**FACULDADES SANTO AGOSTINHO – FASA/FACET**

**DISCIPLINA: LINGUAGEM DE PROGRAMAÇÃO IV**

**PROFESSOR: PETRÔNIO CÂNDIDO DE LIMA E SILVA**

**PERÍODO: 7º PERÍODO/ NOTURNO**

**ORM E JPA**

Montes Claros/MG

Março de 2011

# ORM - OBJECT-RELATIONAL MAPPING

A técnica de desenvolvimento ORM, que traduzido significa Mapeamento objeto relacional, tem por finalidade reduzir a impedância[[1]](#footnote-1) da programação orientada a objetos, fazendo uso de banco de dados relacionais. A técnica ORM sugere os meios de se persistir o estado de um objeto (atributos, relacionamentos e herança) em tabelas de banco de dados relacionais (como SQL Server, MySQL, Oracle, DB2 e outros). Diversas plataformas de programação promovem essa persistência, através de frameworks[[2]](#footnote-2).

A filosofia ORM prevê que o desenvolvedor não tenha mais que se preocupar com os comandos e consultas SQL, muitas vezes, bastante confusas e trabalhosas. Nesse caso, tais consultas serão “desenhadas” por uma interface de programação simples, que irá cuidar de todo o trabalho de persistência.

Assim, não será necessária a correspondência direta entre as classes do programa e as tabelas do BD. Esta relação será posteriormente configurada pelo programador, de modo a isolar o código do programa das alterações à organização dos dados nas tabelas do banco de dados. A forma como este mapeamento é configurado depende da ferramenta utilizada. Por exemplo, o programador que utiliza Hibernate[[3]](#footnote-3) na linguagem Java pode usar arquivos XML ou o sistema de anotações que a linguagem disponibiliza.

## Principais ORM do mercado:

Existem várias ORM disponíveis para a plataforma Java no mercado. Abaixo, uma lista dos desenvolvedores e seus respectivos produtos.

|  |  |
| --- | --- |
| DESENVOLVEDORES | PRODUTO |
| Apache DB Project | (ObJectRelationalBridge) |
| Applied Information Technologies, Inc. | (<SQLTags:>) |
| AlachiSoft | (TierDeveloper) |
| Art Technology Group | (ATG Repository) |
| ARTech | (DeKlarit) |
| The BBA Data Objects Project | (BBA Data Objects) |
| The Castor Project | (Castor) |
| CodeFutures | (FireStorm/DAO) |
| DADO Solution | (DADO Database Mapper) |
| Developer Express, Inc. | (eXpress Persistent Objects for .NET) |
| EntitySpaces, LLC | (EntitySpaces Architecture for .NET 2.0) |
| FireStar Software, Inc. | (ObjectSpark) |
| The GLORP Project | (Object-relational persistence for Smalltalk) |
| The Hibernate Project | (Hibernate) |
| The Hydrate Project | (Hydrate) |
| The iBATIS Project | (iBATIS Data Mapper for Java and .NET) |
| IdeaBlade | (IdeaBlade) |
| The Last Component | (Persistent Datasets for .NET) |
| Lattice Business Software International, Inc. | (Lattice.DataMapper for .NET) |
| Manifest Web Design | (DABL) |
| MetaMatrix, Inc. | (MetaBase and MetaMatrix Server) |
| Metastorage | (Metastorage - PHP object-relational mapping code generator) |
| Michael Campbell Associates, Ltd. | (JCodeBox) |
| MicroDoc Computersysteme, GmbH | (MicroDoc Persistence Frameworks for Java) |
| MyGeneration Software | (MyGeneration) |
| .NET Data Objects | (NDO) |
| The .NET Entity Objects Project | (.NET Entity Objects - NEO) |
| ObjectMapper .NET Project | (ObjectMapper .NET) |
| Object Persistent Framework for .Net 2.0 | (Opf3) |
| ObjectFrontier, Inc. | (FrontierSuite and FrontierSuite for JDO) |
| Objectmatter, Inc. | (Visual BSF) |
| ObjectStyle Group | (Cayenne Project) |
| Olero Software, Inc. | (ORM.NET) |
| OmniSphere Information Systems Corp. | (OmniBuilder) |
| OOmega.net | (OOmega JAVA Connector, XML Connector, and DBMS Connector) |
| The PerlORM Project | (PerlORM - Object relational mapping for Perl) |
| PHP Object Generator | (POG) |
| POTIS Software Development Tools | (POTIS Object-Relational Toolkit - PORT) |
| The Propel Project | (Propel: PHP Object Persistence) |
| Progress Software Corporation | (DataXtend CE) |
| RogueWave Software, Inc. | (SourcePro DB) |
| SimpleORM | (SimpleORM) |
| Software Tree, Inc. | (JDX for Java and NJDX for .NET) |
| SolarMetric, Inc. | (Kodo JDO) |
| Solutions Design | (LLBLGen Pro) |
| Sun Microsystems, Inc. | (Java Studio Enterprise) - acquired by Oracle |
| Sybase, Inc. | (PowerDesigner) |
| TechTalk, Ltd. | (Genome) |
| Thought, Inc. | (CocoBase) |
| Vanatec, GmbH | (Vanatec OpenAccess for .NET) |
| Visual Paradigm International Ltd. | (DB Visual Architect) |
| X-tensive.com Software Development Company | (DataObjects.NET) |
| Xcalia | (LiDO) |
| YTZ Systems, Inc. | (SavePoint for Java) |

# O Padrão JPA

Java Persistence API, ou simplesmente JPA, é uma API padrão do Java, sendo implementadas por frameworks que tem por objetivo seguir o padrão proposto. A proposta do Padrão JPA é definir um meio de mapeamento-relacional para objetos simples e comuns[[4]](#footnote-4). Vários frameworks voltados para desenvolvimento Java, objetos de nosso estudo, implementam o padrão JPA.

O padrão JPA também é responsável pelo gerenciamento do desenvolvimento de entidades do modelo relacional fazendo uso das plataformas nativas Java SE[[5]](#footnote-5) e Java EE[[6]](#footnote-6).

Nessa premissa, o JPA tem por principal característica, a forma simples de mapear bancos de dados relacionais, padronizando o mapeamento O/R[[7]](#footnote-7), sendo, portanto, uma solução completa para mapeamento e persistência de objetos, possuindo modo declarativo de descrever mapeamento O/R, linguagem de consulta, ferramentas para manipular entidades. Todas essas características resultam em ganho de produtividade.

## Surgimento

O padrão JPA surgiu na plataforma Java EE 5.0 (Java Enterprise Edition) na especificação do EJB 3.0 (Enterprise Java Beans 3.0). Este padrão foi introduzido para substituir os Entity Beans (que foram descontinuados) e tornar simples o desenvolvimento de aplicações JEE e JSE que utilizem persistência de dados. Possui uma completa especificação para realizar mapeamento objeto relacional, utilizando anotações da linguagem Java (JSE 5.0 ou superior). Também dá suporte a uma linguagem de consulta, semelhante à SQL, permitindo consultas estáticas ou dinâmicas.

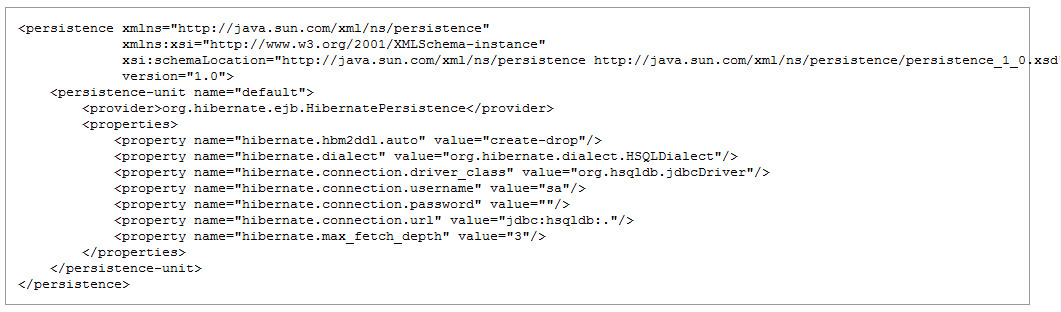
## Principais Implementações do JPA

Existem duas principais implementações do Padrão JPA, Hibernate: implementa a JPA através de três subprojetos: hibernate-core, hibernate-annotations e hibernate-entitymanager e o Oracle Toplink Essentials.

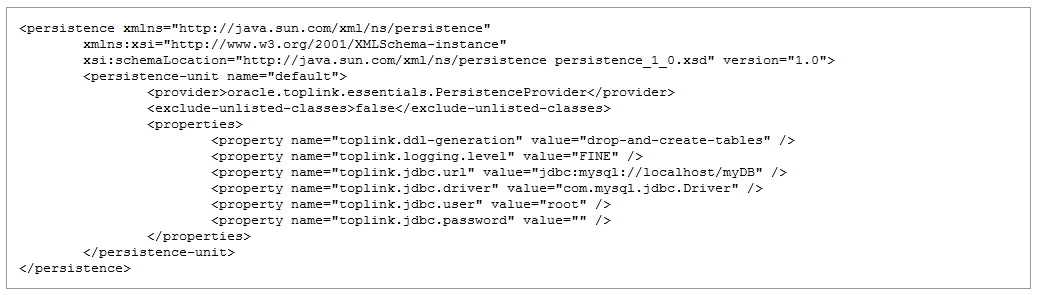
## Configurando o JPA

Primeiramente, para se iniciar a configuração do JPA, é necessário o arquivo META-INF/persistence.xml. Este, por sua vez deve estar no seu classpath. A forma mais fácil é deixar o diretório META-INF na pasta dos seus fontes e criar o arquivo persistence.xml dentro dele.

Abaixo, um exemplo de configuração para Hibernate com o banco de dados HSQLDB:



Agora, outro exemplo para o Oracle Toplink Essentials com o banco de dados MySQL:



**3 – MAPEAMENTOS UTILIZANDO JPA**

**3.1 - Annotations**

Annotations são anotações que podemos definir nas classes, métodos, interface, pacotes, etc. O JPA faz leitura do mapeamento das classes em tempo de execução para poder ler através da reflexão. Elas sempre devem ser digitadas antes do objeto que você queira anotar (método, atributo…) e o nome das anotações são sempre precedidos de @ (arroba).

**Ex:**

@MinhaAnotacao

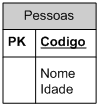
**class** MinhaClasse{

}

**3.2 - Como se faz o mapeamento de classes -> tabelas? Dê exemplos.**

Para mapear uma entidade do mundo real de um modelo Orientado a Objetos em um modelo relacional, nos criamos uma tabela com o mesmo nome da classe. Para cada campo nos podemos adicionar zero ou mais colunas na tabela com o mesmo nome e com o tipo de dados apropriado do banco de dados.

Devemos ainda introduzir um campo para usar como chave primária. Normalmente um campo do tipo inteiro, nomeado como Código ou Identificador. Isto é necessário para atender as exigências do modelo relacional e para tornar a navegação entre objetos mais simples.

http://www.fernandoamaral.com.br/Imp003.png

No exemplo acima, a classe pessoa foi mapeada para a tabela Pessoas. Cada atributo da classe receber uma coluna na tabela. Foi adicionada ainda uma coluna Código com a funcionalidade de chave da tabela.

**3.3- Como se faz o mapeamento de propriedades da classe -> campos em tabelas? Dê exemplos.**

**@Column** por padrão, a JPA assume que o nome de cada atributo corresponde ao mesmo nome na tabela, o mapeando das propriedades é feito da seguinte forma:

Ex 1: ***@Column(name=Nome\_do\_campo);***

Ex 2: ***@Column(name="MED\_ALTURA");***

Ex 3: @Entity

public class Funcionario implements Serializable {

*...*

**@Column(name="FUN\_ID")**

private Long Id;

*...*

}

**3.4- Como se faz o mapeamento de uma chave primária de uma tabela em uma classe? Dê Exemplos:**

**3.5- Coleções (collections)**

Coleções são classes e interfaces voltadas para estrutura de dados, pertencentes ao pacote java.util que representam listas, conjuntos e mapas. É amplamente utilizado em softwares que não necessitam de um vetor, conjunto ou uma lista, além de melhorar o desempenho e a qualidade elas estimulam o reuso.

**Definições**

**Coleções**

*Collections*: Interface que difinem operações comuns de coleções, esta interface possui duas sub-interfaces – Set (Conjunto) e List (Lista);

**Conjuntos:**

*Set:* Implementações de collections que modela um conjunto de elementos únicos.

*HashSet*: Implementação de Set, modela conjuntos não ordenados.

*SortedSet*: Implementação de Set, modela conjuntos ordenados.

**Listas:**

*List:* Modela lista de dados, onde os elementos estão ordenados.

*ArrayList:* Usa métodos não sincronizados e Vector utiliza métodos sincronizados (Synchronized).

*Vector*: É apropriado para o uso de em multithread, porém é mais lento que o ArrayList.

*LinkedList*: É uma lista onde os elementos ligados. Tem uma inserção e deleção muito mais rápidos que ArrayList e Vector.

**Mapas**:

*Map*: Modela Mapeamento entre as chaves não repetidas a valores.

*HashMap:* Subclasse de Map, modela mapas não classificados, com métodos não- sincronizados.

*HashTable:* Subclasse de Map, modela mapas não classificados, com métodos sincronizados.

*SortedMap:* Modelas Mapas Classificados

**Filas:**

*Queue*: É uma interface que modela filas.

**3.6- Como se faz o mapeamento de relacionamentos um-para-um entre duas classes mapeadas? Dê exemplos.**

O relacionamento um para um em Java é o lugar onde o objeto de origem tem um atributo que faz referência a outro objeto de destino e (se) que o objeto alvo tinha a relação inversa de volta para o objeto de origem também seria um relacionamento um para um. Todos os relacionamentos em Java e APP são unidirecionais, em que, se um objeto de fonte de referências um objeto de destino não há nenhuma garantia de que o objeto alvo também tem uma relação com o objeto de origem. Isso é diferente de um banco de dados relacional, em que os relacionamentos são definidos através de chaves estrangeiras e consulta de tal forma que a consulta inversa sempre existe

Ex.:

@Entity

**public** **class** Employee {

@Id

@Column(name="EMP\_ID")

**private** **long** id;

...

@OneToOne(fetch=FetchType.LAZY)

@JoinColumn(name="ADDRESS\_ID")

**private** Address address;

...

}

**3.7- Como se faz o mapeamento de relacionamentos um-para-muitos entre duas classes mapeadas? Dê exemplos:**

Um relacionamento um para muitos em Java é o lugar onde o objeto de origem tem um atributo que armazena uma coleção de objetos de destino e (se) os objetos de destino teve a relação inversa de volta para a origem do objeto que seria um relacionamento muitos para um. Todos os relacionamentos em Java e APP são unidirecionais, em que, se um objeto de fonte de referências um objeto de destino não há nenhuma garantia de que o objeto alvo também tem uma relação com o objeto de origem. Isso é diferente de um banco de dados relacional, em que os relacionamentos são definidos através de chaves estrangeiras e consulta de tal forma que a consulta inversa sempre existe.

APP também define um relacionamento muitos para muitos, que é semelhante ao relacionamento um para muitos. A principal diferença entre um relacionamento um para muitos e um relacionamento muitos para muitos em APP é que o muitos para muitos sempre faz uso de um relacionamento da tabela de junção intermediária para armazenar o relacionamento, onde, o um para muitos pode usar uma tabela de junção, ou uma chave estrangeira é objeto de tabela de referência de destino objeto principal da tabela de chave a fonte. Se o um para muitos usa uma chave estrangeira na tabela de destino do objeto da APP exige que a relação ser bi-direcional (inverso muitos para um relação deve ser definida no objeto de destino), e o objeto de origem deve usar o mappedBy atributo para definir o mapeamento.

Ex.:

@Entity

**public** **class** Employee {

@Id

@Column(name="EMP\_ID")

**private** **long** id;

...

@OneToMany(mappedBy="owner")

**private** List<Phone> phones;

...

}

**3.8- Como se faz o mapeamento de relacionamentos muitos-para-um entre duas classes mapeadas? Dê exemplos.**

Em uma APP muitos para um, a  relação é sempre (bem, quase sempre) necessários para definir um relacionamento um para um , o muitos para muitos sempre define a chave estrangeira ( JoinColumn ) e o relacionamento um para muitos deve usar um mappedBy para definir sua inversa muitos para um .

Ex.:

@Entity

**public** **class** Phone {

@Id

**private** **long** id;

...

@ManyToOne(fetch=FetchType.LAZY)

@JoinColumn(name="OWNER\_ID")

**private** Employee owner;

...

}

**3.9- Como se faz o mapeamento de relacionamentos muitos-para-muitos entre duas classes mapeadas? Dê exemplos.**

O relacionamento muitos para muitos em Java é o lugar onde o objeto de origem tem um atributo que armazena uma coleção de objetos de destino e (se) os objetos de destino teve a relação inversa de volta para o objeto de origem também seria um relacionamento muitos para muitos. Todos os relacionamentos em Java e APP são unidirecionais, em que, se um objeto de fonte de referências um objeto de destino não há nenhuma garantia de que o objeto alvo também tem uma relação com o objeto de origem. Isso é diferente de um banco de dados relacional, em que os relacionamentos são definidos através de chaves estrangeiras e consulta de tal forma que a consulta inversa sempre existe.

Ex.:

@Entity

**public** **class** Employee {

@Id

@Column(name="EMP\_ID")

**private** **long** id;

...

@ManyToMany

@JoinTable(

name="EMP\_PROJ",

joinColumns={@JoinColumn(name="EMP\_ID", referencedColumnName="EMP\_ID")},

inverseJoinColumns={@JoinColumn(name="PROJ\_ID", referencedColumnName="PROJ\_ID")})

**private** List<Project> projects;

...

}

**4- O GERENCIADOR DE ENTIDADES**

**4.1 EntityManager**

EntityManager é o serviço central para todas as ações de persistência. Entidades são objetos de Java claros que são alocados como qualquer outro objeto Java. Eles não ficam persistentes explicitamente até seu código interagir com o EntityManager para os fazer persistente. O EntityManager administra o O/R que o mapea entre uma classe de entidade e uma fonte de dados subjacente. O EntityManager provê APIs para criar consultas, buscando objetos, sincronizando objetos, e inserindo objetos no banco de dados. Também pode prover caching e pode administrar a interação entre uma entidade e serviços transacionais em um ambiente Java EE como JTA. O EntityManager é firmemente integrado com Java EE e EJB mas não é limitado a estes ambientes;  pode ser usado em programas de Java SE.

**4.1.1-Persistence Contexts**

Começaremos através de uma introdução das condições do core do JPA. Uma unidade de persistência ou persistence-unit é uma configuração nomeada de classes de entidade. Um contexto de persistência é um gerenciamento de Entidades onde, todo contexto de persistência é associado com uma unidade de persistência. Entendendo o contexto de persistência é a chave para entender o EntityManager. Uma inclusão de entidade ou exclusão de um contexto de persistência, determinarão no resultado de qualquer operações persistentes nisto.

Se o contexto de persistência participa em uma transação, que está na memória,o estado das entidades gerenciadas será sincronizado ao banco de dados. Ainda apesar do papel importante que faz, o contexto de persistência nunca é realmente visível à aplicação. Sempre é acessado indiretamente pelo entity manager e assumido que estar lá quando nós precisarmos dele.

**4.1.2- Persistence Unit**

Um EntityManager mapea um conjunto de classes a um banco de dados particular. Este conjunto de classes é chamado de *persistence unit* (unidade de persistência). Antes de você poder pensar até mesmo em criar ou consultar entidades com um entity manager, você tem que aprender a empacotar uma unidade de persistência para uso, dentro de um ambiente  Java SE ou Java EE.

Uma unidade de persistência está definida em um arquivo chamado *persistence.xml*. Este arquivo é um descriptor de desenvolvimento exigido no JPA. Um arquivo de persistence.xml pode definir um ou mais unidades de persistência. Este arquivo fica situado no diretório META-INF em: Ambientes SE, esta pasta fica no classpath da aplicação. Em uma aplicação web, este diretório fica localizado nas pasta WEB-INF\classes\META-INF

**4.1.3- EntityManagerFactory**

EntityManagers podem ser criado ou podem ser obtidos de um EntityManagerFactory. Em uma aplicação Java  SE, você tem que usar um EntityManagerFactory para criar instancias de um EntityManager. Usar o factory não é uma exigência em um ambiente Java EE.

Obtendo um EntityManagerFactory em um ambiente Java SE. Em um ambiente Java SE, a classe javax.persistence.Persistence é responsável por amarrar um EntityManagerFactory.

**Ex:**

*public class Persistence {*

*public static EntityManagerFactory createEntityManagerFactory(*

*String unitName*

*);*

*public static EntityManagerFactory createEntityManagerFactory(*

*String unitName,*

*java.util.Map*

*properties*

*);*

A classe javax.persistence.Persistence procura descriptors da distribuição  persistence.xml dentro de seu classpath de Java. O parâmetro do unitName que você passa dentro permitirá que a execução da persistencia encontre um EntityManagerFactory que combine o nome dado.

*EntityManagerFactory factory = Persistence.createEntityManagerFactory("novoExemploPU");*

*...*

*factory.close( );*

Em Java EE, obter um EntityManagerFactory. Pode ser injetado diretamente em um campo ou em um método setter do seu EJBs usando a anotação @javax.persistence.PersistenceUnit

**4.2- Consultas no JPA**

A JPA oferece suporte a consultas estáticas e dinâmicas. A API fornece uma linguagem própria de consulta, que é uma extensão bastante poderosa da EJB QL (a linguagem de consultas do EJB). Essa linguagem pode ser usada como uma alternativa ao SQL, que também é suportado. As consultas suportam polimorfismo,o que significa que quando uma entidade é consultada, todas as entidades descendentes que atendam ao critério da consulta também são retornadas. A criação de consultas é feita através do EntityManager, que fornece métodos específicos para instanciar consultas estáticas e dinâmicas, além de permitir a execução das operações CRUD.

As consultas estáticas possuem nomes e são descritas pela anotação @NamedQuery. Elas são definidas nas entidades correspondentes se ficam “pré-compiladas”. Veja um exemplo de consulta estática para localizar uma pessoa pelo seu CPF:

*@NamedQuery(name = “consultarPorCPF”,*

*query = “SELECT p FROM Pessoa p WHERE p.cpf = :cpf”)*

O EntityManager utiliza o nome da consulta para instanciá-la, o que é feito através do método createNamedQuery(). Depois que a consulta é criada, basta setar os parâmetros e executá-la. A execução pode ser feita pelos métodos getSingleResult() ou getResultList(), a depender do resultado esperado. Por exemplo, para localizar uma pessoa pelo CPF (supondo que o retorno será único), basta executar a consulta conforme o exemplo abaixo:

*Query consulta =manager.createNamedQuery(“consultarPorCPF”);*

*consulta.setParameter(“cpf”, “111.111.111-11”);*

*Pessoa pessoa = consulta.getSingleResult();*

As consultas dinâmicas não possuem nome, e podem ser construídas em tempo deexecução. A criação desse tipo de consulta é feito através do método createQuery():

*Query consulta = manager.createQuery(“SELECT p FROM Pessoa p WHERE p.cpf = :cpf”);*

**4.3- Objeto Query**

**4.4- Java Persistence Query Language - JPQL**

O JPQL é independente de plataforma orientada a objeto linguagem de consulta definida como parte da Java Persistence API especificação. JPQL é usada para fazer consultas com entidades armazenados em um banco de dados relacional. É fortemente inspirado no [SQL](http://www.google.com/url?q=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FSQL&sa=D&sntz=1&usg=AFQjCNGF16SJzeKAm5K8b0YRFhY6nvvV0Q) , e suas pesquisas se assemelham a consultas na sintaxe SQL, mas operam com objetos de entidade JPA e não directamente com as tabelas de banco de dados. Além de recuperar objetos ( SELECT consultas), o JPQL suporta a granel UPDATE e DELETE.

**4.5- Criteria**

**5- Bibliografia:**

<http://www.javasimples.com.br/mundo-java/annotations-o-que-pra-que-serve/>; acessado às 22:50, 03/03/2011.

<http://www.fernandoamaral.com.br/Default.aspx?Artigo=55>, acessado às 23:05, 03/03/2011.

<http://bibdig.poliseducacional.com.br/document/?view=18>, acessado às 23:20, 03/03/2011.

<http://fernandofranzini.wordpress.com/category/jpa/>, acessado às 23:45, 03/03/2011.

<http://www.javanoroeste.com.br/tech_days_votuporanga/palestras/Java_Persistence_API.pdf>, acessado às 23: 58, 03/03/2011.

[http://www.devmedia.com.br/post-5206-Introducao-aoEntityManager.html acessado às 15:00](http://www.devmedia.com.br/post-5206-Introducao-aoEntityManager.html%20acessado%20às%2015:00) acessado às 15:00, 04/03/2011

[http://www.guj.com.br/java/80230-jpa---entitymanagerfactory acessado às 15:20](http://www.guj.com.br/java/80230-jpa---entitymanagerfactory%20acessado%20às%2015:20) acessado as 15:20, 04/03/2011

[http://xa.yimg.com/kq/groups/24791235/236744173/name/Introducao\_JPA.pdf acessado às 16:00](http://xa.yimg.com/kq/groups/24791235/236744173/name/Introducao_JPA.pdf%20acessado%20às%2016:00), 04/03/2011

<http://wiki.sintectus.com/bin/view/GrupoJava/SlidesIntroducaoAsColecoes?skin=slidy> acessado às 16:20, 04/03/2011

<http://www.marcomendes.com/ArquivosBlog/AloMundoJPA.pdf> acessado às 20:00, 04/03/2011

<http://wiki.sintectus.com/bin/view/GrupoJava/SlidesAssociacoesComJPA?skin=slidy#(3)> acessado 20:30, 04/03/2011

1. Refere-se à inerente diferença que existe entre os modelos de dados relacionais e os orientados a objeto. [↑](#footnote-ref-1)
2. Na linguagem Java existe vários frameworks, como o Hibernate, e o padrão, JPA. [↑](#footnote-ref-2)
3. É um framework para o mapeamento objeto-relacional escrito na linguagem Java, também disponível em .Net. [↑](#footnote-ref-3)
4. Os chamados *POJOs*, beans de entidades. [↑](#footnote-ref-4)
5. Java Standard Edition. [↑](#footnote-ref-5)
6. Java Enterprise Edition. [↑](#footnote-ref-6)
7. Objeto/Relacionamento. [↑](#footnote-ref-7)