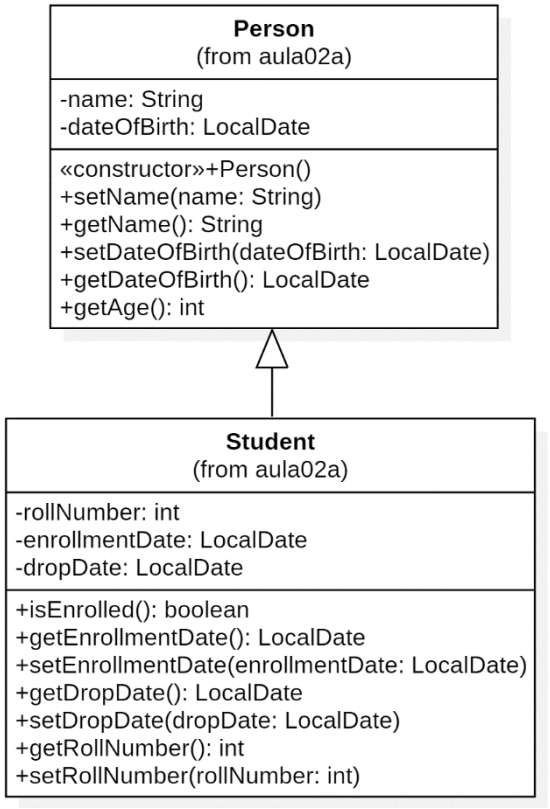
Roteiro 02 – Atividade 01/06

1. Complemente o diagrama iniciado no roteiro 01, adicionando a classe **Student** como uma subclasse de **Person**. Descompacte o arquivo **Student_javadoc.zip** (abra o arquivo index.html), disponibilizado neste roteiro, para ler a documentação **javadoc** desta classe. Para manter as classes desenvolvidas em cada aula, foi adicionado o pacote **aula02** na estrutura **br.univali.kob.poo1** já existente. Para não ter que modelar a classe Person novamente, foi utilizado o recurso de Copy & Paste da ferramenta. Você pode decidir por copiar a classe ou apenas arrastá-la para o diagrama de classe. A partir de agora, você terá liberdade para organizar seus pacotes e classes. Durante a disciplina, iremos discutir de modo recorrente sobre como organizar a documentação e os programas.

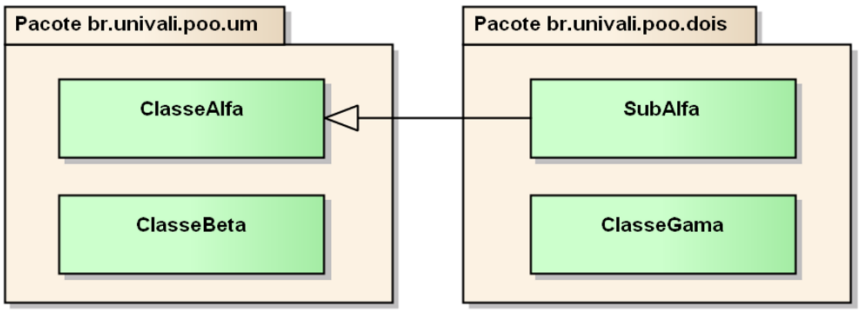


Note que a classe Person já considera a operação `getAge()`. Esta operação deveria ter sido modelada e implementada como um dos resultados do roteiro 01. Se você ainda não a implementou, faça isso agora. Tome cuidado para não deixar atividades acumularem. Veja também que, abaixo do nome da classe, aparece um texto (from aula02a). Este texto é colocado pela ferramenta StarUML para indicar que o pacote no qual a classe está fisicamente é diferente do pacote onde está o diagrama.

Relembrando os modificadores de visibilidade:

Notação visual	Modificador de acesso	A parte é visível...
+	public	dentro da própria classe e para qualquer outra classe
–	private	somente dentro da própria classe
#	protected	somente dentro do próprio pacote e das subclasses em outros pacotes
~	package	somente dentro da própria classe e das classes dentro do mesmo pacote

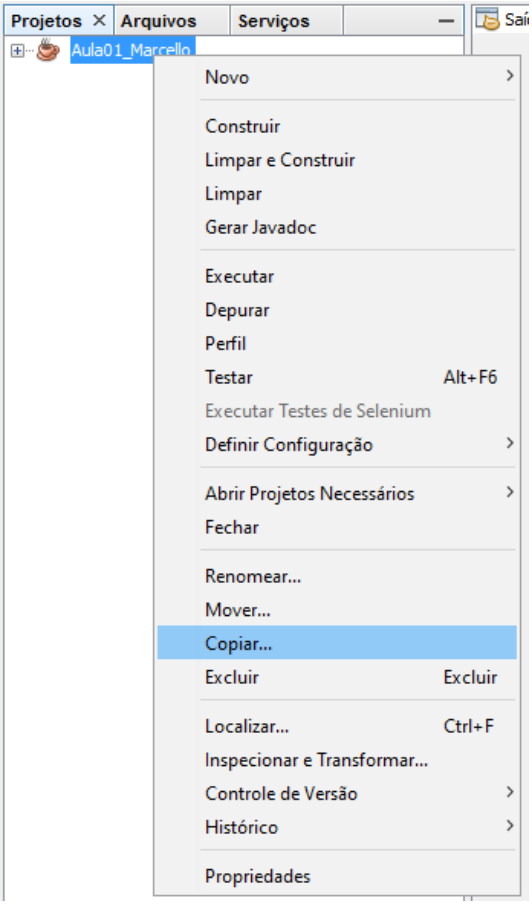
Para ajudar, veja o esquema abaixo (sub = subclasse):



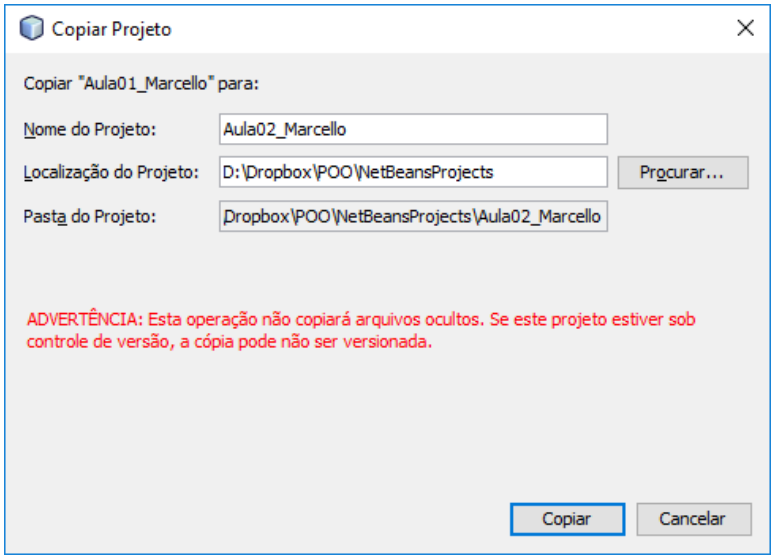
Modificador na ClasseAlfa	ClasseAlfa	ClasseBeta	SubAlfa	Gama
public	S	S	S	S
protected	S	S	S	N
sem modificador	S	S	N	N
private	S	N	N	N

Roteiro 02 – Atividade 01/06

2. Antes de iniciar sua implementação, é preciso decidir em qual projeto você trabalhará. Você pode optar por continuar no projeto iniciado no roteiro anterior. Entretanto, caso você queira criar um novo projeto para manter a associação do projeto com o roteiro, é possível fazer uma cópia de um projeto existente no NetBeans. Clique com o botão direito sobre seu projeto anterior e selecione a opção [Copiar].

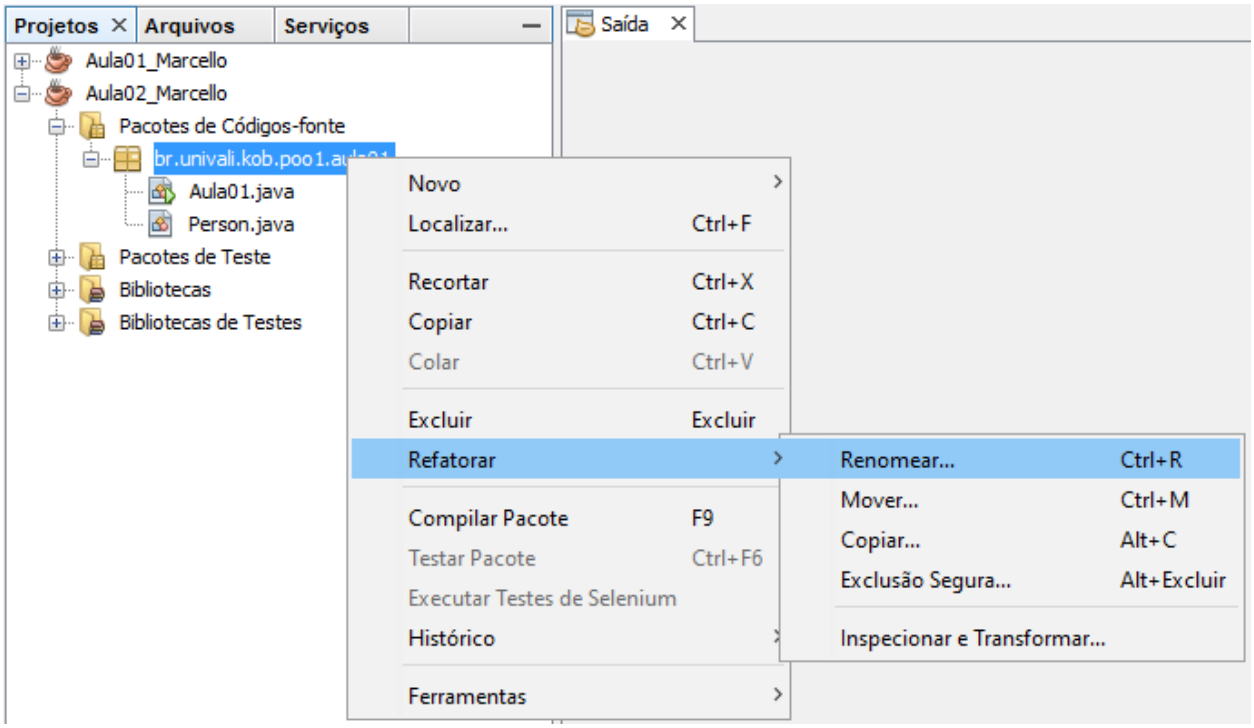


3. Informe o novo nome do projeto. Por exemplo, considerando o padrão adotado no roteiro anterior, utilizei **Aula02_Marcello**. Note que ao alterar o nome, a localização e pasta são alteradas automaticamente.

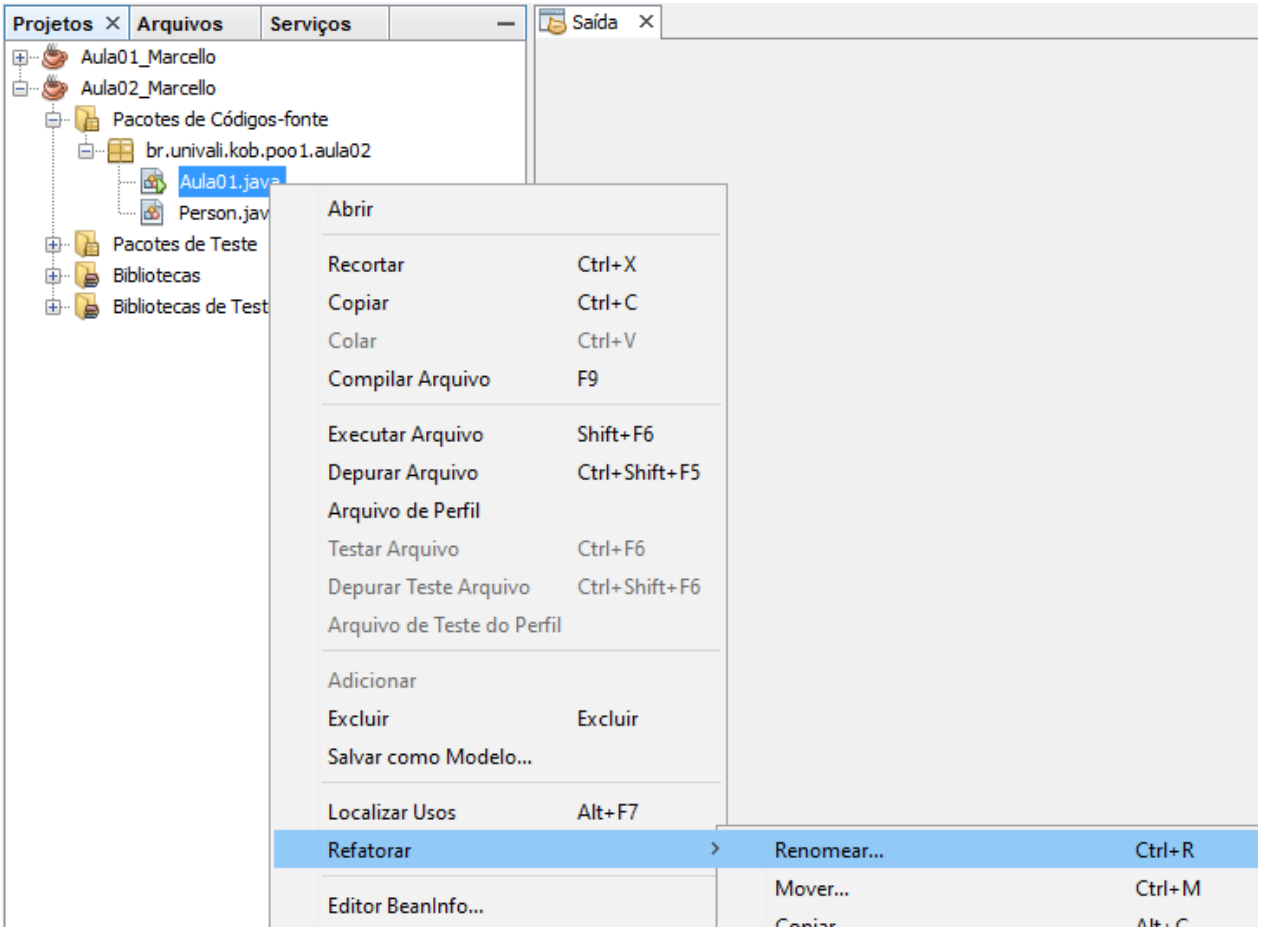


Roteiro 02 – Atividade 01/06

4. Agora, renomeie o pacote (por exemplo, **br.univali.kob.poo1.aula02**):

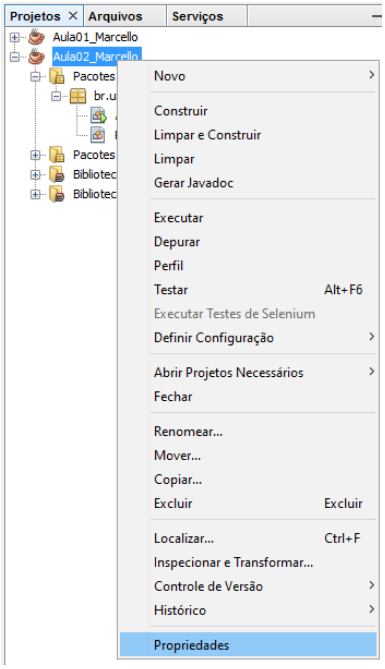


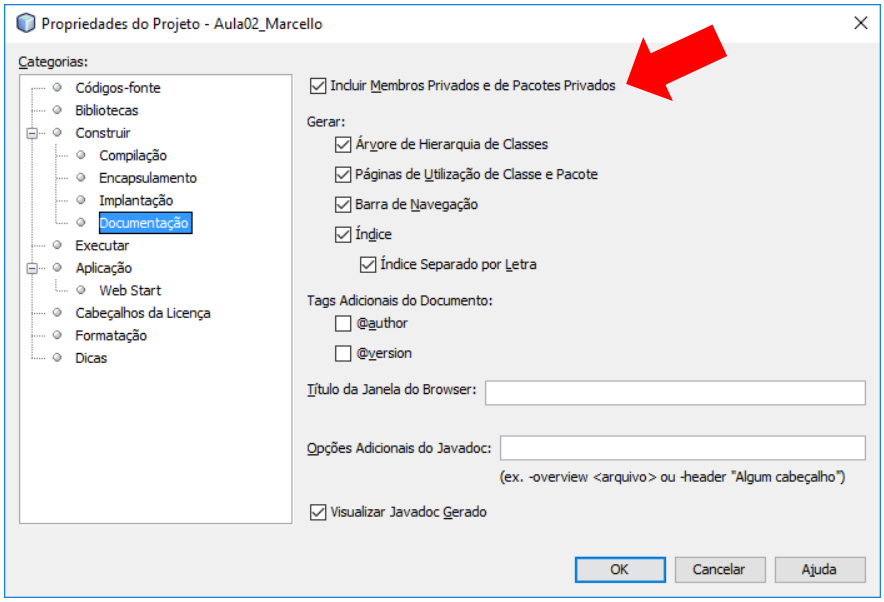
... e a classe principal (por exemplo, **Aula02**):



Roteiro 02 – Atividade 01/06

5. Agora, vamos configurar o NetBeans para que a documentação **javadoc** gerada considere também membros privados e protegidos (tipicamente, o default das ferramentas é permitir a geração apenas para membros públicos). Clique com o botão direito sobre o projeto e selecione [Propriedades]. Na janela, marque o texto "Incluir Membros Privados e de Pacotes Privados". Vale ressaltar que você alterou as propriedades deste projeto. Ou seja, você deve garantir que esta opção está marcada sempre que um novo projeto for criado.





6. Agora leia o arquivo **br.univali.kob.poo1.aula02a.Person.pdf** (disponibilizado com este roteiro) com bastante atenção, considerando os comentários e observando o código implementado. Compare com seu código e faça ajustes se necessário. Documente somente o **javadoc** que realmente faz sentido para a classe (em http://blog.joda.org/2012/11/javadoc-coding-standards.html, você encontrará boas práticas para o javadoc). Repare que vários comentários são orientações e explicações didáticas, podendo ser desconsiderados na sua implementação.

7. Implemente a classe **Student**. Se você ainda tem dúvidas sobre como fazer isso no NetBeans, revise o roteiro 01. Para a implementação da classe **Student**, veja o arquivo **br.univali.kob.poo1.aula02a.Student.pdf**, disponibilizado como parte deste roteiro.

8. Coloque a classe **StudentInheritanceTest.java**, disponibilizada neste roteiro, dentro do seu projeto. Faça os ajustes que forem necessários.

9. Ajuste a implementação da sua classe principal para executar os testes. Analise os resultados.

10. Experimente gerar o javadoc do seu projeto. No NetBeans, clique com o botão direito sobre o projeto e selecione a opção [Executar | Gerar javadoc].

11. Para avaliar o javadoc da classe aberta no editor, clique com o botão direito sobre ela e escolha a opção [Mostrar javadoc]. Para funcionar, lembre-se que você deve fazer isso na área de edição do código e não na árvore de projetos.

Roteiro 02 – Atividade 02/06

1. Adicione ao seu modelo UML uma nova classe chamada **Employee** (Empregado) com os seguintes atributos (utilize o campo **documentation** para que o **javadoc** seja gerado depois):
   - `int id` – identificador único para um empregado (similar ao **rollNumber** em **Student**).
   - `LocalDate hireDate` – data de contratação.
   - `LocalDate terminationDate` – data de demissão.
   - `int hoursPerWorkWeek` – quantidade de horas contratadas por semana.
   - `BigDecimal hourlyRate` – Valor (em R$) da hora contratada.

   > `BigDecimal` é uma classe que fornece muito mais precisão em relação aos tipos `float` e `double`. Note que objetos desta classe são imutáveis (assim como uma `String`). Logo, se você quiser multiplicar o valor atual por x, é necessário fazer
   >
   > `valorBigDecimal = valorBigDecimal.multiply(x);`
   >
   > BigDecimal também permite que você defina a escala (número de casas à direita do ponto decimal) a ser utilizada e o tipo de arredondamento. Veja maiores detalhes em:
   >
   > https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html
   >
   > Outro exemplo de trabalhar com valores financeiros:
   >
   > http://www.javapractices.com/topic/TopicAction.do?Id=13

2. Modele também as operações *getters* e *setters* para a classe **Employee**. Modele também uma operação para verificar se o empregado está trabalhando (**isEmployed**), a partir da data de demissão (se o empregado não tiver data de demissão, então consideraremos que ele está trabalhando).

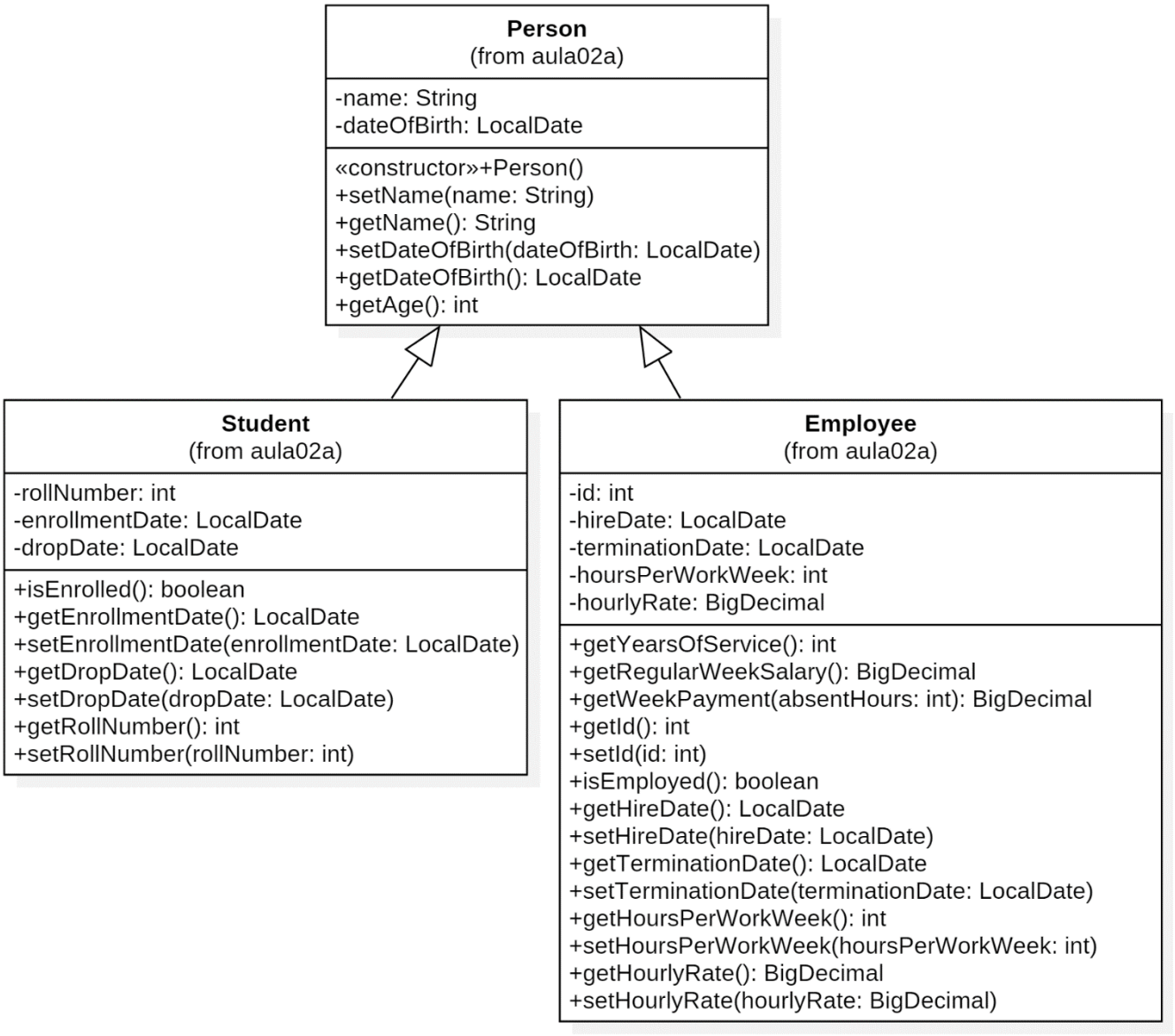
3. Modele uma operação de acesso que retorne o tempo de serviço (em anos) do empregado. Utilizaremos a assinatura "**+ getYearsOfService(): int**". Quando a operação for implementada (criação de um método para ela), verifique se o empregado ainda está trabalhando. Se estiver, considere a data atual no cálculo. Caso contrário, considere a data de demissão. Se você não tem ideia sobre como implementar esta operação, volte ao Roteiro 01 e veja a implementação do cálculo da idade. Evite apenas copiar e colar. Entenda o que você está fazendo.

4. Modele uma operação de acesso que retorne o salário base semanal (em R$) do empregado. Utilize a assinatura "**+ getRegularWeekSalary(): BigDecimal**". O salário é calculado pela multiplicação do valor da hora com a quantidade de horas contratadas por semana. Na implementação, utilize uma escala de 2 com o tipo de arredondamento `BigDecimal.ROUND_HALF_EVEN`. Em Java, o método será algo como:

```
/**
 * Calcula o salário base do empregado em R$, multiplicando a
 * quantidade de horas contratadas por semana pelo valor contratado
 * da hora. O salário base é fixo, não sendo afetado por faltas
 * do empregado.
 *
 * @return o valor do salário base semanal
 */
public BigDecimal getRegularWeekSalary() {
    BigDecimal value = new BigDecimal(hoursPerWorkWeek);
    value = value.multiply(getHourlyRate());
    return value.setScale(2, BigDecimal.ROUND_HALF_EVEN);
}
```

Roteiro 02 – Atividade 02/06

5. Modele uma operação que retorne o salário efetivo a ser pago em uma semana de trabalho. Utilize a assinatura "**+ getWeekPayment(int absentHours): BigDecimal**". O parâmetro `absentHours` indica a quantidade de horas em que o empregado esteve ausente do trabalho. Logo, você precisa subtrair estas horas do total contratado. Na implementação, utilize a mesma escala e tipo de arredondamento.



6. Agora que sua classe está modelada, você já pode implementá-la. Para verificar se sua implementação está funcionando, utilize a classe `EmployeeInheritanceTest` (disponibilizada como parte do material). Faça as contas manualmente para verificar se a saída está correta.

7. Crie uma outra classe de teste `PersonListTest` para montar uma lista de objetos Person que podem ser `Employee` ou `Student`**.** Utilize um vetor para isso. Se você ainda não sabe utilizar vetores em Java, veja o arquivo **POO1 - Slides de referência Java (Marcello Thiry).pdf**, disponibilizado como parte do Roteiro 01. Execute o programa e analise os resultados.

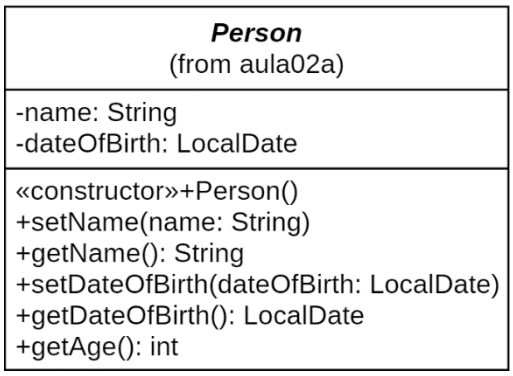
Roteiro 02 – Atividade 02/06

```
public class PersonListTest {

    /** Lista de Person ...5 lines */
    private Person[] personList = new Person[9];

    /** Caso de teste: Instancia e popula um estudante ...3 lines */
    private Student createStudent(int rollNumber, String name,
            String dateOfBirth, String enrollmentDate) {
        System.out.println("test case: createStudent");
        DateTimeFormatter format = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        Student student = new Student();
        student.setRollNumber(rollNumber);
        student.setName(name);
        student.setDateOfBirth(LocalDate.parse(dateOfBirth, format));
        student.setEnrollmentDate(LocalDate.parse(enrollmentDate, format));
        return student;
    }

    /** Caso de teste: Instancia e popula um empregado ...3 lines */
    private Employee createEmployee(int id, String name, String dateOfBirth,
            String hireDate, String hourlyRate, int hoursPerWorkWeek) {
        System.out.println("test case: createEmployee");
        DateTimeFormatter format = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        Employee employee = new Employee();
        employee.setId(id);
        employee.setName(name);
        employee.setDateOfBirth(LocalDate.parse(dateOfBirth, format));
        employee.setHireDate(LocalDate.parse(hireDate, format));
        employee.setHourlyRate(new BigDecimal(hourlyRate));
        employee.setHoursPerWorkWeek(hoursPerWorkWeek);
        return employee;
    }

    /** Caso de teste: Monta uma lista de empregados e estudantes ...3 lines */
    private void createPersonList() {
        System.out.println("test case: createPersonList");
        // criando empregados...
        personList[0] = createEmployee(1, "Orin Curry", "15/01/1976", "01/03/2000", "22.80", 40);
        personList[1] = createEmployee(2, "Susan Kent-Barr", "06/10/1969", "09/08/1995", "25.23", 40);
        // criando estudantes...
        personList[2] = createStudent(100, "Bruce Wayne", "02/05/1996", "01/03/2017");
        personList[3] = createStudent(100, "Emma Grace Frost", "23/09/1994", "31/07/2016");
        // agora percorrendo a lista (note que foram preenchidos apenas
        // 4 elementos de um vetor de 10 posições)
        for (int i = 0; i < 4; i++) {
            System.out.printf("%s, %d years old\n", personList[i].getName(), personList[i].getAge());
        }
        System.out.println();
    }

    /** Carga de teste: executa todos os casos de teste ...3 lines */
    public void run() {
        System.out.printf("\n\n\n******* aula2a: PersonListTest ******** \n\n");
        createPersonList();
    }

}
```

Roteiro 02 – Atividade 03/06

1. Se você já passou pelas perguntas da atividade anterior (caso contrário, eu recomendo que você faça isso agora), você deve ter percebido que instanciar objetos da classe `Person` não faz muito sentido. Por exemplo, se estivéssemos trabalhando em uma aplicação para uma instituição de ensino, quais papeis teriam apenas nome e data de nascimento? A utilidade de um objeto `Person` torna-se questionável, pois precisaremos de objetos mais especializados como estudantes, empregados, professores, etc. Por outro lado, parece adequado mantermos a **reusabilidade** oferecida por esta classe. Outra vantagem de mantermos a classe `Person` é garantirmos a padronização de uma interface comum com todas as suas subclasses (que permite o polimorfismo visto na atividade anterior). Uma solução para esta situação é transformar a classe `Person` em uma classe abstrata.

   Uma **classe abstrata** é uma classe que **não pode ser instanciada**, ou seja, não será possível criar objetos a partir dela. Desta forma, evitamos que os desenvolvedores criem objetos inúteis inadvertidamente. Portanto, classes abstratas só fazem sentido no contexto de uma hierarquia de herança. Atualize sua modelagem, atualizando **Person** como abstrata e verificando se o construtor default já foi modelado. Para colocar **Person** como abstrata, selecione a classe (no diagrama, clique no nome da classe) e marque **isAbstract** nas propriedades. Repare que o **nome da classe agora está em itálico**. Esta é a notação UML para indicar que a classe é abstrata.



ATENÇÃO

Nas provas da disciplina, você pode utilizar a notação **{abstract}** ao lado do nome da classe para indicar que ela é abstrata. Desta forma, não haverá dúvida na correção. Classes sem esta notação, serão consideradas concretas. Exemplo:

Person {abstract}

   Para tornar uma classe abstrata em Java, basta adicionarmos o modificador **abstract**:

```
public abstract class Person {
```

   Se seu projeto ainda tem a classe de teste com o método para criação de objetos **Person**, ele deve apresentar um marcador de erro agora. Caso o teste tenha sido excluído, escreva um simples e perceba que o compilador não deixará você fazer mais `new Pessoa()`.

2. Você já deve ter percebido que o construtor da classe `Person` também está representado na imagem acima (primeira operação). Veja que, antes do construtor, aparece a palavra **constructor** entre os símbolos **«** e **»**. Esta é a notação UML para um estereótipo. **Estereótipos** são utilizados para estender a UML (numa linguagem de programação OO, utilizamos nossas classes para estender a linguagem). Neste caso, o estereótipo está informando que a operação é um construtor.

   A UML é uma linguagem de modelagem independente de tecnologia, ou seja, ela pode ser usada com diferentes linguagens de programação. Como cada linguagem possui suas próprias regras para implementar construtores, a UML utiliza este estereótipo para eliminar esta ambiguidade. Para indicar que uma operação é um construtor em StarUML, basta colocar a palavra **constructor** (sem os símbolos) no campo **stereotype** (nas propriedades da operação). Se você ainda não o fez, faça isso agora e observe se a classe é mostrada como acima.

3. Você pode estar se perguntando se faz sentido termos um construtor em `Person`, uma vez que ela é uma classe abstrata. Ou seja, o construtor nunca seria invocado, certo? Errado. Ele nunca será invocado diretamente, mas o **construtor de uma superclasse abstrata pode ser invocado pelo construtor de uma subclasse concreta**. Com isso temos reusabilidade e evitamos que as subclasses precisem saber de detalhes internos da superclasse.

Roteiro 02 – Atividade 03/06

Vamos considerar que, no nosso estudo de caso (sistema acadêmico), não faz sentido instanciar um estudante ou empregado sem nome e data de nascimento. Ou seja, vamos considerar estes dados de preenchimento obrigatório. Para isso, podemos alterar a modelagem do construtor da classe Person para: **+Person(name: String, dateOfBirth: LocalDate)**. Ajuste sua modelagem agora.

4. Agora, podemos modificar o arquivo **Person.java**:

```
/**
 * Construtor para ser reutilizado pelas subclasses de
 * Person. Nome e data de nascimento são obrigatórios.
 *
 * @param name o nome da pessoa
 * @param dateOfBirth a data de nascimento da pessoa
 */
public Person(String name, LocalDate dateOfBirth) {
    setName(name);
    setDateOfBirth(dateOfBirth);
}
```

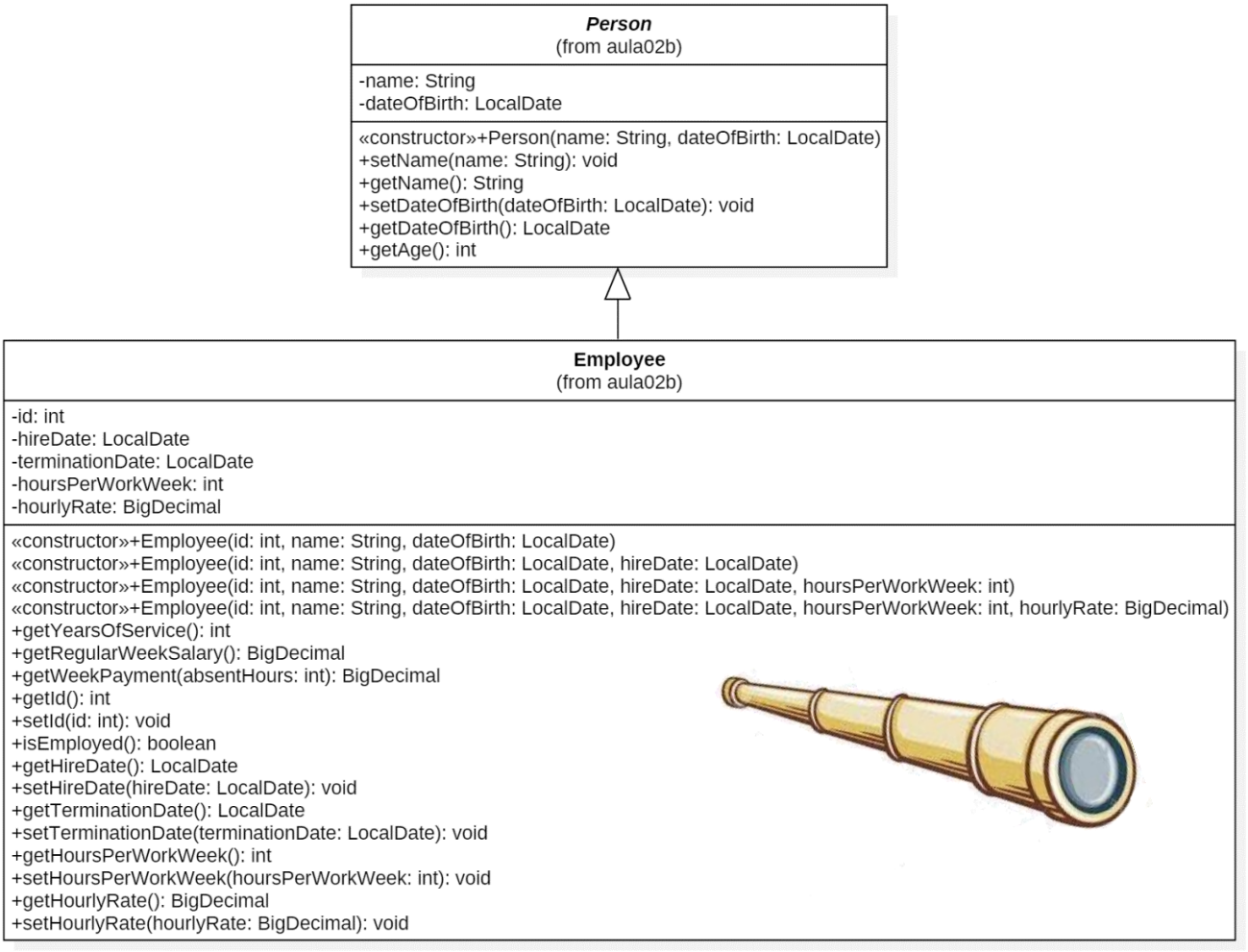
Sempre que uma classe Java tiver, pelo menos, um construtor implementado, o compilador só reconhecerá os construtores implementados na classe. Isso vale também para o construtor default.

Note que foram utilizados *setters* e não atribuições diretas aos atributos. Isso garante que, se adicionarmos mais tarde regras de validação nos *setters*, nosso construtor já estará pronto e não precisará ser modificado.

5. Antes de utilizarmos o construtor implementado em `Person`, vamos analisar o código das nossas classes de teste das atividades anteriores. Veja que, para criar um objeto, foi implementado um método que recebe os valores de cada atributo (parâmetros). Depois, o método "seta" estes valores atributo a atributo. Uma alternativa para instanciar um objeto já passando os valores de seus atributos é utilizar um **padrão de projeto** (design pattern) chamado **Telescoping**. Padrões de projeto são possíveis soluções para problemas recorrentes, as quais já foram testadas, utilizadas e aprimoradas por outros desenvolvedores. O padrão **Telescoping** permite instanciar objetos com várias opções de construtor. Cada implementação do construtor adiciona um novo parâmetro, representando mais um atributo a ser "setado" na criação do objeto. O nome **Telescoping** é uma metáfora (imagine-se abrindo uma luneta) para esta ampliação da assinatura a cada novo construtor.

   Uma classe pode ter vários construtores, desde que a lista de parâmetros (tipos) de cada um seja diferente. Isso é chamado de **sobrecarga (overload) de construtores**. Na prática, podemos sobrecarregar qualquer operação, desde de que mantida a mesma regra de diferentes listas de parâmetros. Entretanto, sobrecarga deve ser utilizada com restrições ("bom senso" se aplica aqui). Sobrecarga traz implicitamente a ideia de que uma mesma operação pode fazer várias coisas (mesmo que, na prática, sejam métodos diferentes). O **uso abusivo de operações sobrecarregadas pode dificultar a legibilidade do código**, pois o desenvolvedor precisará sempre verificar o que aquela versão da operação realmente faz. Uma boa prática é utilizar **sobrecarga somente quando a semântica da operação é mantida**. Ou seja, a operação tem exatamente a mesma finalidade, somente com uma configuração de parâmetros diferente. Se você precisa fazer outra coisa, defina outra operação.

Roteiro 02 – Atividade 03/06



Na classe Person, consideramos que os atributos nome e data de nascimento são obrigatórios, pois não oferecemos outra forma de instanciar o óbjeto que não seja passando estes dois valores. Os construtores das classes `Student` e `Employee` devem considerar isso. Por exemplo, na classe `Employee` (imagem acima) podemos observar que o primeiro construtor define que o identificador também passa de preenchimento obrigatório. Note que não existe um construtor sem que você tenha que informar o identificador. No caso de `Student`, podemos considerar que o primeiro construtor também exige que o número de matrícula seja informado. Atualize seu diagrama de classe, inserindo os construtores para as classes `Employee` e `Student`.

Agora, gere o código a partir da ferramenta CASE[1] StarUML [Tool | Java | Generate Code...]. Infelizmente, esta ferramenta não possui recurso de sincronização com código. Em ferramentas mais completas (tipicamente, em suas versões pagas), é possível manter o sincronismo entre os modelos e os respectivos programas. Em sistemas de maior porte, isso é um requisito essencial para manter a documentação, sem perder produtividade. Aqui, iremos observar o código gerado para entender o resultado. Você pode, eventualmente, copiar partes do código gerado e atualizar suas classes Java manualmente.

Outra limitação desta ferramenta é que ela não interpreta o estereótipo **construtor**. Além de gerar sempre (mesmo não tendo sido modelado) o construtor default, ela gera os construtores como operações comuns (veja imagem abaixo). A ferramenta StarUML pode ser estendida com novas funcionalidades implementadas por terceiros. A extensão de geração de código Java é um exemplo (disponível em https://github.com/staruml/Java). Quando você tiver mais domínio sobre a linguagem Java e conceitos OO, talvez você queira evoluir esta extensão.

[1] Lembre-se que CASE significa Engenharia de Software apoiada por Computador em Inglês. São ferramentas que apoiam qualquer parte do ciclo de desenvolvimento de software.

Roteiro 02 – Atividade 03/06

Seria adequado também conhecimento sobre linguagens formais e compiladores (você verá este conteúdo em outra disciplina do curso).

```
public class Employee extends Person {

    /**
     * Default constructor
     */
    public Employee() {
    }

    ...

    private BigDecimal hourlyRate;

    /**
     * Telescoping pattern.
     * @param id o identificador único do empregado
     * @param name o nome do empregado
     * @param dateOfBirth a data de nascimento do empregado
     */
    public void Employee(int id, String name, LocalDate dateOfBirth) {
        // TODO implement here
    }

    /**
     * Telescoping pattern.
     * @param id o identificador único do empregado
     * @param name o nome do empregado
     * @param dateOfBirth a data de nascimento do empregado
     * @param hireDate a data de contratação
     */
    public void Employee(int id, String name, LocalDate dateOfBirth, LocalDate hireDate) {
        // TODO implement here
    }

    /**
     * Telescoping pattern.
     * @param id o identificador único do empregado
     * @param name o nome do empregado
     * @param dateOfBirth a data de nascimento do empregado
     * @param hireDate a data de contratação
     * @param hoursPerWorkWeek a quantidade de horas constratadas por semana
     */
    public void Employee(int id, String name, LocalDate dateOfBirth, LocalDate hireDate, int hoursPerWorkWeek) {
        // TODO implement here
    }

    ...
```

O construtor default é sempre gerado pela StarUML. No nosso caso, podemos excluí-lo (considerando que queremos forçar a criação com passagem de parâmetros).

Veja que o código gerado tratou nossos construtores como métodos comuns da linguagem, inserindo o tipo de retorno **void**. Para funcionar, precisamos excluir o tipo de retorno em cada um dos nossos construtores. Você também deve ter observado o **javadoc** gerado. Isso não foi mágica. Você precisa preencher o campo **Documentation** na ferramenta.

6. Podemos agora detalhar a implementação de um dos construtores de `Employee`:

```
/**
 * Telescoping pattern.
 *
 * @param id o identificador único do empregado
 * @param name o nome do empregado
 * @param dateOfBirth a data de nascimento do empregado
 */
public Employee(int id, String name, LocalDate dateOfBirth) {
    super(name, dateOfBirth);
    setId(id);
    // Foi considerado que nunca instanciaremos um estudante
    // já desligado
    setTerminationDate(null);
}
```

Observe que a primeira linha utiliza a palavra reservada `super` (de superclasse) para acessar o construtor herdado de `Person` (aquele que implementamos anteriormente). Para construtores, ao invés de utilizamos `super.Person(name, dateOfBirth)`, utilizamos apenas `super(name, dateOfBirth)`. O uso de `super` em um construtor deve ser feito obrigatoriamente na sua primeira linha.

Roteiro 02 – Atividade 03/06

Outra observação é a utilização dos *setters* ao invés fazer a atribuição diretamente no atributo. Já discutimos isso quando implementamos o construtor em Person, mas vale reforçar. Esta prática garante que, ao inserirmos regras de validação em nossos *setters*, não precisaremos modificar a implementação do construtor. Além disso, a data de demissão foi inicializada com um objeto nulo para garantir que um empregado sempre é instanciado com a situação "trabalhando". Esta foi uma consideração arbitrária de negócio aqui. Em um sistema real, precisaríamos verificar quais regras se aplicam.

7. No segundo construtor de `Employee`, podemos observar um uso diferente para a palavra reservada `this`:

```
/**
 * Telescoping pattern.
 *
 * @param id o identificador único do empregado
 * @param name o nome do empregado
 * @param dateOfBirth a data de nascimento do empregado
 * @param hireDate a data de contratação do empregado
 */
public Employee(int id, String name, LocalDate dateOfBirth, LocalDate hireDate) {
    this(id, name, dateOfBirth);
    setHireDate(hireDate);
}
```

Utilizamos `this` dentro de um construtor para invocar (chamar) explicitamente outro construtor já implementado na classe. Quando presente em um construtor, a chamada deve ser obrigatoriamente a primeira linha do construtor.

8. Implemente o padrão *Telescoping* para as classes `Employee` e `Student`. Note que você precisará ajustar os testes, uma vez que não temos mais os construtores default: `Employee()` e `Student()`.

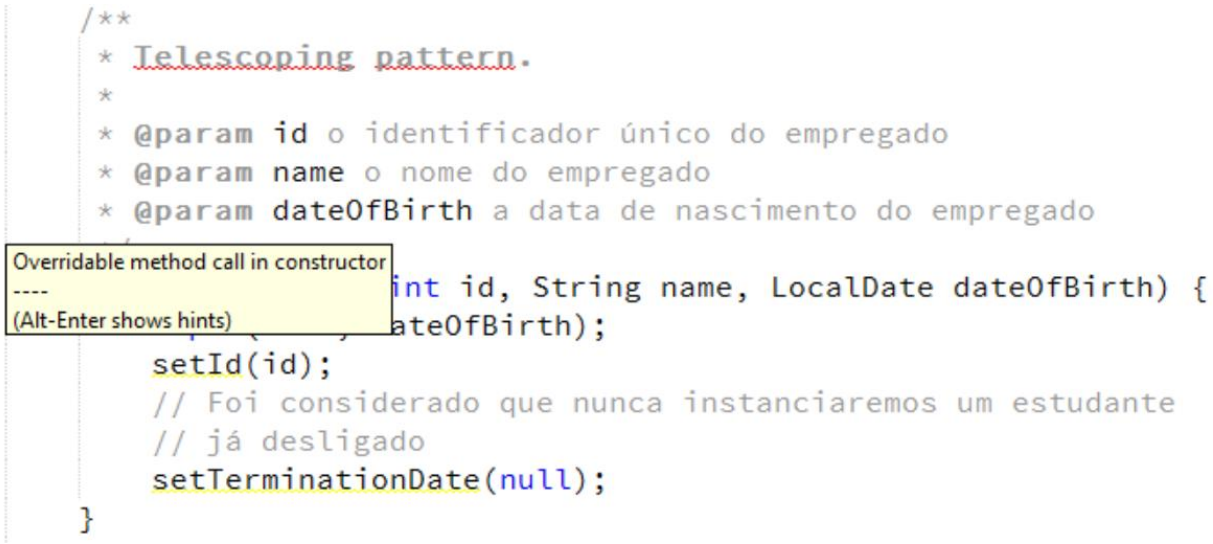
Note que podemos perder legibilidade do código na chamada do construtor, principalmente, se forem utilizadas literais. Neste caso, precisaremos conhecer a posição dos parâmetros para passar os argumentos ou para entender a chamada. Outra desvantagem do padrão Telescoping é que a lista de parâmetros pode ser muito longa (caracterizando um **code smell** / **bad smell** e indicando um ponto de melhoria no código). Além disso, a manutenção do programa poderá ser problemática. Adicionar atributos exigirão que novos construtores sejam implementados (perderemos escalabilidade).

Uma possível otimização é implementar o primeiro construtor com todos os atributos de preenchimento obrigatório. Desta forma, é possível reduzir algumas combinações. Por estes motivos, o Telescoping é algumas vezes chamado de **anti-pattern**, ou seja, o contrário de uma boa prática de programação (um padrão a ser evitado).

Desta forma, não descarte a forma como havíamos construído objetos anteriormente. Você deve sempre analisar o problema para identificar possíveis soluções. A escolha será aquela com melhor custo/benefício. Durante a disciplina, veremos outras alternativas para criar objetos com seus estados iniciais

Roteiro 02 – Atividade 03/06

9. Em tempo, você observou que algumas chamadas de método estão sublinhadas em amarelo. Passe o mouse sobre o marcador de alerta (*warning*):



Você tem alguma ideia do motivo para este alerta? O que você entende por um método substituível (*overridable method*)? Note que seu programa compilou sem problemas. Voltaremos a este tópico no próximo roteiro.