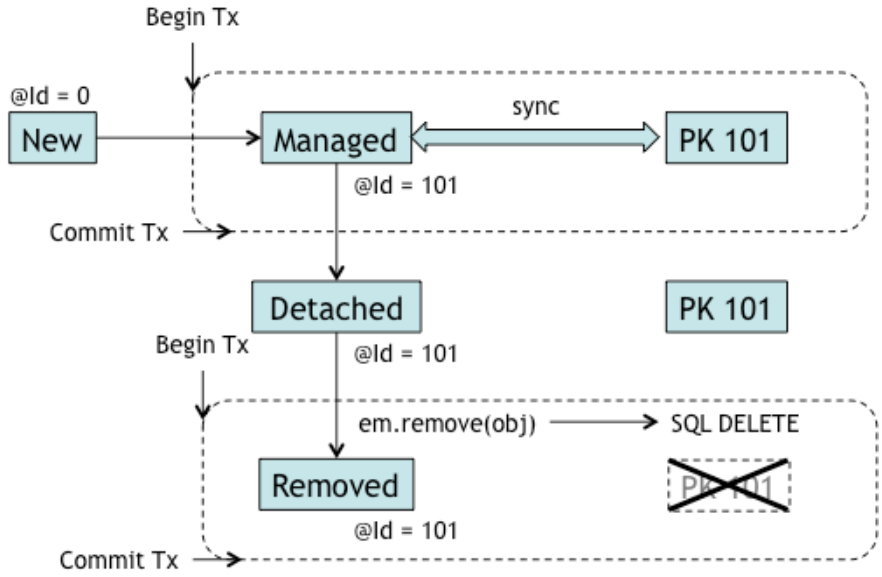
```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaUpdate updateCriteria = builder.createCriteriaUpdate(Livro.class);
Root<Livro> updateRoot = updateCriteria.from(Livro.class);
updateCriteria.where(builder.equal(updateRoot.get(Livro_.id), "1234"));
updateCriteria.set(updateRoot.get(Livro_.titulo), "Novo título");
Query q = em.createQuery(updateCriteria);
q.executeUpdate();
```

2.2 Ciclo de vida e operações de persistência

Uma entidade pode estar em quatro diferentes estados durante o seu ciclo de vida: transiente (*new*), persistente (*managed*), desligado (*detached*) ou removido (*removed*). O estado irá determinar o comportamento do objeto quando forem chamadas operações de persistência

transitiva sobre ele. Toda entidade tem uma chave primária indicada pela anotação *@Id* cujo valor geralmente é determinado pelo sistema de persistência. Quando uma entidade é nova e ainda não foi inserida no banco, o ID é nulo. Uma vez atribuído um ID a uma entidade, esse valor não muda mais.

O diagrama abaixo ilustra o ciclo de vida em entidades JPA:



Uma entidade é *transiente* quando não está associada a uma tabela no banco, e portanto não é persistente. Objetos mapeados são transientes logo depois que são criados e antes de serem persistidos através de uma operação *persist/merge* ou um *cascade* (*merge* disparado por um relacionamento). O identificador de um objeto transiente contém o valor null ou zero. Esse valor é usado pelo mecanismo de persistência para identificar objetos transientes.

A operação *new* cria uma instância no estado *transiente*:

```
Entidade novo = new Entidade();
```

Isto é Java puro. Ainda não usamos JPA. Para tornar a instância *persistente* é preciso utilizar a API do EntityManager dentro de um contexto transacional. Esta API possui vários métodos que controlam a persistência:

- *void* ***persist****(objeto)* – Gera um SQL INSERT e transforma o objeto passado como parâmetro em uma entidade. Causa *EntityExistsException* se o ID já existir no banco. Se o ID não existir no banco ou se for *null* a entidade será criada.
- *Object* ***merge****(objeto)* – Gera um SQL UPDATE ou um SQL INSERT e devolve a entidade construída. O INSERT será gerado se o ID do objeto for null. Caso contrário, o ID será usado como chave primária para realizar um UPDATE. Causa *IllegalArgumentException* se o ID não for *null* e não existir no banco.

- *void* ***remove****(objeto)* – Remove o registro do banco, desligando a entidade permanentemente. Ela não mais poderá ser ligada ao banco (um SQL UPDATE não e mais possível)
- *void* ***detach****(objeto)* – Separa a instancia do seu registro tornando a entidade desligada (detached). O mesmo ocorre com as entidades que são usadas fora do contexto transacional do EntityManager. O método clear() causa um detach em todas as entidades. Um merge() ou refresh() dentro do contexto transacional religa a entidade tornando-a persistente.
- *void* ***refresh****(objeto)* – Sincroniza a entidade com dados obtidos do banco, sobrepondo quaisquer alterações que tenham sido feitas no estado do objeto.
- *void* ***flush****()* – Sincroniza as entidades do contexto de persistência com o banco (oposto de refresh).

Pode-se criar uma nova entidade passando um objeto novo para o método persist(), que cria um registro novo através de um SQL INSERT e o sincroniza com a instância transiente, tornando-a persistente:

```
entityManager.persist(novo);
```

Tambem e possível criar uma entidade através do método merge(), *se* o objeto tiver um ID *null*. Se o ID não for *null*, merge() considera que o objeto não é novo (é transiente) e irá usar a chave primaria para gerar um SQL UPDATE. Diferentemente do *persist()*, o método *merge()* não altera a instância passada como argumento, mas devolve uma cópia persistente dela.

```
Entidade persistente = entityManager.merge(novo);
```

Uma entidade é considerada *desligada* (*detached*) quando já possui uma associação com uma tabela no banco (tem um *@Id* diferente de zero ou *null*), mas não está, no momento, sincronizada com ela por não se encontrar dentro de um contexto transacional, ou porque a transação ainda não foi cometida, ou porque ela foi desligada explicitamente através do método *detach()* ou *clear()*. A entidade poderá ser religada se entrar em outro contexto transacional com outro merge *dentro de um contexto transacional*:

```
Entidade persistente = entityManager.merge(detached);
```

Uma entidade desligada é útil, pois contém estado que pode ser exibido em uma página Web e alterado. Se não houver nenhuma alteração no registro correspondente durante o tempo que o objeto estiver desligado, o objeto poderá posteriormente sincronizado com o banco sem problemas, mas neste período ele também corre risco de se tornar obsoleto e conter dados inconsistentes com o banco, caso outro processo o tenha alterado. Para lidar com essas o JPA oferece travas (otimistas e pessimistas) que podem ser configuradas no mapeamento.

Um objeto pode ser *removido* do banco usando o método remove():

```
Entidade removida = entityManager.remove(persistente);
```

A remoção elimina o registro do banco, no entanto o objeto ainda pode ser usado. Ele não é considerado transiente porque possui um @Id, mas não é mais considerado desligado ou persistente porque não há mais uma tabela associada a ele. Não é possível mais sincronizá-lo com o banco. Um *merge()* causaria uma exceção. Mas é possível enviá-lo para que seus dados sejam lidos por uma interface do usuário, por exemplo, e seus dados poderão retornar ao banco se forem copiados para uma nova instancia, o que fará com que seus dados sejam reinseridos no banco em um novo registro, tornando-se assim novamente persistente (mas com outro @Id). Isto pode ser feito com um *persist()*.

2.3 Listeners

Para lidar com as mudanças de estado de uma entidade, existe em JPA um conjunto de anotações para definir *callbacks* que respondem a eventos de mudanças:

- *@PostLoad* - executado depois que uma entidade foi carregada no contexto de persistência atual (ou *refresh*)
- *@PrePersist* - executado antes da operação *persist* ser executada (ou *cascade* - sincronizado com *persist* se entidade for sincronizada via *merge*)
- *@PostPersist* - executado depois que entidade foi persistida (inclusive via *cascade*)
- *@PreUpdate* - antes de operação UPDATE
- *@PostUpdate* - depois de operação UPDATE
- *@PreRemove* - antes de uma operação de remoção

Os métodos que implementam os listeners podem ter quaisquer nomes. Não devem ser *static* nem *final*. Podem ser declarados na própria instância (nesse caso não devem receber parâmetros) ou em uma classe separada (devem receber um objeto como parâmetro, que corresponde à instância monitorada.) Para declarar os listeners em uma classe separada, pode-se usar a anotação *@EntityListener* na entidade informando o nome da classe que os contém:

```
@Entity
@EntityListeners(LivroListener.class)
public class Livro implements Serializable {
   …
}
```

Neste caso, os métodos devem receber a entidade como parâmetro:

```
public class LivroListener {
   @PostLoad
```

```
    public void livroLoaded(Livro livro) { ... }
    ...
}
```

Além desses métodos, entidades também podem usar os *callbacks* do CDI, *@PostConstruct* e *@PreDestroy*, que são habilitados por default em ambientes Java EE. Listeners também podem ser declarados em *orm.xml*. Essa configuração e ideal se houver mais de um listener para as entidades.

Rode os exemplos que ilustram o funcionamento de listeners usando apenas um objeto. Na seção a seguir apresentaremos outros exemplos com relacionamentos e *cascade*.

3 Mapeamento

3.1 Mapeamento de relacionamentos entre entidades

Associações no JPA funcionam da mesma maneira que associações de objetos em Java. São naturalmente unidirecionais e *não são* gerenciadas pelo container, o que significa que e preciso escrever e chamar explicitamente o código Java necessário para atualizar os campos correspondentes de cada uma instancias que participam do relacionamento, quando dados são adicionados, removidos ou alterados (o container, porém pode gerenciar a propagação da persistência de forma transitiva através de configuração de operações de cascata).

Os relacionamentos são sempre associações entre *entidades* (diretamente, ou indiretamente através de coleções). Pode-se também configurar associações entre objetos que não são relacionamentos. Neste caso, para que o estado desses objetos seja persistente, é preciso que estejam dependentes de uma entidade.

Para mapear relacionamentos, JPA utiliza informações da estrutura do código Java e das anotações. As anotações podem ser mínimas. Serão usados parâmetros *default* até onde for possível. Assim como *@Column* pode ser usado para informar um nome de coluna diferente do nome da propriedade escalar, *@JoinColumn* pode ser usado quando a coluna que contém a chave estrangeira da associação tiver um nome ou configuração diferente da propriedade mapeada.

A *cardinalidade* define o número de entidades que existe em cada lado do relacionamento. Podem ser mapeados dois valores: "*muitos*" e "*um*", resultando em *quatro* possibilidades:

- Um para muitos
- Muitos para um
- Um para um

- Muitos para muitos.

As duas primeiras são equivalentes, portanto há três diferentes associações:

- *@ManyToOne / @OneToMany*
- *@OneToOne*
- *@ManyToMany*

Dentro de cada contexto, é importante também levar em conta a *direção* dos relacionamentos, que pode ser *unidirecional* ou *bidirecional*.

@ManyToOne é a associação mais comum. O modelo relacional é naturalmente um-para-muitos e uma referência simples de objeto é um-para-muitos. Em associações unidirecionais, apenas um lado possui referência para o outro (que não tem conhecimento da ligação). A anotação é usada apenas em um dos lados. Em associações bidirecionais, do outro lado deve possuir uma anotação *@OneToMany* e é preciso informar o nome da referência que determina a associação.

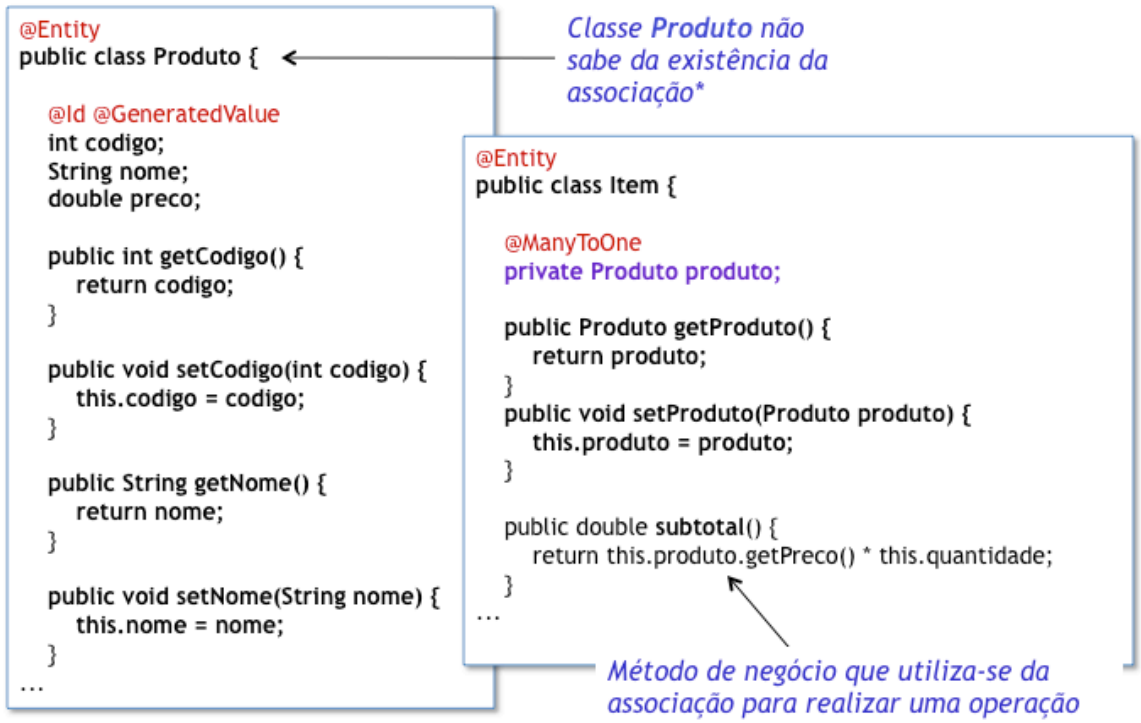
@OneToOne requer constraints (que são gerados) nas tabelas para garantir a consistência da associação. É também possível estabelecer uma relação um-para-um entre entidades usando *@ManyToOne*, desde que seja unidirecional o lado "many" seja limitado a um único elemento.

@ManyToMany mapeia três tabelas a dois objetos. Também é possível estabelecer uma relação muitos-para-muitos entre entidades usando *@ManyToOne*, desde que haja três objetos, dois contendo associações *@ManyToOne* para um terceiro. Isto pode ser usado se for necessário representar a associação como uma entidade, mapeada a uma tabela via *@JoinTable*.

Várias anotações e atributos permitem configurar ajustes finos dos mapeamentos, como regras de cascade, estratégias de carga, caches, transações, direção, etc.

3.1.1 Relacionamentos @ManyToOne

O exemplo a seguir ilustra um relacionamento *unidirecional* usando *ManyToOne*.



* *Esses dados são insuficientes para a geração automática do esquema (as tabelas devem existir antes)*

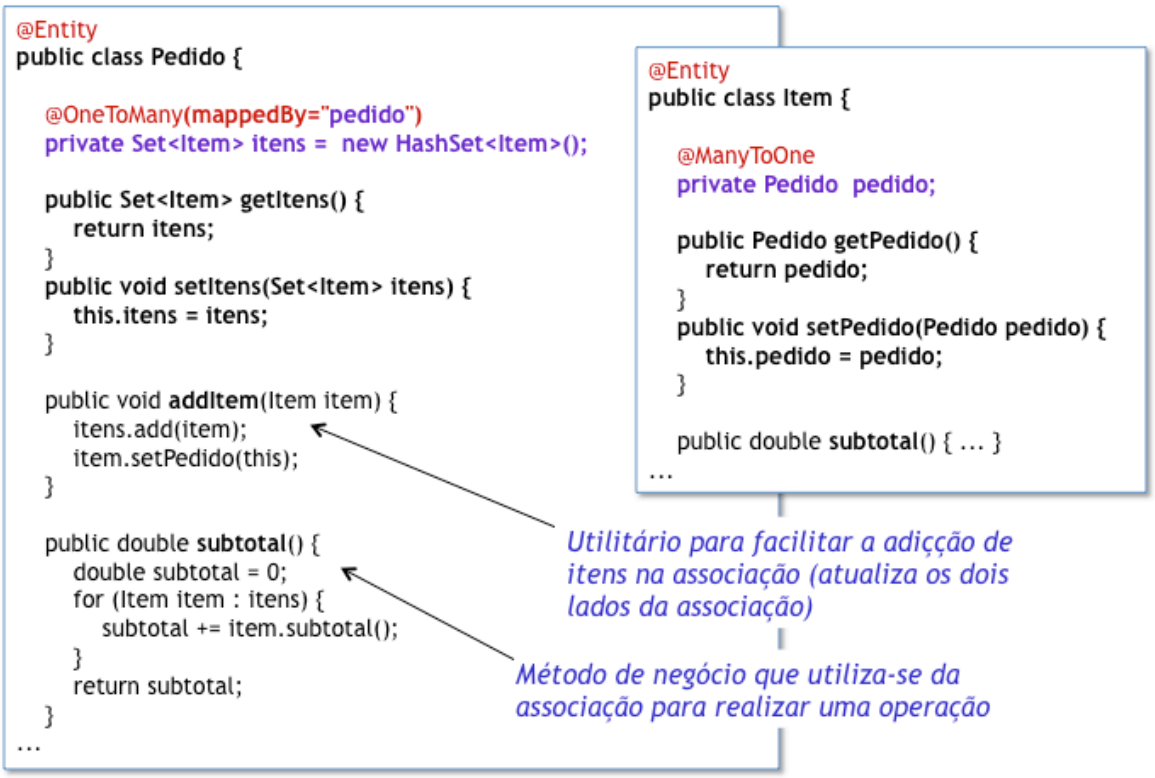
Se as anotações *@OneToOne*, *@ManyToMany* e *@ManyToOne* forem usadas em associações *bidirecionais* é necessário especificar anotações de ambos os lados e utilizar o atributo *mappedBy* (em um dos lados apenas) para informar o nome do atributo da outra classe que faz a associação.

O diagrama abaixo ilustra algumas associações bidirecionais.

`@OneToOne` `CPF cpf;`	→	`@OneToOne(mappedBy="cpf")` `Cliente cliente;`
`@ManyToMany` `Set<Turma> turmas;`	→	`@ManyToMany(mappedBy="turmas")` `Set<Aluno> alunos;`
`@ManyToOne` `Item item;`	→	`@OneToMany(mappedBy="item")` `Set<Produto> produtos;`

Caso sejam usadas ferramentas de geração automática de esquemas e tabelas, poderá ser necessário incluir anotações bidirecionais, pois as anotações unidirecionais poderão ser insuficientes para gerar os esquemas corretamente.

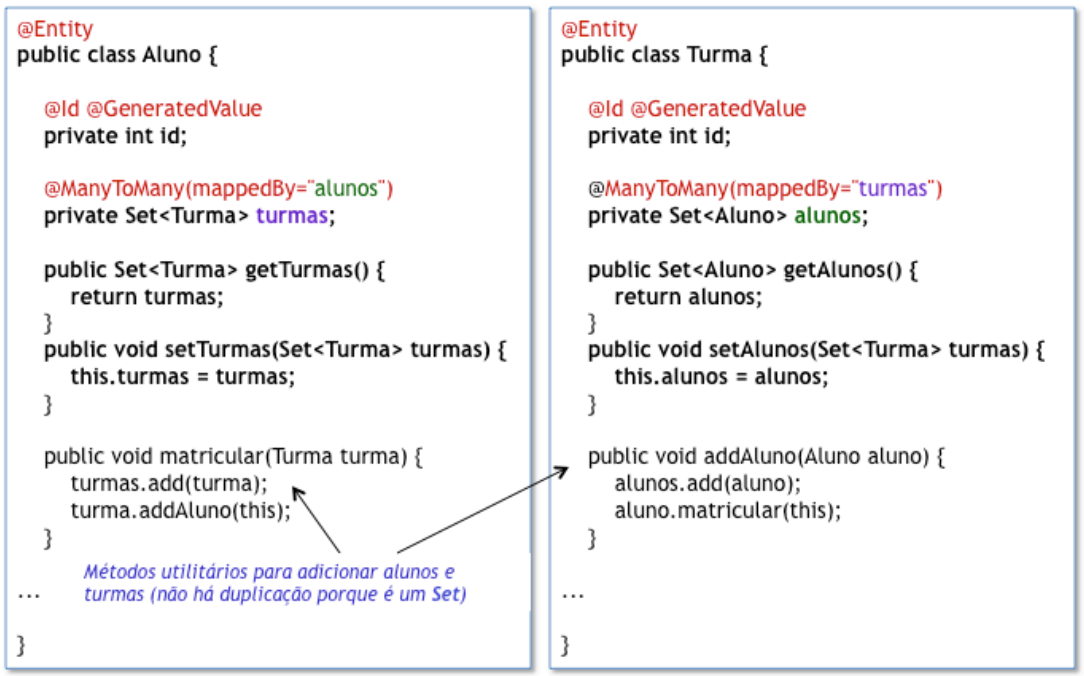
Exemplo de um relacionamento *ManyToOne bidirecional*:



3.1.2 Relacionamentos @ManyToMany

Use *@ManyToMany* nas coleções de cada lado da associação, informando o campo de mapeamento (*mappedBy*) em um dos lados. Usando-se um *Set* e *mappedBy*, garante-se a não duplicação de dados quando objeto é adicionado duas vezes na coleção.

Abaixo um exemplo simples ilustrando o mapeamento *Many to Many*.



3.1.3 Relacionamentos @OneToOne

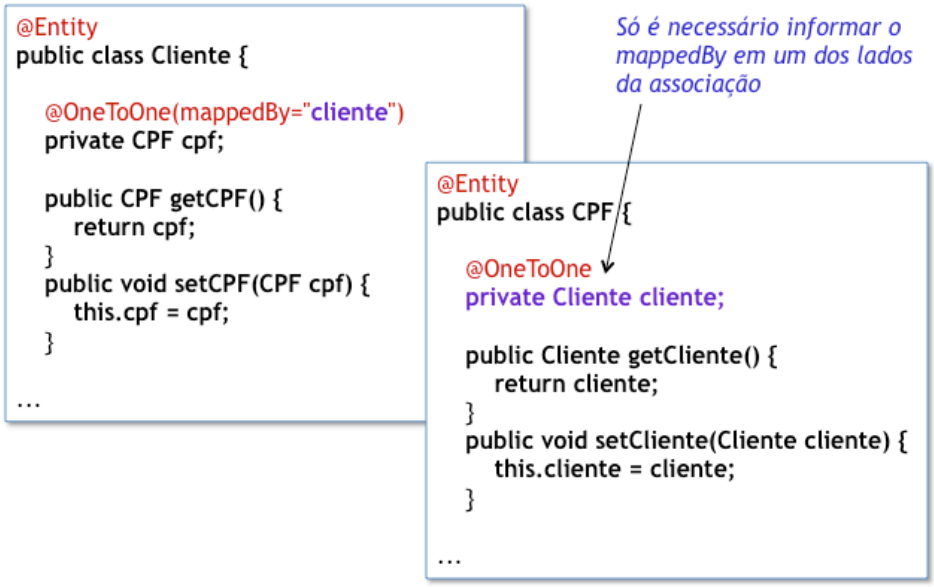
Um relacionamento *um para um* considera que os objetos em um relacionamento têm alto grau de dependência. No modelo relacional isto pode ser implementado de três maneiras:

1. Dois objetos, uma única tabela (dados em colunas da mesma tabela)
2. Um objeto, duas tabelas (dados em colunas de outra tabela)
3. Dois objetos, duas tabelas unidas por associação

Usando JPA, a terceira estratégia pode ser implementada usando relacionamentos *@ManyToOne* (unidirecional) ou *@OneToOne* (com mapeamento via chave estrangeira ou chave primária).

As duas primeiras estratégias não são relacionamentos entre entidades, mas associações com instâncias ou coleções de objetos. Em JPA, a primeira estratégia é implementada usando componentes *@Embeddable* e a segunda através de *@SecondaryTable*.

Para mapear associações um-para-um, inclua *@OneToOne* em um dos lados, se unidirecional. Se for bidirecional, o campo correspondente do outro lado é informado através de *mappedBy*:



3.2 Coleções

3.2.1 Lists, Sets, Collections

Relacionamentos *@ManyToOne* e *@ManyToMany* envolvem coleções. O mapeamento natural para registros em uma tabela é (*java.util.Set*), porque *Sets* não possuem uma ordem especifica e não podem ser duplicados, mas isto não é uma regra. Existem situações onde pode

haver pequenos ganhos de performance usando *Sets* em certos tipos de relacionamento e *Lists* em outros, mas em geral a escolha é motivada pela finalidade dessas estruturas. Basicamente:

- Use *java.util.Set* na maior parte dos casos, para listas de itens não ordenados e que não contém duplicatas (na verdade Sets *podem* ser ordenados no query usando @OrderBy e na instância usando uma implementação ordenada como TreeSet)
- Use *java.util.List* para listas de itens que podem ser duplicadas e que tem uma ordem indexada (onde o índice é importante); Uma alternativa é usar @MapKey com uma lista ordenada para as chaves.
- Use *java.util.Collection* quando a coleção representar uma coleção qualquer, na qual não faz diferença se há duplicações, ou se é ordenada.

Um outro motivo para escolher *List* em vez de *Set* ou vice-versa (quando não faz diferença para a aplicação a escolha de uma ou outra) é devido a compatibilidade com outros componentes (ex: componentes JSF, *Primefaces*, etc) que esperam *List*, ou *Set*, ou *Collection*. Isto evita a necessidade de criar um adaptador para fazer a conversão.

3.2.2 Mapas e @MapKey

E possível mapear uma coleção a um mapa usando *@MapKey* e representando a coleção como um *java.util.Map*. Um *Map* contem um *Set* de chaves e uma *Collection* de valores, e é indicado situações nas quais é mais eficiente recuperar um objeto através de uma chave.

A chave default é a chave-primária dos objetos da coleção:

```
@Entity
    public class LojaVirtual {
        ...
        @OneToMany(mappedBy="lojaVirtual")
        @MapKey  // indexado pela chave primaria
        public Map<Long, Cliente> clientes = new HashMap <>();
        ...
    }
```

Usando o atributo *name* é possível indexar por outras propriedades dos objetos da coleção:

```
    @Entity
    public class LojaVirtual {
        ...
        @OneToMany(mappedBy="lojaVirtual")
        @MapKey(name="cpf")  // indexado pelo CPF
        public Map<String, Cliente> clientes = new HashMap<>();
        ...
    }
```

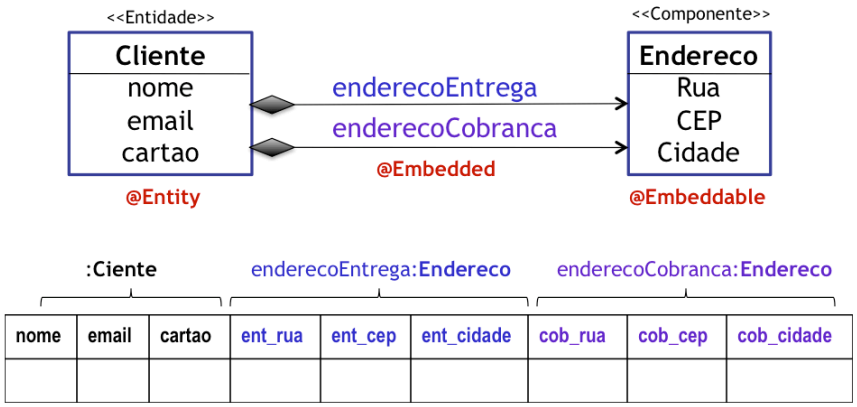
O mapeamento da chave também pode ser feito através das colunas, em vez dos atributos, usando *@MapKeyColumn* e *@MapKeyJoinColumn*. Existem ainda mais duas anotações para MapKey: @MapKeyEnumerated, se a chave for uma enumeração, e *@MapKeyTemporal*, se a chave for uma data ou Timestamp.

3.3 Mapeamento de objetos embutidos (@Embeddable)

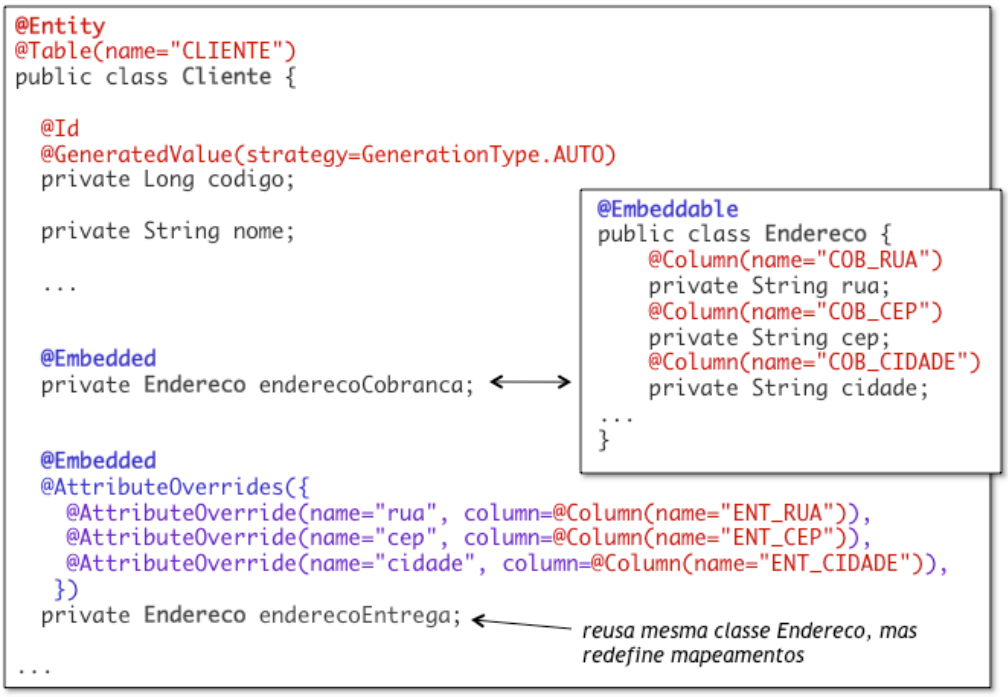
Entidades são objetos persistentes que têm identidade no banco. Objetos que contém dados persistentes mas não têm identidade unívoca no banco são identificados pelo seu estado, ou valor (*value objects*). Em JPA são chamados de *embutíveis* e anotados como *@Embeddable*. Como atributo de uma entidade são anotados com *@Embedded* e são dependentes dela.

No *modelo de domínio* um objeto embutido representa um objeto, mas no *modelo relacional* representa apenas algumas colunas de uma tabela. O objeto embutido é *propriedade* da entidade. Muitas associações 1-1 podem ser implementadas de maneira mais eficiente como objetos embutidos.

Na ilustração abaixo *Cliente* é um objeto e e uma entidade, e *Endereço* é um objeto mas não é uma entidade. Um *Cliente* está associado a dois objetos *Endereco*, mas como não são entidades separadas, ocupam a mesma tabela:



A remoção do registro no banco remove a entidade e seus objetos dependentes. O exemplo abaixo ilustra como utilizar *@Embeddable* e *@Embedded* em JPA.



O uso de *@AttributeOverrides* para redefinir os nomes das colunas e inevitável se houver mais de um *objeto embutido* do mesmo tipo na entidade, já que os nomes das colunas, por default, são gerados a partir dos atributos da classe, e não já uma estratégia default para gerar nomes diferentes automaticamente.

3.3.1 Coleções

Se uma entidade contém atributo com uma coleção de objetos embutidos, ou tipos básicos (que não sejam entidades) ele deve ser anotado com *@ElementCollection*. Por exemplo, a entidade abaixo contém uma coleção de objetos *Endereco*, que são *@Embeddable*, e uma coleção de Strings:

```
@Entity
public class Representantes {
    ...
    @ElementCollection
    private Set<Endereco> enderecos = new HashSet<>();

    @ElementCollection
    private Set<String> emails = new HashSet<>();
    ...
}
```

3.3.2 Chaves compostas

@Embeddable pode ser usado na criação de chaves compostas. Por exemplo, uma chave-primária que consiste de dois campos do tipo String:

```
@Embeddable
public class DependenciaId {
    @Column(name = "appID")
    private String appID;

    @Column(name = "groupID")
    private String groupID;
    //...
}
```

Ela pode ser usada em uma entidade anotada com *@EmbeddedId*:

```
@Entity
 @Table(name = "Dependencia")
 public class Dependencia {

     @EmbeddedId
     private DependenciaId id;
  // ...
 }
```

É responsabilidade da aplicação gerenciar o estado da chave composta, para que a persistência transitiva ocorra corretamente.

3.4 Persistência transitiva, cascading, lazy loading

Os relacionamentos em JPA são implementados em Java puro, e não são gerenciados pelo container. Por exemplo, se um *Item* possui um relacionamento com um *Pedido* da forma "um Pedido contém muitos Itens", para acrescentar um item em um pedido, seria preciso sincronizar dois objetos: *Pedido* e *Item*:

```
Pedido p = new Pedido();
Item i = new Item();
p.addItem(i);       // atualizando o pedido
item.setPedido(p); // atualizando o item (se bidiretional)
```

Além disso, é necessário que essas entidades sejam persistentes. A tentativa de adicionar uma entidade transiente ou nova a uma entidade persistente irá causar uma *TransientObjectException*. É necessário que a entidade transiente ou nova seja antes sincronizada com o banco. Isto pode ser feito explicitamente, dentro de um contexto transacional, através de *persist()/merge()*:

```
em.persist(p);
em.persist(i);
```

Mas isto também pode ser realizado automaticamente através da configuração de persistência transitiva, que pode ser configurada automaticamente no JPA usando o atributo

cascade das anotações de relacionamentos. Para propagar a inserção de entidades pode-se usar *CascadeType.PERSIST*. Por exemplo, na classe *Pedido* pode-se mapear os itens da forma:

```
@OneToMany(cascade={CascadeType.PERSIST}, mappedBy="item" )
private Set<Item> itens = new HashSet<Item>();
```

Isto fara que quando um *Pedido* for persistido no banco, quaisquer itens que estiverem na sua hierarquia de objetos também serão persistidos automaticamente.

3.4.1 Configuração de CascadeType

O atributo cascade recebe um array {} de opções que podem ser combinadas. Há várias opções para *CascadeType*:

1. PERSIST – propaga operações persist() e merge() em objetos novos
2. MERGE – propaga operações merge()
3. REMOVE – propaga operações remove()
4. DETACH – propaga operações detach()
5. REFRESH – propaga operações refresh()
6. ALL – propaga todas as operações. Declarar cascade=ALL é o mesmo que declarar cascade={PERSIST, MERGE, REMOVE, REFRESH, DETACH}

A escolha vai depender do design da aplicação. Um uso típico seria PERSIST com MERGE, para garantir que inserts e updates possam ser feitos automaticamente.

```
@OneToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE})
```

Talvez a remoção de um *Pedido* exija também a remoção de seus *Itens*. Neste caso um CascadeType.REMOVE seria necessário. Pode-se usar CascadeType.ALL para que todas as operações sejam transitivas (mas poderá haver impactos de performance no uso desnecessário de *CascadeType.ALL*).

3.4.2 Lazy loading

Lazy loading um padrão de design usado para adiar a inicialização de um objeto para quando ele for realmente necessário. É uma técnica que contribui para a eficiência da aplicação se usado corretamente.

ORMs usam estratégias de lazy loading para otimizar as pesquisas. A navegação em uma coleção de referências realiza queries dentro de um contexto transacional. Fora desse contexto, apenas as *referências* que foram inicializadas quando estavam no contexto transacional são accessíveis, não as instâncias contendo dados.

Se você usar um cliente externo que não mantém sempre aberta a sessão do EntityManager, poderá ter exceções de inicialização lazy quando acessar objetos dentro de coleções. O query, por default, só retornará a coleção com os ponteiros para os objetos, e eles só podem ser recuperados do banco se houver uma sessão aberta.

A forma como o provedor reage diante dessa situação depende de implementação. Ao tentar acessar uma referência não inicializada, o Hibernate provoca uma *LazyInitializationException*. Ja o EclipseLink, quando acontece a tentativa de acesso a uma coleção de um objeto desligado, mesmo com o *EntityManager* fechado, ele abrirá uma nova conexão no banco e preencherá a coleção com dados obtidos do query, evitando o *LazyInitializationException*. Uma exceção Lazy ainda pode ocorrer se o objeto estiver serializado (pode ocorrer durante passivação em EJB, por exemplo). Neste caso e preciso lidar com o problema buscando alternativas para carregar os dados da coleção previamente.

É possível configurar esse comportamento configurando a recuperação como *eager* (ansiosa), em vez de *lazy* (preguiçosa). Isto pode ser feito de duas maneiras: através de mapeamento (default para todas as situações) e instruções no query (configurado a cada consulta).

O modo de recuperação *eager* pode ser configurado como default através de mapeamento, declarando nas associações o atributo *fetch=FetchType* da forma

```
@OneToMany(mappedBy="pedido", fetch=FetchType.EAGER)
private Set<Item> itens = new HashSet<Item>();
```

Configurar todas as operações para usar eager loading é uma má idéia pois isso forçará o sistema a recuperar toda a árvore de objetos quando baixar uma coleção. O ideal é analisar a aplicação e descobrir o que realmente precisa ser usado, e buscar apenas esses dados. Por exemplo, se há 1000 objetos em uma coleção talvez só seja necessário buscar 10, e depois mais 10 em outra transação, em vez dos 1000 de uma vez. Portanto, *eager loading* idealmente deve ser configurado apenas no query. Mapeamentos devem configurar relacionamentos como lazy por default.

Em JPQL a palavra-chave *fetch* e usada para sinalizar eager loading (em queries Criteria existe o método fetch() da interface *From*).

Para atualizar objetos envolvidos em relacionamentos através de persistência transitiva, declare-os com *CascadeType.MERGE*. Desta forma, quando um objeto for atualizado, o restante da árvore de relacionamentos será atualizada em cascata.

3.5 Mapeamento de Herança

JPA não interfere no modelo de domínio da aplicação. Herança em Java, no domínio do da linguagem, continua funcionando como se espera. Entidades também suportam herança de classes, associações e queries polimórficos. Podem ser abstratas ou concretas, e podem ser subclasses ou superclasses de classes que não são entidades.

Entidades abstratas, assim como qualquer classe abstrata, não podem ser instanciadas. Mas pode-se fazer queries em classes abstratas. Em um query envolvendo classes abstratas, todas as subclasses concretas da entidade abstrata serão envolvidas.

Herança é o descasamento mais visível entre os mundos relacional e orientado a objetos. O mundo OO possui relacionamento "é um" e "tem um", enquanto que o mundo relacional apenas possui relacionamento "tem um". Há três estratégias para lidar com esse problema (sugeridas por Scott Ambler, 2002). Essas estratégias são adotadas na arquitetura do JPA.

3.5.1 MappedSuperclass

Entidades podem ter superclasses que não são entidades. Se a superclasse possuir a anotação *@MappedSuperclass*, seus atributos serão herdados e farão parte do estado persistente das subclasses. Uma *@MappedSuperclass* também pode incluir anotações de persistência nos seus atributos, mas não é uma entidade e não pode ser usada com operações de *EntityManager* ou Query. Ela não está mapeada a uma tabela. Apenas as subclasses de entidades concretas são mapeadas a tabelas e podem conter colunas correspondentes aos atributos herdados.

No exemplo abaixo a classe *Quantia* foi anotada como *@MappedSuperclass*, portanto seu atributo valor e o ID serão mapeados a colunas das tabelas mapeadas às entidades *Pagamento* e *Divida*:

```
@MappedSuperclass
public class Quantia {
  @Id
  protected Long id;
  protected BigDecimal valor; ...
}

@Entity
public class Pagamento extends Quantia { ... }

@Entity
public class Divida extends Quantia { ... }
```

É possível também herdar de classes comuns que não têm nenhuma participação no JPA. Essas classes poderão ter métodos e atributos que serão herdados mas cujo estado *não será persistido no banco* (não haverá colunas correspondentes aos atributos herdados).

A anotação j*avax.persistence.Inheritance* é usada para configurar o mapeamento de herança entre entidades em JPA. Elas implementam as três estratégias citadas que são selecionadas através das constantes da enumeração *InheritanceType*:

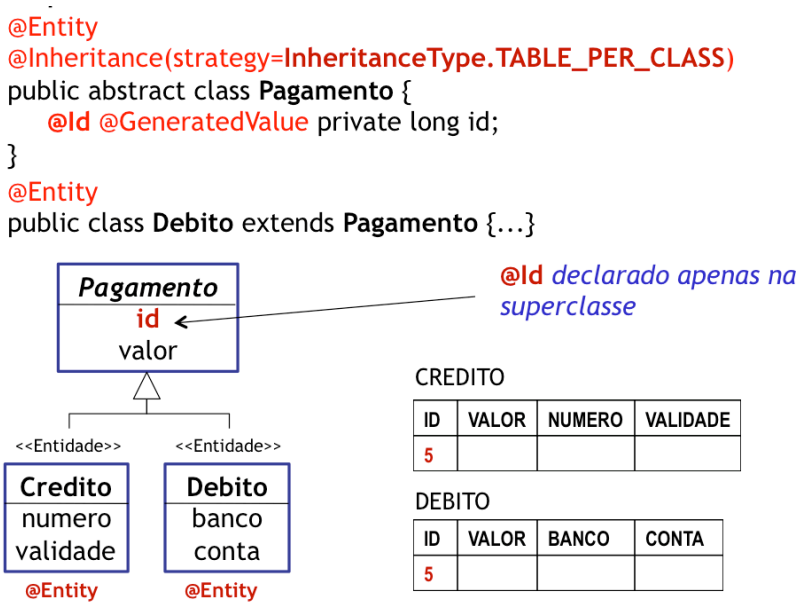
```
public enum InheritanceType {
    SINGLE_TABLE,
    JOINED,
    TABLE_PER_CLASS
};
```

A estratégia default é SINGLE_TABLE, que mapeia todas as classes de uma hierarquia a uma única tabela no banco de dados.

3.5.2 Tabela por classe concreta

A estratégia chamada *"Uma tabela por classe concreta"* é, de acordo com a especificação, opcional, mas é suportada pelos principais provedores JPA usados hoje no mercado. Nela, o modelo relacional ignora herança e polimorfismo (o polimorfismo ocorre implicitamente), mapeando apenas as classes concretas a tabelas. A interface da classe abstrata ainda será usada nos queries, mas é ignorada na implementação que considera apenas a interface que foi herdada em cada classe concreta.

A ilustração abaixo mostra como essa estratégia poderia ser implementada com JPA.



Ou seja, todas as classes são declaradas como *@Entity*, mas apenas as subclasses são mapeadas a tabelas (não se pode usar *@Table* na classe abstrata). Como o ID é o mesmo para as subclasses, ele não é declarado nas entidades concretas e herda todas as suas definições de mapeamento da superclasse. Usando um gerador sequencial de IDs, o sistema irá usar uma contagem única para todas as subclasses.

Essa estratégia tem como vantagem o mapeamento direto entre classe e tabela normalizada. O gravação é realizada sempre através de uma classe concreta, mas a pesquisa pode ser polimórfica, realizada via classe abstrata e com o resultado que inclui todas as subclasses concretas.

Por exemplo, é possível filtrar os Pagamentos por valor, sem precisar saber se são de Crédito ou Débito:

```
select p from Pagamento p where p.valor > 1000
```

A desvantagem dessa estratégia é a complexidade e ineficiência dos queries gerados. Para realizar a pesquisa, será necessário envolver todas as tabelas mapeadas às subclasses das entidades.

Por exemplo, considere a estrutura o *query polimórfico* (realizado na entidade da superclasse abstrata) mostrados acima. Esse query irá gerar um SQL composto de subqueries concatenados com union para cada subclasse.

A listagem abaixo ilustra um *possível* resultado da geração do SQL para o exemplo acima:

```
select ID, VALOR, NUMERO, VALIDADE, BANCO, CONTA from (
    select ID, VALOR, NUMERO, VALIDADE,
              null as BANCO, null as CONTA from CREDITO
    union
    select ID, VALOR, null as NUMERO, null as VALIDADE,
              BANCO, CONTA from DEBITO
) where VALOR > 1000
```

Este é um SQL *conceitual*. É possível e bastante provável que o provedor de persistência utilizado otimize o SQL gerado e produza outro resultado mais eficiente, mas isso depende da implementação usada. O importante aqui é considerar que o query em uma única entidade (*Pagamento*) filtrado por um atributo declarado nessa entidade (*valor*) irá exigir queries em pelo menos duas tabelas (as subclasses concretas que *herdam* o atributo) que terão quer ser combinados. Portanto, essa solução é ineficiente se a hierarquia for grande e se a aplicação realizar muitos queries polimórficos.

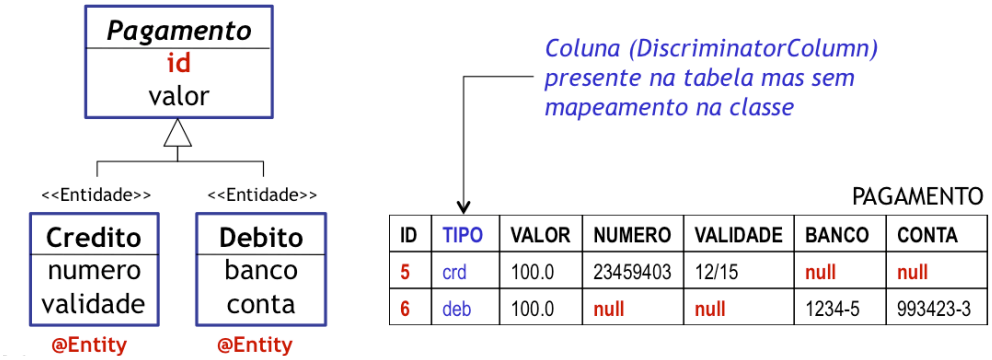
3.5.3 Tabela por hierarquia

A estratégia chamada de "**Uma tabela por hierarquia de classes**" é default em JPA. Ou seja, se nenhuma *@Entity* que participa de uma hierarquia for anotada com *@Inheritance*, esta será a estratégia usada: a hierarquia inteira será mapeada a uma única tabela. Isto permite consultas polimórficas eficientes, já que há uma tabela única, mas as tabelas não serão normalizadas e poderão registros com muitos campos vazios.

Como apenas uma tabela é usada, é necessário que exista uma coluna identificando o tipo. Isto é feito através da anotação *@DiscriminatorColumn*. O nome default é *DTYPE* (essa coluna terá que existir, ter este nome e tipo String caso um *@DiscriminatorColumn* não seja definido explicitamente).

O exemplo abaixo ilustra essa estratégia usando o mesmo modelo de domínio apresentado anteriormente:

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TIPO")
public abstract class Pagamento {
    @Id @GeneratedValue private long id;
}
@Entity
@DiscriminatorValue("crd")
public class Debito extends Pagamento {...}
```



Portanto, é necessário que a tabela tenha colunas para todas as propriedades de todas as classes.

A vantagem é que teremos a forma mais eficiente de fazer uma pesquisa polimórfica, já que tudo acontece em uma única tabela. Este é o SQL conceitual para um query polimórfico usando esta estratégia:

```
select ID, VALOR, BANCO, CONTA, NUMERO, VALIDADE
   from PAGAMENTO
    where VALOR > 1000
```

Se for feito um query em classe concreta, o provedor de persistência poderia gerar o seguinte código SQL:

```
select ID, VALOR, BANCO, CONTA, NUMERO, VALIDADE
   from PAGAMENTO
     where TIPO = 'deb' and VALOR > 1000
```

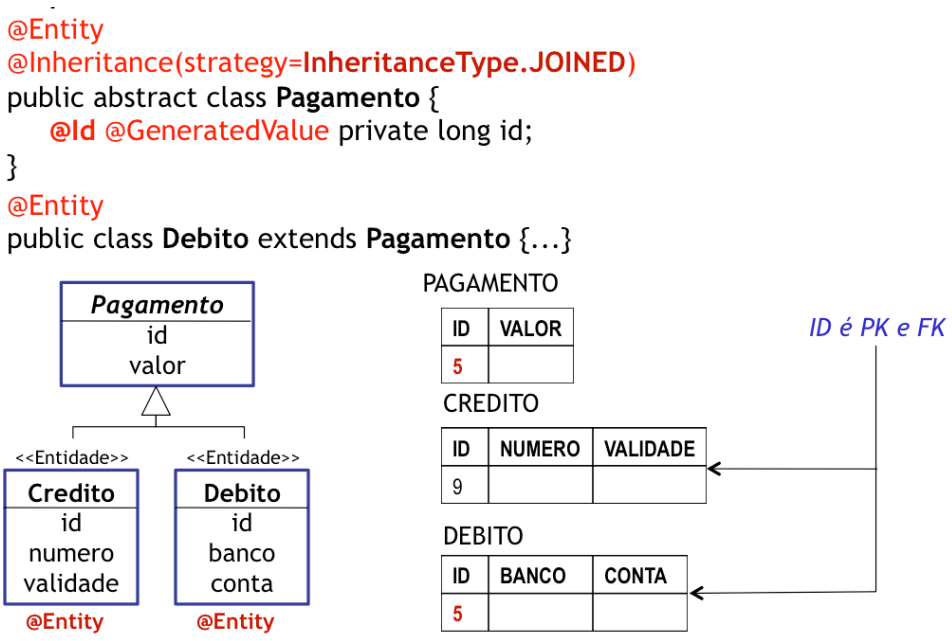
A principal desvantagem dessa estratégia é que as tabelas não são normalizadas. A tabela poderá ter muitas colunas e campos vazios. As colunas de propriedades declaradas em subclasses precisam aceitar valores nulos. Isto pode ser ruim para grandes hierarquias.

3.5.4 Tabela por subclasse

A terceira estratégia, "**Uma tabela por subclasse**" é configurada em JPA usando *InheritanceType.JOINED*. Ela representa os relacionamentos "é um" através de relacionamentos "tem um" (chave estrangeira). Cada subclasse possui a sua própria tabela que contém colunas apenas para os campos não-herdados, e para a chave primária que é chave estrangeira da superclasse. A recuperação de dados é realizada de forma eficiente através de um *join* das tabelas.

As vantagens são utilizar um modelo relacional normalizado, permite que a evolução não tenha efeitos colaterais e facilita a aplicação de restrições de integridade. Novas classes e tabelas podem ser criadas sem afetar classes e tabelas existentes.

O diagrama abaixo ilustra um mapeamento desse tipo:



Essa solução é melhor para gerar que para codificar a mão (bem trabalhosa se for feito em um sistema legado). A performance ainda pode não ser a ideal caso as hierarquias sejam muito

complexas. Queries geralmente usam *outer join* para pesquisas polimórficas, e inner join para queries em classes concretas.

Diante das três opções, a escolha de qual estratégia usar depende do contexto de uso. Seguem algumas sugestões a considerar:

- Se não houver necessidade de queries polimórficos ou associações (e houver suporte por parte do provedor de persistência usado) use *Tabela por Classe Concreta* (*InheritanceType.TABLE_PER_CLASS*).
- Se houver necessidade de queries polimórficos, e hierarquia for simples use Tabela por Hierarquia (*InheritanceType.SINGLE_TABLE*), que é default.
- Se houver necessidade de associações polimórficas mas a hierarquia grande (ou não permitir tabelas não normalizadas) use *Tabela por Subclasse* (*InheritanceType.JOINED*).

Rode os exemplos que exploram estratégias de mapeamento de herança, e observe o log do provedor JPA que mostra o SQL gerado em cada situação.

3.6 Outros mapeamentos

3.6.1 Mapeamento de enumerações

Enumerações podem ser persistidas se marcadas com a anotação @Enumerated. Por exemplo:

```
public enum Regiao {
    NORTE, NORDESTE, SUL, SUDESTE, CENTRO_OESTE;
}
```

Para usar o enum acima em uma entidade e ter os valores persistentes, deve-se usar:

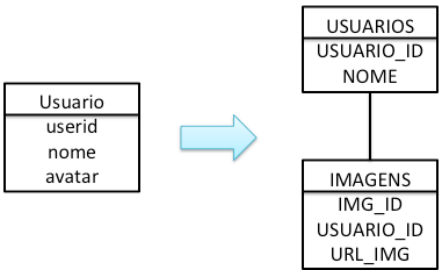
```
@Entity
public class SalaDeCinema {
...
@Enumerated(EnumType.STRING)
private Regiao regiao;
```

O EnumType *default* é *ORDINAL*, que grava um número no registro da tabela. *STRING* é mais seguro porque não é afetado pela *ordem* dos enums na classe (*STRING* grava texto na tabela, que independe da ordenação ou re-ordenação dos elementos do enum). Se forem inseridos outros itens no enum, mudando a ordem das constantes, os registros inseridos antes da mudança ficarão inconsistentes.

3.6.2 Mapeamento de uma instância a duas tabelas

Usa-se *@SecondaryTable* para construir uma entidade que obtém seus dados de duas tabelas.

No exemplo abaixo a entidade *Usuario* está mapeado a uma tabela principal (*USUARIOS*) e uma tabela secundária (*IMAGENS*) de onde obtém o estado do atributo *avatar* da coluna *URL_IMG*.



@Table e usado para mapear a tabela principal (se necessário) e *@SecondaryTable* informa a tabela secundária, indicando a(s) coluna(s) contendo a chave primária usada(s) para realizar a junção.

@PrimaryKeyJoinColumn informa a chave primária da tabela principal, e a chave referenciada na tabela secundária (se forem iguais, apenas o primeiro atributo é necessário). O mapeamento do atributo da segunda tabela usa *@Column* para informar a tabela e a coluna que deve ser mapeada.

```
@Entity
@Table(name="USUARIOS")
@SecondaryTable(name="IMAGENS",
    pkJoinColumns = @PrimaryKeyJoinColumn(name="USUARIO_ID",
                                          referencedColumnName="IMG_ID"))
public class Usuario implements Serializable {
...
    @Id Column(name="USUARIO_ID")
    private Long userId;

    @Column(table="IMAGENS", name="URL_IMG", nullable=true)
    private String avatar;
    ...
```

4 Queries

Uma vez configurados os mapeamentos, a aplicação poderá ser usada para inserir dados, e pesquisas podem ser construídas para localizar e recuperar entidades, com base em seu conteúdo. JPA possui duas estratégias nativas para realizar tais pesquisas:

- JPQL – uma linguagem similar a SQL baseada em comandos de texto, e
- Criteria – uma API usada para construir pesquisas através da hierarquia de objetos.

Em termos de resultado, tanto faz uma API como a outra. Todas possuem vantagens e desvantagens. JPQL é geralmente mais fácil de aprender, mas Criteria permite a criação de queries com maior potencial de reuso e evolução, além de ser recomendada para queries dinâmicas.

4.1 Cláusulas e joins

Queries em objetos têm princípios similares a queries em bancos de dados, mas há algumas diferenças.

Em todas as queries, a cláusula mais importante é a que declara *onde a consulta está sendo executada*. A API Criteria representa essa cláusula por um objeto chamado *Root*, ou raiz, e é representado nas duas formas de consulta pela cláusula *FROM*.

A clausula SELECT indica *o que* está sendo selecionado. Consultas em JPA permitem selecionar entidades e seus atributos, e resultados de operações realizados sobre eles. É possível também selecionar múltiplos elementos.

A cláusula WHERE é opcional inclui filtros que restringem os resultados da pesquisa. Sem uma cláusula WHERE todos os elementos declarados no FROM serão consultados.

Quando as consultas são realizadas sobre objetos que possuem relacionamentos, os objetos que fazem parte do relacionamento são incluídos na consulta através de uma operação de JOIN, que podem e costumam ser implícitos tanto em JPQL como em Criteria. Operações que navegam no grafo de objetos relacionados usando o operador ponto (ex: *cliente.pedido.item*) escondem joins implícitos.

Joins podem ser usados com qualquer um dos quatro tipos de relacionamento, e com mapeamentos de ElementCollection.

Joins em árvores de objetos são parecidos mas não são iguais a joins no mundo relacional. Há três tipos de joins em JPA. :

- Inner Joins (*default* – os joins implícitos são sempre inner joins)
- Left (ou outer) joins
- Fetch joins (que podem ser inner ou left)

Inner Joins consideram apenas os resultados que contém o relacionamento. Esse é o comportamento default. Se dentre os resultados houver elementos que não possuem o

relacionamento declarado no JOIN, eles serão filtrados. Por exemplo, em uma consulta sobre objetos Cliente para obter o total de pedidos, um inner join não retorna os clientes que não têm pedidos. Já o left join retorna tudo, mesmo que o campo seja null.

A forma de inicialização dos elementos de uma coleção geralmente é lazy por default, e o comportamento da aplicação depende de como o contexto transacional é usado, e do provedor de persistência usado. Fetch Joins são usados para garantir que a coleção usada no relacionamento seja inicializada antes do uso. É uma alternativa melhor que declarar esse tipo de comportamento como default no mapeamento.

Exemplos de configuração de joins serão demonstrados nas seções a seguir usando JPQL e Criteria.

4.2 JPQL

JPA Query Language (JPQL) é uma linguagem de recuperação de dados similar a outras linguagens de query de objetos (HQL, EJB-QL, etc.) que a precederam. Parece com SQL, mas é diferente: opera sobre *objetos* e não tabelas, e é mais simples e eficiente para uso em aplicações orientadas a objetos.

Consultas em JPQL são objetos da classe *Query/TypedQuery* e podem ser construídas através de métodos de EntityManager. As instruções do query podem ser passadas diretamente para o método *createQuery()* como parâmetro do tipo String.

```
@PersistenceContext
EntityManager em;
...
TypedQuery query = em.createQuery("SELECT p FROM Produto p", Produto.class);
```

O query pode também ser declarado previamente em anotações *@NamedQuery* em cada entidade:

```
@Entity
@NamedQueries({
     @NamedQuery(name="selectAllProdutos",  query="SELECT p FROM Produto p")
})
public class Produto implements Serializable { ... }
```

associado a um identificador de referência, que é passado para *createNamedQuery()*:

```
TypedQuery query = em.createNamedQuery("selectAllProdutos", Produto.class);
```

Uma consulta pode ser parametrizada. Os parâmetros são declarados em JPQL usando identificadores numéricos ou nomes precedidos por ":":

```
TypedQuery query =
   em.createQuery("SELECT p FROM Produto p WHERE p.preco > :maximo", Produto.class);
```

Se houver parâmetros, eles podem ser preenchidos através de um ou mais métodos setParameter() antes de executar o query.

```
query.setParameter("máximo", 1000);
```

Os métodos de execução retornam os resultados. Se o resultado for uma coleção de entidades, pode-se usar getResultList():

```
List<Produto> resultado = query.getResultList();
```

Se retornar apenas um item (por exemplo, se o produto for selecionado por ID ou um campo unívoco), pode-se usar getSingleResult():

```
Produto produto = query.getSingleResult();
```

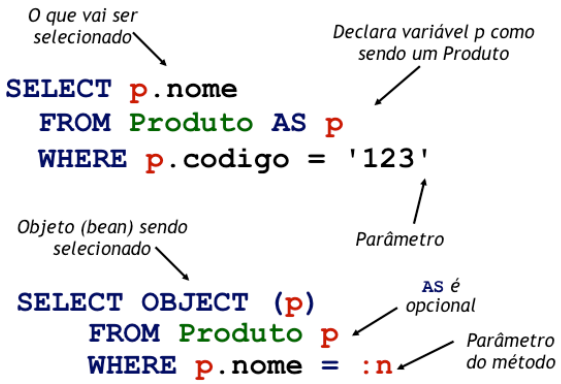
Existem outros métodos, inclusive métodos que não fazem pesquisa mas são usados para remoção ou atualização, como executeUpdate(). Existem também queries mais complexos que recebem muitos parâmetros e que devolvem outros dados, coleções de valores, atributos, e não apenas entidades.

4.2.1 Sintaxe essencial do JPQL

A principal cláusula de JPQL é SELECT. Também há suporte a UPDATE e DELETE, embora sejam menos usados. A sintaxe de uma expressão SELECT é:

```
cláusula_select cláusula_from
[cláusula_where] [cláusula_group_by] [cláusula_having] [cláusula_order_by]
```

As cláusulas entre colchetes são opcionais. O trecho abaixo ilustra a sintaxe elementar da cláusula SELECT:



A cláusula *FROM* é quem informa *qual o objeto que está sendo pesquisado*, e declara aliases usados no resto do query. Cada alias tem um identificador e um tipo. O *identificador* é qualquer palavra não-reservada e o *tipo* é o nome da entidade identificada como *@Entity*. A palavra-chave AS conecta o tipo ao identificador, mas é opcional e raramente é usada:

```
FROM Produto AS p
```

A cláusula FROM também pode selecionar múltiplos objetos e atributo e incluir vários conectores de JOIN (inner join, left outer join).

A cláusula *SELECT* informa *o que se deseja obter com a pesquisa*. Pode retornar *entidades*, *atributos* dos objetos, resultados de expressões, ou múltiplos elementos envolvendo atributos, entidades e resultados de expressões. Pode ser seguida de DISTINCT para eliminar valores duplicados nos resultados. SELECT utiliza o *alias* declarado na cláusula FROM:

```
SELECT p FROM Produto p
SELECT DISTINCT p FROM Produto p
SELECT p.codigo FROM Produto p
SELECT DISTINCT p.codigo FROM Produto p
```

A cláusula *WHERE* é opcional e restringe os resultados da pesquisa com base em uma ou mais expressões condicionais concatenadas. As expressões podem usar: *literais* (strings, booleanos ou números), *identificadores* (declarados no FROM), *operadores*, *funções* e *parâmetros* identificados por um número sequencial (?1, ?2, etc.) ou um identificador (:nome, :item).

Os literais usados nas pesquisas podem ser:

- Strings, representados entre apóstrofes: 'nome'
- Números, que têm mesmas regras de literais Java long e double
- Booleanos, representados por TRUE e FALSE (case-insensitive)

Os operadores do JPQL incluem:

- Expressões matemáticas +, -, *, /
- Expressões de comparação =, >, >=, <, <=, <>
- Operadores lógicos *NOT, AND, OR*
- Outros operadores: *BETWEEN, NOT BETWEEN, IN, NOT IN, LIKE, NOT LIKE, NULL, NOT NULL, IS EMPTY, IS NOT EMPTY, MEMBER, NOT MEMBER*

O operador LIKE possui operadores adicionais usados para indicar a parte de um string que está sendo pesquisada:

- _ representa um único caractere
- **%** representa uma seqüência de zero ou mais caracteres
- \ caractere de escape (necessário para usar _ ou %) literalmente

Funções podem aparecer em várias cláusulas. Algumas das funções agregadas do JPQL incluem:

- Manipulação de strings (usados em WHERE): CONCAT, SUBSTRING, TRIM, LOWER, UPPER
- LENGTH, LOCATE
- Funções aritméticas (usados em WHERE): ABS, SQRT, MOD, SIZE
- Data e hora (usados em WHERE): CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP
- Funções de agregação (usados em SELECT): COUNT, MAX, MIN, AVG, SUM

Cada versão nova do JPA têm introduzido novos comandos, funções, operadores. Para uma abordagem mais completa, consulte a documentação oficial do JPA. Apresentaremos o JPQL através de exemplos que ilustram várias situações típicas. Rode os exemplos fornecidos com este tutorial para verificar os resultados.

4.2.2 Exemplos de queries simples JPQL

Entendendo-se a estrutura básica do JPQL, não é muito difícil compreender os queries. Seguem alguns exemplos de queries usando JPQL:

1) Encontre todos os produtos que são chips e cuja margem de lucro é positiva

```
SELECT p
  FROM Produto p
    WHERE (p.descricao = 'chip' AND (p.preco - p.custo > 0)
```

2) Encontre todos os produtos cujo preço é pelo menos 1000 e no máximo 2000

```
SELECT p
  FROM Produto p
    WHERE p.preco BETWEEN 1000 AND 2000
```

3) Encontre todos os produtos cujo fabricante é Sun ou Intel

```
SELECT p
  FROM Produto p
    WHERE p.fabricante IN ('Intel', 'Sun')
```

4) Encontre todos os produtos com IDs que começam com 12 e terminam em 3

```
SELECT p
  FROM Produto p
    WHERE p.id LIKE '12%3'
```

5) Encontre todos os produtos que têm descrições null

```
SELECT p
  FROM Produto p
    WHERE p.descricao IS NULL
```

6) Encontre todos os pedidos que não têm itens (coleção)

```
SELECT pedido
  FROM Pedido pedido
    WHERE pedido.itens IS EMPTY
```

7) Retorne os pedidos que contem um determinado item (passado como parâmetro)

```
SELECT pedido
  FROM Pedido pedido
    WHERE :item IS MEMBER pedido.itens
```

8) Encontre produtos com preços entre 1000 e 2000 ou que tenham código 1001

```
SELECT p
   FROM Produto p
      WHERE p.preco BETWEEN 1000 AND 2000 OR codigo = 1001
```

4.2.3 Exemplos de queries usando relacionamentos

Pesquisas envolvendo relacionamentos produzem joins (que podem ser implícitos). Os dois queries a seguir produzem o mesmo resultado. O primeiro possui um inner join *implícito*:

9) Selecione todos os clientes com pedidos que tenham total maior que 1000:

```
SELECT c
  FROM Cliente c, IN(c.pedidos) p
    WHERE p.total > 1000
```

O mesmo query poderia ser escrito desta forma, expondo o join:

```
SELECT c
  FROM Cliente c
  INNER JOIN c.pedidos AS p
    WHERE p.total > 1000
```

Os três queries a seguir também obtém o mesmo resultado. O primeiro contém três joins implícitos (cada ponto é um join).

10) Selecione todos os lances onde o item é de categoria que começa com "Celular" e que tenha obtido lance acima de 1000:

```
SELECT lance
  FROM Lance lance
    WHERE lance.item.categoria.nome LIKE 'Celular%'
      AND lance.item.lanceObtido.total > 1000
```

Um dos joins é exposto neste segundo query:

```
SELECT lance
  FROM Lance lance
     JOIN lance.item item
      WHERE item.categoria.nome LIKE 'Celular%'
        AND item.lanceObtido.total > 1000
```

Finalmente, todos os joins são expostos neste query:

```
SELECT lance
  FROM Lance AS lance
   JOIN lance.item AS item
   JOIN item.categoria AS cat
   JOIN item.lanceObtido AS lanceVencedor
    WHERE cat.nome LIKE 'Celular%'
    AND lanceVencedor.total > 1000
```

4.2.4 Exemplos de queries usando funções, group by e having

11)Encontre a média do total de todos os pedidos:

```
SELECT AVG(pedido.total) FROM Pedido pedido
```

12)Obtenha a soma dos preços de todos os produtos dos pedidos feitos no bairro de Botafogo:

```
SELECT SUM(item.produto.preco)
   FROM Pedido pedido
    JOIN pedido.itens item
    JOIN pedido.cliente cliente
       WHERE cliente.bairro = 'Botafogo' AND cliente.cidade = 'Rio de Janeiro'
```

13)Obtenha a contagem de clientes agrupadas por bairro:

```
SELECT cliente.bairro, COUNT(cliente)
   FROM Cliente cliente GROUP BY cliente.bairro
```

14)Obtenha o valor médio dos pedidos, agrupados por pontos, para os clientes que têm entre 1000 e 2000 pontos:

```
SELECT c.pontos, AVG(pedido.total)
   FROM Pedido pedido
     JOIN pedido.cliente c
       GROUP BY c.pontos
         HAVING c.pontos BETWEEN 1000 AND 2000
```

4.2.5 Exemplos de subqueries

É possível usar os resultados de um query como parâmetro de outro através de subqueries. O query abaixo usa como restrição o resultado de um query que será testado com EXISTS.

15)Obtenha os empregados que são casados com outros empregados:

```
SELECT DISTINCT emp
  FROM Empregado emp
    WHERE EXISTS (SELECT conjuge
                    FROM Empregado conjuge
                       WHERE conjuge = emp. conjuge)
```

ALL e ANY são usados com subqueries. ALL retorna true se todos os valores retornados forem true, e ANY só retorna true o resultado da query for vazio ou se todos os valores retornados forem false.

16)Retorne apenas os produtos cujo preço seja maior que o valor incluído em todos os orçamentos:

```
SELECT produto
  FROM Produto p
    WHERE p.preco > ALL (SELECT o.item.preco
                           FROM Orcamento o
                             WHERE o.item.codigo = p.codigo)
```

4.2.6 Queries que retornam múltiplos valores

Queries que retornam múltiplos valores têm os resultados armazenados em arrays de objetos, onde cada índice do array corresponde respectivamente ao item selecionado, na ordem em que é expresso em JPQL. Considere por exemplo a seguinte entidade:

```
@Entity
public class Filme {
    @Id private Long id;
    private String imdb;
    private String titulo;
    @ManyToMany
    private Diretor diretor;
    ...
}
```

A query a seguir retorna três valores: um Long, um String e um Diretor, que é uma entidade, respectivamente.

```
SELECT f.id, f.titulo, d
  FROM Filme f join f.diretores d
    WHERE d.nome LIKE '%Allen'
```

A execução desse query irá retornar cada elemento como um *array* do tipo Object[] onde cada valor será armazenado em um índice. Se houver mais de um resultado, o tipo retornado será List<Object[]>. Por exemplo, para obter os dados da query acima pode-se usar:

```
List<Object[]> resultado = (List<Object[]>)query.getResultAsList();
for(Object obj : resultado) {
    Long id         = (Long)obj[0];
    String titulo   = (String)obj[1];
    Diretor diretor = (Diretor)obj[2];
...
}
```

Muitas vezes, queries que retornam múltiplos valores são pesquisas com o objetivo de gerar relatórios, ou para coletar dados para uma interface. Depois de extrair os dados do array, provavelmente eles serão enviados para um objeto para preencher alguma View. Considere, por exemplo, esta classe, que poderia ser usada para armazenar os dados da query:

```
package com.acme.filmes;
public class DataTransferObject {
    private Long id;
    private String titulo;
    private Diretor diretor;
    public DataTransferObject(Long id, String titulo, Diretor diretor) {
        this.id = id;
        this.titulo = titulo;
        this.diretor = diretor;
    }
    ...
}
```

Poderíamos chama-la dentro do loop:

```
List<Object[]> resultado = (List<Object[]>)query.getResultAsList();
for(Object obj : resultado) {
    Long id         = (Long)obj[0];
    String titulo   = (String)obj[1];
    Diretor diretor = (Diretor)obj[2];
     DataTransferObject dto = new DataTransferObject(id, titulo, director);
    // envia dto para algum lugar
}
```

Uma alternativa a *Object[]* é declarar o tipo do *CriteriaQuery* com a interface *javax.persistence.Tuple*. Os resultados ainda precisam ser extraídos individualmente, mas são então retornados em um objeto que permite recuperar os elementos como uma *List* (tupla.get(0), tupla.get(1), etc.)

Existe, porém, uma sintaxe de SELECT que elimina a necessidade de escrever todo esse código, e permite chamar o construtor diretamente dentro do query:

```
SELECT new com.acme.filmes.DataTransferObject(f.id, f.titulo, d)
  FROM Filme f join f.diretores d
    WHERE d.nome LIKE '%Allen'
```

A especificação do JPA 2.1, porém, requer que o construtor use o nome qualificado da classe. Usando esta sintaxe, o query retornará uma *List<DataTransferObject>* em vez de uma *List<Object[]>*, que poderá ser enviada diretamente para o componente que irá usá-la:

```
@Named
public class ManagedBean {
    public List<DataTransferObject> getDataToPopulateComponent() {
        ...
```

```
        return query.getResultAsList();
    }
    ...
}
```

4.2.7 Named Queries

Queries podem ser declarados em anotações e recuperadas pelo nome. Isto, em geral é uma boa prática porque mantém todos os queries juntos, e pode facilitar a manutenção deles. Por outro lado, os mantém distante do código que cria os queries e preenche os parâmetros, que pode dificultar o uso.

Para declarar queries desta forma, use a anotação *@NamedQueries* como mostrado no exemplo abaixo:

```
@Entity
@NamedQueries({
    @NamedQuery(name="produtoMaisBarato",
        query="SELECT x FROM Produto x WHERE x.preco > ?1"),
    @NamedQuery(name="produtoPorNome",
        query="SELECT x FROM Produto x WHERE x.nome = :nomeParam")
})
public class Produto { ... }
```

Para usar, chame o query pelo nome quando usar o EntityManager através do método createNamedQuery(), e preencha seus parâmetros se houver:

```
EntityManager em = ...
Query q1 = em.createNamedQuery("produtoMaisBarato");
q1.setParameter(1, 10.0);
List<Produto> resultado1 = q1.getResultList();

Query q2 = em.createNamedQuery("produtoPorNome");
q2.setParameter("nomeParam", 10.0);
List<Produto> resultado2 = q2.getResultList();
```

Named queries não são sempre recomendados, mas queries parametrizados sim. Deve-se evitar a construção de queries através da concatenação de strings por vários motivos. Queries parametrizados não apenas tornam o código mais legível e evitam erros, como melhoram a performance e evitam a necessidade de converter tipos.

4.3 Criteria

Criteria é uma API do JPA que permite a construção de queries dinâmicos usando objetos. Os queries são expressos usando construções orientadas a objeto.

Uma das vantagens da API Criteria é a possibilidade de descobrir erros de sintaxe em queries em tempo de compilação. Por exemplo, considere o query JPQL:

```
select p from Produto where p.preco < 50.0
```

É muito fácil esquecer uma letra e é muito difícil achar o erro. O que está errado no query acima?

Apenas uma letra. Este é o query correto:

```
select p from Produto p where p.preco < 50.0
```

Portanto, se o programador esquecer de digitar o *alias* "p" após *Produto*, a sintaxe do query estará incorreta. A menos que tenha alguma ferramenta especial para validar o JPQL em tempo de desenvolvimento, o programador só descobrirá o erro após o deploy, pois erros da string do JPQL não são descobertos em tempo de compilação.

Por outro lado, queries em Criteria têm muitas linhas e são complexos. É preciso conhecer bem a API antes de obter os benefícios dela. E muitas linhas de código, APIs gigantes, também são uma fonte de bugs. O código necessário para construir um query Criteria que retorne os mesmos resultados que o JPQL mostrado acima seria:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class); // tipo do select
Root<Produto> raiz = query.from(Produto.class);  // tipo do from
Predicate condicao = cb.lessThan(raiz.get("preco"), 50.0); // predicado
query.where(condicao); // adiciona a clausula where
query.select(raiz); // adiciona a clausula select (diz o que vai ser selecionado)
```

Depois o query pode ser manipulado da forma como era manipulado em JPQL, passar parâmetros se houver, executar e extrair resultados.

Apesar de burocráticos e longos, queries Criteria são ideais para a criação de consultas dinâmicas, evitando a arriscada concatenação de strings costuma acontecer em consultas dinâmicas expressas em JPQL.

Queries de Criteria são um *grafo de objetos*. A raiz do grafo é encapsulado no *CriteriaQuery*. Um query mínimo requer o uso de três classes de *javax.persistence.criteria*: *CriteriaBuilder*, que encapsula vários métodos para construir o query, *CriteriaQuery*, que representa a consulta, e *Root* que é raiz da consulta e representa os objetos selecionados pela cláusula *FROM*.

4.3.1 Sintaxe essencial de Criteria

O primeiro passo é criar um CriteriaQuery através da classe CriteriaBuilder:

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

E em seguida obter o objeto que irá encapsular o query:

```
CriteriaQuery<Produto> query = builder.createQuery( Produto.class );
```

O objeto raiz do grafo é declarado através da interface *Root*, que constrói a cláusula "from" do query. Isto é equivalente a fazer *"from Produto p"* em JPQL:

```
Root<Produto> p = query.from (Produto.class);
```

Finalmente constrói-se a cláusula "select" do query usando o método select():

```
query.select(p);
```

O query está pronto. Neste momento ele é equivalente ao string JPQL. Para executá-lo é preciso passar o objeto query como parâmetro de um TypedQuery, para em seguida obter os resultados:

```
TypedQuery<Produto> query = em.createQuery(query);
List<Produto> resultado = query.getResultList();
```

Consultas envolvendo múltiplos objetos podem ser feitas com from(), join() ou fetch() e métodos similares. O query JPQL:

```
SELECT p1, p2 FROM Produto p1, Produto p2
```

Pode ser escrito usando Criteria da seguinte forma:

```
CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);
Root<Produto>  p1 = criteria.from(Produto.class);
Root<Produto> p2 = criteria.from(Produto.class);
criteria.multiselect(p1, p2);
```

Um join como na query abaixo:

```
SELECT item, produto.nome FROM Item item LEFT OUTER JOIN item.produto produto
```

Pode ser escrito em Criteria da seguinte forma:

```
CriteriaQuery<Tuple> criteria = cb.createQuery(Tuple.class);
Root<Item> item = criteria.from(Item.class);
Join<Item> produto = item.join("produto", JoinType.LEFT);
criteria.multiselect(item, produto.get("nome"));
```

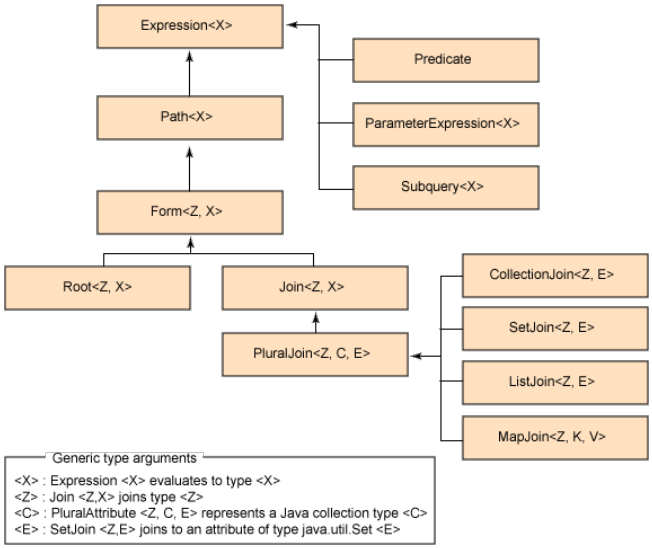
A interface Join (que é retornada pelo método join()) é subinterface da interface *From*, portanto pode ser usada em metódos que esperam um *From*. Um join usando *fetch* como este em JPQL:

```
SELECT item FROM Item item JOIN FETCH item.produto
```

pode ser expresso em Criteria usando a seguinte sintaxe:

```
CriteriaQuery<Item> criteria = cb.createQuery(Item.class);
Root<Item> item = criteria.from(Item.class);
Fetch<Item, Produto> produto = item.fetch("produto");
criteria.select(item);
```

A API Criteria possui muitas interfaces, classes e métodos. O diagrama a seguir ilustra a hierarquia das expressões de query do Criteria, que são usadas para construir as cláusulas da consulta. Um query típico usa a maioria dessas interfaces:



(fonte: http://www.ibm.com/developerworks/library/j-typesafejpa/)

Algumas dessas interfaces são:

- **Expression** – é uma interface que representa uma expressão (é superinterface para uma grande hierarquia de expressões). Expressões são construídas recebendo outras expressões como parâmetro. A maioria é retornada por métodos de CriteriaQuery e CriteriaBuilder (ex: sum(), diff(), equal())
- **Predicate** – é um tipo especial de Expression que representa a conjunção (ou disjunção) das restrições de um query e representa um valor booleano. A cláusula where recebe um Predicate. Predicados complexos são composições de outros predicados. Grande parte é criado por métodos de CriteriaBuilder (ex: equal(), and(), or(), between())
- **Path** – Representa um caminho de navegação em um grafo de objetos, por exemplo: *objeto.referencia.atributo*. O método get() de Root permite acesso a atributos de objetos através de expressões de path.
- **Join** – Representa o join para uma @Entity, objeto @Embeddable ou @Basic.

Algumas outras classes e interfaces importantes da API Criteria são:

- **Selection** – superinterface que representa qualquer item que é retornado em um resultado de query, como uma expressão (Expression), subquery (SubQuery), predicado (Predicate), caminho (Path), condicionais (Case), etc.
- **SubQuery** – Representa um subquery.

A hierarquia que representa um query em Criteria possui duas classes: CriteriaQuery (query principal) e Subquery. Elas têm uma superclasse em comum chamada de AbstractQuery. Os métodos da classe AbstractyQuery representam *cláusulas* e outras partes de um query e valem para queries principais e subqueries. Os principais métodos são:

- Herdados de AbstractQuery (valem para queries principais e subqueries):
  distinct(), from(), groupBy(), having(), subQuery(), where()
- Definidos em CriteriaQuery (valem apenas para queries principais):
  multiselect(), select()
- Definidos em SubQuery (valem apenas para subqueries):
  correlate(), getParent()

A classe CriteriaBuilder possui métodos para a construção de todas as expressões usadas nas cláusulas dos queries e subqueries. As expressões são encapsuladas em instâncias de Expression ou subclasses e podem ser criadas simplesmente chamando os factory methods de CriteriaBuilder. As expressões envolvem desde a simples criação de literais até condicionais, expressões boleanas, de comparação, etc.

Exemplo:

```
Expression<Integer> i1 = builder.literal(123);
Expression<Integer> i2 = builder.sum(3,4);
Predicate p1 = builder.and(true, false);
Predicate p2 = builder.not(p1);
```

Normalmente os métodos são chamados diretamente na construção de condições (usadas em cláusulas where(), por exemplo). Os métodos englobam todas as expressões disponíveis em JPQL e têm nomes auto-explicativos.

A documentação oficial cobre de forma abrangente e com muitos exemplos o uso de queries Criteria. Foge ao escopo deste curso introdutório explorar esses recursos em detalhes, mas, como fizemos com JPQL, apresentaremos vários exemplos que envolvem situações típicas mais comuns. Tente executar o código dos exemplos fornecidos e construir seus próprios queries.

4.3.2 Exemplos de queries simples usando Criteria

Os exemplos abaixo mostram queries em Criteria API que produzem os mesmos resultados que os que foram mostrados anteriormente usando JPQL. Compare as duas formas:

1) Encontre todos os produtos que são chips e cuja margem de lucro é positiva

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
```

```
Predicate igual = cb.equal(root.get("descricao"), "chip");
Expression<Double> subtracao = cb.diff(root.get("preco"), root.get("custo"));
Predicate maiorQue = cb.greaterThan(subtracao, 0.0);
Predicate clausulaWhere = cb.and(igual, maiorQue);

query.where(clausulaWhere);
query.select(root);

TypedQuery<Produto> q = em.createQuery(query);
    // ...
```

2) Encontre todos os produtos cujo preço é pelo menos 1000 e no máximo 2000

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Predicate between = cb.between(root.get("preco"), 1000.0, 2000.0);
query.where(between);
query.select(root);
```

3) Encontre todos os produtos cujo fabricante é Sun ou Intel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(root.get("fabricante").in("Intel", "Sun"));
query.select(root);
```

4) Encontre todos os produtos com IDs que começam com 12 e terminam em 3

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(cb.like(root.get("id"), "12%3"));
query.select(root);
```

5) Encontre todos os produtos que têm descrições null

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);
query.where(cb.isNull(root.get("descricao")));
query.select(root);
```

6) Encontre todos os pedidos que não têm itens (coleção)

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pedido> query = cb.createQuery(Pedido.class);
Root<Pedido> root = query.from(Pedido.class);
query.where(cb.isEmpty(root.get("itens")));
query.select(root);
```

7) Retorne os pedidos que contem um determinado item (passado como parâmetro)

```
Item item = new Item(); // item a ser testado
// ...
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pedido> query = cb.createQuery(Pedido.class);
Root<Pedido> root = query.from(Pedido.class);
Predicate isMember = cb.isMember(item, root.get("itens"));
query.where(isMember);
query.select(root);
```

8) Encontre produtos com preços entre 1000 e 2000 ou que tenham código 1001

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Predicate between = cb.between(root.get("preco"), 1000.0, 2000.0);
Predicate igual   = cb.equal(root.get("codigo"), 1001);

query.where(cb.and(between, igual));
query.select(root);
```

4.3.3 Exemplos de queries usando relacionamentos

Pesquisas envolvendo relacionamentos produzem joins (que podem ser implícitos). Os dois queries a seguir produzem o mesmo resultado. O primeiro possui um inner join *implícito*:

9) Selecione todos os clientes com pedidos que tenham total maior que 1000:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cliente> query = cb.createQuery(Cliente.class);
Root<Cliente> root  = query.from(Cliente.class);
Path<Pedido> pedido = root.get("pedidos");

query.where(cb.greaterThan(pedido.get("total"), 1000.0));
query.select(root);
```

Mesmo query usando um inner join explícito:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Cliente> query = cb.createQuery(Cliente.class);
Root<Cliente> root = query.from(Cliente.class);

Join<Cliente, Pedido> pedido = root.join("pedidos");

query.where(cb.greaterThan(pedido.get("total"), 1000.0));
query.select(root);
```

10) Selecione todos os lances onde o item é de categoria que começa com "Celular" e que tenha obtido lance acima de 1000:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

```
CriteriaQuery<Lance> query = cb.createQuery(Lance.class);
Root<Lance> root = query.from(Lance.class);

query.where(cb.and(
    cb.like(root.get("item").get("categoria").get("nome"), "Celular%"),
    cb.greaterThan(root.get("item").get("lanceObtido").get("total"), 1000.0))
);

query.select(root);
```

Mesmo query usando vários inner joins explícitos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Lance> query = cb.createQuery(Lance.class);
Root<Lance> root = query.from(Lance.class);

Join<Lance, Item> item     = root.join("item");
Join<Item, Categoria> cat  = item.join("categoria");
Join<Item, Lance> vencedor = item.join("lanceObtido");

query.where(cb.and(cb.like(cat.get("nome"), "Celular%"),
                   cb.greaterThan(vencedor.get("total"), 1000.0)));
query.select(root);
```

4.3.4 Exemplos de queries usando funções, group by e having

11)Encontre a média do total de todos os pedidos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Double> query = cb.createQuery(Double.class); // avg retorna Double
Root<Pedido> root = query.from(Pedido.class); // from Ingresso i
Expression<Double> media = cb.avg(root.get("total"));
query.select(media);
```

12)Obtenha a soma dos preços de todos os produtos dos pedidos feitos no bairro de Botafogo:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<BigDecimal> query = cb.createQuery(BigDecimal.class);
Root<Pedido> root = query.from(Pedido.class);

Join<Pedido, Item>    item     = root.join("itens");
Join<Pedido, Cliente> cliente  = item.join("cliente");

Predicate where = cb.and(
     cb.equal(cliente.get("bairro"), "Botafogo"),
     cb.equal(cliente.get("cidade"), "Rio de Janeiro")
);
query.where(where);
Expression<BigDecimal> total = cb.sum(item.get("produto").get("preco"));
query.select(total);
```

13)Obtenha a contagem de clientes agrupadas por bairro:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
```

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Cliente> root = query.from(Cliente.class);
query.multiselect(root.get("bairro"), cb.count(root));
query.groupBy(root.get("bairro"));

TypedQuery<Object[]> q = em.createQuery(query);
...
```

14) Obtenha o valor médio dos pedidos, agrupados por pontos, para os clientes que têm entre 1000 e 2000 pontos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Pedido> root = query.from(Pedido.class);
query.multiselect(root.get("pontos"), cb.avg(root.get("total")));
Join<Pedido, Cliente> cliente  = root.join("cliente");
query.groupBy(cliente.get("pontos"));
query.having(cb.between(cliente.get("pontos"), 1000.0, 2000.0));
```

4.3.5 Exemplos com subqueries

15) Obtenha os empregados que são casados com outros empregados:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Empregado> query = cb.createQuery(Empregado.class);
Root<Empregado> root = query.from(Empregado.class);

Subquery<Empregado> subquery = query.subquery(Empregado.class);
Root<Empregado> conjuge  = subquery.from(Empregado.class);
subquery.where(cb.equal(conjuge.get("conjuge "), root.get("conjuge ")));
subquery.select(conjuge);

query.where(cb.exists(subquery));
query.select(root).distinct(true);
```

16) Retorne apenas os produtos cujo preço seja maior que o valor incluído em todos os orçamentos:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Produto> query = cb.createQuery(Produto.class);
Root<Produto> root = query.from(Produto.class);

Subquery<Empregado> subquery = query.subquery(Empregado.class);
Root<Orcamento> orcamento  = subquery.from(Orcamento.class);
subquery.where(cb.equal(orcamento.get("item ").get("codigo"), root.get("codigo")));
subquery.select(orcamento.get("item").get("preco"));

query.where(cb.greaterThan(root.get("preco"), cb.all(subquery)));
query.select(root);
```

4.3.6 Queries que retornam múltiplos valores

O query usando select com construtor mostrado na seção anterior com JPQL pode ser construído com Criteria da forma abaixo:

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Lugares> query = cb.createQuery(DataTransferObject.class);
Root<Filme> root = query.from(Filme.class);
Join<Filme, Diretor> diretor = root.join("etapas");
query.where(cb.like(root.get("nome"),  "%Allen"));
query.select(cb.construct(Lugares.class,
     etapa.get("origem").get("nome"),
     etapa.get("destino").get("nome"))
);
```

4.3.7 Typesafe Criteria com static metamodel

É bem mais fácil encontrar erros em queries Criteria, comparados a JPQL, porque o compilador ajuda na tarefa e detecta queries incorretos. Mas eles ainda podem acontecer já que a leitura dos campos das entidades feita através de um get() recebe um String. Por exemplo, a linha abaixo para ler o campo "preco" não contém erros de compilação:

```
Predicate condicao = qb.lt(raiz.get("prco"), 50.0);
```

Mas se o string estiver errado, o query está incorreto. Portanto, existem erros de sintaxe em queries do Criteria que também não serão capturados em tempo de compilação.

Mas há uma solução: o *typesafe query*.

```
CriteriaBuilder qb = em.getCriteriaBuilder();
CriteriaQuery<Produto> cq = qb.createQuery(Produto.class);
Root<Produto> raiz = cq.from(Produto.class)
Predicate condicao = qb.lt(raiz.get(Produto_.preco), 50.0);
cq.where(condicao);
TypedQuery<Person> query = em.createQuery(cq);
```

Observe que o nome do atributo agora é informado através de código Java, usando um atributo estático e público da classe *Produto_* . Essa classe é chamada de *metamodelo estático* (*static metamodel*). Usando os atributos através dela em vez de usar strings torna os queries Criteria typesafe.

Elas precisam ser geradas. Todos os IDEs que suportam JPA 2 têm ferramentas para isto. Por exemplo, em um projeto Maven com provedor EclipseLink, pode-se incluir a geração automática em uma das fazes do POM.xml através de um plugin que requer esta dependência:

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
```

```
  <version>2.5.0</version>
</dependency>
```

E a configuração abaixo. Com isso os metamodelos serão gerados durante a fase de geração de código do build (o plugin usa as classes declaradas no persistence.xml):

```
<plugin>

  <groupId>org.bsc.maven</groupId>
  <artifactId>maven-processor-plugin</artifactId>
  <version>2.2.4</version>
  <executions>
    <execution>
      <id>eclipselink-jpa-metamodel</id>
      <goals>
        <goal>process</goal>
      </goals>
      <phase>generate-sources</phase>
      <configuration>
        <processors>
          <processor>
            org.eclipse.persistence.internal.jpa.modelgen.CanonicalModelProcessor
          </processor>
        </processors>
        <outputDirectory>
            ${project.build.directory}/generated-sources/meta-model
        </outputDirectory>
      </configuration>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
      <version>2.5.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

</plugin>
```

5 Tuning em JPA

Várias operações JPA precisam de um contexto transacional para execução, e todas utilizam um ou mais níveis de cache. Um método transacional pode ser insuficiente para proteger a integridade de um processo e estratégias de locks otimistas ou pessimistas poderão ser necessárias.

Nesta seção serão discutidas algumas configurações que podem ser realizadas em aplicações JPA e que envolvem transações, locks, caches e operações em lote.

5.1 Transações

Transações em JPA podem ser distribuídas (*JTA*) ou configuradas como um recurso local *RESOURCE_LOCAL*. Apenas transações JTA podem ser gerenciadas pelo container e usadas para configurar suporte transacional a métodos de forma transparente e declarativa. Transações JTA também podem ser controladas programaticamente através da API da classe *UserTransaction*, que pode ser injetada como resource em componentes Java EE.

5.1.1 Resource-local javax.persistence.EntityTransaction

Usada em ambientes Java SE ou onde não há suporte a transações distribuídas, o método *getTransaction()* de *EntityManager* pode ser usado para obter o contexto transacional obrigatório para operações de persistência. As transações são delimitadas pela chamada dos métodos *begin()* e *commit()/rollback()*:

```
public class AlunoDAO {
    private EntityManagerFactory emf;

    AlunoDAO() {
        emf = Persistence.createEntityManagerFactory("escola-PU");
    }

     public void addAluno(Aluno aluno) {
         EntityManager em = emf.getEntityManager();

         try
             em.getTransaction().begin();
             em.persist(aluno);
             em.getTransaction().commit();
         } catch(Exception e) {
              em.rollback();
         } finally {
             em.close();
         }
     }
}
```

O *persistence.xml* deve indicar *RESOURCE_LOCAL* como transaction-type:

```
<persistence-unit name="tutorial-jpa" transaction-type="RESOURCE_LOCAL">
```

5.1.2 JTA javax.transaction.UserTransaction

Disponível em servidores Java EE. Pode ser obtida e injetada como um *@Resource* em WebServlets, EJBs, e outros componentes que rodam no container Java EE. Em ambientes Java EE com CDI também pode ser injetada com @Inject.

```
public class AlunoDAOBean {
    @PersistenceContext(unitName="escolar-PU")
    private EntityManager;

    @Resource
    UserTransaction ut;

    public void addAluno(Aluno aluno) {
        try
            ut.begin();
            em.persist(aluno);
            ut.commit();
        } catch(Exception e) {
             ut.rollback();
        } finally {
            em.close();
        }
    }
}
```

Em EJB transações gerenciadas pelo container (CMT) *é o comportamento default* e os métodos de um SessionBean são automaticamente incluídos em um contexto transacional.

```
@Stateless
public class AlunoSessionBean {

    @PersistenceContext
    EntityManager em;

    public void addAluno(Aluno aluno) {
        em.persist(aluno);
    }
}
```

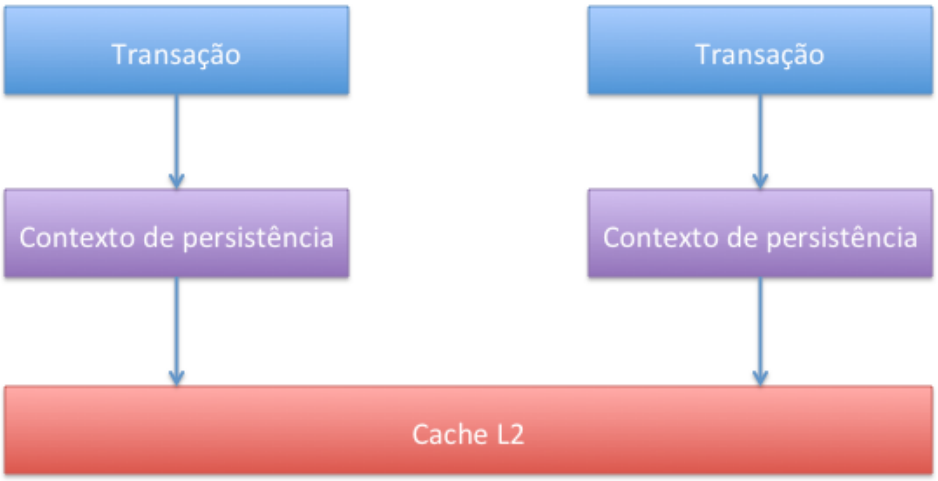
Em CDI o mesmo comportamento pode ser obtido nos métodos declarados como *@Transactional*.

```
@Model
public class AlunoSessionBean {

    @Inject
    EntityManager em;

    @Transactional
    public void addAluno(Aluno aluno) {
```

```
            em.persist(aluno);
        }
    }
```

5.2 Cache

O JPA possui dois níveis de cache. O primeiro é o *Contexto de Persistência*, controlado pelo EntityManager. O segundo nível de cache (L2) é um mecanismo compartilhado. Cache é um recurso que deve ser usado com cuidado, pois há grande risco de resultar em inconsistência de dados que podem inclusive introduzir bugs. Mas o custo-benefício com os ganhos de performance poderá compensar o esforço extra de gerenciar o cache.



5.2.1 Cache de primeiro nível (L1, EntityManager)

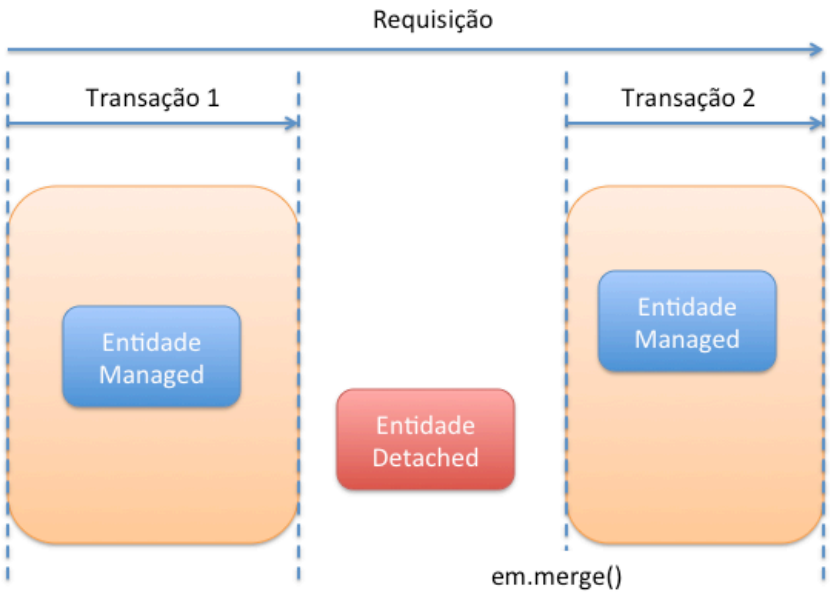
O EntityManager normalmente está vinculado a uma transação, que dura no mínimo o tempo de uma operação. Durante essa operação o objeto é sincronizado com o banco e copiado para o contexto de persistência.

Nos ambientes que possuem transações gerenciadas pelo container, a duração do contexto de persistência é a mesma do contexto transacional, que geralmente é aplicado em métodos que podem conter várias operações do EntityManager. Dentro do contexto transacional, o contexto de persistência mantém o estado dos objetos e evita que eles tenham que ser recuperados do banco.

O padrão recomendado para ambientes com transações controladas pela aplicação é também abrir a o contexto transacional logo após a obtenção da sessão do EntityManager, e fechar o EntityManager logo após cometer ou fazer rollback da transação.

O EntityManager garante uma única instância por contexto de persistência. Mas pode haver vários contextos (ex: transações de outros usuários). Neste caso ainda é preciso usar estratégias de locking para garantir a consistência de dados.

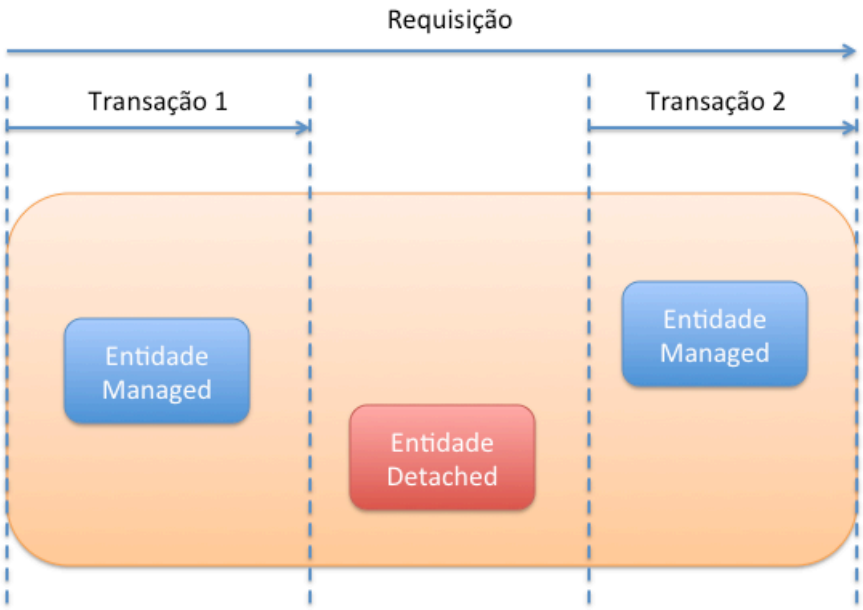
Quando uma transação e o contexto de persistência são fechados, as entidades entram no estado datached (desligadas) e precisam de alguma política de locking para garantir sua integridade, caso precisem ser religadas na próxima transação. A imagem abaixo ilustra essa situação.



O contexto de persistência pode ser estendido para manter-se aberto além do escopo de uma transação. Isto permitirá que a aplicação tenha acesso aos objetos carregados no contexto de persistência, que poderão ser usados para leitura ou exibição em uma View. Em transações controladas pela aplicação, o EntityManager é aberto no início da requisição, e só é fechado (com close()) no final. O objeto poderá, nesse intervalo, passar por várias transações e usando os dados do cache do contexto de persistência.

Em ambientes gerenciados pelo container, o EntityManager pode ser injetado com a opção *PersistenceContextType.EXTENDED* que manterá o contexto de persistência aberto entre transações.

```
@PersistenceContext(PersistenceContextType.EXTENDED)
private EntityManager entityManager;
```



Por um lado existe a vantagem de não precisar fazer uma nova chamada ao banco para ter acesso a uma entidade. Por outro, se a operação se estende por muito tempo há o risco do objeto ficar obsoleto, há uma chance maior dele ser alterado em outra transação (e necessitar de *locking*) e do uso possivelmente desnecessário de recursos.

5.2.2 Cache de segundo nível (L2)

O cache de segundo nível (L2) é compartilhado entre contextos de persistência. É preciso habilitá-lo xplicitamente. Pode-se habilitar o cache para todas as entidades introduzindo a seguinte linha no persistence.xml:

```
<persistence-unit ...>
   <shared-cache-mode>ALL</shared-cache-mode>
</persistence-unit>
```

ou através de uma propriedade:

```
<persistence-unit ...>
...
  <properties>
      ...
      <property name="javax.persistence.sharedCache.mode" value="ALL"/>
  </properties>
</persistence-unit>
```

Habilitar o cache para todas as entidades pode não ser uma boa idéia. A estratégia oposta é desabilitar para todas as classes *exceto* aquelas anotadas como *@Cacheable*. Neste caso, em vez de ALL, use *ENABLE*_SELECTIVE. Que desabilita o cache para todas as entidades, a menos que elas tenham a anotação *@Cacheable*.

```
@Cacheable
@Entity
public class Produto { … }
```

Se a maioria dos objetos deve participar do cache, pode-se usar a opção *DISABLE_SELECTIVE*. Neste caso, todas as entidades vão para o cache L2 a menos que tenham declarado *@Cacheable(false).*

Se o serviço estiver habilitado, objetos que não forem encontrados no contexto de persistência serão buscados no cache L2.

Para que um objeto tenha acesso ao cache é preciso utilizar a API da classe Cache. Pode-se obter uma instância de Cache através de

```
Cache cache = emf.getCache();
```

Os principais métodos dessa classe são:

- *contains(classe, entidade)* – que retorna true se o objeto estiver no cache.
- *evict(classe, objeto)* – remove o objeto do cache
- *evictAll()* – esvazia o cache

O trecho de código abaixo mostra como verificar se um objeto está no cache, e remove-lo do cache se for o caso.

```
boolean isCached = cache.contains(Produto.class, Long.valueOf(123));
if (isCached) {
     cache.evict(Produto.class, Long.valueOf(123));
}
```

Pode-se também invalidar todas as entidades de uma classe:

```
cache.evict(MyEntity.class);
```

Ou ainda esvaziar o cache inteiro:

```
cache.evictAll();
```

Os objetos cacheáveis são automaticamente adicionados ao cache durante o commit().

O cache também pode ser configurado para um EntityManager específico, ou mesmo para um query específico usando a propriedade *javax.persistence.cache.retrieveMode*.

```
Query query = em.createQuery("Select p from Produto p");
query.setHint("javax.persistence.cache.retrieveMode", CacheStoreMode.BYPASS);
```

A propriedade *retrieveMode* informa um modo de gravação para o query (*CacheStoreMode*), que pode ter os valores:

- *USE* (usa o objeto do cache)

- *BYPASS* (ignora o objeto do cache)
- REFRESH (atualiza o objeto do cache)

Usar o cache de segundo nível tem vantagens e desvantagens. Entre as vantagens está evitar acessar o banco para entidades que já foram carregadas. Isto é recomendado para entidades que são acessadas com frequência e nunca ou raramente são modificadas. Entidades que são boas candidatas ao cache L2 são aquelas que são lidas com frequência, modificadas raramente e que não sejam inválidas se desatualizadas.

Como desvantagens estão o consumo de memória maior e objetos que podem ficar obsoletos (caso sejam atualizados no banco). Obviamente esse cache não deve ser usado para objetos que serão atualizados com frequência.

5.3 Locks

Locks travam um objeto para evitar alterações. Há duas estratégias:

- Lock otimista: não impede o acesso ao objeto, mas incrementa um número de versão das alterações. Se o número mudar antes do commit, rejeita as alterações. Esta é a estratégia default.
- Lock pessimista: impede acesso a um objeto, geralmente por um determinado tempo e por um determinado modo (leitura, gravação)

5.3.1 Locks otimistas

Uma trava otimista é configurado através de mapeamento e aplicado automaticamente no commit de uma transação.

Para usar um campo de versão é preciso definir um atributo numérico e anotá-lo com *@Version*. Pode ser especificada em um atributo numérico inteiro (Long/long, Integer/int, Short/short) ou Timestamp:

```
@Entity
public class Produto {
  @Version
  long version;
}
```

O provedor de persistência incrementa automaticamente esse campo a cada commit realizado com sucesso (não deve ser manipulado pela aplicação). Se outra transação tenta alterar a entidade e a versão foi alterada desde a última leitura ocorrerá uma *OptimisticLockException*.

Em alguns provedores JPA o lock otimista é automático. Em outros é necessário configurar uma coluna extra nas tabelas (*@Version*) explicitamente.

5.3.2 Locks pessimistas

Se uma colisão precisar ser revelada antes do commit da transação é preciso usar uma trava pessimista. Travas pessimistas obtém exclusividade de acesso a uma entidade durante o período que uma aplicação estiver usando-a. Apenas uma transação poderá tentar atualizar a entidade por vez. Essa estratégia limita o acesso concorrente aos dados.

Travas pessimistas são criadas e usadas através do EntityManager. Para travar um objeto, use o método *lock*() escolhendo o tipo com *LockModeType*. Há vários tipos. Os principais são *PESSIMISTIC_READ* (compartilhada) e *PESSIMISTIC_WRITE* (exclusiva):

```
em.lock(produto, LockModeType.PESSIMISTIC_WRITE);
```

O método lock() requer uma transação ativa (será lançada uma exceção se ela não estiver presente). Poderá haver também uma exceção (*LockTimeoutException*) se a trava não puder ser fornecida (ex: se for uma trava compartilhada e outro usuário tiver uma trava exclusiva, por exemplo).

Definir um timeout é importante para, entre outras coisas, evitar deadlock. A configuração default pode ser feita em persistence.xml

```
<properties>
    <property name="javax.persistence.lock.timeout" value="1000"/>
</properties>
```

Mas também pode ser especificada programaticamente via EntityManager ou até mesmo para um query específico. As travas são sempre liberadas ao final da transação.

As travas podem ser usadas durante a recuperação de objetos (métodos find(), refresh() e queries). É possível também sobrepor o timeout default.

```
Map<String,Object> props = new HashMap();
properties.put("javax.persistence.lock.timeout", 2000);
Produto produto = em.find(Produto.class, 1, LockModeType.PESSIMISTIC_WRITE, props);
em.refresh(produto, LockModeType.PESSIMISTIC_WRITE, props);
```

O uso de travas pessimistas é raro e tem impacto na escalabilidade. Deve ser usada como último recurso e não como primeira opção para garantir a integridade de um objeto.

5.4 Operações em lote

Operações em lote (batch) são soluções para um problema conhecido como dos "N+1 queries" comuns em aplicações ORM que gera queries desnecessários e tem grande impacto na

performance, principalmente em consultas grandes. Analisando o SQL gerado, encontra-se, para queries simples, vários queries extras que não são usados. Exemplo:

```
SELECT E.* FROM EMPLOYEE E WHERE E.STATUS = 'Part-time'
... N selects to ADDRESS
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 123
SELECT A.* FROM ADDRESS A WHERE A.ADDRESS_ID = 456
...
... N selects to PHONE
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 789
SELECT P.* FROM PHONE P WHERE P.OWNER_ID = 135
...
```

Uma das formas de resolver esse problema é através de join fetching (*@JoinFetch*). Outra é usando batch fetching que pode ser configurado com a anotação *@BatchFetch*.

```
@BatchFetch(type=BatchFetchType.EXISTS)
```

As opções são EXISTS, IN e JOIN. O ganho de performance é notável em qualquer uma das opções (costuma ser um pouco pior para IN, mas isso depende do tamanho e tipo de dados envolvidos.

6 Referências

[1] Java Persistence 2.1 Expert Group. JSR 338: JavaTM Persistence API, Version 2.1. Oracle. 2013. http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html

[2] Christian Bauer et al. *Java Persistence with Hibernate*, Second Edition. Manning, 2015.

[3] Eric Jendrock et al. *The Java EE 7 Tutorial*. Oracle. 2014. https://docs.oracle.com/javaee/7/JEETT.pdf

[4] Linda deMichiel and Bill Shannon. *JSR 342. The Java Enterprise Edition Specification. Version 7*. Oracle, 2013. http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-spec/JavaEE_Platform_Spec.pdf

[5] Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014.