# Tópicos Especiais em Análise e Desenvolvimento de Sistemas

Prof. Rafael Odon (rafael.alencar@prof.una.br)

**ORIENTAÇÕES PARA ENTREGA:**

- O roteiro deverá ser feito continuando **com a mesma dupla** do Roteiro 01 ou **individual** para quem decidiu não fazer em dupla.
- Enviar até o início da aula do dia do dia 14/03/2014 o código e as respostas de **todas as perguntas** desse roteiro para o e-mail do professor (rafael.alencar@prof.una.br) com o título:
  - `Roteiro JPA 02 – Nome do 1º Aluno / Nome do 2º Aluno`
- Lembre-se: **todas as informações** para responder as perguntas são dadas em **sala de aula**. Preste atenção na aula e faça notas pessoais!
- As **respostas** devem ser pessoais e **escritas com suas palavras**. Não serão aceitas respostas longas e/ou **copiadas** de forma literal do material de referência ou Internet.

# Java Persistence API & Hibernate

## Roteiro de Aula 02 – 28/02/2014

Esse Roteiro se baseia no uso dos softwares:

- Netbeans 7.4.1 Java EE
- Java DB (Apache Derby)
- Java Development Kit (JDK) 1.7

### Fase 1: Configurando o JPA na aplicação

No Roteiro de Aula 01, aprendemos como criar uma Unidade de Persitência. Para o Roteiro atual, devemos repetir esses passos de modo a obter novamente um projeto Java no Netbeans e uma Unidade de Persistência que conecte no banco de dados. **Recorra ao Roteiro 01 se necessário.**

1. Crie um projeto Java no Netbeans chamado *SysFrota*.
2. Crie um banco de dados no Apache Derby Java DB chamado *db_sysfrota*
3. Crie o arquivo META-INF/persistence.xml no seu projeto
4. Inclua uma Unidade de Persistência no persistence.xml para conectar a aplicação ao banco de dados *db_sysfrota.*
5. Configue sua Unidade de Persistência para **mostrar SQL** no console e para **apagar e criar o banco de dados** sempre que executarmos a aplicação.

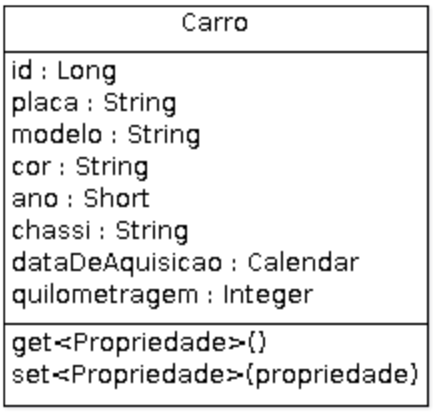
```
...
<persistence-unit name="db_sysfrota_01" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <properties>
    ...
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    ...
  </properties>
</persistence-unit>
...
```

6. Não se esqueça dos outros detalhes do roteiro 01, como por exemplo certificar-se de que as **bibliotecas JPA do Hibernate** e o **Driver JDBC do Apache Derby Java DB** foram adicionados ao seu projeto.

## Fase 2: Anotando JavaBeans para descrever mapeamentos para Tabelas

No Roteiro de Aula 01, modelamos uma classe Carro. Na  mesma linha, vamos continuar modelando a classe Carro. Para isso vamos usar anotações do pacote *javax.persistence* para marcar propriedades de nossos JavaBeans informando que elas possuem algum significado especial.

1. De acordo com o Diagrama de Classes UML abaixo, crie uma classe Java chamada Carro no pacote *sysfrota.entidades* e inclua nela as propriedades necessárias, bem como os métodos get e set de modo a compor o JavaBean.




2. Anote a classe Carro com @Entity para indicar que esse objeto também se trata de uma entidade de um Banco de Dados Relacional. Classes de entidade também devem ser serializáveis, ou seja, implementar a interface *java.io.Serializable.*

3. Anote a propriedade **id** com @Id e @GeneratedValue para indicar que ela é a chave primária (identificador único) da entidade, bem como é um valor auto-incrementado (no SQL Server equivale a uma coluna IDENTITY)

4. Anote a propriedade **dataDeAquisicao** com @Temporal(TemporalType.DATE) para indicar que no banco de dados essa coluna guardará informação de tempo, mas que porém, apenas a porção mês, dia e ano será usada, ignorando a horas, minutos, segundos e milisegundos.

5. Anote a propriedade **placa** com @Column(**length = 8**, **nullable = false**) para indicar que no banco de dados essa coluna deverá ter **tamanho de 8 caracteres** e que **não pode ser nula (NOT NULL)**.

6. Construa a classe Bootstrap no pacote *sysfrota.util* com método *main* abaixo. O papel dessa classe futuramente será executar uma carga inicial no Banco de Dados Relacional criando alguns dados. Por agora, usaremos ela para testar a evolução de nossa aplicação.

```
...
public static void main(String [] args) throws Exception{

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("nome da sua Unidade de Persistência");
    EntityManager em = emf.createEntityManager();

    em.getTransaction().begin();


    //Cria um Uno
```

```
        Carro uno = new Carro();
        uno.setPlaca("ABC-0001");
        uno.setModelo("Fiat Uno 2011");
        uno.setCor("Prata");
        uno.setChassi("A1B2C3456789DEF");
        uno.setAno((short) 2010);

        uno.setDataDeAquisicao(Calendar.getInstance());
        uno.setQuilometragem(200000);
        em.persist(uno);

        em.getTransaction().commit();

        System.out.println("Pressione uma tecla para continuar...");
        System.in.read();


        em.close();
        emf.close();
    }
    ...
```

7. Lembre-se de adaptar as partes necessárias, tais como o **nome da sua Unidade de Persistência** na criação do Entity Manager.
8. Importe corretamente as classes do pacote *javax.persistence*.
9. Execute a classe Bootstrap e antes de pressionar alguma tecla, verifique no Banco de Dados a Tabela Criada bem como se o carro foi inserido lá corretamente.

### Fase 3: Brincando com as *Annotations*

É hora de investigar a API e pesquisar no material didático e na Internet para se virar sozinho com as anotações! Vamos propor alguns desafios:

1. Faça a coluna *chassi* ter 20 caracteres e ser não nula;
2. Faça a coluna *cor* ter 30 caracteres;
3. Faça a coluna *modelo* e ter 60 caracteres e ser não nula;
4. Pesquise e descubra como fazer para que a tabela mapeada pela entidade Carro no Banco de Dados tenha o nome *veiculo*
5. Pesquise e descubra como fazer para que a coluna *ano* no Banco de Dados tenha o nome *ano_de_fabricacao;*
6. Modifique a classe Bootstrap para criar mais dois carros, com dados diferentes. Execute-a e observe se as configurações feitas via *annotations* foram aplicadas na definição das tabelas.

**RESPONDA:**

I. O que é uma Anotação ou *Annotation*? Quais as vantagens de usar *Annotations* no Mapeamento Objeto Relacional?

II. Sobre as anotações que você conhece:
a) Qual delas indica que uma propriedade é Chave Primária?
b) Qual delas indica que o valor é auto-incrementado?
c) Qual delas indica que uma propriedade é do tipo Data/Tempo? Que opções a enumeração *javax.persistence.TemporalType* fornece para indicar a precisão da data? Exemplifique dados temporais que poderiam usar cada dessas opções.
d) Qual delas permite parametrizar a coluna mapeada na propriedade? Dê um exemplo.
e) Qual delas permite parametrizar a tabela mapeada pela classe? Dê um exemplo.

III. Qual o código de uma classe Java anotada para ser mapeada para a tabela abaixo? Para economizar espaço na resposta, não inclua *getters* e *setters* da classe, apenas suas propriedades.

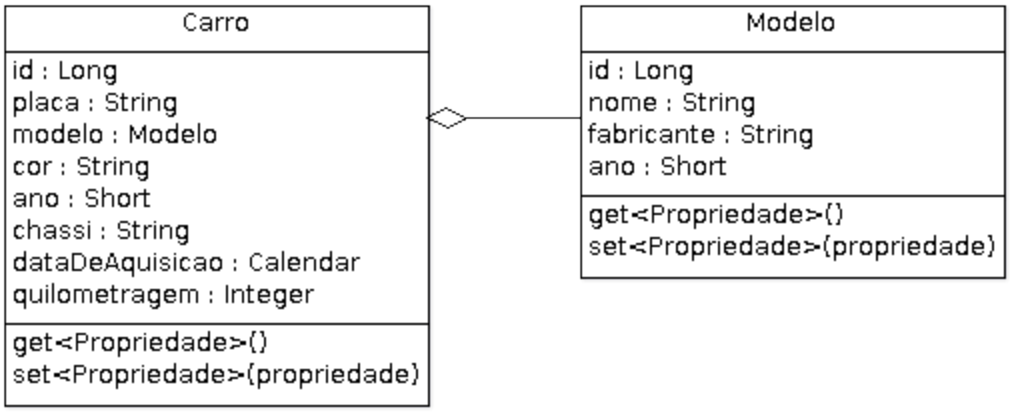
```
CREATE TABLE modelo_de_veiculo(
   id_modelo bigint IDENTITY PRIMARY KEY,
   descricao VARCHAR(80) NOT NULL
);
```

IV. Escreva a DDL (comando *create table*) da tabela obtida no final da Fase 3 desse roteiro.

## Fase 4: *Mapeamentos Muitos-Para-Um*

Já aprendemos como anotar uma classe para que suas instâncias sejam persistidas em tabelas. Vamos agora fazer com que outras classes existam no nosso diagrama, e vamos refletir também os relacionamentos entre as classes no Banco de Dados:

1. Crie no pacote *sysfrota.entidades* o JavaBean da classe *Modelo* de acordo com sua descrição no diagrama abaixo, e faça-o ser uma entidade.




2. Evolua a classe *Carro* para que, ao invés de ter uma propriedade **modelo** do tipo *String*, passe a ter uma propriedade **modelo** do tipo *Modelo*. Em outras palavras: existirá uma agregação de *Carro* com a classe *Modelo*. Não se esqueça de atualizar os respectivos métodos get e o set para utilizar o novo tipo.

3. Anote a nova propriedade **modelo** de carro com ***@ManyToOne*** para indicar que se trata de um relacionamento Muitos-para-um (N..1). Isso significa que muitos carros podem estar relacionados com um determinado modelo.

```
...
public class Carro implements Serializable{
    ...
    @ManyToOne
    Modelo modelo;
    ...
}
```

4. Evolua a classe *Bootstrap* para exemplificar esse relacionamento, **criando modelos** de carros e **relacionando-os com os carros** já criados. Veja o exemplo abaixo:

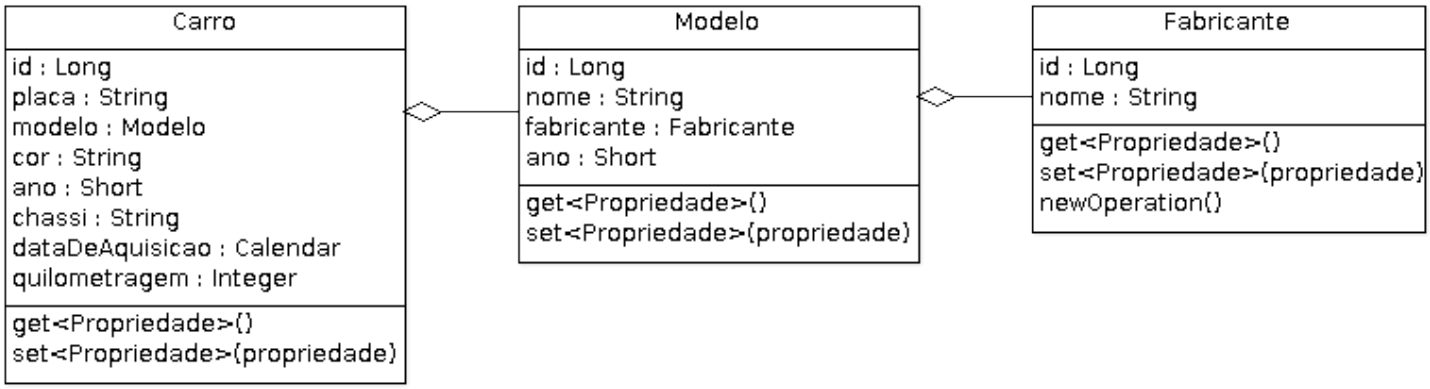
```
...
public static void main(String [] args) throws Exception{
    ...
    //Cria o Modelo Fiat Uno
    Modelo fiatUno = new Modelo();
    fiatUno.setNome("Uno");
```

```
        fiatUno.setFabricante("Fiat");
        fiatUno.setAno((short) 2011);
        em.persist(fiatUno);

        //Cria um Uno
        Carro meuUno = new Carro();
        ...
        meuUno.setModelo(fiatUno);
        ...
    }
    ...
```

5. Execute a classe *Bootstrap* e antes de pressionar alguma tecla observe se a nova Entidade *Modelo* foi refletida no banco de dados.

6. Continue evoluindo seu sistema para agora contemplar também a classe *Fabricante.* Observe *Modelo* deixa de ter uma propriedade *String* e passa a ter uma propriedade *Fabricante,* e o relacionamento entre as classes é o mesmo que foi feito anteriormente entre *Carro* e *Modelo.*




5. Não se equeça que todas essas classes nova são também entidades! Anote-as corretamente e observe outros detalhes necessários.

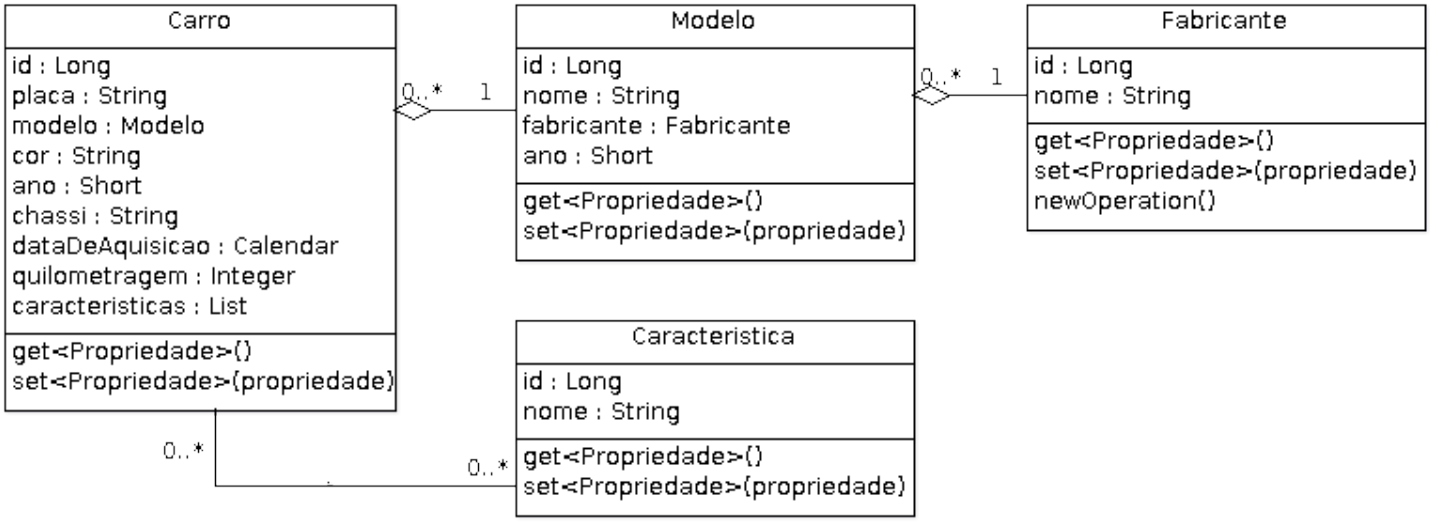
6. Evolua a classe *Bootstrap* para exemplicar o novo relacionamento, **criando fabricantes, relacionando-os com modelos,** que por sua vez **se relacionam com carros.**

```
    ...
    public static void main(String [] args) throws Exception{
        ...
        //Fabricante Fiat
        Fabricante fiat = new Fabricante();
        fiat.setNome("Fiat");
        em.persist(fiat);
        ...
        //Cria o Modelo Uno relacionado a Fiat
        Modelo fiatUno = new Modelo();
        ...
        fiatUno.setFabricante(fiat);
        ...
        //Cria um Uno
        Carro meuUno = new Carro();
        ...
        meuUno.setModelo(fiatUno);
        ...
    }
    ...
```

## Fase 5: *Mapeamentos Muitos-Para-Muitos*

Temos também a opção do relacionamento Muitos-Para-Muitos. Vamos explorá-lo!

1. Crie o JavaBean de *Caracteristica* de acordo com o diagrama abaixo. Faça-a ser uma entidade.




2. Observe que o Carro detém uma lista de caracteristicas, representada por uma propriedade **caracteristicas** que pode ser do tipo java.util.List<Caracteristica>. Nesse caso, muitos carros poderão se relacionar com muitas características. Adicione essa propriedade, e não se esqueça do get e do set respectivo.

3. Anote a nova propriedade **caracteristicas** de carro com ***@ManyToMany*** para indicar que se trata de um relacionamento Muitos-para-Muitos (N...N).

```
...
public class Carro implements Serializable{
    ...
    @ManyToMany
    List<Caracteristica> caracteristicas
    ...
}
```

4. Evolua a classe *Bootstrap* para exemplicar o novo relacionamento, **criando características**, e **relacionando-as com os carros.**

```
...
public static void main(String [] args) throws Exception{
    ...
    //cria Características
    Caracteristica quatroPortas = new Caracteristica();
    quatroPortas.setNome("Quatro portas");
    em.persist(quatroPortas);

    Caracteristica ar = new Caracteristica();
    ar.setNome("Ar condicionado");
    em.persist(ar);

    Caracteristica direcao = new Caracteristica();
    direcao.setNome("Direção Hidráulica");
    em.persist(direcao);
    ...
    //Cria um Uno
    Carro meuUno = new Carro();
    ...
    meuUno.setCaracteristicas(new LinkedList<Caracteristica>());
    meuUno.getCaracteristicas().add(quatroPortas);
    meuUno.getCaracteristicas().add(ar);
    ...
    em.persist(meuUno);
}
...
```

**RESPONDA:**

I. Que anotação é utilizada para representar o mapeamento Muitos-Para-Um?
II. Qual seria o relacionamento contrário ao Muitos-Para-Um? Na prática, o que ele significa? Que tipo de propriedade pode ser mapeada com ele?
III. Que anotação é utilizada para representar o mapeamento Muitos-Para-Muitos?
IV. Que outra forma existe de mapear a relação Muitos-Para-Muitos?
V. Pesquise e descubra sobre os tipos de carregamento de entidades relacionadas: FetchType.**Lazy** e FetchType.**Eager**. O que cada um deles significa? Onde eles podem ser configurados?
VI. Em que casos seria interessante um relacionamento Um-Para-Um? Pesquisa e descubra qual anotação pode ser utilizada para denotar esse relacionamento.