

# Técnicas de Programação I



Curso Superior de Tecnologia em Desenvolvimento de  
Software Multiplataforma

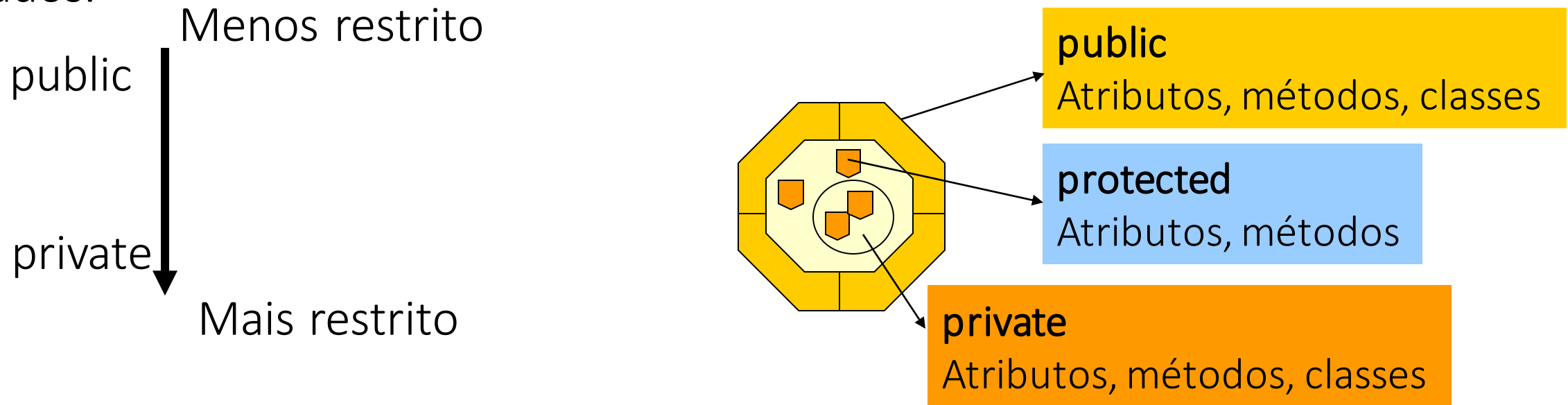
Aula 07

Prof. Claudio Benossi

# Encapsulamento de Dados

# Encapsulamento de Dados

É uma proteção adicional dos dados do objeto de possíveis modificações impróprias, forçando o acesso a um nível mais baixo para tratamento de dados.



## Exemplo:

*Quando temos um arquivo protegido por senha de acesso, podemos dizer que ele está protegido, pois, apenas podemos lê-lo sem fazermos alteração*

# Modificadores de Acesso

- ✓ Há três **modificadores** de acesso: **public**, **protected** e **private**;
- ✓ Atributos e Métodos podem ter os **três níveis** de acesso.
- ✓ Elementos **públicos** podem ser acessados diretamente por qualquer outra classe, utilizando um ponto (.) após o nome da variável.
- ✓ Elementos **privados** e **protegidos** não podem ser acessados diretamente utilizando o ponto.

# Modificadores de Acesso

public	<ul style="list-style-type: none"><li>✓ Pode ser acessado por qualquer classe</li></ul>
protected	<ul style="list-style-type: none"><li>✓ Pode ser acessado por qualquer sub-classe</li><li>✓ Pode ser acessado por qualquer classe do mesmo pacote</li><li>✓ Pode ser acessado pela própria classe</li></ul>
default	<ul style="list-style-type: none"><li>✓ Pode ser acessado por qualquer classe do mesmo pacote</li><li>✓ Pode ser acessado pela própria classe</li></ul>
private	<ul style="list-style-type: none"><li>✓ Pode ser acessado somente pela própria classe</li></ul>

# Exercícios da aula anterior ...

De acordo com a classe **Funcionário** abaixo, crie um construtor sem parâmetros (vazio) que deverá atribuir ao cargo o valor “assistente” e um outro construtor que recebe parâmetros correspondentes aos atributos.

Funcionario
- cracha: int - salario: float - cargo: String
Funcionario() Funcionario(c: int, s: float, car: String) //Métodos de acesso calculaAumento(porcentagem: float) calculaAumento(tempo: int)

Este método aplica a porcentagem de aumento no salário.

Este método soma R\$150,00 no salário para cada ano trabalhado (recebido por parâmetro).

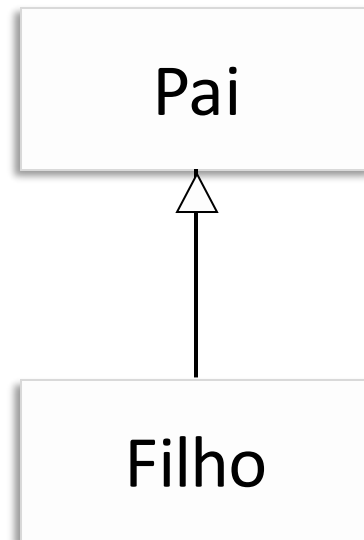
# Lembrando. Herança e Modificadores de Acesso.

- ✓ **protected**: podem ser acessados pelos membros da subclasse
- ✓ **private**: só podem ser acessados pela própria classe



## Super e Sub classe:

A nomenclatura mais encontrada é que Pai é a **superclasse** de Filho. Desta forma, Filho é uma **subclasse** de Pai. Pode-se dizer também que todo Filho **é um** Pai.



# Lembrando – Classes Abstratas vs Interfaces



Use classes abstratas quando você quiser definir um “template” para subclasses e você possui alguma implementação (métodos concretos) que todas as subclasses podem utilizar.



Use interfaces quando você quiser definir uma regra que todas as classes que implementem a interface devem seguir, independentemente se pertencem a alguma hierarquia de classes ou não.

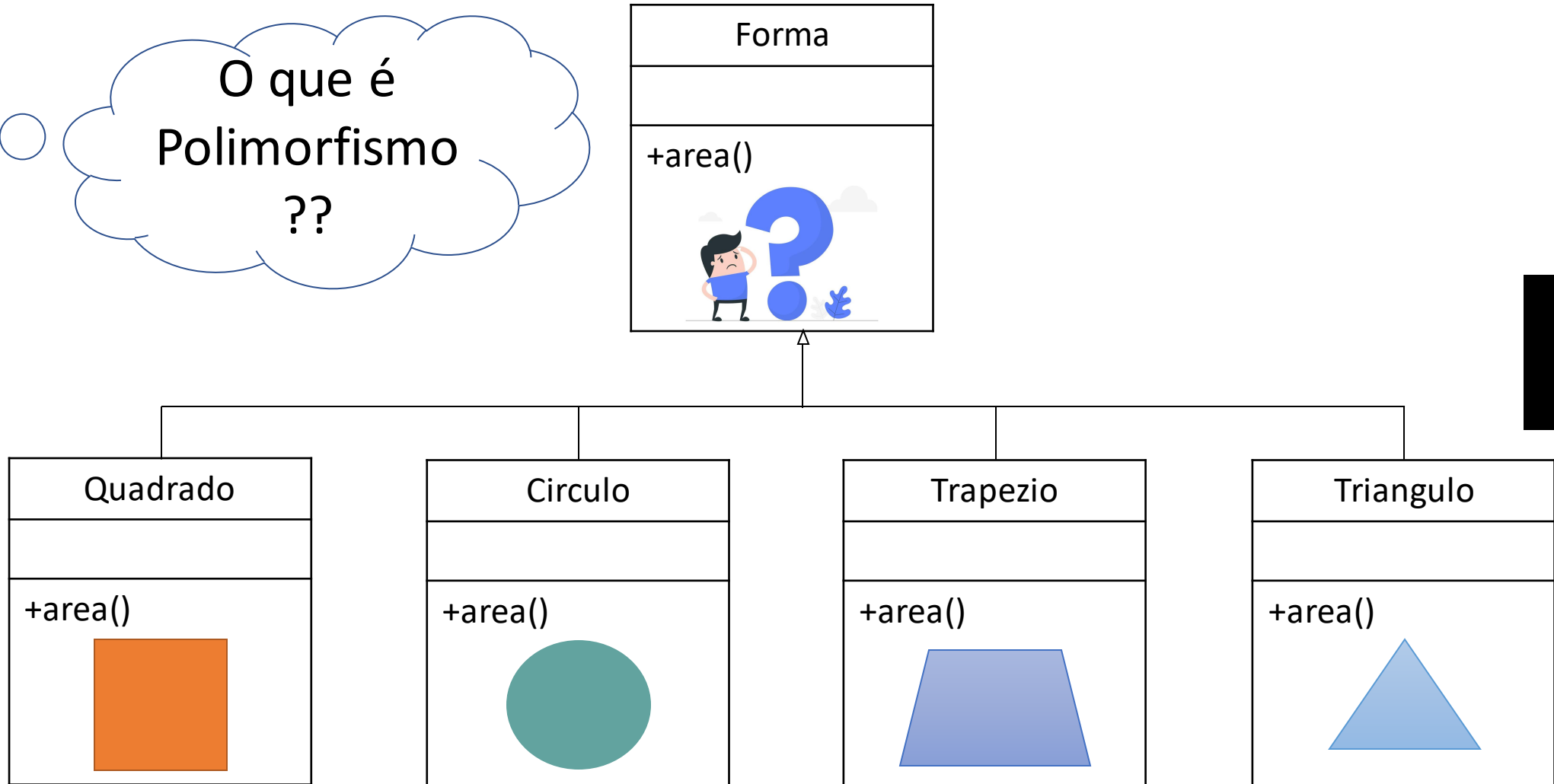


# Polimorfismo

# Polimorfismo

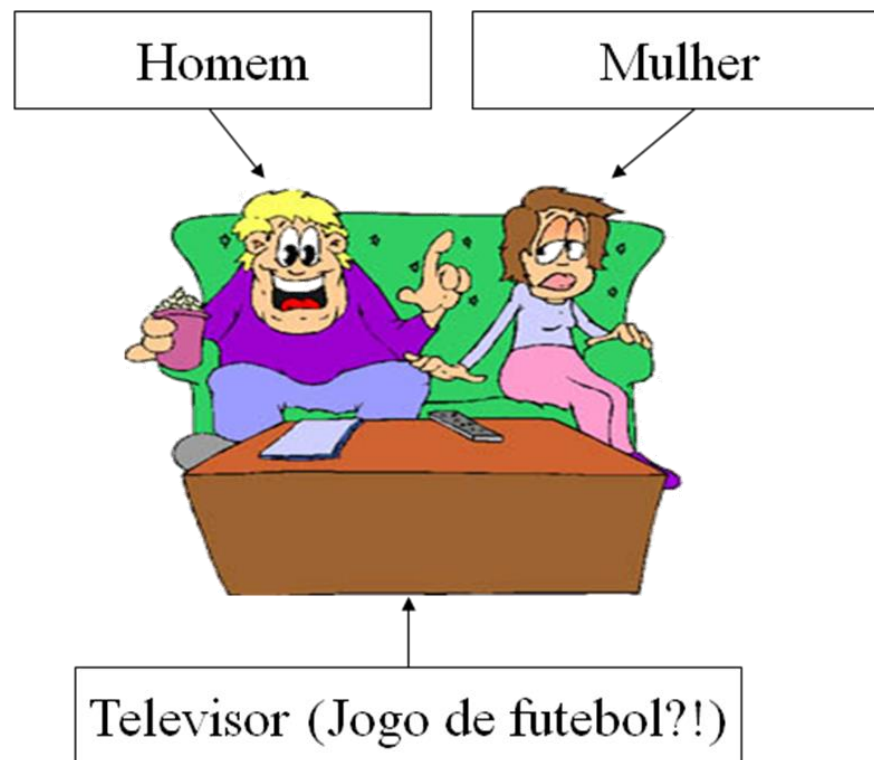


O que é  
Polimorfismo  
??



# Polimorfismo

É a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras.





# Polimorfismo

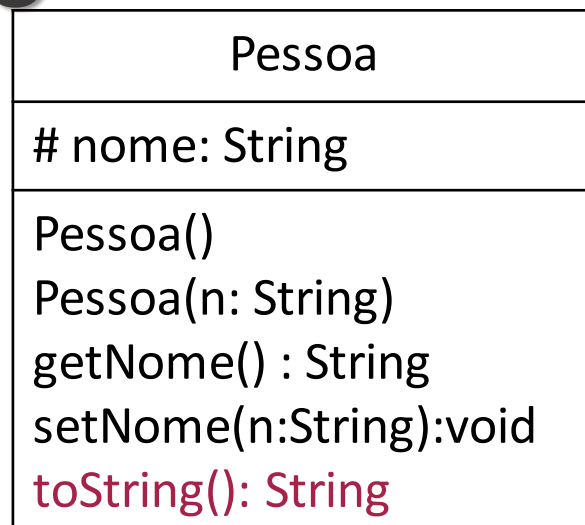
O termo polimorfismo é originário do grego e significa "**muitas formas**" (poli = muitas, morphos = formas).

O polimorfismo permite a um objeto ter várias formas.

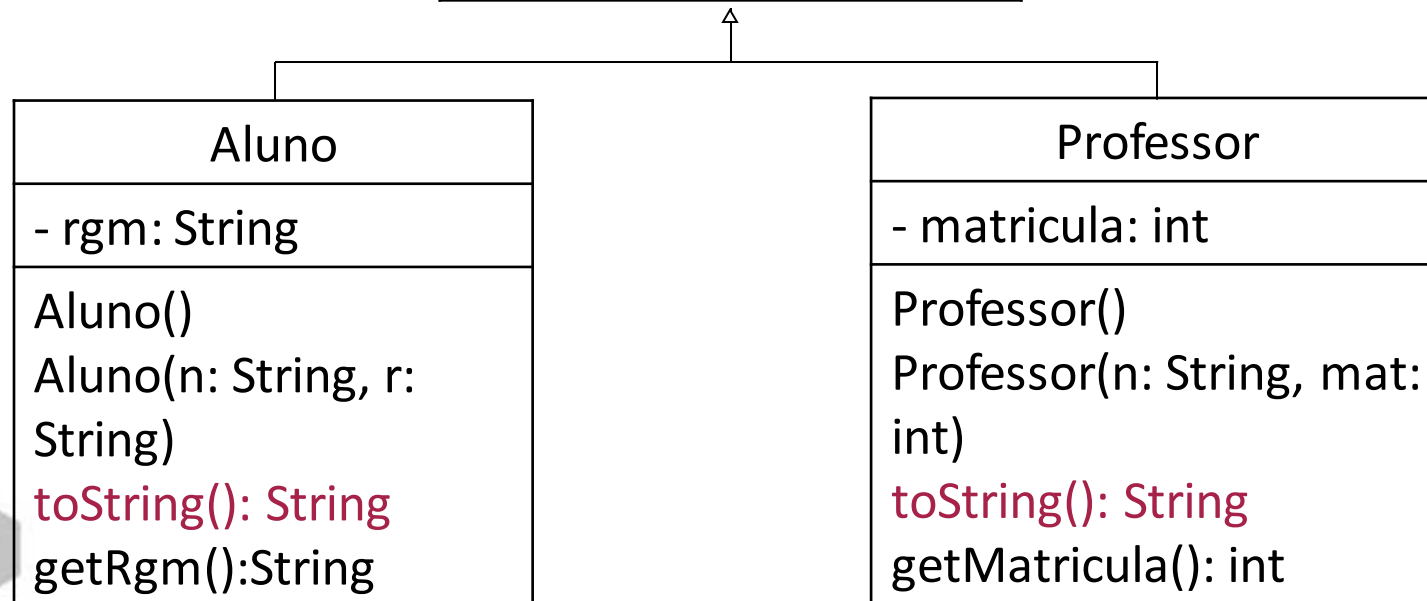
Um objeto se torna polimórfico quando é declarado como uma de suas superclasses e se instancia uma de suas subclasses.



# Exemplo



Pessoa s = new Aluno("Fulano","33333-3")



# Exemplo

```
Pessoa p[] = new Pessoa[3];  
p[0] = new Pessoa("José");  
p[1] = new Aluno("Lukas", "11111-1");  
p[2] = new Professor("Obama", 111223);
```

```
for (int i=0; i < p.length; i++){  
    System.out.println(p[i].toString());  
}
```

← Irá chamar o método toString() dependendo do tipo do objeto

Saída


```
run:  
José  
11111-1 : Lukas  
111223 : Obama
```

## Regras em tempo de compilação **vs** Regras em tempo de execução

### Tempo de compilação:

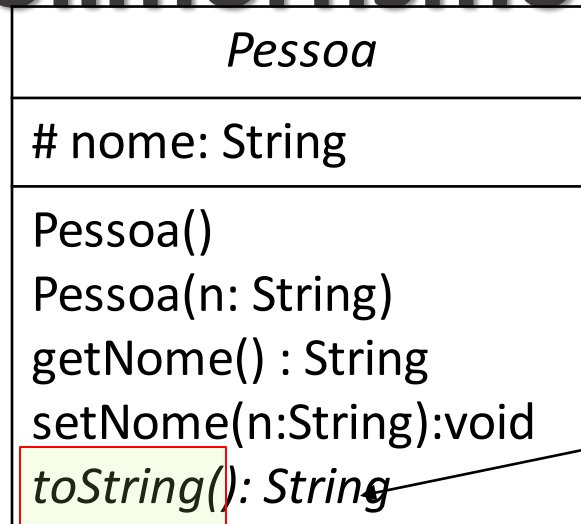
- ✓ O compilador só conhece o tipo de referência dos objetos (superclasse)
- ✓ Verifica os métodos chamados somente na classe da referência

### Tempo de execução:

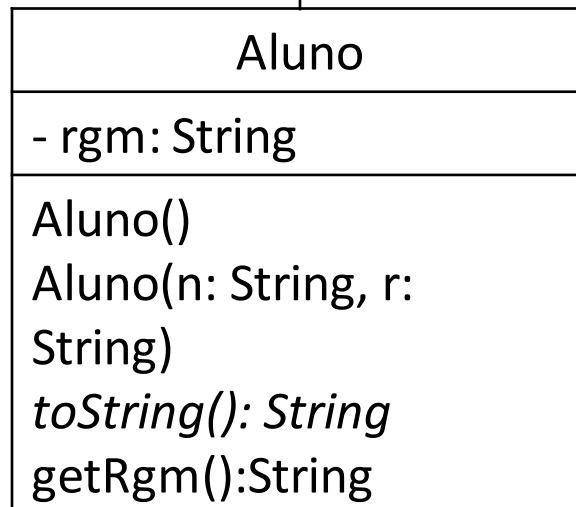
- ✓ Procura pelo tipo exato do objeto para achar um método
  - ✓ Combina o método em tempo de execução com o método apropriado do objeto da classe
- 

# Polimorfismo: tempo de compilação

Classe de referência



```
Pessoa s = new Aluno("Fulano","33333-3")
s.toString();
```

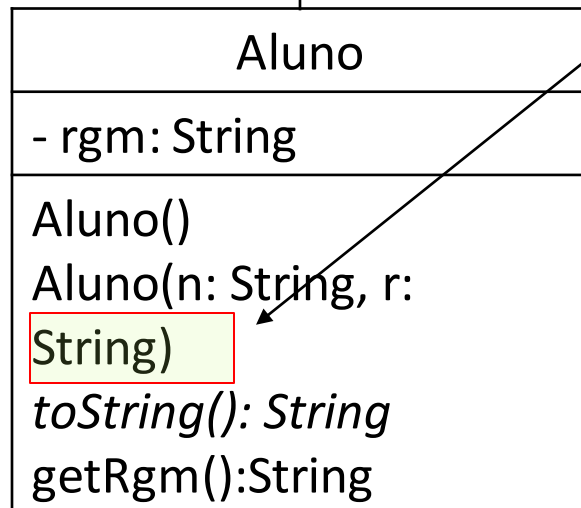
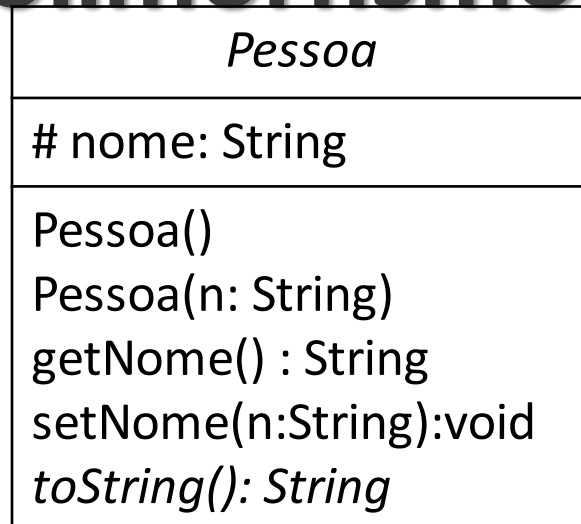


Em tempo de compilação, o compilador irá verificar se existe o método com a assinatura **String toString()** na classe de referência, neste caso, a classe Pessoa.



# Polimorfismo: tempo de execução

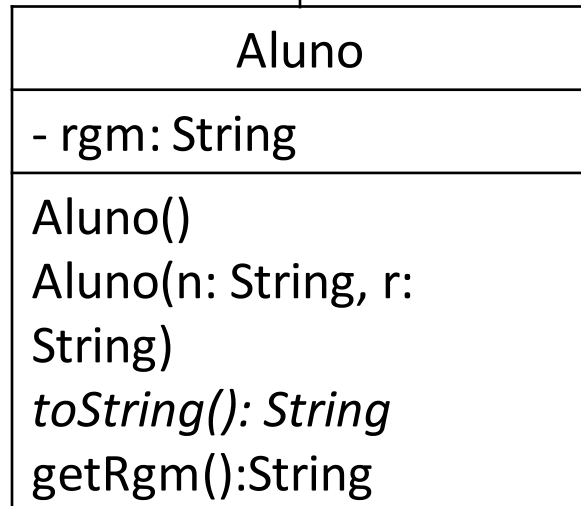
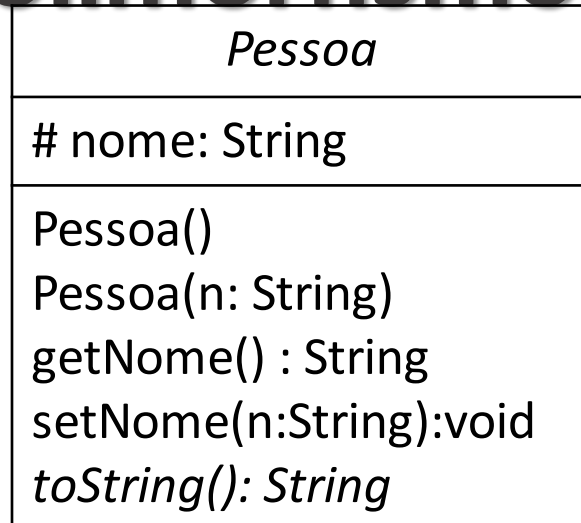
Classe de referência



```
Pessoa s = new Aluno("Fulano","33333-3")
s.toString();
```

Em tempo de execução, será levado em consideração a classe a qual o objeto foi instanciado, e com isso, será executado o método com a assinatura **String toString()** na classe de referência, neste caso, a classe Aluno.

# Polimorfismo



```
Pessoa s = new Aluno("Leo", "22222-2");
s.getRgm();
```

Ocorre erro de compilação, pois em tempo de compilação, o compilador irá verificar se existe o método `getRgm()` na classe `Pessoa`.

Correto, com o uso de Casting

```
Pessoa s = new Aluno("Leo", "22222-2");
s.getRgm();
( (Aluno)s ).getRgm();
```

# Polimorfismo

No exemplo anterior, poderíamos ter uma instância de Pessoa, neste caso teríamos um erro em tempo de execução (ClassCastException) na chamada do método getRgm().

```
Pessoa s = new Pessoa("Leo");  
( (Aluno)s ).getRgm();
```

Erro em tempo de execução

Uma solução para esse problema, é utilizar o relacionamento “is-a”, ou seja, verificar se um objeto é de um certo tipo de classe com o comando **instanceof**.

```
Pessoa s = new Pessoa("Leo");  
//System.out.println( ( (Aluno)s ).getRgm() );  
if (s instanceof Aluno){  
    //só irá executar se S for do tipo ALUNO  
    System.out.println( ( (Aluno)s ).getRgm() );  
}
```

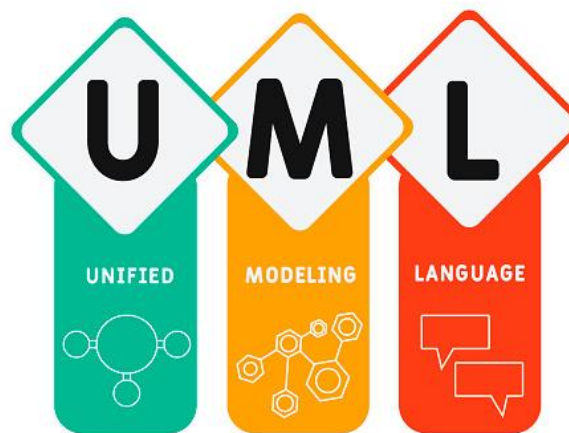
Indica se o objeto **já instanciado** pertence a uma classe específica.

# Associação

A **associação** é quando temos uma relação entre duas classes, na qual estas classes são independentes.

É utilizada quando um atributo de uma classe é do tipo de uma outra classe.

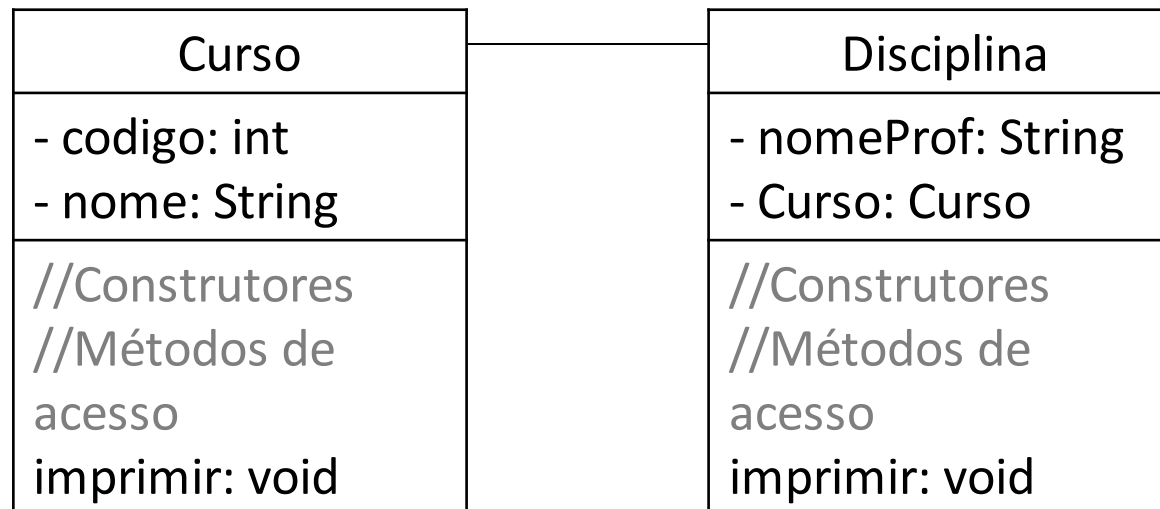
\_\_\_\_\_ convenção  
utilizada



# Associação: Exemplo

Por exemplo, uma disciplina pode estar associada à um curso. Neste caso, não existe um relacionamento de posse entre esses objetos.

- ✔ Todos os objetos são independentes.
- ✔ Uma disciplina pode existir sem a necessidade de um curso, da mesma forma que é possível existir um curso sem a necessidade da existência de uma disciplina.



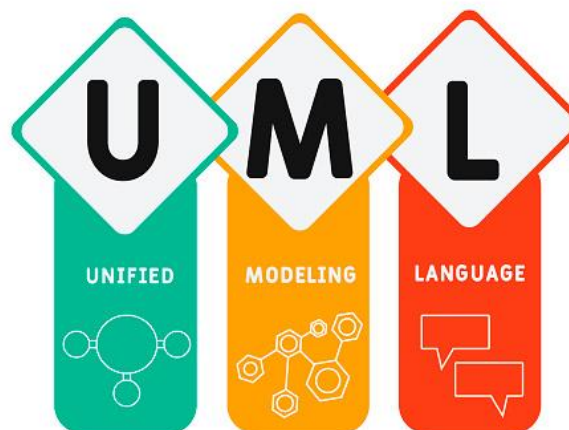
# Agregação

A **agregação** é um tipo especial de associação onde as informações de um objeto precisam ser complementadas pelas informações contidas em um ou mais objetos de outras classes.

Representa um todo (uma classe) que é composto de várias partes (outras classes).

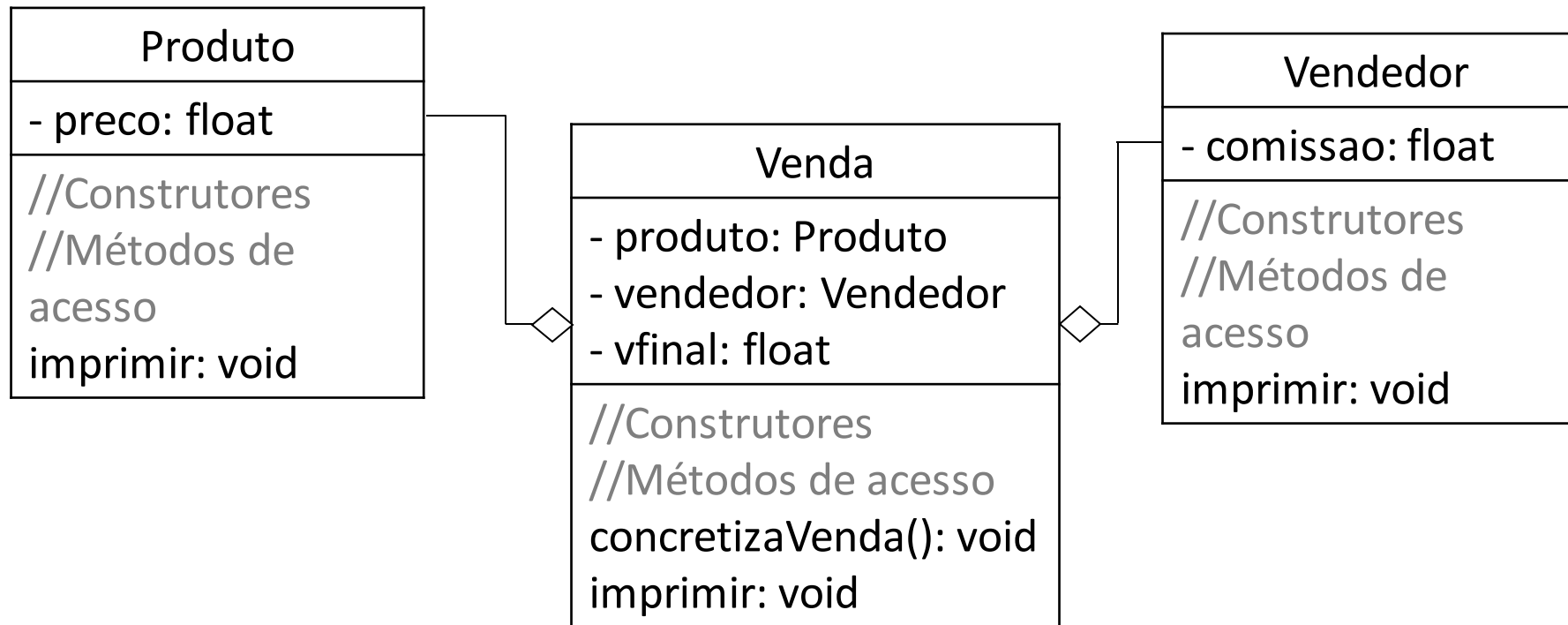


convenção  
utilizada



# Agregação - Exemplo

Por exemplo, uma venda (classe Venda) tem agregada a ela dados do produto (classe Produto) e do vendedor (classe Vendedor)



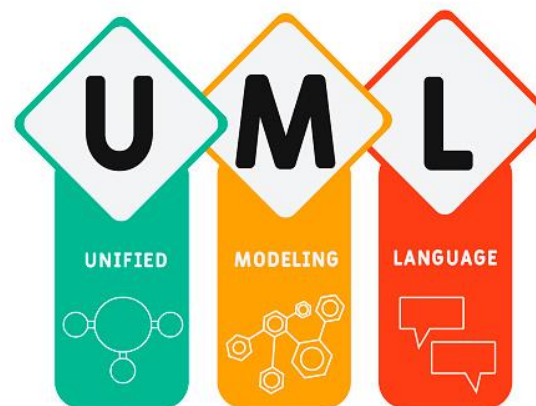
Venda é o objeto definido como sendo o todo. E este objeto somente pode existir caso os demais (Produto e Vendedor) também existam.

# Composição

A **composição** é um relacionamento semelhante a agregação, que especifica que uma classe está composta por objetos de outras classes, ou seja, uma classe representando o **todo** e outras classes **partes**.

- ✓ A classe composta (contêiner) **está composta por** objetos componentes (instâncias de outras classes). Também podemos dizer que os objetos componentes **fazem parte** da classe composta.
- ✓ Sua principal diferença ocorre quando o objeto todo deixar de existir, os seus objetos partes deixarão de existir também.

◆ convenção utilizada

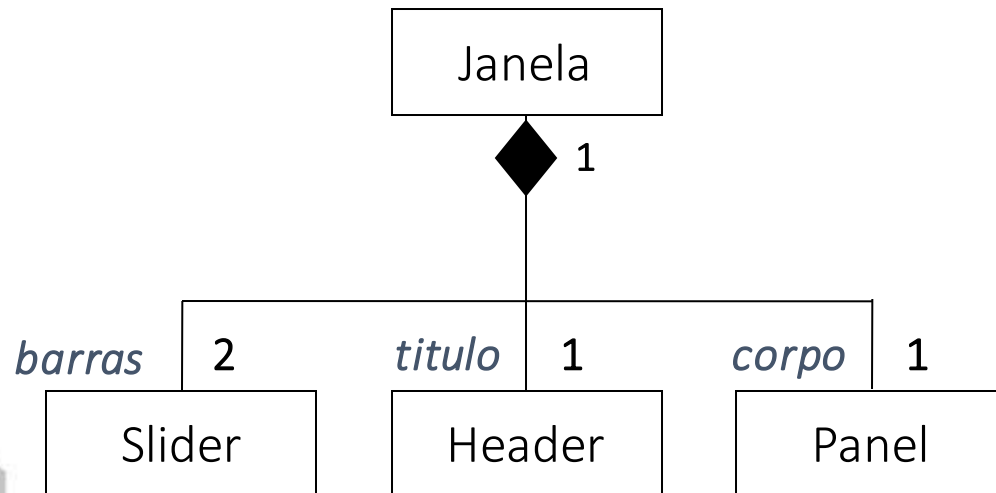




# Composição nos diagramas de classes de UML



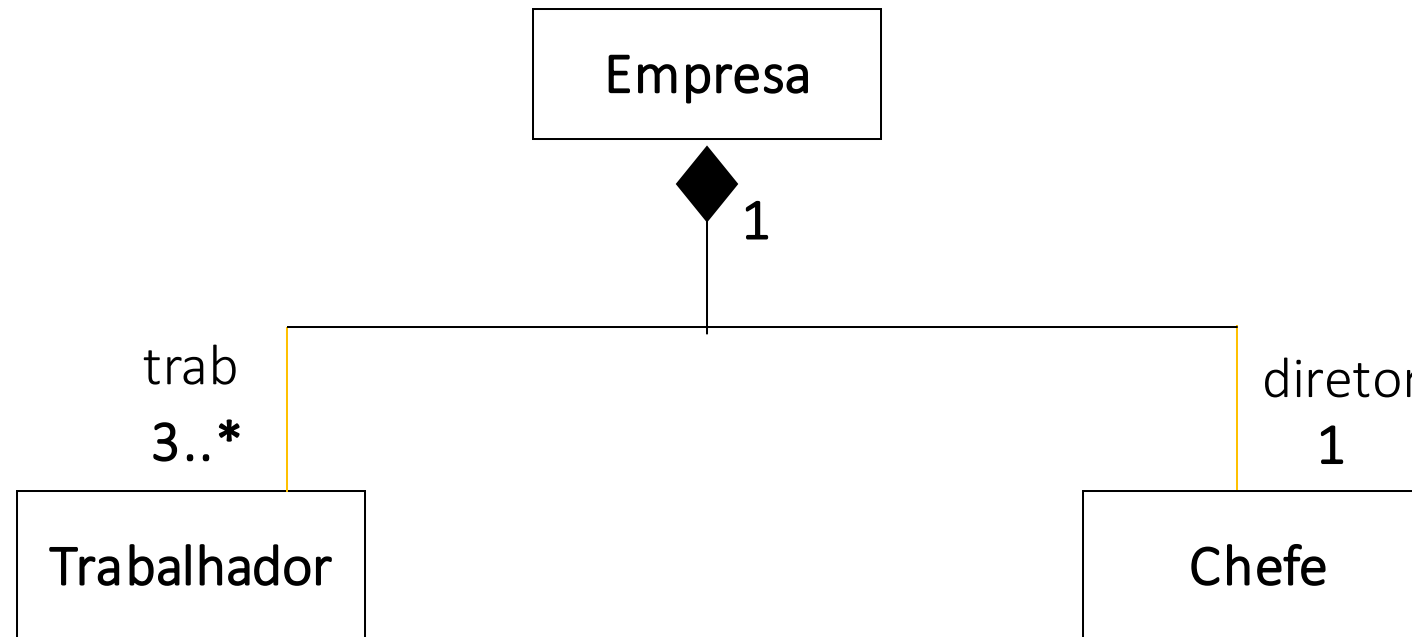
Classe Janela, especificando seus dados membros, que são Objetos das classes Slider, Header e Panel.



## Diagrama com relacionamentos de composição:

- ✓ A classe *Janela* é composta por 2 objetos (*barras*) da classe *Slider*, 1 (*titulo*) da classe *Header* e 1 (*corpo*) da classe *Panel*.
- ✓ Os retângulos são classes e nas setas (com losangos preenchidos) colocamos os nomes dos objetos e a **cardinalidade** ou **multiplicidade**.

# Composição: exemplo



- ✓ Uma **empresa** estará composta por 3 ou mais trabalhadores e um diretor (Chefe)
- ✓ **trab** e **diretor** são **objetos**
- ✓ **Empresa**, **Trabalhador** e **Chefe** são **classes**

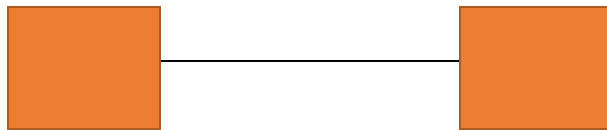
# Composição: exemplo em Java

```
class Trabalhador {    ...    }  
class Chefe extends Trabalhador {    ...    }
```

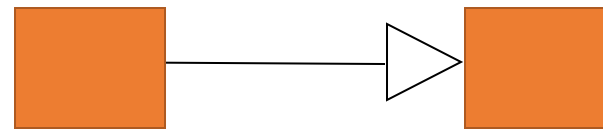
```
class Empresa {  
    // objetos da classe Trabalhador:  
    private Trabalhador trab [ ];  
    // objetos de outras classes:  
    private Chefe diretor;  
}
```

composição!

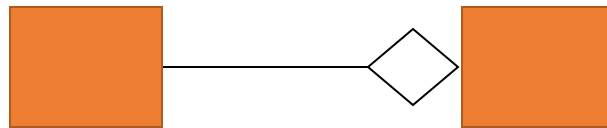
# Diagramas de Classes UML: Relacionamentos



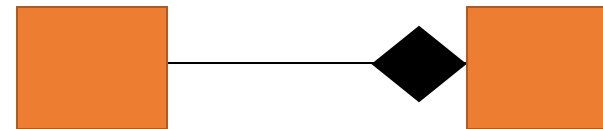
Associação



Generalização / especialização



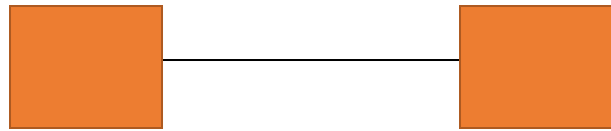
Agregação



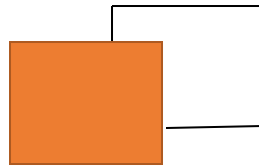
Composição

- ✓ A **agregação** e a **composição** são relacionamentos do tipo **faz parte de**, sendo que na composição as partes são possuídas e não compartilhadas e na agregação as partes têm uma existência “independente”.
- ✓ Observe que na composição o losango aparecerá preenchido na cor preta.
- ✓ Em todos os casos os retângulos são classes que aparecem em diferentes relacionamentos.

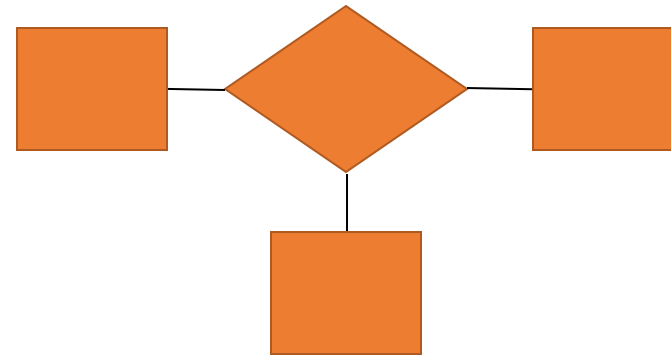
# Diagramas de Classes UML: Grau



associação binária



associação  
reflexiva



associação ternária



**Obs:** Em todos os casos os retângulos são classes que aparecem em diferentes relacionamentos.

# Diagramas de Classes UML: Multiplicidade



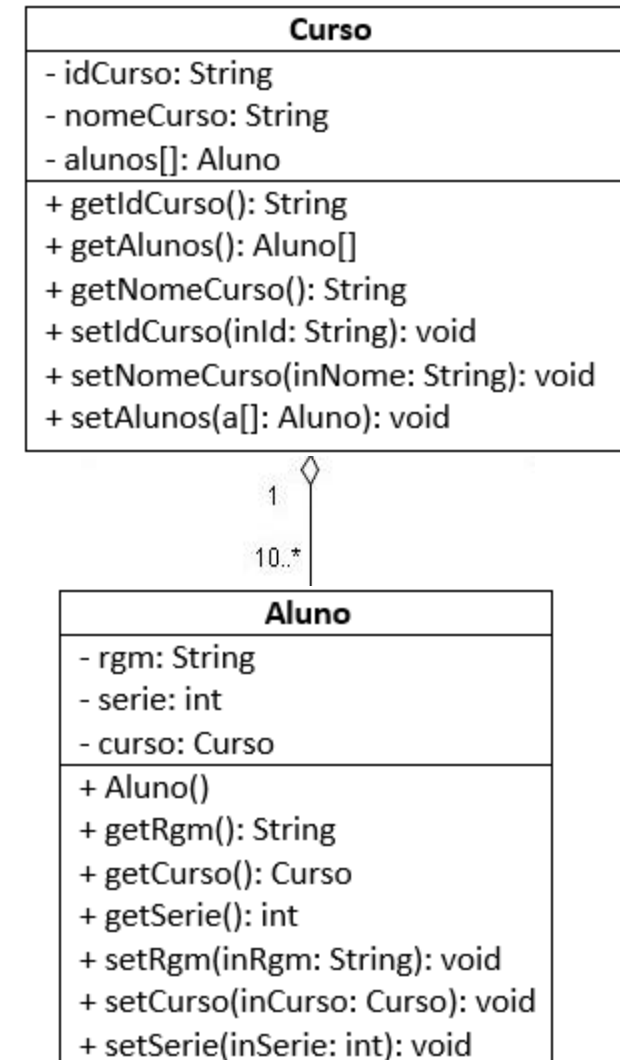
**Obs:** Em todos os casos os retângulos são classes que aparecem em diferentes relacionamentos.

# Diagrama de Classe com Agregação

O relacionamento de **agregação** entre as classes explica que os alunos fazem **parte de** um curso.

Relacionamentos de agregação, composição e associação poderão ter **multiplicidade**.

No exemplo, um curso poderá estar composto por 10 ou mais alunos e um aluno estará em 1 curso

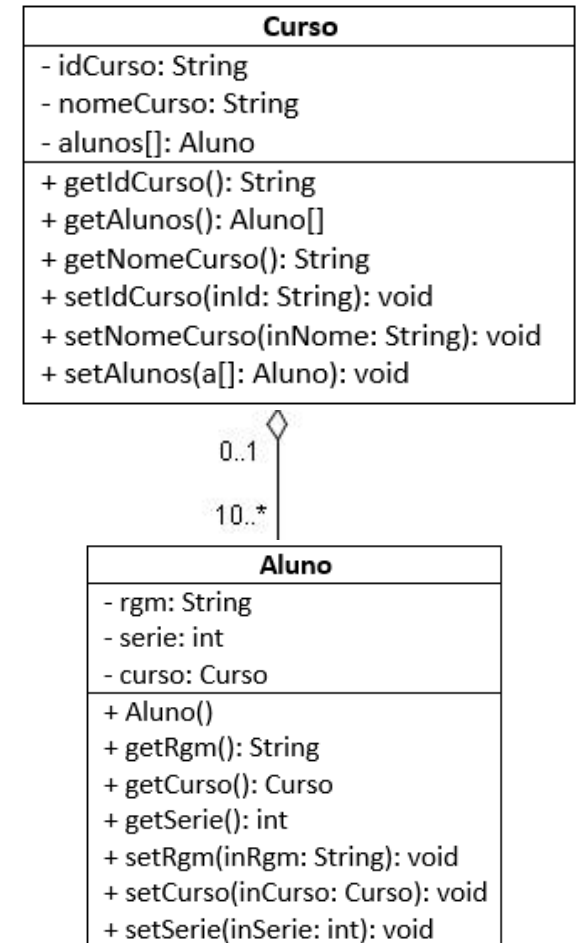


# Diagrama de Classes: Analisando a Multiplicidade

Veja algumas possibilidades de **multiplicidade** no Diagrama de classes, com as classes Aluno e Curso.

- ✓ Nos três casos o relacionamento de **agregação** entre as classes explica que os alunos fazem **parte de** um curso.
- ✓ Veja o primeiro caso:

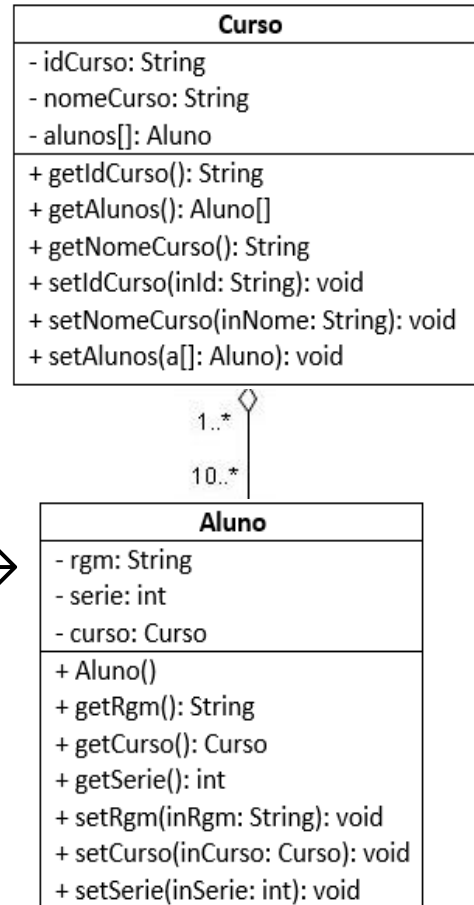
Neste caso, especificamos que um curso poderá estar composto por 10 ou mais alunos e um aluno estará ou fará parte de 1 curso ou de nenhum.



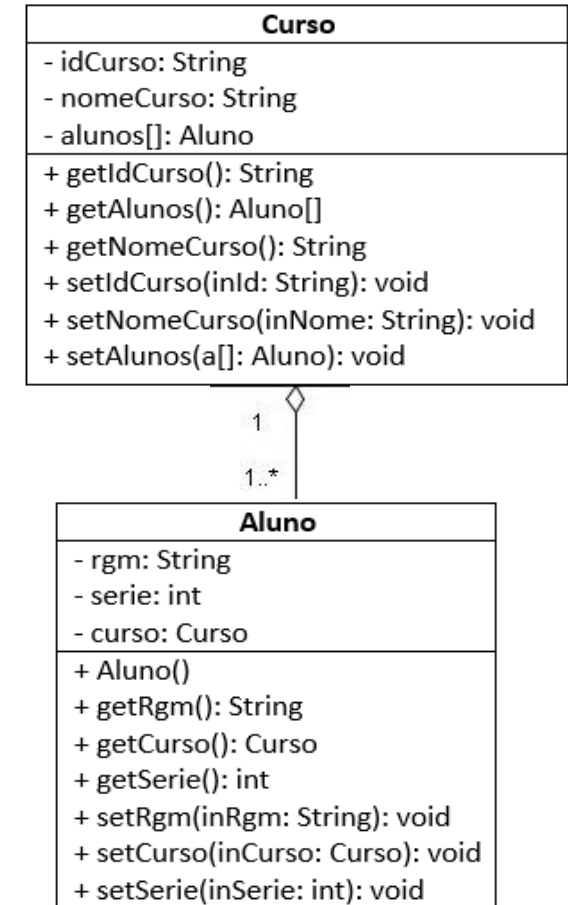


# Diagrama de Classes: Analisando a Multiplicidade

Neste caso, especificamos que um curso poderá estar composto por 10 ou mais alunos e um aluno estará em 1 ou mais cursos.



Neste caso, especificamos que um curso poderá estar composto por 1 ou mais alunos e um aluno estará ou faz parte de 1 curso (somente).



# Composição ou agregação?



 **Composição e agregação são conceitos bastantes semelhantes, mas...**

# Composição e agregação

Segundo a UML\*, os relacionamentos de composição têm as seguintes propriedades:

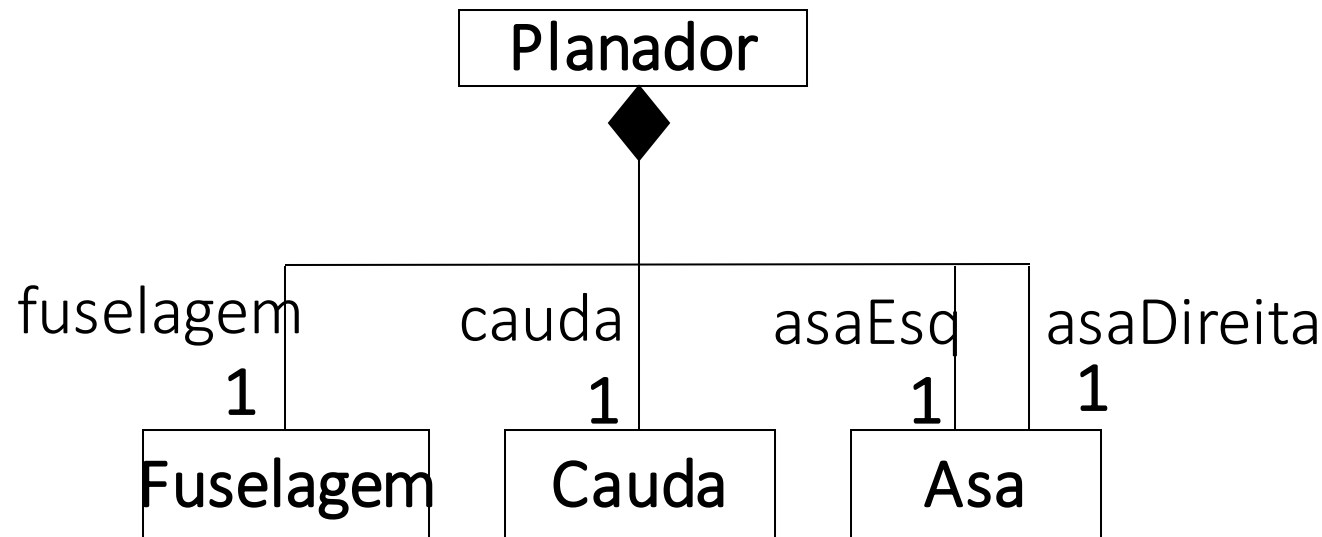
1. No relacionamento somente existirá uma classe que representa o todo (a classe **composta**), que estará composta pelas partes (objetos componentes).
2. As partes no relacionamento de **composição** só existem enquanto o todo existir (não existirão os componentes se não existe a classe composta). O todo (a classe composta) será responsável pela criação e destruição de suas partes (objetos componentes).
3. Uma parte (objeto componente) pode pertencer a somente um todo de cada vez (a somente uma classe composta).

\* (Unified Modeling Language™, do Object Management Group™, [www.uml.org](http://www.uml.org); estudar no Deitel, Java como programar, pp. 78-79, 6ª Ed.)

# Composição e agregação

- ✓ A **agregação** é considerada uma forma mais fraca de composição, porque poderiam ser violadas as propriedades 2 e 3 da composição.
- ✓ Por exemplo: um Computador e um Monitor (“o computador tem um monitor”) participariam de um relacionamento de **agregação**, porque eles poderiam existir independentemente (viola a propriedade 2) e o mesmo monitor poderia ser anexado a vários computadores de uma vez (viola a propriedade 3).
- ✓ Na programação (Java) não existe nada que diferencie composição e agregação.

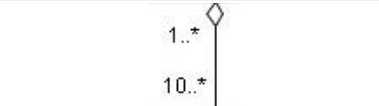
# Comparando exemplos de composição e agregação



## Composição

analise que as partes só existirão se o todo existe e  
que uma parte pertence somente a um todo de cada vez

Curso
- idCurso: String
- nomeCurso: String
- alunos[]: Aluno
+ getIdCurso(): String
+ getAlunos(): Aluno[]
+ getNomeCurso(): String
+ setIdCurso(inId: String): void
+ setNomeCurso(inNome: String): void
+ setAlunos(a[]: Aluno): void



```
classDiagram
    class Curso
    class Aluno
    Curso "1..*" o-- "10..*" Aluno
```

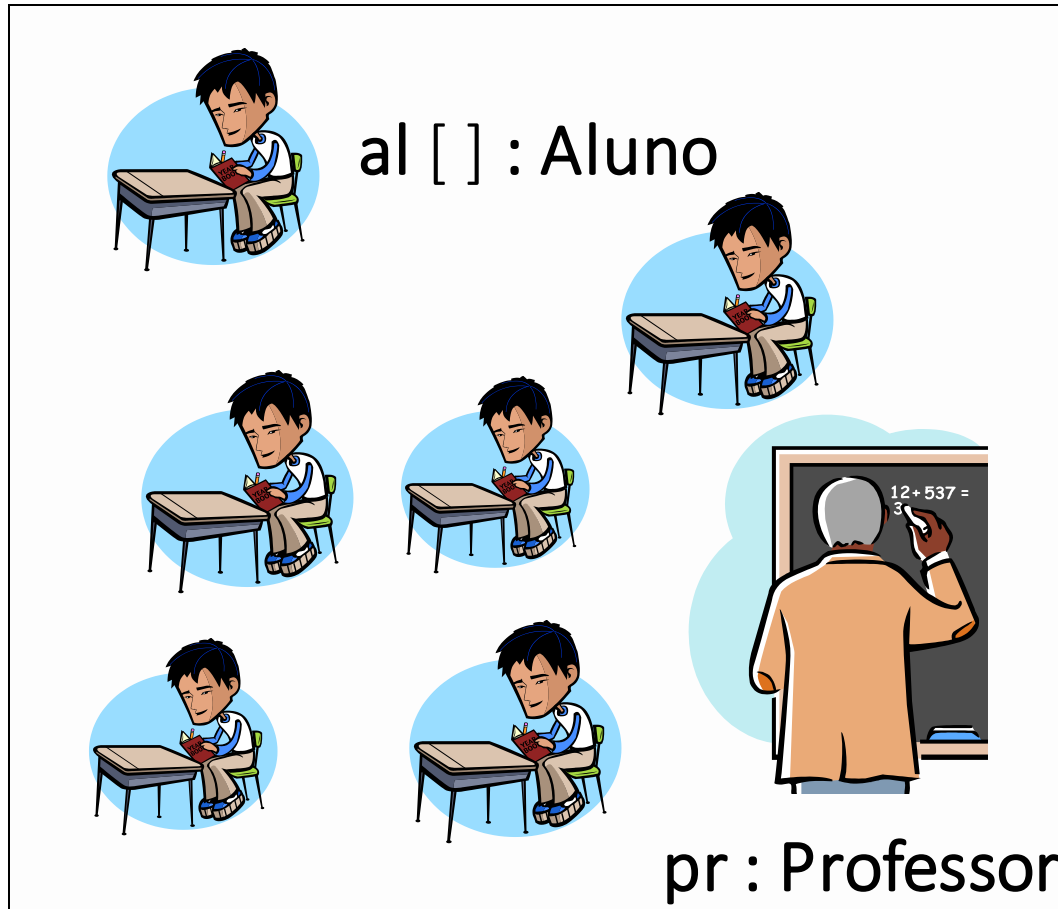
The diagram shows an aggregation relationship between **Curso** and **Aluno**. A hollow diamond symbol is at the **Curso** end, and a line connects it to the **Aluno** class. Multiplicities are 1..\* at the **Curso** end and 10..\* at the **Aluno** end.

Aluno
- rgm: String
- serie: int
- curso: Curso
+ Aluno()
+ getRgm(): String
+ getCurso(): Curso
+ getSerie(): int
+ setRgm(inRgm: String): void
+ setCurso(inCurso: Curso): void
+ setSerie(inSerie: int): void

## Agregação

analise que...

# Na agregação

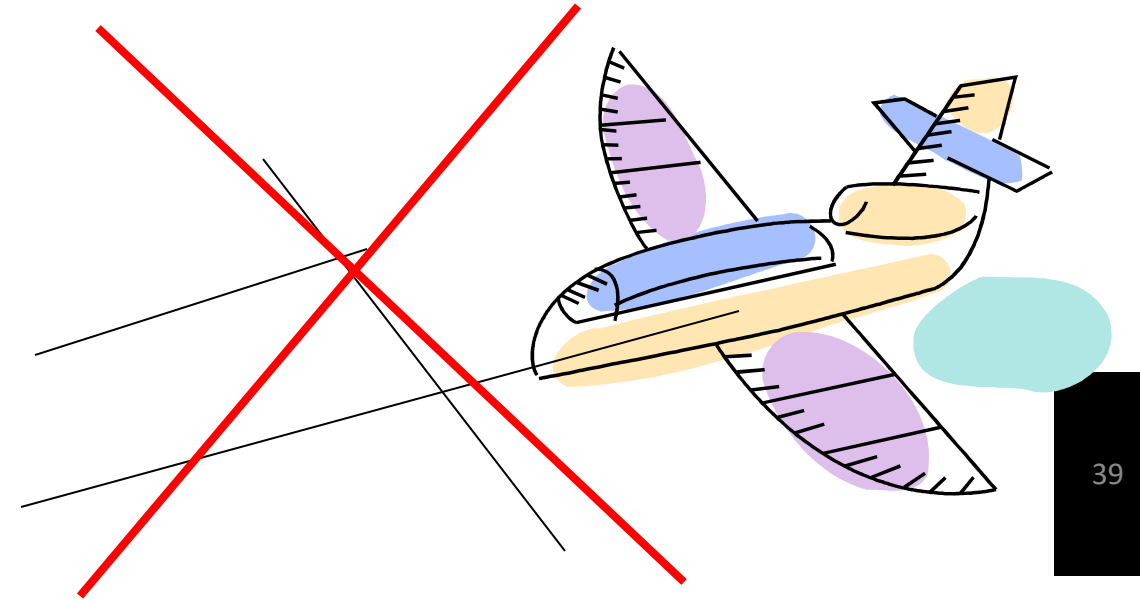
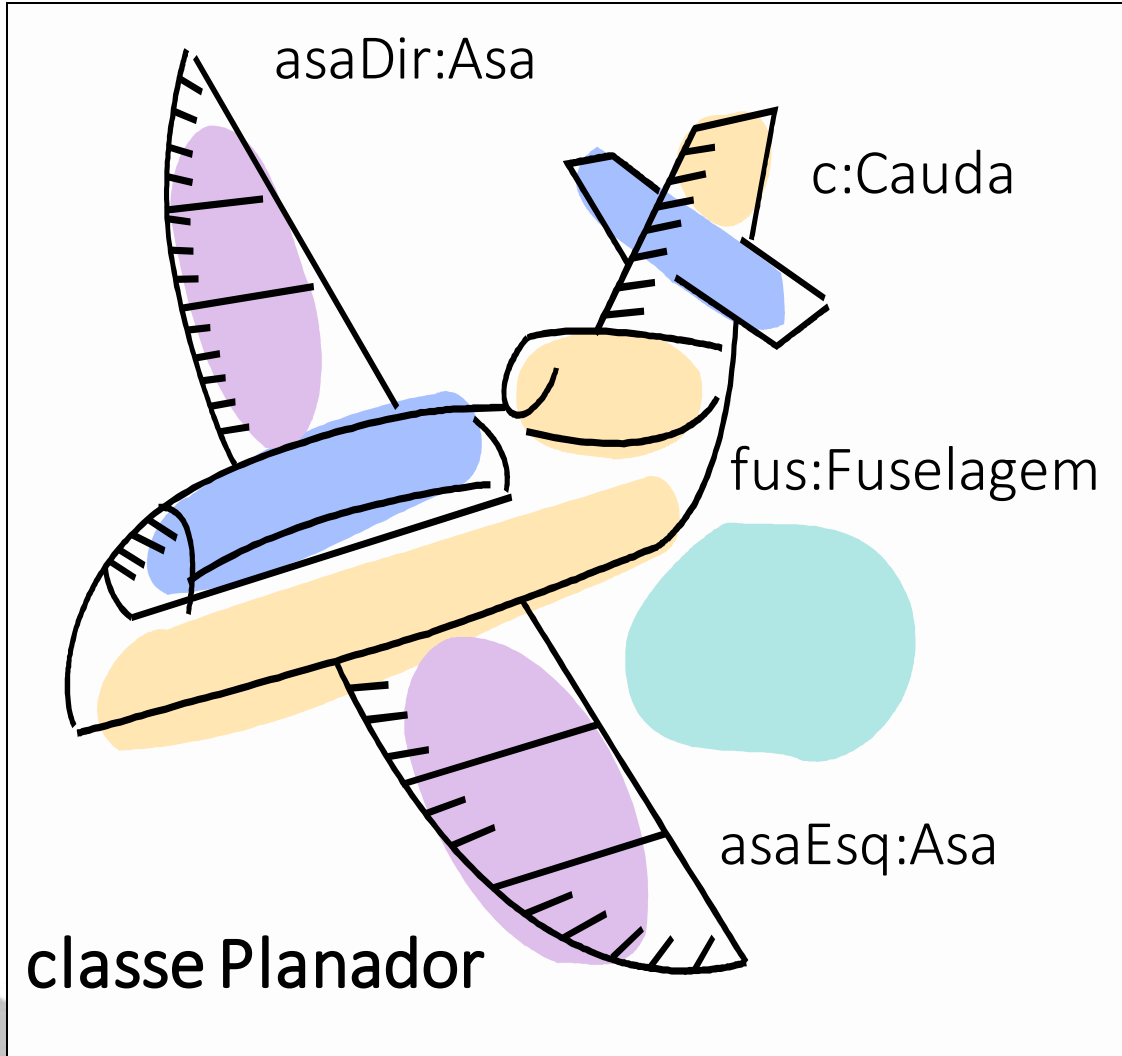


classe Escola



Os objetos componentes poderiam existir independentemente da classe composta e até ser componentes de várias classes compostas.

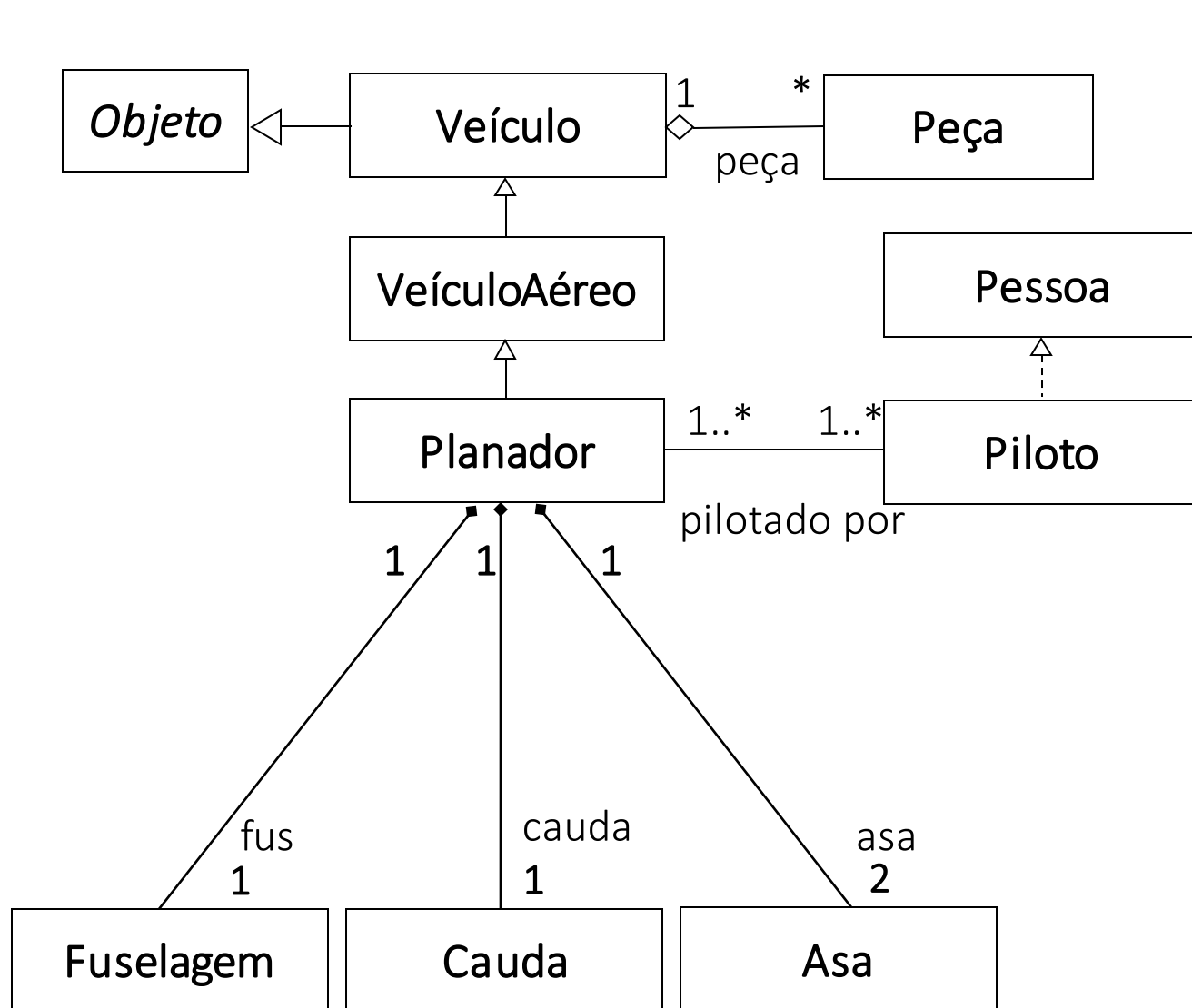
# Na composição



Os objetos componentes não poderiam existir independentes da classe composta, nem ser componentes de várias classes compostas simultaneamente.



# Diagrama de classes UML com vários tipos de relacionamentos



◆ representa **composição**

◊ representa **agregação**

Entre as classes Planador e Piloto existe um relacionamento de *associação*.

↑ significa **herança**.

↑ significa **herança por implementação**

(interface)

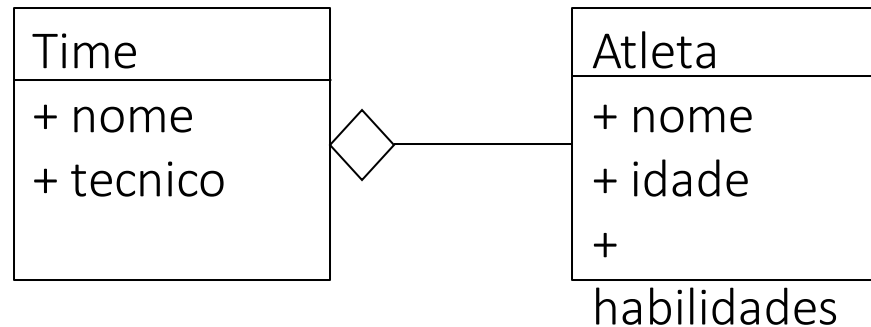
Classes abstratas, *Objeto*, em itálica



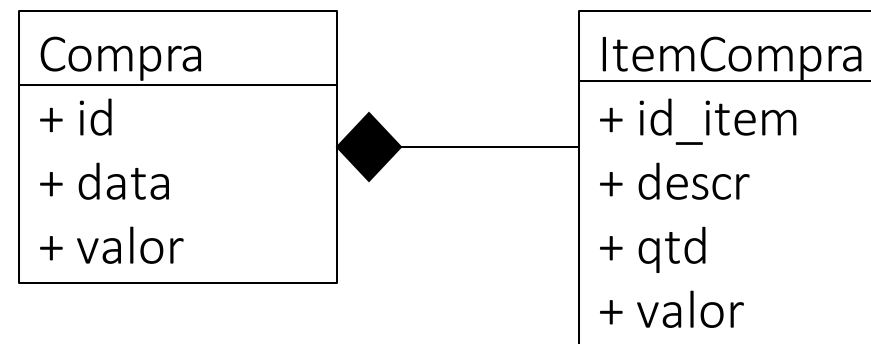
# Resumindo Agregação e Composição

São conceitos de associação entre classes, demonstram a relação do todo ser formado por partes, essa relação pode ser forte ou fraca.

- ✓ **Agregação**, a existência do objeto parte faz sentido, mesmo não existindo o objeto todo.



- ✓ **Composição** é uma agregação mais forte, ou seja, a existência do objeto parte **não** faz sentido se o objeto todo não existir.



# Resumindo Agregação e Composição

Esses conceitos de associação podem ser implementados em Java como:  
Atributos do tipo referência (pode ser usado pela agregação e composição).

- ✔ Classes aninhadas (mais comum ser usado na composição).

```
public class Pessoa {  
    Endereco end; ←  
    ...  
    void Pessoa(Endereco endereco)  
    {  
        end = endereco;  
        ...  
    }  
}
```

Referência

```
public class Endereco {  
    String rua;  
    String cep;  
    ...  
}
```

Classes aninhadas

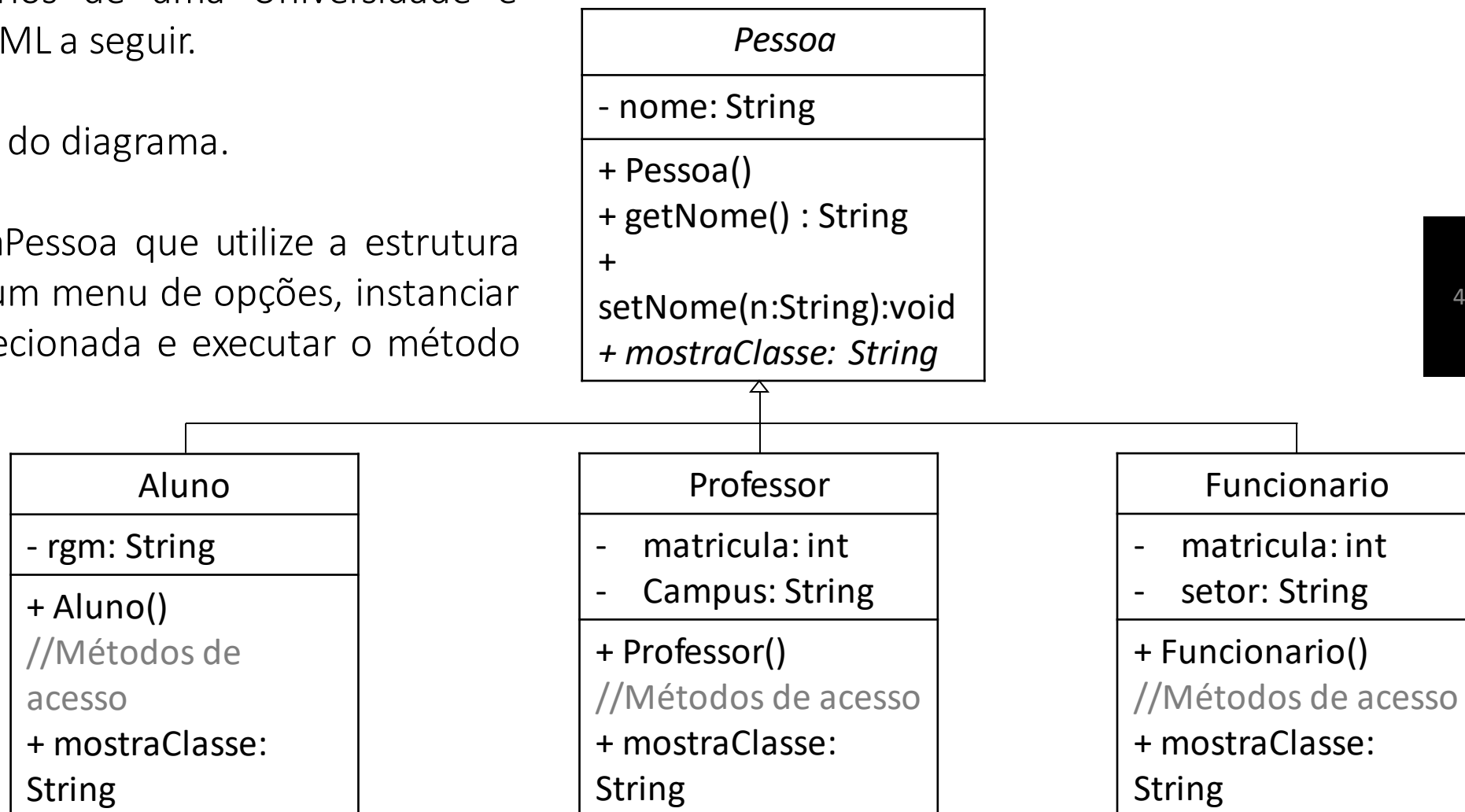
```
public class Empresa {  
    Departamento depto[];  
    ...  
    ...  
    public class Departamento {...}  
        ...  
    }  
}
```

# Exercícios

O quadro de funcionários de uma Universidade é mostrado no diagrama UML a seguir.

a) Implemente as classes do diagrama.

b) Crie uma classe TestaPessoa que utilize a estrutura switch para, a partir de um menu de opções, instanciar um objeto da classe selecionada e executar o método mostraClasse().



# Exercícios

```
public static void main(String[] args) {
    Pessoa p = null;

    while (true) {
        int tipo = Integer.parseInt(JOptionPane.showInputDialog(null, "Digite uma opção:"
            + "\n1- Aluno"
            + "\n2- Professor"
            + "\n3- Funcionário"
            + "\n4- Sair"));

        switch (tipo) {
            case 1:
                p = new Aluno();
                break;
            case 2:
                p = new Professor();
                break;
            case 3:
                p = new Funcionario();
                break;
            case 4:
                System.out.println("Bye... bye");
                System.exit(0);
                break;
            default: {
                System.out.println("Opção inválida");
                System.exit(0);
            }
        }

        p.mostraClasse();
    }
}
```


O método mostraClasse()  
exibe o nome da classe  
selecionada!!!

Deverá chamar o método mostraClasse()  
de acordo com o tipo de objeto  
instanciado



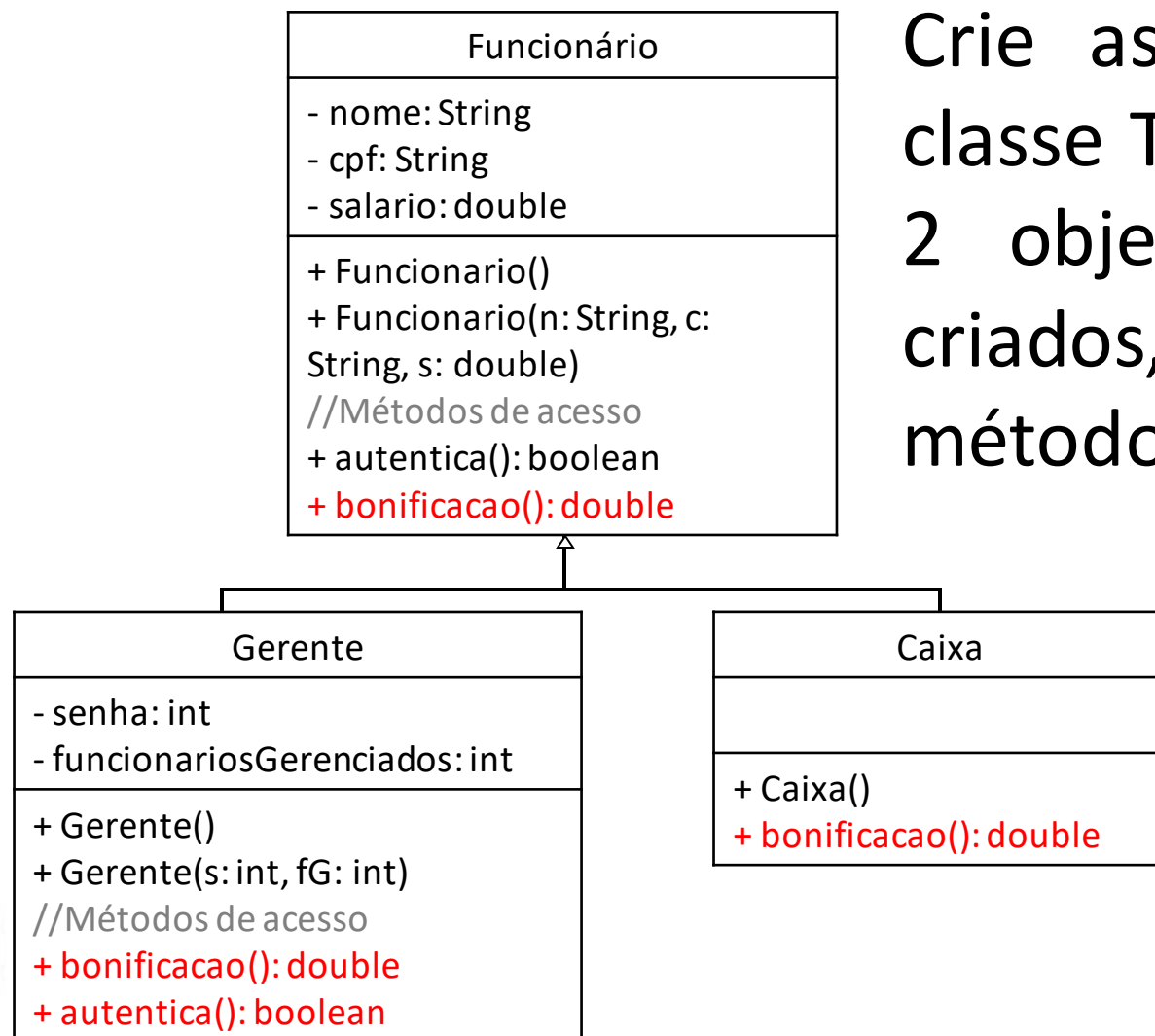
# Exercício

Como toda empresa, um Banco possui funcionários, que podem ser gerentes, caixas, entre outros.

- ✔ Considere que o gerente do banco hipotético possui, além das informações comuns a todos os funcionários, uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia.
  - ✔ Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.
- 

# Exercício

Crie as classes e depois, em uma classe TestFuncionario (principal), crie 2 objetos usando os construtores criados, também utilize os outros métodos.



# Vetores e Matrizes (Array)

# Vetores ou Arrays

- Definição
  - Conjunto de variáveis de mesmo tipo de dado.
- Exemplos:
  - Média de alunos;
  - Altura de atletas;
  - Idades de eleitores, etc.



# Vetores ou Arrays

- Declaração de um vetor:

Java:

```
double media [ ] = new double[6];
```



media[0] media[1] media[2] media[3] media[4] media[5]

Vetor media

# Vetores ou Arrays

- Inicialização de um vetor:

**Java:**

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```



media[0] media[1] media[2] media[3] media[4] media[5]

Vetor media

# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```



Vetor media

# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```



Vetor media

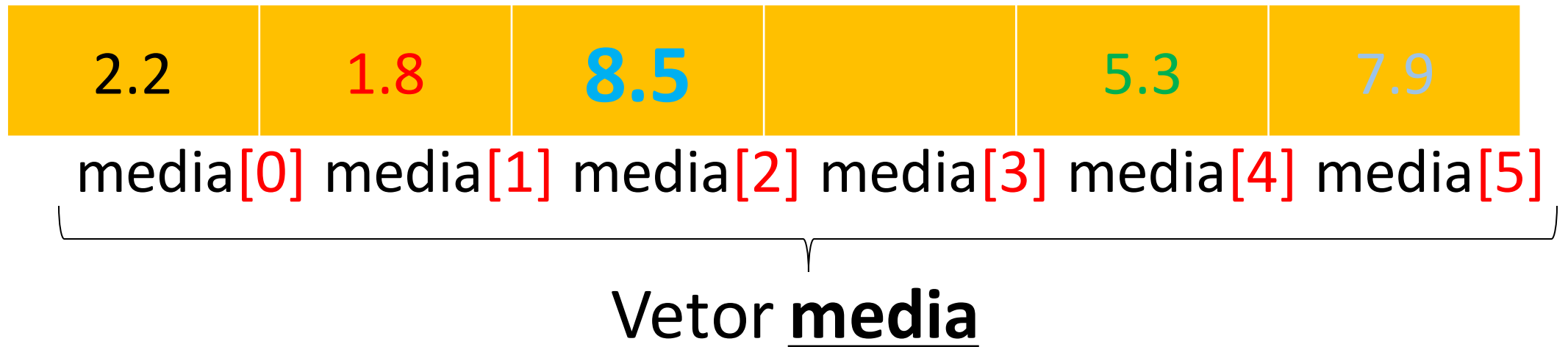
# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```

53



# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```



media[0] media[1] media[2] media[3] media[4] media[5]

Vetor media

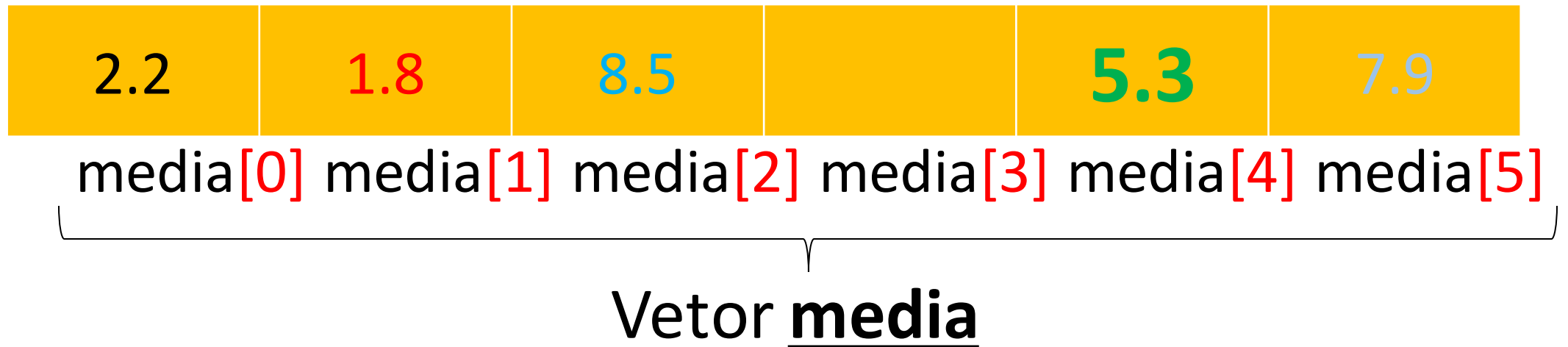
# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```

55



# Vetores ou Arrays

- Inicialização de um vetor:

Java:

```
double [ ] media = {2.2, 1.8, 8.5, 0.0, 5.3, 7.9};
```

56

2.2

1.8

8.5

5.3

7.9

media[0] media[1] media[2] media[3] media[4] media[5]

Vetor media



```

1 package introducao_jsp.exemplo_vetor;
2
3 public class ExemploVetor {
4     public static void main(String[] args){
5         final int TAMANHO = 30;
6         int vetorNumeros[] = new int[TAMANHO];
7
8         for(int indice = 0; indice < TAMANHO; indice++){
9             vetorNumeros[indice] = indice + 10;
10
11             if (vetorNumeros[indice]%2 ==0){
12                 switch(vetorNumeros[indice]){
13                     case 20:
14                         System.out.print("Vinte");
15                         break;
16                     case 30:
17                         System.out.print("Trinta");
18                         break;
19                     default:
20                         System.out.print(vetorNumeros[indice] + " ");
21                         break;
22                 }
23             }
24         }
25     }
26 }

```

A decorative graphic in the top-left corner consisting of a cluster of hexagons in various colors including yellow, orange, red, and grey.

# Vetores ou Arrays

## ArrayList

- Pertence à classe *java.util.ArrayList*.



# Vetores ou Arrays

## ArrayList

- Pertence à classe *java.util.ArrayList*.
- O *array* pode ter tamanho variável.

# Vetores ou Arrays

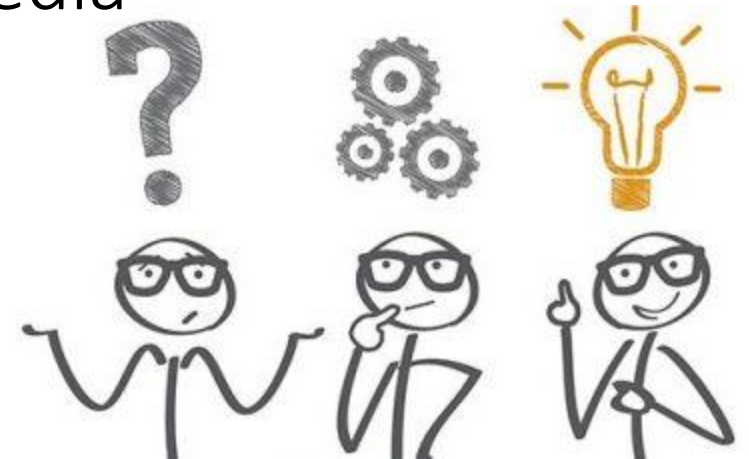
## ArrayList

- ▶ Pertence à classe *java.util.ArrayList*.
- ▶ O *array* pode ter tamanho variável.
- ▶ Possui métodos muito úteis:
  - *add(valor OU objeto); add(índice, valor OU objeto); size(); get(índice); remove(índice); set(índice, valor OU objeto); clear();* dentre outros.

# O problema!!

Faça um programa que escreva “Parabéns!” nas melhores provas de uma disciplina com 3 alunos. O programa deve:

- ✓ Ler os nomes e as notas de 3 alunos
- ✓ Calcular a média da turma
- ✓ Listar os alunos tiveram nota acima da média



# O problema!!

1

```
public class Exemplo1 {  
    public static void main(String[] args) {  
        float nota1, nota2, nota3, media;  
        String nome1, nome2, nome3;  
  
        nome1 = JOptionPane.showInputDialog(null, "Digite o nome do primeiro aluno");  
        nota1 = Float.parseFloat(JOptionPane.showInputDialog(null,  
                                                             "Digite a nota do aluno " + nome1));  
        nome2 = JOptionPane.showInputDialog(null, "Digite o nome do segundo aluno");  
        nota2 = Float.parseFloat(JOptionPane.showInputDialog(null,  
                                                             "Digite a nota do aluno " + nome2));  
        nome3 = JOptionPane.showInputDialog(null, "Digite o nome do terceiro aluno");  
        nota3 = Float.parseFloat(JOptionPane.showInputDialog(null,  
                                                             "Digite a nota do aluno " + nome3));  
        media = (nota1 + nota2 + nota3) / 3;  
        System.out.println("A média da turma foi: " + media);  
        if (nota1 > media)  
            System.out.println("Parabéns " + nome1);  
        if (nota2 > media)  
            System.out.println("Parabéns " + nome2);  
        if (nota3 > media)  
            System.out.println("Parabéns " + nome3);  
    }  
}
```

# O problema!!

```

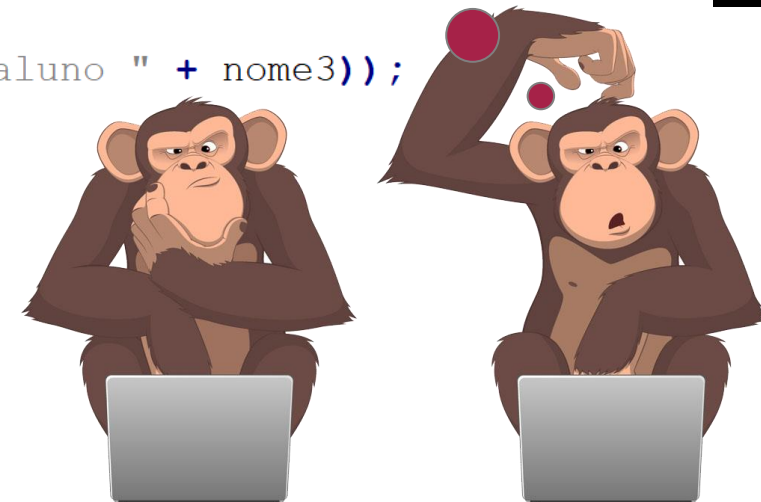
1 public class Exemplo1 {
    public static void main(String[] args) {
        float nota1, nota2, nota3, media;
        String nome1, nome2, nome3;

        nome1 = JOptionPane.showInputDialog(null, "Digite o nome do primeiro aluno");
        nota1 = Float.parseFloat(JOptionPane.showInputDialog(null, "Digite a nota do primeiro aluno " + nome1));
        nome2 = JOptionPane.showInputDialog(null, "Digite o nome do segundo aluno");
        nota2 = Float.parseFloat(JOptionPane.showInputDialog(null, "Digite a nota do segundo aluno " + nome2));
        nome3 = JOptionPane.showInputDialog(null, "Digite o nome do terceiro aluno");
        nota3 = Float.parseFloat(JOptionPane.showInputDialog(null, "Digite a nota do terceiro aluno " + nome3));

        media = (nota1 + nota2 + nota3) / 3;
        System.out.println("A média da turma foi: " + media);
        if (nota1 > media)
            System.out.println("Parabéns " + nome1);
        if (nota2 > media)
            System.out.println("Parabéns " + nome2);
        if (nota3 > media)
            System.out.println("Parabéns " + nome3);
    }
}

```

E se fossem  
40  
alunos????



# Estruturas de dados

Uma variável é capaz de armazenar apenas um valor de cada vez. Existem situações em que há necessidade de armazenar uma grande quantidade de valores e para isso não iremos declarar várias variáveis.

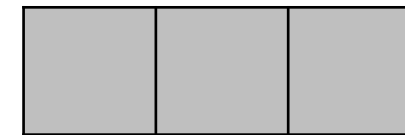
Para resolver esse problema construímos novos tipos que têm um formato denominado **estrutura de dados**, que define como os tipos primitivos estão organizados.



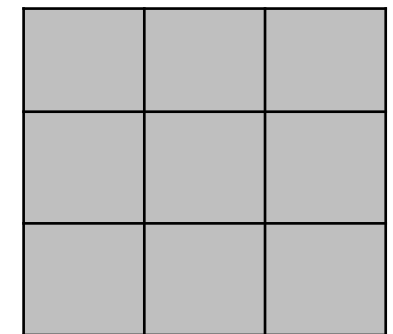


# Arrays

- ✓ Um **array** é um conjunto de elementos ou itens que respondem a um mesmo nome e que podem ser acessados segundo a posição (índice) que ocupam dentro do arranjo.
- ✓ Cada **elemento** ou item de um array está em determinada **posição** e armazena um **valor**, que poderá ser float, int, boolean, char, String, entre outros.
- ✓ Os arrays podem ser:
  - **Unidimensionais:** conhecidos como **vetores**, possuem somente um índice.
  - **Bidimensionais:** conhecidos como **matrizes**, possuem dois ou mais índices.



**vetor**



**matriz**

# Arrays unidimensionais: vetores

- ✓ São utilizados para armazenar um conjunto de dados cujos elementos podem ser endereçados por um único índice.
- ✓ Os **vetores** são coleções de objetos ou tipos de dados primitivos, que têm como características:
  - Tamanho Fixo: Vetores não podem ser redimensionados após sua construção, seria necessário criar um novo vetor e copiar os valores do antigo.
  - Verificados em tempo de execução: uma tentativa de acessar índices inexistentes provoca, na execução, um erro do tipo ArrayIndexOutOfBoundsException.
  - Tipo Definido: deve-se restringir o tipo dos elementos que podem ser armazenados.

# Arrays unidimensionais: vetores

O acesso às posições é feito colocando o nome de identificação e o número da posição (índice);

A posição do primeiro elemento de um vetor é sempre 0 (zero);

O exemplo abaixo ilustra um vetor de nome "c", com nove elementos:

Posição do elemento no vetor						Nome do vetor		
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]
-45	6	0	72	1543	-89	0	62	-3

Conteúdo de uma posição (um elemento) do vetor

# Declaração de vetores

Não podemos utilizar um vetor antes da sua declaração **E** inicialização:

- ✓ Declaração de um vetor:

**tipo[]** nomeVar; ou **tipo** nomeVar[]; //tanto faz

- ✓ Inicialização de um vetor:

nomeVar = **new mesmotipo**[quantidade];

- ✓ Pode-se criar diretamente, na mesma linha:

**tipo[]** nomeVar = **new mesmotipo**[quantidade];

- ✓ Ou ainda, podemos criar um vetor já com elementos, utilizando chaves:

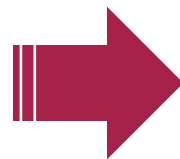
**int[]** nomeVar = {1, 2, 3, 4, 5};

**String[]** outraVar = { "Ana Paula", "Cristiane", "Leonardo", "Alcides" };

# Exemplos

```
//Declaração dos vetores
String[] vetor1, vetor2;
//Declaração dos tamanhos
vetor1 = new String[2];
vetor2 = new String[5];
//Declaração completa
float[] vetor3 = new float[3];
//Preenchimento do vetor1
vetor1[0] = "Juliana";
vetor1[1] = "Pedro";
//Declaração com atribuição de valores
int[] = vetor4 = {1, 2, 3, 4}
```

# Exemplo: Algoritmo imprime-inverso



algoritmo imprime-inverso

**inicio**

```
    inteiro i, vetor[10]
    para(i = 0; i < 10; i++){
        escreva("Entre o " + (i+1)
                + "º número")
        leia(vetor[i])
    }
```

```
    para(i = 0; i >= 0; i++){
        escreva(vetor[i])
    }
```

**fim**

```
int i;
```

```
int[] vetor = new int[10];
```

```
for (i=0; i<10; i=i+1){
    vetor[i] = Integer.parseInt(
        JOptionPane.showInputDialog(
            "Entre com o " + (i+1) + "º num"));
}
for (i=9; i>=0; i=i-1){
    System.out.println(vetor[i]);
}
```


# A propriedade length

- ✓ Todo vetor em Java possui esta propriedade, que informa o número de elementos que possui.
- ✓ Esta função é extremamente útil quando não se sabe o tamanho do vetor, como por exemplo na entrada de parâmetros da função **main**:

```
public static void main(String[] args){  
  
    //Quantos parâmetros nós temos?  
    for (int i=0; i<args.length; i++){  
        System.out.println(args[i]);  
    }  
}
```

# A propriedade length: exemplo

```
public class LengthDemo {  
  
    public static void main(String[] args) {  
        int[] lista = new int[10];  
        int[] numeros = { 1, 2, 3 };  
        int[][] tabela = { //tabela de tamanho variável  
            {1, 2, 3}, {4, 5}, {6, 7, 8, 9}};  
  
        System.out.println("O tamanho da lista é: " + lista.length);  
        System.out.println("O tamanho de numeros é: " + numeros.length);  
        System.out.println("O tamanho da tabela é: " + tabela.length);  
        System.out.println("O tamanho da tabela [0] é: " + tabela[0].length);  
        System.out.println("O tamanho da tabela [1] é: " + tabela[1].length);  
        System.out.println("O tamanho da tabela [2] é: " + tabela[2].length);  
    }  
}
```



```
Saída - LengthDemo (run) x  
run:  
O tamanho da lista é: 10  
O tamanho de numeros é: 3  
O tamanho da tabela é: 3  
O tamanho da tabela [0] é: 3  
O tamanho da tabela [1] é: 2  
O tamanho da tabela [2] é: 4
```



# Retomando...

- 1 Faça um programa que escreva “Parabéns!” nas melhores notas de uma disciplina com 3 alunos. O programa deve:
  - ✓ Ler os nomes e as notas de 3 alunos
  - ✓ Calcular a média da turma
  - ✓ Listar os alunos tiveram nota acima da média

E se fossem  
40  
alunos?????



# Retomando...

1

```
public class Exemplo1 {  
    public static void main(String[] args) {  
        float[] nota = new float[40];  
        String[] nome = new String[40];  
        float soma = 0, media;  
  
        for(int i=0; i < 40; i++){  
            nome[i] = JOptionPane.showInputDialog(null, "Digite o nome do " + (i+1)  
                                                + "° aluno");  
            nota[i] = Float.parseFloat(JOptionPane.showInputDialog(null,  
                                                                    "Digite a nota do aluno " + nome[i]));  
            soma += nota[i]  
        }  
        media = soma / 40;  
        for(int i=0; i < 40; i++){  
            if(nota[i] > media)  
                System.out.println("Parabéns " + nome[i]);  
        }  
    }  
}
```

# Copiando Vetores

Como fazer para copiar um vetor?

- ✔ Criamos um outro vetor, do mesmo tipo e mesmo tamanho:

```
int[] vetor2 = new int[vetor1.length];
```

- ✔ Fazemos um looping para todos os índices, e copiamos um a um os valores do original na cópia:

```
for (int i=0; i<vetor1.length; i++) {  
    vetor2[i] = vetor1[i];  
}
```

# Matrizes

- ✔ Seguindo o conceito de vetores, matrizes também têm uma declaração e uma inicialização (um pouco mais complexa).
- ✔ Quando inicializada, uma matriz também tem seus elementos inicializados com valores padrão.

Posição do livro

	0	1	2	...	n-1
0	788	598	265	...	156
1	145	258	369	...	196
2	989	565	345	...	526
⋮	⋮	⋮	⋮	⋮	⋮
m-1	845	153	564	892	210

Prateleira

# Matrizes

- ✓ A diferença na declaração é que utilizamos mais um par de colchetes:

```
int[][] matriz; ou int matriz[][];
```

- ✓ Na inicialização, podemos colocar as duas dimensões:

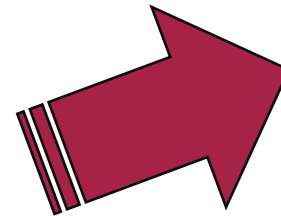
```
matriz = new int[2][3];
```

- ✓ Mas, da mesma forma que vetores, também temos “elementos” de matrizes para já inicializá-la com os valores pré-definidos, como se fosse um “**vetor de vetores**”:

```
int[][] matriz = {{1,2,3},{4,5,6}};
```

# Matrizes: exemplo

```
public static void main(String[] args) {  
    //Declaração dos vetores  
    String[][] nomeCompleto;  
  
    //Declaração do tamanho  
    nomeCompleto = new String[10][2];  
  
    //Preenchimento do vetor  
    nomeCompleto[0][0] = "Leonardo";  
    nomeCompleto[0][1] = "Carlos";  
    nomeCompleto[1][0] = "Ana";  
    nomeCompleto[1][1] = "Paula";  
}
```



	0	1
0	"Leonardo"	"Carlos"
1	"Ana"	"Paula"
2	null	null
3	null	null
4	null	null
5	null	null

# Matrizes: exemplo

algoritmo matriz

inicio

inteiro matriz[6][6], i, j

para(i = 0; i < 6; i++){

para(j = 0; j < 6; j++){

escreva("digite os numeros")

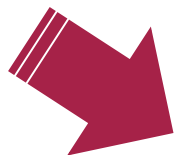
leia matriz[i][j]

escreva matriz[i][j]

}

}

fim



```
public static void main(String[] args) {  
    int[][] matriz = new int[6][6];  
    int i, j;  
    for (i=0; i<6; i++){  
        for (j=0; j<6; j++){  
            matriz[i][j] = Integer.parseInt(  
                JOptionPane.showInputDialog("Entre com o número"));  
            System.out.println(matriz[i][j]);  
        }  
    }  
}
```



# Matrizes: características em Java

Como na verdade matrizes são "**vetores de vetores**", nada impede de cada linha possuir um vetor de tamanho diferente, se inicializarmos uma por uma separadamente:

```
int[][] matriz;
matriz = new int[3][]; //Inicializando a quantidade de linhas
matriz[0] = new int[3]; //Inicializando a primeira linha
matriz[1] = new int[5]; //Inicializando a segunda linha
matriz[2] = new int[2]; //Inicializando a terceira linha
```

O problema agora é como controlar os loopings i e j. Mas podemos utilizar o atributo length tanto para linhas como para as colunas:

```
for (int i=0; i<matriz.length; i++){
    for (int j=0; j<matriz[i].length; j++){
        System.out.println(matriz[i][j]);
    }
}
```

Matriz

0	0	0	0		
1	0	0	0	0	0
2	0	0			



# Matrizes: exemplo

2

Login

- String nome
- String acesso
- String password
- String[][] users

Login();

Login(nome: String, password:  
String)

//Métodos de acesso

autentica(): boolean

# Matrizes: exemplo

2

```
public class Login {
    private String nome;
    private String acesso;
    private String password;

    private String[][] users = { {"marco", "123", "admin"},
                                   {"luiza", "123", "enfermeira"},
                                   {"luka", "123", "médico"}};

    public Login() {}
    public Login(String nome, String password) {
        this.nome = nome;
        this.password = password;
    }
    public String getNome() { return nome; }
    public String getAcesso() { return acesso; }
    public boolean autentica() {
        for (int i = 0; i < users.length; i++) {
            if (users[i][0].equals(nome) && users[i][1].equals(password)) {
                acesso = users[i][2];
                return true;
            }
        }
        return false;
    }
}
```

# Matrizes: exemplo

2

```
public class TestaLogin {  
  
    public static void main(String[] args) {  
        Login user;  
        String nome, password;  
  
        nome = JOptionPane.showInputDialog(null, "Digite o nome de usuário");  
        password = JOptionPane.showInputDialog(null, "Digite a senha");  
  
        user = new Login(nome, password);  
        if (user.autentica()) {  
            System.out.println("Bem-vindo " + user.getNome()  
                               + "\nSeus privilégios de acesso são: " + user.getAcesso());  
        } else {  
            System.out.println("Dados incorretos");  
        }  
    }  
}
```

# Vetores de Objetos

- Assim como podemos criar Vetores e Matrizes (Arrays) de tipos primitivos, podemos também criar de **objetos**.
- Seja uma classe de nome “Aluno”, podemos construir:

- Pegamos como exemplo uma classe Aluno:

```
Aluno vetor[] = new Aluno[4];
```

Aluno
<ul style="list-style-type: none"> <li>- String rgm;</li> <li>- String nome;</li> <li>- int idade;</li> </ul>
Aluno(); + setRgm(String rgm); ... + void print();

# Vetores de objetos

Quando se deseja criar vetores de objetos teremos três as etapas necessárias. As duas primeiras etapas podem ser simultâneas ou não:

**1- Declaração do vetor de objetos:** nesta etapa o nome do vetor será associado a seu tipo, ou seja, a sua Classe.

```
Aluno vetor[];
```

**2- Criação do vetor de objetos na memória:** cada elemento apontará para um objeto.

```
vetor = new Aluno[4];
```

# Vetores de objetos

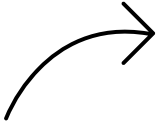
Quando se deseja criar vetores de objetos teremos três as etapas necessárias. As duas primeiras etapas podem ser simultâneas ou não:

**3 - Criação do objeto em cada posição do vetor:** Lembre-se que sempre que um objeto é criado na memória o que é armazenado sobre o mesmo é o seu endereço. No caso dos vetores de objetos, cada elemento guardará uma referência para um objeto na memória. Assim, para cada elemento do vetor (cada elemento fará referência a um objeto), será necessário criar o objeto na memória através de seu construtor.

```
for(int i=0; i< vetor.length; i++)  
    vetor[i] = new Aluno();
```

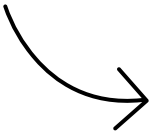
# Vetores de objetos: memória

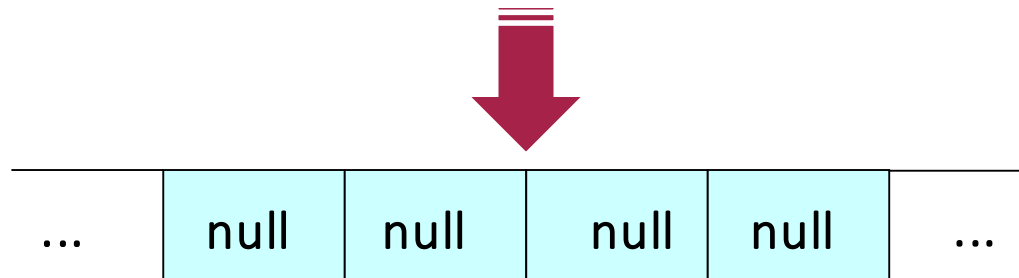
## ✓ Etapa 1

`Aluno vetor[];`  O nome vetor é associado à classe Aluno;

## ✓ Etapa 2

`vetor = new Aluno[4];`

 É criado um vetor de 4 posições na memória onde cada elemento apontará para um objeto, neste momento todos os elementos conterão *null*.

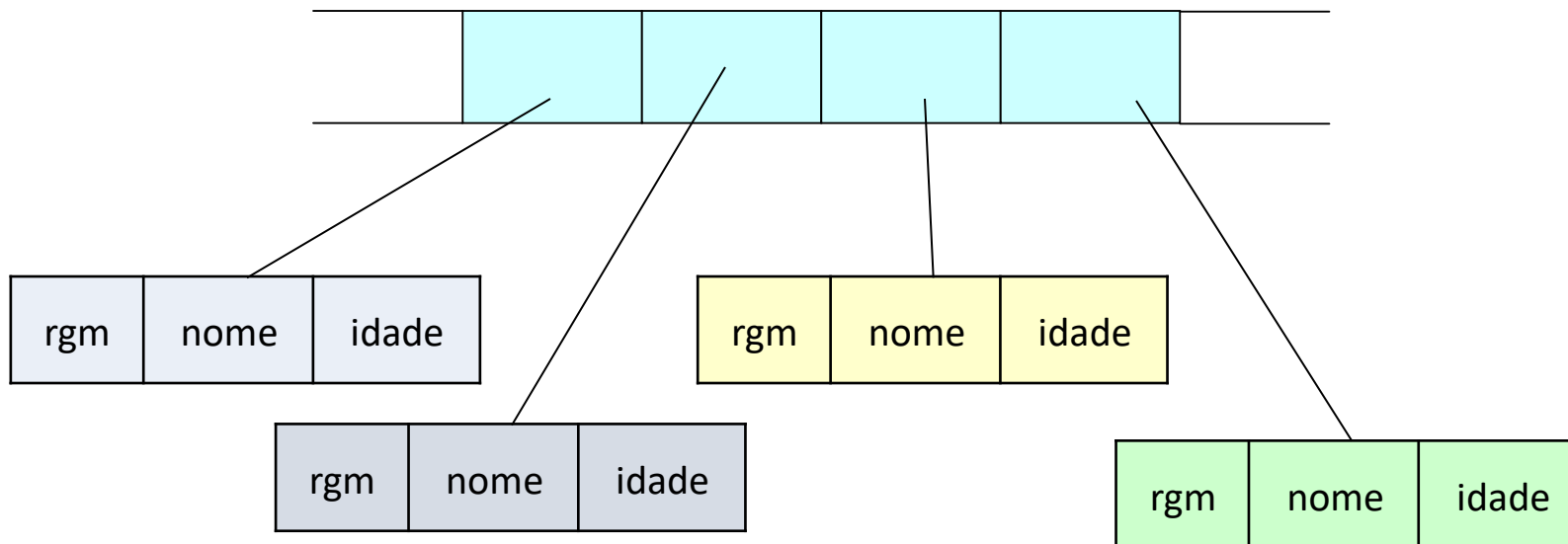


# Vetores de objetos: memória

## ✓ Etapa 3

```
for(int i=0; i< vetor.length; i++)  
    vetor[i] = new Aluno();
```

→ A cada iteração do laço um objeto é criado na memória e sua referência armazenada no vetor.





# Vetores de objetos: exemplos

3

Aluno

- String rgm;
- float notas[];
- float media;

Aluno();

Aluno(rgm: String, qtde: int)

Aluno(rqtde: int)

//Métodos de acesso

leitura(): void

calculaMedia(): float

print(): void

# Vetores de objetos: exemplos

3

```
public class Aluno {  
    private String rgm;  
    private float[] notas;//declaração do vetor de notas  
    private float media;  
  
    public Aluno() {}  
  
    public Aluno(String rgm, int qtde) {  
        this.rgm = rgm;  
        this.notas = new float[qtde];//instancia o vetor de notas  
    }  
  
    public Aluno(int qtde){  
        this.notas = new float[qtde];//instancia o vetor de notas  
    }  
  
    public String getRgm() {return rgm;}  
    public void setRgm(String rgm) {this.rgm = rgm;}  
    public float[] getNotas() {return notas;}  
    public void setNotas(float[] notas) { this.notas = notas; }
```

# Vetores de objetos: exemplos

3

```
public void leitura() {
    rgm = JOptionPane.showInputDialog(null, "Digite o RGM");
    for(int i=0; i < notas.length; i++){
        notas[i] = Float.parseFloat(JOptionPane.showInputDialog(null, "Digite a nota " + (i+1)));
    }
}

public float calculaMedia() {
    float soma = 0;
    for(int i = 0; i < notas.length; i++){
        soma += notas[i];
    }
    media = soma/notas.length;
    return media;
}

public void print() {
    String saida;
    saida = "RGM: " + rgm + "\n";
    saida += "Notas: ";
    for(int i = 0; i < notas.length; i++){
        saida += "[" + notas[i] + " ]";
    }
    saida += "\n";
    saida += "Média: " + media + "\n";
    System.out.println("Dados do aluno: \n" + saida);
}
```

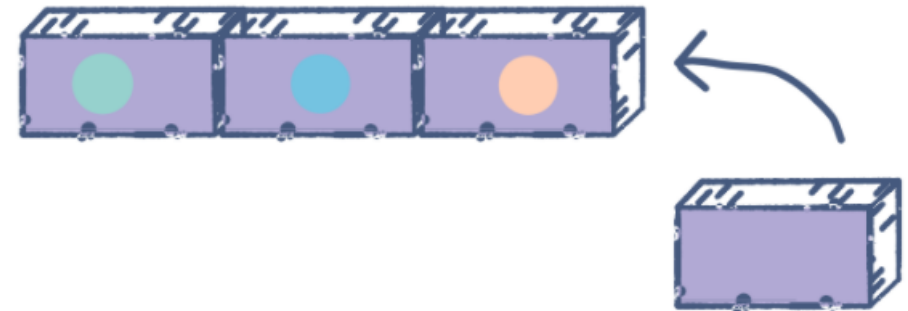
# Vetores de objetos: exemplos

3

```
public class TestaAluno {  
    public static void main(String[] args) {  
        int qtdeAlunos, qtdeNotas;  
        float mediaGeral = 0;  
        Aluno alunos[]; //declaração do vetor de objetos do tipo Aluno  
  
        qtdeAlunos = Integer.parseInt(JOptionPane.showInputDialog(null,  
                                                                    "Digite a quantidade de alunos da turma"));  
        alunos = new Aluno[qtdeAlunos]; //instancia o vetor de objetos do tipo Aluno  
  
        for(int i = 0; i < alunos.length; i++){  
            qtdeNotas = Integer.parseInt(JOptionPane.showInputDialog(null,  
                                                                    "Digite a quantidade de notas do aluno " + (i+1)));  
            //instancia o objeto na posição do vetor  
            //instancia para cada objeto o vetor de notas  
            alunos[i] = new Aluno(qtdeNotas);  
            alunos[i].leitura();  
            mediaGeral += alunos[i].calculaMedia();  
            alunos[i].print();  
        }  
  
        System.out.println("Media geral da turma: \n" + mediaGeral/qtdeAlunos);  
    }  
}
```

# Arrays dinâmicos

- ✓ Observe que, antes de usar, precisamos definir o tamanho do vetor criado.
- ✓ Para resolver esse problema, em Java temos a implementação de vetores dinâmicos cujo tamanho pode aumentar à medida da necessidade.
- ✓ Exemplos de classes são o Vector e **ArrayList**.



Adiciona uma nova caixa quando todas as caixas estão preenchidas



Remove uma caixa quando está vazia e libera espaço extra na memória

# Arrays dinâmicos: principais métodos

- ✓ **add():** Adiciona um objeto no final da lista;
- ✓ **get(pos):** Retorna o elemento da posição pos;
- ✓ **size():** retorna o tamanho da lista;
- ✓ **clear():** remove todos os elementos da lista;
- ✓ **remove(pos):** remove um elemento da lista da posição pos;

# Arrays dinâmicos: exemplo

```
public class TesteAlunoArray {  
    public static void main(String[] args) {  
        int qtdeNotas;  
        Aluno aluno;  
        ArrayList alunos = new ArrayList();  
        String op;  
  
        do{  
            qtdeNotas = Integer.parseInt(JOptionPane.showInputDialog(null,  
                                                                    "Digite a quantidade de notas do aluno: "));  
            aluno = new Aluno(qtdeNotas);  
            aluno.leitura();  
            aluno.calculaMedia();  
            alunos.add(aluno);  
            op = JOptionPane.showInputDialog(null, "Deseja continuar?");  
        }while(op.equalsIgnoreCase("s"));  
  
        for(int i=0; i<alunos.size(); i++){  
            aluno = (Aluno)alunos.get(i);  
            aluno.print();  
        }  
    }  
}
```


Adiciona um objeto no vetor  
dinâmico

Faz um *casting* para capturar de volta o  
objeto guardado



# Exercício

Desenvolva um programa na linguagem java, que receba a média da temperatura diária durante um período de 7 dias (armazene as informações em um vetor), calcule a média da temperatura desse período (semanal) e informe quantos dias a temperatura ficou acima da média e quantos ficou abaixo da média semanal.





“Coragem é ir de  
falha em falha sem  
perder o  
entusiasmo”



Winston Churchill

# Obrigado!

## Se precisar ...

Prof. Claudio Benossi

[Claudio.benossi@fatec.sp.gov.br](mailto:Claudio.benossi@fatec.sp.gov.br)

