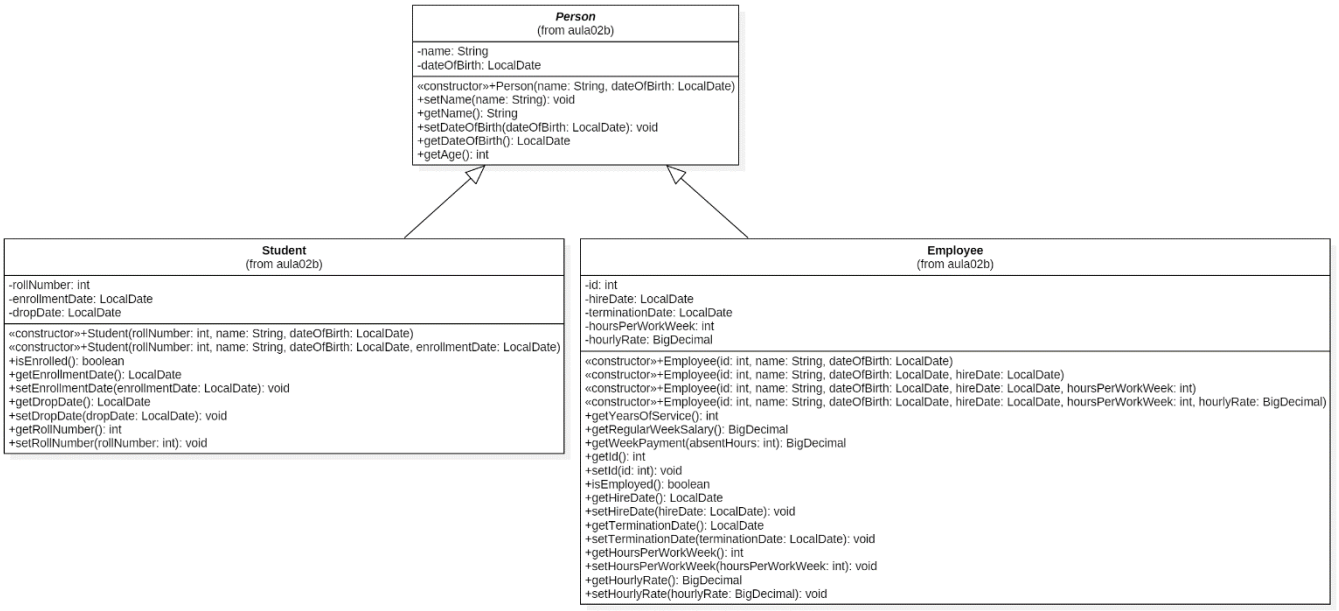
Roteiro 03 – Atividade 01/03

1. Se você fez todas as atividades até aqui, seu diagrama de classe deve estar assim:



2. Nesta atividade, incluiremos mais uma classe em nossa modelagem: **Professor**. Dentro do contexto de um sistema acadêmico, podemos considerar que um professor é um empregado que possui algumas informações adicionais. Por exemplo, uma instituição de ensino tem interesse em controlar a titulação atual de seus professores. Titulação acadêmica ou escolaridade é o nível de educação concluído por um profissional ou estudante. No nosso cenário, vamos considerar que um professor pode possuir uma das seguintes titulações: bacharelado (*bachelor*), mestrado (master) ou doutorado (*doctorate*).

   Como parte da solução, criaremos um novo atributo: `int academicDegree`. Também assumiremos valores arbitrários para cada titulação (0 - bacharelado; 1 - mestrado; e 2 - doutorado). Porém, utilizar valores numéricos no código pode dificultar sua legibilidade. Por exemplo, para analisar a expressão "`if (academicDegree == 1) ...`", o programador precisaria lembrar que 1 significa mestrado. Uma alternativa é utilizarmos constantes. Desta forma, a expressão anterior ficaria "`if (academicDegree == MESTRADO) ...`", aumentando a legibilidade.

   Entretanto, a linguagem Java não possui o conceito explícito de constante. Para contornarmos o problema, utilizaremos um artifício. Criaremos atributos públicos com valores já atribuídos e utilizaremos dois modificadores (`static` e `final`) para garantir que estes valores não poderão ser modificados depois. Desta forma, não estaremos quebrando nosso princípio de proteção aos dados:

```
/**
 * Titulação de um professor: bacharelado.
 */
public static final int BACHELOR = 0;
/**
 * Titulação de um professor: mestrado.
 */
public static final int MASTER = 1;
/**
 * Titulação de um professor: doutorado.
 */
public static final int DOCTORATE = 2;
```

   Utilizamos caixa alta para constantes como uma boa prática, pois facilita a legibilidade do código. Quando houver palavras combinadas, a separação deve ser feita com o caractere *underscore* (ex: `MINHA_CONSTANTE`). O modificador Java `static` (na UML, é representada pela propriedade `isStatic`) é utilizado para definir atributos de classe (também chamados atributos estáticos). Estes atributos pertencem a classe, podendo ser compartilhados por todos os objetos. Desta forma, não é necessário instanciar um objeto para ter acesso a eles. Depois de implementar esta classe, você pode tentar algo como:

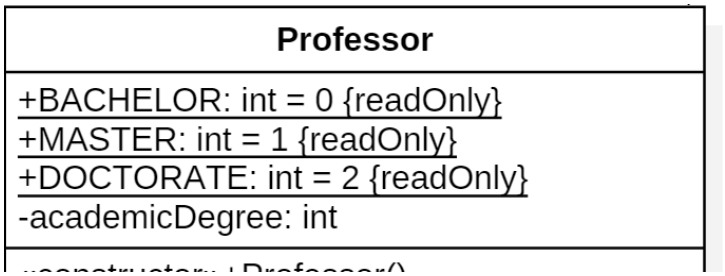
```
System.out.println(Professor.MASTER);
```

Roteiro 03 – Atividade 01/03

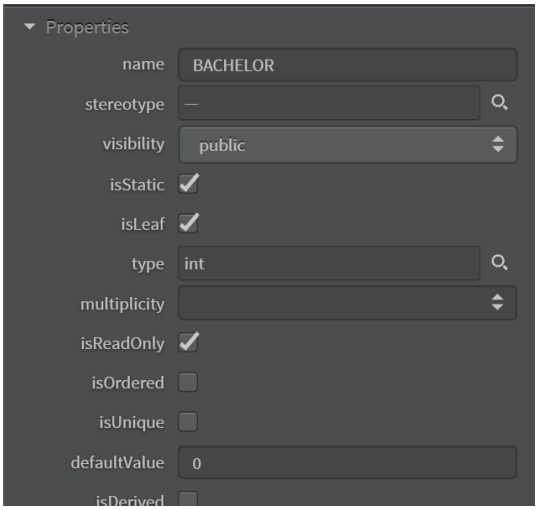
Note que você está acessando a constante diretamente da classe e não de um objeto. Quando um atributo de classe não é definido como final, qualquer modificação de seu valor se reflete para todos os objetos instanciados da classe.

O modificador `final` (na UML, `isLeaf` + `isReadOnly`) indica que o atributo não poderá mais ter seu valor alterado (`isReadOnly`) e também que não poderá ser redefinido (*override*) em alguma subclasse (`isLeaf`). Para ser útil, a constante precisa ser pública.

3. Modele a classe Professor. Na figura abaixo, você pode ver o resultado na nossa ferramenta CASE. Ao marcar um atributo como estático, ele fica sublinhado. Esta é a notação UML para atributos de classe. Quando você marca o atributo como `isReadOnly`, a UML utiliza a notação `{readOnly}`. Note ainda que você precisa informar o valor default, conforme o que foi discutido até agora.



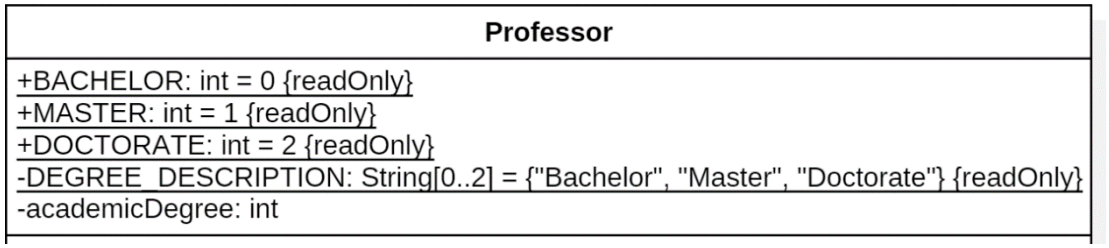
...



4. Considere também o padrão **Telescoping**. Porém, faremos uma otimização aqui. Considere que todos os atributos são obrigatórios na inicialização (o atributo `terminationDate` não deve ser informado, assim como em `Employee`).

5. Implemente a classe `Professor` no seu projeto e faça alguns testes básicos para confirmar que tudo está funcionando como esperado. Lembre-se que para criar um objeto Professor, você precisará passar a titulação como um dos parâmetros (ex: `Professor.MASTER`).

6. Mas e se precisarmos imprimir a descrição da titulação de um professor? Não faz muito sentido que um programa qualquer utilize "ifs" aninhados cada vez que ele quiser mostrar a titulação. Ok, podemos resolver isso modelando uma operação para isso (ex: `getAcademicDegreeDescription`). Porém, em qual classe devemos colocar esta operação? Se você não pensou imediatamente em Professor, revise seus conceitos de abstração. Além disso, podemos criar um vetor já inicializado com as descrições. Na UML, ficaria assim:



Para representar que queremos um vetor de 3 posições, iniciando em 0, utilizamos a propriedade *Multiplicity* (0..2). Note que após gerar o código, você precisará realizar um pequeno ajuste.

7. Implemente a alteração, crie novos testes e veja se a operação abaixo está retornando os valores corretamente.

```
System.out.printf(professor.getAcademicDegreeDescription());
```

Roteiro 03 – Atividade 01/03

8. Continuando nosso cenário, vamos considerar que professores recebem um bônus ao valor contratado da hora de acordo com a sua titulação. A **regra de negócio** (*business rule*) para o nosso sistema acadêmico é que o valor da hora de um professor tem 15% de acréscimo quando ele tem mestrado e 30% quando ele tem doutorado. Uma regra de negócio define ou restringe algum aspecto do negócio. Elas podem descrever políticas ou procedimentos que devem ser seguidos para que o negócio funcione de forma adequada. No nosso caso, o negócio é definido pelo sistema acadêmico. Poderíamos ter regras de negócio definidas para explicar: a) como calcular a média de um aluno; b) regras e condições para trancamento; c) regras e condições para justificar uma falta; d) como calcular a multa de um empréstimo na biblioteca; etc.

   Voltando ao nosso problema, precisamos decidir como resolvê-lo. Inicialmente, vamos calcular o bônus referente à titulação. Para isso, modelaremos uma nova operação na classe Professor: **+ getAcademicBonus(): BigDecimal**. Atualize o diagrama de classe. Depois, implemente esta operação (crie o método em Java):

```java
/**
 * Calcula o valor (em R$) do bônus do professor de acordo com sua
 * titulação atual: 15% para mestrado e 30% para doutorado. Note que
 * o valor ainda deve ser acrescentado ao valor da hora do professor.
 *
 * @return o valor da bônus (em reais) que deve ser acrescentado ao valor
 *          da hora, conforme titulação do professor
 */
public BigDecimal getAcademicBonus() {
    BigDecimal bonus = new BigDecimal("0.0").setScale(2, BigDecimal.ROUND_HALF_EVEN);
    if (academicDegree == MASTER) {
        bonus = super.getHourlyRate().multiply(new BigDecimal("0.15"));
    } else if (academicDegree == DOCTORATE) {
        bonus = super.getHourlyRate().multiply(new BigDecimal("0.30"));
    }
    //bônus já vem com a escala e arredondamento
    return bonus;
}
```

   O algoritmo em si é trivial, mas vamos focar em alguns aspectos de OO e Java aqui. Inicialmente, note que a variável local `bonus` foi inicializada com zero. Logo em seguida, a escala e o tipo de arredondamento foram também alterados. Isso garante que, caso o professor não tenha mestrado ou doutorado, o valor do bônus retorne com o formato financeiro definido em nosso estudo de caso. Outro ponto a ser observado é o uso da palavra reservada `super`. Esta é a forma definida pelo Java para termos acesso à versão herdada do método (`super` de superclasse). Mas, porque precisamos chamar explicitamente a versão herdada? Pela ideia de herança, poderíamos chamar diretamente o método `getHourlyRate()`. Teste o programa com cada uma das versões e compare os resultados.

9. Na sua versão original, implementada na classe `Employee`, o método `getHourlyRate()` simplesmente retornava o valor do respectivo atributo. Entretanto, se o empregado for um professor, precisamos aplicar a regra definida anteriormente. Para tal, iremos redefinir este método com o objetivo de adicionar o bônus. Para redefinir ou sobrescrever um método em Java, utilizamos a *annotation* `@Override` antes da assinatura da operação. Em Java, todas os métodos são **virtuais** por default. Quando um método é **virtual**, ele pode ser redefinido. Quando você desejar que um método não seja virtual em Java, você precisa explicitamente adicionar o modificador `final`. Ajuste sua modelagem UML e complete a implementação abaixo, conforme a lógica da regra de negócio definida.

Roteiro 03 – Atividade 01/03

```
/**
 * {@inheritDoc} Ajusta o valor da hora do professor de acordo com sua
 * titulação.
 *
 * @return o valor da hora (em reais), conforme titulação do professor
 */
@Override
public BigDecimal getHourlyRate() {
    //...
```

Se você não entendeu o que a *annotation* javadoc `{@inheritDoc}` faz, leia novamente o material disponibilizado no Roteiro 01. Lembre-se também que objetos BigDecimal são imutáveis. Em tempo, o que acontece agora se não utilizarmos a palavra reservada `super` na operação `getAcademicDegreeBonus()`?

10. Monte um programa de teste para criar uma lista de empregados (sejam empregados ou professores). Depois, percorra esta lista e imprima o salário de cada um. Você já tem a base para este teste em implementações anteriores.

Polimorfismo

Princípio pelo qual entidades de tipos diferentes podem ser acessadas através de uma única interface[1]. Você já fez isso antes, quando utilizou a interface da classe `Person` para acessar objetos `Student` e `Employee`.

Conectar a chamada de uma operação com o método que irá implementá-la é um processo chamado ligação (*binding* em Inglês). Quando esta ligação é realizada durante a compilação ou interpretação do programa, a linguagem utiliza o mecanismo de **ligação prematura** (*early binding*) ou **ligação estática** (*static binding*). Neste caso, o compilador ou interpretador já gera o código de máquina sabendo o endereço do método a ser invocado. Este é o único mecanismo adotado por linguagens procedurais como C e Pascal. Métodos estáticos (`static`), privados (`private`) ou que não podem ser redefinidos (`final` em Java) utilizam este tipo de mecanismo para serem invocados. Em Java um método privado é internamente definido como `final`.

Linguagens de programação orientadas a objetos implementam **polimorfismo de inclusão (ou de subtipo)** para permitir que o comportamento de uma subclasse (inclusão de uma nova subclasse/subtipo) possa ser estendido através da herança. Os métodos herdados (que não tenham sido explicitamente definidos como `final`) podem então ser redefinidos. Desta forma, classes de uma mesma hierarquia de herança (ou generalização) compartilharão uma mesma interface (conjunto de operações públicas). Ao declararmos um objeto `Person`, ele pode ser atribuído a qualquer subclasse de `Person`. Podemos então invocar quaisquer métodos públicos definidos em `Person`, mas estaremos solicitando para o objeto instanciado, seja ele, por exemplo, um `Student` ou um `Employee`. O problema é que os métodos (implementação) compartilham o mesmo nome (operação), comprometendo a capacidade do compilador ou interpretador identificar antecipadamente o endereço a ser invocado. O mecanismo de identificar qual método deve ser invocado é feito então em tempo de execução, sendo chamado **ligação tardia** (ligação dinâmica, ligação em tempo de execução, *late binding*, *dynamic binding* ou *runtime binding*).

Tipicamente, cada classe contém uma estrutura de dados interna chamada **tabela virtual**, a qual contém ponteiros (endereços) para os métodos virtuais implementados. Cada objeto contém um ponteiro para a tabela virtual de sua classe, a qual é então consultada sempre que um método polimórfico (virtual) é chamado. Na nossa implementação da lista de `Person`, o compilador não tem como saber qual objeto (`Employee` ou `Student`) é apontado pela referência na posição 0. Em tempo de execução, o objeto procura qual o método deve ser invocado de acordo com sua classe.

Embora chamadas que utilizam ligação prematura tenham desempenho levemente superior àquelas que utilizam ligação tardia, a utilização dos modificadores `static` e `final` deve ser tipicamente uma decisão de projeto (design, solução técnica) e não uma decisão default para melhorar o desempenho geral do sistema.

[1] http://www.stroustrup.com/glossary.html#Gpolymorphism

Roteiro 03 – Atividade 02/03

1. A implementação da classe Professor tem um problema de projeto relacionado com o tratamento dado para o atributo `academicDegree`. Embora nossa solução esteja funcional, teremos que ter controles adicionais para garantir que este atributo receba um valor válido (0, 1 ou 2). Além disso, outro controle será necessário para garantirmos a consistência com a sua descrição (`DEGREE_DESCRIPTION`). Desta forma, faremos uma melhoria em nossa classe com a utilização de uma enumeração.

   Uma **enumeração** é uma lista fechada de itens fortemente relacionados entre si. Por ser uma lista fechada, após definida, os valores não podem ser modificados. Alguns exemplos:

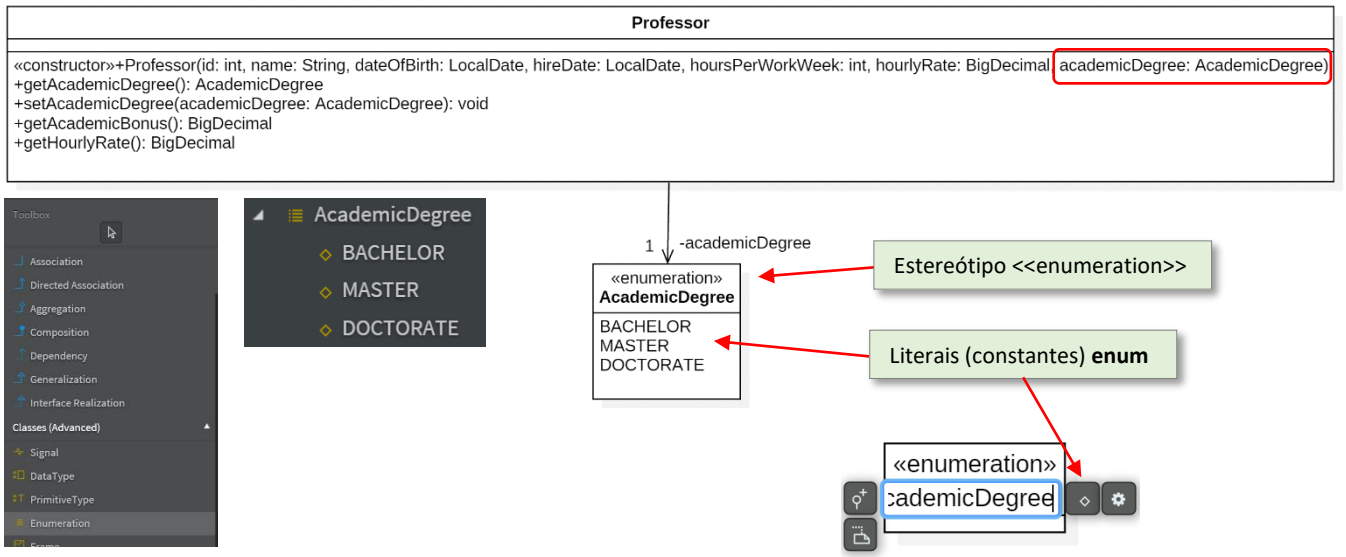
   - NORTE, SUL, LESTE, OESTE
   - SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO, DOMINGO
   - BACHARELADO, MESTRADO, DOUTORADO
   - ESPADAS, OUROS, PAUS, COPAS
   - ALTO, MEDIO, BAIXO
   - REGIONAL, NACIONAL, INTERNACIONAL
   - SENIOR, PLENO, JUNIOR,

   Na linguagem Java, uma enumeração é representada pelo tipo de dados **enum**, o qual é um tipo especial de classe. Por ser um tipo, é impossível atribuir um valor inválido para uma variável enum (***type-safe***). Por exemplo, se você tentar atribuir uma titulação que não esteja entre os itens BACHARELADO, MESTRADO ou DOUTORADO, haverá um erro de compilação ou em tempo de execução (dependendo da forma que a atribuição foi feita). Deste modo, enumerações são uma alternativa robusta ao uso de *strings* ou constantes numéricas. A representação mais simples de uma enumeração em Java é mostrada abaixo.

```java
/**
 * Titulação acadêmica
 *
 * @author Marcello Thiry
 */
public enum AcademicDegree {

    BACHELOR, MASTER, DOCTORATE

}
```

```java
public class Professor extends Employee {

    /**
     * Titulação de um professor.
     */
    private AcademicDegree academicDegree;
```

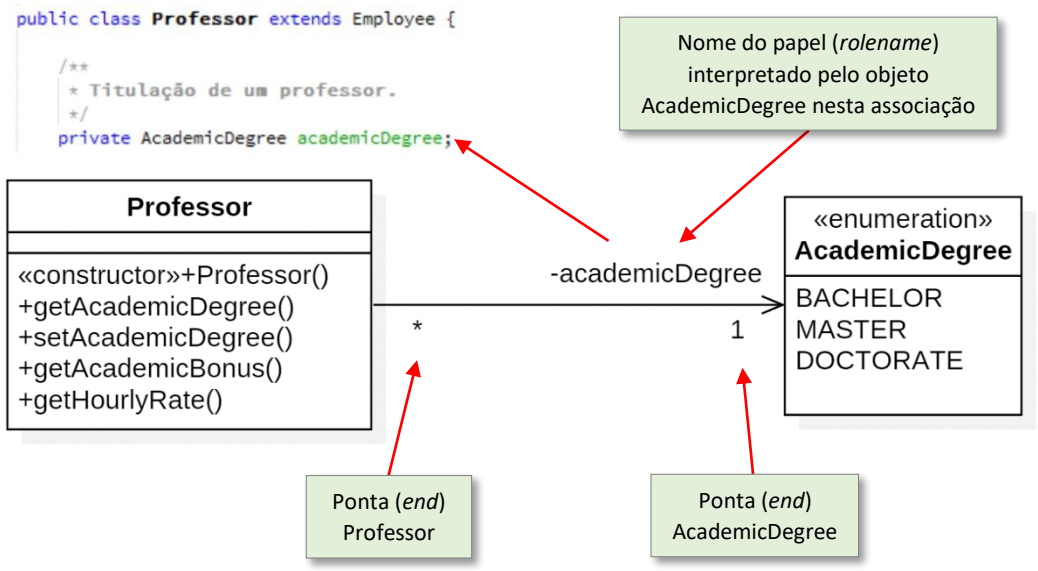
No nosso exemplo, foi definida uma lista de três itens (chamadas constantes Enum). Na classe Professor, o atributo `academicDegree` pode assumir um e somente um destes valores por vez. Uma vez que os valores são constantes, utilizamos a notação caixa alta (mesma utilizada quando declaramos constantes com `static` e `final`). A quantidade de itens disponíveis é fixada na definição do **enum** e não pode ser alterada durante a execução do programa. Portanto, o tipo **enum** é adequado quando temos uma lista fechada de valores. A notação UML para um enum é apresentada na imagem abaixo.

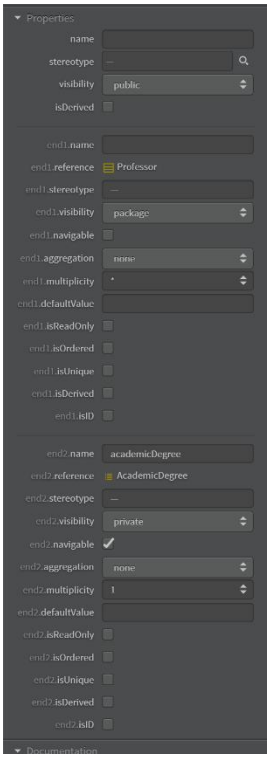


Roteiro 03 – Atividade 02/03

2. Aproveitando o diagrama de classe apresentado, podemos rever o conceito do relacionamento de associação. **Associação** é um relacionamento entre classes, utilizado para mostrar quais **instâncias** de destas classes poderiam estar ligadas entre si. A partir do diagrama (abaixo, as assinaturas das operações de `Professor` foram escondidas para facilitar a leitura), podemos afirmar que:

   - **UM objeto** `Professor` está associado com **UM item** (constante enum) de `AcademicDegree`
     - **Multiplicidade** da ponta (*end* em Inglês) `AcademicDegree`: **[1]**. A multiplicidade é a definição de cardinalidade (número de elementos) de alguma coleção de elementos por meio de um intervalo (inteiros não negativos) para especificar o número permitido de instâncias do elemento descrito. Aqui, estamos dizendo que apenas UM item enum é permitido.
     - **Navegável** para esta ponta (`Professor` para `AcademicDegree`). A direção da associação mostra isso visualmente (seta aberta).
     - Para ser possível navegar de `Professor` para `AcademicDegree`, é necessário ter uma referência em `Professor` que aponte para o item `AcademicDegree`. Esta referência é o atributo privado `academicDegree`, também representado no diagrama (nome de papel ou *rolename*). Notem que o atributo não deve ser repetido na notação da classe `Professor`.
   - **UM item** `AcademicDegree` pode estar ligado a **VÁRIOS (0, 1 ou mais) objetos** `Professor`
     - **Multiplicidade** da ponta `Professor`: **[*]**. Ou seja, 0, 1 ou mais objetos Professor podem referenciar um mesmo item enum.
     - **Não navegável** para esta ponta (`AcademicDegree` para `Professor`). Notem que a direção da associação mostra isso visualmente (sem seta).





3. Atualize seu diagrama e faça as alterações necessárias no programa. Serão necessários alguns pequenos ajustes na sua classe de teste. Para mostrar o uso, veja como seria o uso do enum no programa.

```
if (academicDegree == AcademicDegree.MASTER) {
    bonus = super.getHourlyRate().multiply(new BigDecimal("0.15"));
} else if (academicDegree == AcademicDegree.DOCTORATE) {
    bonus = super.getHourlyRate().multiply(new BigDecimal("0.30"));
}
...
System.out.println(AcademicDegree.MASTER);
```

Roteiro 03 – Atividade 02/03

4. Até o momento, utilizamos a forma mais simples de enum em Java. Entretanto, a linguagem Java permite que sejam definidos atributos e operações comuns a cada um dos itens enum. Por exemplo, o valor do bônus utilizado no cálculo da hora de trabalho de um professor é dependente da titulação que ele tem. Ou seja, estes valores são fortemente relacionados com os itens enum. Logo seria interessante perguntar a um item enum diretamente qual o valor do bônus que devemos utilizar. Além disso, atualmente, o valor impresso de um item enum será exatamente o identificador utilizado para ele. Poderíamos ter uma descrição alternativa para cada item.

```
/**
 * Titulação acadêmica
 *
 * @author Marcello Thiry
 */
public enum AcademicDegree {

    BACHELOR ("Bachelor", "0.00"),
    MASTER ("Master", "0.15"),
    DOCTORATE ("Doctorate (PhD)", "0.30");

    /**
     * Texto que descreve cada item enum.
     */
    private String description;
    /**
     * Valor do bônus em formato string para ser
     * utilizado em valores BigDecimal.
     */
    private String bonus;

    public String getDescription() {
        return description;
    }

    public String getBonus() {
        return bonus;
    }

    private AcademicDegree(String description, String bonus) {
        this.description = description;
        this.bonus = bonus;
    }

}
```

Cada item enum tem uma descrição e um valor de bônus. Estas informações são utilizadas pelo construtor (ver abaixo) criar cada item enum.

Métodos comuns a todos os itens definidos neste enum.

O construtor de um enum deve ser SEMPRE privado. Ele é chamado internamente pelo sistema de execução JVM no momento em que as classes do programa são carregadas. O construtor utiliza os argumentos de cada item. Você não pode criar instâncias enum via o

Todos os itens enum são implicitamente `static` e `final`.

5. Veja como a implementação do cálculo seria simplificada agora:

```
public BigDecimal getAcademicBonus() {
    BigDecimal bonus = super.getHourlyRate().multiply(new BigDecimal(academicDegree.getBonus()));
    return bonus.setScale(2, BigDecimal.ROUND_HALF_EVEN);
}
```

E para utilizar a descrição:

```
System.out.println(professor.getAcademicDegree().getDescription());
```

6. Faça as modificações no seu programa e analise os resultados.