

# **Introdução à programação orientada a objetos**

# Bibliografia

- BARKER, J. **Beginning Java Objects**. From Concepts to Code. New York: Springer, 2<sup>a</sup> ed. 2005.
- SANTOS, Rafael. **Introdução à programação orientada a objetos usando JAVA**.2. ed. Rio de Janeiro : Elsevier, 2013. 313 p, il.
- WINBLAD, Ann L; EDWARDS, Samuel D; KING, David R. **Software orientado ao objeto**. Sao Paulo : Makron Books, 1993. xxvi, 314p, il.

# Paradigmas de programação

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa

```
program Fibonacci;

function fib(n: Integer): Integer;
var a: Integer = 1;
    b: Integer = 1;
    f: Integer;
    i: Integer;
begin
  if (n = 1) or (n = 2) then
    fib := 1
  else
    begin
      for i := 3 to n do
        begin
          f := a + b;
          b := a;
          a := f;
        end;
        fib := f;
      end;
  end;

begin
  WriteLn(fib(10));
end.
```

```
import Text.Printf

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

main = printf "%d\n" (fib 10)
```

Funcional

```
fib(1, 1).
fib(2, 1).

fib(X, Y):-
  X > 1,
  X1 is X - 1,
  X2 is X - 2,
  fib(X1, Z),
  fib(X2, W),
  Y is W + Z.

main :-
  fib(10,X), write(X), nl.
```

Lógico

```
public class Fibonacci {

  public long fibonacci(int n) {
    if(n == 0) {
      return 0;
    } else if(n == 1) {
      return 1;
    } else {
      return fibonacci(n - 1) +
             fibonacci(n - 2);
    }
  }

  public static void main(String[] args) {
    Fibonacci f = new Fibonacci();
    System.out.printf(f.fibonacci(10));
  }
}
```

Orientado a objetos

Imperativo

# Paradigma orientado a objetos

- Surgiu na década de 60
- Primeiras linguagens comerciais surgiram na década de 90
- É uma visão contemporânea que utiliza a perspectiva de objetos
- Capaz de ser usado em qualquer tipo de sistema
- Principais objetivos são:
  - Melhorar a compreensão do sistema
  - Auto grau de reutilização
  - Facilidade de manutenção
  - Facilidade de evolução
  - Maior qualidade
  - Maior produtividade e menor custo
- Em contrapartida:
  - Maior curva de aprendizagem
  - Programas maiores
  - Não recomendável para qualquer tipo de problema

# **Conceitos básicos de Programação orientada a objetos (POO)**

# Fazendo uma analogia



Objetos



Classe



Objetos

# Classes e objetos

- Em programação orientada a objetos:
  - Um **objeto** geralmente representa um elemento do mundo real. Todo objeto pertence a uma classe.
  - Uma **classe** descreve as características comuns dos seus objetos.

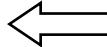
# Problema a ser resolvido

- Calcular o IMC (índice de massa corpórea) de Marta:



- Nome: Marta da Silva
- Idade: 21 anos
- Altura: 1,71 m
- Peso: 56 kg
- Cor preferida: verde
- Signo: aquário
- Naturalidade: Blumenau
- etc

*Formas de caracterizar Marta*



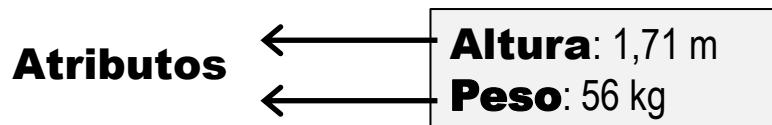
- Altura: 1,71 m
- Peso: 56 kg

*Formas uteis de caracterizar  
Marta para resolver este problema*

$$IMC = \frac{Peso}{Altura^2}$$

# Objetos – Atributos e operações

- Objetos são caracterizados por um conjunto de **atributos**.  
Exemplo: o objeto que representa a Marta é caracterizado através da altura e do peso.



- Afirmamos que os objetos possuem um estado. O estado corresponde ao valor de seus atributos

Altura: **1,71 m**  
Peso: **56 kg**

**Estado do objeto**

O estado do  
objeto pode  
mudar.

Altura: **1,71 m**  
Peso: **56,5 kg**

Novo estado do objeto

- Observar que o valor de um atributo é um dado.

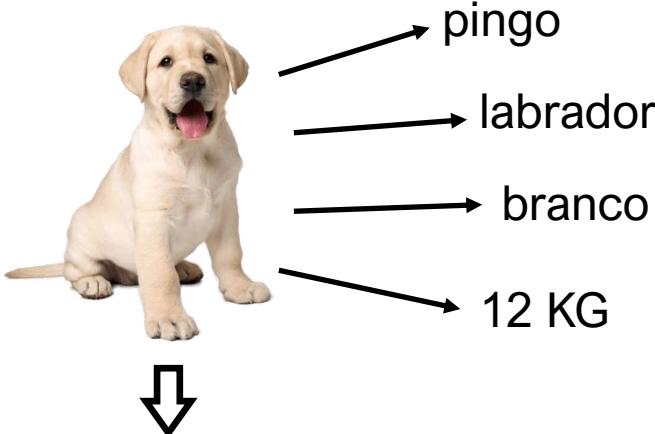
# Objetos – Atributos e operações

- “No desenvolvimento de software orientado a objetos, primeiro damos foco às estruturas de dados” (BAKER, 2005).
- Os dados estão contidos dentro do objeto e pertencem apenas aquele objeto.
- Além de dados, os objetos são capazes de executar operações
  - As operações podem executar alguma ação com os dados do próprio objeto.
  - Exemplo: o objeto que representa a Marta é capaz de calcular o IMC da Marta.

# Classe

- Todo objeto que se quer criar pertence a uma *classe* de objetos
- Através da classe definimos:
  - Quais atributos os objetos podem possuir
  - Quais operações os objetos podem realizar
- Toda classe possui um nome

# Exemplo

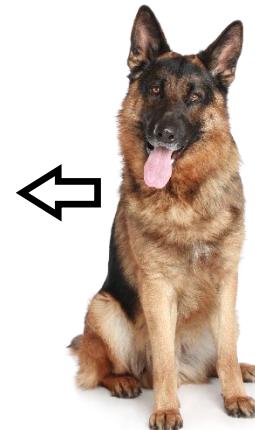
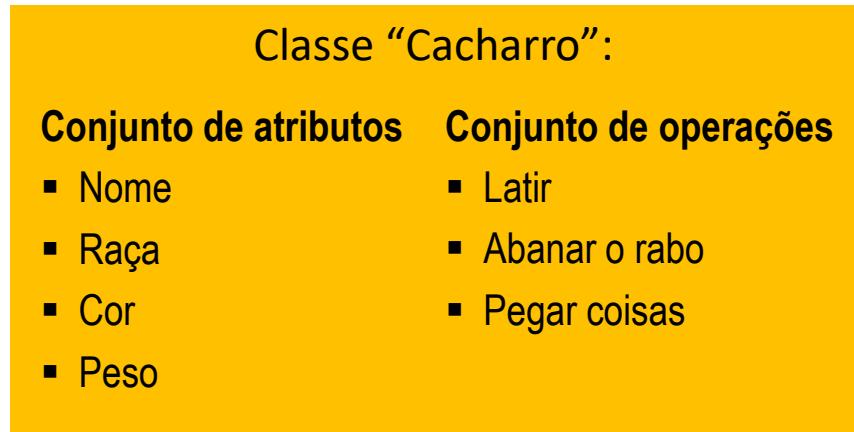


## Objeto 1 da classe Cachorro

- Nome: pingo
- Raça: labrador
- Cor: areia
- Peso: 12 KG

## Objeto 2 da classe Cachorro

- Nome: brutus
- Raça: pastor alemão
- Cor: marrom
- Peso: 21 KG



# Resumindo...

- Conforme (BARKER, 2005):
  - Objeto é uma construção de software que empacota *estado* (dados) e comportamento (funções) que representam uma abstração do mundo real;
  - Uma classe é uma abstração que descreve as características comuns de todos os objetos num grupo de objetos comuns;
  - Uma classe pode ser vista como sendo um modelo para criar objetos.

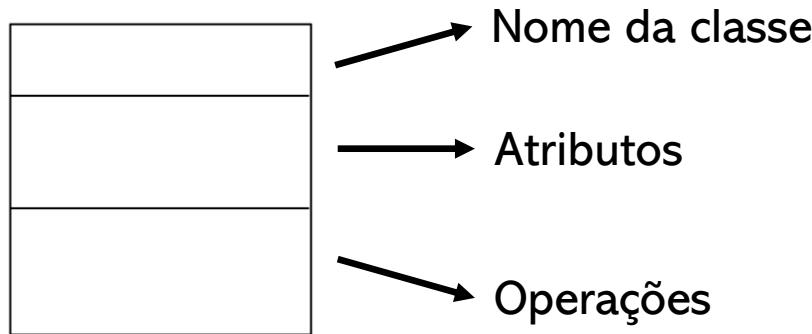
# Diagrama de classes

# UML

- A UML (Unified Modeling Language – linguagem de modelagem unificada) é uma linguagem visual utilizada para modelar sistemas computacionais por meio do paradigma de orientação a objetos
- É uma linguagem padrão de modelagem de software adotada internacionalmente pela indústria de Engenharia de Software
- Constituído de 14 diagramas
  - “Embora cada diagrama tenha sua utilidade, nem sempre é necessário modelar um sistema utilizando-se de todos os diagramas, pois alguns deles possuem funções muito específicas”

# Diagrama de classes

- O diagrama de classes mostra um conjunto de classes (entre outros elementos) e seus relacionamentos
- O diagrama de classes é usado para modelar uma visão estática de um software



- Regras aplicáveis para nomes de classes, atributos ou operações:
  - Não se utilizam espaços
  - Geralmente não se utiliza preposições

# Nome da classe

- São **substantivos** ou expressões breves, definidos a partir do vocabulário do sistema
- Utilizam o primeiro caractere em letra maiúscula para cada palavra existente no nome da classe (*estilo UpperCamelCase*)
- Exemplos de nomes de classes:
  - Cliente
  - Estudante
  - Venda
  - ContaBancaria
  - SensorTemperatura

# Atributos

- Geralmente é um **substantivo** ou expressão que representa alguma propriedade da classe
- Utilizam o primeiro caractere em letra maiúscula para cada palavra existente no nome, exceto para a primeira letra (estilo *lowerCamelCase*)
- Todo atributo possui um tipo de dado, que delimita os possíveis valores para o atributo. Os principais tipos em Java, são:

`boolean` Indica um tipo lógico, com dois valores possíveis  
(verdadeiro ou falso)

`int` Permite representar números inteiros

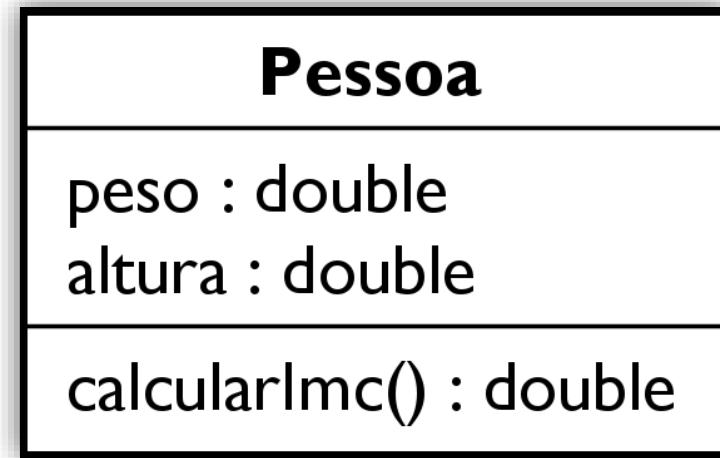
`double` Permite representar números decimais

`String` Permite representar texto

# Operações

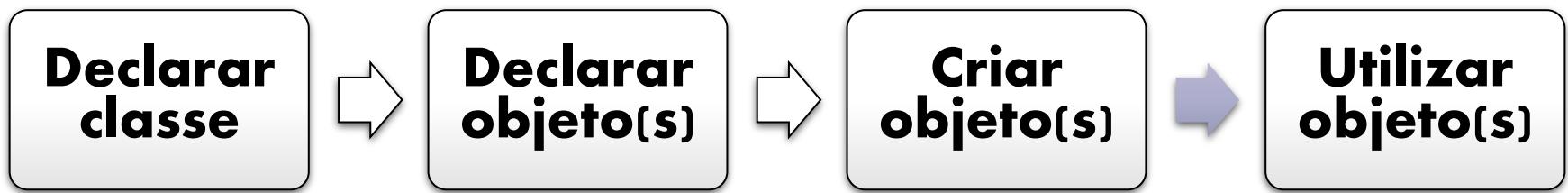
- O nome é um **verbo** ou locução verbal, representando algum comportamento (ação) da classe
- Utilizam o estilo *lowerCamelCase*
- Contém um par de parênteses após o nome
- Podem ser de dois tipos:
  - Mudam o estado do objeto
  - A partir do estado do objeto, realizam alguma operação, resultando num valor.

# Exemplo de diagrama de classes



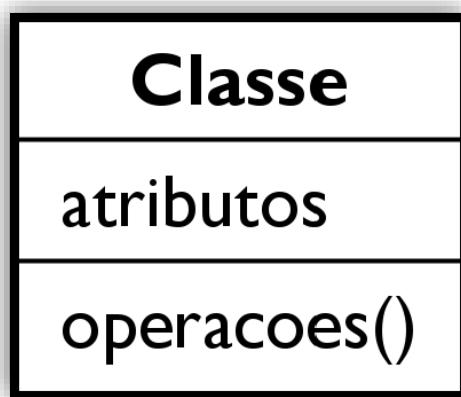
# **Implementação Classes e objetos**

# Utilização de POO



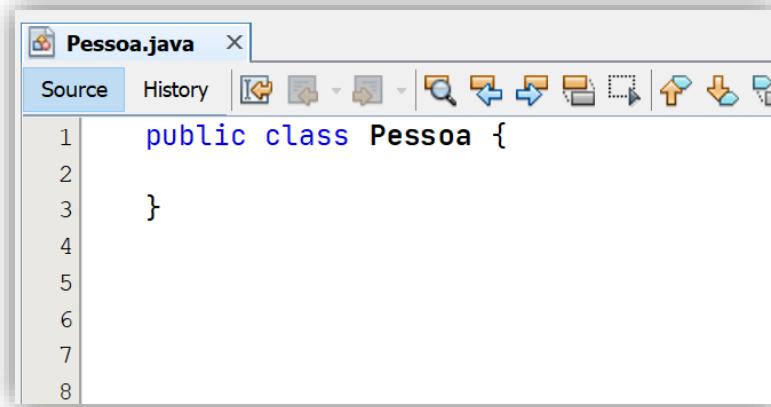
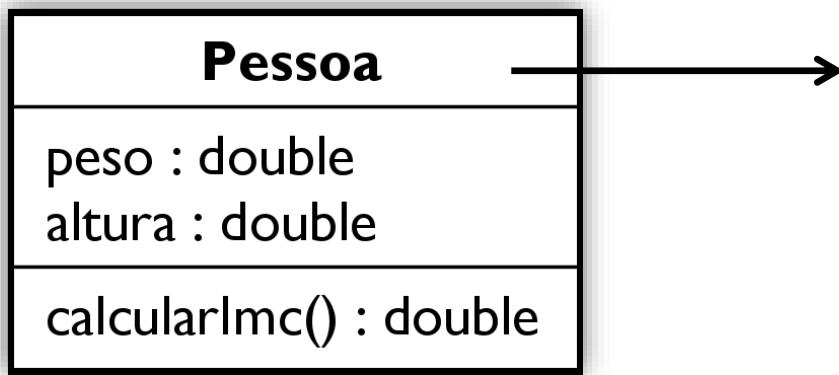
# **Declaração de classe**

# Elementos da classe



- **Uma classe Java (arquivo .java)**
- **Variáveis**
- **Métodos**

# Classe Java

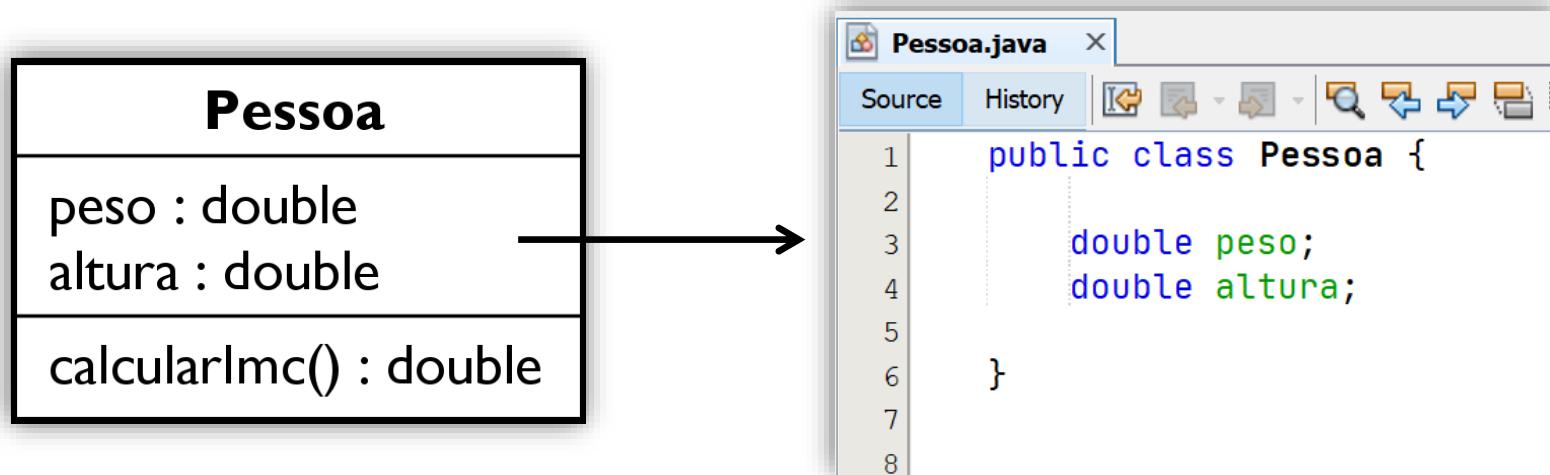


The screenshot shows a Java code editor window titled "Pessoa.java". The "Source" tab is selected. The code displayed is:

```
public class Pessoa {  
}
```

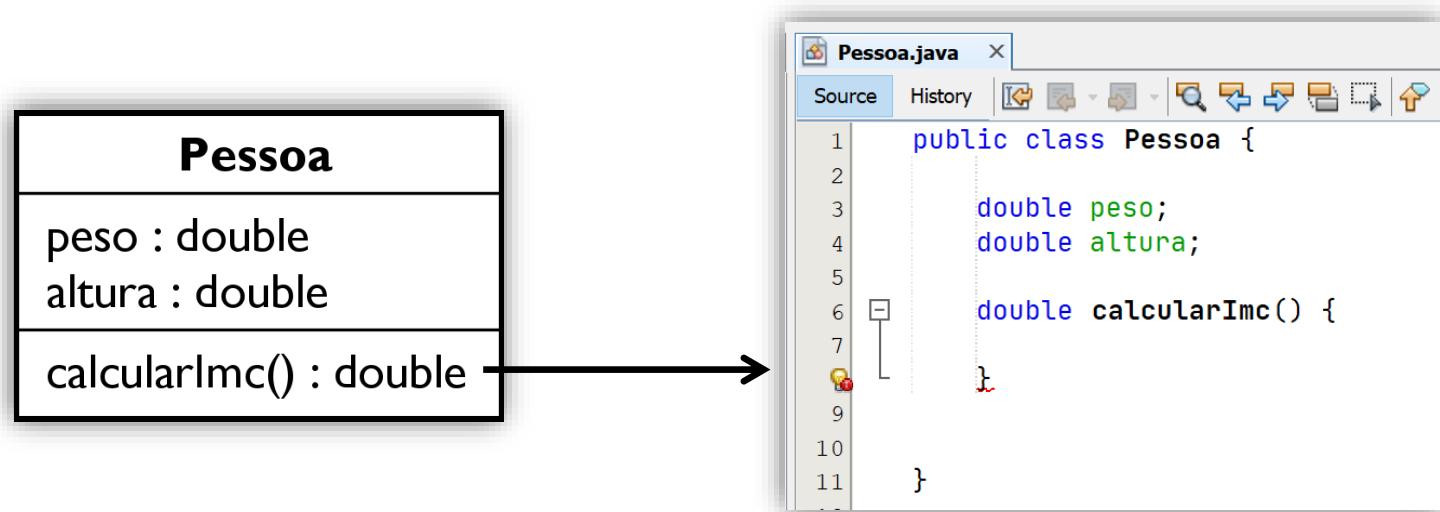
# Tradução dos atributos em variáveis de instância

- Em geral, os atributos da classe são traduzidos em Java para **variáveis de instância**.
- Sintaxe: *tipoDado nomeAtributo*



# Tradução das operações em métodos

- As operações da classe são traduzidas em Java em **métodos**
- Sintaxe: ***tipoDeDados nomeMétodo(parâmetros)***



- O corpo do método realiza alguma computação e pode:
  - Alterar o estado do objeto
  - Não alterar o estado do objeto, mas devolver um valor

# Métodos - devolvendo um valor

Método que não altera o estado do objeto, mas devolve um valor

```
12 public class Pessoa {  
13  
14     double altura;  
15     double peso;  
16  
17     double calcularImc() {  
18         return peso / (altura * altura);  
19     }  
20  
21 }
```

Métodos que retornam um valor devem declarar o tipo do dado devolvido

O corpo do método pode utilizar o valor dos atributos para realizar alguma computação

# Métodos – alterando estado do objeto

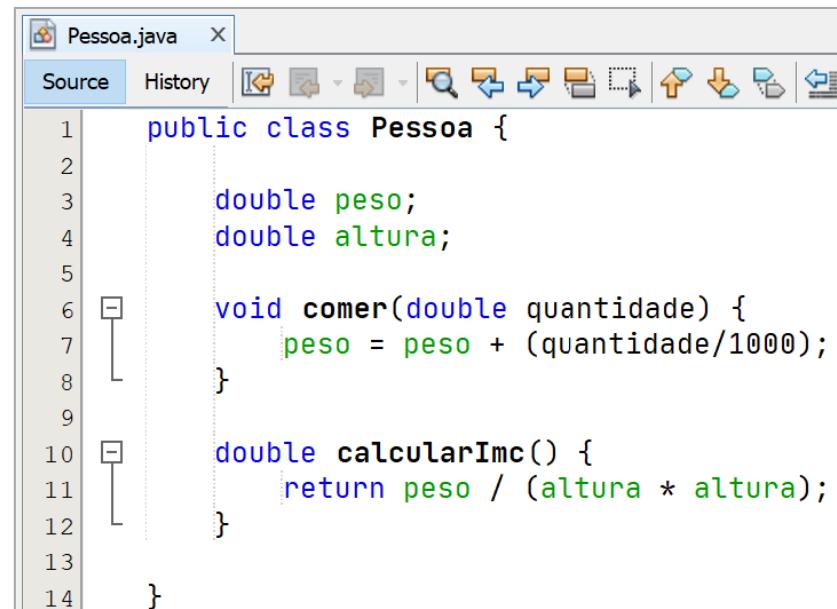
Talvez o método realiza alguma computação que causa a mudança de estado do objeto. Neste caso, geralmente não devolve um valor.

Pessoa
peso : double
altura : double

calcularImc() : double
comer(quantidade : double) : void

Como o método apenas alterará o estado do objeto, deve indicar que não devolverá valor algum (`void`)



```
Pessoa.java x
Source History
public class Pessoa {
    double peso;
    double altura;

    void comer(double quantidade) {
        peso = peso + (quantidade/1000);
    }

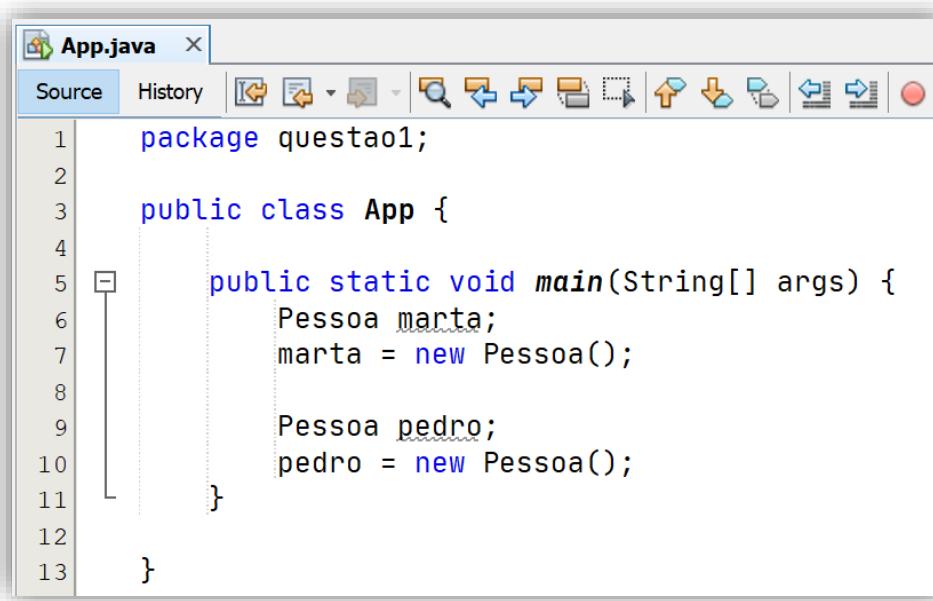
    double calcularImc() {
        return peso / (altura * altura);
    }
}
```

Método utilizado para indicar que a pessoa se alimentou de uma determinada quantidade que está expressa em gramas.

# **Declaração e criação de objetos**

# Declaração e criação de objetos

- Declaração de variável de referência, para acessar um objeto:  
Sintaxe: Classe nomeVariavel;
- Criação de objetos:  
Sintaxe: nomeVariavel = new Classe();



The screenshot shows a Java code editor window titled "App.java". The code is as follows:

```
1 package questao1;
2
3 public class App {
4
5     public static void main(String[] args) {
6         Pessoa marta;
7         marta = new Pessoa();
8
9         Pessoa pedro;
10        pedro = new Pessoa();
11    }
12
13 }
```

# Manipulação de objetos

# Manipulação de objetos

- Para alterar valor de atributos de um objeto ou solicitar a execução de métodos, deve-se utilizar o **operador de membro**
- O operador de membro é o ponto (.)
- A sintaxe de uso é:

```
identificador.membro
```

- Exemplo:

```
p1.altura = 1.70;  
p1.peso = 79;  
double imc = p1.calcularImc();
```

# Variável do tipo referência

```
Pessoa p1;  
p1 = new Pessoa();
```



Não é variável de tipo de dado primitivo.  
É uma variável do **tipo referência**

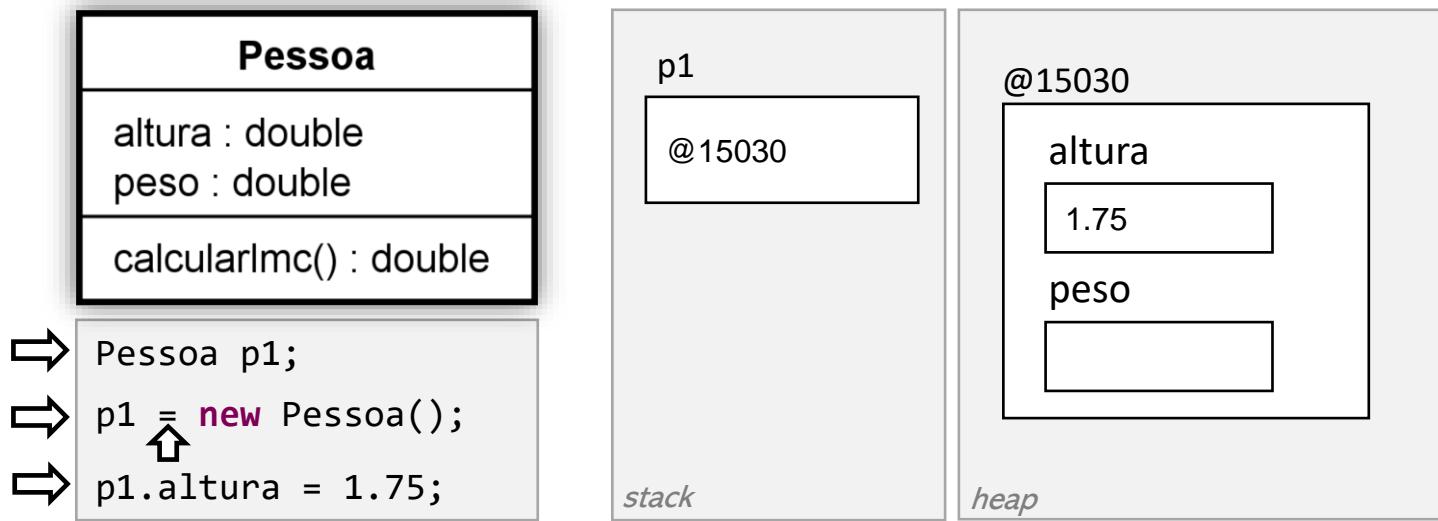
Variável do tipo referência armazena o endereço de um objeto.

# Áreas de Memória do Java

- **Stack**
  - Área de memória utilizada para armazenar:
    - variáveis locais
    - Variáveis paramétricas (parâmetros de métodos)
    - chamadas de funções
- **Heap**
  - Armazena os objetos

# O operador new

- O operador **new** cria objetos (instanciação de classe).
- O operador **new** faz três operações:
  - 1) Cria o objeto na memória, alocando espaço para armazenar valores para suas variáveis de instância;
  - 2) Inicializa as variáveis de instância
  - 3) Retorna o endereço de memória criado pelo objeto.



# **Comandos de leitura de texto e exibição de texto**

# Comando para **exibição** de texto

- **Exibição de texto no Console:**

Sintaxe: `System.out.println( <texto> )`

Exemplos:

```
System.out.println("Boa noite");
```

```
System.out.println(18);
```

# Comandos para leitura de texto

- **Leitura de texto no Console:**

Necessário obter acesso ao teclado, instanciando a classe Scanner:

```
Scanner teclado = new Scanner(System.in);
```

A partir deste momento, é possível ler dados através do método nextLine(), que *retorna* o texto digitado pelo usuário - nextLine() é uma função:

```
teclado.nextLine();
```

# Conversão de dados

- A partir de uma String, é possível obter o mesmo dado em outro formato:

Operação	Sintaxe	Exemplo
String → Integer	Integer.parseInt( <i>string</i> )	Integer.parseInt("15")
String → Double	Double.parseDouble( <i>string</i> )	Double.parseDouble("21.9")

# Exemplos

# Interação via Console

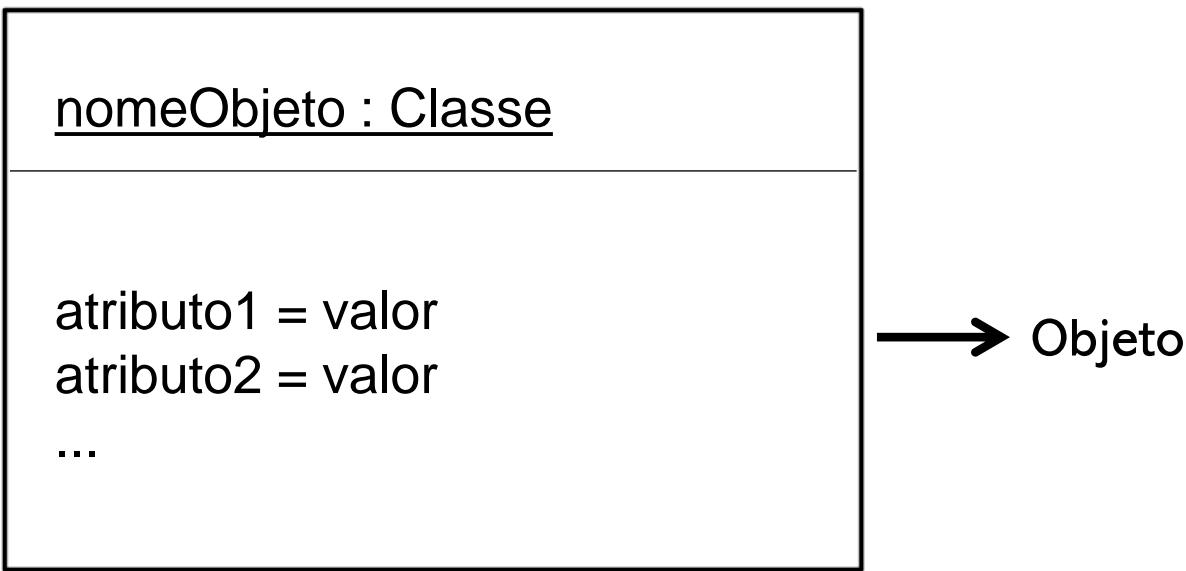
```
1 import java.util.Scanner;
2
3 public class ExemploConsole {
4
5     public static void main(String[] args) {
6
7         System.out.print("Informe seu nome: ");
8
9         Scanner teclado = new Scanner(System.in);
10        String nome = teclado.nextLine();
11
12        System.out.print("Digite sua idade: ");
13        int idade = Integer.parseInt(teclado.nextLine());
14
15        System.out.println("Obrigado " + nome + ", por informar " +
16                            "que você tem " + idade + " anos" );
17    }
18
19 }
```

# Diagrama de Objetos

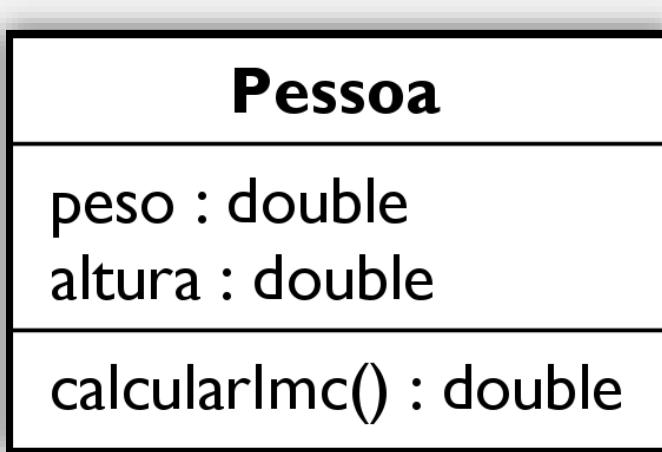
# Diagrama de Objetos

- É um diagrama que mostra uma fotografia do estado detalhado de um sistema, num determinado instante do tempo;
- É um diagrama da UML que contém somente objetos (não contém classes);
- Fornece uma perspectiva concreta de objetos e seus relacionamentos;
- Uso limitado, pois apresenta somente estruturas de dados

# Elementos do diagrama



# Exemplo – Questão 1



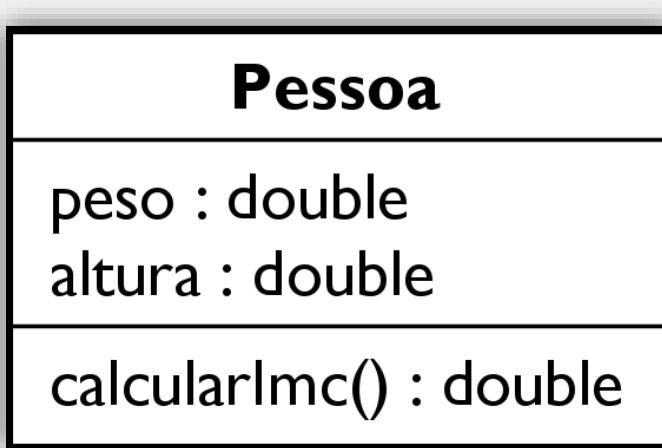
*Diagrama de classes*

marta : Pessoa

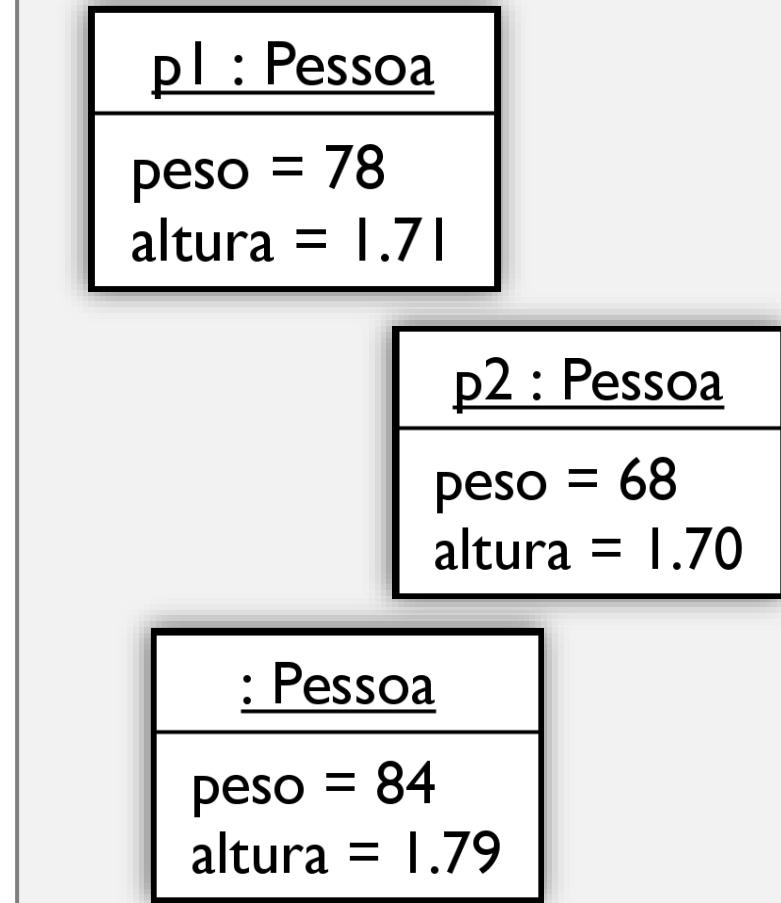
peso = 78  
altura = 1.71

*Diagrama de objetos*

# Exemplo



*Diagrama de classes*



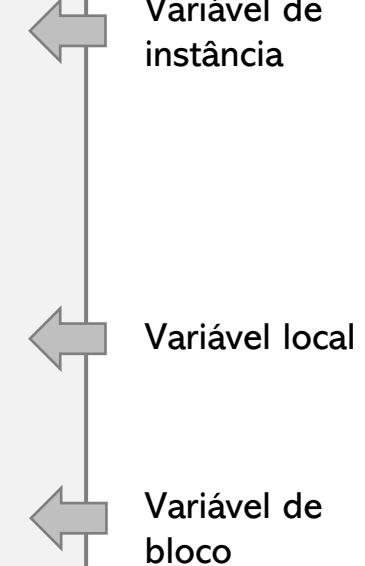
*Diagrama de objetos*

# **Escopo de Variáveis**

# Escopo de variáveis

- O escopo de uma variável denota sua visibilidade no programa, isto é, onde a variável é acessível
- Fora deste contexto, o identificador não pode ser utilizado, isto é, está fora do escopo da variável.

```
1 public class MinhaClasse {  
2  
3     int var1;  
4  
5     void metodoA() {  
6         var1 = 20;  
7     }  
8  
9     void metodoB() {  
10        String var2;  
11        var2 = "TESTE";  
12  
13        do {  
14            int var3 = 10;  
15        } while (false);  
16  
17        System.out.println(var2);  
18    }  
19  
20 }
```



# Escopo de variáveis de instâncias

O escopo da  
**variável de instância**  
são todos os métodos  
da classe

```
1 public class MinhaClasse {  
2  
3     int var1;  
4  
5     public void meuMetodoA() {  
6         var1 = 20;  
7     }  
8  
9     public void meuMetodoB() {  
10        String var2;  
11        var2 = "TESTE";  
12  
13        do {  
14            int var3 = 10;  
15        } while (false);  
16  
17        System.out.println(var2);  
18    }  
19  
20 }
```

*Escopo da variável var1*

# Escopo de variáveis locais

- A **variável local** é uma variável criada dentro do corpo de um método
- O escopo de variáveis locais é o próprio método em que a variável foi declarada.

```
1 public class MinhaClasse {  
2  
3     int var1;  
4  
5     void metodoA() {  
6         var1 = 20;  
7     }  
8  
9     void metodoB() {  
10        String var2;  
11        var2 = "TESTE";  
12  
13        do {  
14            int var3 = 10;  
15        } while (false);  
16  
17        System.out.println(var2);  
18    }  
19  
20 }
```

*Escopo da variável var2*

# Escopo de variáveis de bloco

- Uma variável de bloco é uma variável criada dentro de um bloco num método
- O escopo da variável de bloco é o bloco onde a variável foi declarada

```
1 public class MinhaClasse {  
2  
3     int var1;  
4  
5     void metodoA() {  
6         var1 = 20;  
7     }  
8  
9     void metodoB() {  
10        String var2;  
11        var2 = "TESTE";  
12  
13        do {  
14            int var3 = 10;  
15        } while (false);  
16  
17        System.out.println(var2);  
18    }  
19  
20 }
```

*Escopo da variável var3*

# Inicialização de variáveis

- As variáveis declaradas num método ou num bloco não possuem valor inicial
  - Somente é possível ler o valor da variável depois de atribuir explicitamente um valor para a variável
- As variáveis de instância têm valor padrão. Java automaticamente inicializa (atribui um valor) para estas variáveis:
  - Variáveis numéricas são inicializadas com 0 (zero)
  - Variáveis booleanas são inicializadas com false
  - Variáveis de referência são inicializadas com null

# Escopo de variáveis

É possível declarar duas variáveis com mesmo nome mas escopos diferentes.

```
1 public static void main(String[] args) {  
2  
3     {  
4         int x = 0;  
5         System.out.println(x);  
6     }  
7  
8     {  
9         String x = "10";  
10        System.out.println(x);  
11    }  
12  
13 }
```

# Escopo de variáveis

- É possível utilizar, num contexto, um mesmo identificador para duas variáveis, sendo uma delas, variável local, e a outra, variável de instância

Neste caso, a linguagem  
utiliza a variável  
declarada dentro do  
bloco

```
1 public class Classe1{  
2  
3     int var1 = 20;  
4  
5     void exibir(int a) {  
6         int var1;  
7         var1 = 10;  
8  
9         System.out.println(var1);  
10    }  
11  
12 }  
13 }
```

# Encapsulamento

# Encapsulamento - Motivação

Considerar a classe abaixo, a ser utilizada para representar contas bancárias:

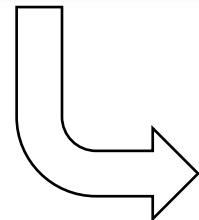
<b>ContaBancaria</b>
titular : String saldo : double
depositar(valor : double) : void sacar(valor : double) : void

# Encapsulamento – Motivação

## ContaBancaria

titular : String  
saldo : double

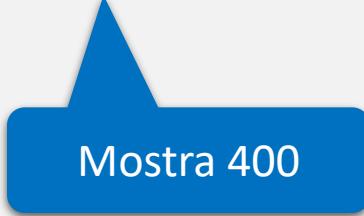
depositar(valor : double) : void  
sacar(valor : double) : void



```
1 public class ContaBancaria {  
2  
3     String titular;  
4     double saldo;  
5  
6     void depositar(double valor) {  
7         saldo = saldo + valor;  
8     }  
9  
10    void sacar(double valor) {  
11        saldo = saldo - valor;  
12    }  
13  
14 }
```

# Encapsulamento - Motivação

```
1 public class CaixaEletronico {  
2  
3     public static void main(String[] args) {  
4  
5         ContaBancaria conta1 = new ContaBancaria();  
6         conta1.titular = "Sandro da Silva";  
7         conta1.depositar(500);  
8         conta1.sacar(100);  
9         System.out.println(conta1.saldo);  
10    }  
11  
12 }  
13 }
```



Mostra 400

# Encapsulamento - Motivação

```
1 public class CaixaEletronico {  
2  
3     public static void main(String[] args) {  
4  
5         ContaBancaria conta1 = new ContaBancaria();  
6         conta1.titular = "Sandro da Silva";  
7         conta1.depositar(500);  
8         conta1.sacar(100);  
9         conta1.saldo = 10000;  
10    }  
11  
12 }  
13 }
```

Este comando permite que seja definido um valor de saldo, sem que seja feito um depósito correspondente

# Encapsulamento

- O acesso ao atributo deve ser “controlado”, para garantir integridade dos dados
  - Isto é, o estado do objeto precisa ser “controlado”
- Somente o próprio objeto deveria manipular o valor de seus atributos
- Esta técnica se chama **encapsulamento** de dados
- Em Java, para aplicar o encapsulamento é preciso tornar o atributo **privado**.

# Encapsulamento

Em UML, para expressar que um atributo é encapsulado, será apresentado um sinal de “-” na frente do atributo, como no exemplo:

<b>ContaBancaria</b>
<ul style="list-style-type: none"><li>- titular : String</li><li>- saldo : double</li></ul>
<ul style="list-style-type: none"><li>depositar(valor : double) : void</li><li>sacar(valor : double) : void</li></ul>

# Encapsulamento

Outros símbolos podem ser utilizados nos membros de uma classe (atributos e operações) para indicar seu grau de visibilidade:

ContaBancaria	Símbolo UML	Nome	Palavra reservada*	Significado
- titular : String - saldo : double	-	Privado	private	Somente visível pela própria classe
depositar(valor : double) : void sacar(valor : double) : void	+	Público	public	Visível para qualquer classe
	#	Protegido	protected	Estudaremos mais tarde
	~	“de pacote”	(ausência de símbolo)	Estudaremos mais tarde

\* Conhecido também como “modificador de acesso”

Sintaxe para traduzir um atributo em Java:

Modificador de acesso tipo de dado identificador;

Exemplo:

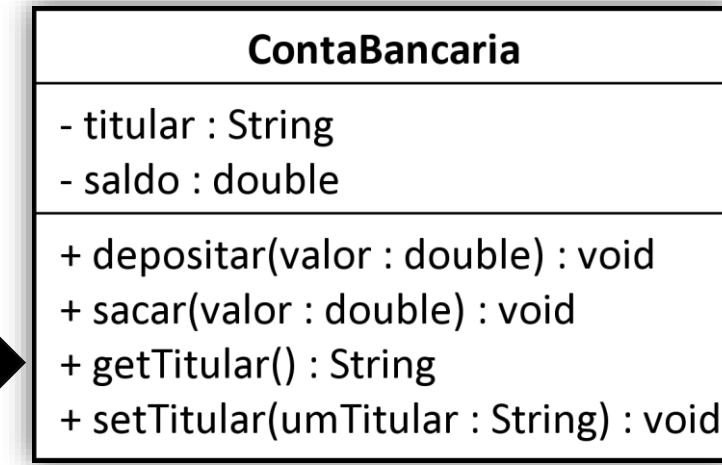
`private double saldo;`

# Encapsulamento – Métodos de acesso

- Todos os atributos de um objeto deveriam ser encapsulados.
- Os atributos que precisam ser acessados por outras classes poderão ser acessados por meio de **métodos de acesso**.
- Os métodos de acesso são métodos, geralmente públicos. Dividem-se em:
  - **Getters**: métodos usados para recuperar valor de atributos de um objeto da classe. O nome de um *método getter* deve ser escrito com o prefixo *get*, seguido do nome do atributo com a inicial maiúscula.  
Exceção: se o atributo for lógico (booleano), usar o prefixo “is”.
  - **Setters**: métodos usados para atribuir valor de atributos da classe. O nome de um método *setter* deve ser escrito com o prefixo *set*, seguido do nome do atributo com a letra inicial maiúscula.

# Exemplo

Método *getter* do  
atributo titular



Método *setter* do  
atributo titular

Métodos *getter* :

- nunca têm parâmetro
- sempre são do tipo função
- sempre retornam um dado cujo tipo é igual ao da variável em que é *getter*

Métodos *setter* :

- sempre têm um parâmetro
- sempre são do tipo procedimento
- o parâmetro deve ser declarado com um tipo de dado igual ao da variável que é *setter*

# Exemplo de *getter* e *setter*

```
public class ContaBancaria {

    private String titular;
    private double saldo;

    void depositar(double valor) {
        saldo = saldo + valor;
    }

    void sacar(double valor) {
        saldo = saldo - valor;
    }

    public void setTitular(String novoTitular) {
        titular = novoTitular;
    }

    public String getTitular() {
        return titular;
    }
}
```

# this

- **this** refere-se ao objeto corrente – o objeto na qual o método foi chamado.
- O principal motivo em usar a palavra **this** é quando há algum parâmetro de método que possui o mesmo nome de uma variável de instância.
  - Quando há dois identificadores com o mesmo nome, por padrão Java sempre reconhece que o identificador utilizado no comando é aquele com menor escopo

# Exemplo de *getter* e *setter*

```
public class ContaBancaria {  
  
    private String titular;  
    private double saldo;  
  
    void depositar(double valor) {  
        saldo = saldo + valor;  
    }  
  
    void sacar(double valor) {  
        saldo = saldo - valor;  
    }  
  
    public void setTitular(String titular) {  
        this.titular = titular;  
    }  
  
    public String getTitular() {  
        return titular;  
    }  
}
```

# Encapsulamento

- Sempre dar preferência por encapsular todos os atributos de uma classe
- Somente é admissível utilizar **public** para constantes
- Se for necessário expor o valor de atributo para outros objetos/classes, implementar um método *getter* para o atributo
- Se for necessário permitir que outros objetos/classes definam o valor de um atributo, é necessário implementar um método *setter* para o atributo

# Encapsulamento de métodos

- Ao utilizar POO, é possível ocultar a complexidade do trabalho interno executado pelo objeto:
  - Criar uma forma simplificada e comprehensível de utilizar o objeto – favorece a reutilização
  - Por exemplo: o motorista não precisa compreender como o mecanismo interno de combustão funciona para ligar o carro.

# Lançamento de exceções

# Exceções

- Uma exceção é um evento, que ocorre durante a execução do programa, que interrompe o fluxo normal de execução.
- Quando uma operação incorreta é identificada dentro de um método, o método pode criar um objeto de uma classe que caracteriza o erro e notificá-lo ao sistema
  - Este objeto é denominado de “objeto de exceção”
  - O objeto de exceção contém informação sobre o erro
  - Esta operação (criar objeto e notificar o sistema) é conhecido como “lançamento de exceção”
- O efeito de uma exceção lançada é (por enquanto) abortar a execução do programa
  - Não poderá existir comandos, no mesmo método, após a instrução que lança a exceção

# Lançamento de exceções em Java

- Sintaxe:

```
throw new RuntimeException("mensagem");
```

- Onde:

- Mensagem: indica uma mensagem que pode ser apresentada quando a exceção for gerada

- Exemplo:

```
public void setSalario(double novoSalario) {  
    if (novoSalario < 0) {  
        throw new RuntimeException("Salário incorreto");  
    }  
    salario = novoSalario;  
}
```

# Membros de classe

# Membros de classe

- São membros (variáveis ou métodos) que pertencem à classe.  
Não pertencem a nenhuma instância em particular.
- Membros de classe podem ser utilizados sem que haja uma instância da classe
- No diagrama de classes, os membros de classe são sublinhados.

Classe
<u>- atributo1 : int</u> <u>- atributo2 : int</u>
<u>+ operacao1() : void</u> <u>+ operacao2() : void</u>

# Variáveis de classe

- Variável de classe:
  - Uma variável que é comum à todas as instâncias
    - Uma variável que é compartilhada entre todas as instâncias
  - Podem ser manipulados sem que haja uma instância
    - Para acessar utilizar a sintaxe: `classe.identificador`
  - Também chamados de “variáveis estáticas” ou “campos de classe”.
- Sintaxe:

`Modificador static tipo de dado identificador;`
- Exemplo:

`private static int atributo1;`

# Métodos de classe

- Métodos de classe:
  - Podem manipular variáveis de classe;
  - Não podem manipular variáveis de instância sem que haja uma instância explícita;
  - Não podem reusar métodos de instância;
  - Não podem utilizar a palavra `this`.
- Sintaxe:

```
Modificador static tipo de dado identificador(parametros);
```

# Exemplos de membros estáticos

- Integer.MAX\_VALUE
- Math.sqrt()
- Math.abs()
- Math.max()
- JOptionPane.showInputDialog()
- JOptionPane.showMessageDialog()

# Sobrecarga de métodos

# Sobrecarga de métodos

- A linguagem Java suporta a sobrecarga de métodos, isto é, implementação de vários métodos com mesmo nome.
- Os métodos devem ter assinaturas diferentes
  - Métodos podem ter mesmo nome se a lista de parâmetros for diferente.
  - O compilador não considera o tipo de retorno para diferenciar o método. Por isso, dois métodos com a mesma assinatura mas retornos distintos não podem ser implementados na mesma classe
- Deve ser utilizado com moderação pois pode tornar o código menos legível

# Construtores

# Construtores

- São similares à métodos, com exceção de que são invocados exclusivamente durante a criação de objetos
- Geralmente utilizados para “inicializar” um objeto
- A declaração de construtor é semelhante à declaração de métodos, porém não possuem tipo de dado de retorno e seu identificador é igual ao da classe
- Não é preciso criar construtor para a classe. Quando não é implementado um construtor, o compilador automaticamente fornece um construtor padrão.
  - Um “construtor padrão” é um construtor sem argumentos.

# O operador new

- O operador **new** faz então quatro operações:
  1. Cria o objeto na memória, alocando espaço para armazenar valores para suas variáveis de instância
  2. Inicializa as variáveis de instância
  3. Executa o construtor que foi utilizado no operador **new**
  4. Retorna o endereço de memória criado pelo objeto.

# Exceções

# Exceções

- Uma exceção é um evento, que ocorre durante a execução do programa, que interrompe o fluxo normal de execução.
- Quando uma operação incorreta é identificada dentro de um método, o método pode criar um objeto de uma classe que caracteriza o erro e notificá-lo ao sistema
  - Este objeto é denominado de “objeto de exceção”
  - O objeto de exceção contém informação sobre o erro
  - Esta operação (criar objeto e notificar o sistema) é conhecido como “lançamento de exceção”
- O efeito de uma exceção lançada é (por enquanto) abortar a execução do programa
  - Não poderá existir comandos, no mesmo método, após a instrução que lança a exceção

# Lançamento de exceções em Java

- Sintaxe:

```
throw new RuntimeException("mensagem");
```

- Onde:

- Mensagem: indica uma mensagem que pode ser apresentada quando a exceção for gerada

- Exemplo:

```
public void setSalario(double novoSalario) {  
    if (novoSalario < 0) {  
        throw new RuntimeException("Salário incorreto");  
    }  
    salario = novoSalario;  
}
```

# Interface gráfica de usuário

# Introdução

- Uma **interface gráfica de usuário** (GUI – *Graphics User Interface*) permite ao usuário interagir com a aplicação utilizando ícones e desenhos, ao contrário de interfaces baseadas apenas em texto
  - Fornecem apresentação mais amigável ao usuário.
- GUIs podem ser construídas reutilizando componentes de bibliotecas gráficas.
  - Os componentes também são conhecidos por “controles GUI” ou simplesmente “controles”;
- As principais bibliotecas nativas do Java para construção de GUIs são:
  - AWT
  - Swing
  - JavaFX

# AWT



AWT – Design e comportamentos distintos entre os sistemas operacionais

# Netbeans GUI Builder

- Ferramenta do Netbeans que permite arrastar e posicionar *componentes GUI* de uma **paleta de componentes** para uma **área de desenho**
- Cada componente GUI inserido na área de desenho é um objeto Java. A classe do objeto depende do tipo de componente utilizado
- O GUI Builder procura alinhar automaticamente os componentes inseridos na área de desenho

# Acionando o GUI Builder

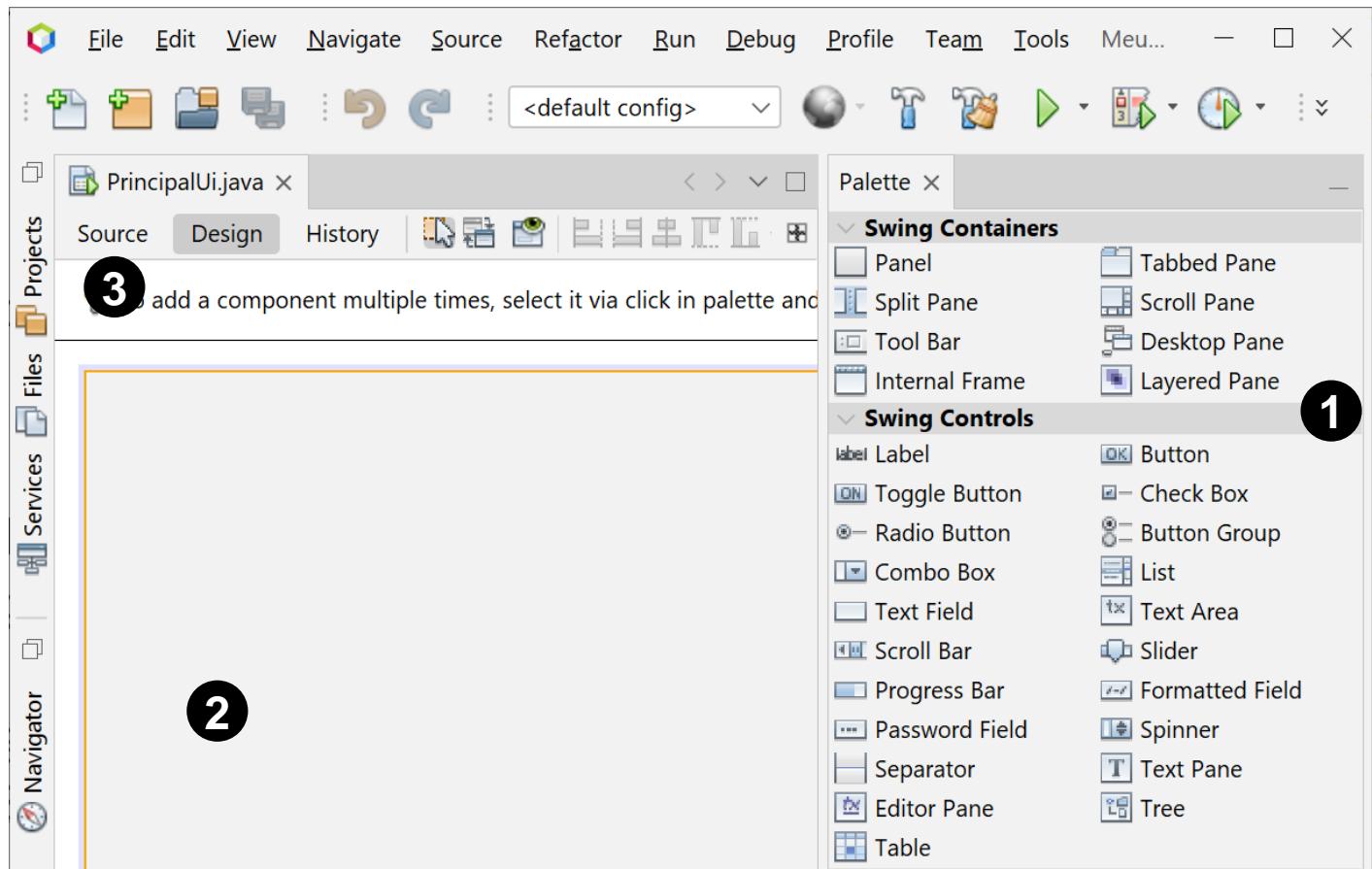
- Para criar uma GUI:
  - Clicar em *File > New File*
  - Em *Categories*, selecionar ***Swing GUI Forms***
  - Em *Tipos de Arquivo*, selecionar ***JFrame Form***
  - Clicar em ***Next***
  - Em ***Nome da Classe*** informar um nome para a classe da GUI

# Utilizando o GUI Builder

1 – Selecione um componente na *Paleta de Componentes*

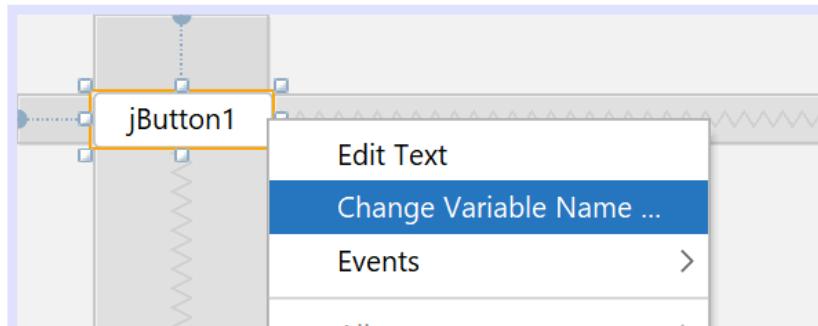
2 – Clique sobre a *área de desenho* para incluir o componente previamente selecionado

3 – Clique em *Source* para visualizar o código Java gerado pelo GUI Builder. Para retornar ao desenho da GUI, clique em *Design*



# Utilizando o GUI Builder

Como cada componente GUI constitui de um objeto Java, é possível definir o nome de cada variável que referencia o objeto



- 1 - Clicar com o botão direito sobre o componente
- 2 - Selecionar *Change Variable Name*
- 3 - Informar um novo nome de variável

É altamente recomendável utilizar nomes legíveis para as variáveis que referenciam os objetos de controles GUI.

# Alguns componentes GUI

Nome do componente na Paleta de Componentes do GUI Builder	Classe do Componente
label Label	JLabel
Text Field	JTextField
Button	JButton
Check Box	JCheckBox
Combo Box	JComboBox
List	JList
Panel	JPanel
Table	JTable

# Programação orientada a eventos

- É um estilo de programação na qual o fluxo de execução do programa é determinado por eventos
- Um evento é um sinal recebido pelo programa indicando que algo aconteceu
- São considerados eventos:
  - **Ações do usuário: movimentos do mouse, teclado**
  - Mensagens de outros programas
  - Periféricos enviando sinais ao programa
- O programa pode escolher responder ou ignorar o evento

# Programando um evento

- Na *área de desenho*, ao clicar duplo sobre um *botão*, o *GUI Builder* cria um método para inserção do programa que será executado quando o usuário clicar sobre o botão.
- Exemplo:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
}
```

Digitar aqui o que  
deve acontecer  
quando o botão for  
clicado pelo usuário

# JLabel

JLabel
- text : String
+ setText(text : String) : void
+ getText() : String

label Label

Método	Descrição
getText()	Obtém o valor exibido no componente
setText()	Altera o valor exibido pelo componente

# JTextField

<b>JTextField</b>
- text : String
+ setText(text : String) : void
+ getText() : String
+ setEnabled(edt : boolean) : void



Método	Descrição
getText()	Obtém o valor exibido no componente
setText()	Altera o valor exibido pelo componente
setEnabled()	Habilita ou desabilita a edição

# JCheckBox

JCheckBox
 Check Box
+ isSelected() : boolean
+ setSelected(selected : boolean) : void
+ setEnabled(enabled : boolean) : boolean

Método	Descrição
isSelected()	Retorna true se o componente estiver marcado
setSelected()	Marca ou desmarca a caixa de seleção
setEnabled()	Habilita ou desabilita a edição

# **Arquitetura de software em camadas**

# Bibliografia

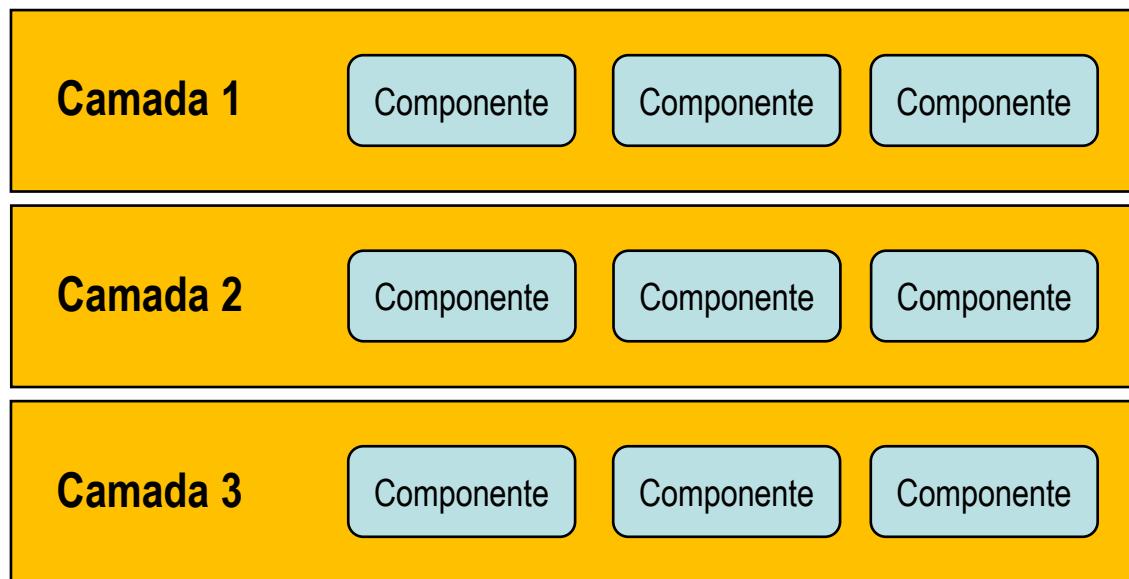
1. IEEE Computer Society, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems: IEEE Std 1471-2000. 2000.
2. CARNEGIE MELLON UNIVERSITY. Community Software Architecture Definitions. **Software Engineering Institute**. Disponível em:  
[<http://www.sei.cmu.edu/architecture/start/glossary/community.cfm>](http://www.sei.cmu.edu/architecture/start/glossary/community.cfm).

# Arquitetura de software

- A arquitetura de software descreve a forma como os componentes que compõe o software se relacionam.
- A arquitetura de software é “a organização fundamental de um software materializada pelos seus componentes, seus relacionamentos e o ambiente, e os princípios que regem a sua concepção e evolução” [1].
- Em OOP, é popular a arquitetura conhecida como “arquitetura em camadas”

# Arquitetura em camadas

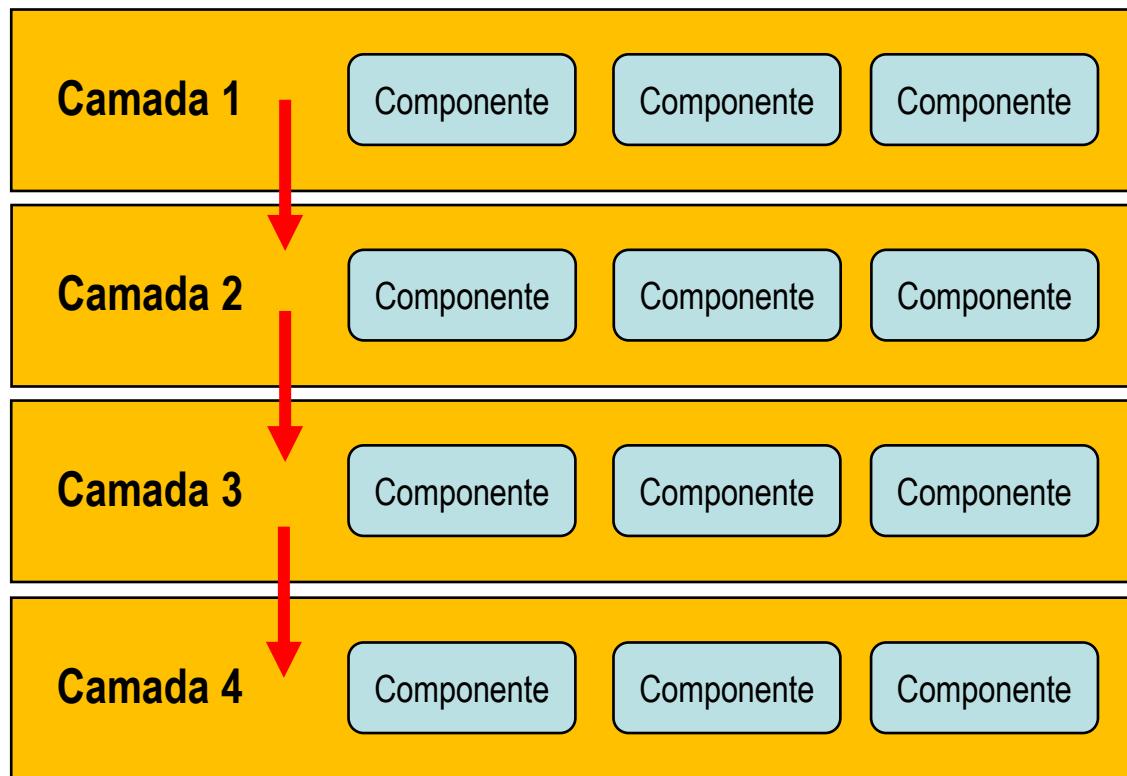
- Na “arquitetura de software em camadas” o software é organizado em camadas horizontais.
- Cada camada possui uma função específica na aplicação.



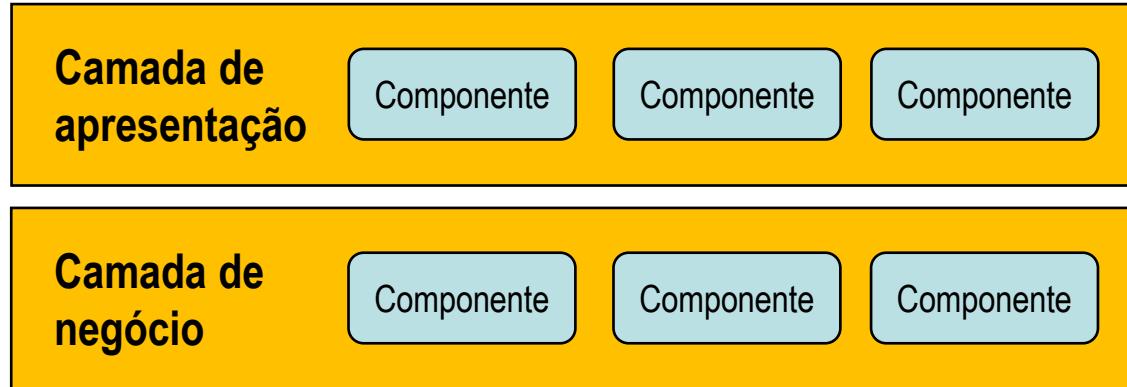
- Não há um número determinado de camadas

# Arquitetura em camadas

- Em geral, as camadas são fechadas de forma que a camada conhece apenas a camada imediatamente inferior.



# Exemplo de arquitetura em duas camadas



- Camada de apresentação
  - Responsável por se comunicar com o usuário, exibindo informações e lidando com as ações do usuário.
- Camada de negócio
  - Lida com as regras de negócio.
  - Desconhece como funciona a camada de apresentação

# Exemplo de arquitetura em camadas

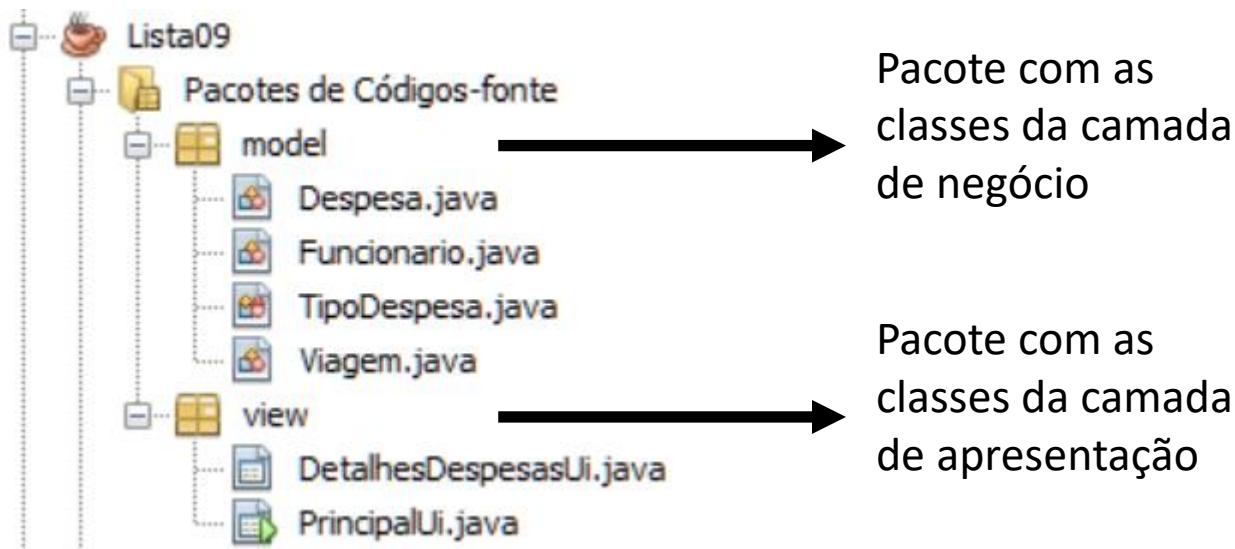


# Vantagens da arquitetura em camadas

- Modularização
  - Princípio denominado “separação de conceitos”.
  - Cada camada mantém o foco num aspecto.
    - Os componentes dentro de uma camada se preocupam com a lógica pertinente àquela camada.
- Torna fácil desenvolver, testar e manter a aplicação

# Organização de projetos Java

- Alguns desenvolvedores organizam as classes Java em pacotes de acordo com as camadas da arquitetura.
- Exemplo:



# **Qualidade de Software e Testes de Unidade**

# Bibliografia

- IBM. Minimizing code defects to improve software quality and lower development costs, 2008. Disponível em: <<ftp://ftp.software.ibm.com/software/rational/info/documents/RW14109USEN.pdf>>. Acesso em: 2018.
- PRESSMAN, R. S. **Engenharia de software: Uma Abordagem Profissional.** 7a. ed. Porto Alegre: Bookman, 2011.
- SHALLOWAY, A.; Trott, J. R. **Explicando padrões de projeto:** uma nova perspectiva em projeto de orientado a objeto. Porto Alegre: Bookman, 2004.

# Impactos de falhas de software

- **Falha em software deixou supermercados fechados**

QUA, 07 DE MARÇO DE 2012 17:00 ACESSOS: 433



Curtir



Seja o primeiro de seus amigos a curtir isso.

Um problema no software utilizado por alguns supermercados de Linhares causou transtorno nesta quarta-feira (07), quando simplesmente parou de funcionar. As lojas permaneceram fechadas até às 10h40.

De acordo com informações de funcionários destes estabelecimentos o software que é usado nos caixas dos supermercados deu uma “pane” e parou de funcionar.

- Máquina de terapia radiológica Therac-25

# Custos de correção



Quanto mais cedo os erros forem detectados no ciclo de desenvolvimento, é possível reduzir de forma significativa o custo para desenvolver o software

Fonte: (IBM, 2008)

\*X é uma unidade de medida de custo que pode ser expressa em termos de horas, ou reais, etc.

# Qualidade

- Conceito de “qualidade” usado pela indústria:
  - “Um produto com qualidade é um produto que cumpre com sua especificação” (CROSBY, 1979)
- Teste de software
  - Teste é a execução do software de maneira controlada para avaliar se ele se comporta ou não conforme o especificado.

# Plano de testes

- Testes precisam ser planejados
  - Construção de um “plano de testes”
- Um plano de testes é um documento para registrar formalmente o planejamento
- O plano de testes pode conter: propósito, identificação, itens a serem testados, critérios de aceite, documentos produzidos (como logs, relatórios, etc.), ambiente a ser testado.
- Um plano de testes contém um conjunto de “casos de testes”
  - Um caso de teste possui uma descrição de um teste particular
  - Cada caso possui um conjunto de dados de entrada diferente.

# Tipos de Testes

- Testes de Sistema
  - Execução do sistema sob o ponto de vista do usuário, indo em busca de falhas em relação aos requisitos propostos.  
Os testes são executados em condições próximas da realidade do usuário, isto é, ambiente de hardware, restrições de recursos, volume de dados, etc.
- Testes de Unidade (ou testes unitários)
  - Testam-se unidades, isto é, partes individuais (funcionalidades pequenas e específicas) de software.
    - Em OOP, a “unidade” é o método
  - Realizado por desenvolvedores, para verificar se o software faz o que deve fazer
  - Vamos construir testes de unidade para testar classes da camada de negócio

# Exemplo de plano de testes

- Considerar a classe:

Calculadora
+ somar(num1 : int, num2 : int) : int + subtrair(num1 : int, num2 : int) : int + multiplicar(num1 : int, num2 : int) : int + dividir(num1 : int, num2 : int) : int

- Um plano de testes:

Plano de testes PL01 – Validar funcionamento da classe Calculadora			
Caso	Descrição	Entrada	Saída esperada
1	Validar soma de números inteiros	Somar números 10 e 75	Deve resultar em 85
2	Validar subtração de inteiros	Subtrair 35 de 100	Deve resultar em 65
3	Validar multiplicação de números inteiros	Multiplicar 5 por 8	Deve resultar em 40
4	Validar divisão de números inteiros	Dividir 10 por 2	Deve resultar em 5.

# **Testes de unidade com JUnit**

# Bibliografia

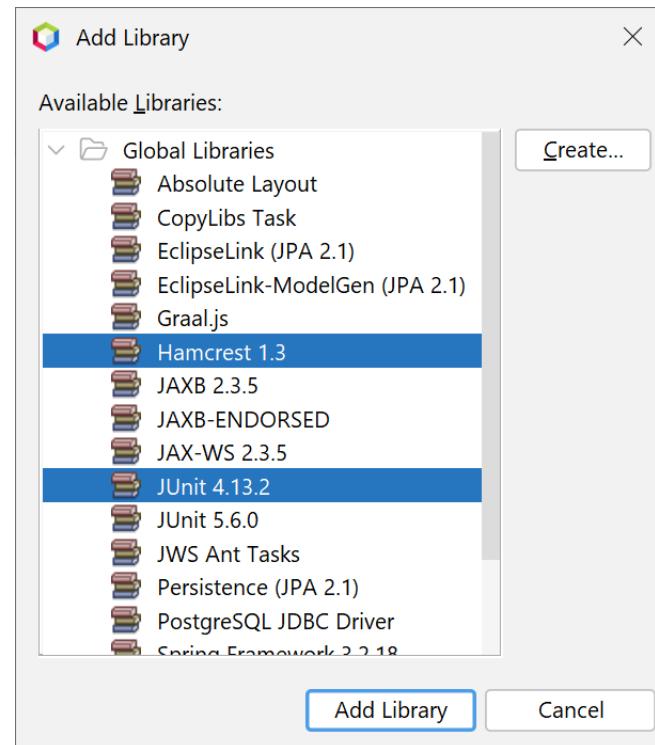
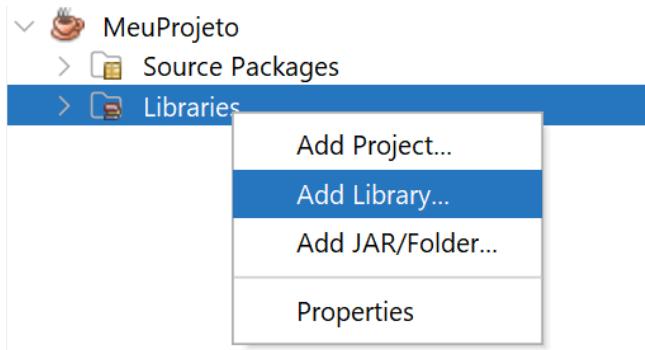
- MASSOL, V.; HUSTED, T. **JUnit em Ação**. Nova York: Manning Publications, 2003.
- HIGHTOWER, R.; LESIECKI, N. **Java Tools for Extreme Programming: Mastering Open Source Tools, Including Ant, JUnit, and Cactus**. New York: Wiley Computer Publishing, 2002.
- OBJECT MENTOR. **JUnit.org - Resources for Test Driven Development**. Disponivel em: <<http://www.junit.org/>>. Acesso em: 15 dez. 2011.

# Sobre o JUnit

- JUnit é um framework para execução de teste de unidade em Java
- Criado em 1997 por Erich Gamma e Kent Beck
- Hospedado no Source Forge
- Distribuído sob a licença IBM Common Public Lisence
- Modelo adotado por outras linguagens
- Possui três implementações (versão 3, versão 4 e versão 5)

# Como testar com Junit no NetBeans

- Para habilitar a execução de testes, clicar com o botão direito em “Libraries” e selecionar “Add Library”
- Selecionar as bibliotecas:
  - Hamcrest 1.3 e
  - JUnit 4.13.2



# Como testar com JUnit no Netbeans

- Clicar em Arquivo > Novo arquivo
- Em “Categoria”, selecionar “Unit Tests”
- Em “File Types”, selecionar “Test for Existing Class”
- Selecionar a classe a ser testada
- Desmarcar todas as caixas de seleção
- Clicar em Finalizar

# Testes com JUnit

- Como vamos implementar testes para a camada de negócio, haverá uma classe “paralela” (de teste) para cada classe a ser testada
- Em geral, o nome desta classe é igual ao nome da classe a ser testada, porém com sufixo “Test”
- Para cada “caso de teste”, criar um método e introduzir a anotação @Test
  - O método deve ser público e não pode ter parâmetros
  - O método não pode retornar dados (void)
  - Dentro deste método, deve-se utilizar um comando *assert*, para validar uma situação

# Como testar com JUnit

- Exemplo:

```
1 import static org.junit.Assert.*;
2 import org.junit.*;
3
4 public class CalculadoraTest {
5
6     @Test
7     public void test1_SomarNumeros() {
8         Calculadora calc = new Calculadora();
9         int resultado = calc.somar(10, 75);
10        assertEquals(85, resultado);
11    }
12
13 }
```



“Espero que o somar()  
resulte em 85”

# Métodos *Assert*

- Os métodos *assert* são utilizados para verificar se uma condição é verdadeira. Caso a verificação não seja bem sucedida, é lançada uma exceção

Método	Descrição
<code>assertEquals(long esperado, long real)</code>	Verifica se os dois números inteiros são iguais
<code>assertEquals(double esperado, double real, double limite)</code>	Verifica se dois números decimais são iguais, sem atingir o limite de diferença
<code>assertEquals(objetoEsperado, objetoReal)</code>	Verifica se dois objetos são iguais
<code>assertTrue(boolean condição)</code>	Verifica se a condição é verdadeira
<code>assertFalse(boolean condição)</code>	Verifica se a condição é falsa
<code>assertNotNull(objeto)</code>	Verifica se a variável referencia um objeto
<code>assertNull(objeto)</code>	Verifica se a variável não referencia um objeto

# Casos de teste com o mesmo contexto

- Quando dois ou mais casos de teste compartilham o mesmo contexto, é possível definir o contexto num método que contém a anotação `@Before`

```
@Before  
public void inicializarContexto() {  
    calc = new Calculadora();  
}
```

- Igualmente, é possível definir um método para ser executado após cada caso de teste, com a anotação `@After`

```
@After  
public void finalizarContexto() {  
    arquivo.close();  
}
```

# Exceções

- A anotação @Test permite utilizar o parâmetro “expected” para implementar casos de teste que esperam que ocorra uma determinada exceção

```
1 @Test(expected=IllegalArgumentException.class)
2 public void test013() throws Exception {
3     imp.setSalario(-100);
4 }
```

# Tempo limite de execução

- Para limitar o tempo máximo de execução de um teste, pode ser utilizado o parâmetro *timeout*

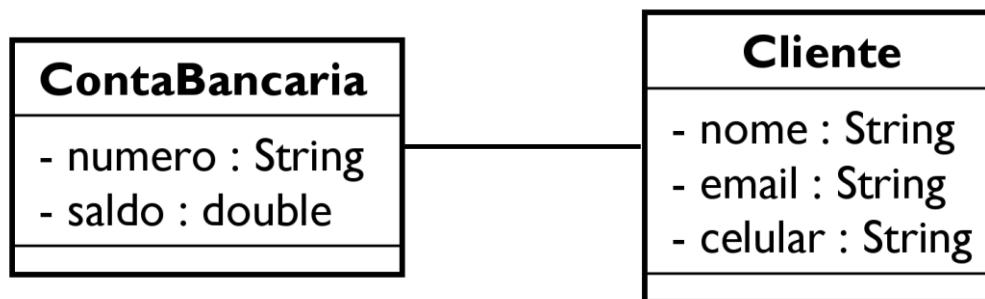
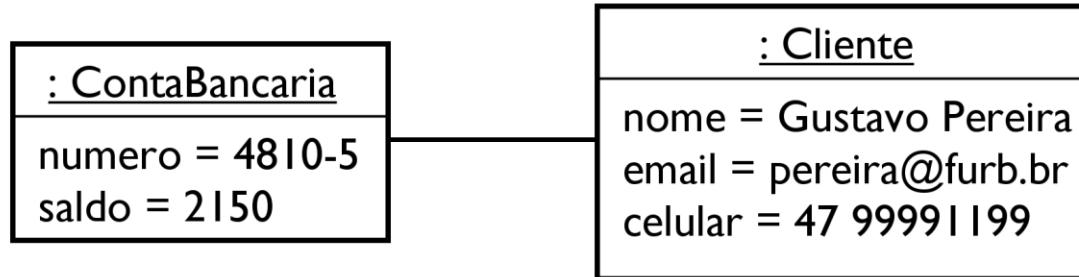
```
1 @Test(timeout = 1000)
2 public void testLoadUserCfg() {
3     db.setUser("teste");
4     db.loadUserConfig();
5 }
```

# Associações

# Associação

- Dois objetos podem estar ligados um ao outro
  - A ligação permite navegar de um objeto ao outro
- Para que seja possível ligar objetos, as classes destes objetos devem estar relacionadas através de uma **associação**
  - A associação é um tipo de relacionamento que conecta duas classes

# Associação

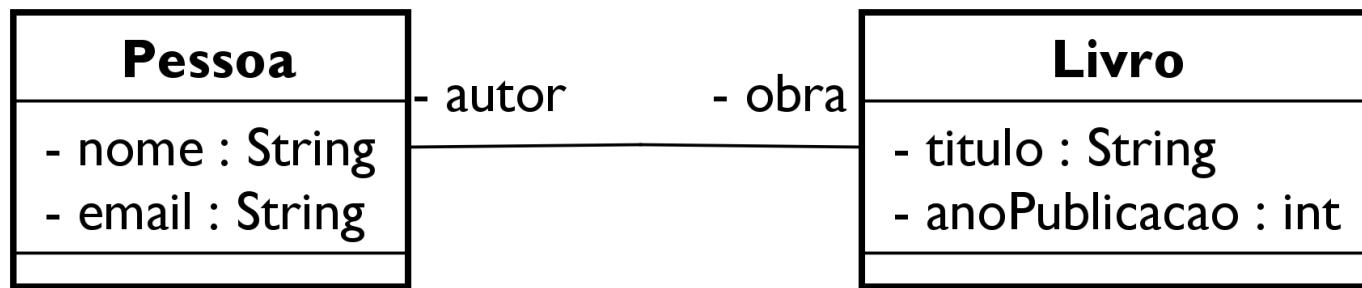


# Aprimoramentos da associação

- As associações podem ser melhor detalhadas através de aprimoramentos (também conhecidos como *adornos*)
- Existem 4 tipos de aprimoramentos:
  - Papel
  - Nome
  - Multiplicidade
  - Navegabilidade

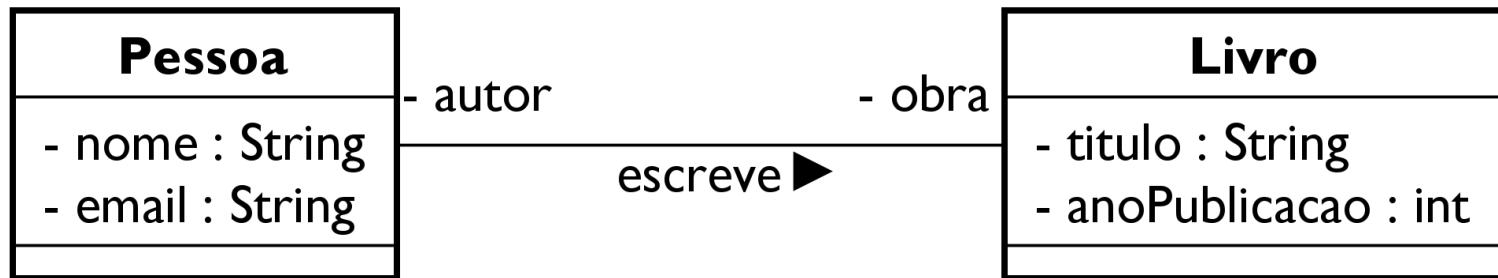
# Associações - Papel

- Cada classe que participa de uma associação tem um papel específico
- É possível nomear explicitamente o papel de uma classe no relacionamento



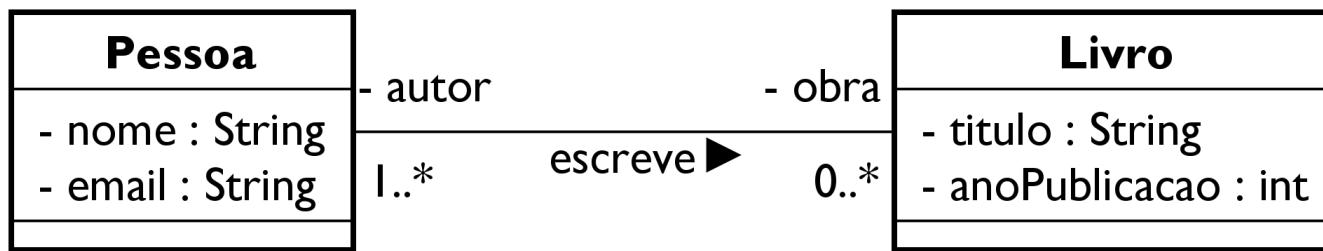
# Associações - Nome

- Uma associação pode ter um nome, que pode ser utilizado para descrever a natureza do relacionamento
- Pode ser indicada a direção da leitura do nome, utilizando-se uma seta

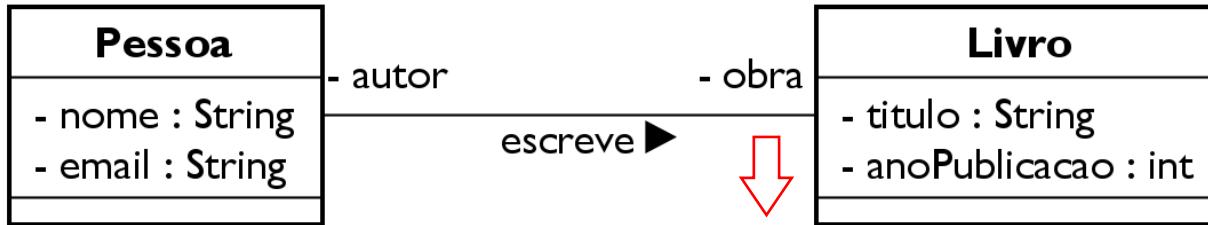


# Multiplicidade

- Determina a quantidade de objetos que podem ser interconectados
- A “quantidade” é chamada de multiplicidade
- Escrita com uma expressão indicando valor mínimo e máximo



# Multiplicidade - Exemplo



Para preencher a multiplicidade aqui, se perguntar:

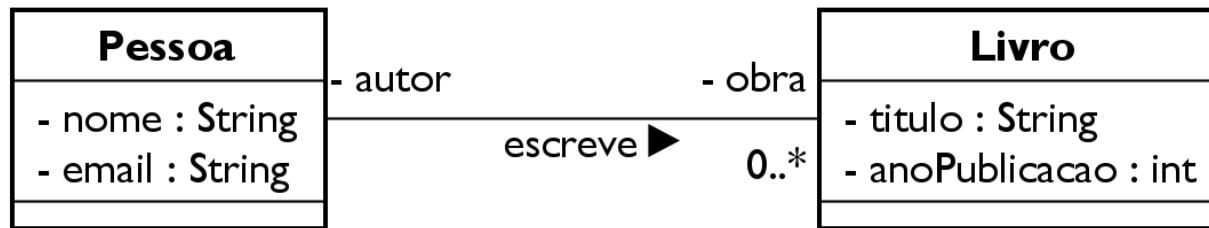
Dado uma **pessoa** em particular:

- Qual a quantidade mínima de **livros** que ela pode escrever?
- Qual a quantidade máxima de **livros** que ela pode escrever?

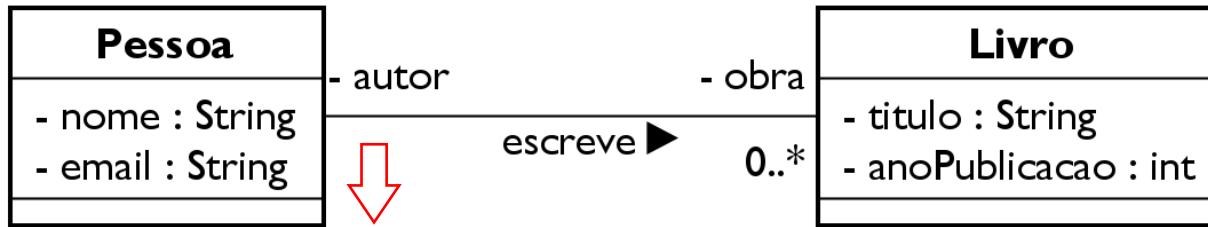
Neste exemplo, as respostas são:

- No mínimo: nenhum livro (0)
- No máximo: indeterminado (\*).

Logo:



# Multiplicidade - Exemplo



Para preencher a multiplicidade aqui, se perguntar:

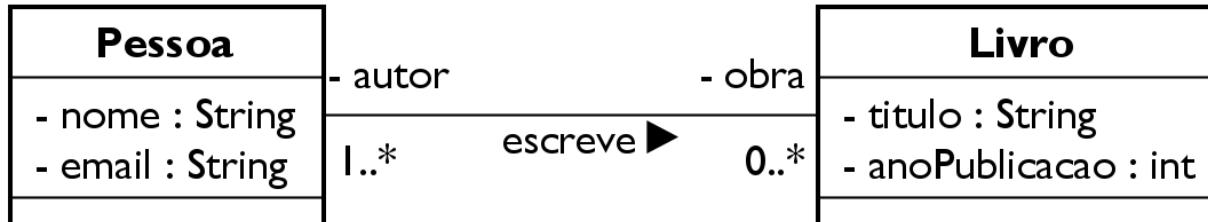
Dado um **livro** em particular:

- a) Qual a quantidade mínima de **pessoas** que podem escrevê-lo?
- b) Qual a quantidade máxima de **pessoas** que podem escrevê-lo?

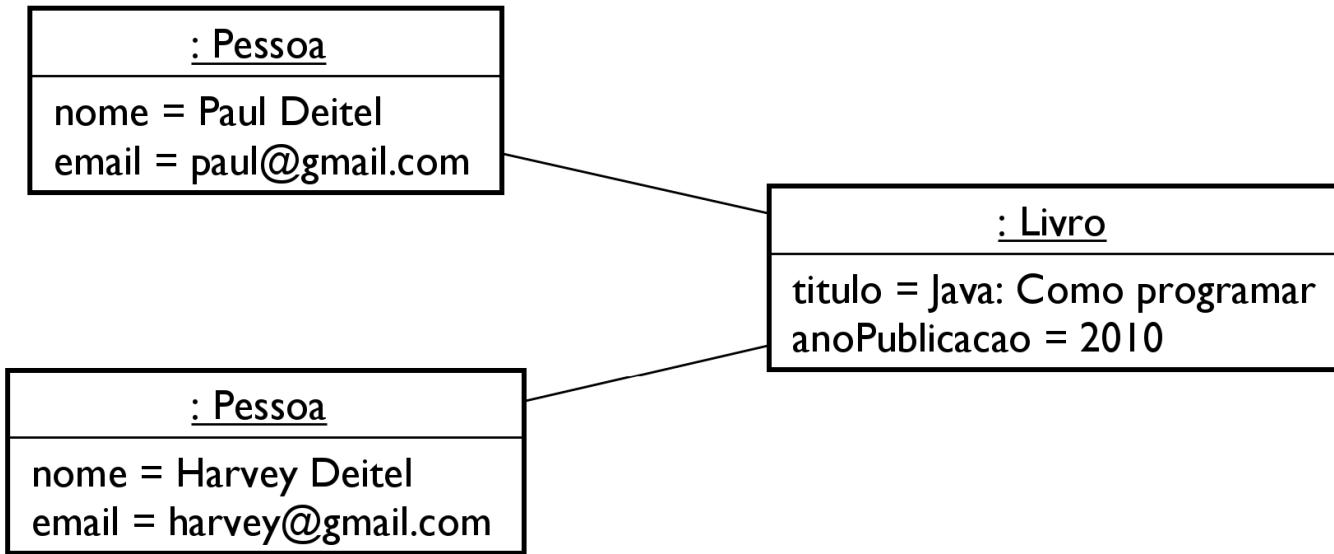
Neste exemplo, as respostas são:

- a) No mínimo: uma pessoa (1)
- b) No máximo: indeterminado (\*).

Logo:



# Diagrama de objetos exemplo

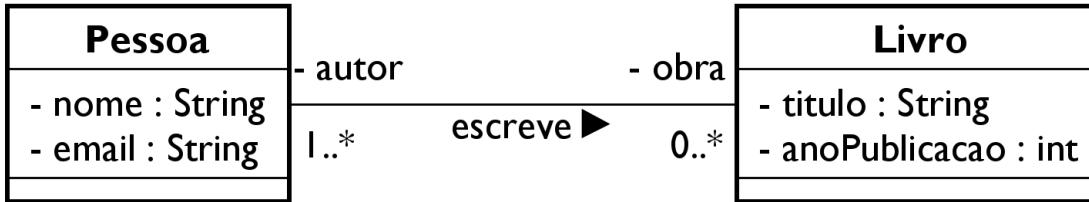


# Multiplicidade

Multiplicidade	Significado
0..1	Indica que os objetos das classes associadas não precisam obrigatoriamente estar relacionados, mas se houver relacionamento, indica que apenas uma instância da classe se relaciona com as instâncias da outra classe
1 (ou 1..1)	Indica que apenas um objeto da classe se relaciona com os objetos de outra classe
0..*	Indica que pode ou não haver instâncias da classe participando do relacionamento.
1..*	Indica que há pelo menos um objeto envolvido no relacionamento, podendo haver muitos objetos
3..5	Indica que existem pelo menos 3 instâncias envolvidas no relacionamento, mas não mais do que 5.

# Navegabilidade

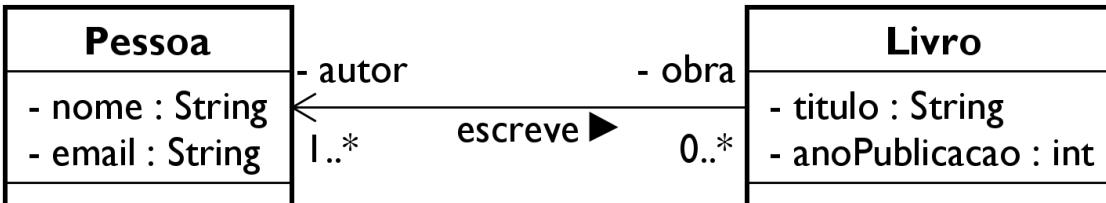
- Por padrão, a navegação entre objetos é **bidirecional**



A partir de um livro é possível navegar até seus autores.

A partir de uma pessoa é possível navegar até suas obras.

- É possível limitar a navegação para uma única direção, desenhando-se uma seta. Trata-se de uma navegação **unidirecional**

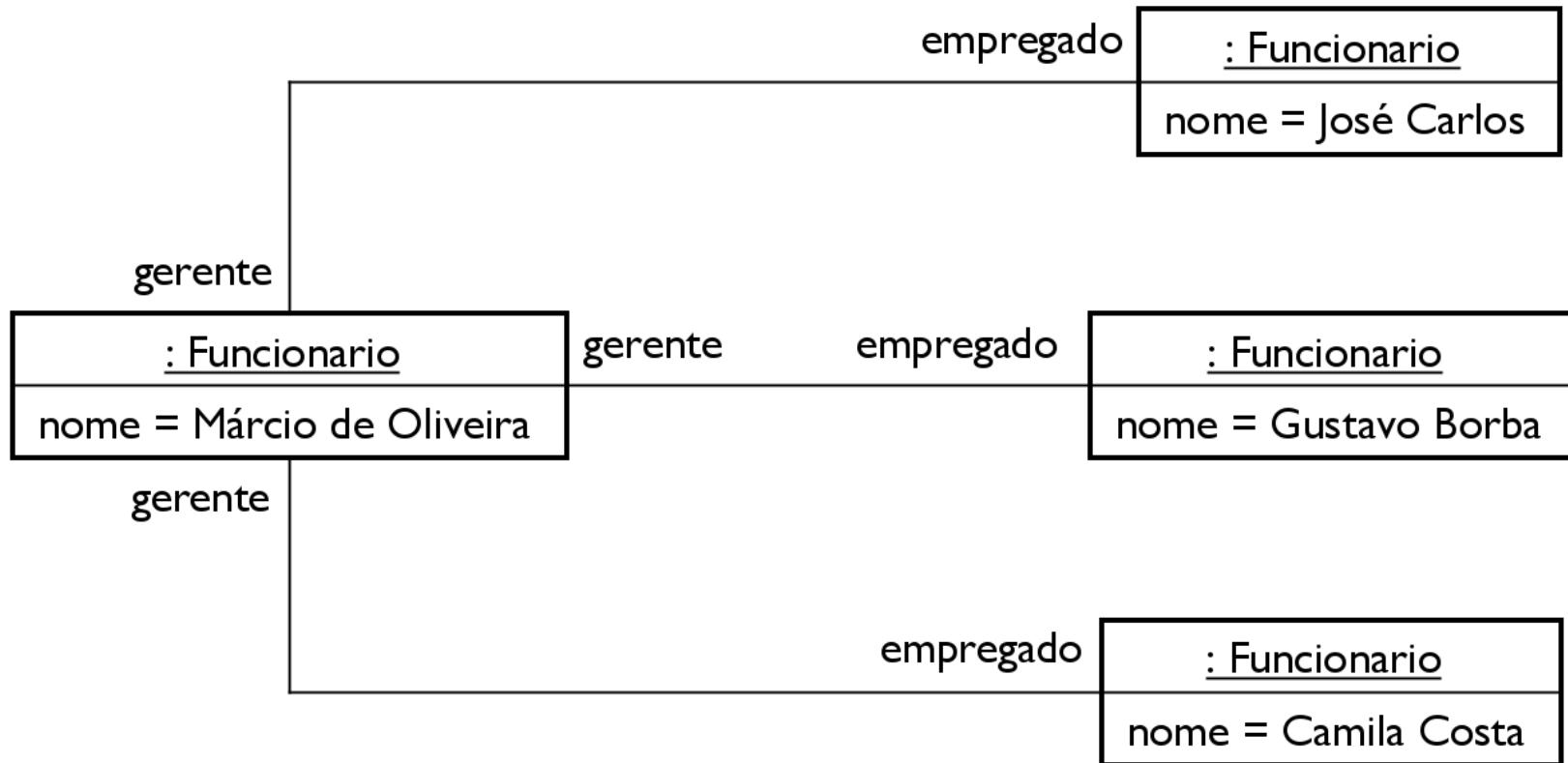


A partir de um livro é possível navegar até seus autores.

A partir de uma pessoa **não** é possível navegar até suas obras.

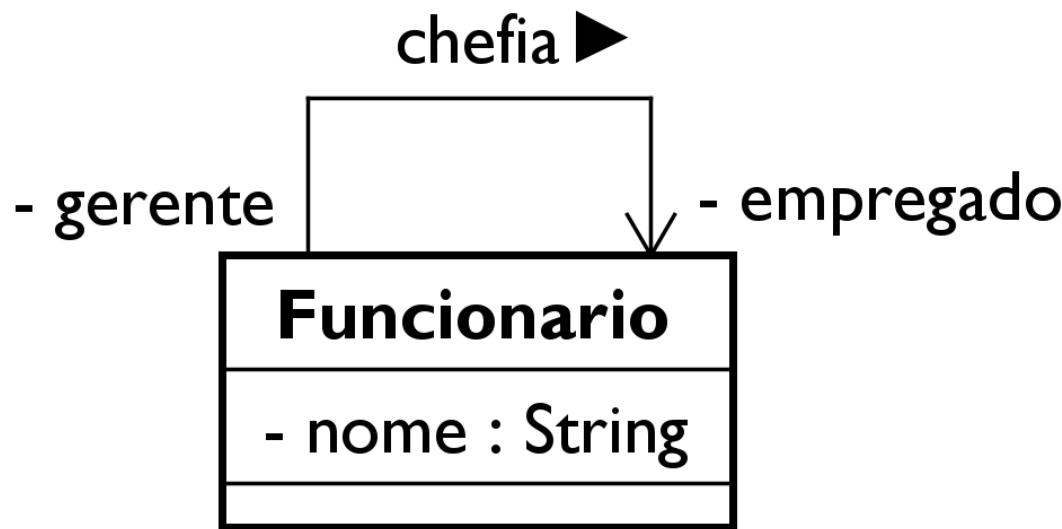
# **Associação reflexiva**

# Associação Reflexiva



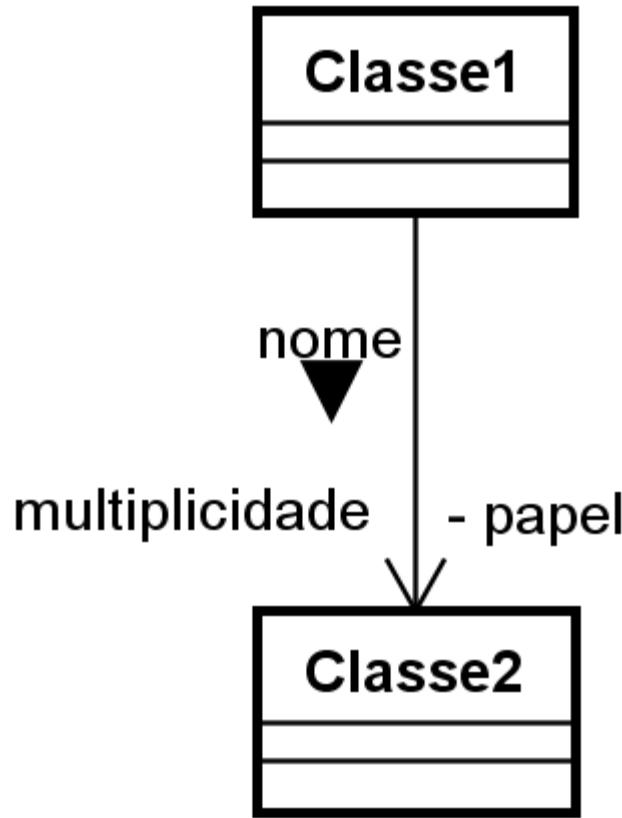
# Associação Reflexiva

- É uma associação que estabelece uma conexão entre objetos de uma mesma classe



# **Relacionamento entre objetos**

# Associações



- Acompanhada de adornos (ou aprimoramentos):
  - Nome
  - Papel
  - Multiplicidade
  - Navegabilidade

# **Associações com multiplicidade 0..1 ou 1**

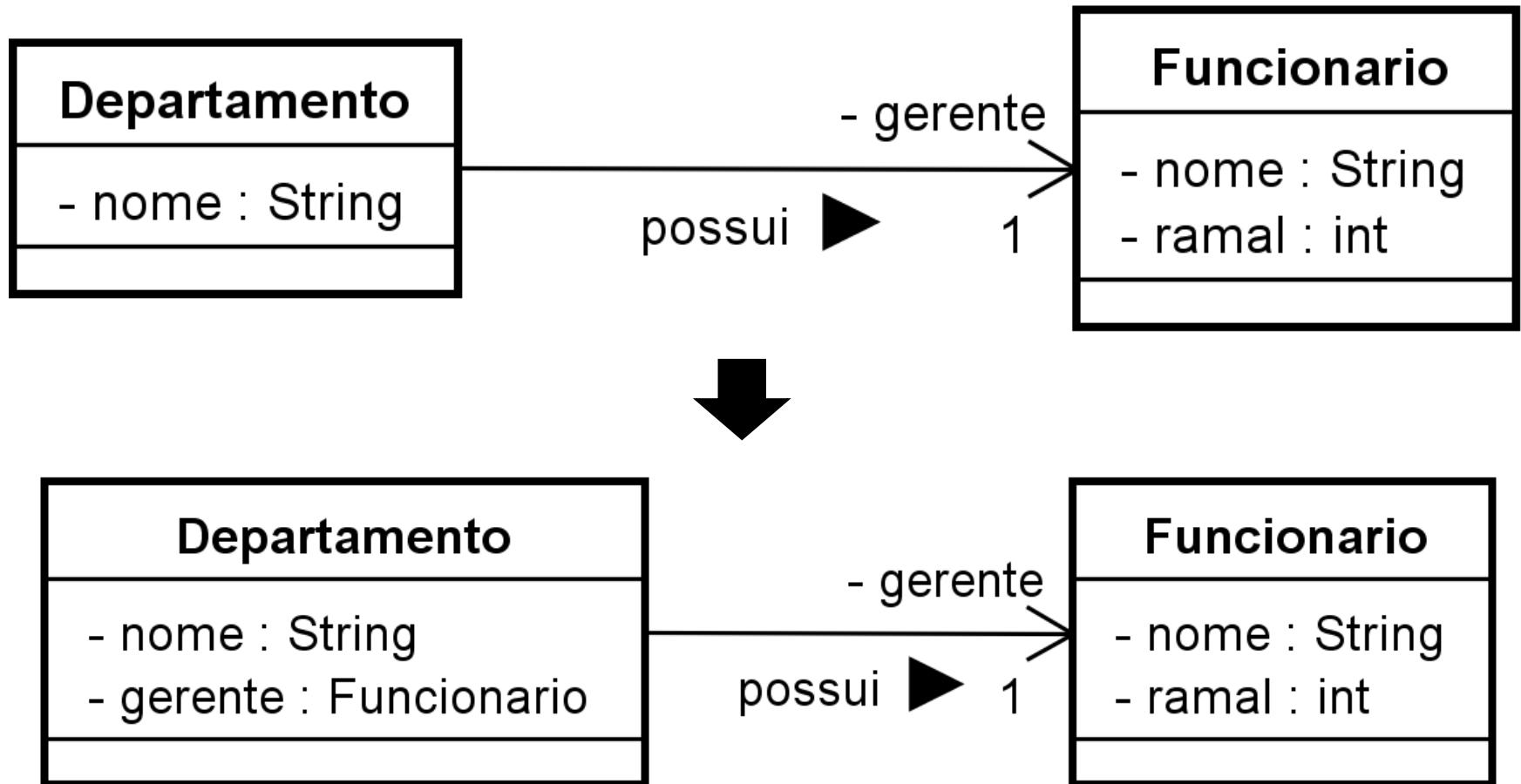
# Associações com multiplicidade 0..1 ou 1



- Tradução para Java requer:

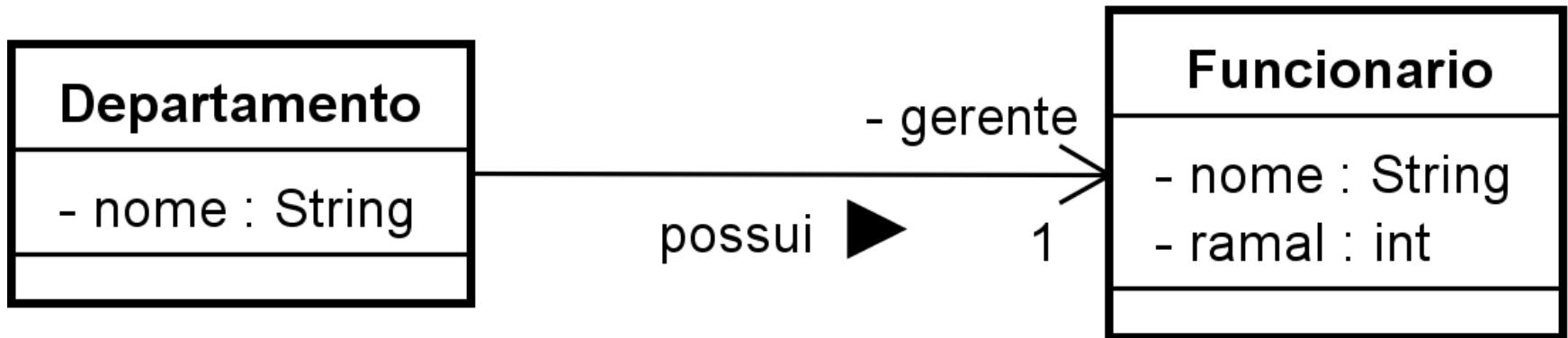
- Criar uma variável de instância na classe de origem
  - O *identificador* desta variável deve ser igual ao papel que está próximo à classe de destino
  - Na ausência de papel, utilizar o nome da classe como identificador
  - O tipo de dado desta variável deve ser igual à classe de destino

# Associações com multiplicidade 0..1 ou 1

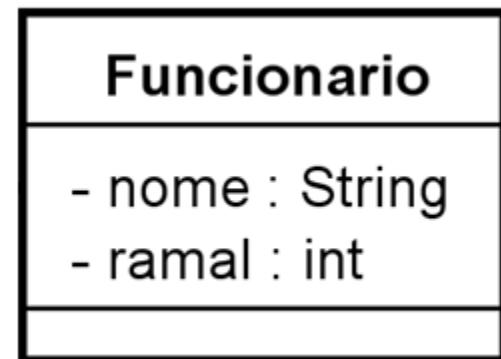
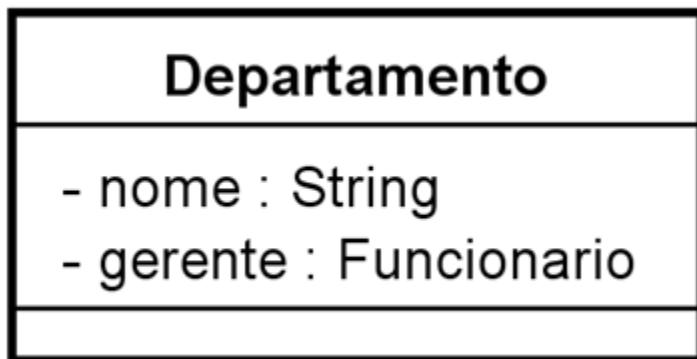


Representação alternativa, que apresenta o relacionamento de forma redundante

# Associações com multiplicidade 0..1 ou 1



É equivalente à:



# Associações com multiplicidade 0..1 ou 1

- Geralmente utiliza-se métodos de acesso para manipular a nova variável de instância

## Departamento

- nome : String  
- gerente : Funcionario

+ getGerente() : Funcionario  
+ setGerente(func : Funcionario) : void

# Exemplo em Java

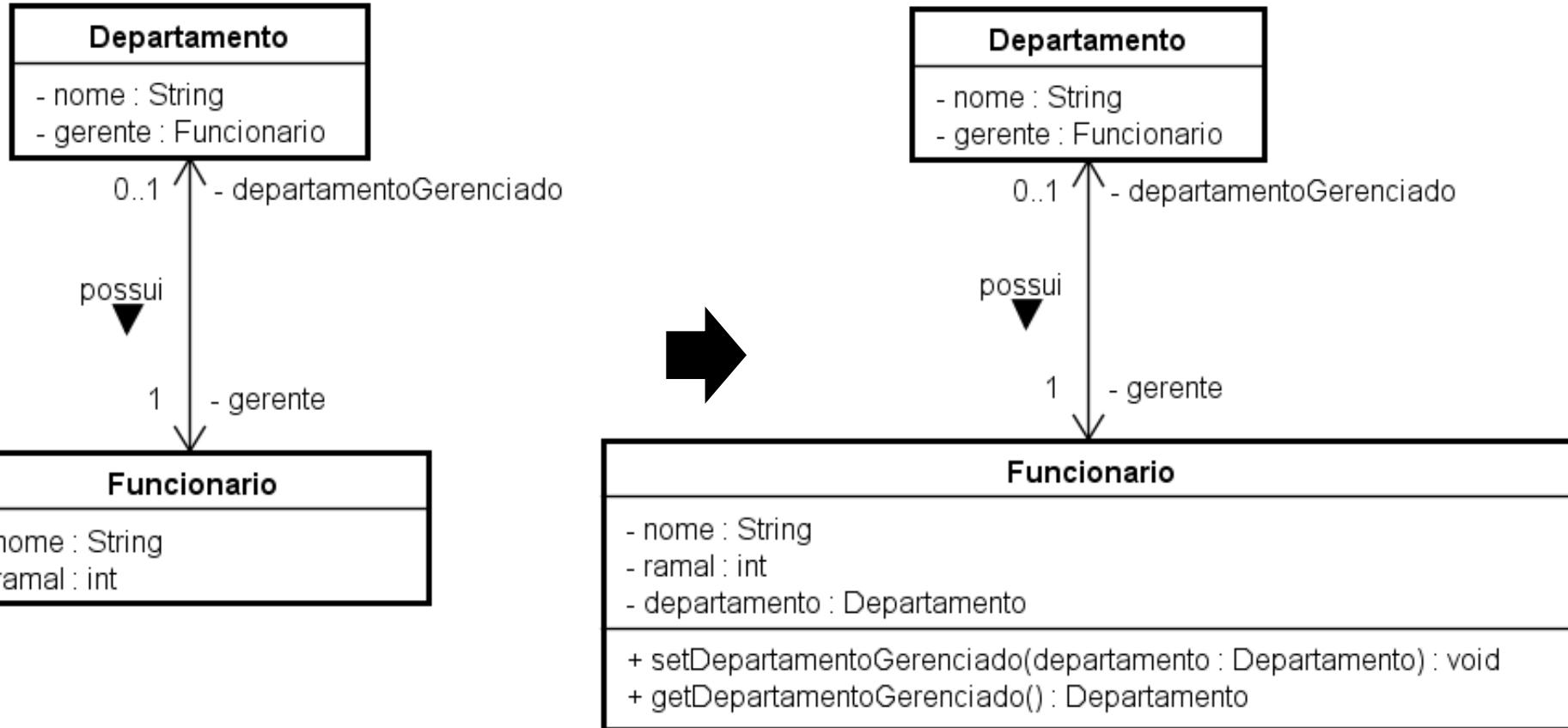
```
public class Departamento {  
  
    private String nome;  
    private Funcionario gerente;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public Funcionario getGerente() {  
        return gerente;  
    }  
  
    public void setGerente(Funcionario gerente) {  
        this.gerente = gerente;  
    }  
}
```

# Exemplo de utilização

```
1 public void exemplo() {  
2     Funcionario f1 = new Funcionario();  
3     f1.setNome("Juliano Korz");  
4     f1.setRamal(4901);  
5  
6     Departamento d1 = new Departamento();  
7     d1.setNome("Contabilidade");  
8  
9     d1.setGerente(f1);  
10 }
```

```
System.out.println( d1.getGerente().getNome() );
```

# Relacionamento bidirecional



# Exemplo

```
1 public void exemplo() {  
2     Funcionario f1 = new Funcionario();  
3     f1.setNome("Juliano Korz");  
4     f1.setRamal(4901);  
5  
6     Departamento d1 = new Departamento();  
7     d1.setNome("Contabilidade");  
8  
9     d1.setGerente(f1);  
10    f1.setDepartamentoGerenciado(d1);  
11  
12    System.out.println( d1.getGerente().getNome() );  
13    System.out.println( f1.getDepartamentoGerenciado().getNome() );  
14 }
```

# ArrayList

# ArrayList

- Uma instância da classe ArrayList pode armazenar diversos objetos
  - Forma alternativa ao vetor para guardar dados
  - Não possui tamanho limitado, como no vetor
  - Não armazena dados primitivos
- Os objetos armazenados no objeto da classe ArrayList são recuperados pela posição

# A classe ArrayList

ArrayList	E
+ add(e : E) : boolean	
+ get(index : int) : E	
+ remove(o : Object) : boolean	
+ size() : int	
Método	Descrição
add(E)	Guarda um objeto
get(int)	Obtém um objeto na posição indicada
remove(Object)	Remove o objeto
size()	Retorna a quantidade de objetos armazenados

O construtor padrão cria um `ArrayList` vazio. Usamos o operador *diamante* para indicar o tipo dos objetos que se quer armazenar:

```
ArrayList<Aluno> turma;  
alunos = new ArrayList<>();
```

É possível combinar a declaração e criação:

```
ArrayList<Aluno> turma = new ArrayList<>();
```

# Exemplo

## Aluno

- nome : String
+ Aluno(nome : String)
+ getNome() : String

```
public static void main(String[] args) {  
  
    ArrayList<Aluno> turma = new ArrayList<>();  
  
    Aluno a1 = new Aluno("Leonir Santos");  
    turma.add(a1);  
  
    Aluno a2 = new Aluno("Anderson Gonçalves");  
    turma.add(a2);  
  
    turma.add( new Aluno("Jean Cardoso") );  
  
    ...  
}
```

# Exemplo – Percorrendo o ArrayList

```
Aluno aluno;

for (int i=0; i<turma.size(); i++) {
    aluno = turma.get(i);
    System.out.println(aluno.getNome());
}
```

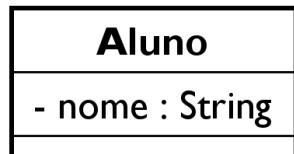
Também é possível usar o comando for-each:

```
for (Aluno aluno : turma) {
    System.out.println(aluno.getNome());
}
```

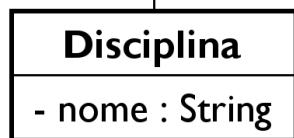
# **Associações com multiplicidade acima de 1**

# Passos para tradução da associação

Classe de destino



1..\*



Classe de origem

## 1. Deve-se criar uma variável na classe de origem

O identificador da variável será igual ao papel. Na ausência, o nome do identificador deve ser derivado do nome da classe de destino. Geralmente, é igual ao nome da classe, mas no plural.

## 2. O tipo de dado desta variável deve ser **ArrayList**

Utilizar o operador diamante informando o nome da classe de destino

## 3. Criar a instância da classe **ArrayList**

Pode ser feito na declaração da variável ou no construtor.

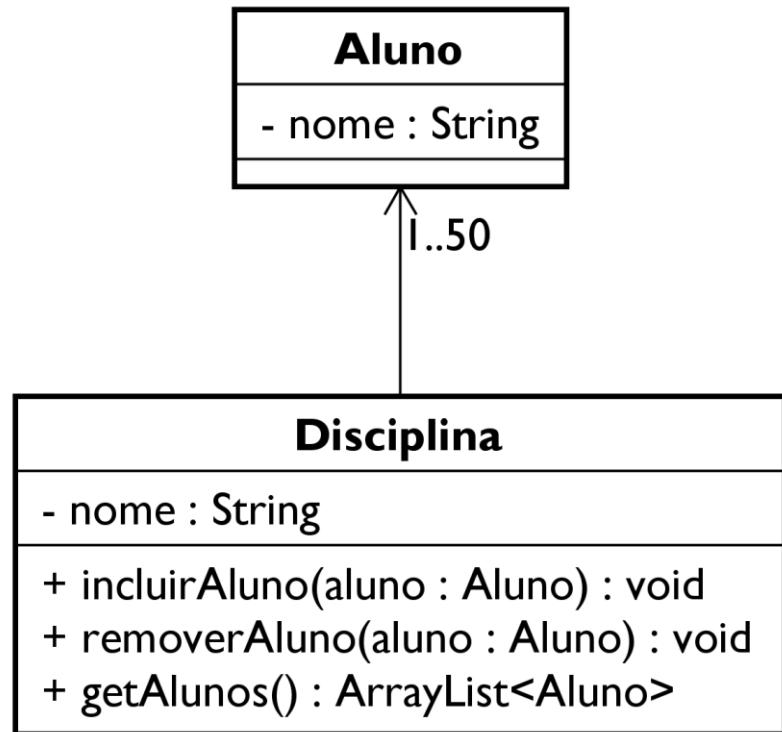
```
3  import java.util.ArrayList;
4
5  public class Disciplina {
6
7      private String nome;
8      private ArrayList<Aluno> alunos = new ArrayList<>();
```

Exemplo de tradução

# Passos para tradução da associação

4. A classe de origem deve implementar métodos para **incluir, remover** e **obter** os objetos contidos na associação

{



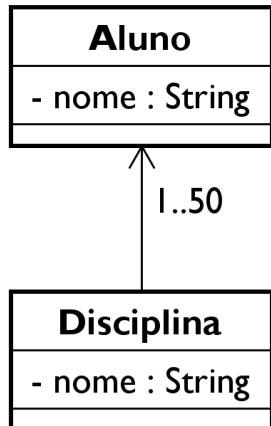
# Passos para tradução da associação

Método para incluir alunos na disciplina

```
public void incluirAluno(Aluno aluno) {  
    alunos.add(aluno);  
}
```

Deve receber como parâmetro um objeto a ser adicionado

Se a associação for limitada, deve-se impedir incluir mais objetos do que o permitido



```
public void incluirAluno(Aluno aluno) {  
    if (alunos.size() == 50) {  
        throw new RuntimeException("Não é possível incluir "  
            + "mais alunos na disciplina");  
    }  
    alunos.add(aluno);  
}
```

# Passos para tradução da associação

## Método para remover alunos na disciplina

Deve receber como parâmetro um objeto a ser removido

```
public void removerAluno(Aluno aluno) {  
    alunos.remove(aluno);  
}
```

# Passos para tradução da associação

*Getter* para a variável que mantém a associação

```
public ArrayList<Aluno> getAlunos() {  
    return alunos;  
}
```

**Não** existe setter para a variável que mantém a associação

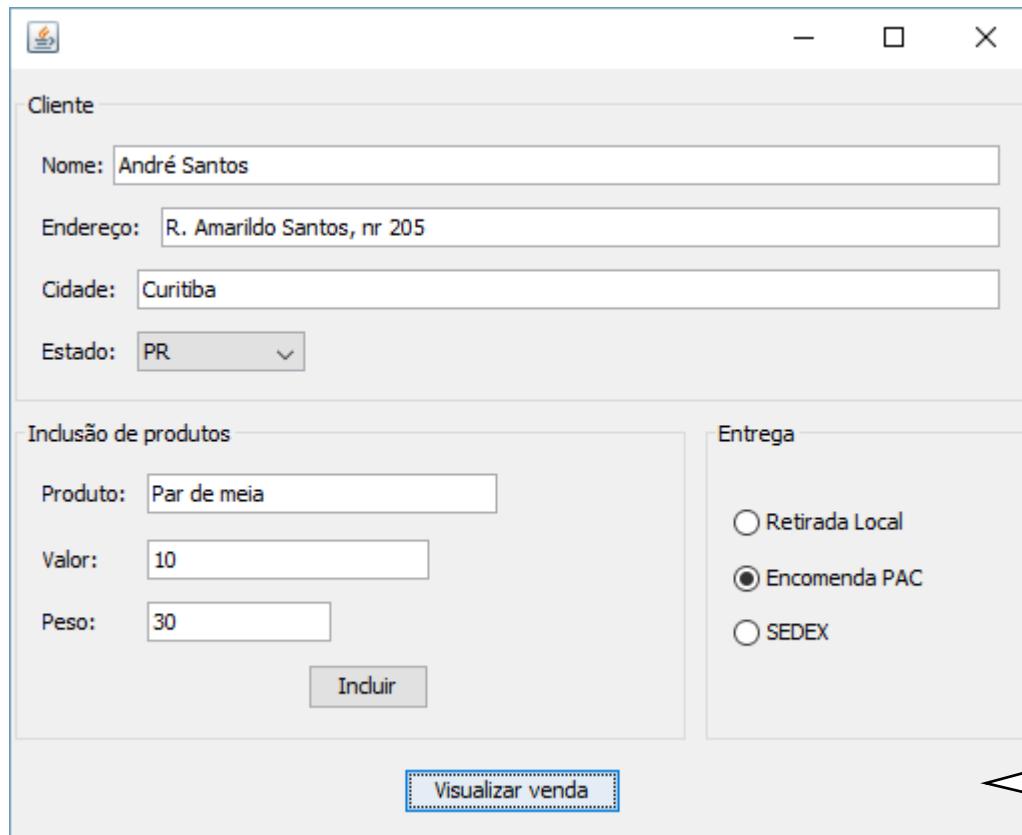
# **Outros componentes GUI Swing**

# Outros componentes GUI

- Painel
- Botões de rádio
- Grupo de botões
- Área de texto
- Tela modal

# Painel

- Permite agrupar logicamente outros controles GUI



Painéis são objetos da classe JPanel.

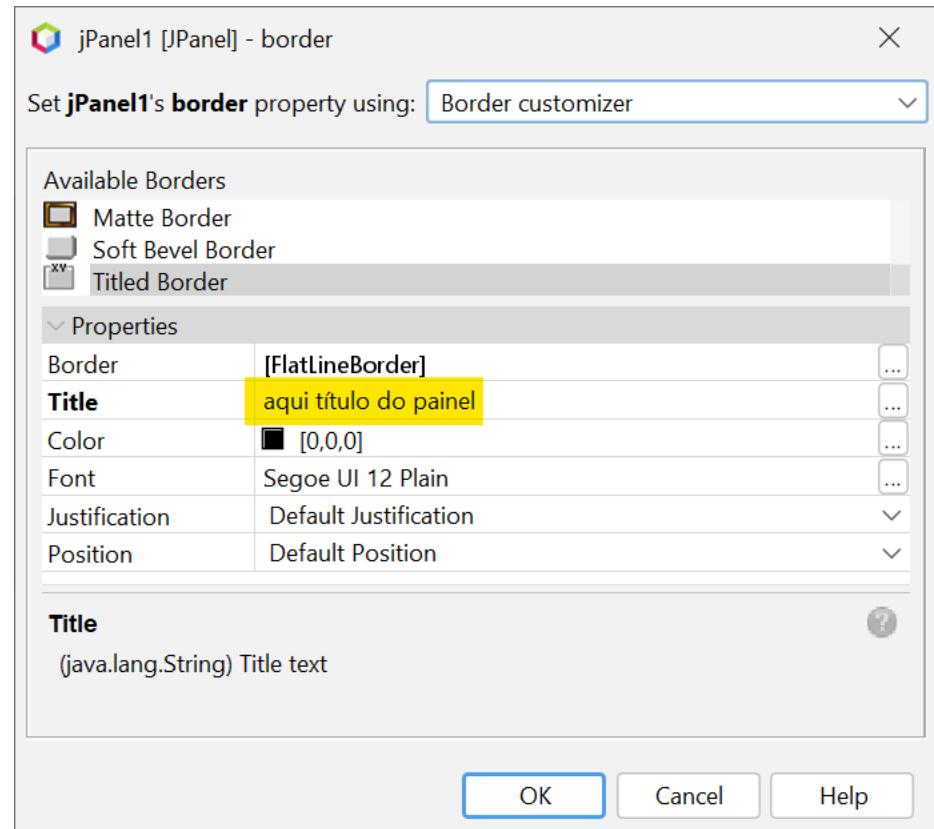
Para adicionar um objeto JPanel, localize-o na paleta de componentes, no grupo “Swing Containers”

Tela contendo  
3 painéis

# Painel

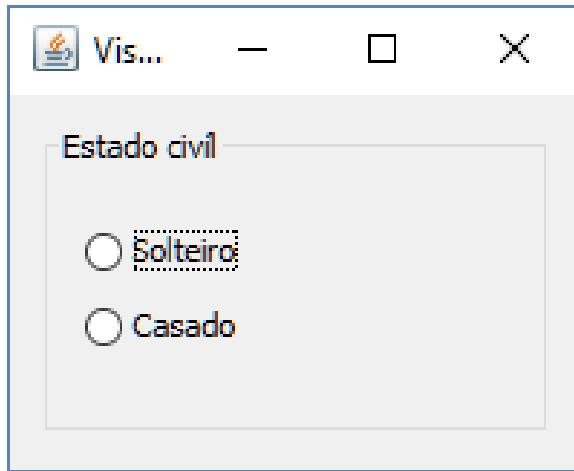
Para inserir uma borda no painel e atribuir um título:

- Selecionar o controle
- No painel *Properties*, acesse a propriedade *Border*;
- Selecione a opção *Titled Border*;
- Altere o valor da propriedade *Title*.



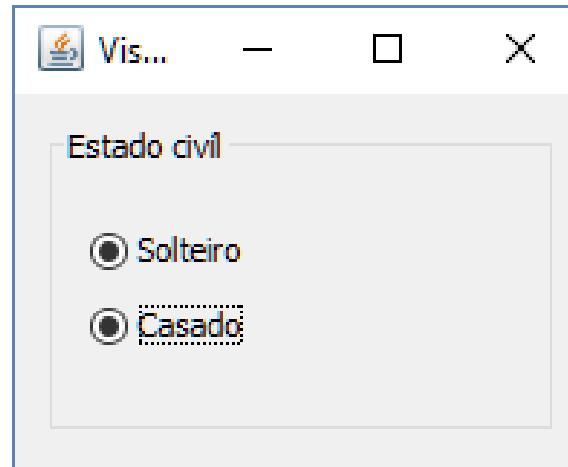
# Agrupamento de botões de rádio

Considere esta tela



...

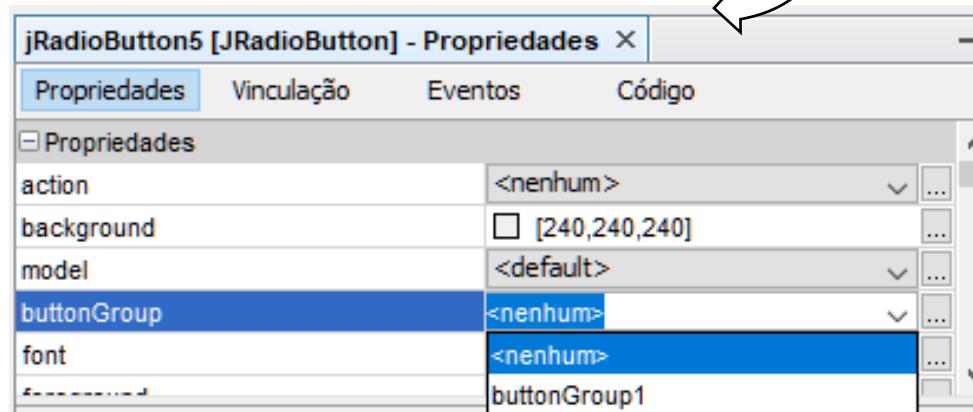
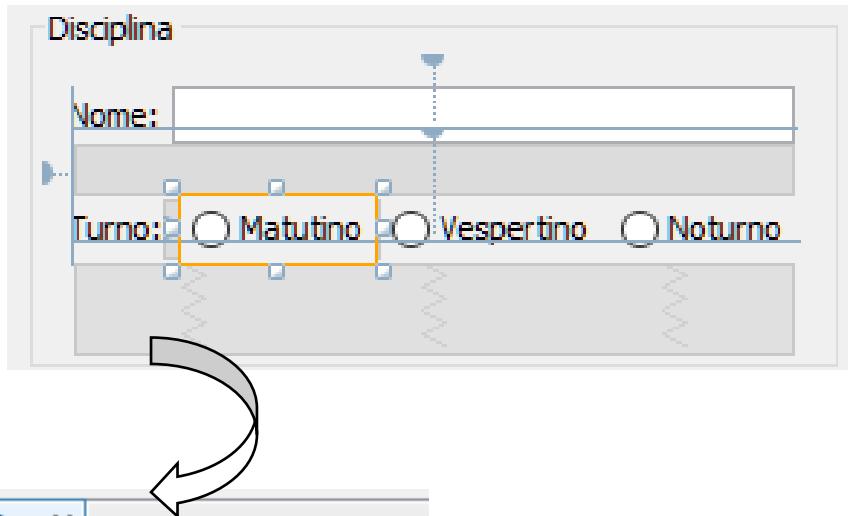
É possível marcar as opções como abaixo:



Para impedir a seleção de múltiplos botões, adicione um *grupo de botões*. Um *grupo de botões* é um controle não visual da classe ButtonGroup

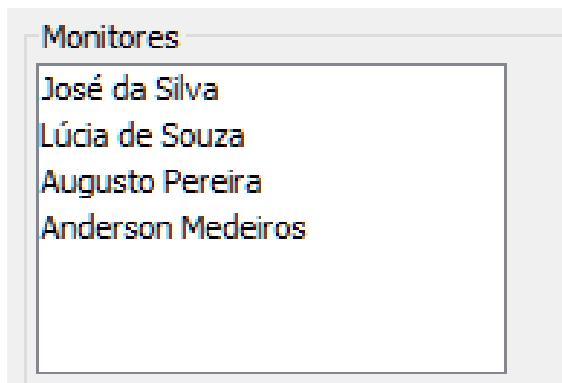
# Agrupamento de botões de rádio

Em seguida, altere a propriedade buttonGroup de cada objeto da classe JRadioButton para que seja igual ao objeto recentemente criado



# Controle para armazenar texto (com várias linhas)

Objetos da classe `JTextArea` permitem exibir texto em várias linhas.



Além de exibir dados, objetos desta classe permitem que o usuário digite texto.

Para adicionar uma linha de texto num objeto da classe `JTextArea`, utilize o método `append()`, como em:  
`taMonitores.append( m.getNome() ) ;`

# Tela modal

- Permite exibir telas modais:
  - Tela modal é uma tela que precisa ser fechada para que o usuário possa acessar outras telas da aplicação
- Telas criadas a partir de JDDialog podem ser modais
- Para criar e exibir uma tela modal:

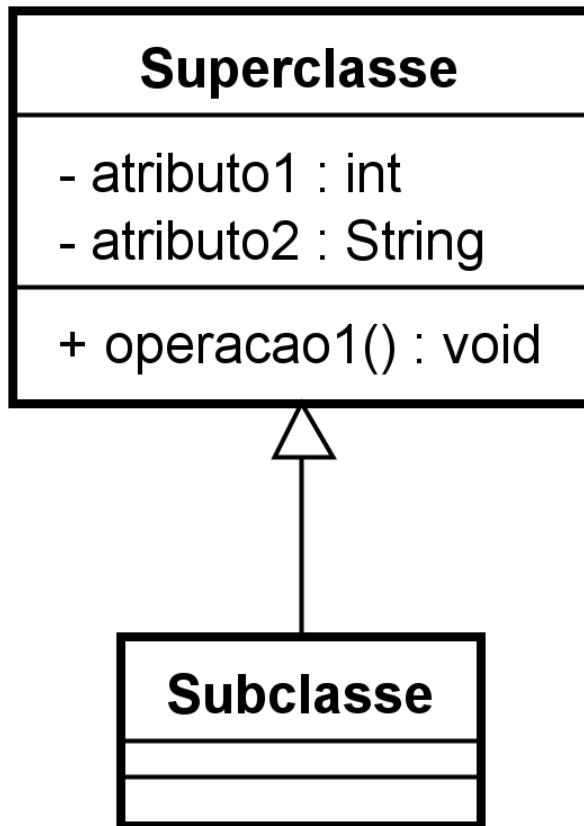
```
classe identificador = new classe (this, true);  
identificador.setVisible(true);
```

# Herança

# Herança

- Uma forma de reuso de software em que uma nova classe é criada absorvendo os membros (atributos e métodos) de uma classe já existente.
- Ao criar uma classe, ao invés de declarar membros completamente novos, é possível definir que a nova classe deve herdar os membros de uma classe já existente
  - A classe já existente é chamada de *superclasse, classe pai ou classe base*
  - A nova classe é chamada de *subclasse, classe filha ou classe derivada*
- A subclasse pode adicionar seus próprio membros
  - A subclasse pode possuir os mesmos comportamentos de sua superclasse mas pode também adicionar comportamento específico.

# Herança – Representação em UML



- A herança é uma forma de reuso
- A subclasse é mais específica que a sua superclasse, representando um grupo de objetos mais especializado
  - A herança também é conhecida como “especialização”

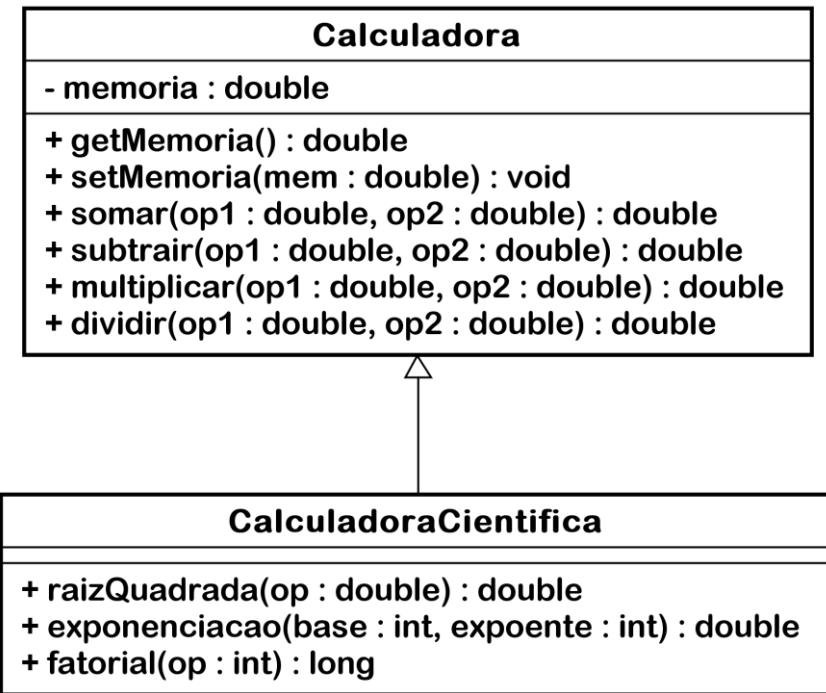
```

public class Calculadora {
    private double memoria;
    public double getMemoria() {
        return memoria;
    }
    public void setMemoria(double mem) {
        this.memoria = mem;
    }
    public double somar(double op1, double op2) {
        return op1+op2;
    }
    public double subtrair(double op1, double op2) {
        return op1-op2;
    }
    public double multiplicar(double op1, double op2) {
        return op1*op2;
    }
    public double dividir(double op1, double op2) {
        return op1/op2;
    }
}

```

Calculadora
- memoria : double
+ getMemoria() : double
+ setMemoria(mem : double) : void
+ somar(op1 : double, op2 : double) : double
+ subtrair(op1 : double, op2 : double) : double
+ multiplicar(op1 : double, op2 : double) : double
+ dividir(op1 : double, op2 : double) : double

# Exemplo



- A subclasse é uma versão especializada da superclasse
- Dizemos que **CalculadoraCientifica** **estende** a classe **Calculadora**

Ou

- **CalculadoraCientifica** **herda** a classe **Calculadora**

# Exemplo

```
public class CalculadoraCientifica extends Calculadora {  
  
    public double raizQuadrada(double op) {  
        // ...  
    }  
  
    public double exponenciacao(int base, int expoente) {  
        // ...  
    }  
  
    public long fatorial(int op) {  
        // ...  
    }  
}
```

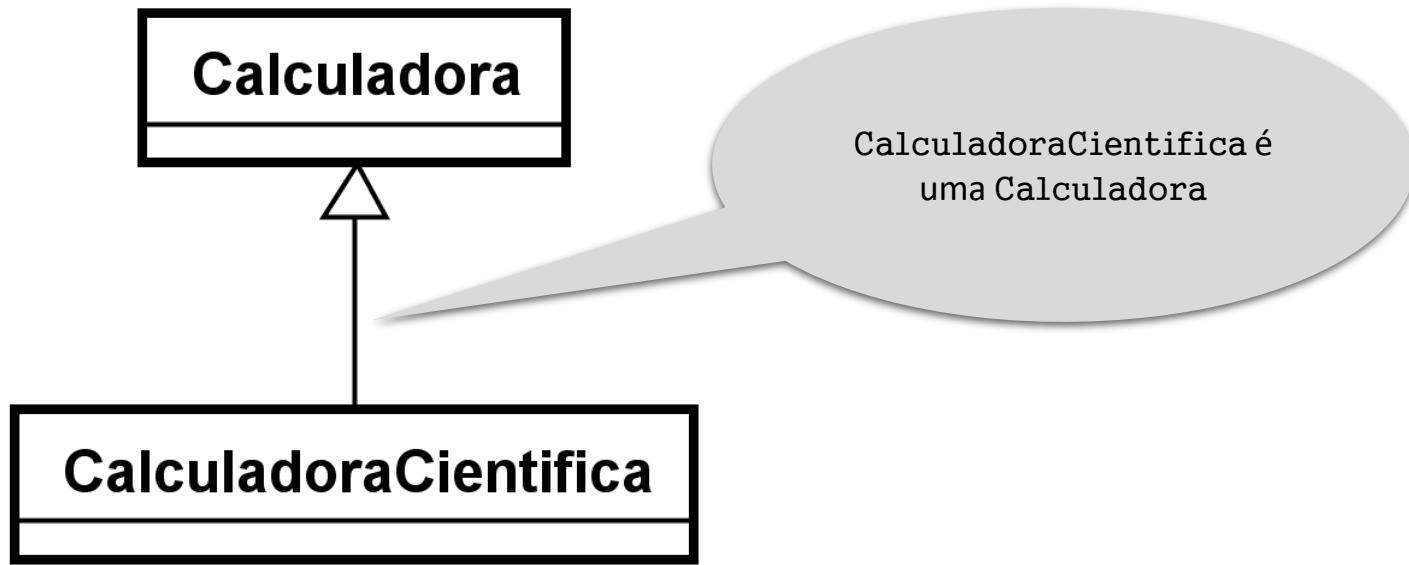
# Exemplo – Programa Java

- Exemplo de programa principal.

```
public static void main(String[] args) {  
    CalculadoraCientifica cs = new CalculadoraCientifica();  
    System.out.println(cs.somar(12,13));  
    System.out.println(cs.fatorial(5));  
}
```

# Herança é um relacionamento

- A herança é um relacionamento entre classes
- Lemos o relacionamento com a expressão “é um(a)”

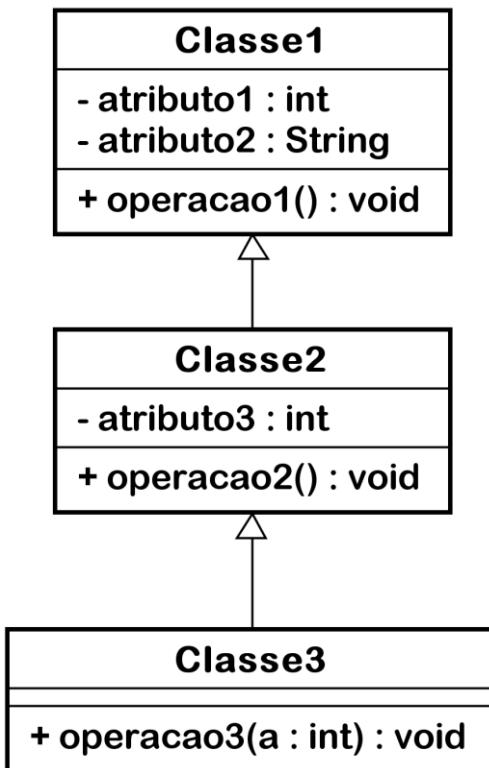


# Herança

- É possível especializar classes construídas pelo próprio programador, bem como especializar classes de terceiros, como as classes da própria linguagem Java.

# Herança

Cada subclasse pode ser uma superclasse de futuras subclasses.



A **superclasse direta** é a superclasse herdada diretamente por uma subclasse

A **superclasse indireta** é a superclasse herdada indiretamente

# Estrutura hierárquica de herança

As relações de herança formam estruturas hierárquicas parecidas com uma árvore. Esta hierarquia de classes também é conhecida como **hierarquia de herança**

Overview Package **Class** Use Tree Deprecated Index Help

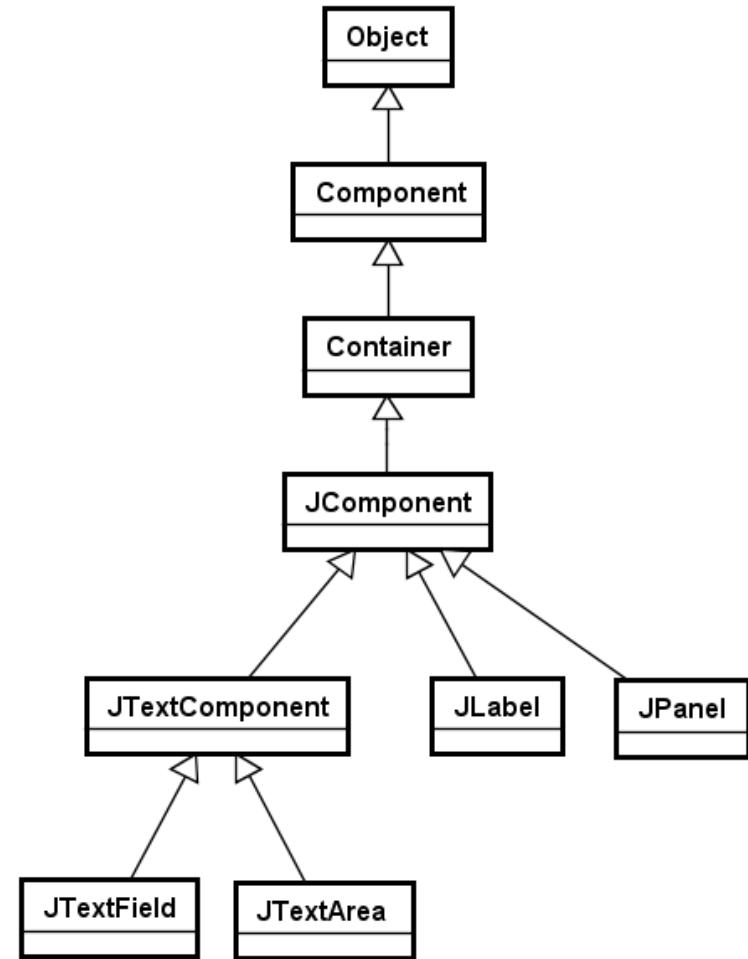
Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

javax.swing

## Class JTextArea

java.lang.Object  
  java.awt.Component  
    java.awt.Container  
      javax.swing.JComponent  
        javax.swing.text.JTextComponent  
          javax.swing.JTextArea



# Herança X membros privados

Classe1
- valor : int
+ getValor() : int
+ setValor(v : int) : void

Classe2
+ metodo1() : void

Uma subclasse não pode acessar membros privados de sua superclasse.

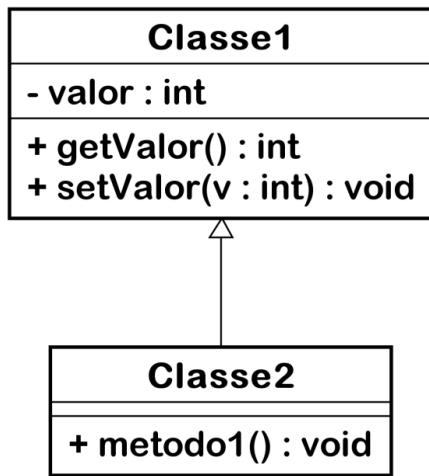
```
public class Classe1 {  
  
    private int valor;  
  
    public void setValor(int valor) {  
        this.valor = valor;  
    }  
  
    public int getValor() {  
        return valor;  
    }  
}
```

```
public class Classe2 extends Classe1 {  
  
    private void metodo1() {  
        valor = 10;  
    }  
}
```

Erro de compilação

# Herança X membros privados

- Porém, a subclasse pode alterar o valor de variáveis privadas da superclasse através de métodos **não privados** da superclasse.



```
public class Classe1 {
    private int valor;

    public void setValor(int valor) {
        this.valor = valor;
    }

    public int getValor() {
        return valor;
    }
}

public class Classe2 extends Classe1 {
    private void metodo1() {
        setValor(10);
    }
}
```

The code illustrates the implementation of the classes shown in the UML diagram. **Classe1** contains a private attribute `valor` and methods `setValor` and `getValor`. **Classe2** extends **Classe1** and overrides the `metodo1` method, which calls `setValor(10)`.

# Herança X membros privados

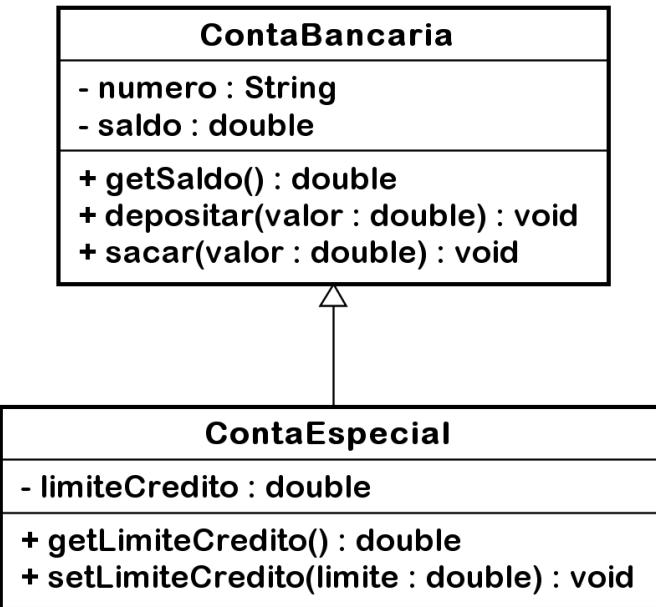
- Se a variável é privada e o *setter* não é público (ou não há *setter*), para acessar a variável na subclasse, recomenda-se:
  - Manter privada a variável de instância
  - Tornar (ou criar) o *setter* com modificador de acesso *protected* (protegido)
- O modificador de acesso *protected* torna o membro acessível:
  - pela própria classe,
  - classes do mesmo pacote e
  - Subclasses
- Em UML, o membro protegido utiliza o símbolo #

# Herança e sobrescrita de método

Na subclasse, além de incluir novos atributos e novos métodos, é possível modificar (redefinir) o comportamento de métodos herdados. Esta funcionalidade é chamada de “sobrescrita de método”

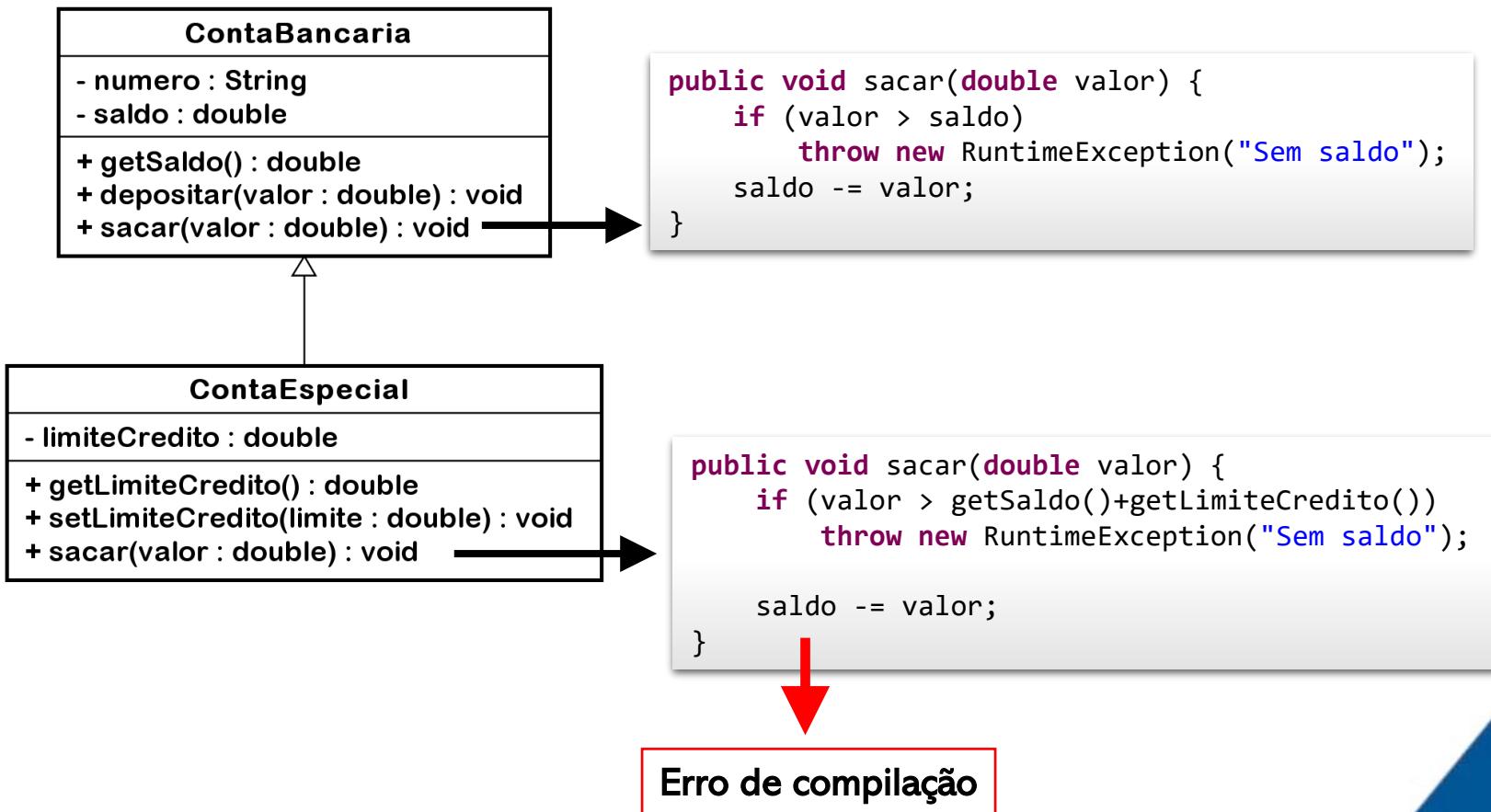
# Exemplo de sobrescrita de método

- Motivação



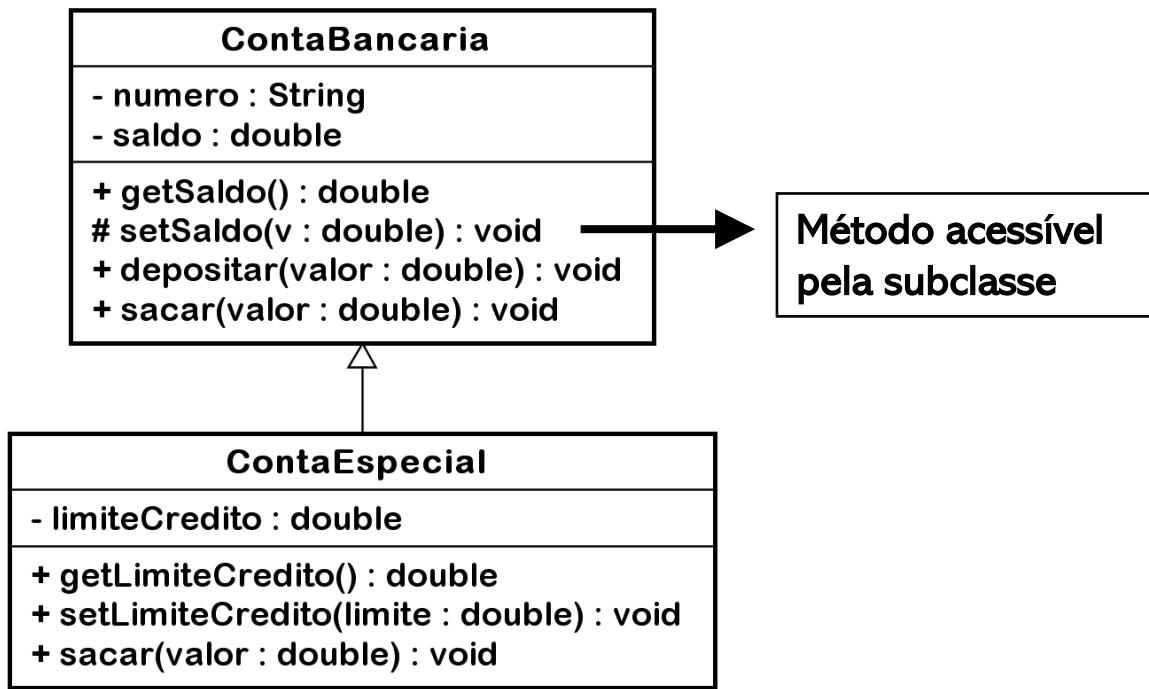
```
public void sacar(double valor) {
    if (valor > saldo)
        throw new RuntimeException("Sem saldo");
    saldo -= valor;
}
```

# Exemplo de sobrescrita de método

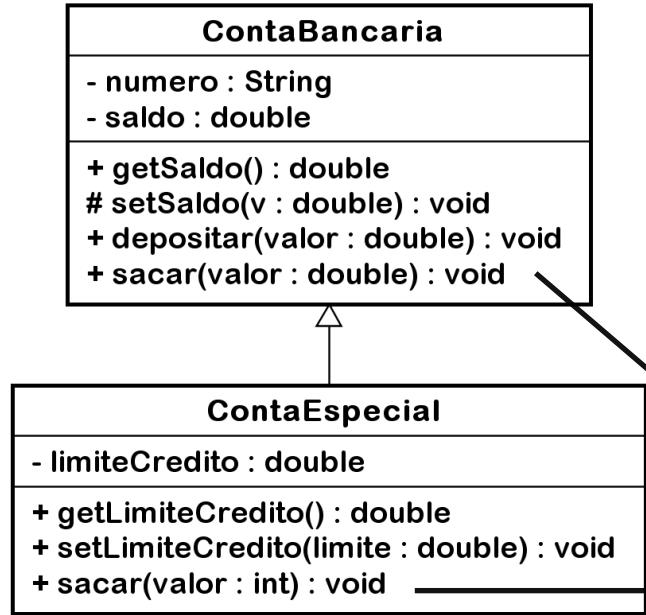


# Exemplo de sobrescrita de método

Como a subclasse precisa alterar o saldo, é preciso implementar o método `setSaldo()` na superclasse, porém de forma protegida.



# Sobrescrita de método



Na tentativa de sobrescrever um método, o programador pode escrever o nome ou parâmetros incorretos, e ao invés de sobrescrever o método, introduz um novo método sobrecarregado.

A classe ContaEspecial agora tem dois métodos sacar(), um para sacar valores com decimais e outro para sacar valores inteiros.

Para evitar este problema, na subclasse, recomenda-se utilizar a anotação `@Override` imediatamente antes do método para que o compilador confira se o método existe na superclasse.

# Sobrescrita de método

- É frequente o método sobreescrito de uma subclasse querer reutilizar o método da superclasse para executar parte do trabalho. Para isso, utilizar a palavra **super** para referenciar o método da superclasse.
- **Exemplo:**

```
@Override  
public void calcularTotal() {  
    setDesconto(10);  
    setIcms(7);  
    super.calcularTotal();  
}
```

# Classe Object

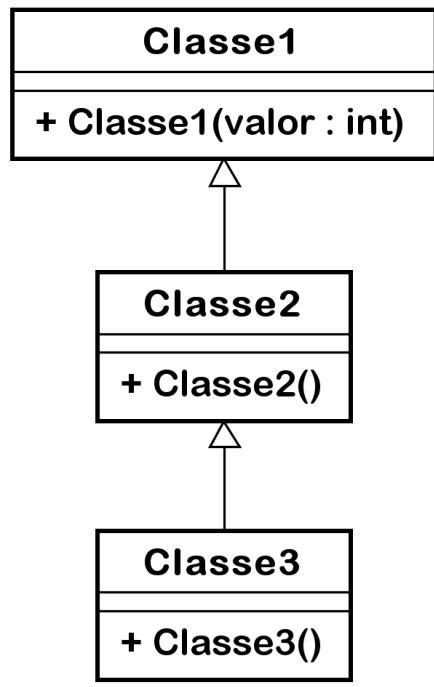
- Em Java, todas as classes herdam a classe Object (direta ou indiretamente), que está no pacote java.lang.

Object
+ Object()
+ hashCode() : int
+ equals(obj : Object) : boolean
+ toString() : String

Método	Descrição
equals(Object)	Compara o objeto atual e o objeto recebido como parâmetro, para conferir se são iguais
toString()	Retorna uma representação textual do objeto

# Herança X Construtores

Em Java, os construtores da superclasse não são herdados nas subclasses.



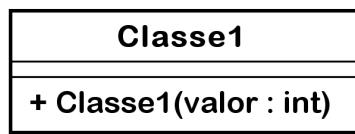
- Ao criar um objeto de uma classe, todos os seus construtores devem chamar algum construtor da superclasse imediata
- Para chamar o construtor da superclasse:

```
super();
```
- Exemplo:

```
public class Classe2 extends Classe1 {  
  
    public Classe2() {  
        super(30);  
        System.out.println("Classe2");  
    }  
}
```
- O comando `super()` deve ser o primeiro comando do construtor.

# Herança X Construtores

A chamada obrigatória de um construtor da superclasse leva a uma execução em cascata do construtor de todas as classes da hierarquia de herança



```
public class Classe1 {  
    public Classe1(int valor) {  
        System.out.println(valor);  
    }  
}  
  
public class Classe2 extends Classe1 {  
    public Classe2() {  
        super(30);  
        System.out.println("Classe2");  
    }  
}  
  
public class Classe3 extends Classe2 {  
    public Classe3() {  
        super();  
        System.out.println("Classe3");  
    }  
}
```

```
void executar() {  
    new Classe3();  
}
```



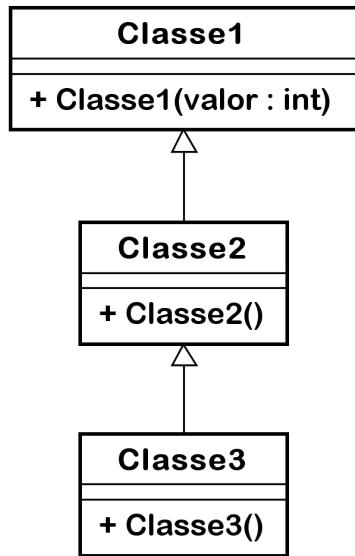
```
30  
Classe2  
Classe3
```

A execução do construtor ocorre da classe de mais alto nível até atingir a classe que está sendo instanciada.

# Construtores X Herança

- Se a superclasse imediata possuir um construtor padrão, este é chamado automaticamente em todos os construtores da subclasse
  - O compilador introduz o comando `super()` como sendo a primeira linha de código de cada construtor
  - Se a superclasse imediata não possuir um construtor padrão, o programador deverá chamar o construtor da superclasse de forma explícita.

# Construtores X Herança



```
public class Classe1 {
    public Classe1(int valor) {
        System.out.println(valor);
    }
}

public class Classe2 extends Classe1 {
    public Classe2() {
        super(30);
        System.out.println("Classe2");
    }
}

public class Classe3 extends Classe2 {
    public Classe3() {
        super();
        System.out.println("Classe3");
    }
}
```

O construtor da superclasse deve ser chamado explicitamente, já que a superclasse não tem construtor padrão

O compilador adiciona esta chamada automaticamente

# Construtores X Herança

```
public class Classe2 extends Classe1 {  
  
    public Classe2() {  
        super(30);  
        System.out.println("Classe2");  
    }  
  
}  
  
public class Classe3 extends Classe2 {  
  
}
```

Se nenhum construtor é declarado, Java introduz o construtor padrão, fazendo uma chamada ao construtor padrão da superclasse.

```
public class Classe3 extends Classe2 {  
  
    public Classe3() {  
        super();  
    }  
  
}
```

```
public class Classe1 {  
  
    public Classe1(int valor) {  
        System.out.println(valor);  
    }  
  
}  
  
public class Classe2 extends Classe1 {  
  
}
```

Se nenhum construtor é declarado e a superclasse não tem construtor padrão, o programador deve criar explicitamente um construtor

**Erro de compilação:** a introdução automática do construtor padrão aqui não compilaria também

# Diversos

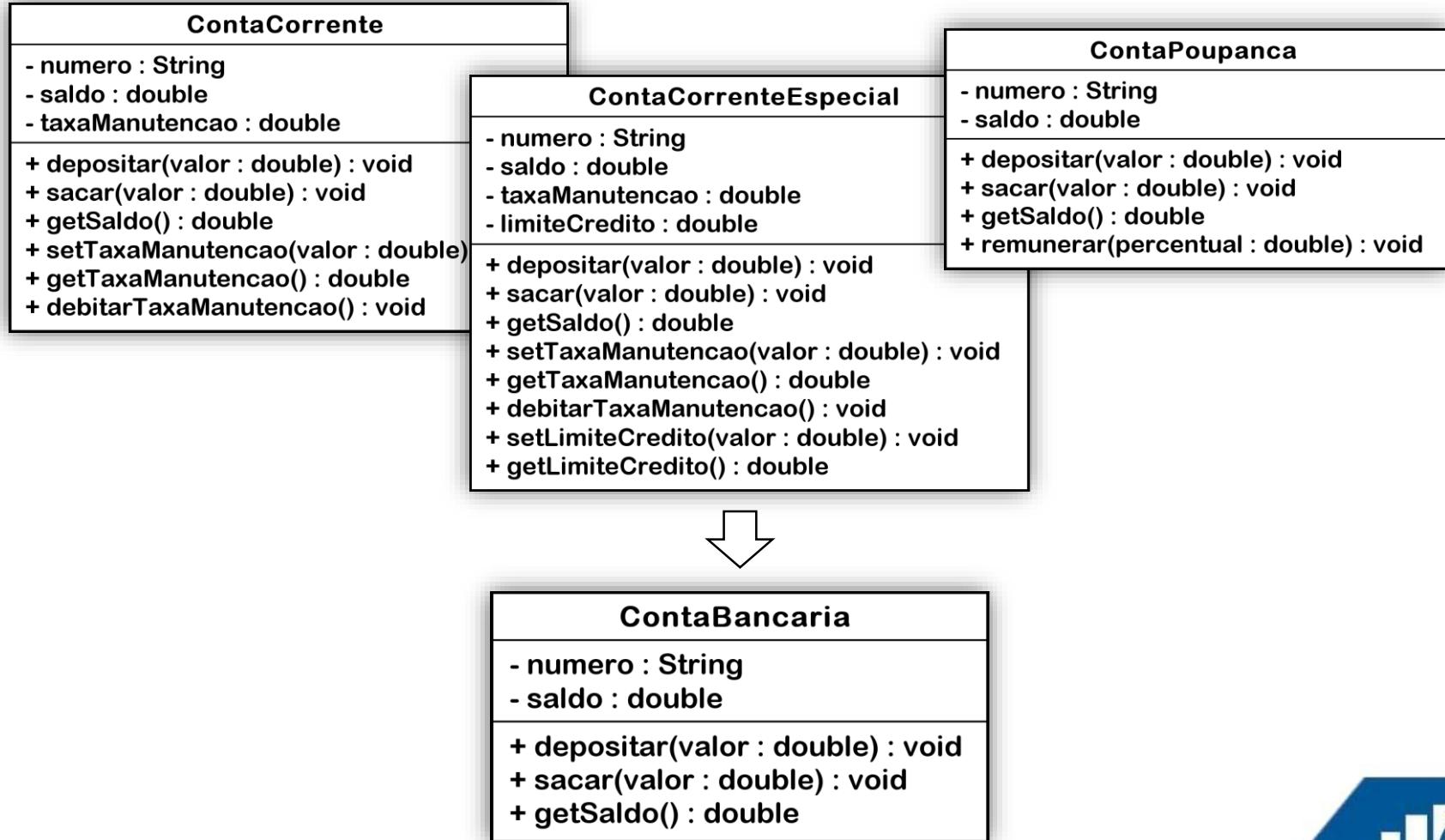
Java é uma linguagem de *herança simples*, ao contrário de algumas linguagens que são de *herança múltipla*, permitindo várias superclasses diretas para uma mesma classe

# Herança

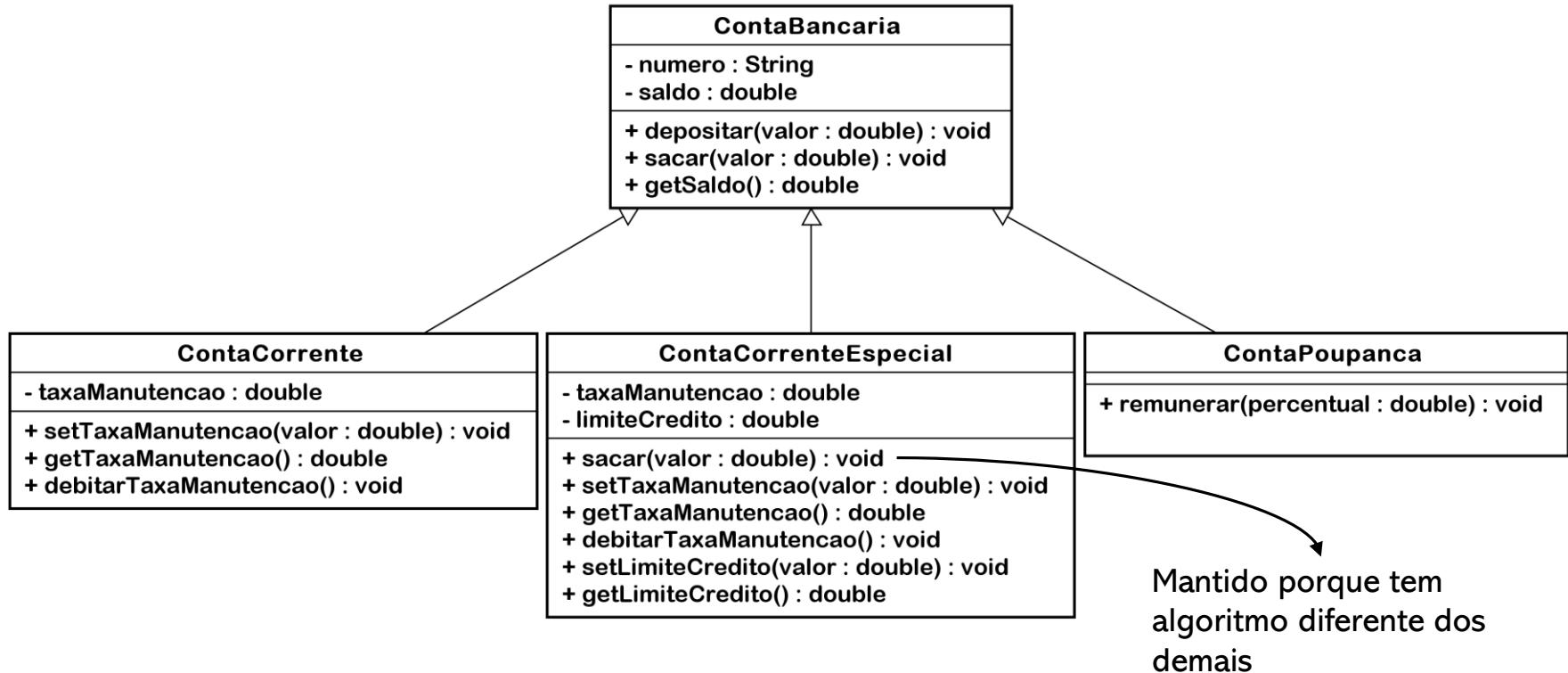
# Generalização

- É um processo de encontrar e criar superclasses:
  - Num conjunto de classes relacionadas, localizar os membros (métodos e atributos) comuns entre as classes;
  - Mover os membros comuns para uma classe, tornando-a superclasse das classes originais
  - Para os métodos que tiverem método diferenciado, manter na classe original

# Exemplo de aplicação



# Exemplo

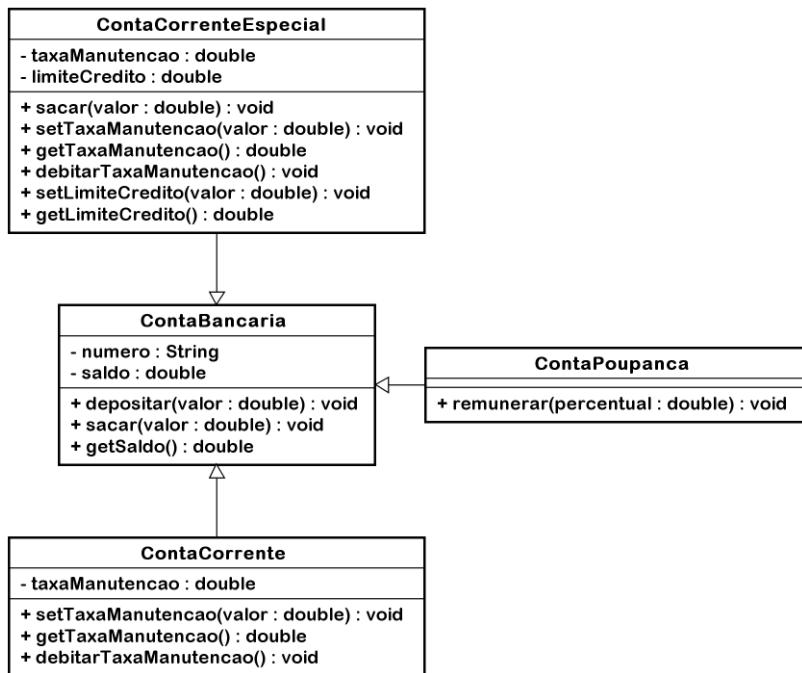


# Generalização

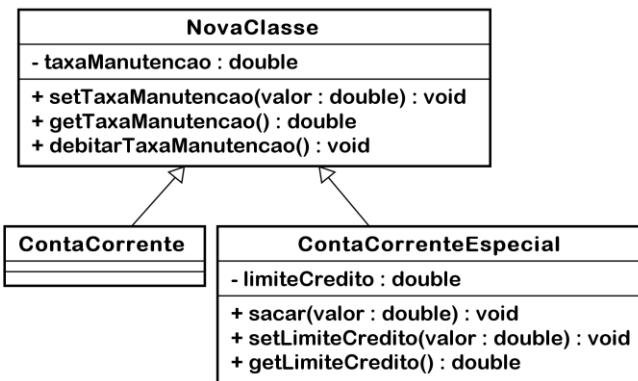
- O processo de generalização pode ser aplicado mais vezes, com um conjunto menor de classes
- Se após aplicar o processo de generalização, uma classe não possuir membro algum, esta classe pode se promovida para ser superclasse das demais classes.

# Exemplo

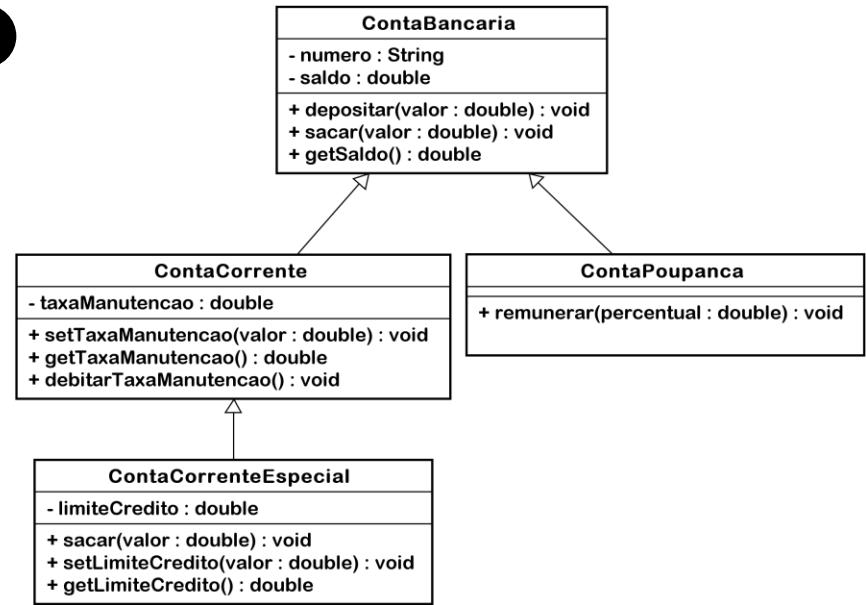
1



2



3



# Generalização

- Em muitas ocasiões, a superclasse que surgiu a partir do processo de generalização não tem um significado no mundo real
  - Trata-se de um efeito do processo de generalização
  - A superclasse pode representar um conceito abstrato que não existe no mundo real

# Classe abstrata

# Classe abstrata

- Uma classe que representa um conceito genérico
- Em UML, é expressa com o nome da classe em itálico

<i>ContaBancaria</i>
- numero : String - saldo : double
+ depositar(valor : double) : void + sacar(valor : double) : void + getSaldo() : double

- Em Java, utiliza-se o modificador *abstract*, como em:

```
public abstract class ContaBancaria {  
  
    // ...  
  
}
```

# Classe Abstrata

- Por ser uma abstração, não é possível criar objetos de classes abstratas

```
public abstract class Classe1 {  
    // ...  
}
```

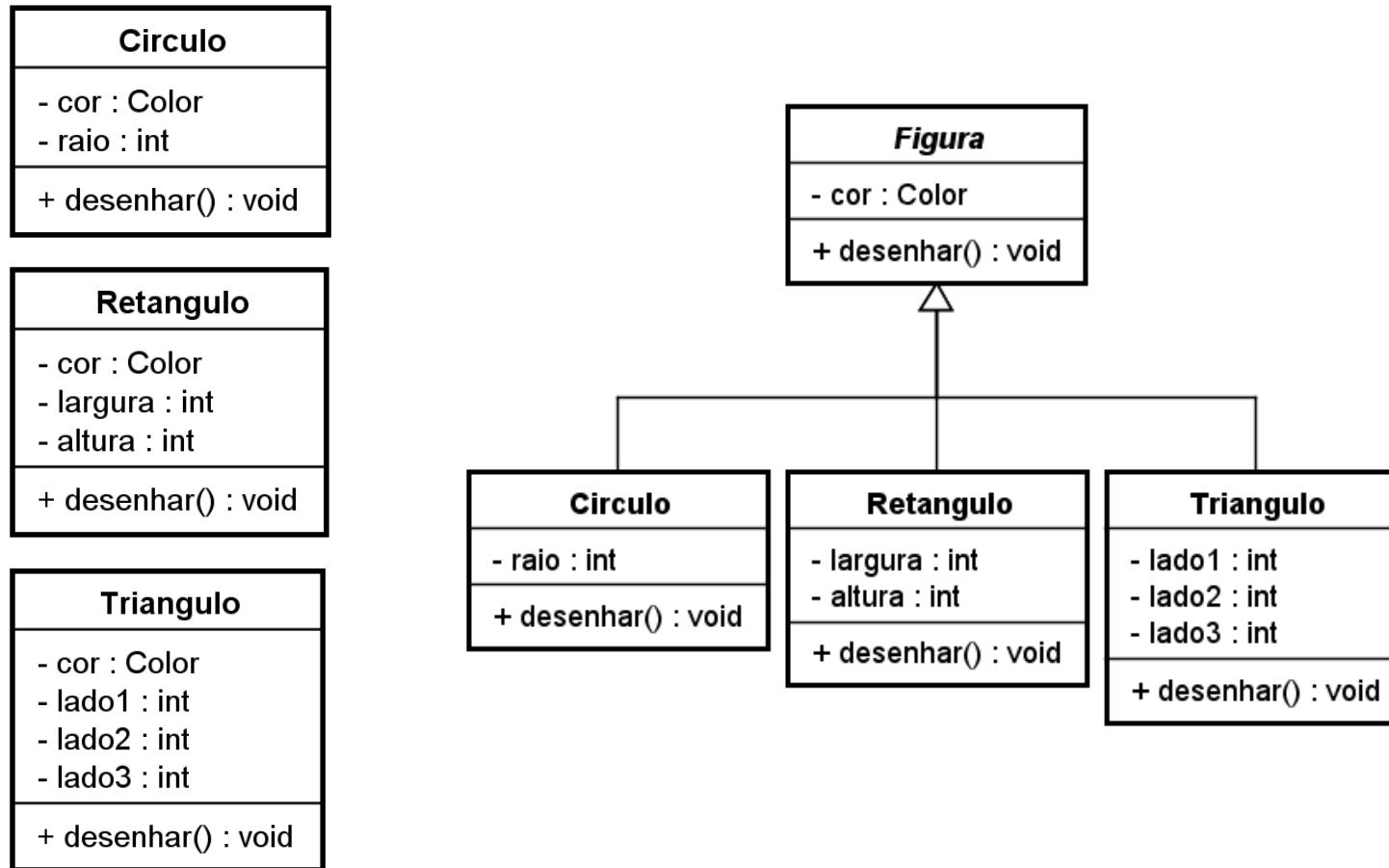
```
Classe1 c1 = new Classe1();
```



Erro de compilação: “não pode instanciar Classe1”

- Classes abstratas precisam ser estendidas para serem reusadas.
  - Classes que não são abstratas são conhecidas como **classes concretas**
  - Somente classes concretas podem ser instanciadas

# Motivação - Métodos abstratos



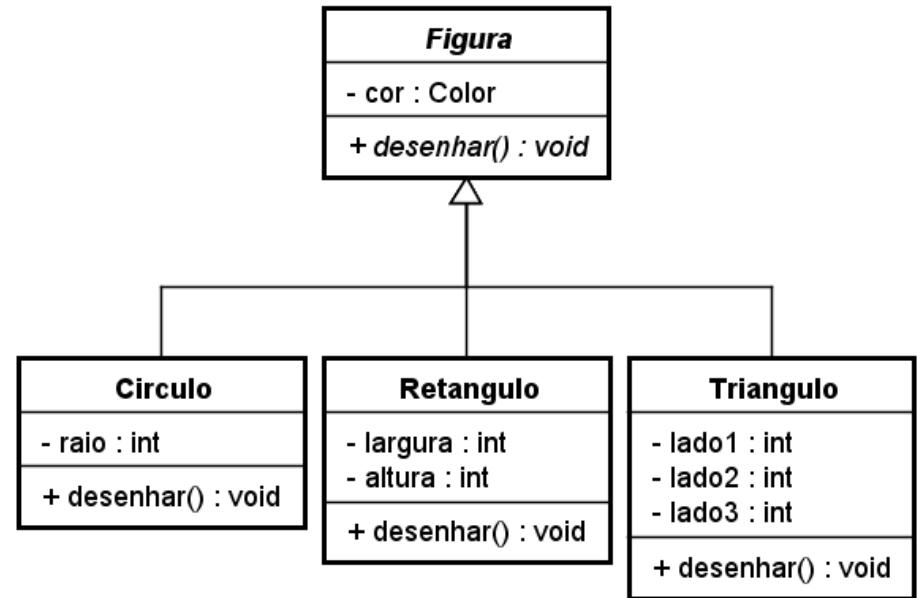
# Métodos abstratos

- Como evitar esta implementação?

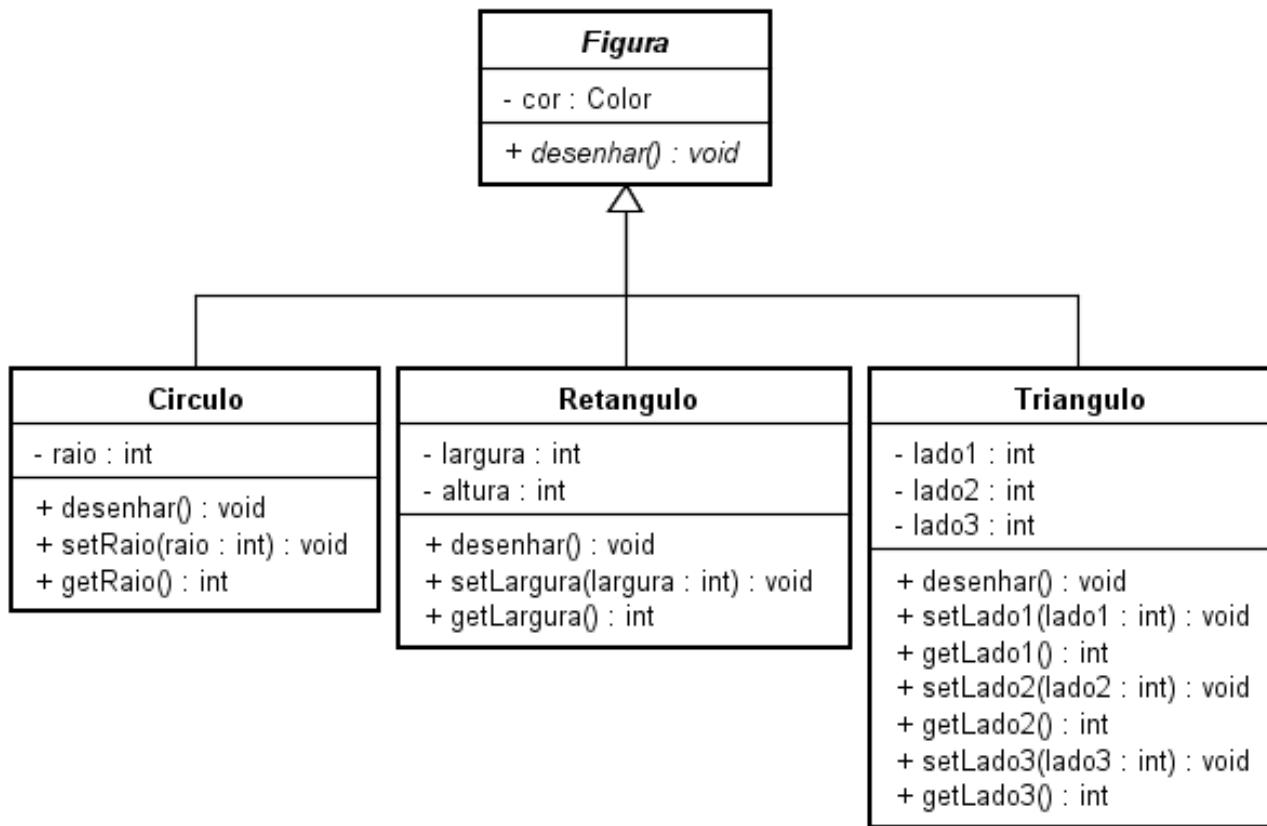
```
public class Figura {  
  
    private Color cor;  
  
    public void desenhar() {  
  
    }  
}  
  
public class Pentagono extends Figura {  
  
}
```

# Métodos abstratos

- Definindo um método como abstrato, instruímos o compilador a exigir que as subclasses implementem um método com tal assinatura
- Em UML, os métodos abstratos são escritos em itálico



# Exemplo



# Métodos abstratos

- Ao especificar o método desenhar() como abstrato:

```
public class Pentagono extends Figura {  
}
```



Erro de compilação: “Pentagono precisa implementar o método abstrato desenhar()”

# Métodos abstratos

- Um método abstrato é um método que não possui implementação, pois espera-se que as subclasses se encarreguem de implementá-lo.
- Declaramos um método abstrato para indicar que o método deve existir na subclasse, embora não há implementação para o método na superclasse
- Qualquer classe que contém um método abstrato deve ser abstrata também

# Método abstrato

- Métodos abstratos são declarados com o modificador `abstract` antes do tipo de retorno;
- Não possuem corpo;

```
public abstract class Figura {  
  
    private Color cor;  
  
    public abstract void desenhar();  
  
}
```

# Classes abstratas

- Uma classe abstrata pode ser especializada a partir de uma classe concreta

# **Classes e métodos que não podem ser estendidos/sobrescritos**

# Impedir que um método seja sobrescrito

- Em algumas ocasiões, um método possui uma implementação que não deveria estar sujeito a ser alterado através da sobrescrita de métodos em subclasses, pois a mudança de algoritmo pode tornar o estado do objeto inconsistente.
- Para impedir que um método seja sobrescrito na subclasse, utilizar a palavra reservada `final`, antes do tipo de dado de retorno, como em:

```
public final void metodo1() {  
    // ...  
}
```

# Impedir que uma classe seja estendida

- É possível impedir a extensão de uma classe. Para isso, utilizar a palavra reservada `final` antes da palavra `class`, como em:

```
public final class Classe1 {  
    // ...  
}
```

```
public class Classe2 extends Classe1 {
```

```
}
```



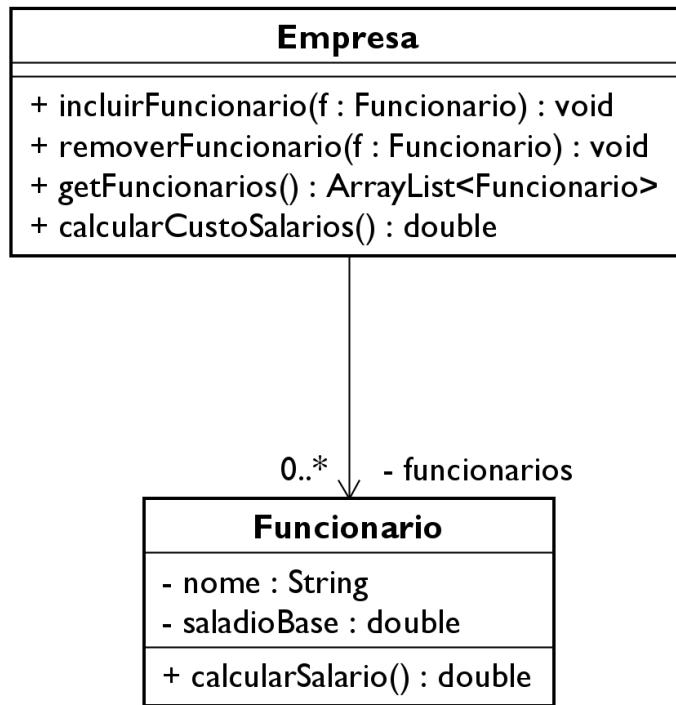
Erro de compilação

- Este recurso é útil, por exemplo, para criar classes de objetos imutáveis, como os objetos da classe `String`.

# Polimorfismo

# Motivação

Uma empresa quer saber quanto deve pagar de salário para todos os seus funcionários



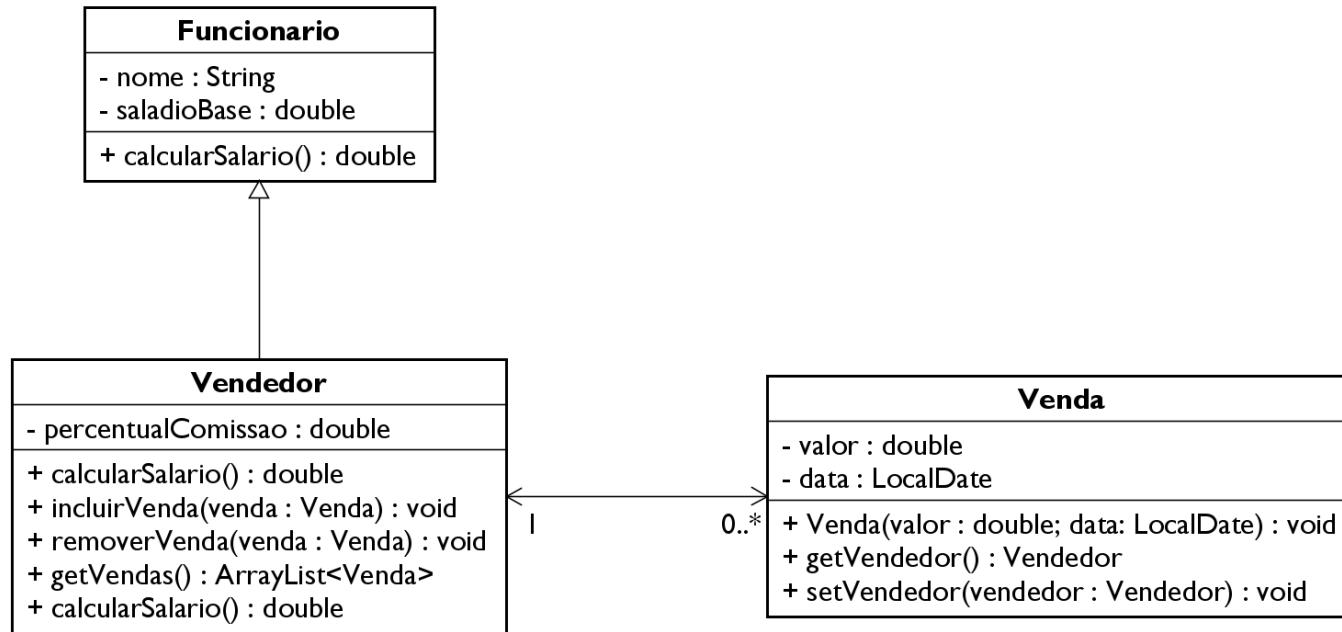
```
public double calcularCustoSalarios() {
    double total = 0;

    for (Funcionario f : funcionarios) {
        total += f.calcularSalario();
    }

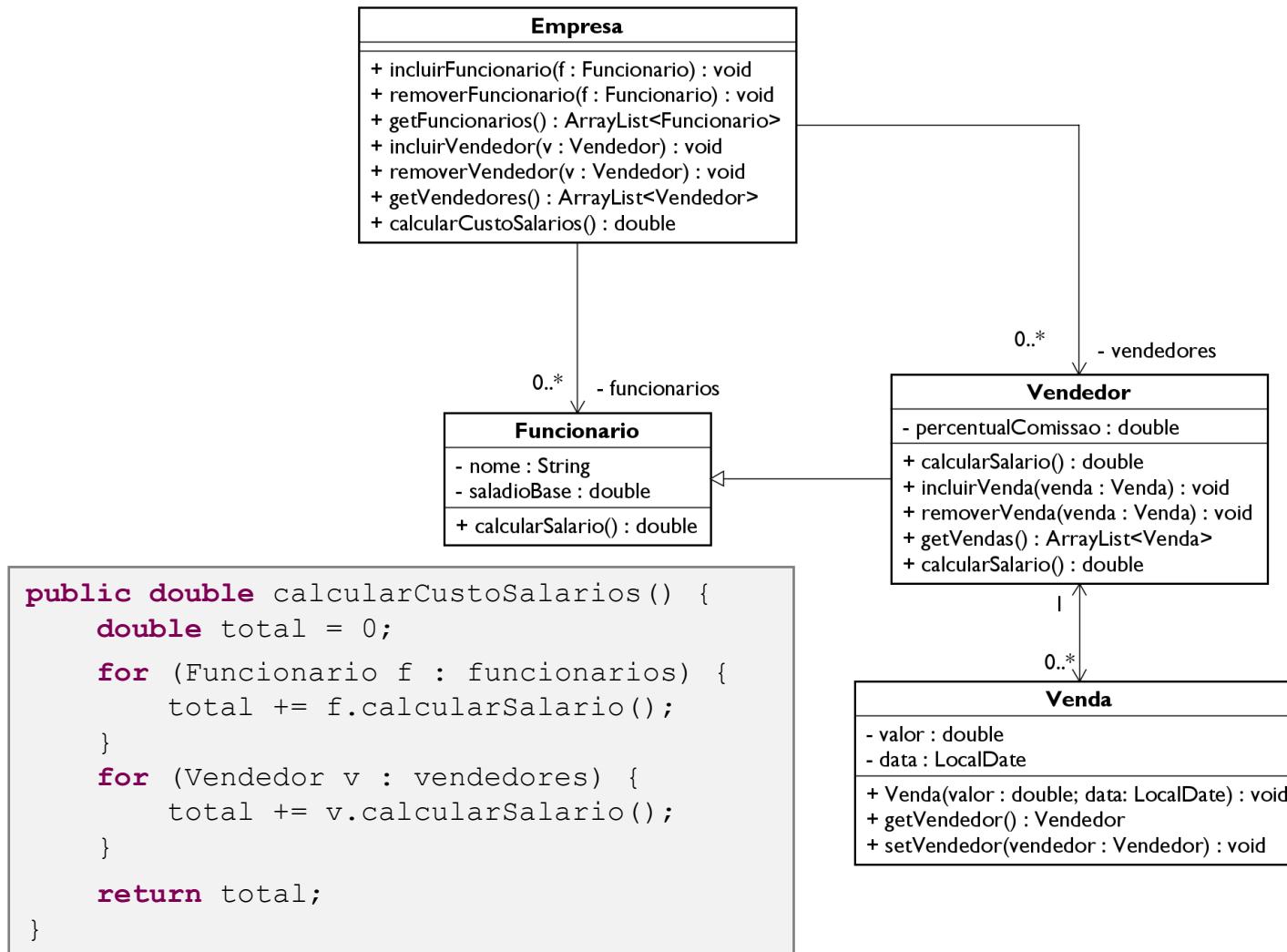
    return total;
}
```

# Motivação

Considerar que, existem alguns funcionários que trabalham com vendas. Os vendedores são comissionados



# Motivação

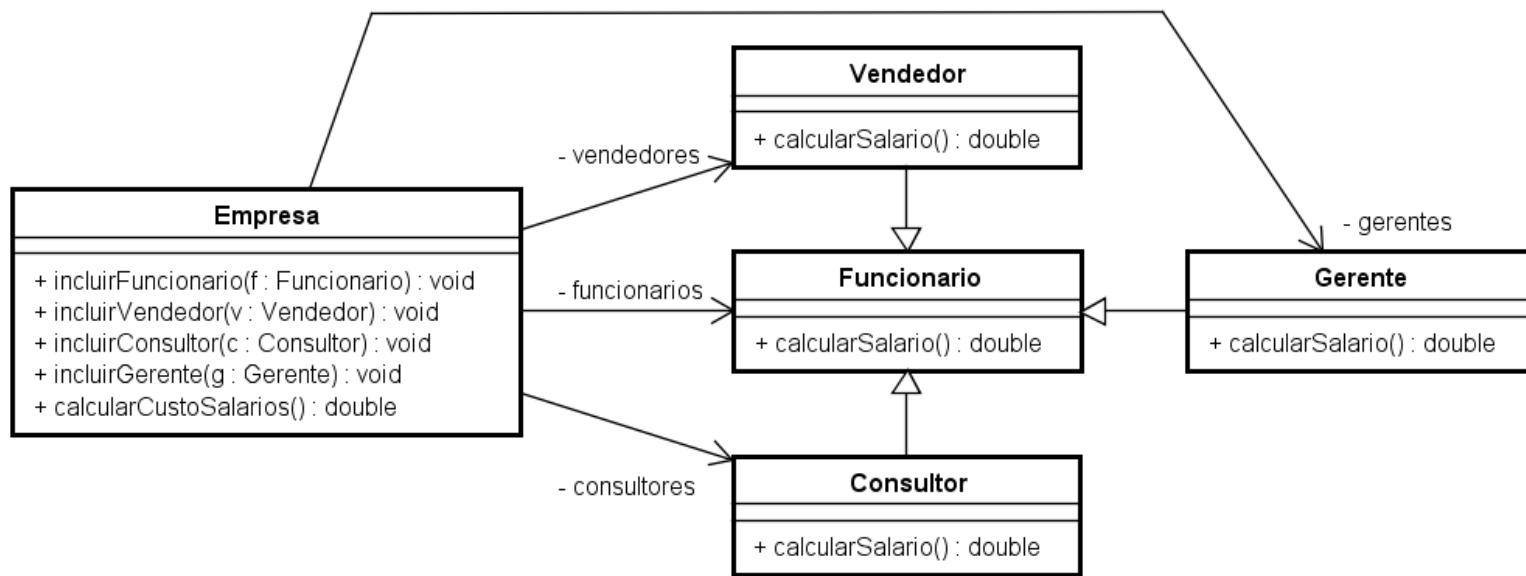


# Motivação

Considerar que a empresa possua também outros tipos de funcionários com cálculos de salários distintos.

*Consultor:* além de um valor fixo, o consultor recebe um valor extra a cada viagem a negócios que realizar

*Gerente:* além de um valor fixo, o gerente recebe um valor extra quando suas metas são atingidas



# Motivação

```
public double calcularCustoSalarios() {  
    double total = 0;  
  
    for (Funcionario f : funcionarios) {  
        total += f.calcularSalario();  
    }  
  
    for (Vendedor v : vendedores) {  
        total += v.calcularSalario();  
    }  
  
    for (Consultor c : consultores) {  
        total += c.calcularSalario();  
    }  
  
    for (Gerente g: gerentes) {  
        total += g.calcularSalario();  
    }  
  
    return total;  
}
```

# Polimorfismo

Polimorfismo é um conceito utilizado em Programação Orientada a Objetos para definir a habilidade que diferentes objetos tem de responder, cada um da sua maneira, à chamadas idênticas de mensagens (métodos).

# Polimorfismo

- Para atingir o polimorfismo, utilizamos uma **variável polimórfica**
  - Uma variável que pode referenciar tipos de objetos distintos
  - (por enquanto) é uma variável cujo tipo de dado é uma superclasse
- Dada uma variável polimórfica, ela pode referenciar:
  - Objetos de sua própria classe
  - Objetos cuja classe são subclasse (direta ou indireta) da classe da variável

# Exemplo

- Declaração de variável polimórfica:  
Funcionario f;
- Criação de objeto:  
f = **new** Consultor();
- Observar a existência de duas classes:
  - Tipo declarado na variável – tipo **estático**
  - Tipo utilizado para instanciar o objeto – tipo **dinâmico**

# Exemplo de polimorfismo

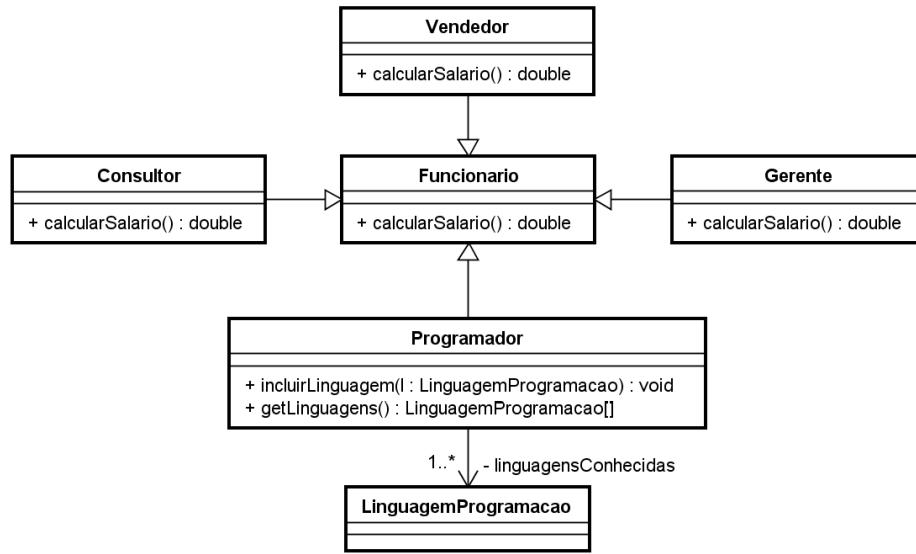
```
Funcionario[] funcionarios = new Funcionario[10];  
funcionarios[0] = new Funcionario("José Silva");  
funcionarios[1] = new Vendedor("Pedro Alcantara");  
funcionarios[2] = new Funcionario("Cristina Lima");  
funcionarios[3] = new Vendedor("Jorge Luna");  
funcionarios[4] = new Consultor("Marcia Cristina de Souza");  
...  
funcionarios[9] = new Gerente("Lucas Gentil");  
...  
for (Funcionario f : funcionarios) {  
    System.out.println(f.calcularSalario());  
}
```

O código Java que é  
acionado para atender  
`calcularSalario()` depende  
do tipo dinâmico

Variáveis  
polimórficas

Quando `f` referenciar o tipo  
dinâmico `Vendedor`, será acionado o  
método `calcularSalario()` da  
classe `Vendedor` e não o da classe  
`Funcionario`.

# Exemplo de polimorfismo



```
Funcionario f = new Programador("Márcio Santiago");
f.calcularSalario();
```

Quando o tipo dinâmico  
não sobrescreve o  
método, é acionado o  
código da superclasse

# Polimorfismo

- Um objeto da classe **Vendedor** pode ser acessado através de uma variável cujo tipo é igual a sua superclasse (direta ou indireta), isto é, através de uma variável polimórfica
- Por padrão, ao utilizar uma variável polimórfica, somente os membros da classe do tipo desta variável podem ser utilizados
- Isto é:

```
Funcionario f = new Programador();
```

...

```
f.calcularSalario();
```

→ compilável

```
f.incluirLinguagem(1);
```

→ Não compilável

# Downcasting

- Para acessar um método específico da subclasse, a partir de uma variável polimórfica, é necessário efetuar uma operação de conversão (cast). Esta operação é conhecida como *downcasting*.

```
Funcionario f = new Programador("Pedro");
```

```
LinguagemProgramacao l = new LinguagemProgramacao("C");
```

```
((Programador) f).incluirLinguagem(l);
```

# Polimorfismo

- Esta é uma operação ilegal e não compila.  
`Vendedor v = new Funcionario("André Simas");`
- Dada uma variável do tipo referência, é possível conferir se uma determinada classe pertence a sua hierarquia de classes.
- Exemplo:

```
for (Funcionario f : funcionarios) {  
    System.out.print("\nNome: " + f.getNome() +  
                    ". Salário Base: " + f.getSalarioBase());  
    if (f instanceof Vendedor) {  
        System.out.print(". Total vendas: " +  
                        ((Vendedor) f).calcularTotalVendas());  
    }  
}
```

# Exemplo de polimorfismo

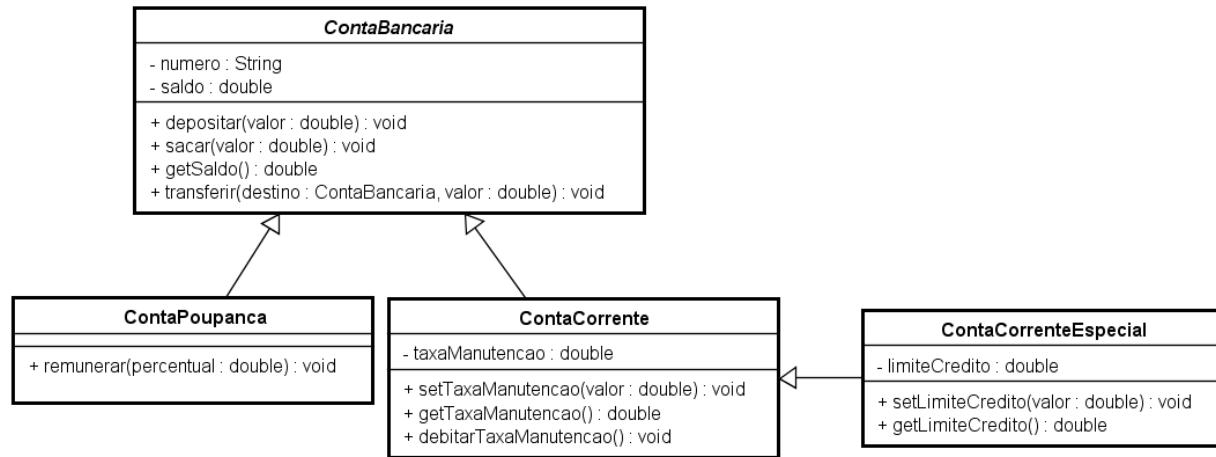
```
public class PrintStream {  
  
    public void println(Object x) {  
        String s = String.valueOf(x);  
        print(s);  
        newLine();  
    }  
  
}
```

```
public final class String {  
  
    public static String valueOf(Object obj) {  
        return (obj == null) ? "null" : obj.toString();  
    }  
  
}
```

# Polimorfismo

- O polimorfismo permite que os desenvolvedores programem de forma mais abstrata
- Permite evitar comandos condicionais para tratamento de casos especiais

# Exemplo 2

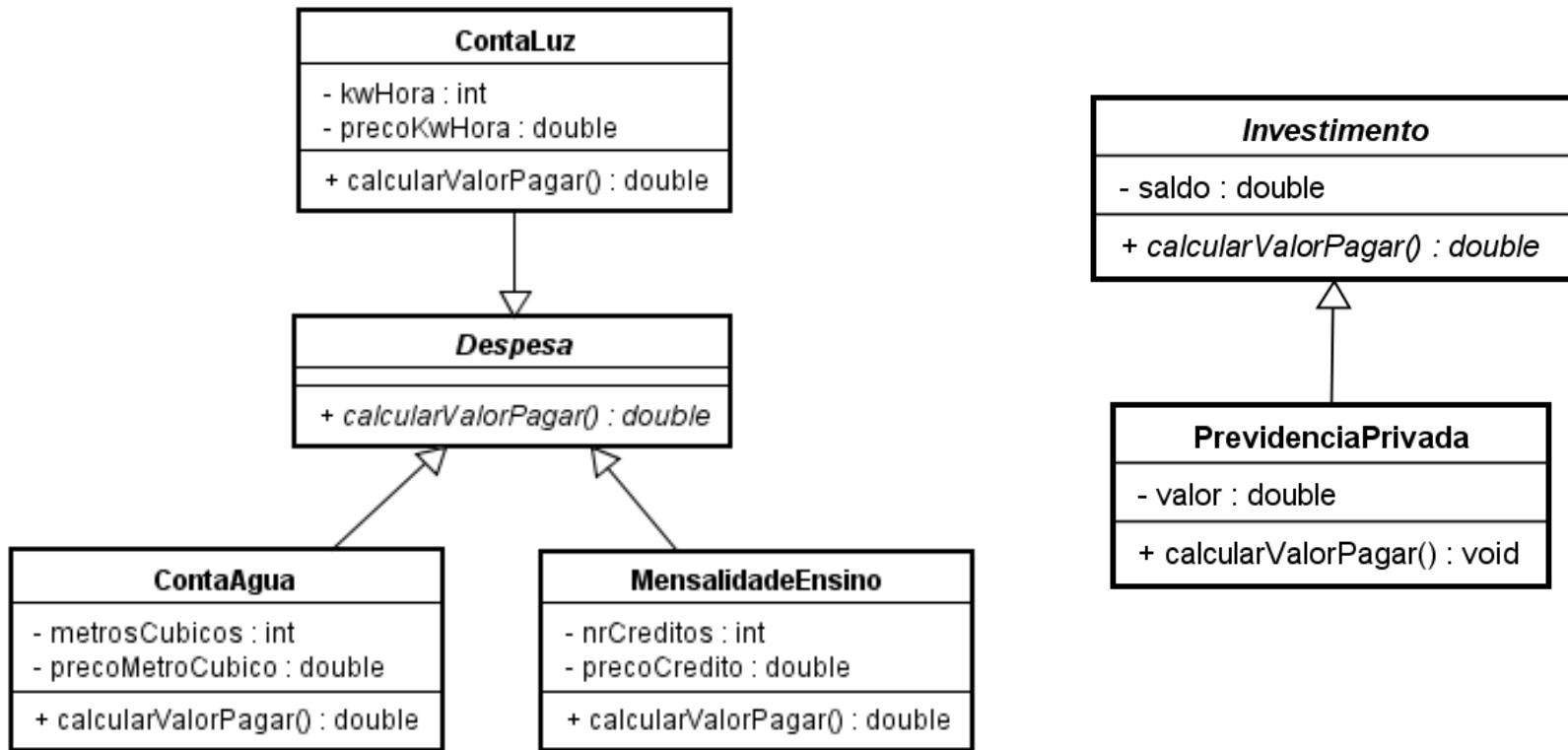


Método **transferir()** da classe **ContaBancaria**:

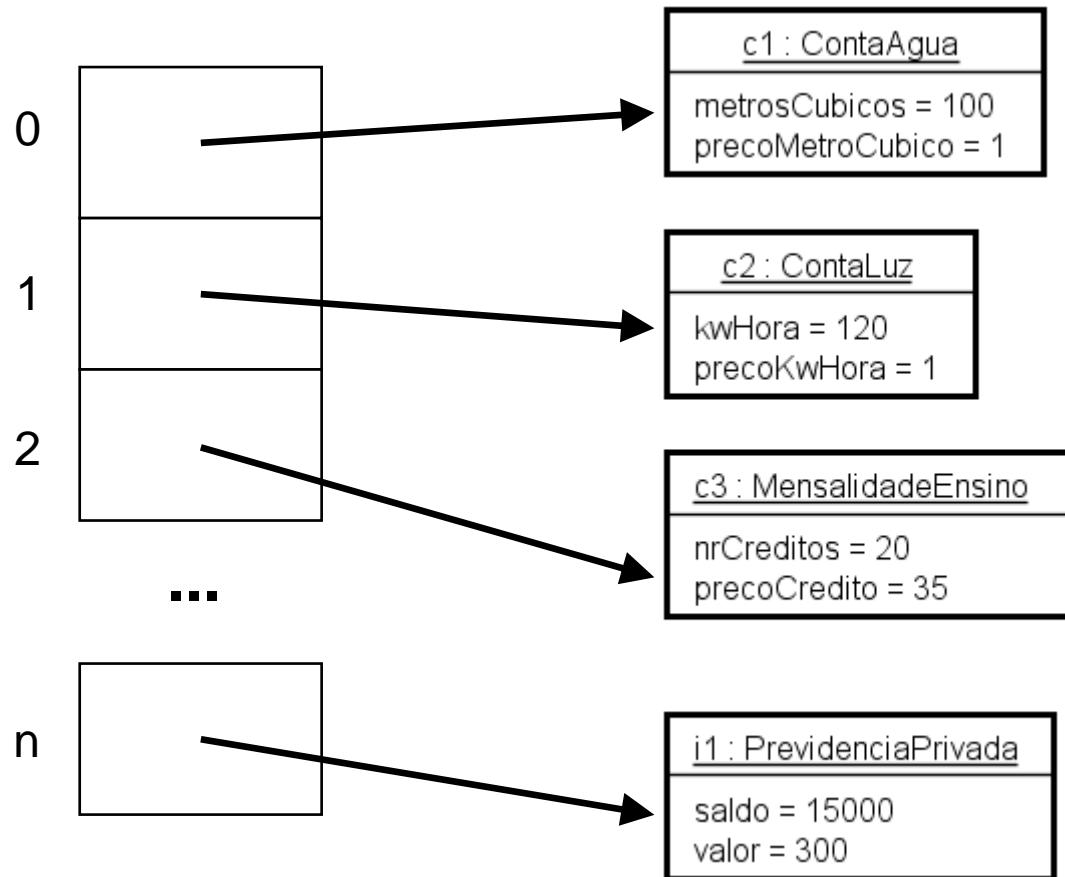
```
public void transferir(ContaBancaria contaDestino, double valor) {
    this.sacar(valor);
    contaDestino.depositar(valor);
}
```

# Interface

# Motivação



# Motivação



# Motivação

```
Object[] pagamentos = new Object[4];

pagamentos[0] = new ContaAgua();
pagamentos[1] = new ContaLuz();
pagamentos[2] = new PrevidenciaPrivada();
pagamentos[3] = new MensalidadeEstudo();

...
double totalPagamentos = 0;

for (Object o : pagamentos) {
    if (o instanceof Despesa) {
        totalPagamentos += ((Despesa) o).calcularValorPagar();
    } else if (o instanceof Investimento) {
        totalPagamentos += ((Investimento) o).calcularValorPagar();
    }
}

System.out.println("Total de pagamentos = " + totalPagamentos);
```

# Interface

- Interfaces permitem especificar um conjunto de comportamentos desejáveis em objetos. Pode ser aplicado à objetos de classes distintas (não relacionadas)
- A interface contém um conjunto de definições de métodos e constantes.
  - A *Interface* não contém implementação de métodos
  - Também não contém variáveis

# Interface

- Classes podem implementar uma ou mais interfaces
- A classe que implementa uma interface concorda em implementar todos os métodos definidos pela interface, portanto, concorda em desempenhar determinado(s) comportamento(s).

# Em UML

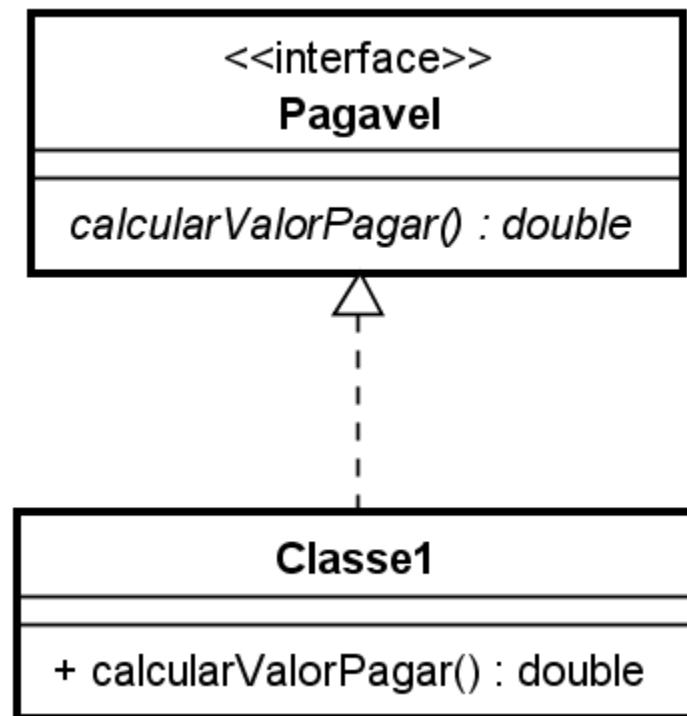
- Uma interface pode ser representada como uma classe estereotipada.

```
<<interface>>
NomeDaInterface
-----
operacao1() : void
operacao2() : void
```

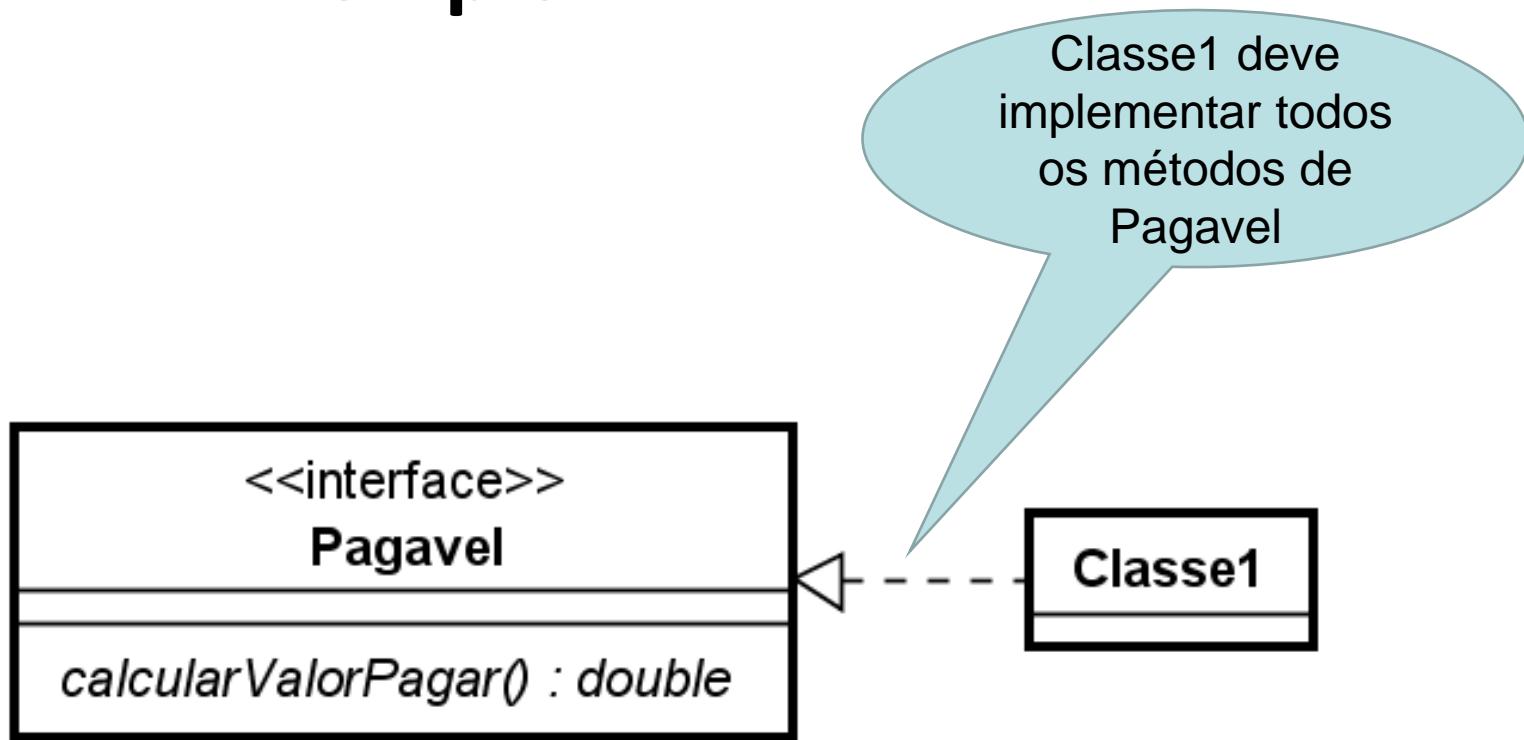
```
<<interface>>
Pagavel
-----
calcularValorPagar() : double
```

# Exemplo

Para indicar que uma classe desempenha o(s) comportamento(s) de uma interface, utilizamos o relacionamento de “realização”.



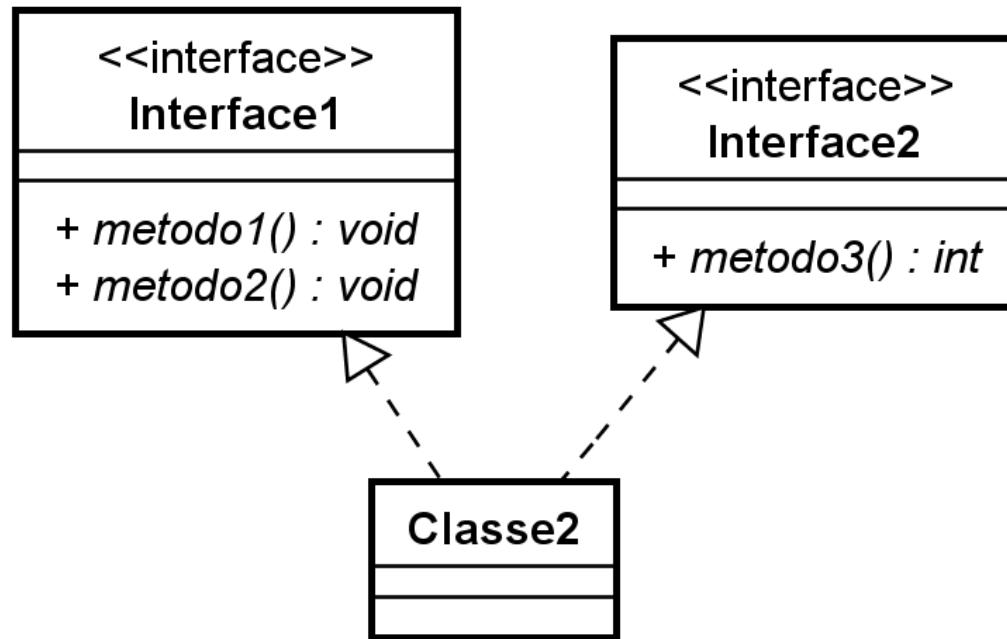
# Exemplo



Afirmamos que:  
Classe1 implementa a interface Pagavel ou  
Classe1 realiza a interface Pagavel

# Múltiplas Interfaces

- Em Java, uma classe pode implementar múltiplas interfaces



# Tradução para a linguagem Java

<<interface>>

Pagavel

*calcularValorPagar() : double*

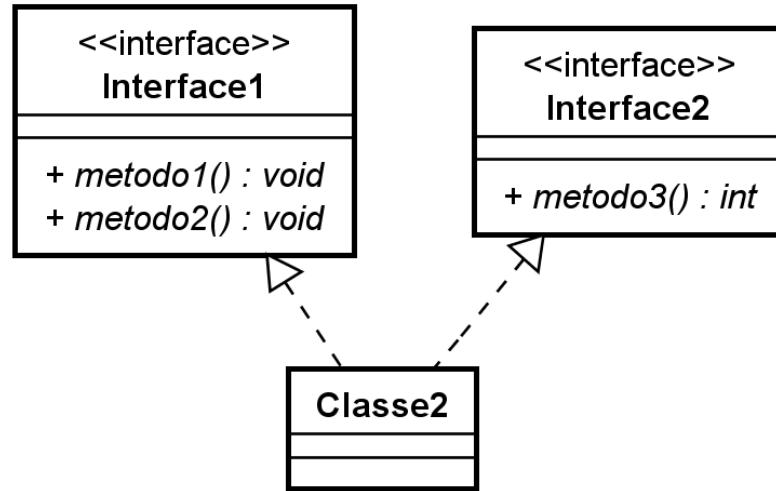
```
public interface Pagavel {  
  
    double calcularValorPagar();  
  
}
```

# Tradução para a linguagem Java

- Utilizamos a palavra reservada **implements** para anunciar as interfaces que a classe realiza.

```
public class Classe2 implements Pagavel {  
  
    @Override  
    public double calcularValorPagar() {  
        ...  
    }  
}
```

# Tradução para a linguagem Java



```
public class Classe2 implements Interface1, Interface2 {  
  
    public void metodo1() {...}  
    public void metodo2() {...}  
    public int metodo3() {...}  
  
}
```

# Polimorfismo X Interface

- O conceito de polimorfismo também pode ser aplicado quando tratamos de objetos de classes que implementam interfaces.
- Podemos criar uma variável polimórfica definindo que seu tipo é uma *interface*.

# Polimorfismo X Interface

- Exemplo:

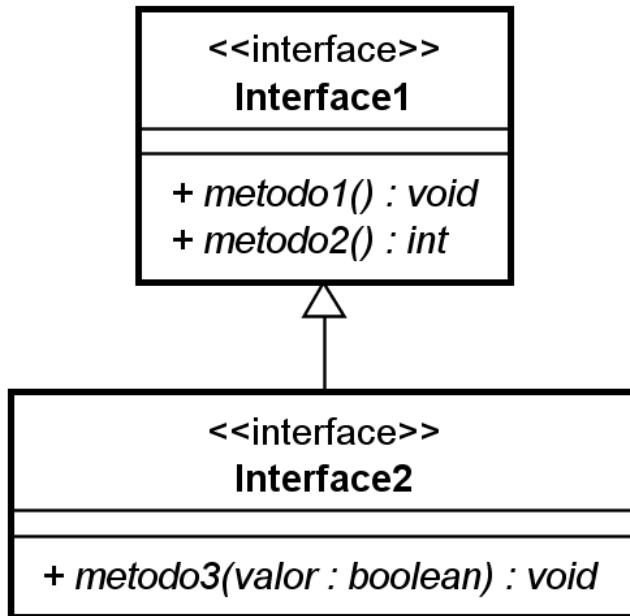
```
Pagavel[] pagamentos = new Pagavel[4];  
  
pagamentos[0] = new ContaAgua();  
pagamentos[1] = new ContaLuz();  
pagamentos[2] = new PrevidenciaPrivada();  
pagamentos[3] = new MensalidadeEstudo();  
  
double totalPagamentos = 0;  
  
for (Pagavel p : pagamentos) {  
    totalPagamentos += p.calcularValorPagar();  
}  
  
System.out.println("Total de pagamentos = " + totalPagamentos);
```

# Observações

- Os métodos declarados numa interface não precisam utilizar modificador de acesso.
  - O único que é admissível é *public*.
- Ao indicar que uma classe abstrata implementa uma interface, a classe abstrata não precisa implementar os métodos da interface, porém suas subclasses concretas devem implementar (subclasses diretas abstratas também não precisam).

# Herança de interface

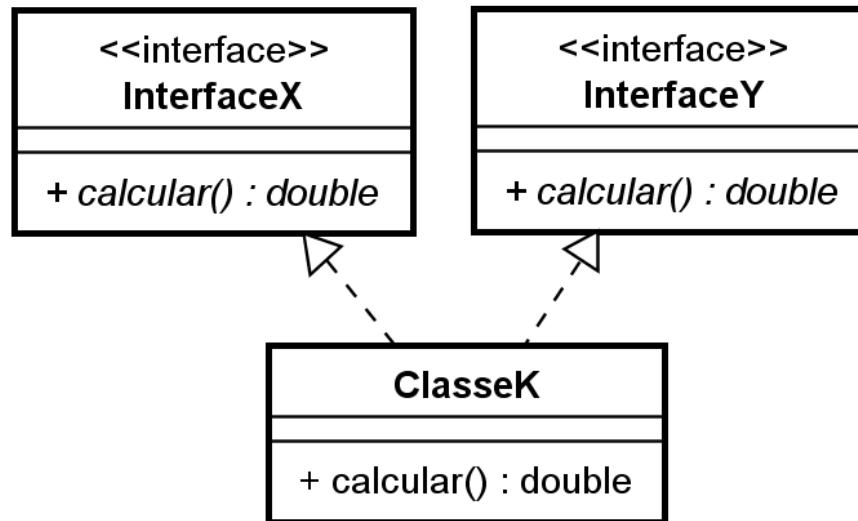
- Uma interface pode herdar a assinatura de métodos de outra interface



A classe que quiser implementar a interface *Interface2* deve possuir algoritmos para os métodos:

- *metodo1()*
- *metodo2()*
- *metodo3()*

# Herança de interface



```
public class ClasseK implements InterfaceX, InterfaceY {  
  
    public double calcular() {  
        return 0;  
    }  
}
```

# Interface no Java8

# Interface no Java 8

- Incluir novos métodos em interfaces “antigas” requer implementar estes métodos novos;
- Com Java 8 é possível definir uma “implementação padrão” para um método.

```
public interface Interface1 {  
  
    void metodo1(String str);  
  
    default void log(String str) {  
        System.out.println("Log:"+str);  
    }  
  
}
```

- Como interface não definem variáveis, métodos com implementação limitam-se a utilizar dados originados de parâmetros

# Tratamento de Exceções

# Tipos de erros

- Durante a execução de um programa, podemos nos deparar com dois tipos de erros:
  - Erros de lógica
  - Erros de execução

# Erros de lógica

- São erros de concepção de algoritmo

```
double[] nota = new double[3];
nota[0] = 10;
nota[1] = 7;
nota[2] = 8;

double somaNotas = 0;
for (int i=0; i<=nota.length; i++) {
    somaNotas += nota[i];
}

System.out.println("A média é: " + (somaNotas/nota.length));
```

- Os erros de lógica **devem** ser evitados

# Erros de execução

- Erros causados por operações que não possuem suporte pelo ambiente. Exemplos:
  - Entrada de dados inadequada
  - Manipulação indevida de arquivos
  - Operação aritmética ilegal
  - Comando não atendido pelo periférico
  - Falta de memória
- Os erros de execução correspondem a condições excepcionais ou anormais, frequentemente chamadas de **exceções**
- Os erros de execução, muitas vezes, não podem ser evitados.
  - Na ocorrência de tais erros, como o programa deve se comportar?

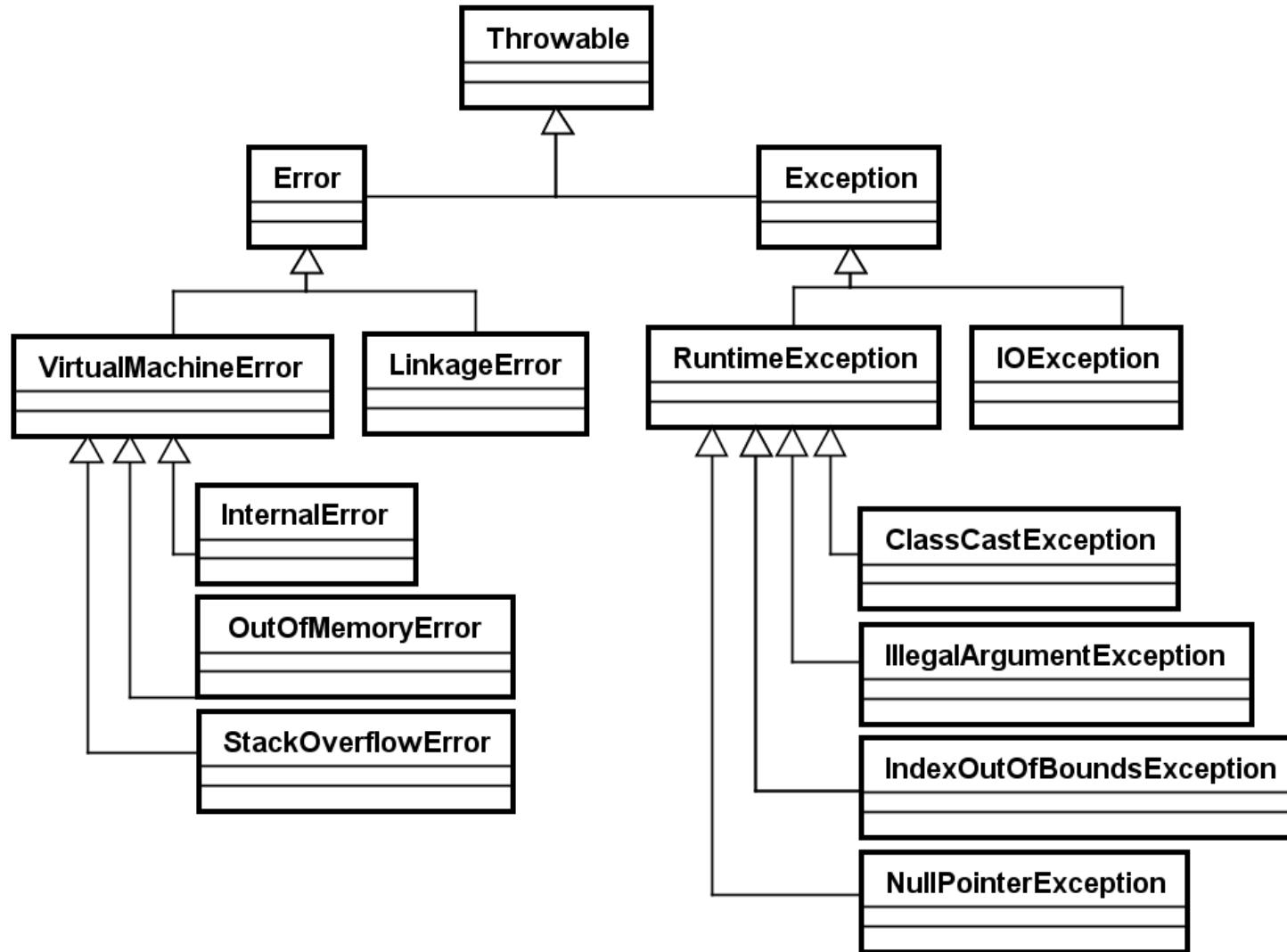
# Erros

- Se o usuário digitar uma URL incorreta no navegador de internet, o que deveria acontecer?
  - Nada – O navegador finaliza
  - O navegador solicita outra URL
- Dependendo da natureza da exceção, o programa pode tratar adequadamente a situação. Chamamos este procedimento de **tratamento de exceção**.
- Quando uma exceção não é tratada, o programa é abortado – finaliza de forma anormal

# Erros em Java

- Os erros quando ocorrem são detectados pela JVM
- A JVM cria um objeto que caracteriza o erro
- O programa que causou o erro é notificado pela JVM
- O programa pode tratar o erro
  - Podemos acessar o objeto criado pela JVM ao tratar a exceção
- Java possui vários tipos de classe que representam erros. As principais são:
  - Classe Error
  - Classe Exception

# Hierarquia de classes de erros



# Hierarquia de classes de erros

- A classe `Error` e todas suas subclasses são reservadas para indicar erros graves.
  - Não se espera que sejam tratados pelos programas
- A classe `Exception` e todas suas subclasses indicam erros que poderiam ser tratados pelos programas.

# Tratamento de exceções

- Embora a JVM detecte erros, o programador também pode criar explicitamente objetos para sinalizar situações de erro;
- Também é possível criar novas classes de erro, desde que sejam suclasses de Throwable ou de alguma de suas subclasses.

# Como tratar exceções

- O tratamento de exceções é feito através do comando `try`, cuja sintaxe básica consta abaixo:

```
try {  
    comando  
    comando  
    comando  
    ...  
} catch (ClasseExcecao objeto) {  
    comando caso ocorra erro  
    comando caso ocorra erro  
    ...  
}
```

O comando `catch` exige um parâmetro. Podemos definir qualquer classe que estenda de `Throwable`

- Qualquer erro ocorrido num dos comandos do bloco `try` causará o desvio do fluxo de execução para o bloco `catch`

# O comando try..catch

No início do comando try, o ambiente passa a monitorar a execução de todos os comandos declarados em seu corpo. Se nenhum erro ocorrer durante a execução dos comandos monitorados, então o comando try termina normalmente e os comandos de bloco catch não são executados

# Exemplo

```
1 Scanner teclado = new Scanner(System.in);
2 int idade;
3 try {
4     idade = Integer.parseInt(teclado.nextLine());
5 } catch (NumberFormatException objErro) {
6     System.out.println("Valor incorreto. ");
7 }
8 System.out.println("Fim")
```

Dizemos que este fragmento é capaz de “capturar” o erro NumberFormatException

# Exemplo

```
1 Scanner teclado = new Scanner(System.in);
2 int num;
3
4 System.out.println("Digite um número: ");
5 while (true) {
6     try {
7         num = Integer.parseInt(teclado.nextLine());
8         System.out.println("O número informado é: " + num);
9         break;
10    } catch (NumberFormatException objErro) {
11        System.out.println("Valor incorreto. Por favor, digite novamente");
12    }
13 }
```

# Múltiplos catch

- O comando try...catch aceita várias cláusulas catch.

```
1 Scanner teclado = new Scanner(System.in);
2
3 try {
4     int a[] = new int[2];
5     a[4] = 30 / Integer.parseInt(teclado.nextLine());
6     System.out.println("Operação concluída com êxito");
7 } catch (NumberFormatException e) {
8     System.out.println("Valor digitado é inválido");
9 } catch (ArithmetricException e) {
10    System.out.println("Falha na divisão");
11 } catch (ArrayIndexOutOfBoundsException e) {
12    System.out.println("Não conseguiu atribuir ao vetor");
13 } catch (Exception e) {
14    System.out.println("Qualquer outra exceção");
15 }
16 System.out.println("Fora do bloco");
```

# Múltiplos catch

- Ao ocorrer um erro causado por um dos comandos do bloco try, um objeto de uma classe que caracteriza o erro é criado e o programa é notificado.
- O comando que causou o erro é interrompido.
- Em seguida, as cláusulas catch são verificadas, na ordem em que foram implementadas, para verificar se há alguma que pode tratar a exceção
  - Quando uma cláusula catch que pode tratar o erro é encontrada, o fluxo é desviado para o primeiro comando do bloco desta cláusula. Ao fim da execução desta cláusula, o comando try é finalizado normalmente e o fluxo prossegue após o comando try.

# Tratando erros através de polimorfismo

- Um erro de uma dada classe pode ser capturado por uma cláusula catch que tenha como parâmetro uma variável da mesma classe ou de uma superclasse do erro ocorrido, utilizando uma variável polimórfica

```
1 try {  
2     criarArquivo();  
3     gravarArquivo();  
4 } catch (IOException e) {  
5     System.out.println("Falha ao manipular arquivo");  
6 }
```

# Tratando erros através de superclasse

- Como o fluxo da execução é desviado para a primeira cláusula-catch que pode tratar a exceção, não é válido declarar cláusulas que especifiquem classes de erros que sejam subclasses de classes especificadas em cláusulas catch anteriores.

```
1 try {  
2     int a[] = new int[2];  
3     a[4] = 30 / Integer.parseInt(teclado.nextLine());  
4     System.out.println("Operação concluída com êxito");  
5 } catch (Exception e) {  
6     System.out.println("Aconteceu algum erro...");  
7 } catch (NumberFormatException e) {  
8     System.out.println("Valor digitado é inválido");  
9 } catch (ArithmetricException e) {  
10    System.out.println("Falha na divisão");  
11 } catch (ArrayIndexOutOfBoundsException e) {  
12    System.out.println("Não conseguiu atribuir ao vetor");  
13 }
```

# Mesmo tratamento para várias classes de erro

- A partir do Java 7, é possível definir uma cláusula catch capaz de tratar várias classes de erro.
- Exemplo:

```
1 try {  
2     ...  
3 } catch ( IOException | SQLException ex ) {  
4     ...  
5 }
```

# Propagação de erros

- Quando um método é interrompido por erro, o controle retorna ao método chamador, que poderá tratar o erro.

```
1 public static int obterValor() {  
2     Scanner teclado = new Scanner(System.in);  
3     int valor = Integer.parseInt(teclado.nextLine());  
4     return valor;  
5 }  
6  
7 public static void teste1() {  
8     while (true) {  
9         try {  
10             int num = obterValor();  
11             System.out.println("Número informado = " + num);  
12             break;  
13         } catch (NumberFormatException e) {  
14             System.out.println("Valor inválido. Informe novamente");  
15         }  
16     }  
17  
18     System.out.println("Fim");  
19 }
```

# Propagação de erros

- Se o método chamador não puder tratar o erro que causou a interrupção, também será interrompido.
- A propagação do erro ocorre até que ele possa ser tratado por algum método da cadeia de chamada de métodos.
  - O último método que tem “oportunidade” de tratar o erro é o método `main()`.
  - Se o método `main()` não tratar o erro, o programa é interrompido

# Exceções verificadas pelo compilador

- Alguns métodos podem solicitar ao compilador exigir que o programador trate a exceção no local em que ocorreram ou devem ter o tratamento explicitamente postergado.
- São os erros das **classes verificáveis**. As classes Error e RuntimeException, bem como todas as suas subclasses, não são verificáveis. As demais classes são verificáveis.

# Exceções verificadas pelo compilador

```
public void abrirArquivo() {  
    arquivo = new FileReader("arquivo.txt");  
}
```

Este método causa erro de compilação porque FileReader pode lançar uma exceção da classe FileNotFoundException e deve ser tratado

```
public void abrirArquivo() {  
    try {  
        arquivo = new FileReader("arquivo.txt");  
    } catch (FileNotFoundException e) {  
        System.out.println("Arquivo não encontrado");  
    }  
}
```

# Cláusula throws

- A cláusula throws é utilizada num método para informar que o método chamador deverá tratar a possibilidade daquelas exceções ocorrerem
- Utilizado com classes de erros verificáveis
- Quando um método utiliza a classe throws, ele delega o tratamento daquelas exceções para os métodos chamadores

```
public void abrirArquivo() throws FileNotFoundException {  
    arquivo = new FileReader("arquivo.txt");  
}
```

- É possível informar várias exceções na cláusula throws, basta separá-las por vírgula.

# Cláusula finally

- A cláusula `finally` está associada ao comando `try` e é utilizado para definir um conjunto de comandos que sempre serão executados, quer o comando `try` execute com ou sem erros.
- Sintaxe:

```
try {  
    comando  
    comando  
    ...  
} catch (ClasseExcecao objeto) {  
    comando caso ocorra erro  
    comando caso ocorra erro  
    ...  
} finally {  
    comando...  
    comando...  
}
```

# Cláusula finally

- O uso da cláusula `finally` geralmente está associado à liberação de recursos alocados durante o bloco `try`.

# Observações

- Todo comando try define dois ou mais blocos referentes ao seu corpo e às suas cláusulas-catch. Estes blocos definem escopo de variáveis.
- Se um método não declara exceções numa superclasse, não é possível sobrescrever o método declarando exceções
- Também podem ser considerados como “erros de lógica”, já que indicam que o programa não tratou adequadamente a possibilidade de ocorrência de erros.

# Acesso ao sistema de arquivos

# Bibliografia

ANDRADE, A. G. P. D. **Java IO, Java NIO e NIO.2: Quando Utilizar?**, 2017. Disponível em:  
<https://medium.com/@antonio.gabriel/java-io-java-nio-e-nio-2-quando-utilizar-8c900b1c57a1>.

# Categorias das operações de E/S em arquivos

- As operações de E/S em arquivos são divididas em duas categorias:

## Operações para acesso ao sistema de arquivos

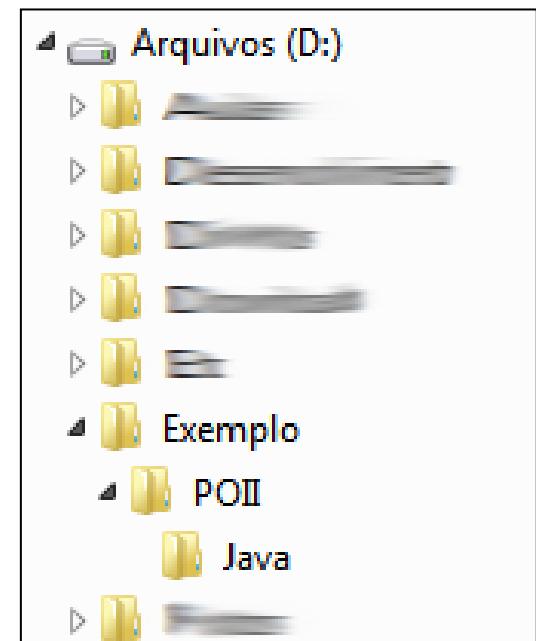
- Obter a relação de arquivos e subdiretórios de um diretório
- Ler propriedades de um arquivo ou diretório
- Criar diretórios, apagar diretórios, renomear arquivos, etc.

## Operações de leitura e edição de conteúdo de arquivos

- Ler o conteúdo de arquivos
- Alterar (gravar) dados em arquivos

# Diretórios

- **Caminho:** local (diretório) de um arquivo
  - Utiliza-se um caractere separador para expressar o caminho de um arquivo
    - barra (/) ou
    - barra invertida (\)
- **Caminho absoluto**
  - Contém todos os diretórios desde a raiz
    - Exemplo: D:\Exemplo\POII\Java\Slide1.pdf
- **Caminho relativo**
  - O caminho é relativo ao diretório atual
    - Exemplo: Java\Slide1.pdf



# Acesso ao sistema de arquivos

File
+ separatorChar : char = fs.getSeparator()
+ separator : String = ""+separatorChar
+ pathSeparatorChar : char = fs.getPathSeparator()
+ pathSeparator : String = ""+pathSeparatorChar
+ File(pathname : String)
+ getName() : String
+ getParent() : String
+ getParentFile() : File
+ getPath() : String
+ isAbsolute() : boolean
+ getAbsolutePath() : String
+ getAbsoluteFile() : File
+ getCanonicalPath() : String
+ getCanonicalFile() : File
+ canRead() : boolean
+ canWrite() : boolean
+ exists() : boolean
+ isDirectory() : boolean
+ isFile() : boolean
+ isHidden() : boolean
+ lastModified() : long
+ length() : long
+ createNewFile() : boolean
+ delete() : boolean
+ list() : String[]
+ listFiles() : File[]
+ mkdir() : boolean
+ mkdirs() : boolean
+ renameTo(dest : File) : boolean
+ setLastModified(time : long) : boolean
+ setReadOnly() : boolean
+ setWritable(writable : boolean) : boolean
+ setReadable(readable : boolean) : boolean
+ setExecutable(executable : boolean) : boolean
+ canExecute() : boolean
+ listRoots() : File[]
+ getTotalSpace() : long
+ getFreeSpace() : long
+ getUsableSpace() : long
+ createTempFile(prefix : String, suffix : String) : File

A classe **File** representa um arquivo ou um diretório.

Membro	Descrição
<b>File()</b>	Cria um objeto que representa um arquivo ou um diretório, que pode ou não existir
<b>getName()</b>	Retorna o nome do arquivo ou diretório
<b>exists()</b>	Retorna true se o arquivo/diretório existe
<b>isDirectory()</b>	Retorna true se o objeto representar um diretório
<b>isFile()</b>	Retorna true se o objeto representar um arquivo
<b>length()</b>	Retorna o tamanho, em bytes, do arquivo (válido apenas para arquivos)
<b>listFiles()</b>	Retorna um array com arquivos e diretórios contidos no diretório representado pelo objeto
<b>createTempFile()</b>	Cria um arquivo no diretório temporário

# Exemplo

```
File diretorio = new File("C:\\Windows");

File[] conteudoDiretorio = diretorio.listFiles();
for (File item : conteudoDiretorio) {
    if (item.isDirectory()) {
        System.out.println("Diretório: " + item.getName());
    } else {
        System.out.println("Arquivo " + item.getName() +
                           " tem " + item.length() + " bytes");
    }
}
```

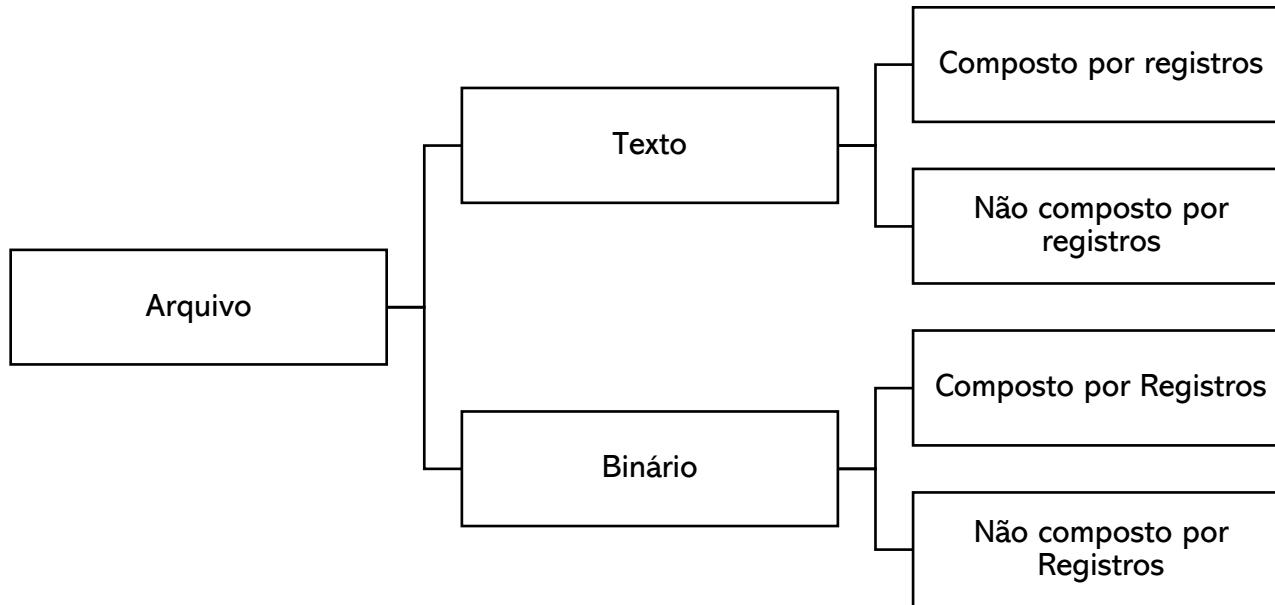
# Persistência de arquivos binários

Introdução

# Arquivo Texto X Arquivo Binário

- Arquivo texto
  - Os bits representam caracteres
  - Podem ser lidos por editores de texto
  - São “legíveis” para os humanos
- Arquivos binários
  - Os bits representam dados
  - Utilizam qualquer sequencia de bytes
  - Mais eficiente de processar

# Conteúdo de arquivos

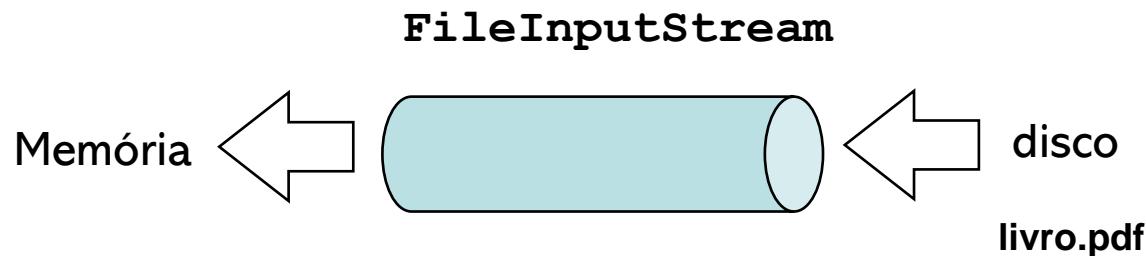


Registro = conjunto de dados organizados

Geralmente são organizados por tamanho fixo ou caractere de separação.

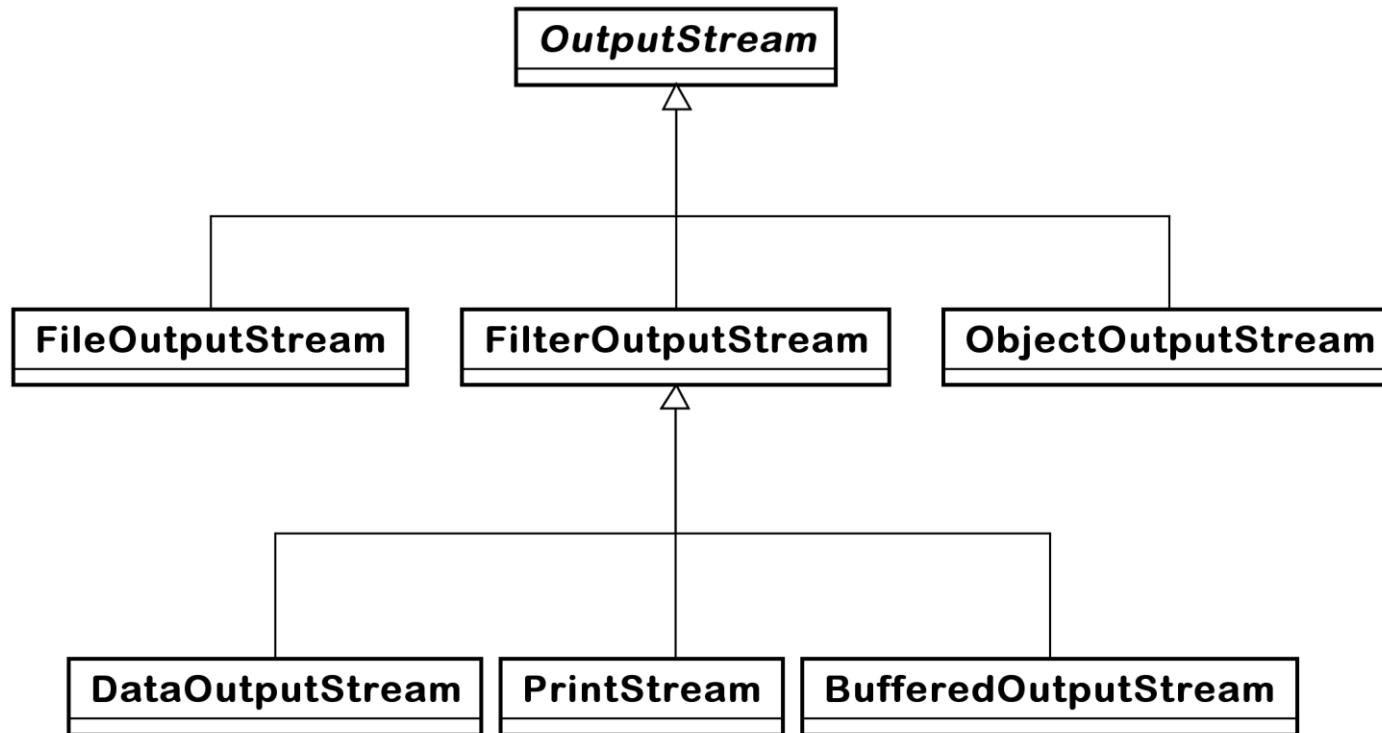
# Stream

- Um **Stream** é um objeto que tanto obtém dados de uma fonte (como teclado, arquivo, rede) como grava dados (tela, arquivo, etc).



# **Gravação de arquivos binários**

# Gravação de arquivos binários



# FileOutputStream

A classe **FileOutputStream** é utilizada para gravar arquivos.

<i>OutputStream</i>	Membro	Descrição
+ write(b : int) : void + write(b : byte[]) : void + flush() : void + close() : void	<b>FileOutputStream(File)</b>	Cria um objeto para gravar um arquivo. Se o arquivo já existir, será recriado (destruindo o conteúdo que havia).
+ FileOutputStream(name : String) + FileOutputStream(name : String, append : boolean) + FileOutputStream(file : File) + FileOutputStream(file : File, append : boolean) + write(b : int) : void + write(b : byte[]) : void + close() : void	<b>FileOutputStream(File, boolean)</b>	Cria um objeto para gravar um arquivo, possibilitando acrescentar novos dados no arquivo.
	<b>write(int)</b>	Grava um byte no arquivo
	<b>write(byte[])</b>	Grava os dados de um array no arquivo
	<b>flush()</b>	Provoca a atualização no arquivo
	<b>close()</b>	Fechá o arquivo

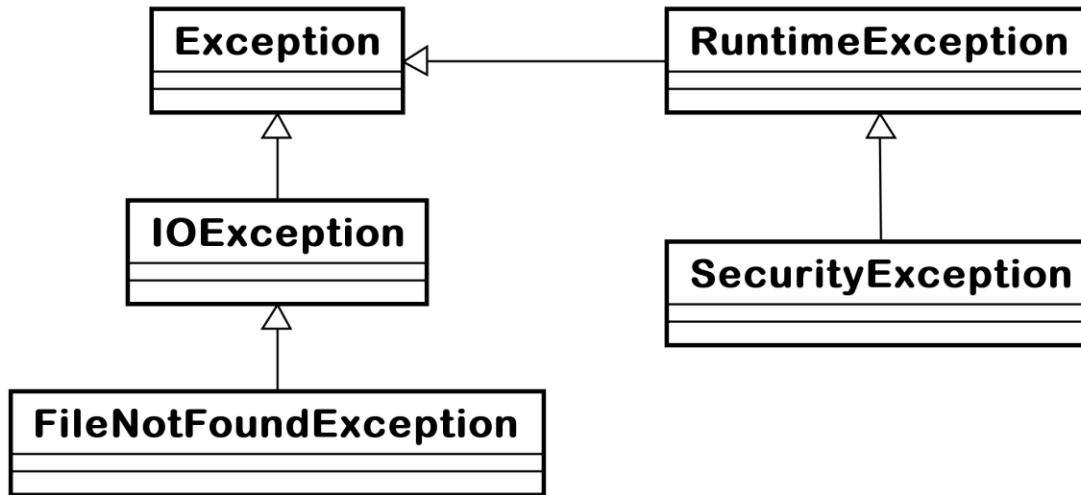
# Exemplo

```
File arquivo = new File("D:\\dados.dat");
FileOutputStream fos = new FileOutputStream(arquivo);
fos.write(79);
fos.write(105);
fos.close();
```

O construtor `FileOutputStream(File,boolean)` permite indicar que se quer acrescentar dados.

```
File arquivo = new File("D:\\dados.dat");
FileOutputStream fos = new FileOutputStream(arquivo, true);
fos.write(65);
fos.close();
```

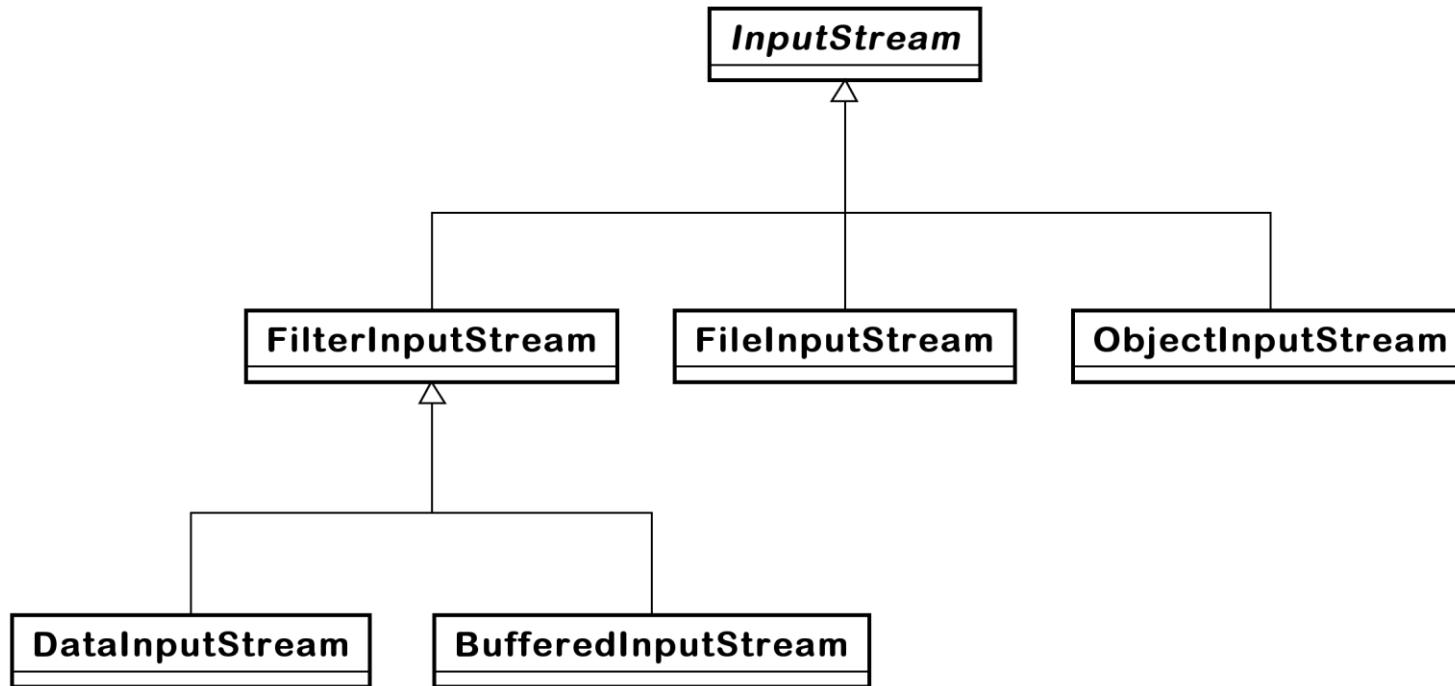
# Exceções que podem ser lançadas na gravação



Exceção	Descrição
<b>SecurityException</b>	Usuário sem permissão para gravar dados no arquivo
<b>FileNotFoundException</b>	Diretório não existe
<b>IOException</b>	Falha de gravação

# **Leitura de arquivos binários**

# Leitura de arquivos binários

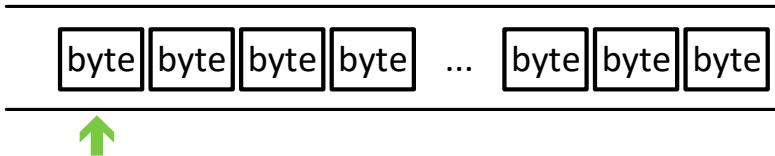


# Ponteiro do Arquivo

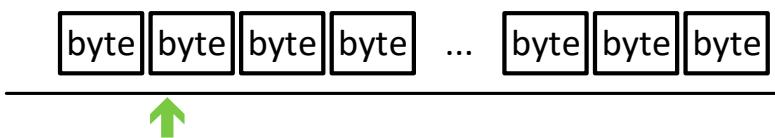
- O ponteiro é um número (long) que representa o número do byte do arquivo (começando de 0) em que o ponteiro está posicionado.
- Qualquer operação de leitura ou gravação sempre é feita na posição atual do ponteiro do arquivo.
- O ponteiro avança automaticamente quando ocorre uma operação de leitura.

# Ponteiro do arquivo

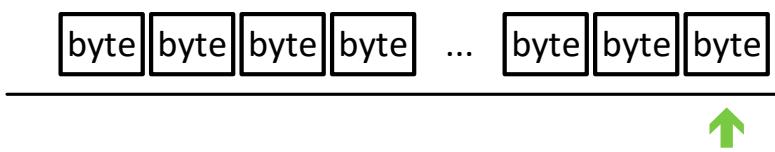
- 1 Após abrir o arquivo :



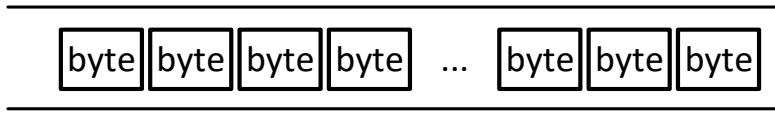
- 2 Após executar `fileInputStream.read()` :



- 3 Depois de vários `fileInputStream.read()` ...

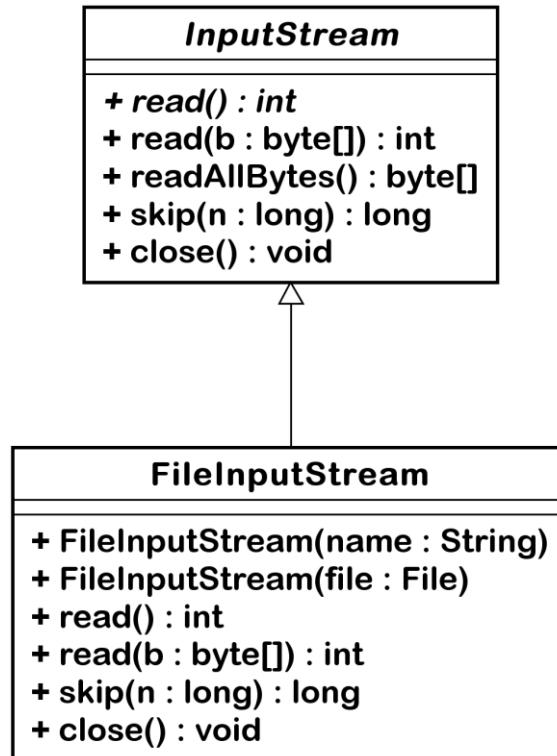


- 4 Após executar `fileInputStream.read()` :



# FileInputStream

A classe **FileOutputStream** é utilizada para ler arquivos.



Membro	Descrição
<code>FileInputStream(File)</code>	Cria um objeto para ler dados de um arquivo.
<code>read()</code>	Lê um byte do arquivo. Retorna -1 quando atingir o final do arquivo.
<code>read(byte[])</code>	Lê os dados do arquivo, suficiente para alimentar o vetor. Retorna quantidade de bytes lidos.
<code>readAllBytes()</code>	Lê e retorna um vetor com todos os dados do arquivo.
<code>skip()</code>	Salta para uma posição do arquivo, em relação à posição atual
<code>close()</code>	Fecha o arquivo

# Exemplo

```
File f = new File("D:\\dados.dat");
FileInputStream fis = new FileInputStream(f);
int dado;

while (true) {
    dado = fis.read();
    if (dado != -1) {
        System.out.println(dado);
    } else {
        break;
    }
}
fis.close();

System.out.println("Terminou");
```

```
File f = new File("D:\\dados.dat");
FileInputStream fis = new FileInputStream(f);
int dado;

while ((dado = fis.read()) != -1) {
    System.out.println(dado);
}

fis.close();
```

# Persistência de dados primitivos em arquivos binários

# Motivação

```
System.out.print("Informe o preço: ");

Scanner teclado = new Scanner(System.in);
double preco = Double.parseDouble(teclado.nextLine());

System.out.println("Valor digitado: " + preco);

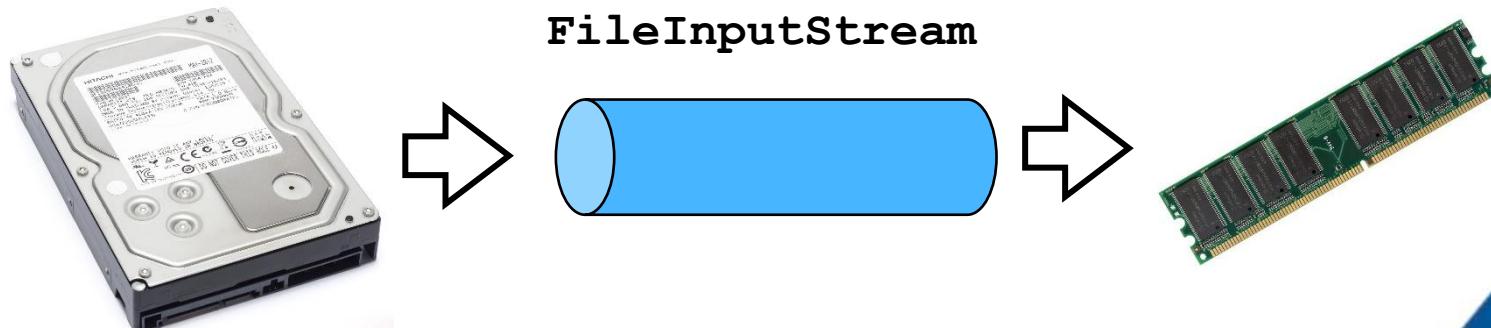
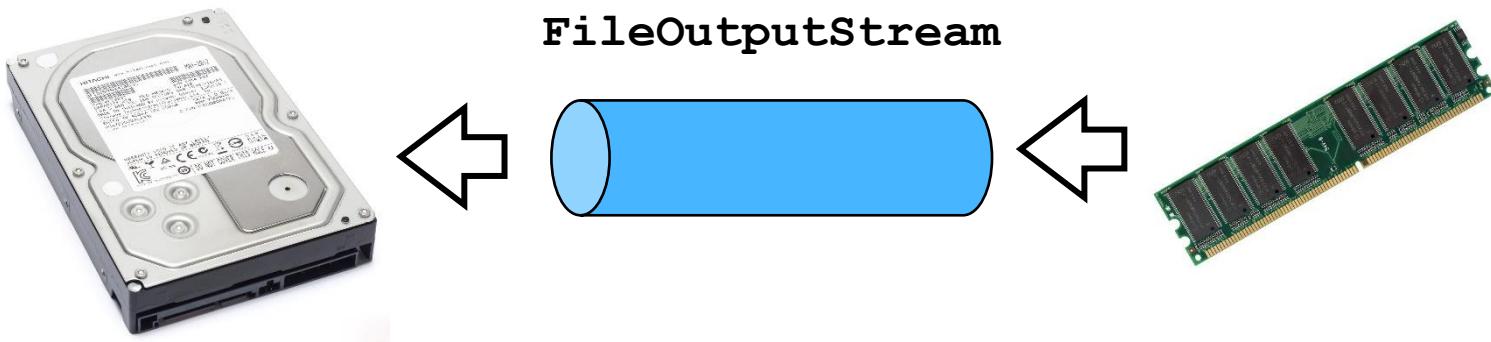
File f = new File("c:\\dados.bin");
FileOutputStream fos = new FileOutputStream(f);
fos.write(preco);
```

Esperado int  
informado double

Como gravar o valor da variável **preco** no arquivo?

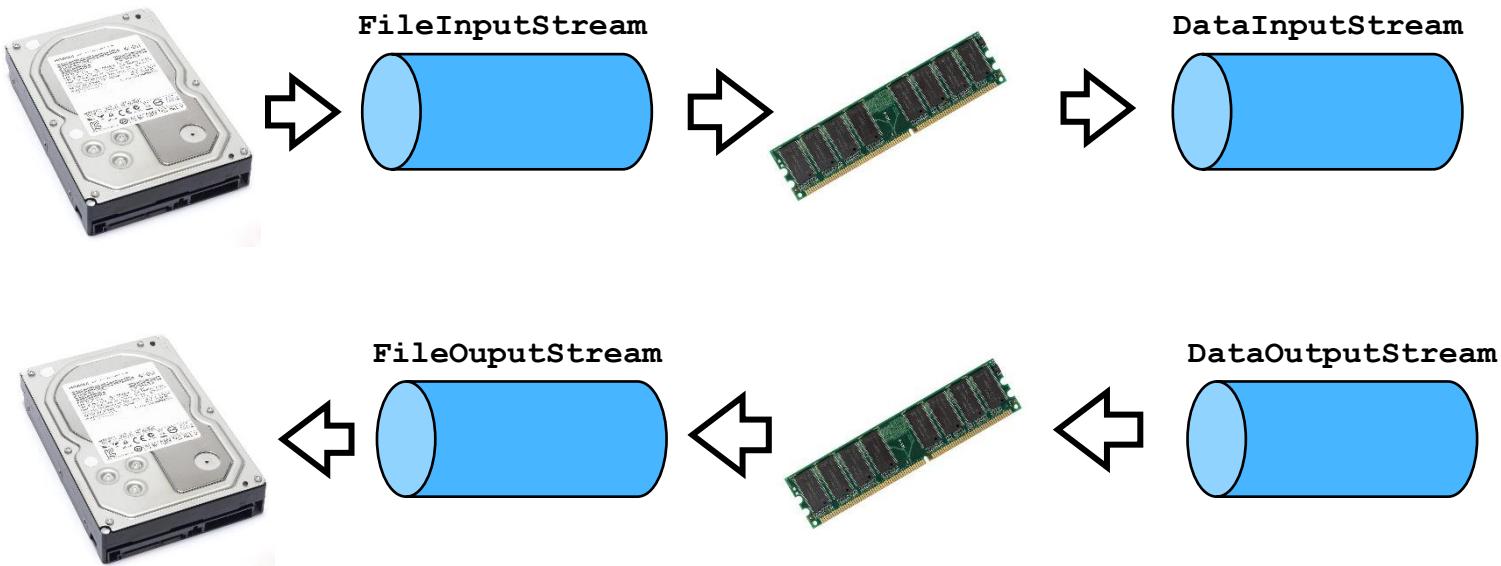
# Stream

- Um **Stream** é um objeto que tanto obtém dados de uma fonte (como teclado, arquivo, rede) como grava dados (tela, arquivo, etc).

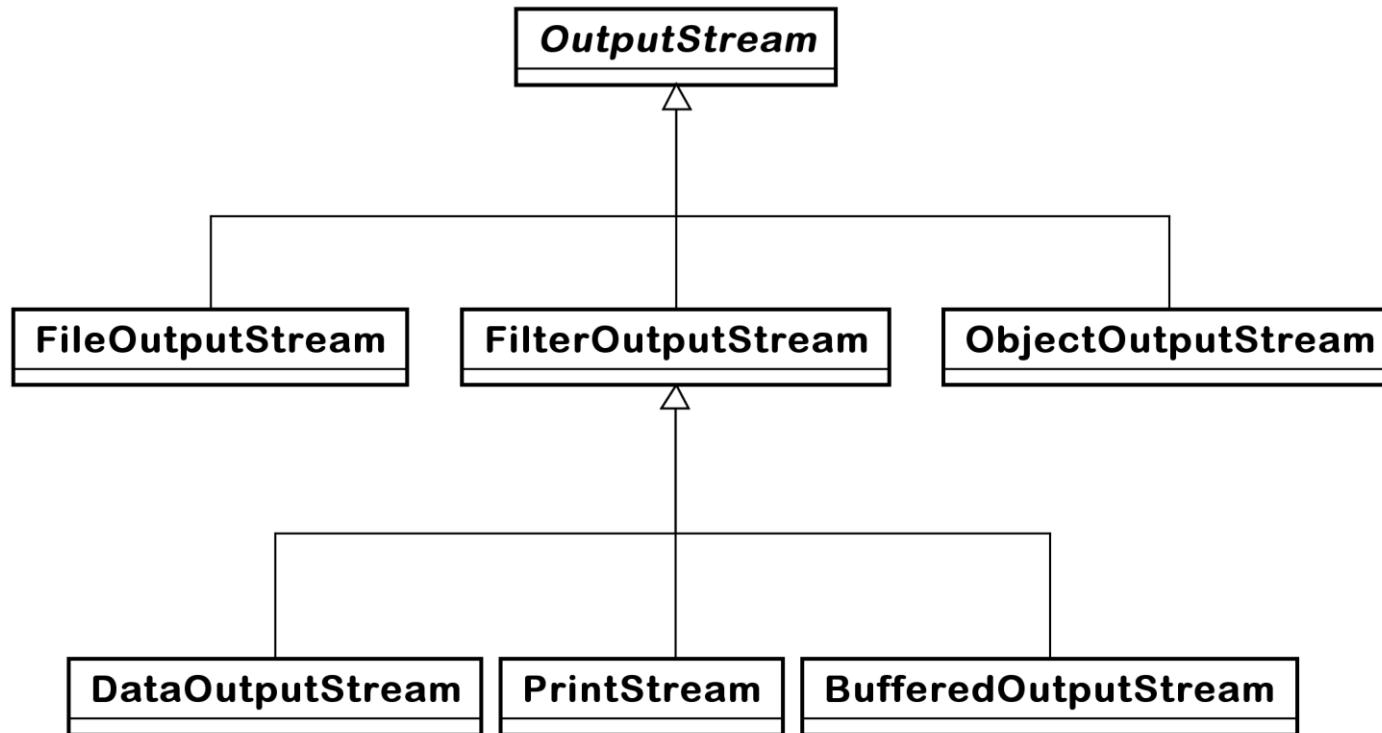


# Stream

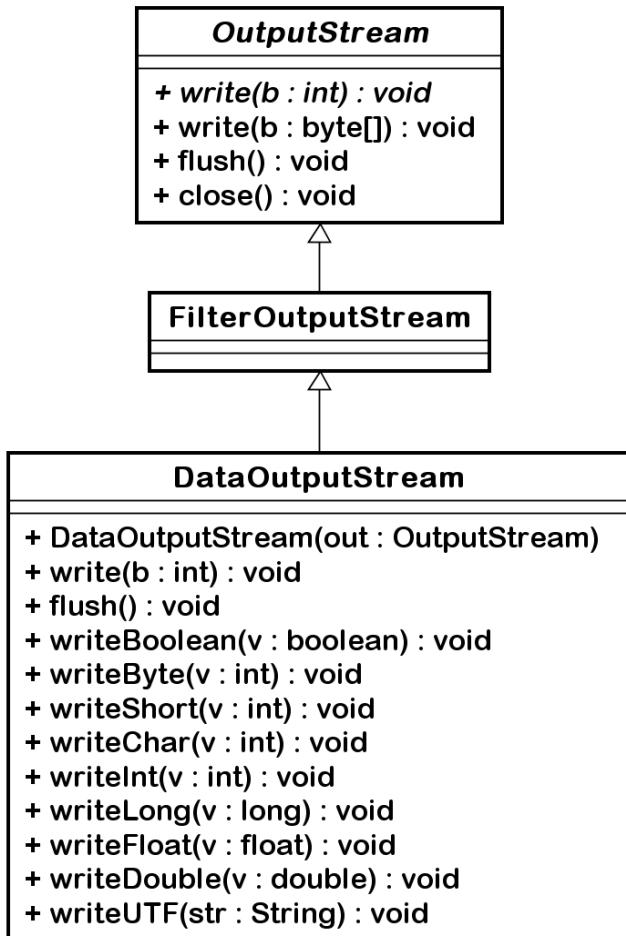
- É possível encadear outro objeto que estende *OutputStream* para criar outra forma alternativa de ler os dados



# Hierarquia de classes para gravação de dados



# DataOutputStream



- `DataOutputStream` oferece uma série de métodos para gravação de dados primitivos e string

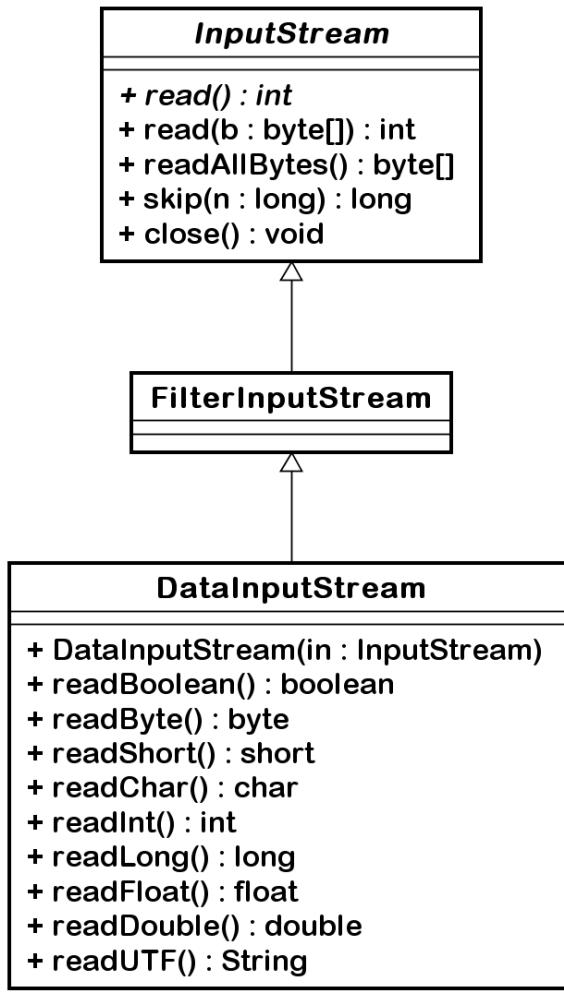
# Exemplo

```
File f = new File("c:\\dados.bin");
FileOutputStream fos = new FileOutputStream(f);
DataOutputStream dos = new DataOutputStream(fos);
dos.writeDouble(preco);
dos.writeBoolean(true);
dos.writeUTF("FURB");
dos.close();
```

# Observações

- Somente o objeto da classe **DataOutputStream** precisa ser fechado
  - Ao encadear streams, somente o objeto mais externo ao encadeamento precisa ser fechado

# Leitura de arquivos



- `DataInputStream` oferece uma série de métodos para leitura de dados primitivos e string
- Após a leitura, o ponteiro é posicionado após a quantidade de bytes lidos.

# Observações

- **DataInputStream** e **DataOutputStream** respectivamente, lê e grava tipos primitivos Java e Strings de forma independente de máquina
- Os dados devem ser lidos na mesma ordem em que foram gravados.
- A exceção **EOFException** é lançada quando há a tentativa de ler dados além do fim do arquivo

# Exemplo

```
File f = new File("D:\\dados.bin");
FileInputStream fis = new FileInputStream(f);
DataInputStream dis = new DataInputStream(fis);
double preco = dis.readDouble();
boolean bool = dis.readBoolean();
String instituicao = dis.readUTF();
dis.close();
```

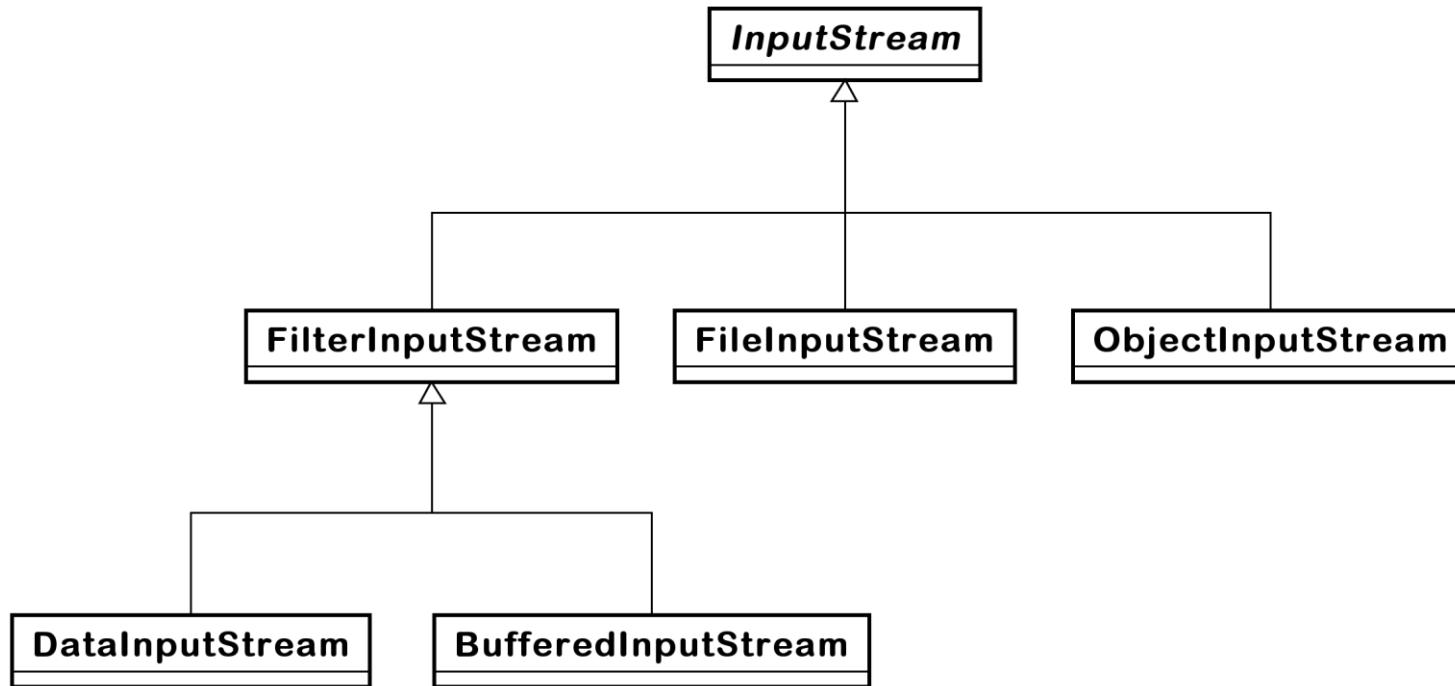
# Exemplo

```
File f = new File("c:\\dados.bin");
FileOutputStream fos = new FileOutputStream(f);
DataOutputStream dos = new DataOutputStream(fos);
dos.writeDouble(preco);
dos.writeBoolean(true);
dos.writeUTF("FURB");
dos.close();
```

00000008	0	1	2	3	4	
00000000	64	94	192	0	0	@^À..
00000008	0	0	0	1	0	.....
0000000a	4	70	85	82	66	.FURB
0000000f	...	...	...	...	...	.....

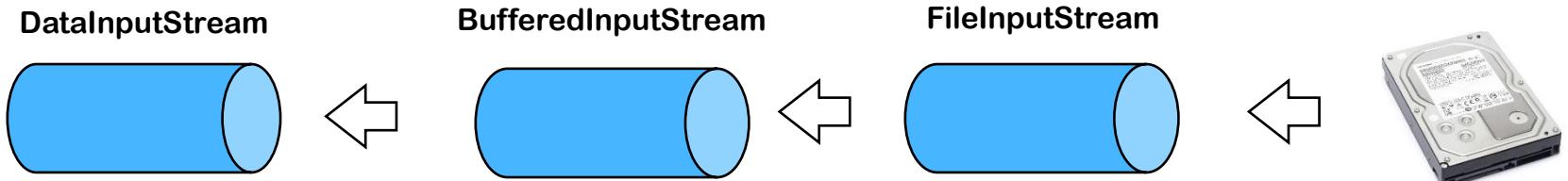
# *Buffer* de dados na leitura/gravação

# Leitura de arquivos binários

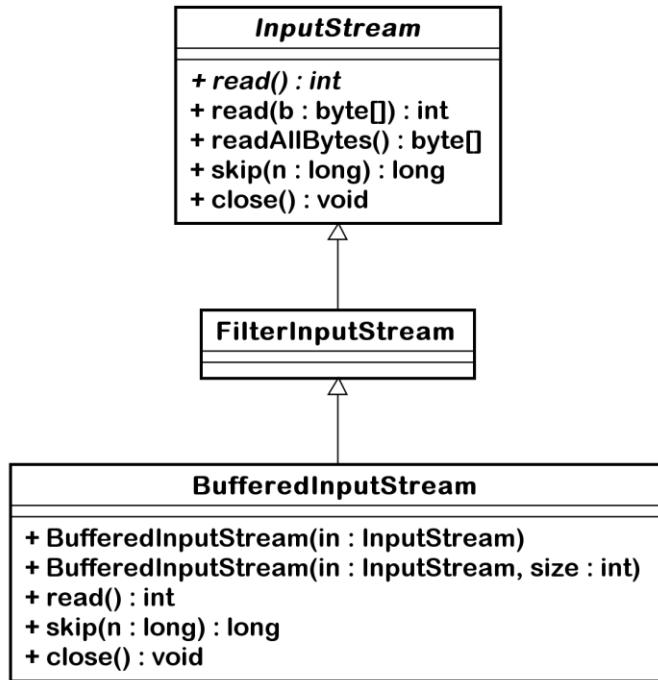


# BufferedInputStream

- A classe **BufferedInputStream** é utilizada para reduzir a quantidade de acessos ao dispositivo de armazenamento, tornando a leitura mais otimizada



# BufferedInputStream



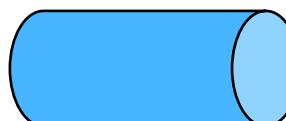
Membro	Descrição
BufferedInputStream(InputStream, size)	Cria um objeto para ler dados de um <i>inputStream</i> . Define o tamanho do buffer.
BufferedInputStream(InputStream)	Cria um objeto para ler dados de um <i>inputStream</i> . Utiliza um buffer de 8 KB.
read()	Lê um byte do <i>inputstream</i> . Retorna -1 quando atingir o final.
skip()	Salta para uma posição a frente do <i>inputstream</i> , em relação à posição atual. Deve ser um valor positivo
close()	Fechá o <i>inputstream</i>

# Exemplo:

DataInputStream



BufferedInputStream



FileInputStream



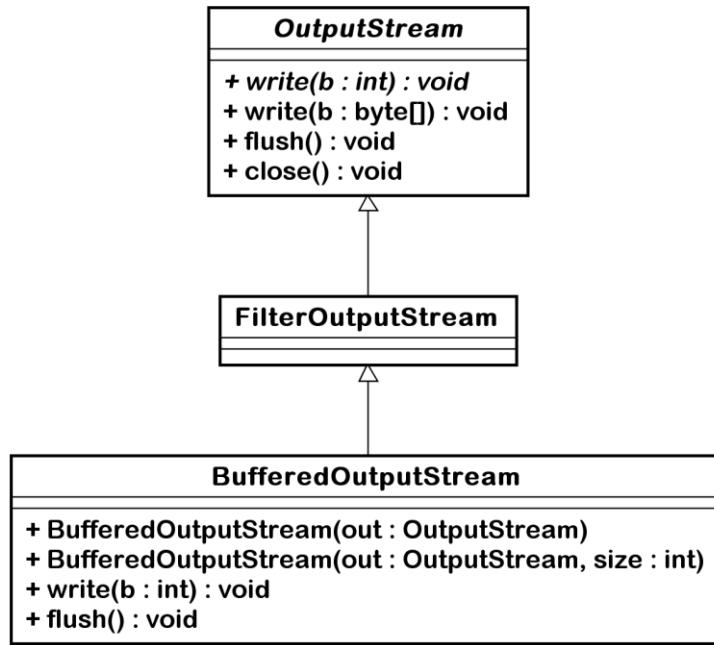
```
File arquivo = new File("D:\\dados.dat");

FileInputStream fis = new FileInputStream(arquivo);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);

dis.close();
```

Neste exemplo, deve-se utilizar os métodos do **DataInputStream** para ler os dados do stream

# BufferedOutputStream



Membro	Descrição
<code>BufferedOutputStream(OutputStream, size)</code>	Cria um objeto para gravar dados em um <code>outputStream</code> . Define o tamanho do buffer.
<code>BufferedInputStream(InputStream)</code>	Cria um objeto para gravar dados em um <code>inputStream</code> . Utiliza um buffer de 8 KB.
<code>write(int)</code>	Grava um byte no <code>outputstream</code> .
<code>flush()</code>	Grava os dados imediatamente

# **Abrindo um arquivo para leitura e edição**

# Editar dados de um arquivo

- A classe DataOutputStream somente funciona com duas modalidades:
  - Recriar um arquivo
  - Acrescentar dados ao final do arquivo.
- Não é possível alterar dados já gravados com DataOutputStream
  - Não possui skip()

# Classe RandomAccessFile

```
RandomAccessFile

+ RandomAccessFile(name : String, mode : String)
+ RandomAccessFile(file : File, mode : String)
+ read() : int
+ read(b : byte[]) : int
+ readFully(b : byte[]) : void
+ skip(n : int) : int
+ write(b : int) : void
+ write(b : byte[]) : void
+ length() : long
+ setLength(newLength : long) : void
+ close() : void
+ readBoolean() : boolean
+ readByte() : byte
+ readShort() : short
+ readChar() : char
+ readInt() : int
+ readLong() : long
+ readFloat() : float
+ readDouble() : double
+ readUTF() : String
+ writeBoolean(v : boolean) : void
+ writeByte(v : int) : void
+ writeShort(v : int) : void
+ writeChar(v : int) : void
+ writeInt(v : int) : void
+ writeLong(v : long) : void
+ writeFloat(v : float) : void
+ writeDouble(v : double) : void
+ writeUTF(str : String) : void
```

```
<<interface>>
DataInput

+ readFully(b : byte[]) : void
+ skipBytes(n : int) : int
+ readBoolean() : boolean
+ readByte() : byte
+ readShort() : short
+ readChar() : char
+ readInt() : int
+ readLong() : long
+ readFloat() : float
+ readDouble() : double
+ readUTF() : String
```

```
<<interface>>
DataOutput

+ write(b : int) : void
+ write(b : byte[]) : void
+ writeBoolean(v : boolean) : void
+ writeByte(v : int) : void
+ writeShort(v : int) : void
+ writeChar(v : int) : void
+ writeInt(v : int) : void
+ writeLong(v : long) : void
+ writeFloat(v : float) : void
+ writeDouble(v : double) : void
+ writeBytes(s : String) : void
+ writeUTF(s : String) : void
```

# RandomAccessFile

- O parâmetro **mode** pode ser:
  - **r** – para acesso somente leitura.
  - **rw** – para leitura e gravação.
  - **rws** – para leitura e gravação. Cada gravação é aplicada imediatamente no arquivo. Causa a alteração da data de modificação do arquivo
  - **rwd** – Similar ao **rws**, porém somente causa a alteração da data de modificação do arquivo quando o arquivo é fechado.
- Os modos **rws** e **rwd** são mais lentos.

# Principais métodos de RandomAccessFile

Membro	Descrição
<code>skip(long)</code>	Salta para uma posição a frente do <i>inputstream</i> , em relação à posição atual. Deve ser um valor positivo.
<code>seek(long)</code>	Salta para uma posição no <i>inputStream</i> . A posição é em relação ao primeiro byte do arquivo. Exemplo: independente da posição que o ponteiro estiver, <code>seek(50)</code> posiciona no 51º byte do arquivo.
<code>setLength(long)</code>	Permite truncar o arquivo. Sempre informar um valor menor do que o atual.

# Fechamento de *streams*

# Fechamento de arquivo

Depois de manipulado o arquivo, ele deve ser fechado:

```
File arquivo = new File("D:\\Temp\\dados.bin");
FileInputStream fis = new FileInputStream(arquivo);

int dado;
while ((dado = fis.read()) != -1) {
    System.out.println( dado );
}

fis.close();
```

Se ocorrer erro na leitura de arquivo,  
esta linha não é executada

# Fechamento de arquivo

```
File arquivo = new File("D:\\Temp\\dados.bin");
FileInputStream fis = null;

try {
    fis = new FileInputStream(arquivo);

    int dado;
    while ((dado = fis.read()) != -1) {
        System.out.println( dado );
    }
} finally {
    if (fis != null)
        fis.close();
}
```

```
try (FileInputStream fis = new FileInputStream(arquivo)) {
    int dado;
    while ((dado = fis.read()) != -1) {
        System.out.println( dado );
    }
}
```

Sintaxe resumida  
utilizando o *try with resources*

Para usar try(...) os objetos instanciados devem ser de classes que implementam a interface java.lang.AutoCloseable

# Serialização

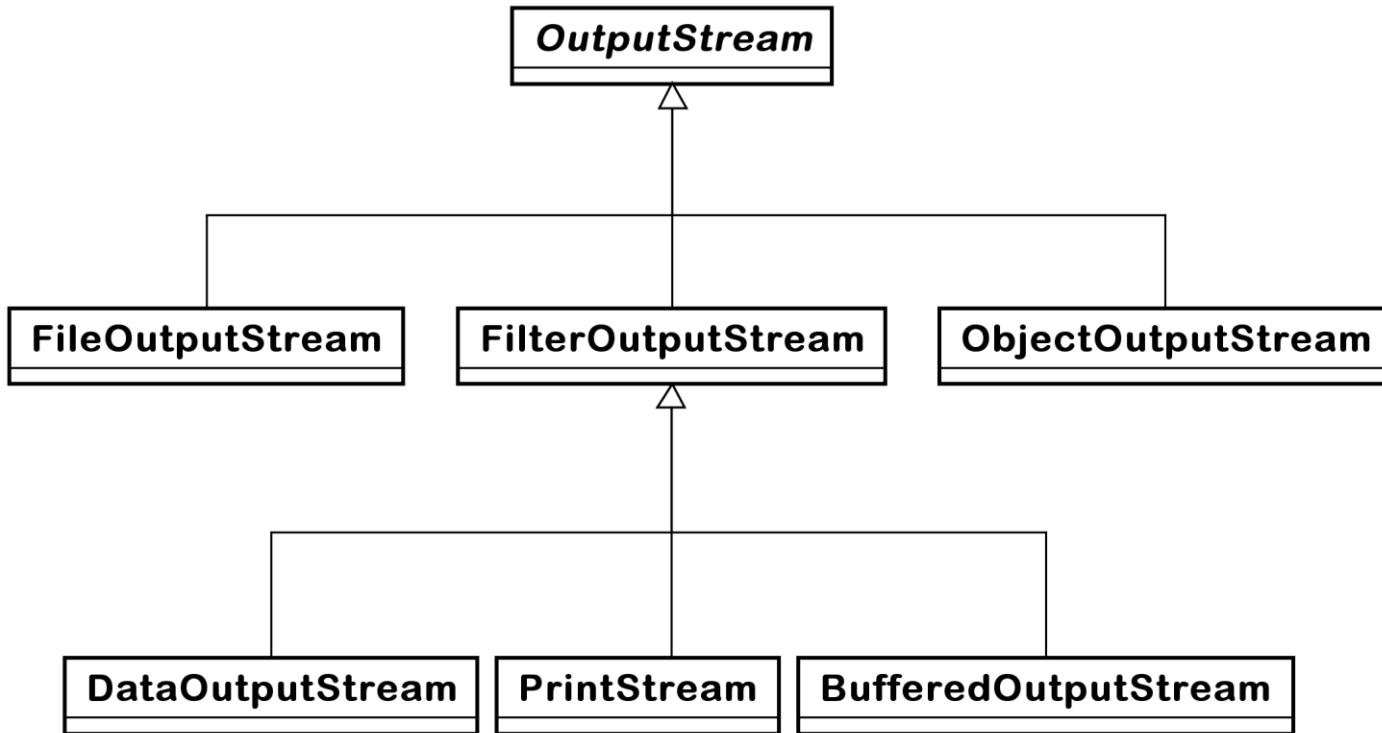
# Bibliografia

Oracle. **Object Serialization Stream Protocol.**

[https://docs.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html.](https://docs.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html)

Acesso em 2019.

# Classes de persistência



# ObjectOutputStream e ObjectInputStream

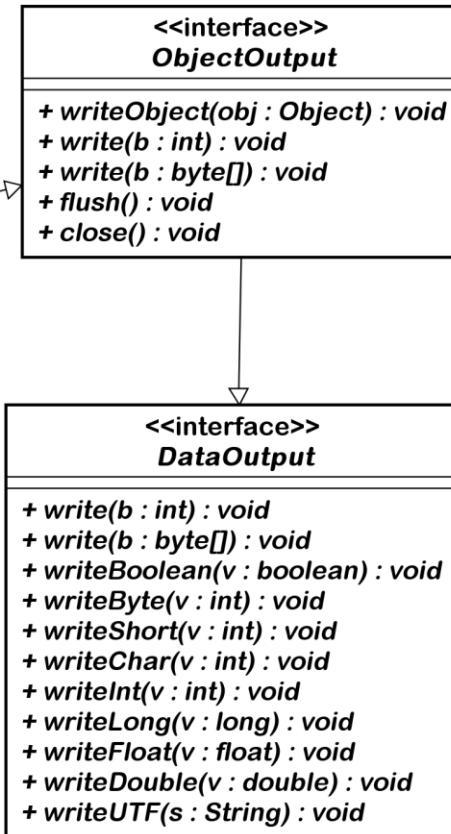
A **serialização** de objetos consistem em converter os dados de estado de um objeto numa representação binária ou textual, a fim de armazená-lo num meio de armazenamento secundário

A operação inversa (a partir de uma cadeia de bits reconstruir um objeto), é chamada de **desserialização**

# ObjectOutputStream

ObjectOutputStream
+ ObjectOutputStream(out : OutputStream)
+ writeObject(obj : Object) : void
+ write(val : int) : void
+ write(buf : byte[]) : void
+ write(buf : byte[], off : int, len : int) : void
+ flush() : void
+ close() : void
+ writeBoolean(val : boolean) : void
+ writeByte(val : int) : void
+ writeShort(val : int) : void
+ writeChar(val : int) : void
+ writeInt(val : int) : void
+ writeLong(val : long) : void
+ writeFloat(val : float) : void
+ writeDouble(val : double) : void
+ writeBytes(str : String) : void
+ writeChars(str : String) : void
+ writeUTF(str : String) : void

**writeObject()** é o principal método de **ObjectOutputStream**.



# Exemplo - Serialização

```
File f = new File("Objetos.bin");
try (FileOutputStream fos = new FileOutputStream(f);
     ObjectOutputStream o = new ObjectOutputStream(fos)) {
    o.writeObject("Boa noite");
    o.writeObject(new ContaBancaria("1", "Lucas", 300));
    o.writeObject(new Veiculo("MIE-3932", "Anderson"));
}
```

# Observações

- Podem ser gravados vários objetos no *stream*.
- Para um objeto ser serializado, este deve implementar a interface **Serializable**
  - **Serializable** é uma *interface de marcação*
- Valores de variáveis estáticas não são serializados.
- Para não serializar determinada variável de instância, introduzir o modificador **transient**, como abaixo:

```
private transient String atributo;
```

# Exemplo - Dessorialização

```
String objeto1;
ContaBancaria objeto2;
Veiculo objeto3;

try (FileInputStream fis = new FileInputStream(f);
     ObjectInputStream ois = new ObjectInputStream(fis)) {
    objeto1 = (String) ois.readObject();
    objeto2 = (ContaBancaria) ois.readObject();
    objeto3 = (Veiculo) ois.readObject();
}
```

Para ler os objetos do *stream*, deve-se executar as operações de leitura na mesma ordem em que foram executadas as operações de gravação

# Exceções lançadas

## **ClassNotFoundException**

Ocorre quando se tenta desserializar objeto de uma classe desconhecida

## **NotSerializableException.**

Ocorre quando se quer serializar um objeto de uma classe que não implementa a interface **Serializable**

# Serialização - Versionamento

- É possível “versionar” uma classe ao persistir um objeto.
- O versionamento garante que objetos somente possam ser reconstituídos quando recuperados a partir da mesma versão de classe que persistiu o objeto.
- Para estabelecer um número de versão, acrescentar:  
`private static final long serialVersionUID = 1L;`

# Serialização - Versionamento

- *serialVersionUID* deveria ser um identificador único para uma versão.
- Quando não há declaração de *serialVersionUID*, Java gera um número de versão
- É recomendável que todas as classes serializáveis declarem explicitamente um valor para *serialVersionUID*.
- A exceção `java.io.InvalidClassException` é lançada quando ocorre tentativa de desserializar um objeto que foi serializado com versão diferente.

# Serialização

- Para obter a versão de uma classe, utilizar o utilitário “serialver” na linha de comando, fornecendo o nome da classe (incluindo seu pacote).

# StreamCorruptedException

- Uma exceção da classe `StreamCorruptedException` pode ser lançada ao ler um arquivo com objetos persistidos
- Ocorre ao ler um arquivo que foi gravado com `FileOutputStream` com objetivo de acrescentar dados.
- Falha ocorre porque sempre que `ObjectOutputStream` é criado, grava um novo cabeçalho no arquivo, constituído de 4 bytes

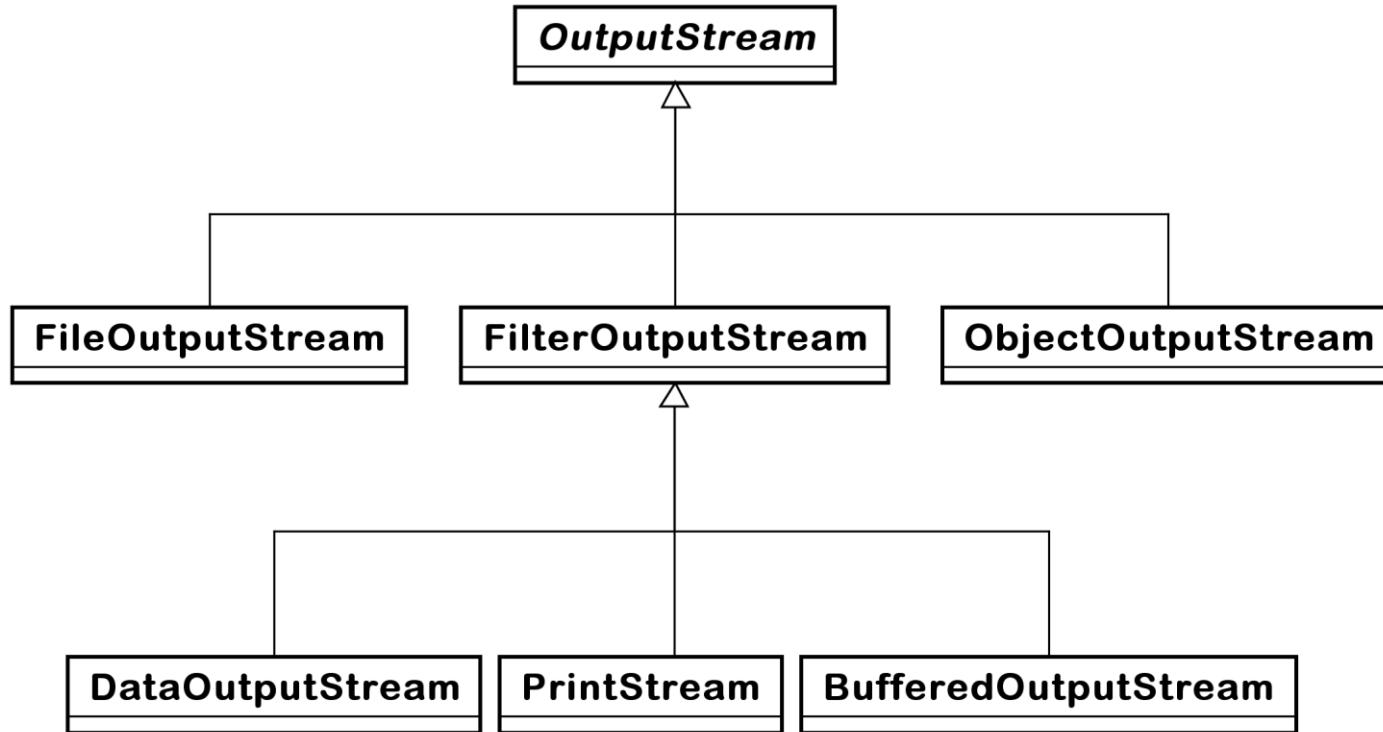
00	01	02	03	04	05	06	07	08	09
ac	ed	00	05	74	00	09	42	6f	61
74	00	0e	53	65	67	75	6e	64	61

# Adicionar novos objetos num stream

```
1 import java.io.IOException;
2 import java.io.ObjectOutputStream;
3 import java.io.OutputStream;
4
5
6 public class MeuObjectOutputStream extends ObjectOutputStream {
7
8     public MeuObjectOutputStream(OutputStream out) throws IOException {
9         super(out);
10    }
11
12
13     @Override
14     protected void writeStreamHeader() throws IOException {
15
16    }
17
18 }
```

# **Leitura e gravação de arquivos textos**

# Persistência de texto



# Persistência de texto

## PrintWriter

```
+ PrintWriter(out : OutputStream)
+ PrintWriter(out : OutputStream, autoFlush : boolean)
+ PrintWriter(fileName : String)
+ PrintWriter(fileName : String, csn : String)
+ PrintWriter(file : File)
+ PrintWriter(file : File, csn : String)
+ flush() : void
+ close() : void
+ write(s : String) : void
+ print(b : boolean) : void
+ print(c : char) : void
+ print(i : int) : void
+ print(l : long) : void
+ print(f : float) : void
+ print(d : double) : void
+ print(s : char[]) : void
+ print(s : String) : void
+ print(obj : Object) : void
+ println() : void
+ println(x : boolean) : void
+ println(x : char) : void
+ println(x : int) : void
+ println(x : long) : void
+ println(x : float) : void
+ println(x : double) : void
+ println(x : char[])
+ println(x : String) : void
+ println(x : Object) : void
```

A classe PrintWriter é utilizada para gravar texto

# Gravação de arquivo texto

- Para gravação de arquivo de texto livre podemos utilizar a classe PrintWriter:

```
File arquivo = new File("etc/texto.txt");
PrintWriter arquivoTexto = new PrintWriter(arquivo);
```

- Para adicionar texto, utilizar os métodos: print() e println() :

```
arquivoTexto.println("Primeira linha");
arquivoTexto.print(true);
arquivoTexto.printf("Preço: %20.2f", 1234.5);
```

# Caracteres de escape - print

Sequência	Descrição
\t	Tab (caractere 9)
\b	Backspace (8)
\n	Nova linha(10)
\r	Retorno de carro(13)
\f	Caractere 12
\'	Aspas simples
\”	Aspas duplas
\\"	Barra inversa

# PrintWriter

- No arquivo texto, o caractere de separação de linha é diferente entre as plataformas:

Sistema Operacional	Caractere(s)
Windows	\r\n
UNIX/Linux	\n
Mac OS X	\n

- O método `println()` gera o separador específico da plataforma.

# Métodos

Método	Descrição
PrintWriter(File) PrintWriter(String)	Cria um arquivo de texto
PrintWriter(OutputStream)	Edita o arquivo representado pelo stream
close()	Fechá o arquivo

Para abrir o arquivo para inclusão de novo texto:

```
File arquivo = new File("etc/texto.txt");
FileOutputStream fos = new FileOutputStream(arquivo, true);
PrintWriter arquivoTexto = new PrintWriter(fos);
arquivoTexto.println("nova linha");
```

# Scanner

A classe Scanner pode ser utilizada para efetuar operações de leitura em arquivo texto

Método	Descrição
Scanner(File) Scanner(String)	Abre o arquivo
close()	Fecha o arquivo
String nextLine()	Lê uma string
byte nextByte()	Lê um byte
int nextInt()	Lê um int
long nextLong()	Lê um long
float nextFloat()	Lê um float
double nextDouble()	Lê um double
String next()	Retorna uma palavra
boolean hasNext()	Retorna true se há dado para ser lida no arquivo

# Exemplo

O exemplo abaixo exibe o conteúdo de um arquivo texto:

```
Scanner arqTexto = new Scanner(new File("etc\\texto.txt"));
while (arqTexto.hasNext()) {
    System.out.println(arqTexto.nextLine());
}
```

# Manipulação com outros mapas de caracteres

PrintWriter
+ PrintWriter(out : OutputStream)
+ PrintWriter(out : OutputStream, autoFlush : boolean)
+ PrintWriter(fileName : String)
+ PrintWriter(fileName : String, csn : String)
+ PrintWriter(file : File)
+ PrintWriter(file : File, csn : String)

- Mapa de caracteres (*character set*): corresponde ao conjunto de caracteres que podem ser utilizados. Alguns dos mapas mais conhecidos são:
  - US-ASCII (ISO/IEC 646): contém caracteres do alfabeto inglês
  - Latin-x (ISO/IEC 8859-x): contém caracteres ocidentais (inclusive latinos)
  - Unicode (ISO/10646): contém os caracteres de todas as linguagens (atualmente capaz de representar aproximadamente 2 milhões de símbolos distintos)
- Num mapa de caracteres, cada caractere possui um código (*code point*)

# Mapas de caracteres

- O mapa de caracteres Unicode pode ser codificado (*encoding*) binariamente de várias formas:
  - UTF-8: eficiente para representar texto com caracteres ocidentais, inclusive latinos, pois utiliza 1 byte para estes caracteres. Demais caracteres podem usar de 2 à 4 bytes.
  - UTF-16: utiliza 2 ou 4 bytes para representar qualquer símbolo do mapa.
  - UTF-32: utiliza 4 bytes para representar qualquer símbolo do mapa.
- Exemplo: o caractere €, tem code point igual à 8464.
  - Em UTF-8, sua representação é 0xE2 0x82 0xAC
  - Em UTF-16, sua representação é 0xAC 0x20
- A distinção dos termos *mapa de caracteres* e *codificação* são aplicáveis apenas para Unicode já que em outros mapas, o *code point* e a representação binária são iguais.

# Leitura e gravação com codificações específicas

- Para utilizar a classe PrintWriter para gravar um arquivo com uma codificação específica:

```
String linha = "áéíóúç!";  
out = new PrintWriter("etc/arq4.txt", "UTF-8");  
out.write(linha);
```

# Leitura e gravação com codificações específicas

- A classe Scanner permite ler arquivos com uma codificação específica:

```
File arquivo = new File("etc/arq4.txt");
Scanner sc = new Scanner(arquivo, "UTF-8");
while (sc.hasNext()) {
    System.out.println(sc.nextLine());
}
```

# Igualdade de objetos

# Igualdade de Objetos

- **Igualdade de referência**

- quando as variáveis fazem referência ao mesmo objeto.

```
Ponto p1 = new Ponto(2, 10);
Ponto p2 = p1;
```

- a igualdade de objetos é verificada com o operador ==

- **Igualdade lógica**

- quando um conjunto de atributos de dois objetos possuem os mesmos

```
Ponto p1 = new Ponto(3, 5);
Ponto p2 = new Ponto(3, 5);
```

- A igualdade de valores é verificada com o método `equals()`

```
if (p1.equals(p2)) {
    System.out.print("Mesmo ponto");
}
```

**Ponto**

- x : int  
- y : int

# Igualdade de objetos

- Na igualdade lógica, considera-se o conjunto de atributos que identificam unicamente o objeto.
- Exemplo:

Aluno
- matricula : int
- nome : String
- endereco : String
- dataNascimento : LocalDate

- O conjunto de atributos que identificam unicamente alunos é composto pelo atributo **matricula**.

# Exemplo

## Aluno

- matricula : int
- nome : String
- endereco : String
- dataNascimento : LocalDate

### a1 : Aluno

matricula = 205108  
nome = Fávia Moritz  
endereco = Rua São Paulo, 295  
dataNascimento = 05/01/2001

### a2 : Aluno

matricula = 205108  
nome = null  
endereco = null  
dataNascimento = null

```
Aluno a1 = new Aluno();
a1.setMatricula(205108);
a1.setNome("Flávia Moritz");
a1.setEndereco("Rua São Paulo, 295");
a1.setDataNascimento(LocalDate.of(2001, 01, 05));
```

```
Aluno a2 = new Aluno();
a2.setMatricula(205108);
```

# Utilização de igualdade lógica na busca

```
ArrayList<Aluno> alunos = new ArrayList<>();

Aluno a1 = new Aluno();
a1.setMatricula(205108);
a1.setNome("Flávia Moritz");
a1.setEndereco("Rua São Paulo, 295");
a1.setDataNascimento(LocalDate.of(2001, 01, 05));

alunos.add(a1);

alunos.add(new Aluno(...));
alunos.add(new Aluno(...));
alunos.add(new Aluno(...));

Aluno alunoProcurar = new Aluno();
alunoProcurar.setMatricula(205108);

// procurar...
int posicao = alunos.indexOf(alunoProcurar);
if (posicao > -1) {
    Aluno alunoEncontrado = alunos.get(posicao);
    System.out.println(alunoEncontrado.getNome());
}
```

# Igualdade de Objetos

- Método **equals()** está na classe **Object** e sua implementação é:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

- para que seja possível realizar a igualdade lógica, devemos sobrescrever o método **equals()**.

# Igualdade de Objetos

- A implementação do `equals()` deve seguir um padrão:

```
1 public boolean equals(Object obj) {  
2     if (this == obj)  
3         return true;  
4     if (obj == null)  
5         return false;  
6     if (getClass() != obj.getClass())  
7         return false;  
8     Ponto other = (Ponto) obj;  
9     if (getX() != other.getX())  
10        return false;  
11     if (getY() != other.getY())  
12        return false;  
13     return true;  
14 }
```

# Exemplo

```
public class Aluno {  
  
    private int matricula;  
    private String nome;  
    private String endereco;  
    private LocalDate dataNascimento;  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Aluno other = (Aluno) obj;  
        if (matricula != other.matricula)  
            return false;  
        return true;  
    }  
    ...  
}
```

# Biblioteca de estrutura de dados

# Bibliografia

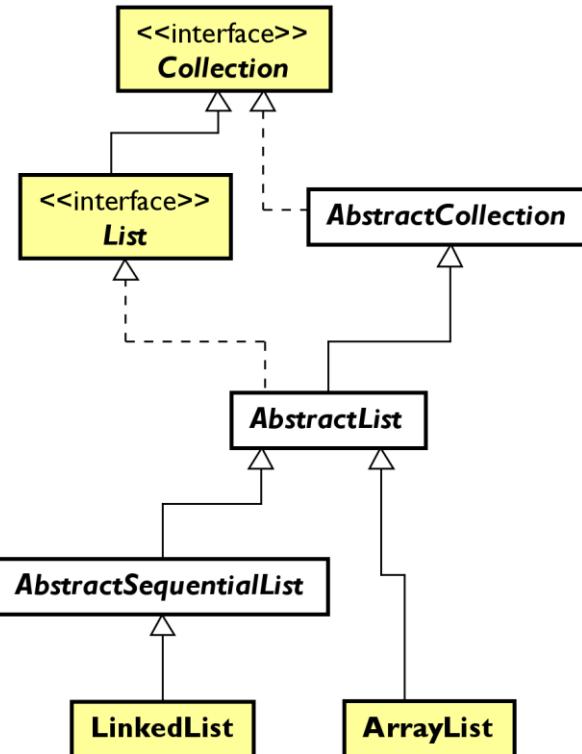
- Documentação Oficial da Java Collections Framework.  
<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html>
- HORSTMANN, Cay S. **Big Java.** Porto Alegre : Bookman, 2004. xi, 1125 p, il., 1 CD-ROM. Acompanha CD-ROM.
- DEITEL, Paul J; DEITEL, Harvey M. **Java: como programar.** 8. ed. São Paulo: Pearson, 2010. xxix, 1144 p, il.
- LIANG, Y. Daniel. **Introduction to Java Programming and Data Structures.** 11<sup>a</sup> ed. Person, 2019. 1232p.

# Introdução

- Uma “Estruturas de dados” é uma forma de armazenar e organizar dados na memória do computador.
- “Escolher a melhor estrutura de dados e algoritmos para uma tarefa particular é uma das chaves para o desenvolvimento de software de alta performance” (LIANG, 2019)
- Java possui uma biblioteca para manipular estruturas de dados, denominado de **Java Collection Framework** (JCF)
  - Embora seja chamado de “framework”, a JCF é uma biblioteca
- As classes de manipulação de estrutura de dados estão agrupadas em dois grandes grupos: **Coleções e Mapas**

# Coleções

# Principais coleções



ArrayList implementa uma lista estática  
LinkedList implementa uma lista dinâmica

## Diferenças

Operação	ArrayList	LinkedList
get(index)	Extremamente rápido	Lento
add(E)	Rápido na maior parte das vezes	Extremamente rápido
add(index, E)	Lento	Lento
remove(index, E)	Lento	Lento

# Interface Collection

E	
<<interface>> <b>Collection</b>	
+ size() : int	
+ isEmpty() : boolean	
+ contains(o : Object) : boolean	
+ iterator() : Iterator<E>	
+ toArray() : Object[]	
+ toArray(a : T[]) : T[]	
+ add(e : E) : boolean	
+ remove(o : Object) : boolean	
+ containsAll(c : Collection<?>) : boolean	
+ addAll(c : Collection<E>) : boolean	
+ removeAll(c : Collection<?>) : boolean	
+ retainAll(c : Collection<?>) : boolean	
+ clear() : void	
+ equals(o : Object) : boolean	
+ hashCode() : int	

Método	Descrição
size()	Retorna a quantidade de elementos armazenados
isEmpty()	Retorna true se a estrutura estiver vazia
contains(Object)	Retorna true se o objeto está armazenado na estrutura
iterator()	Retorna um objeto que permite percorrer a estrutura
toArray()	Devolve um vetor com os dados da estrutura
add(E)	Adiciona o objeto na estrutura
remove(Object)	Remove o objeto da estrutura
containsAll(Collection)	Retorna true se todos os elementos pertencerem à coleção
addAll(Collection)	Adiciona todos os elementos na coleção
removeAll	Remove todos os elementos que pertencem à coleção
retainAll(Collection)	Mantém apenas os elementos que estão na coleção informada como parâmetro
clear()	Remove os elementos da coleção

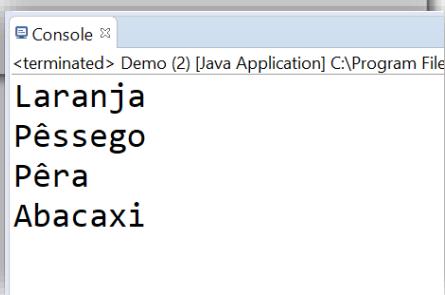
# Exemplo

- O **ArrayList** implementa indiretamente **Collection**:

```
Collection<String> frutas = new ArrayList<>();
frutas.add("Laranja");
frutas.add("Morango");
frutas.add("Pêssego");
frutas.add("Pêra");
frutas.add("Abacaxi");
frutas.remove("Morango");

for (String fruta: frutas) {
    System.out.println(fruta);
}

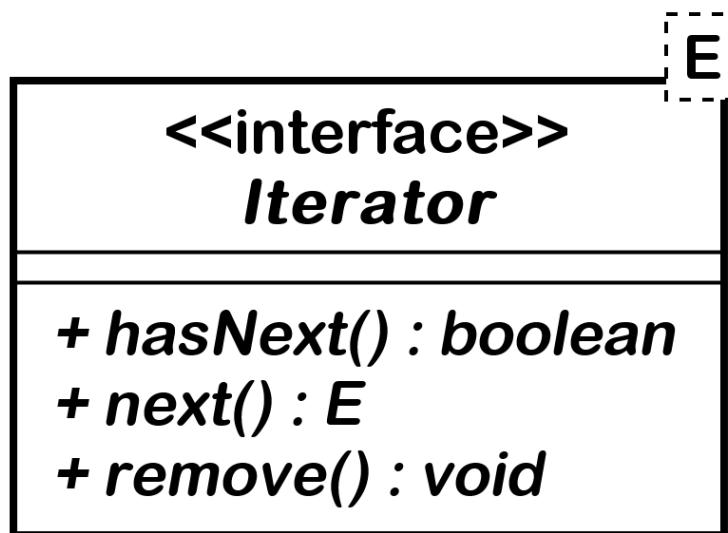
frutas.clear();
```



A screenshot of a Java application's console window. The title bar says "Console". The text area contains the following output:  
<terminated> Demo (2) [Java Application] C:\Program File  
Laranja  
Pêssego  
Pêra  
Abacaxi

# O método iterator()

- A interface **Collection** prevê que todas as classes concretas que a implemente, retorne um *iterador*, através do método **iterator()**.



Um *iterador* é um objeto que possibilita percorrer uma estrutura de dados.

Método	Descrição
hasNext()	Retorna true se tem objetos na coleção ainda não lidos
next()	Retorna um objeto da coleção
remove()	Remove o último objeto lido pelo iterador (através de next()), da coleção

# Utilidade do iterador

Remover as frutas que começam com “P”:

```
Collection<String> frutas = new ArrayList<>();
frutas.add("Laranja");
frutas.add("Morango");
frutas.add("Pêssego");
frutas.add("Pêra");
frutas.add("Abacaxi");

for (String fruta : frutas) {
    if (fruta.startsWith("P")) {
        frutas.remove(fruta);
    }
}
```

Não é seguro, pois lança a exceção:

```
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
at teste.Demo.main(Demo.java:18)
```

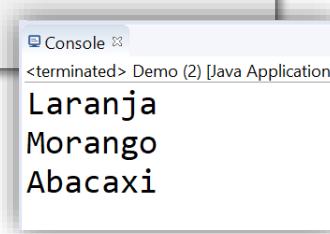
# Utilidade do iterador

- Usando o iterador:

```
Collection<String> frutas = new ArrayList<>();
frutas.add("Laranja");
frutas.add("Morango");
frutas.add("Pêssego");
frutas.add("Pêra");
frutas.add("Abacaxi");

Iterator<String> iterador = frutas.iterator();
while (iterador.hasNext()) {
    String fruta = iterador.next();
    if (fruta.startsWith("P"))
        iterador.remove();
}

for (String fruta : frutas) {
    System.out.println(fruta);
}
```



# Interface List

E	
<b>&lt;&lt;interface&gt;&gt;</b> <b>List</b>	
+ <i>get(index : int) : E</i>	
+ <i>set(index : int, element : E) : E</i>	
+ <i>add(index : int, element : E) : void</i>	
+ <i>indexOf(o : Object) : int</i>	
+ <i>lastIndexOf(o : Object) : int</i>	
+ <i>listIterator() : ListIterator&lt;E&gt;</i>	
+ <i>subList(fromIndex : int, toIndex : int) : List&lt;E&gt;</i>	
Método	Descrição
<i>get(int)</i>	Retorna o objeto que ocupa a posição informada
<i>set()</i>	Altera o objeto armazenado na posição indicada
<i>add()</i>	Acrescenta um objeto na posição indicada
<i>indexOf()</i>	Procura por um objeto
<i>lastIndexOf()</i>	Retorna a posição da última ocorrência do objeto
<i>listIterator()</i>	Devolve um iterador que suporta navegação bidirecional
<i>subList()</i>	Retorna uma sub-lista

E
<b>&lt;&lt;interface&gt;&gt;</b> <b>ListIterator</b>
+ <i>hasNext() : boolean</i>
+ <i>next() : E</i>
+ <i>hasPrevious() : boolean</i>
+ <i>previous() : E</i>
+ <i>nextIndex() : int</i>
+ <i>previousIndex() : int</i>
+ <i>remove() : void</i>

Interface que prevê a navegação bidirecional

# Ordenação de coleções

# Ordenação de objetos de uma coleção

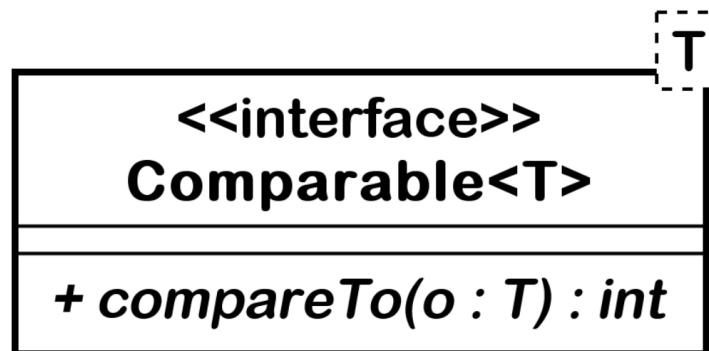
- Numa lista contendo objetos da classe **Aluno**, quais atributos devem ser usados como critério para ordenação?

Aluno
- matricula : int
- nome : String
- endereco : String
- dataNascimento : LocalDate

- Existem dois conceitos de ordenação em Java:
  - Ordenação **natural** – é a ordenação típica de objetos de uma determinada classe. Cada classe possui apenas uma ordenação natural;
  - Ordenação **artificial** – uma forma alternativa de ordenar objetos de uma classe. Cada classe pode possuir várias ordenações artificiais

# Definindo a ordenação natural

- A ordenação natural de objetos de uma classe é especificada ao implementar a interface **Comparable**:



- O método **compareTo()** deve retornar:
  - 1**: se **this < outro**
  - 0**: se **this** for igual à **outro**
  - +1**: se **this > outro**

# Exemplo

```
public class Aluno implements Comparable<Aluno> {

    private int matricula;
    private String nome;
    private String endereco;
    private LocalDate dataNascimento;

    @Override
    public int compareTo(Aluno o) {
        if (this.matricula < o.matricula)
            return -1;
        if (this.matricula > o.matricula)
            return +1;
        return 0;
    }
}
```

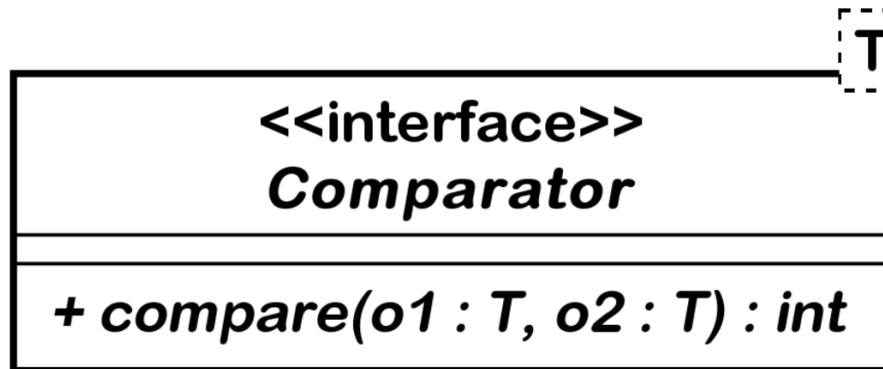
# Ordenando uma coleção

- Para ordenar coleções, utilizar o método `sort()` da classe `Collections`.

```
ArrayList<Aluno> alunos = new ArrayList<>();  
  
alunos.add(new Aluno(1010, "José", "R.XV de Novembro 192",  
LocalDate.of(2000,2,18)));  
  
alunos.add(new Aluno(6925, "Marta", "R. 7 de setembro,  
942",LocalDate.of(2001,9,7)));  
  
Collections.sort(alunos);
```

# Definindo a ordenação artificial

- A ordenação artificial de objetos de uma classe é especificada ao implementar a interface **Comparator**:



- O método **compare()** deve retornar:
  - 1: se **o1 < o2**
  - 0: se **o1** for igual à **o2**
  - +1: se **o1 > o2**

# Exemplo

```
import java.util.Comparator;

public class OrdenacaoPorNome implements Comparator<Aluno> {

    @Override
    public int compare(Aluno o1, Aluno o2) {
        return o1.getNome().compareTo(o2.getNome());
    }

}
```

```
ArrayList<Aluno> alunos = new ArrayList<>();
alunos.add(new Aluno(1010, "Zé", "R.XV de Novembro 192", LocalDate.of(2000,2,18)));
alunos.add(new Aluno(6925, "Marta", "R. 7 de setembro, 942", LocalDate.of(2001,9,7)));

Collections.sort(alunos, new OrdenacaoPorNome());
```

# Exemplo

```
import java.util.Comparator;

public class OrdenacaoPorNomeMatricula implements Comparator<Aluno> {

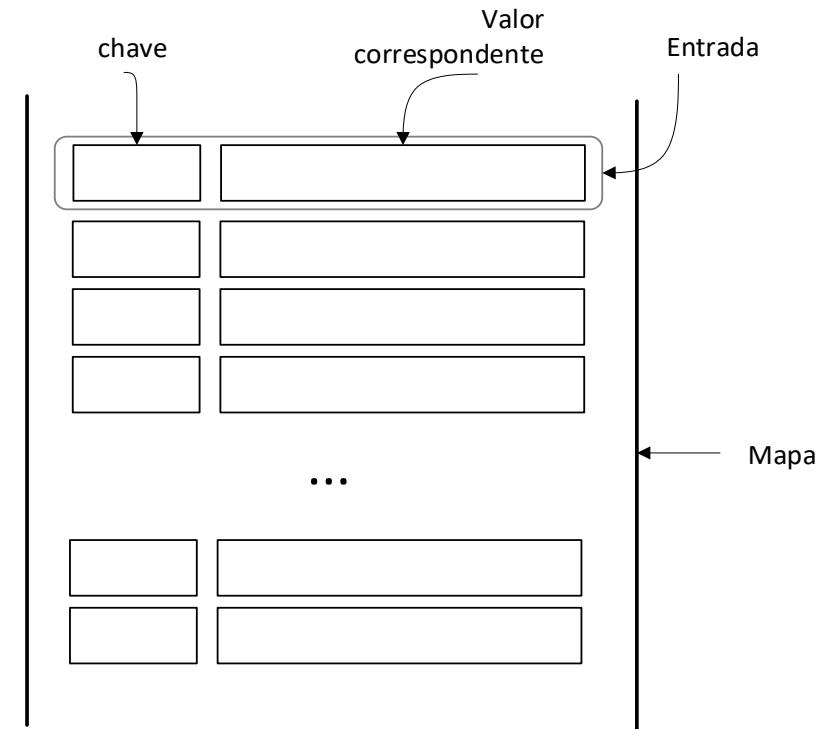
    @Override
    public int compare(Aluno o1, Aluno o2) {
        int ordem = o1.getNome().compareTo(o2.getNome());
        if (ordem == 0) {
            if (o1.getMatricula() < o2.getMatricula()) {
                ordem = -1;
            } else if (o1.getMatricula() > o2.getMatricula()) {
                ordem = +1;
            } else {
                ordem = 0;
            }
        }

        return ordem;
    }
}
```

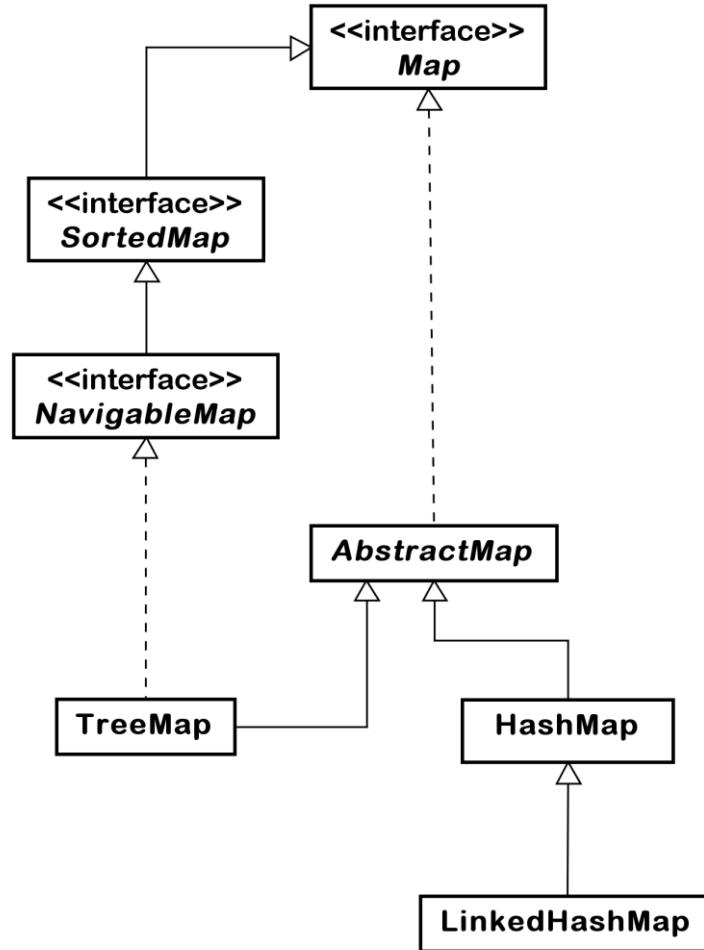
# Mapa de dispersão

# O que é um mapa de dispersão?

- Um mapa armazena uma coleção de pares *chave/valor*.
- As chaves funcionam como índices.
- Cada chave mapeia um valor.
- Permite executar busca, exclusão e atualização de dados (a partir da chave) de forma rápida
- Também conhecido como tabela hash (*hashtable*), mapa de dispersão, etc.



# Mapas de dispersão



# Declaração de uma mapa de dispersão

- Sintaxe para declarar um mapa de dispersão:

```
HashMap<Classe1, Classe2> identificador;
```

Classe  
da chave

Classe do objeto  
a ser armazenado

- Exemplo:

```
HashMap<Integer, Aluno> alunos;
```

# Criação de um mapa de dispersão

- Sintaxe:

```
identificador = new HashMap<>()
```

- Exemplo:

```
alunos = new HashMap<>()
```

# Incluir objetos no mapa de dispersão

- Usar o método `put()` para incluir um objeto no mapa
- Requer informar a chave do objeto
- Exemplo:

```
alunos.put(287812, new Aluno(287812, "José da Silva"));
alunos.put(128444, new Aluno(128444, "Leandro Souza"));
alunos.put(166886, new Aluno(166886, "Marina Albuquerque"));
```

```
Aluno aluno = new Aluno(190443, "Marta Antunes");
alunos.put(190443, aluno);
```

# Localizar um objeto no mapa

- A localização deve ser feita pela chave do objeto.
- Utilizar o método `get()` para localizar o objeto
- Exemplo:

```
Aluno alunoPesquisado = alunos.get(166886);
```

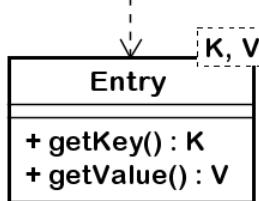


Procura um aluno cuja  
matrícula seja 166886

- Se a chave não tiver sido inserida previamente, o método `get()` retornará `null`.

# A interface Map

<b>&lt;&lt;Interface&gt;&gt;</b> <b>Map</b>	<b>K, V</b>	
	<b>Método</b>	<b>Descrição</b>
<pre>+ size() : int + isEmpty() : boolean + containsKey(key : Object) : boolean + containsValue(value : Object) : boolean + get(key : Object) : V + put(key : K, value : V) : V + remove(key : Object) : V + putAll(m : Map&lt;K,V&gt;) : void + clear() + keySet() : Set&lt;K&gt; + values() : Collection&lt;V&gt; + entrySet() : Set&lt;Entry&lt;K,V&gt;&gt; + remove(key : Object, value : Object) : boolean + replace(key : K, value : V) : V</pre>	put(K, V)	Armazena no mapa o par com chave K e valor V. Se a chave já foi inserida, retorna o valor já associado e o altera no mapa. Se a chave ainda não tinha sido inserida, armazena o par no mapa.
	get(Object)	Retorna o valor associado à chave. Retorna null, se não existir (ou se o valor associado for nulo).
	remove(Object)	Remove o objeto a partir da chave informada. Devolve o valor associado à chave removida (ou null, se a chave não existe no mapa ou está associada ao valor null)
	containsKey()	Retorna true se há um par no mapa com a chave informada
	putAll(Map<K,V>)	Copia os dados de um mapa.
	keySet(): Set<K>	Retorna uma coleção do tipo Set contendo as chaves armazenadas no mapa
	values()	Retorna uma coleção do tipo Collection contendo os valores armazenados no mapa.
	entrySet()	Devolve uma coleção do tipo Set com os pares (chave/valor) armazenados no mapa.



Classe de objetos que  
armazenam o par chave/valor

# O método hashCode()

- É responsabilidade da classe da chave implementar o método hashCode()
- O hashCode é um valor que pertence ao objeto. Trata-se de um número de 32 bits, que permite que o objeto seja utilizado numa estrutura de dados *mapa de dispersão*
  - A partir deste número é possível calcular qual a posição em que se encontra o par
- Dois objetos iguais devem retornar o mesmo hashCode.
  - Se dois objetos (*a* e *b*) são iguais, isto é, o *a.equals(b) == true*, então o método hashCode() deve retornar o mesmo valor para os dois objetos.
  - Sempre que o método equals() é implementado, deve-se sobrescrever hashCode() também, e vice versa.
- Dois objetos distintos podem retornar o mesmo hashCode, mas se possível, deve-se evitar.

# Diagrama de classes

