



C206/C06 – Programação Orientada a Objetos  
com Java



# Herança e Polimorfismo

Prof. Christopher Lima  
christopher@inatel.br



# Objetivos



- ☕ Entender o conceito de Herança
- ☕ Entender como aplicar o Polimorfismo
- ☕ Fazer sobrescrita de métodos
- ☕ Reutilizar código

# Modelando os uma loja de brownies



☕ Suponha que exista 3 tipos de Brownies

☕ BrownieCafe

☕ BrownieDoceDeLeite

☕ BrownieNutella

☕ Vamos escrever a classe para modelar o BrownieNutella



```
package br.inatel.cdg.exercicio1.brownies;
```

```
public class Brownie {
```

```
    protected String nome;
```

```
    protected double preco;
```

```
    protected String sabor;
```

```
    public Brownie(String nome, double preco, String sabor) {
```

```
        this.nome = nome;
```

```
        this.preco = preco;
```

```
        this.sabor = sabor;
```

```
    }
```

```
    public void addCarrinhoDeCompras() {
```

```
        System.out.println("Adicionando no carrinho de compras um: "+ nome);
```

```
    }
```

```
    public void calculaValorTotalCompra() {
```

```
        System.out.println("Calculando valor total da compra de um: "+ nome +": "+preco);
```

```
    }
```

# Modelando os uma loja de brownies



☕ Vamos agora escrever a classe para modelar o BrownieCafe

```
package br.inatel.cdg.exercicio1.brownies;
```

```
public class BrownieCafe {
```

```
    protected String nome;  
    protected double preco;  
    protected String sabor;
```

```
    public BrownieCafe(String nome, double preco, String sabor) {  
        this.nome = nome;  
        this.preco = preco;  
        this.sabor = sabor;  
    }
```

```
    public void addCarrinhoDeCompras() {
```

```
        System.out.println("Adicionando no carrinho de compras um: "+ nome);  
    }
```

```
    public void calculaValorTotalCompra() {
```

```
        System.out.println("Calculando valor total da compra de um: "+ nome + ": "+preco);  
    }
```

```
    public void adicionaCafe() {
```

```
        System.out.println(super.nome +" adicionando mais café");  
    }
```



# Modelando os uma loja de brownies



- ☕ Elas estão bem parecidas correto?
- ☕ Se olhar rapidamente, *parecem a mesma classe!*
- ☕ O método *adicionaMaisCafe()* o BrownieNutella não possui
- ☕ Precisamos *repetir todo esse código* para cada novo Brownie que queremos modelar nessa loja?



# Modelando os uma loja de brownies



- ☕ Podemos ver que essas **classes compartilham** muitas **característica**!
- ☕ Deve existir algum **recurso para escrevermos menos código**
- ☕ E se essas classes compartilhassem uma mesma **abstração**, algo como uma classe **“Brownie”**? E em seguida pudessem **herdar** os membros e métodos?







Herança!

# Herança



- ☞ É um **recurso** do paradigma orientado a objetos
- ☞ Permite que classes possam **herdar métodos e membros de uma classe Mãe**, também conhecida como **superclasse**. As classes que herdam são **classes filhas, ou subclasse**.
- ☞ Como ficaria nosso exemplo da loja de Brownies?
- ☞ Vamos primeiro criar uma classe Brownie com os **membros e métodos comuns**.



```
package br.inatel.cdg.exercicio1.brownies;
```

```
public class Brownie {
```

```
    protected String nome;
```

```
    protected double preco;
```

```
    protected String sabor;
```

```
    public Brownie(String nome, double preco, String sabor) {
```

```
        this.nome = nome;
```

```
        this.preco = preco;
```

```
        this.sabor = sabor;
```

```
    }
```

```
    public void addCarrinhoDeCompras() {
```

```
        System.out.println("Adicionando no carrinho de compras um: "+ nome);
```

```
    }
```

```
    public void calculaValorTotalCompra() {
```

```
        System.out.println("Calculando valor total da compra de um: "+ nome +": "+preco);
```

```
    }
```

# Modelando os uma loja de brownies



- ☕ Observe um novo modificador chamado `protected`. Ele possui uma visibilidade mais limitada que o `public` e menos restrita que `private`. Com esse modificador, `somente a própria classe e as subclasses podem ter acesso a esses membros`.
- ☕ Normalmente esse modificador é utilizado nos membros das `superclasses`.



# Modelando o Brownie

- ☕ Observe que na classe Brownie, **não existe método *adicionaMaisCafe()***, pois é específico do BrownieCafe. Na classe Brownie deixamos apenas o que for **comum a TODOS** os Brownies!
- ☕ Para que as classes BrownieNutella e BrownieCafe possam herdar da classe Brownie, usamos a palavra chave **extends** (para a linguagem Java).
- ☕ Outras linguagens OO como C# e C++ possuem outra sintaxe para que as classes possam **herdar** das superclasses.
- ☕ Vamos ao resultado!

```
package br.inatel.cdg.exercicio1.brownies;
```

```
public class BrownieCafe extends Brownie {
```

```
    public BrownieCafe(String nome, double preco, String sabor) {
```

```
        super(nome,preco,sabor);
```

```
    }
```

```
    public void adicionaCafe() {
```

```
        System.out.println(super.nome + " adicionando mais café");
```

```
    }
```

```
package br.inatel.cdg.exercicio1.brownies;
```

```
public class BrownieNutella extends Brownie {
```

```
    public BrownieNutella(String nome, double preco, String sabor) {
```

```
        super(nome,preco,sabor);
```

```
    }
```

```
    public void adicionaNutella() {
```

```
        System.out.println(super.nome + " adicionando mais nutella");
```

```
    }
```

```
}
```



# Modelando o Brownie com Herança





- ☕ As classes BrownieNutella e BrownieCafe ficaram **bem menores** não é mesmo?
- ☕ Agora fica bem mais fácil evoluir esse software



# Modelando o Brownie com Herança



## Observações

-  A palavra chave `super` se refere a `superclasse`, nesse exemplo a `Brownie`. Ou seja, estamos `chamando o construtor da superclasse Inimigo`.
-  Na classe `BrownieCafe`, colocamos o método `adicionaMaisCafe()`, pois é uma `especialização` dessa classe. Não existe razão para colocarmos ela na classe `BrownieNutella`.

 Vamos testar essas classes!





# Modelando o Brownie com Herança

```
public class Main {  
  
    public static void main(String[] args) {  
  
        BrownieCafe bwCafe = new BrownieCafe("Brownie de Café", 10, "Café");  
        BrownieNutella bwNutella = new BrownieNutella("Brownie de Nutella", 70, "Nutella");  
  
        bwNutella.addCarrinhoDeCompras();  
        bwCafe.addCarrinhoDeCompras();  
        bwCafe.adicionaCafe();  
    }  
}
```

Adicionando no carrinho de compras um: BrownieNutella  
Adicionando no carrinho de compras um: BrownieCafe  
Adicionando mais café...

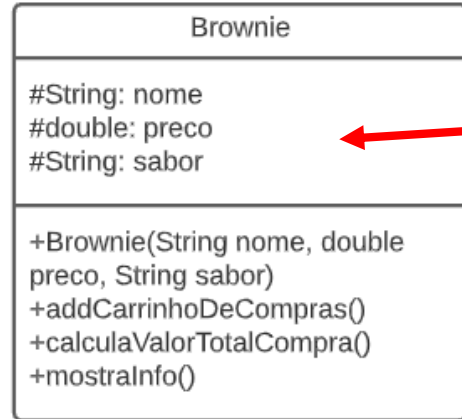
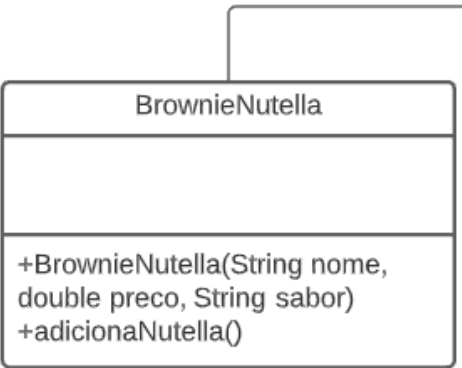


# UML com Herança

BrownieNutella *herda de*  
Brownie

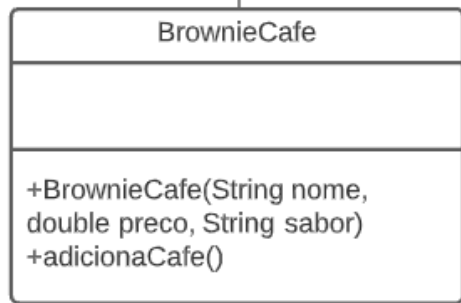
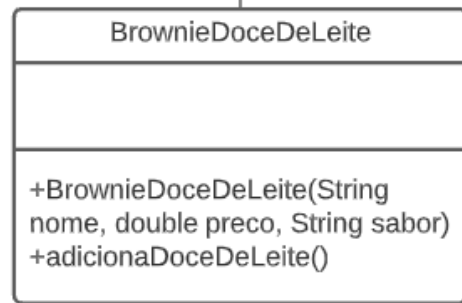
BrownieCafe *herda de*  
Brownie

BrownieDoceDeLeite *herda de*  
Brownie



# Indica protected

Indica Herança





# Polimorfismo

☕ POLI = muitas

☕ MORFO = formas

☕ Muitas formas de se fazer alguma coisa.

☕ Permite que um **mesmo nome**, represente **vários comportamentos diferentes**.



# Sobrescrita de Métodos

☕ Temos como especializar comportamento?

☕ Sim! Através da sobrescrita!

# Sobrescrita de Métodos



```
public class BrownieNutella extends Brownie {  
  
    public BrownieNutella(String nome, double preco, String sabor) {  
  
        super(nome, preco, sabor);  
    }  
  
    @Override  
    public void addCarrinhoDeCompras() {  
  
        System.out.println("Adicionando um Brownie de Nutella no carrinho de compras");  
    }  
  
}
```

# Sobrescrita de Métodos



- ☕ Observe que colocamos uma anotação `@Override` em cima do método “`addCarrinhoDeCompras()`” para indicar que estamos **sobrescrevendo um método da superclasse**. Fizemos isso na classe `BrownieNutella`.
- ☕ Podemos assim **especializar comportamento nas subclasses**.
- ☕ Vamos testar essa chamada.



```
public class Main {  
  
    public static void main(String[] args) {  
        BrownieNutella bn = new BrownieNutella(nome: "BrownieNutella", preco: 100, sabor: "Nutella");  
        BrownieCafe bc = new BrownieCafe(nome: "BrownieCafe", preco: 75, sabor: "CafeEspecial");  
  
        bn.addCarrinhoDeCompras();  
        bc.addCarrinhoDeCompras();  
        bc.adicionaMaisCafe();  
    }  
}
```

```
Adicionando um Brownie de Nutella no carrinho de compras  
Adicionando no carrinho de compras um: BrownieCafe  
Adicionando mais café...
```



# Polimorfismo

- ☞ Quando dizemos que BrownieNutella **herda** da classe Brownie, dizemos um BrownieNutella **É UM** Brownie.
- ☞ Com essa definição, se uma classe Comprador é capaz de comprar um Brownie, ele pode comprar qualquer classe que **herda** de Brownie.
- ☞ Se toda subclasse de Brownie **É UM** Brownie, então podemos salvar uma referência de BrownieNutella em uma variável do tipo Brownie?
- ☞ Sim!
- ☞ Veja o código a seguir



# Polimorfismo



```
public static void main(String[] args) {  
    Brownie bn = new BrownieNutella( nome: "BrownieNutella", preco: 100, sabor: "Nutella");  
    Brownie bc = new BrownieCafe( nome: "BrownieCafe", preco: 75, sabor: "CafeEspecial");  
  
    bn.addCarrinhoDeCompras();  
    bc.addCarrinhoDeCompras();  
}
```

# Polimorfismo



- ☕ Perceba que agora **as instâncias** de `BrownieNutella` e `BrownieCafe` foram **salvas em uma variável do tipo `Brownie`**
- ☕ Mas isso não significa que essas instâncias mudaram de tipo. O que mudou foi a forma como elas estão sendo referenciadas. **Elas continuam sendo instâncias de `BrownieNutella` e `BrownieCafe`**, mas suas referências podem ser armazenadas em qualquer variável **que seja de uma superclasse**, como a `Brownie` por exemplo!

# Polimorfismo



```
public class Main {  
  
    public static void main(String[] args) {  
        Brownie bn = new BrownieNutella( nome: "BrownieNutella", preco: 100, sabor: "Nutella");  
        Brownie bc = new BrownieCafe( nome: "BrownieCafe", preco: 75, sabor: "CafeEspecial");  
  
        bn.addCarrinhoDeCompras();    Adicionando um Brownie de Nutella no carrinho de compras  
        bc.addCarrinhoDeCompras();    Adicionando no carrinho de compras um: BrownieCafe  
    }  
}
```

☕ O que será impresso ao executar o código? O método `addCarrinhoDeCompras()` original na classe `Brownie`, ou o método `addCarrinhoDeCompras()` que foi sobrescrito nas classes `BrownieNutella` e `BrownieCafe`?

# Polimorfismo



- ☕ Dado que as instâncias são de fato BrownieNutella e BrownieCafe, o método chamado **será o presente nessas classes** e não ao da classe Brownie.
- ☕ Se não tivéssemos feito uma sobrescrita, seria chamado o método original definido na classe Brownie.
- ☕ Porém, perceba que como as instâncias estão salvas em variáveis do tipo Brownie, **não podemos invocar métodos específicos das instâncias.**
- ☕ Exemplo: Não conseguimos invocar o método adicionaMaisCafe(), definido na classe BrownieCafe, **pois a classe Brownie não “conhece” esse método.**

# Polimorfismo



```
public static void main(String[] args) {  
    Brownie bn = new BrownieNutella( nome: "BrownieNutella", preco: 100, sabor: "Nutella");  
    Brownie bc = new BrownieCafe( nome: "BrownieCafe", preco: 75, sabor: "CafeEspecial");  
  
    bn.addCarrinhoDeCompras();  
    bc.addCarrinhoDeCompras();  
    bc.adicionaMaisCafe();  
}
```



# Polimorfismo

- ☕ Mas como fazemos para invocar o método `adicionaMaisCafe()`?
- ☕ Primeiro precisamos verificar se uma instância é de um tipo, e depois trocar o tipo de variável de referencia.
- ☕ Podemos usar o operador `instanceof()`. Com esse operador podemos testar se uma instância é de um determinado tipo. Se for, podemos fazer o `casting` e em seguida trabalhar com métodos específicos.

# Polimorfismo



```
public static void main(String[] args) {  
    Brownie bn = new BrownieNutella(nome: "BrownieNutella", preco: 100, sabor: "Nutella");  
    Brownie bc = new BrownieCafe(nome: "BrownieCafe", preco: 75, sabor: "CafeEspecial");  
  
    bn.addCarrinhoDeCompras();  
    bc.addCarrinhoDeCompras();  
  
    if (bc instanceof BrownieCafe){  
        BrownieCafe browniecafe = (BrownieCafe) bc;  
        browniecafe.adicionaMaisCafe();  
    }  
}
```



# Polimorfismo

- ☕ Qual a vantagem do Polimorfismo?
- ☕ Com ele podemos criar **classes que lidam apenas com as abstrações** (ou superclasse) e assim podemos deixar nossas **funcionalidades mais genéricas**.
- ☕ Considere uma classe Comprador, que possui um método que recebe um parâmetro do tipo Brownie e faz alguma operação nessa variável.



# Polimorfismo



```
public class Comprador {  
  
    public void comprarBrownies(Brownie brownie){  
        System.out.println("Comprando um brownie" + brownie.getNome());  
    }  
}
```

- ☕ Qualquer instância que **herda** de Brownie pode ser passada como parâmetro!
- ☕ Isso traz **poder e flexibilidade para escrever programas**.
- ☕ Perceba como fica fácil evoluir esse software, criando novos tipos de Brownies.



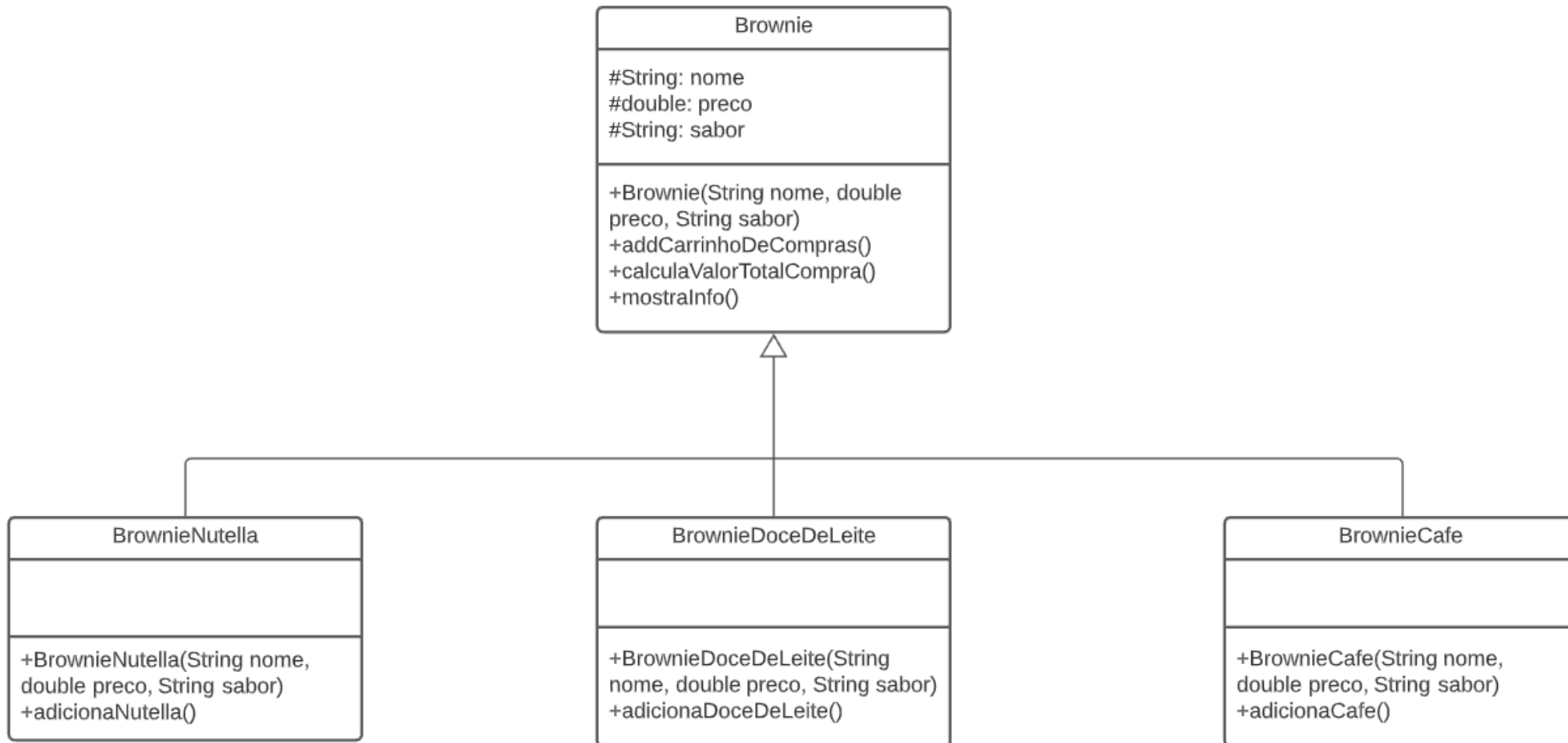
## Exercício 1 – Brownie



- ☕ Crie classes que modelam o diagrama UML do próximo *slide*
- ☕ Faça testes na classe Main
- ☕ Os métodos devem imprimir de forma trivial o comportamento, sempre exibindo o nome de quem está fazendo aquela ação.
- ☕ O método `mostraInfo()` deve mostrar o valor de todos os atributos.
- ☕ O método `calculaValorTotalCompra()` deve imprimir, além do nome, o preço do Brownie.



# Exercício 1 – Brownie





## Exercício 2 – Brownie PT2

☕ Faça uma modificação no código do Exercício 1, e permita que cada classe que herde de Brownie, faça a sobrescrita do método *addCarrinhoDeCompras()*, e personalize a mensagem. Chama esse método na Main.





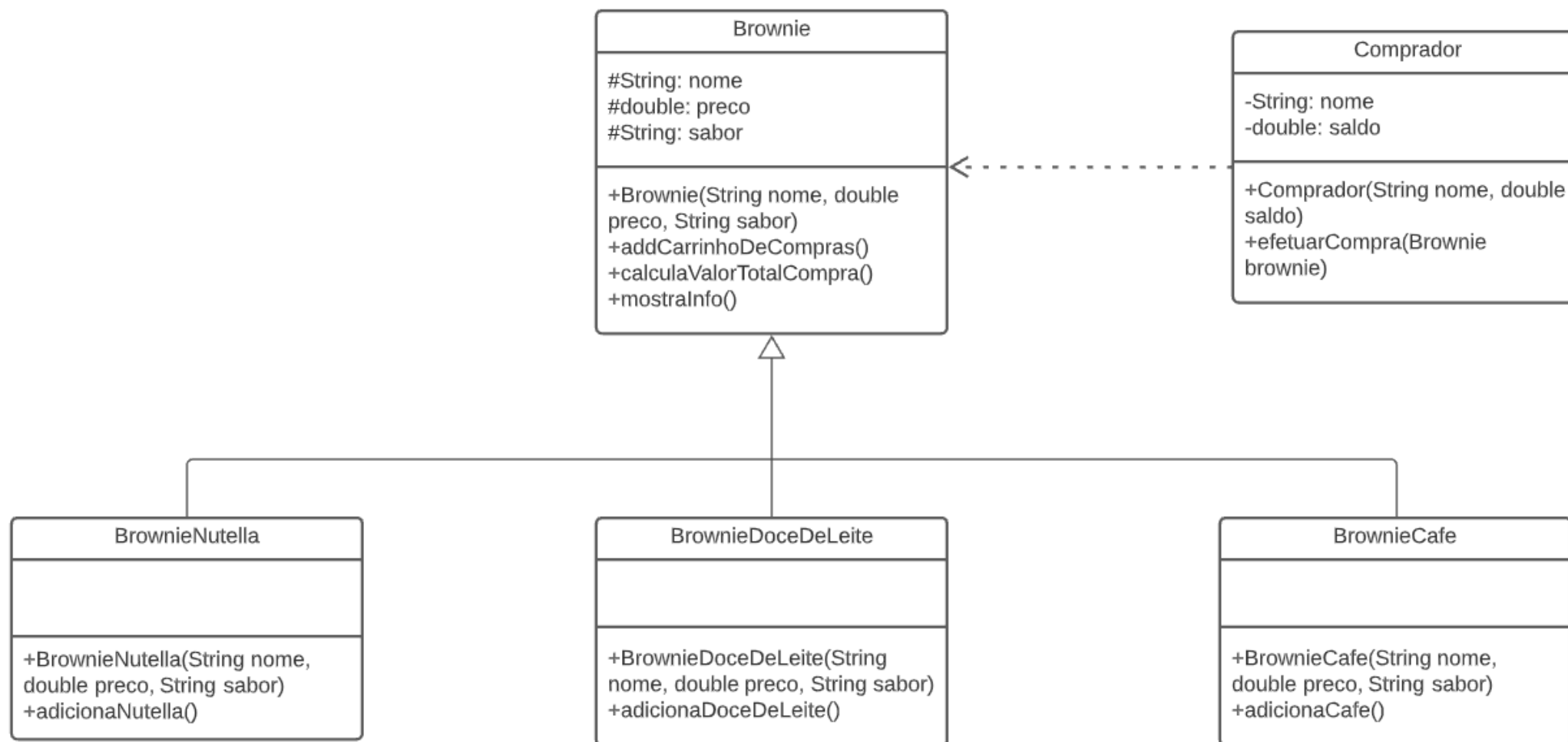
## Exercício 3 – Brownie PT3



- ☕ Considere o UML do próximo Slide.
- ☕ Crie uma classe Comprador, que recebe um Brownie e efetua uma compra. Faça com o que o próprio Brownie invoque o método `addCarrinhoDeCompra()` e `calculaValorTotalCompra()`. Imprima também o nome do Brownie que está sendo comprado.
- ☕ Verifique na Main o fluxo de chamadas feitas pelo Comprador.



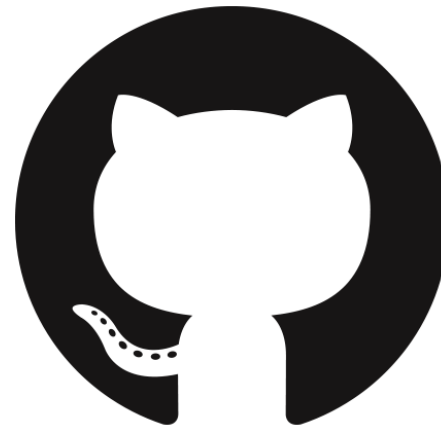
# Exercício 3 – Brownie PT3



# Resolução dos Exercícios



[https://github.com/chrislima-inatel/C206\\_C125](https://github.com/chrislima-inatel/C206_C125)



# Material Complementar



- ☕ Capítulo 9 da apostila FJ-11
  - ☕ Herança, Reescrita e Polimorfismo

