



*Linguagem de Programação  
Orientada a Objetos*

Profa. Joyce Miranda

# Apresentação do Módulo

## ► Visão Geral

**Programação  
Estruturada**

**Módulos (Funções)**  
elementos ativos

**Dados**  
repositórios passivos

**Programação  
Orientada a Objetos**

**Classes**

---

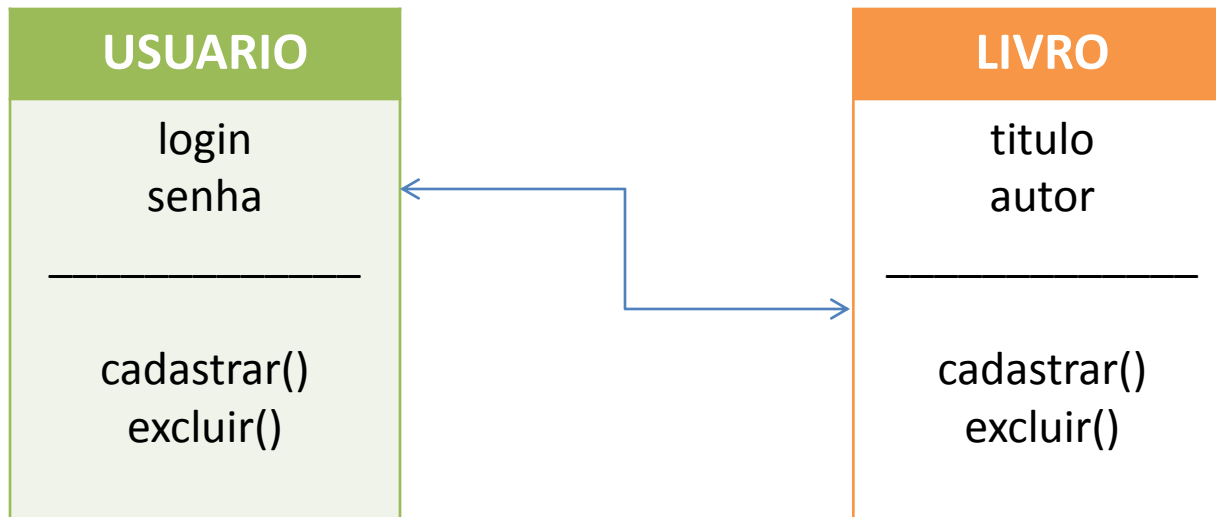
Atributos (Dados)

---

Métodos (Funções)

# Introdução

- ▶ Programação **Orientada a Objetos** (POO)
  - ▶ Entende o sistema como um conjunto de objetos, com características e comportamentos próprios, que interagem entre si.

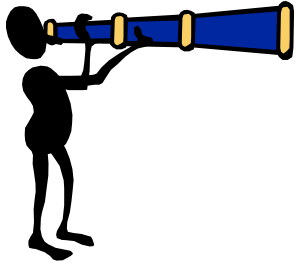


# Introdução

---

## ► Paradigma **Orientado a Objetos** (OO)

► Baseia-se na *abstração*.



INDIVÍDUO



REALIDADE



COR: VERDE  
QTDE RODAS: 4  
QTDE PASSAGEIROS: 1  
NUMERO: 1  
AÇÃO: CORRE, ACELERA

ESTRUTURA

# Conceitos - Objetos

---

- ▶ Um objeto é um conceito, uma abstração, algo com **limites** e **significados nítidos** em **relação ao domínio de uma aplicação**.



Domínio Acadêmico



Domínio Locadora



# Conceitos - Objetos

- Pode-se pensar sobre o mundo real como uma coleção de objetos relacionados



## Domínio Acadêmico

Professor Fulano
Professor Ciclano
Professor Beltrano

Curso Informática
Curso Edificações
Curso Turismo

Aluno Alfa
Aluno Beta
Aluno Gama

# Conceitos - Objetos



- Pode-se pensar sobre o mundo real como uma coleção de objetos relacionados



## Domínio Locadora

Filme X
Filme Y
Filme Z

Cliente Alfa
Cliente Beta
Cliente Gama

Locação 1
Locação 2
Locação 3



# Conceitos - Objetos

---

- ▶ Objeto é uma unidade dinâmica, composta por um **estado** interno privativo e um **comportamento**.
- ▶ **Estado**
  - ▶ Revela seus dados importantes de um objeto.
- ▶ **Comportamento**
  - ▶ São ações que um objeto pode exercer ou que podem ser exercidas a partir de um objeto.

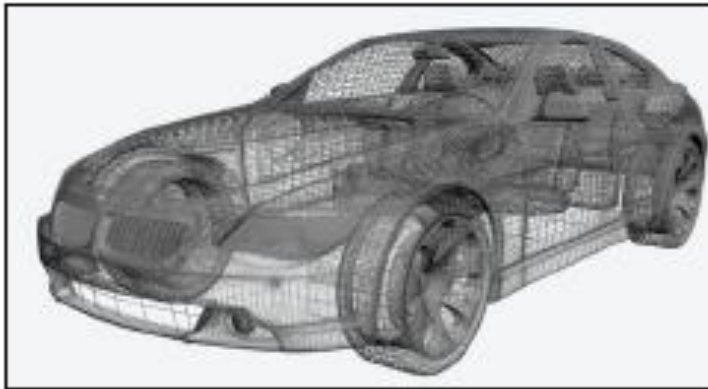




# Conceitos - Classes

---

- ▶ Objetos podem ser agrupados em classes
- ▶ Uma classe é um modelo que **define** os atributos e os métodos comuns a todos os objetos do mesmo tipo.



**Classe**



**Objeto**

# Conceitos - Classes

---

## ► Exemplos de classes por domínio



Domínio Acadêmico
Professor
Aluno
Curso
Disciplina



Domínio Locadora
Filme
Cliente
Locação
Estoque

# Conceitos - Classes

---

- ▶ Exemplos de classes por domínio



Domínio Bancário



# Conceitos - Classes

---

- ▶ Uma classe é a descrição de um grupo de objetos com ***propriedades semelhantes*** (atributos), ***mesmo comportamento*** (métodos), ***mesmos relacionamentos*** com outros objetos de outras classes.



# Conceitos - Classes

---

## ▶ Atributos


- ▶ São as características de um objeto. Basicamente a estrutura de dados que vai representar a classe. Exemplos:
  - ▶ Funcionário: nome, endereço, telefone, CPF;
  - ▶ Livro: autor, editora, ano;



# Conceitos - Classes

## ► Atributos

- Os valores dos atributos de um objeto definem seu Estado.

	<u>Maria</u>	<u>José</u>	<u>João</u>
• idade	31	28	43
• endereço	Rua xx	Rua yy	Av zz
• sexo	Fem	Masc	Masc
• etc			
			
	Maria	José	João



# Conceitos - Classes

---

## ▶ Métodos

- ▶ São as tarefas que o objeto pode realizar;

Condicionador de Ar
TemperaturaDesejada
AtribuirTemperaturaDesejada
LerTemperatura
AtivarRefrigeração
DesativarRefrigeração
ManterRefrigeração

# Conceitos - Classes

---

- ▶ Exemplos de classes com seus respectivos atributos por domínio



## Domínio Acadêmico

Professor
Nome
Titulação
Formação
Data da Contratação

Aluno
Nome
Matrícula
Ano de Ingresso
Situação

Curso
Nome
Sigla
Objetivo
Coordenador



# Conceitos - Classes

---

- ▶ Exemplos de classes com seus respectivos atributos por domínio



**Domínio Bancário**

Agência

Conta Corrente

# Conceitos - Classes

---

- ▶ Exemplos de classes com seus respectivos atributos por domínio



**Domínio Locadora**

Filme

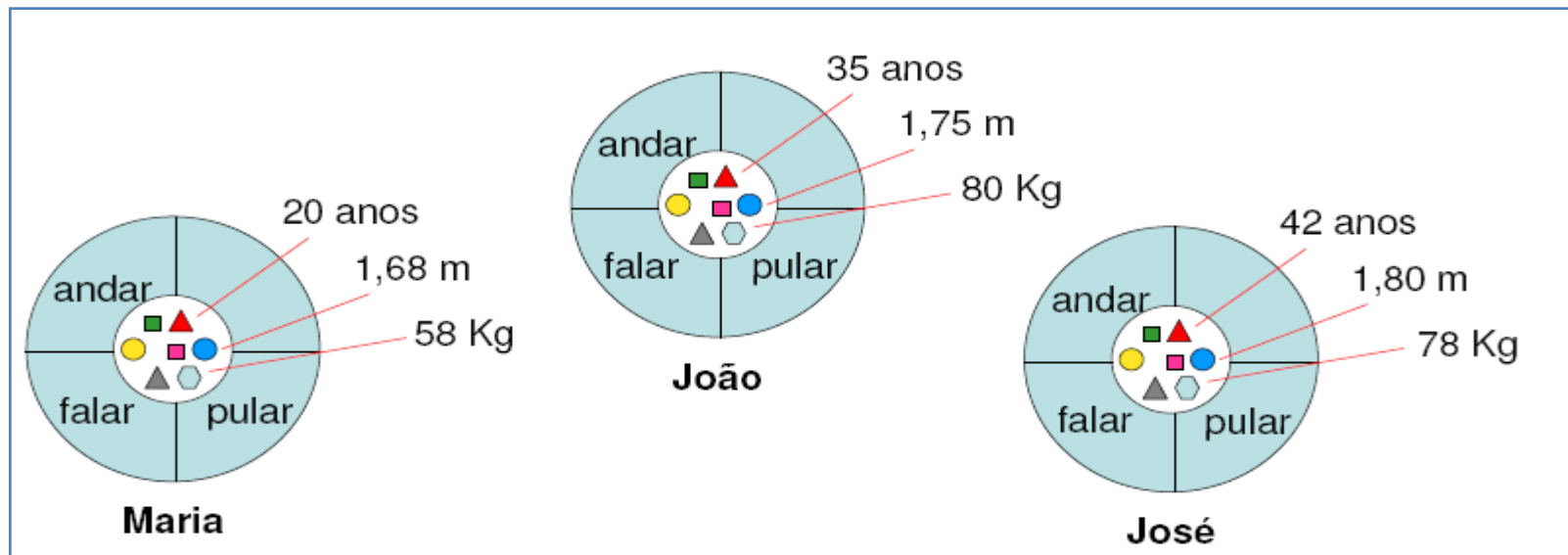
Locação



# Conceitos - Objetos

## ► Instância

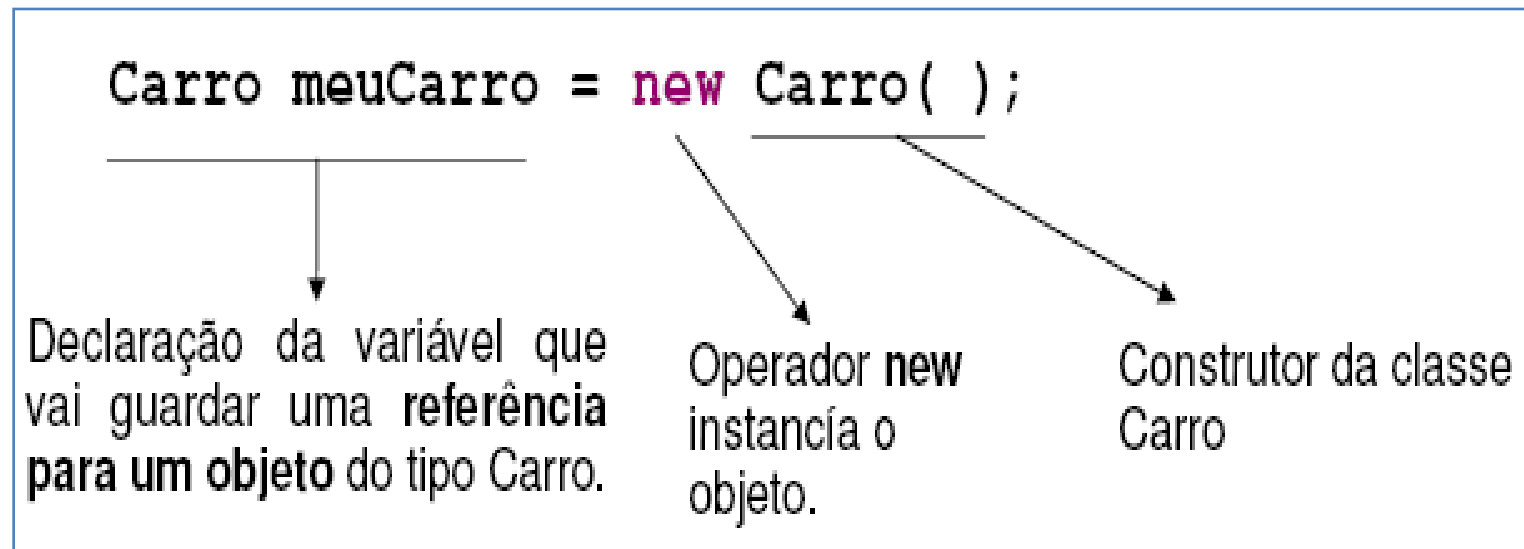
- Um sistema pode conter um ou mais objetos ativos.
- Cada objeto ativo no sistema em particular é chamado de **instância**.
- As diferentes instâncias possuem seu próprio estado.





# Objetos na Prática

- ▶ Um objeto, nada mais é do que uma instância de um tipo de dado específico (classe).

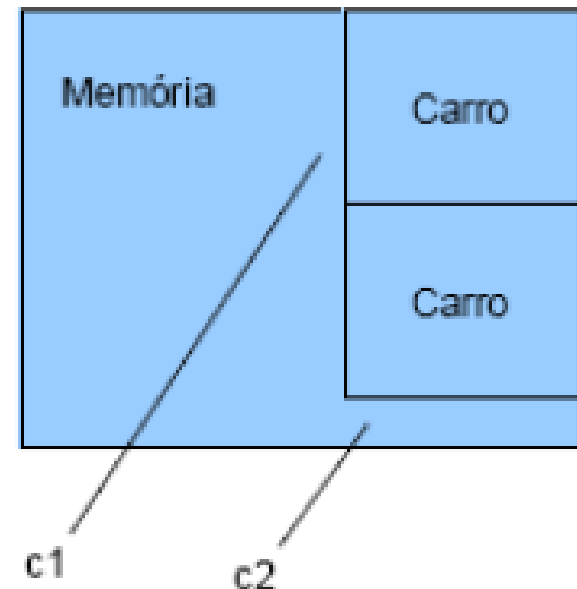




# Objetos na Prática

- ▶ As variáveis não guardam os objetos, mas sim uma **referência para a área de memória** onde os objetos estão alocados.

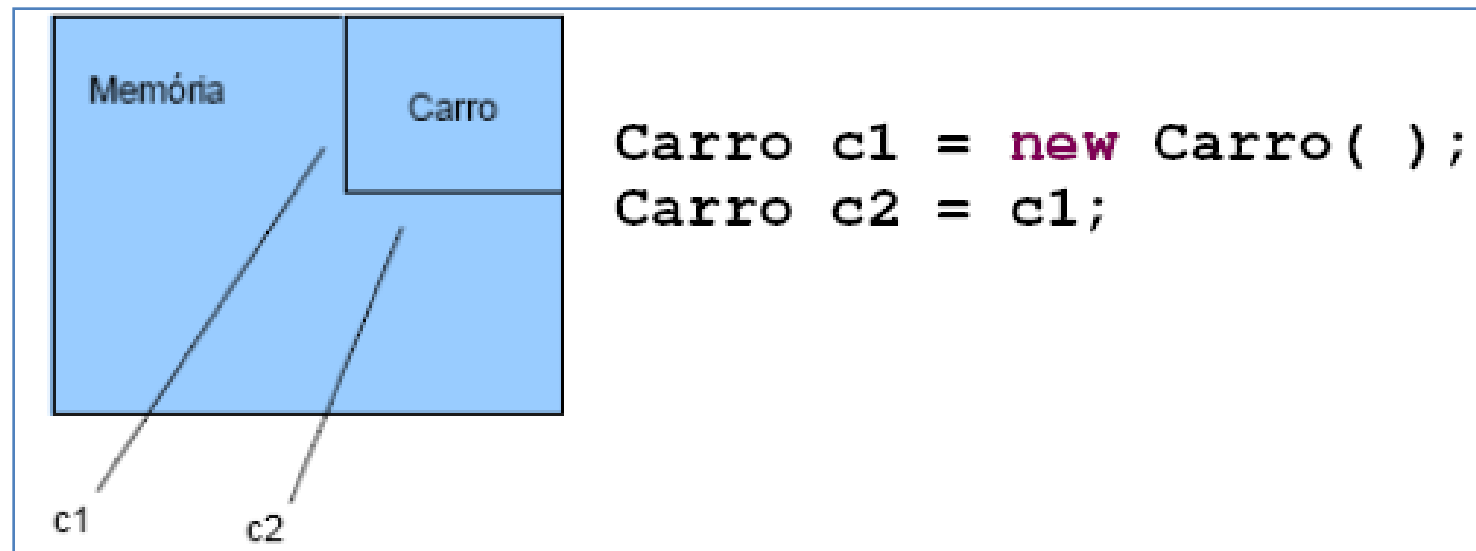
```
Carro c1 = new Carro( );  
Carro c2 = new Carro( );
```





# Objetos na Prática

- Imagine, agora, duas variáveis diferentes, *c1* e *c2*, *ambas* referenciando o mesmo objeto. Teríamos, agora, um cenário assim:



# Conceitos - Classes

---



- ▶ Considere um sistema para gerenciar um banco.



- ▶ Entidade Fundamental: CONTA

# Conceitos - Classes

---

- ▶ O que toda conta deve possuir?
  - ▶ Número da conta
  - ▶ Nome do titular da conta
  - ▶ Saldo
  - ▶ Limite

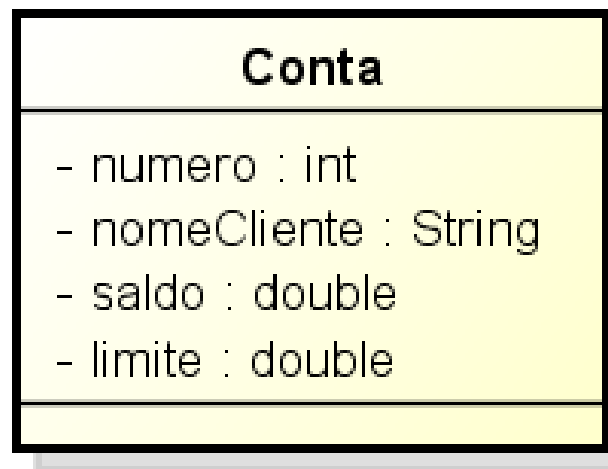




# Conceitos - Classes

---

- ▶ Projeto de Conta
  - ▶ Definição da Classe
  - ▶ Identificação dos Atributos



powered by astah\* 

# Conceitos - Classes

## ► Classes na Prática

Conta
- numero : int - nomeCliente : String - saldo : double - limite : double

powered by astah\*

```
public class Conta {  
  
    int numero;  
    String nomeCliente;  
    double saldo;  
    double limite;  
  
}
```

# Conceitos - Classes

---

## ► Usando a Classe

- Criar uma classe de execução que implemente o método *main*
- Instanciar -> criar objeto

```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        //instanciando -> criando objeto  
        Conta minhaConta = new Conta();  
  
    }  
  
}
```

# Conceitos - Classes

---

```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        //instanciando ->criando objeto  
        Conta minhaConta = new Conta();  
  
        minhaConta.numero = 0001;  
        minhaConta.nomeCliente = "Fulano de Tal";  
        minhaConta.saldo = 1000.00;  
        minhaConta.limite = 300.00;  
  
        System.out.println("O saldo da conta do cliente: " +  
                           minhaConta.nomeCliente +  
                           " é :" +  
                           minhaConta.saldo);  
    }  
}
```

- ▶ Utiliza-se o ponto (.) para acessar os atributos e métodos de um objeto.



# Classes na Prática

---

## ▶ Métodos

- ▶ A utilidade dos métodos é a de separar uma determinada função em pedaços menores.

```
<tipo de retorno> <nome do método>( [lista dos atributos] ) {  
    // implementação do método  
}
```

### ▶ tipo de retorno

- ▶ Pode ser um tipo primitivo ou um tipo de um classe.
- ▶ Caso o método não retorne nada, ele deve ser **void**.

### ▶ A lista de atributos não precisa ser informada se não há passagem de argumentos.

- ▶ Caso haja, os argumentos devem ser informados com seu tipo e nome, separados por vírgula se houver mais de um.



# Classes na Prática

---

## ► Métodos

```
void somaValores(int a, int b){  
    int soma;  
    soma = a + b;  
    System.out.println("A soma é: " + soma);  
}
```



# Classes na Prática

---

## ▶ Métodos

### ▶ Retorno dos Métodos

- ▶ A palavra reservada ***return*** causa o retorno do método.
- ▶ Quando os métodos são declarados com o tipo de retorno **void**, então o método não pode e nem deve retornar nada.
- ▶ Os métodos que retornam algum valor, devem retornar dados do tipo de retorno declarado, ou de tipos compatíveis.



# Classes na Prática

---

## ► Métodos

### ► Retorno dos Métodos

```
int somaValores(int a, int b){  
    int soma;  
    soma = a + b;  
    return soma;  
}
```



# Conceitos - Classes



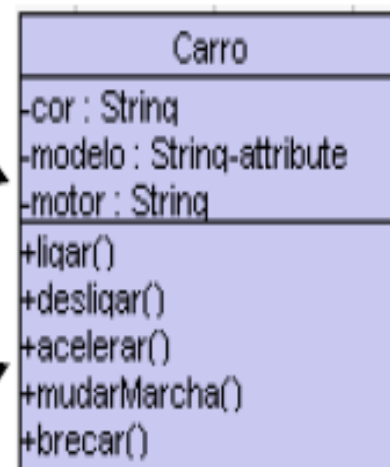
Entidade do mundo real

## Propriedades:

- ✓ modelo
- ✓ cor
- ✓ motor

## Comportamento:

- ✓ liga
- ✓ desliga
- ✓ muda marcha
- ✓ acelera
- ✓ breca



Modelo UML da classe

```
package OO;
public class ClasseCarro {
    String cor;
    String modelo;
    String motor;
    void ligar() {
        System.out.println( "Ligando o carro" );
    }
    void desligar() {
        System.out.println( "Desligando o carro" );
    }
    void acelerar() {
        System.out.println( "Acelerando o carro" );
    }
    void breicar() {
        System.out.println( "Brecando o carro" );
    }
    void mudarMarcha() {
        System.out.println( "Marcha engatada" );
    }
}
```



# Objetos na Prática

## ► Utilizando a Classe Carro

```
class ExemploCarro{  
    public static void main(String args[]){  
        //criando uma instância da classe Carro  
        Carro umCarro = new Carro();  
        //atribuindo os valores dos atributos  
        umCarro.modelo = "Gol";  
        umCarro.cor = "preto";  
        umCarro.motor = "1.0";  
        //executando os métodos do objeto  
        umCarro.ligar();  
        umCarro.mudarMarcha();  
        umCarro.acelerar();  
        umCarro.brecar();  
        umCarro.desligar();  
        //atribuindo null para a variável diz que  
        //agora ela não aponta para lugar nenhum  
        umCarro = null;  
    }  
}
```

# Conceitos - Classes

---

- ▶ Que ações podem ser feitas sobre a Conta?
  - ▶ Realizar o saque de um valor
  - ▶ Depositar um valor
  - ▶ Consultar/Imprimir informações da conta



# Conceitos - Classes

---

- ▶ Projeto de Conta
  - ▶ Identificação dos Métodos

Conta
<ul style="list-style-type: none"><li>- numero : int</li><li>- nomeCliente : String</li><li>- saldo : double</li><li>- limite : double</li></ul>
<ul style="list-style-type: none"><li>+ sacarDinheiro(valor : double) : boolean</li><li>+ depositarValor(valor : double) : boolean</li></ul>

powered by Astah 

# Conceitos - Classes

---

## ► Métodos

```
public class Conta {  
  
    int numero;  
    String nomeCliente;  
    double saldo;  
    double limite;  
  
    boolean sacarDinheiro(double valor){  
        if(saldo < valor){  
            return false;  
        }else{  
            saldo = saldo - valor;  
            return true;  
        }  
    }  
}
```

# Conceitos - Classes

---

## ► Chamando Métodos

```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        Conta minhaConta = new Conta();  
  
        minhaConta.saldo = 1000.00;  
        boolean consegui = minhaConta.sacarDinheiro(500.00) ;  
        if(consegui){  
            System.out.println("Saque realizado com sucesso!");  
        }else{  
            System.out.println("Saque não realizado!");  
        }  
    }  
}
```



# Pratique!

## ► Implementar as classes

Carro
- cor : String - modelo : String - motor : String
+ ligar() : void + desligar() : void + acelerar() : void + mudarMarcha() : void

powered by astah\*

Conta
- numero : int - nomeCliente : String - saldo : double - limite : double
+ sacarDinheiro(valor : double) : boolean + depositarValor(valor : double) : boolean

powered by Astah



# Pratique!

---



## Triangulo

- base : double
- altura : double
- + calcularArea() : double

## Quadrado

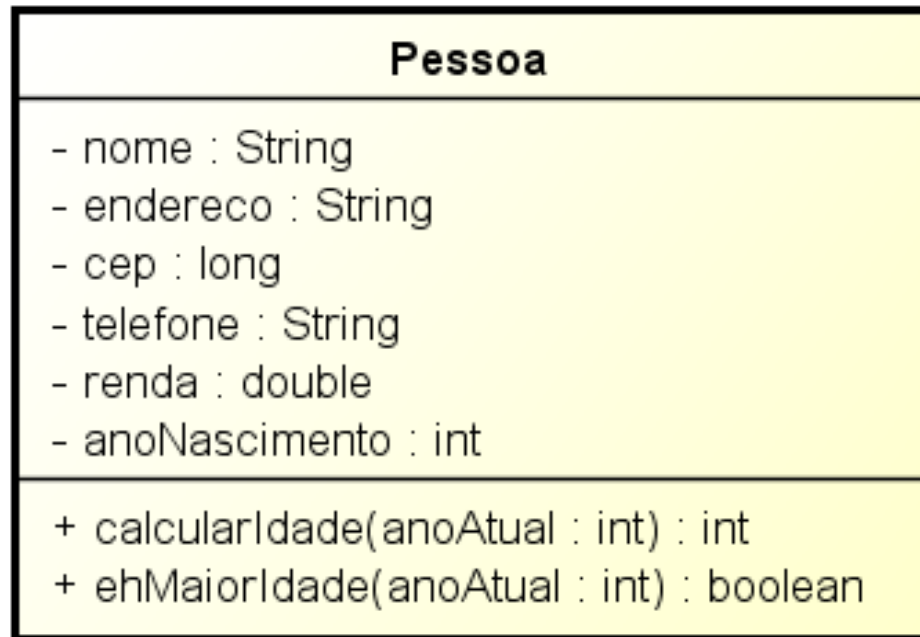
- area : double
- lado : double
- + calcularArea() : void

powered by Astah 



# Pratique!

---



//recuperando ano atual

```
Calendar c = Calendar.getInstance();
```

```
int anoAtual = c.get(Calendar.YEAR);
```





# Classes na Prática

---

## ► Construtores

- Quando usamos a palavra chave new, estamos construindo um objeto.
- Sempre quando o new é chamado, ele executa o construtor da classe.
  - Fazem a função de iniciação do objeto criado.

```
Conta minhaConta = new Conta();
```

Chamada do Construtor



# Classes na Prática

---

## ► Construtores

- O construtor da classe é um bloco declarado com o mesmo nome que a classe.
- Se nenhum construtor for declarado, um construtor *default* será criado.

```
Conta minhaConta = new Conta();
```



# Contrutores

```
public class Conta {
```

```
    int numero;
```

```
    String nomeCliente;
```

```
    double saldo;
```

```
    double limite;
```

```
    Conta(){
```

```
}
```

```
    boolean sacarDinheiro(double valor){
```

```
        if(saldo > valor){
```

```
            return false;
```

```
        }else{
```

```
            saldo = saldo - valor;
```

```
            return true;
```

```
        }
```

```
    }
```

```
}|
```

**Declaração implícita do Construtor**

```
Conta minhaConta = new Conta();
```

```
public class Conta {
```

```
    int numero;  
    String nomeCliente;  
    double saldo;  
    double limite;
```

**Alteração do Construtor**

```
    Conta(int numero, String nomeCliente, double saldo, double limite){  
        this.numero = numero;  
        this.nomeCliente = nomeCliente;  
        this.saldo = saldo;  
        this.limite = limite;  
    }
```

```
    boolean sacarDinheiro(double valor){  
        if(saldo > valor){  
            return false;  
        }else{  
            saldo = saldo - valor;  
            return true;  
        }  
    }  
}
```

```
Conta minhaConta = new Conta(0001, "Fulano", 1000.00, 600.00);
```



# Classes na Prática

---

## ► Construtores

```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        Conta minhaConta = new Conta(0001, "Fulano", 1000.00, 600.00);  
  
        System.out.println("Nome do Cliente: " + minhaConta.nomeCliente);  
        System.out.println("Saldo atual: " + minhaConta.saldo);  
    }  
}
```

Nome do Cliente: Fulano Saldo atual: 1000.0
--



# Classes na Prática

## ► Sobrecarga – *Overload*

- Ocorre quando mais de um método com o mesmo nome é implementado.
- Pode se dar pela diferenciação do tipo de retorno e/ou dos argumentos do método.

Operacao
- operacao : String
+ somar(a : int, b : int) : int
+ somar(a : int, b : int, c : int) : int
+ somar(a : double, b : double) : double
+ somar(a : double, b : double, c : double) : double





# Classes na Prática

## ► Sobrecarga de métodos

```
public class Operacao {  
  
    String operacao;  
  
    int somar(int a, int b){  
        return a + b;  
    }  
  
    int somar(int a, int b, int c){  
        return a + b + c;  
    }  
  
    double somar(double a, double b){  
        return a + b;  
    }  
  
    double somar(double a, double b, double c){  
        return a + b + c;  
    }  
}
```



# Classes na Prática

---

## ► Sobrecarga de Construtores

```
public class Operacao {  
  
    String operacao;  
  
    Operacao() {  
    }  
  
    Operacao(String operacao) {  
        this.operacao = operacao;  
    }  
}
```



# Classes na Prática

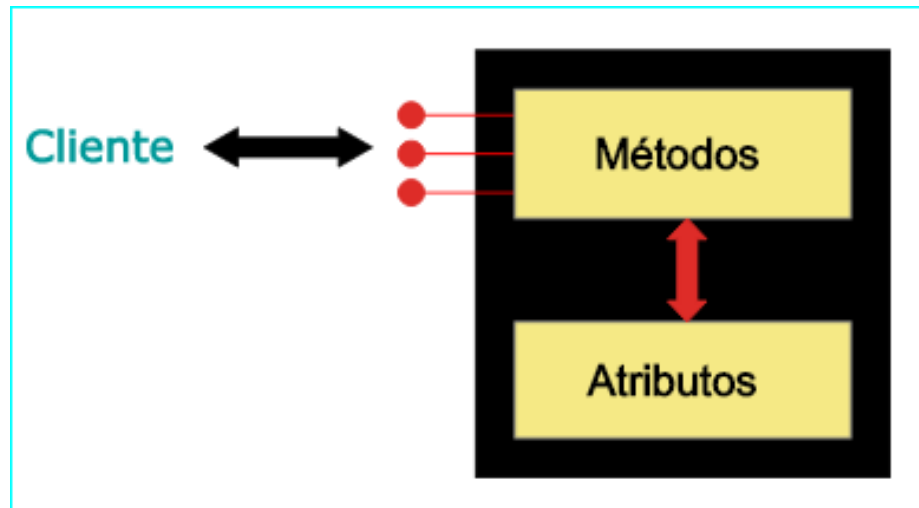
## ► Visões de um objeto

### ► Interna

- Atributos e métodos da classe que o define

### ► Externa

- Serviços que o objeto proporciona e como ele interage com outros objetos

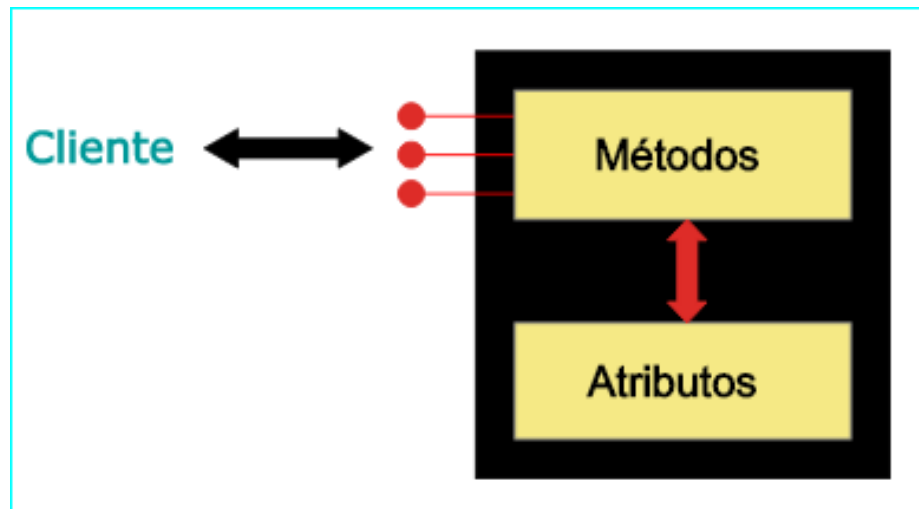




# Classes na Prática

## ► Visões de um objeto

- Externamente, um objeto deve ser uma entidade encapsulada
  - Um objeto pode usar os serviços providos por outro objeto
  - Mas não deve saber como estes serviços são implementados





# Classes na Prática

---

## ► Encapsulamento

- Proteger os dados referentes a um objeto
  - Previne o problema de interferência externa indevida sobre os dados de um objeto
- O objeto será visto como uma caixa preta
  - Os detalhes internos da classe permanecerão ocultos aos objetos
- A implementação do encapsulamento é possível a partir dos **modificadores de acesso**.



# Classes na Prática

---

## ► **Modificadores de Acesso**

- Determinam a visibilidade da classe e de seus membros (atributos e métodos).
- Ao todo são quatro modificadores principais
  - Public
  - Protected
  - Private
  - *Default/Package – (não é atribuído modificador)*



# Classes na Prática

## ► Modificadores de Acesso

```
[modificador] class Pessoa {  
    [modificador] String nome;  
    [modificador] int idade;  
  
    [modificador] void imprimirNomeIdade() {  
        System.out.println("nome="+nome);  
        System.out.println("idade="+idade);  
    }  
}
```

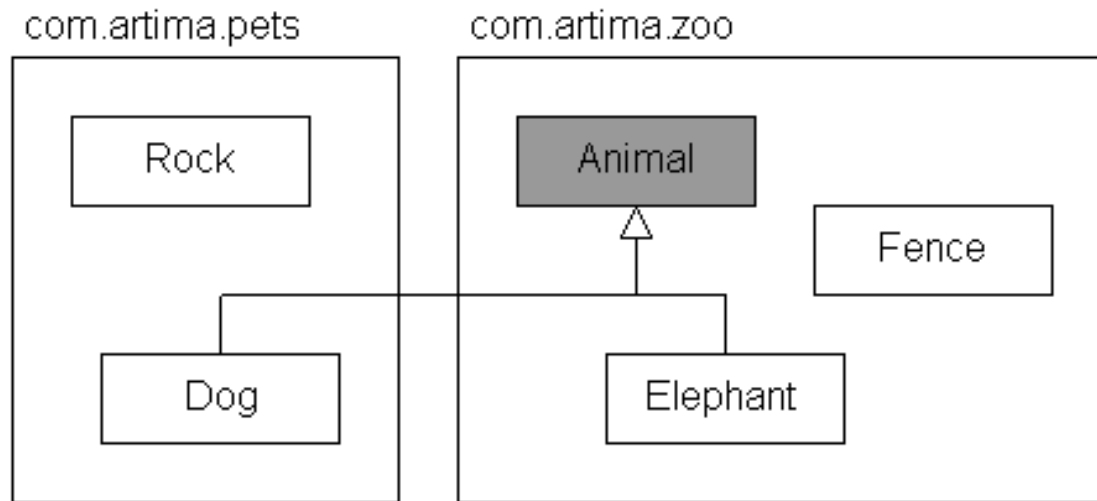
	private	default	protected	public
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim



# Classes na Prática

## ► Modificadores de Acesso

### Private Access



- Has access to **private** member of `Animal`
- Does not have access

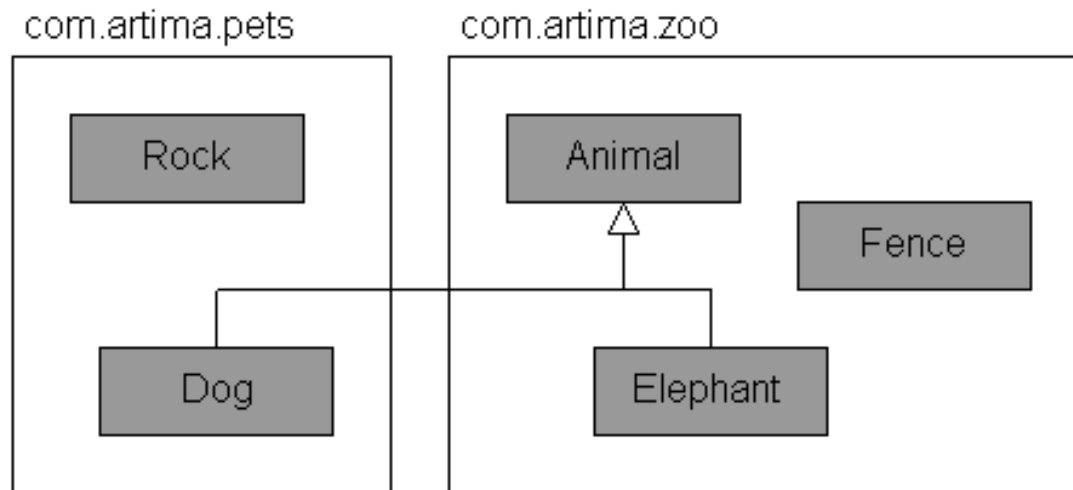




# Classes na Prática

## ► Modificadores de Acesso

### Public Access



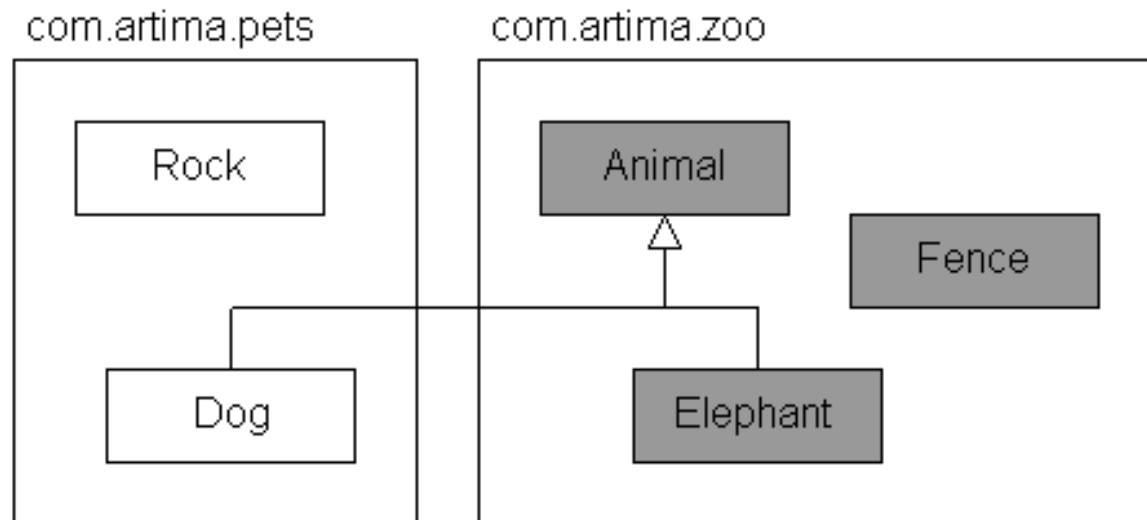
- Has access to **public** member of `Animal`
- Does not have access



# Classes na Prática

## ► Modificadores de Acesso

### Package Access



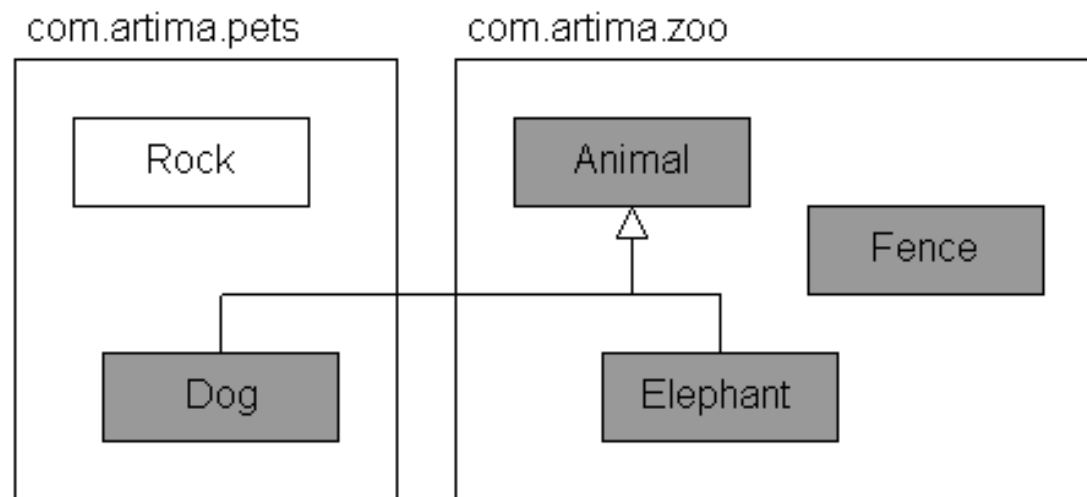
- Has access to **package** member of Animal
- Does not have access



# Classes na Prática

## ► Modificadores de Acesso

### Protected Access



- Has access to **protected** member of Animal
- Does not have access



# Classes na Prática

---

## ► Encapsulamento

- Ocultar detalhes internos da classe
- Segundo o encapsulamento, os atributos não podem ser acessados/alterados diretamente.
  - Acesso => Atributos *PRIVATE*
  - Alteração => Métodos *SET* e *GET*.



# Classes na Prática

---

## ► Encapsulamento

### ► Métodos *get*

- Responsável por retornar o valor de uma variável

```
tipoAtributo getAtributo() {  
    return this.atributo;  
}
```

### ► Métodos *set*

- Responsável por atribuir o valor a uma variável

```
void setAtributo(tipoAtributo atributo) {  
    this.atributo = atributo;  
}
```



# Classes na Prática

---

## ► Encapsulamento

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    public Pessoa() {}  
  
    public void setNome( String nome ) {  
        this.nome = nome;  
    }  
  
    public void setIdade( int idade ) {  
        this.idade = idade;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public int getIdade() {  
        return idade;  
    }  
}
```



# Classes na Prática

---

## ► Encapsulamento

### ► Passos para criar

- Modificar o acesso aos atributos para private
- Criar métodos get e set para cada atributo



# Classes na Prática

---

## ► Encapsulamento

```
public class Conta {  
  
    private int numero;  
    private String nomeCliente;  
    private double saldo;  
    private double limite;  
  
}
```



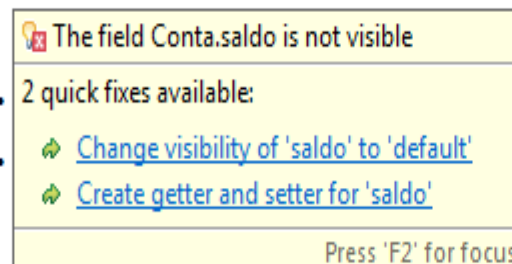


# Classes na Prática

## ► Encapsulamento

```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        Conta minhaConta = new Conta();  
  
        minhaConta.nomeCliente = "Fulano";  
        minhaConta.saldo = 0.0;  
  
        System.out.  
        System.out.  
  
    }  
  
}
```

**!!ERRO!!**



```
        System.out.println("Nome: " + minhaConta.nomeCliente);  
        System.out.println("Saldo: " + minhaConta.saldo);
```



# Classes na Prática

---

```
public int getNumero() {  
    return numero;  
}  
public void setNumero(int numero) {  
    this.numero = numero;  
}  
public String getNomeCliente() {  
    return nomeCliente;  
}  
public void setNomeCliente(String nomeCliente) {  
    this.nomeCliente = nomeCliente;  
}  
public double getSaldo() {  
    return saldo;  
}  
public void setSaldo(double saldo) {  
    this.saldo = saldo;  
}  
public double getLimite() {  
    return limite;  
}  
public void setLimite(double limite) {  
    this.limite = limite;  
}
```



# Classes na Prática

---

## ► Encapsulamento

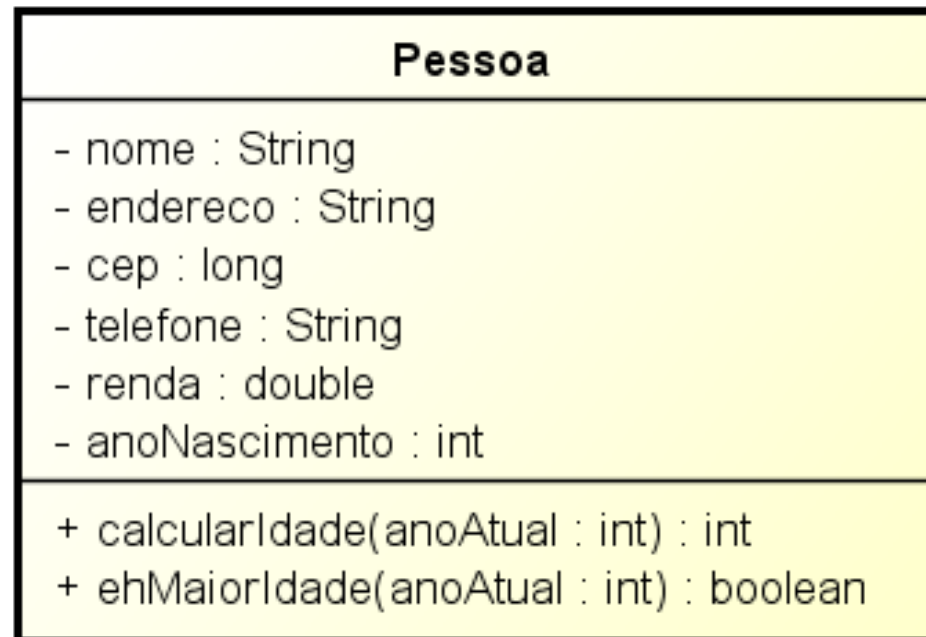
```
public class ExecConta {  
  
    public static void main(String args[]){  
  
        Conta minhaConta = new Conta();  
  
        minhaConta.setNomeCliente("Fulano");  
        minhaConta.setSaldo(0.0);  
  
        System.out.println("Nome do Cliente: " + minhaConta.getNomeCliente());  
        System.out.println("Saldo atual: " + minhaConta.getSaldo());  
    }  
  
}
```



# Exercício 01

---

## ► Encapsulamento



powered by Astah 



## Exercício 02

---

### ► Encapsulamento + Construtor

Operacao
- valor1 : double - valor2 : double
+ somar() : double + subtrair() : double + multiplicar() : double + dividir() : double

powered by Astah 



## Exercício 03

---

### ► Encapsulamento + Construtor

Triangulo
- base : double - altura : double
+ calcularArea() : double

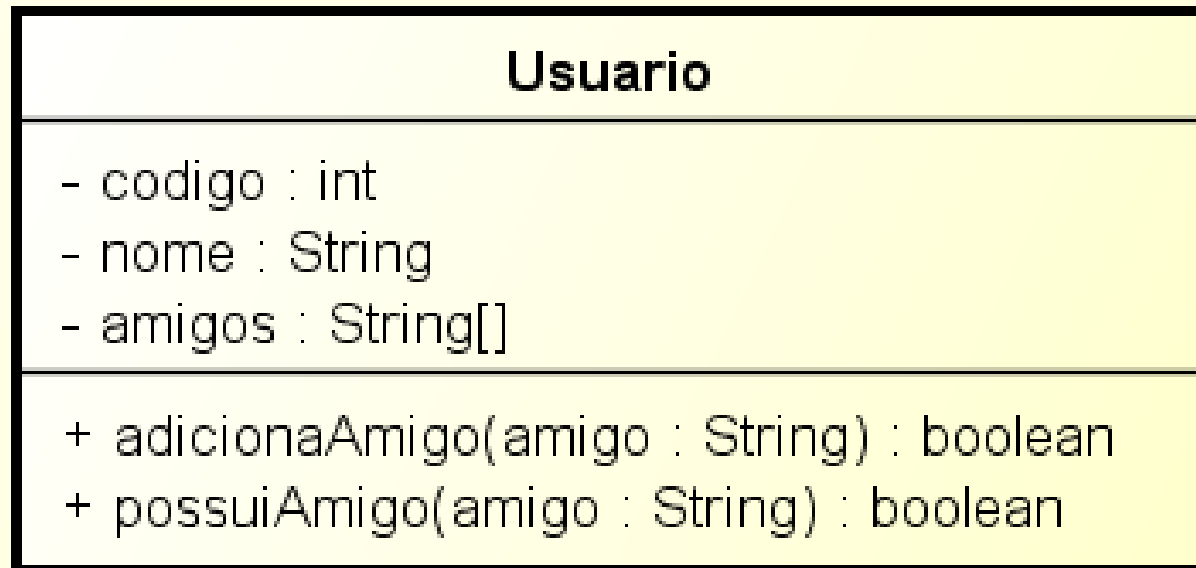
Quadrado
- area : double - lado : double
+ calcularArea() : void

powered by Astah 



## Exercício 04

### ► Encapsulamento + Construtor



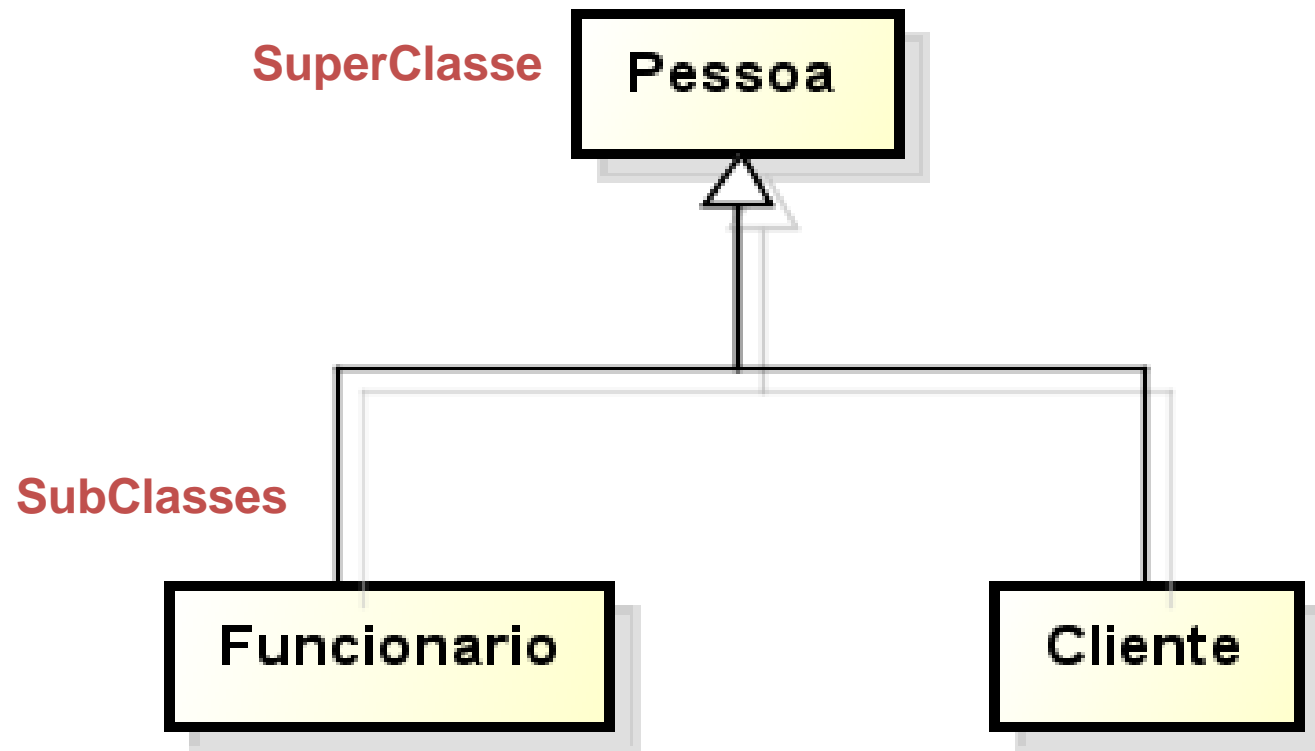
powered by Astah 



# Conceitos - Herança

---

- ▶ Permite criar novas classes a partir de classes já existentes.
- ▶ Reflete um relacionamento de especialização

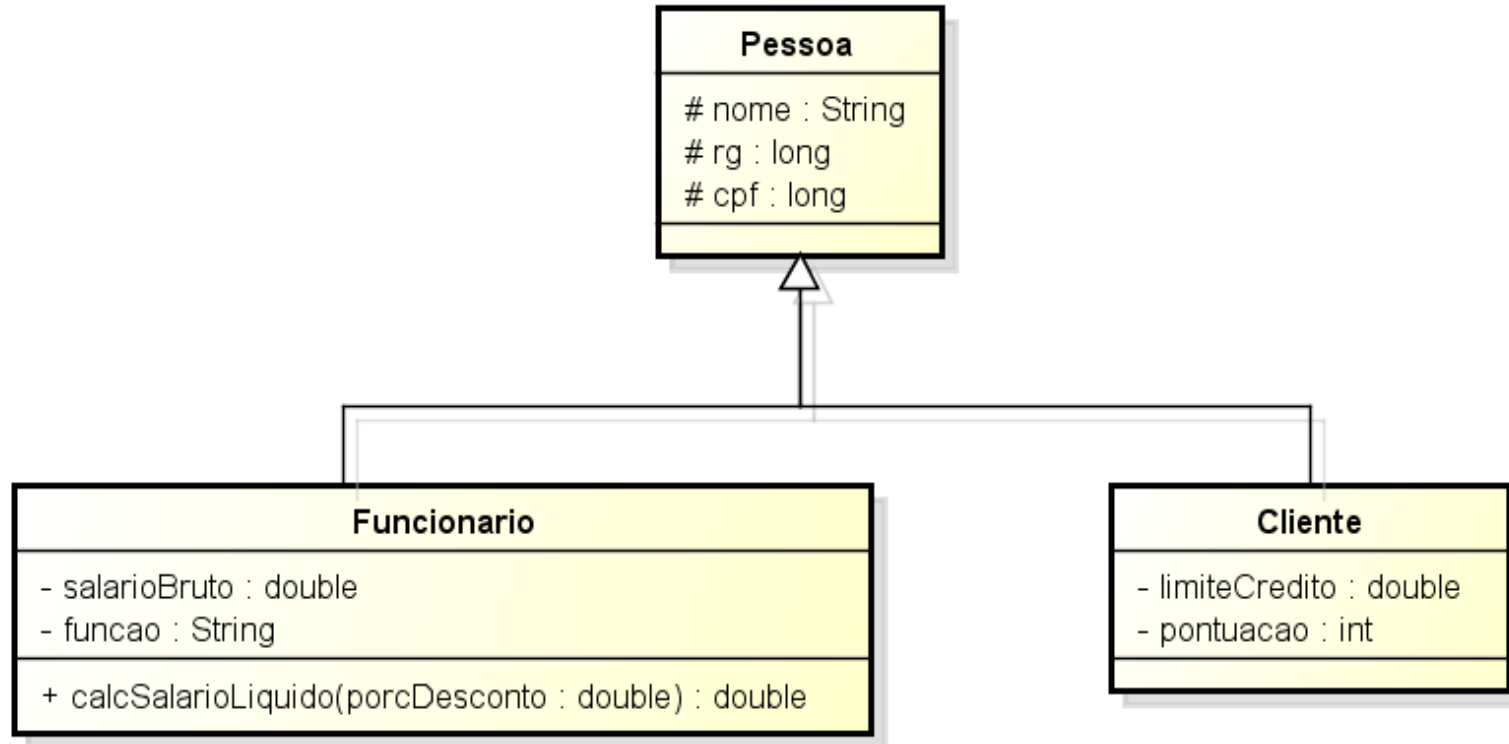






# Conceitos - Herança

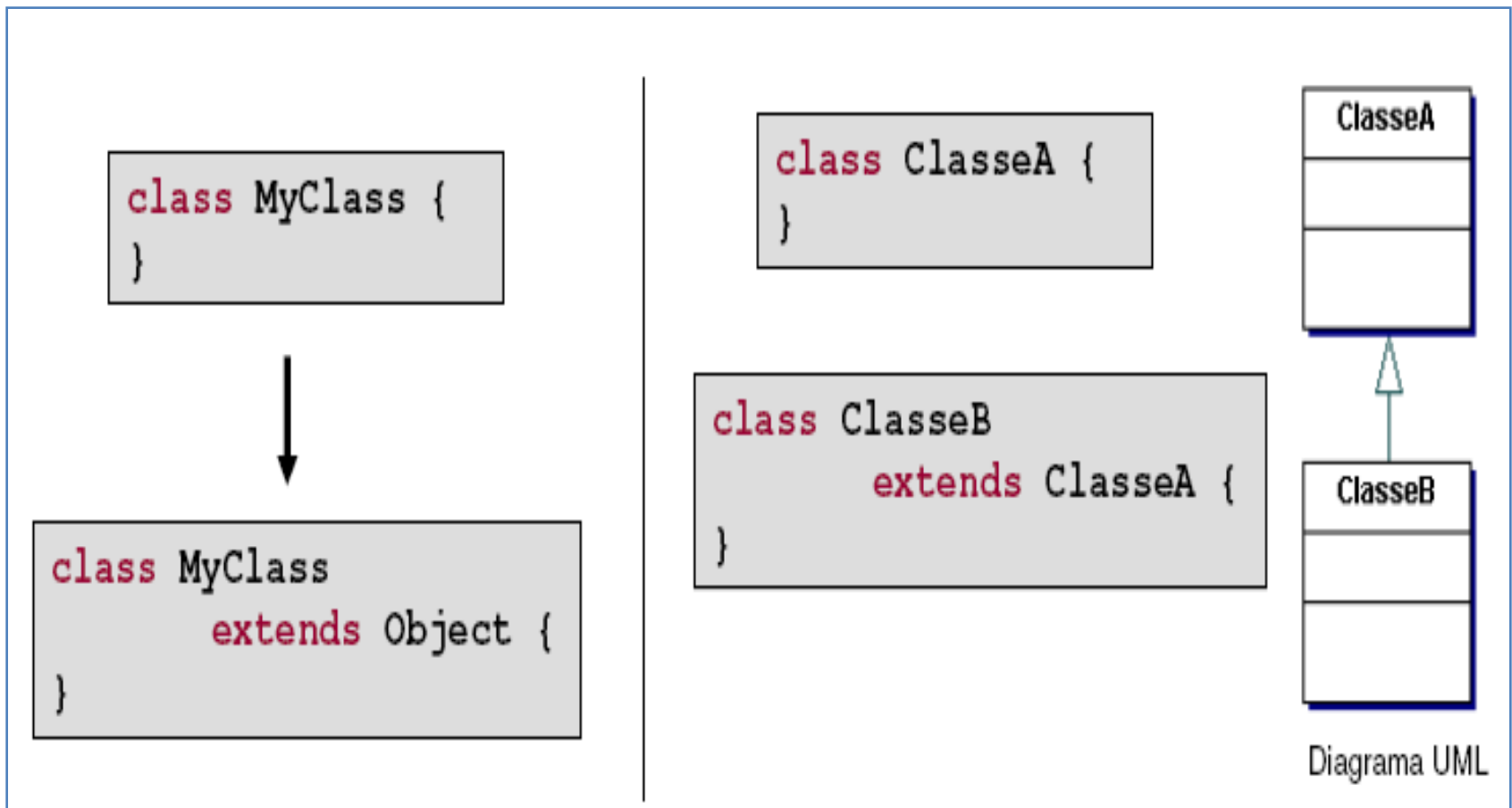
- ▶ Todos os métodos e atributos (**public e protected**) são herdados pelas subclasses
- ▶ Os construtores não são herdados.





# Herança na Prática

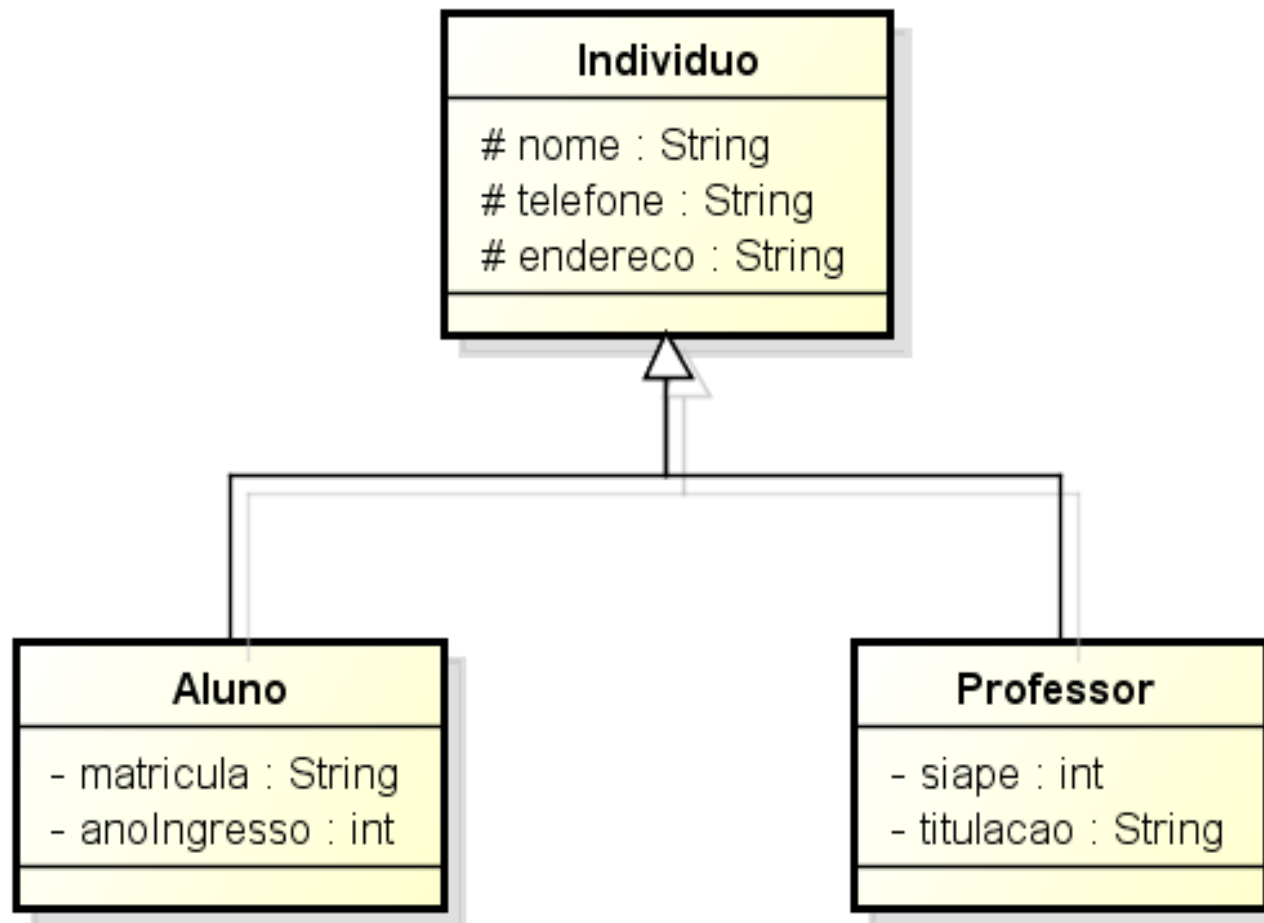
- ▶ Em Java, a herança é conseguida através da palavra ***extends***;





# Herança na Prática

---





# Herança na Prática

```
8 public class Indivíduo {
9
10     protected String nome;
11     protected String telefone;
12     protected String endereco;
13
14     public String getNome() { ...3 lines }
17
18     public void setNome(String nome) { ...3 lines }
21
22     public String getTelefone() { ...3 lines }
25
26     public void setTelefone(String telefone) { ...3 lines }
29
30     public String getEndereco() { ...3 lines }
33
34     public void setEndereco(String endereco) { ...3 lines }
```



# Herança na Prática

```
public class Aluno extends Individuo {  
  
    private String matricula;  
    private int anoIngresso;  
  
    public String getMatricula() { ...3 lines }  
  
    public void setMatricula(String matricula) { ...3 lines }  
  
    public int getAnoIngresso() { ...3 lines }  
  
    public void setAnoIngresso(int anoIngresso) { ...3 lines }  
  
}
```



# Herança na Prática

```
public class ExecAluno {  
  
    public static void main(String args[]){  
  
        Aluno aluno = new Aluno();  
  
        aluno.setNome("Fulano");  
        aluno.setTelefone("(00) 0000-0000");  
        aluno.setEndereco("Rua X, Bairro Y");  
        aluno.setMatricula("123456789");  
        aluno.setAnoIngresso(2016);  
  
        System.out.println("Nome: " + aluno.getNome() +  
                           " Matrícula: " + aluno.getMatricula());  
  
    }  
}
```



# Herança na Prática

- ▶ O construtor da superclasse é chamado automaticamente, se outra chamada não for feita.
- ▶ A palavra **super** referencia a superclasse.

```
public class Pessoa {  
    protected String nome;  
    protected String cpf;  
  
    public Pessoa() {}  
}
```

```
public class Empregado extends Pessoa {  
    public Empregado() {  
        super(); //ocorre automaticamente  
    }  
}
```

```
public class Gerente extends Pessoa {  
}
```

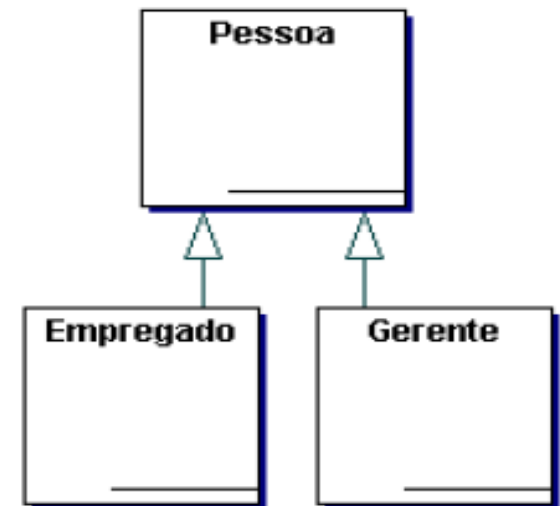


Diagrama UML



# Herança na Prática

## ► Modificando construtor

```
public class Indivíduo {  
  
    protected String nome;  
    protected String telefone;  
    protected String endereco;  
  
    public Indivíduo(String nome, String telefone, String endereco) {  
        this.nome = nome;  
        this.telefone = telefone;  
        this.endereco = endereco;  
    }  
  
    public String getNome() { ...3 lines }
```





# Herança na Prática

## ► Modificando construtor

constructor Individuo in class Individuo cannot be applied to given types;  
required: String,String,String  
found: no arguments  
reason: actual and formal argument lists differ in length

----

(Alt-Enter shows hints)

```
public class Aluno extends Individuo {  
  
    private String matricula;  
    private int anoIngresso;  
  
    public String getMatricula() {  
        return matricula;  
    }  
}
```



# Herança na Prática

## ► Modificando construtor

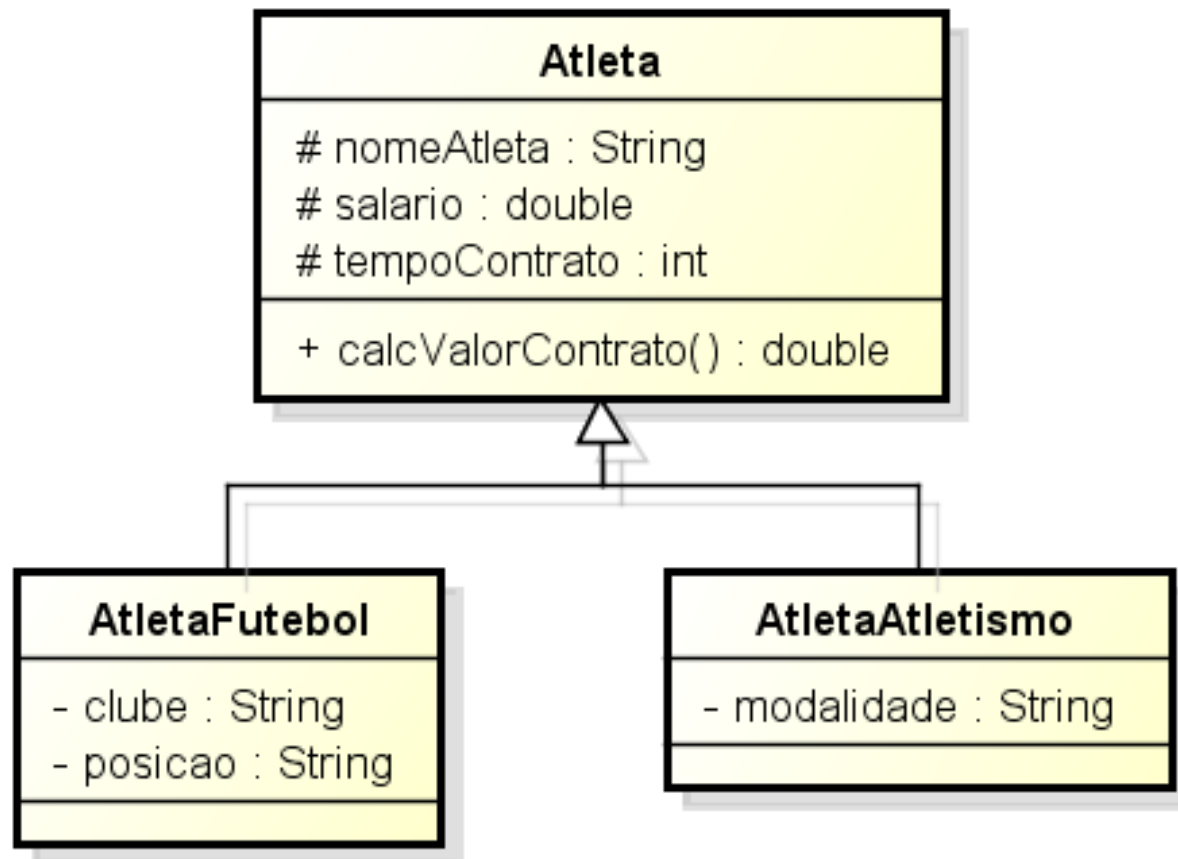
```
public class Aluno extends Individuo {  
  
    private String matricula;  
    private int anoIngresso;  
  
    public Aluno(String matricula, int anoIngresso,  
                 String nome, String telefone, String endereco) {  
        super(nome, telefone, endereco);  
        this.matricula = matricula;  
        this.anoIngresso = anoIngresso;  
    }  
  
    public String getMatricula() {  
        return matricula;  
    }  
}
```



## Exercício 05

### ► HERANÇA

*calcValorContrato =  
salario \* tempoContrato*



powered by Astah 

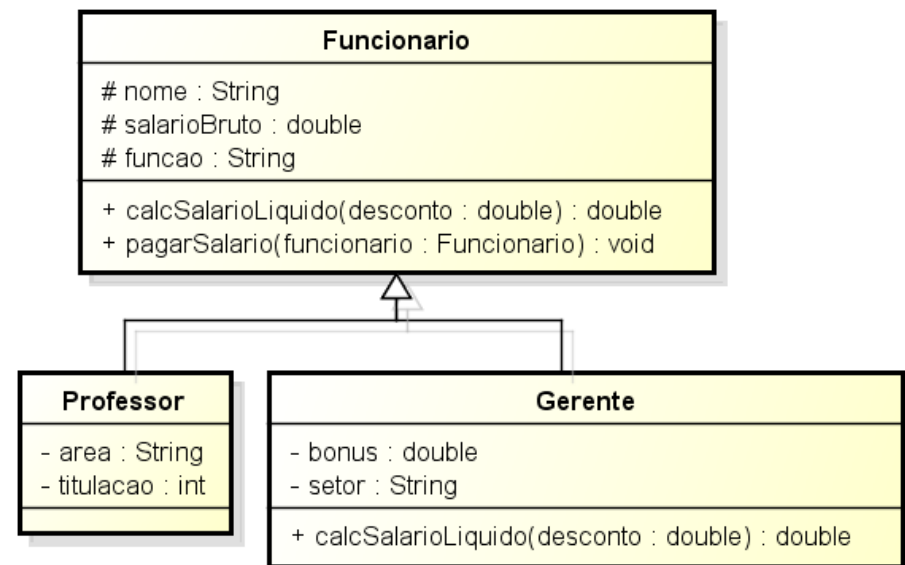


# Polimorfismo Dinâmico

## ► Definições

- Capacidade de um objeto poder ser referenciado de várias formas.
- Princípio a partir do qual objetos derivados de uma mesma classe são capazes de invocar métodos que, **embora apresentem a mesma assinatura**, se comportam de maneira diferente.

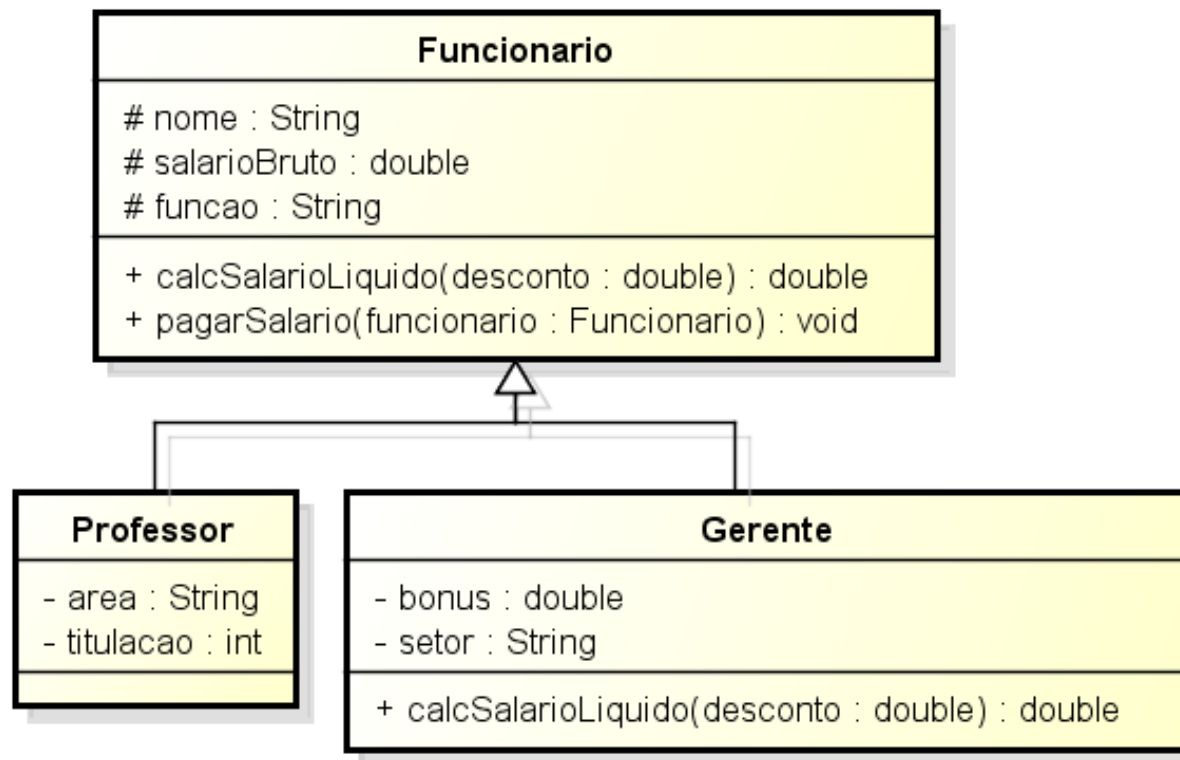
```
Funcionario f = new Funcionario();  
Funcionario p = new Professor();  
Funcionario g = new Gerente();  
  
f.calcSalarioLiquido();  
p.calcSalarioLiquido();  
g.calcSalarioLiquido();
```





# Polimorfismo Dinâmico

- ▶ Se dá pela redefinição/sobrescrita (mesma assinatura) de métodos herdados



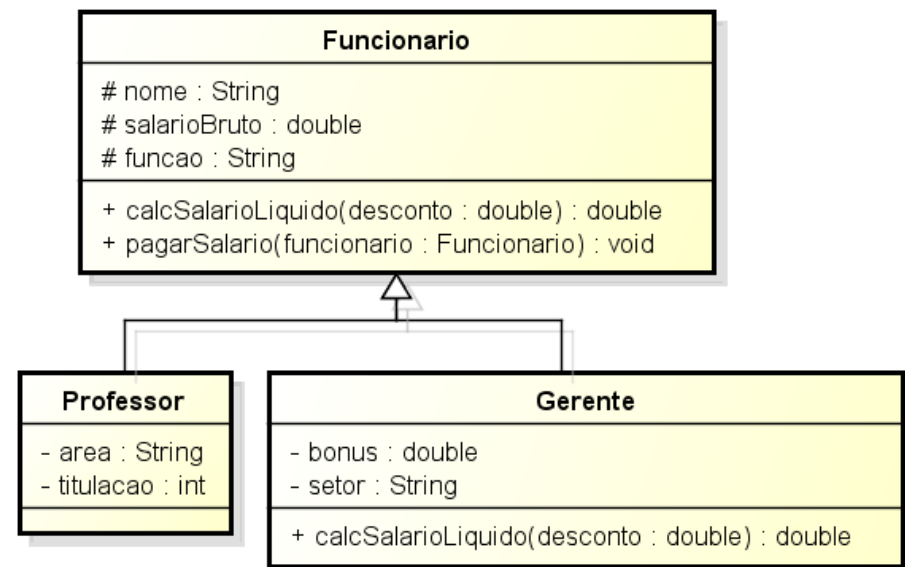
powered by Astah



# Polimorfismo Dinâmico

- ▶ A invocação de um método é decidida em tempo de execução
  - ▶ Primeiro procura-se pelo objeto em memória, depois é decidido qual método será executado.
  - ▶ Sempre relaciona o objeto com a sua classe original e não a que está sendo usada para referenciá-la.

```
Funcionario f = new Funcionario();  
Funcionario p = new Professor();  
Funcionario g = new Gerente();  
  
f.calcSalarioLiquido();  
p.calcSalarioLiquido();  
g.calcSalarioLiquido();
```





# Polimorfismo Dinâmico

- Qual a utilidade disso?

```
boolean pagarFuncionario(Funcionario f){  
    double salario = f.calcSalarioLiquido();  
    return depositar(salario);  
}
```

```
Professor professor = new Professor();  
Gerente gerente = new Gerente();
```

```
pagarFuncionario(gerente);  
pagarFuncionario(professor);
```

- *não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.*



## Exercício 06

### ► HERANÇA + Polimorfismo

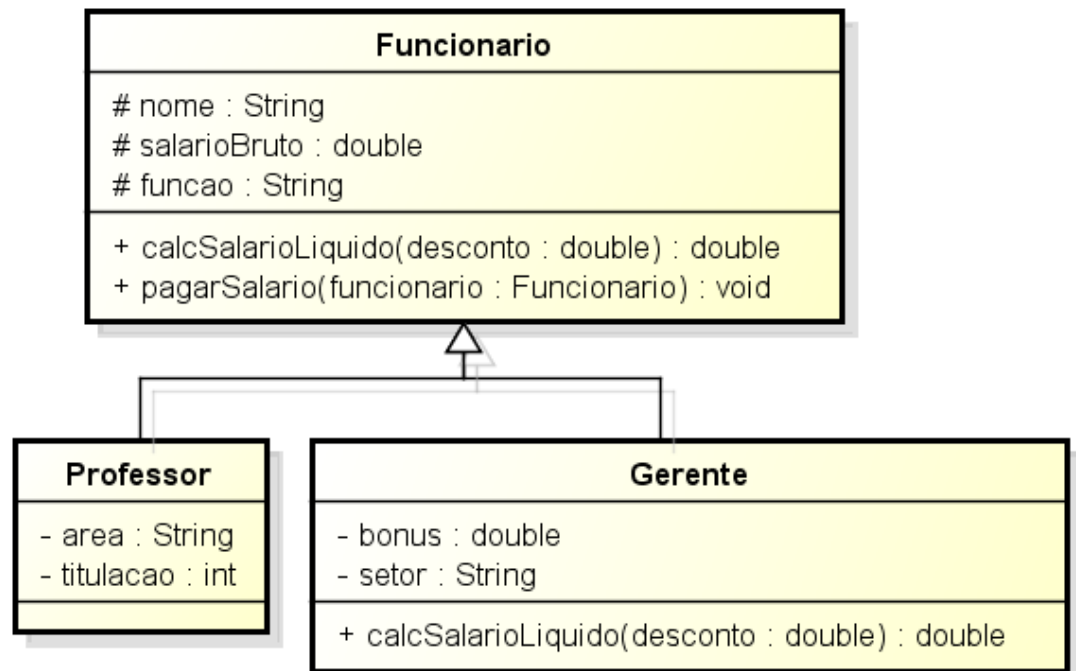
#### **Funcionario**

*calcSalarioLiquido =  
salarioBruto – desconto*

*pagarSalario = monta Msg:  
“Depositando {salarioLiquido}  
na conta de {nomeFuncionario}”*

#### **Gerente**

*calcSalarioLiquido =  
salarioBruto – desconto +  
bonus*



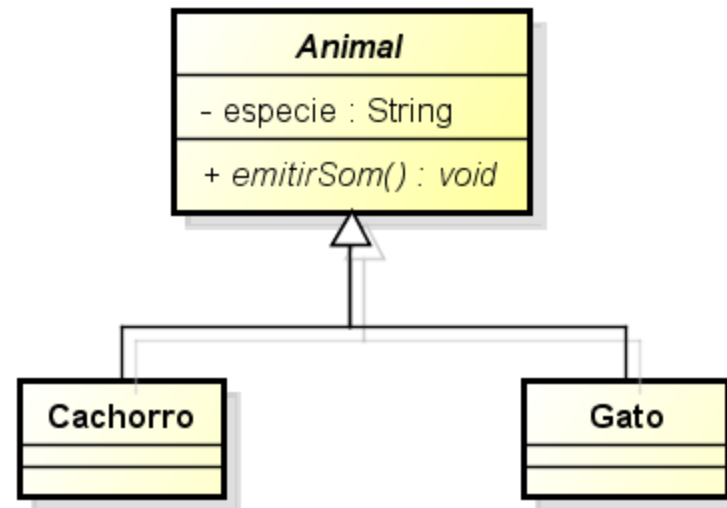
powered by Astah





# Polimorfismo – Classes Abstratas

- ▶ Conceito aplicado quando a classe for apenas um ponto de referência para as subclasses.
- ▶ Se a classe possuir um método abstrato ela obrigatoriamente deverá ser definida como abstrata.
- ▶ Classes abstratas não podem ser instanciadas



powered by Astah

# Polimorfismo – Classes Abstratas

---

```
public abstract class Animal {  
    protected String especie;  
    abstract void emitirSom();  
}
```

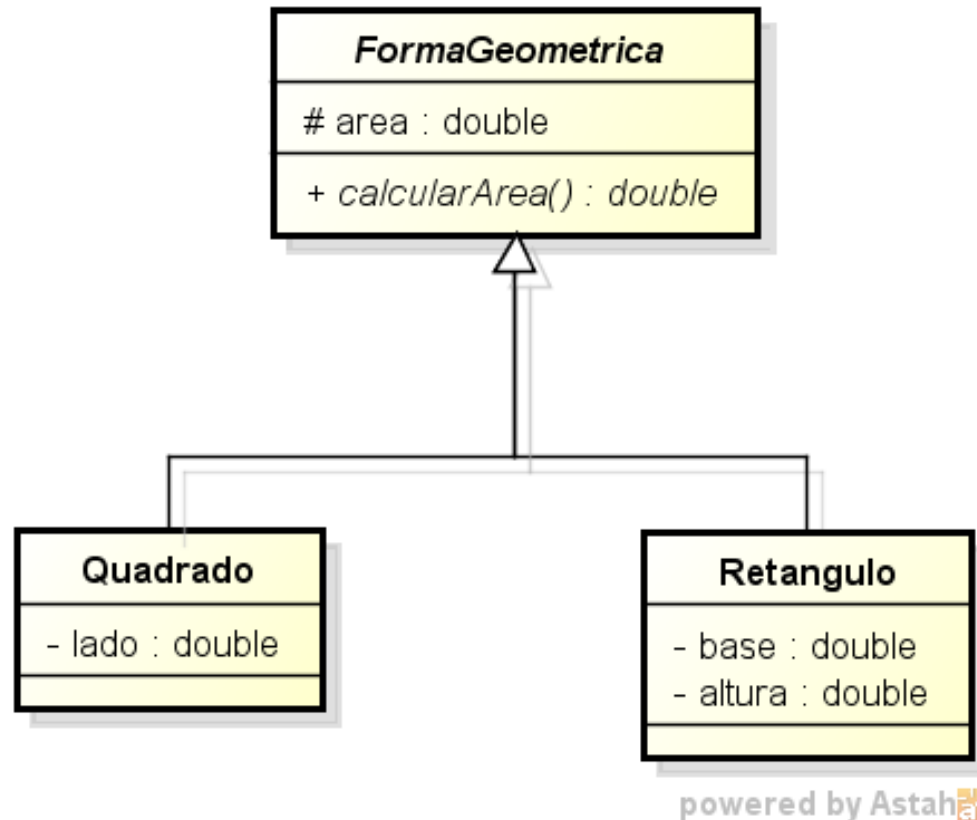
```
public class Cachorro extends Animal {  
  
    void emitirSom() {  
        System.err.println("Au au!!");  
    }  
  
}
```

```
public class Gato extends Animal{  
  
    void emitirSom() {  
        System.err.println("Miau miau!!");  
    }  
  
}
```



## Exercício 07

### ► HERANÇA + Polimorfismo (Abstract)





# Interface

---

- ▶ As interfaces atuam como um **contrato** que define parte do comportamento de outras classes.
- ▶ Uma interface pode definir uma série de métodos, **mas nunca conter implementação deles.**
  - ▶ Expõe o que o objeto deve fazer, e não como ele faz, nem o que ele tem.
- ▶ As classes podem implementar mais de uma interface.

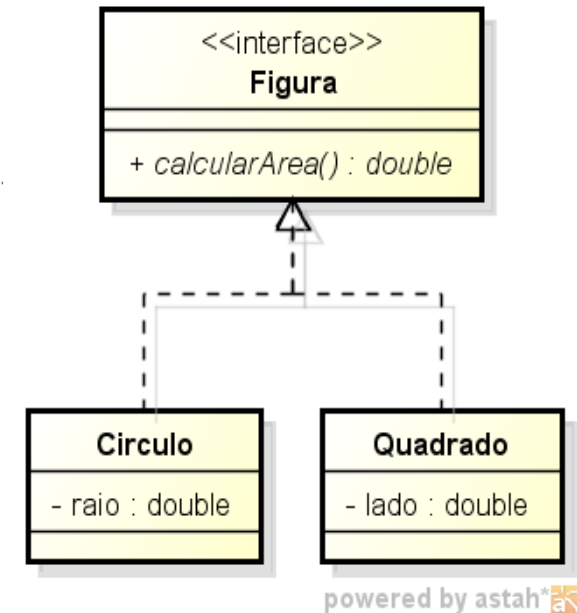
# Interface

---

```
public interface Figura {  
    public double calcularArea();  
}
```

```
public class Circulo implements Figura {  
    public double calcularArea() {  
        //faz o cálculo da área do círculo  
    }  
}
```

```
public class Quadrado implements Figura {  
    public double calcularArea() {  
        //faz cálculo da área do quadrado  
    }  
}
```





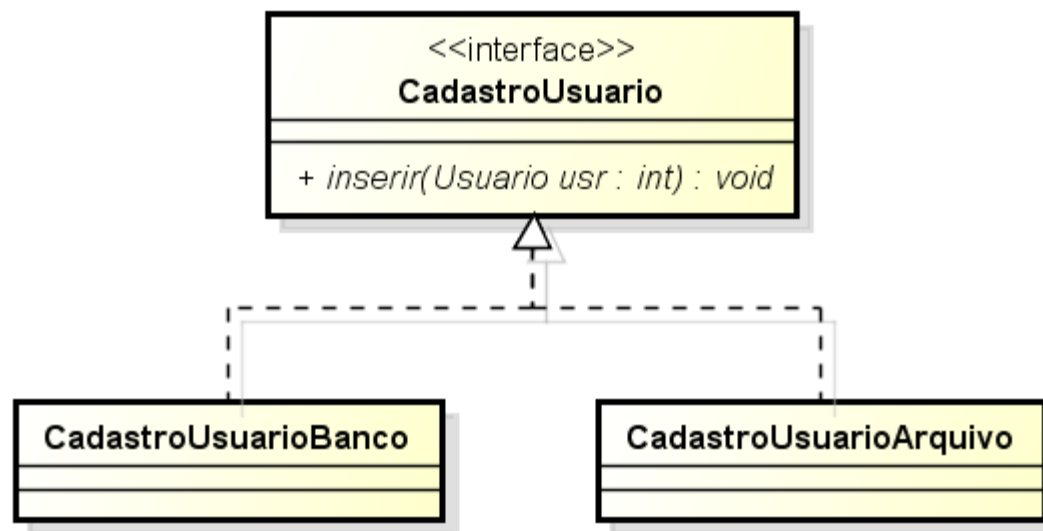
# Interface

---

- ▶ Exemplo Classe de Execução
  - ▶ ExecFigura

```
Figura fig = new Circulo(10);  
double area = fig.calcularArea();  
  
fig = new Quadrado(8);  
area = fig.calcularArea();
```

# Interface



powered by astah\*

```
public interface CadastroUsuario {
    public void inserir( Usuario usr ) throws Exception;
}
```

```
public class CadastroUsuarioBanco implements CadastroUsuario {
    public void inserir( Usuario usr ) throws Exception {
        //insere os dados no banco
    }
}
```

```
public class CadastroUsuarioArquivo implements CadastroUsuario {
    public void inserir( Usuario usr ) throws Exception {
        //insere os dados no arquivo
    }
}
```



# Interface

---

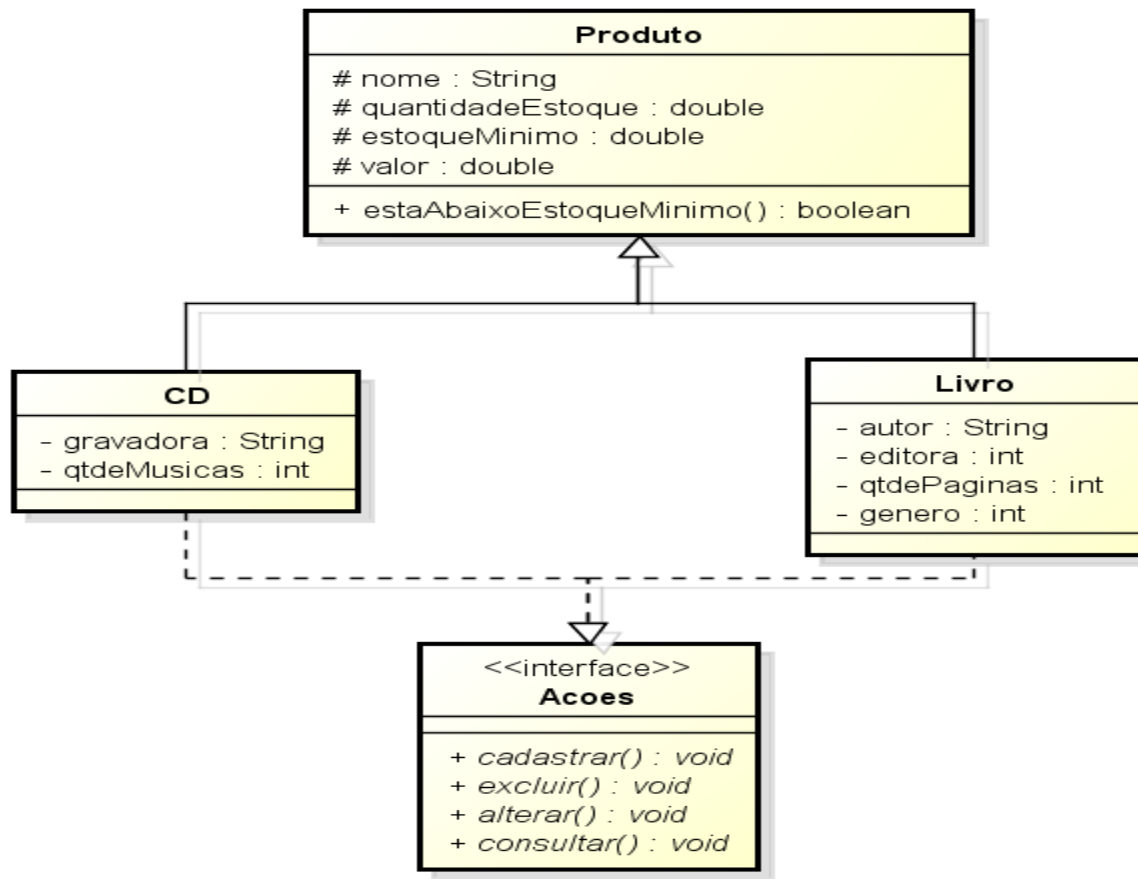
- ▶ Exemplo Classe de Execução
  - ▶ ExecCadastroUsuario

```
//...  
CadastroUsuario cad = new CadastroUsuarioBanco();  
cad.inserir( usuario );  
cad = new CadastroUsuarioArquivo();  
cad.inserir( usuario );  
//...
```



# Exercício 08

## ► Herança + Interface



powered by astah®



# Herança x Interface

Interface	Herança
Uma classe pode implementar mais de uma interface	Uma classe só pode ter uma ancestral.
Uma classe é obrigada a implementar TODOS os métodos da interface, caso contrário será considerada como ABSTRATA.	Uma classe já recebe a implementação de TODOS os métodos de sua ancestral. Ela pode, opcionalmente, fazer um Override dos métodos herdados.
Uma interface declara métodos e/ou constantes, sem implementação.	Uma classe ancestral pode ser totalmente funcional.
Métodos em uma interface não podem ser “private” ou “protected”. Além disto os modificadores: “transient”, “volatile” e “synchronized” não podem ser utilizados.	Todos os modificadores podem ser utilizados em uma classe ancestral de outras.



# Array de Referências

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

String[] nomes = new String[5];

for (int i = 0; i < nomes.length; i++) {
    nomes[i] = s.nextLine();
}

for (String nome : nomes) {
    System.out.println(nome);
}
```

Conta
- numero : int - nomeCliente : String - saldo : double - limite : double
+ sacarDinheiro(valor : double) : boolean + depositarValor(valor : double) : boolean

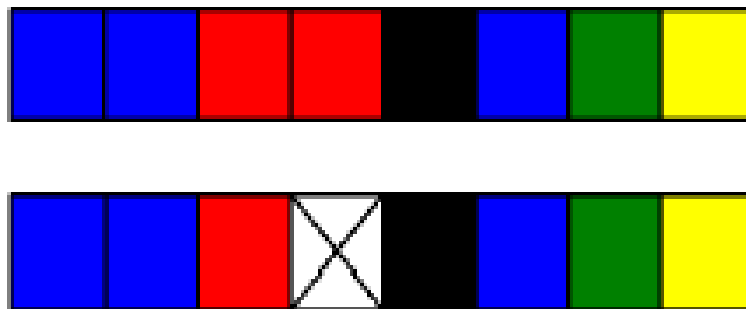
- ▶ Crie o código para ler, percorrer e exibir um array de objetos do tipo Conta



# Limitações do Array

---

- ▶ Não podemos redimensionar um array;
- ▶ Quantos elementos foram inseridos?
- ▶ Quais posições estão livres?



**Retire a quarta Conta**

`conta[3] = null;`



# API Collections

---

- ▶ Permite criar e gerenciar coleções de vários tipos de objetos.
- ▶ Arquitetura

## Interfaces

- Permitem que as coleções sejam manipuladas independentes de suas implementações

## Implementações

- Implementam uma ou mais interfaces do *framework*

## Algoritmos

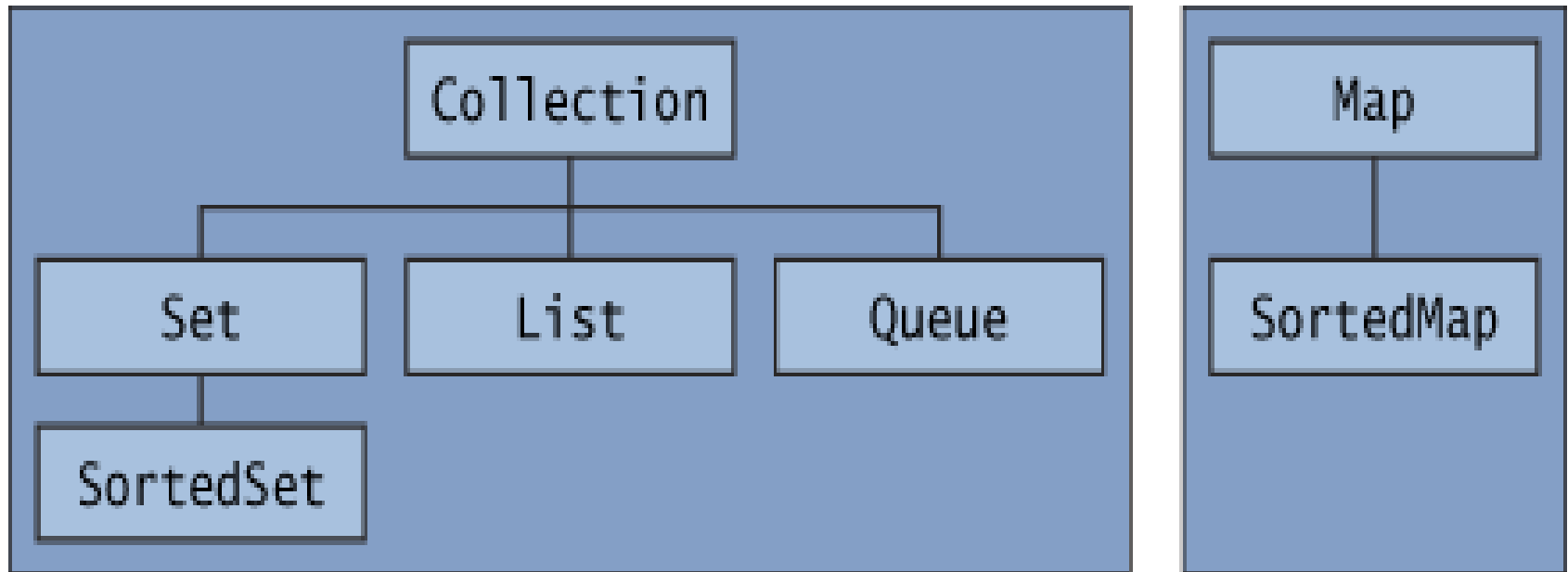
- São métodos que realizam operações (*sort*, *reverse*, *binarySearch*) sobre as coleções



# Collections API

---

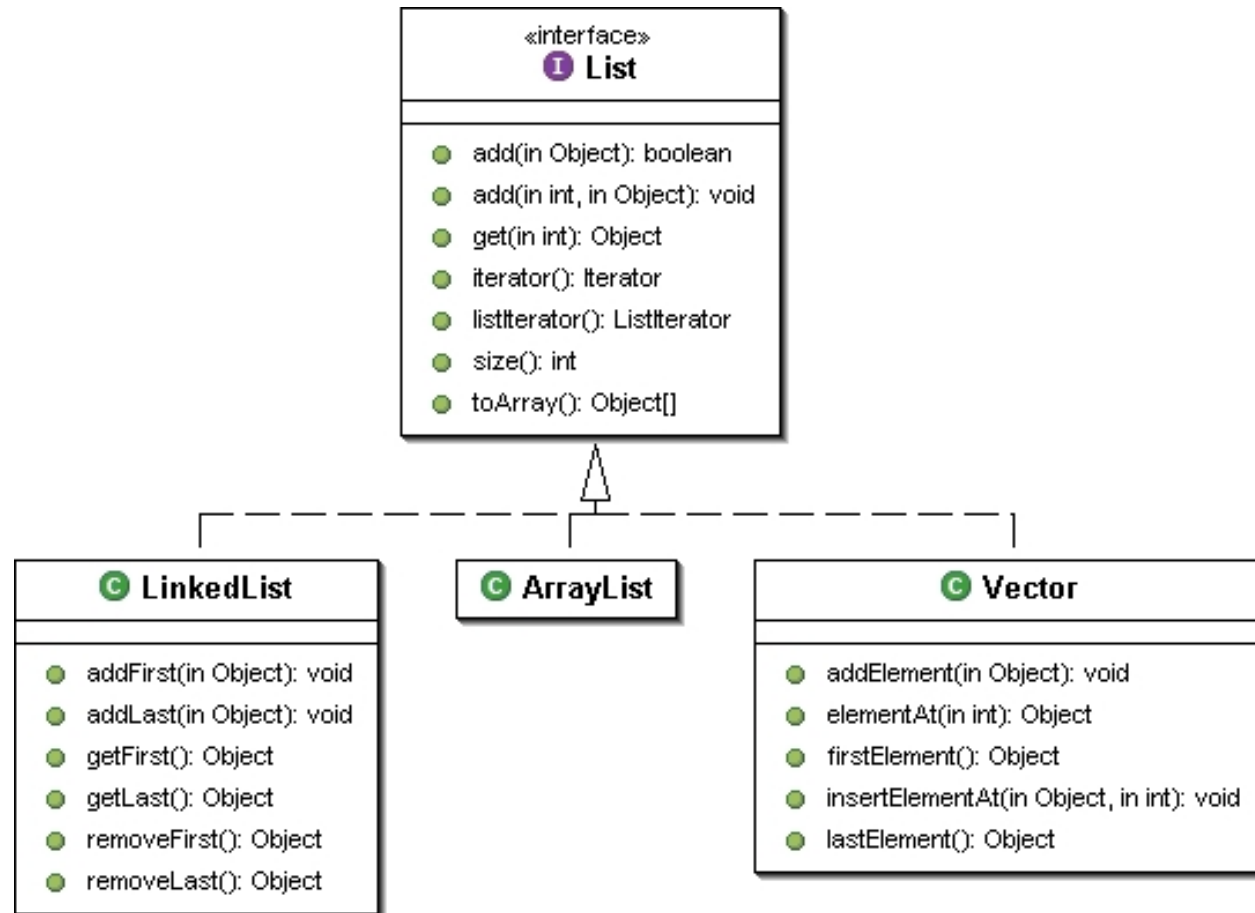
## ► Interfaces





# Collections API

## ► Implementações





# Collections API

## ► Implementações

<b>boolean</b> add(Object)	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna true ou false indicando se a adição foi efetuada com sucesso.
<b>boolean</b> remove(Object)	Remove determinado elemento da coleção. Se ele não existia, retorna false.
<b>int</b> size()	Retorna a quantidade de elementos existentes na coleção.
<b>boolean</b> contains(Object)	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método equals() do objeto, e não através do operador ==.
Iterator iterator()	Retorna um objeto que possibilita percorrer os elementos daquela coleção.





# Collections API

---

- ▶ Listas: java.util.List

- ▶ ArrayList

- ▶ Criando

```
List lista = new ArrayList();
```

- ▶ Adicionando

```
List lista = new ArrayList();  
lista.add("Manoel");  
lista.add("Joaquim");  
lista.add("Maria");
```



# Collections API

---

## ► ArrayList

```
ContaCorrente c1 = new ContaCorrente();  
c1.deposita(100);
```

```
ContaCorrente c2 = new ContaCorrente();  
c2.deposita(200);
```

```
ContaCorrente c3 = new ContaCorrente();  
c3.deposita(300);
```

```
List contas = new ArrayList();  
contas.add(c1);  
contas.add(c3);  
contas.add(c2);
```

```
System.out.println(contas.size());
```



# Collections API

---

## ▶ ArrayList <Generics>

- ▶ Restringir as listas a um determinado tipo de objetos

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();  
contas.add(c1);  
contas.add(c3);  
contas.add(c2);
```



# Collections API

---

## ► ArrayList <Generics> - Percorrendo

```
for(int i = 0; i < contas.size(); i++) {  
    ContaCorrente cc = contas.get(i); // sem casting!  
    System.out.println(cc.getSaldo());  
}
```



# Collections API

---

## ▶ ArrayList <Generics> – Percorrendo

### ▶ *Enhanced For*

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();  
for(ContaCorrente c: contas){  
    System.out.println(c.getSaldo());  
}
```



# Collections API

---

## ► Ordenando

### ► Método *sort()*

```
List<String> listaNomes = new ArrayList<String>();  
listaNomes.add("Julia");  
listaNomes.add("Maria");  
listaNomes.add("Ana");  
listaNomes.add("Bia");
```

```
Collections.sort(listaNomes);
```

```
for (String nome : listaNomes) {  
    System.out.println(nome);  
}
```

```
Collections.sort(listaNomes, Collections.reverseOrder());
```



# Collections API

## ► Ordenando

### ► Método *sort()*

► *Como ordenar uma lista de contas?*

Conta
- numero : int
- nomeCliente : String
- saldo : double
- limite : double

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();  
contas.add(c1);  
contas.add(c3);  
contas.add(c2);
```



# Collections API

---

## ► Ordenando

### ► Método *sort()*

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

```
    public int compareTo(ContaCorrente outraConta) {  
        if(this.saldo > outraConta.saldo){  
            return 1;  
        }else if(this.saldo < outraConta.saldo){  
            return -1;  
        }else{  
            return 0;  
        }  
    }  
}
```

```
Collections.sort(contas);
```





## Exercício 09

---

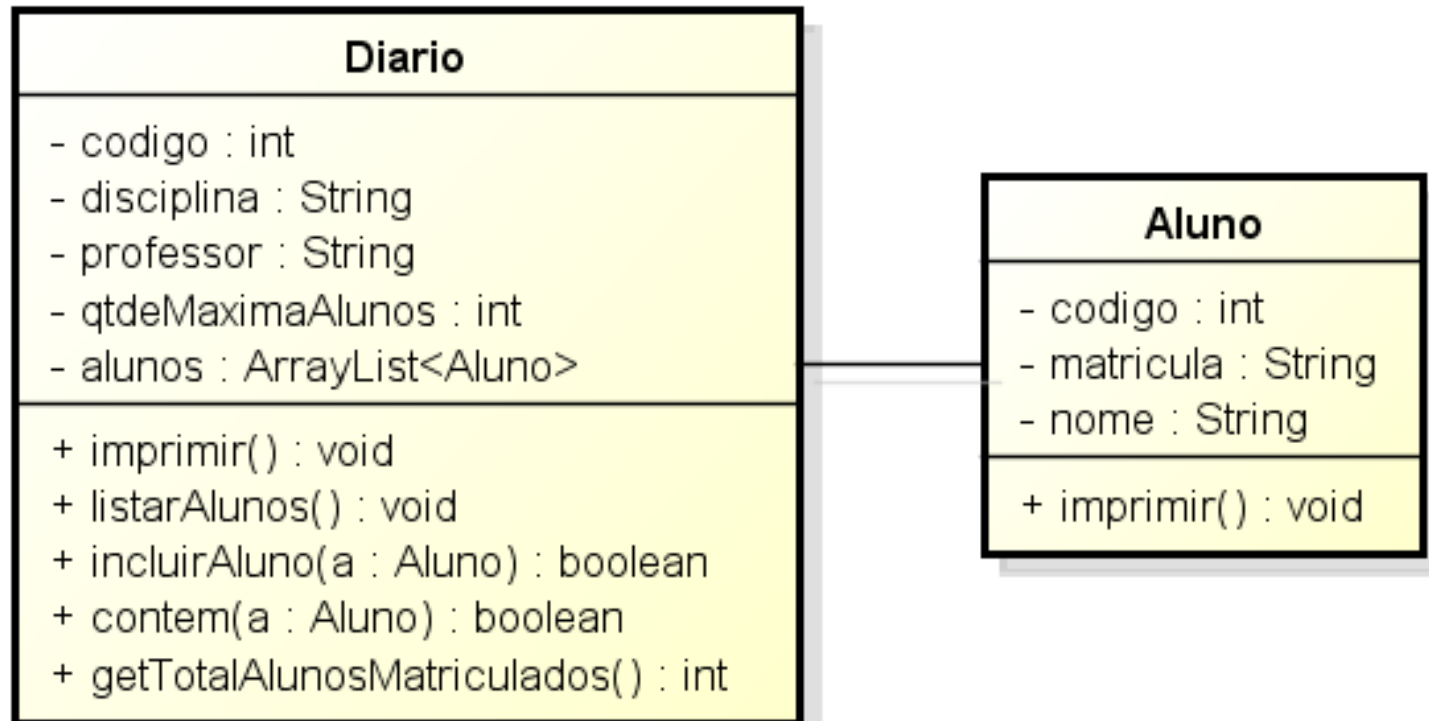
User
<ul style="list-style-type: none"><li>- codigo : int</li><li>- nome : String</li><li>- amigos : ArrayList&lt;String&gt;</li></ul>
<ul style="list-style-type: none"><li>+ imprimir() : void</li><li>+ listarAmigos() : void</li><li>+ adicionarAmigos(nome : String) : boolean</li><li>+ possuiAmigo(nome : String) : boolean</li></ul>

powered by Astah

- ▶ Crie uma coleção de objetos do tipo User
- ▶ Ordene a coleção pelo nome do usuário e imprima o resultado



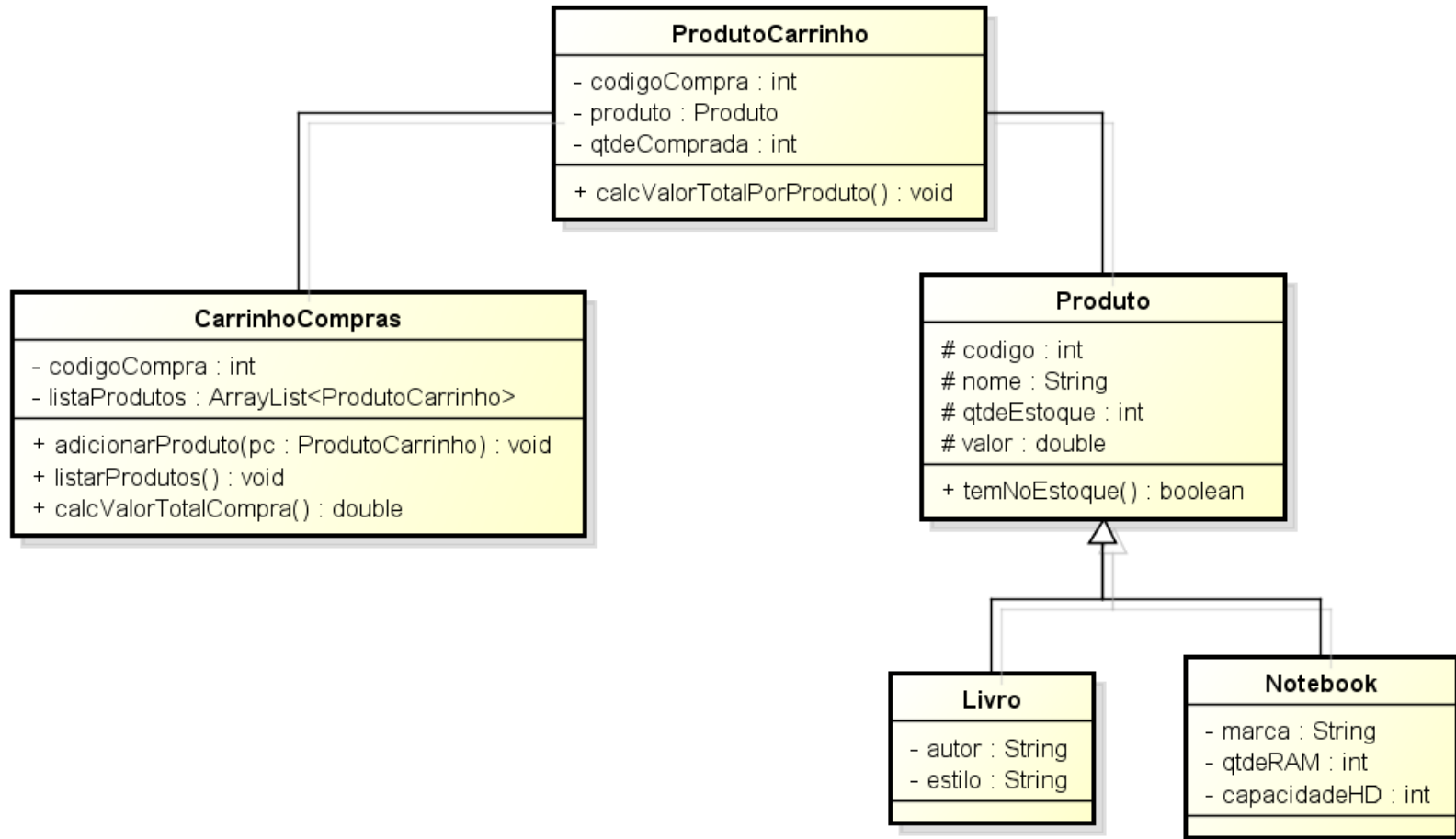
## Exercício 10



powered by Astah

# Exercício 11

## ► HERANÇA





# Polimorfismo – Classes Abstratas

## ► Modificador: *abstract*

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
abstract	<p>O elemento é virtual e <b>deve ser redefinido em sub-classes</b>.</p> <p>Se uma classe possuir um método declarado como “abstract”, ela deve ser declarada como “abstract”.</p> <p><b>Você não pode implementar o método abstract em uma classe abstrata, mas deve implementá-lo em qualquer sub-classe.</b></p> <p>A classe abstrata não pode ser instanciada</p>	X	X	-----	-----



# Modificadores

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
Final	<p>Significa que o elemento não pode ser alterado.</p> <p>Uma classe “final” não pode ter sub-classes (não pode ser herdada).</p> <p>Um método “final” não pode ser redefinido (sobrescrito) e uma variável “final” não pode ser alterada.</p>	X	X	X	_____



# Modificadores

---

## ► Classe *final*

- Uma classe “final” não pode possuir sub-classes. É o contrário de uma classe Abstrata.

```
public final class carro {  
}
```

```
public class onibus extends carro {  
}
```



# Modificadores

- ▶ **Método *final***
- ▶ Um método “final” não pode ser redefinido (override). Redefinir um método é reescrevê-lo em sub-classes. Por exemplo:

```
public class veiculo {  
    protected boolean ligado;  
    public final boolean ligar() {  
        ligado = true;  
        return ligado;  
    }  
}
```

```
public class carro extends veiculo {  
    public boolean ligar() {  
        ligado = true;  
        return ligado;  
    }  
}
```



# Modificadores

## ► Variáveis *final*

- São atributos de uma classe que não mudam de valor. O modificador *final* indica que o atributo é imutável

```
public class Constante {  
    static final double PI = 3.14159265;  
}
```

```
class teste {  
    public static void main( String[] args ) {  
        System.out.println(Constante.PI);  
        Constante.PI = 0;  
    }  
}
```





# Modificadores

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
Static	<p>Se aplicado a uma variável, significa que ela pertence à classe e não à instância do Objeto.</p> <p>Se aplicado a um método, este passa a ser também da Classe e somente pode acessar suas variáveis Estáticas.</p> <p>Se aplicado a um trecho de código, este será executado no momento em que a Classe for carregada pelo JVM.</p>	_____	X	X	X



# Modificadores

---

## ▶ **Static**

### ▶ Atributos Estáticos

- ▶ Atributos estáticos não precisam de uma instância da classe para serem usados.
  - Eles podem ser acessados diretamente
- ▶ Eles são compartilhados por todas as instâncias da classe (cuidado ao usá-los)
  - Como se fossem variáveis globais



# Modificadores

## ► *Static*

### ► Atributos Estáticos

```
class Contador {  
    static int count = 0;  
    void incrementar() {  
        count++;  
    }  
}
```

```
class TestandoContador_1{  
    public static void main( String[] args ) {  
        System.out.println("Contador: " + Contador.count);  
        System.out.println(Contador.count++);  
  
        Contador c1 = new Contador();  
        System.out.println(c1.count);  
  
        Contador c2 = new Contador();  
        System.out.println(c2.count);  
  
    }  
}
```



# Modificadores

---

## ▶ ***Static***

### ▶ Métodos Estáticos

- ▶ Não precisam de uma instância da classe para serem usados .
- ▶ Métodos estáticos NÃO podem chamar métodos não-estáticos sem uma instância.



# Modificadores

## ► *Static*

```
class MetodoEstatico {  
    public static void main( String[] args ) {  
        MetodoEstatico me = new MetodoEstatico();  
        me.metodoNaoEstatico();  
        me.metodoEstatico();  
        MetodoEstatico.metodoEstatico();  
        metodoEstatico();  
    }  
  
    static void metodoEstatico() {  
        //metodoNaoEstatico(); //ERRADO  
        // (new MetodoEstatico()).metodoNaoEstatico(); //OK  
    }  
  
    void metodoNaoEstatico() {  
        metodoEstatico(); //OK  
    }  
}
```