




# CURSO DE PROGRAMAÇÃO EM JAVA

Aula 12   
Classes e objetos: um  
exame mais  
profundo

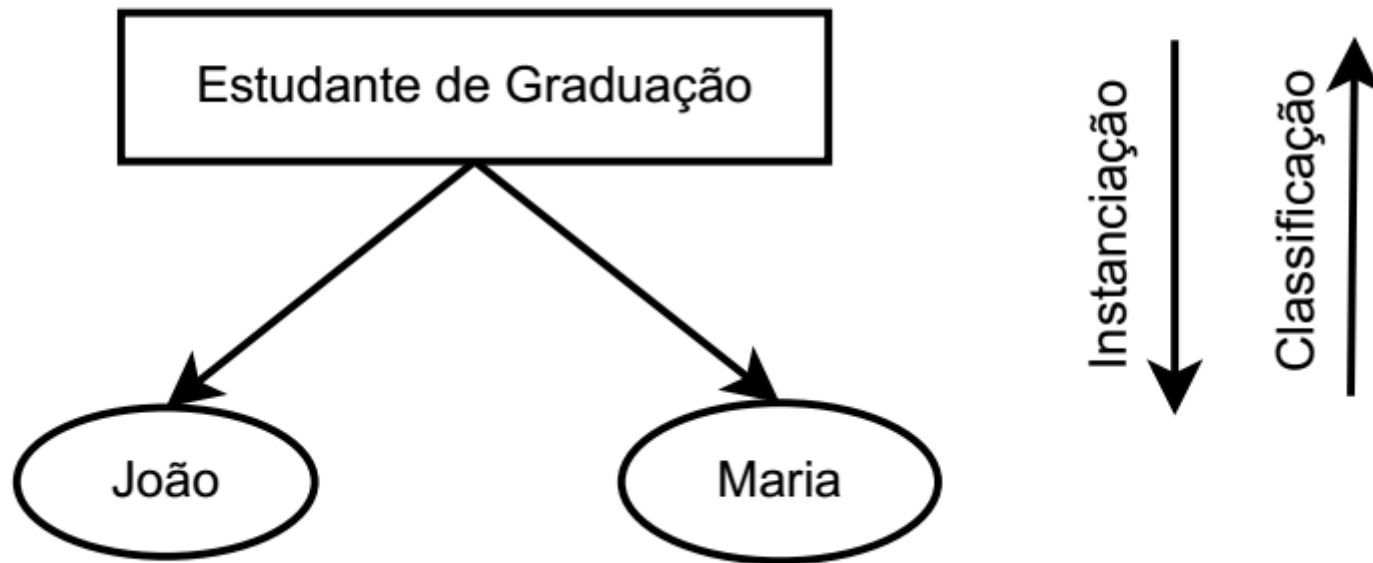
1.

Revisão

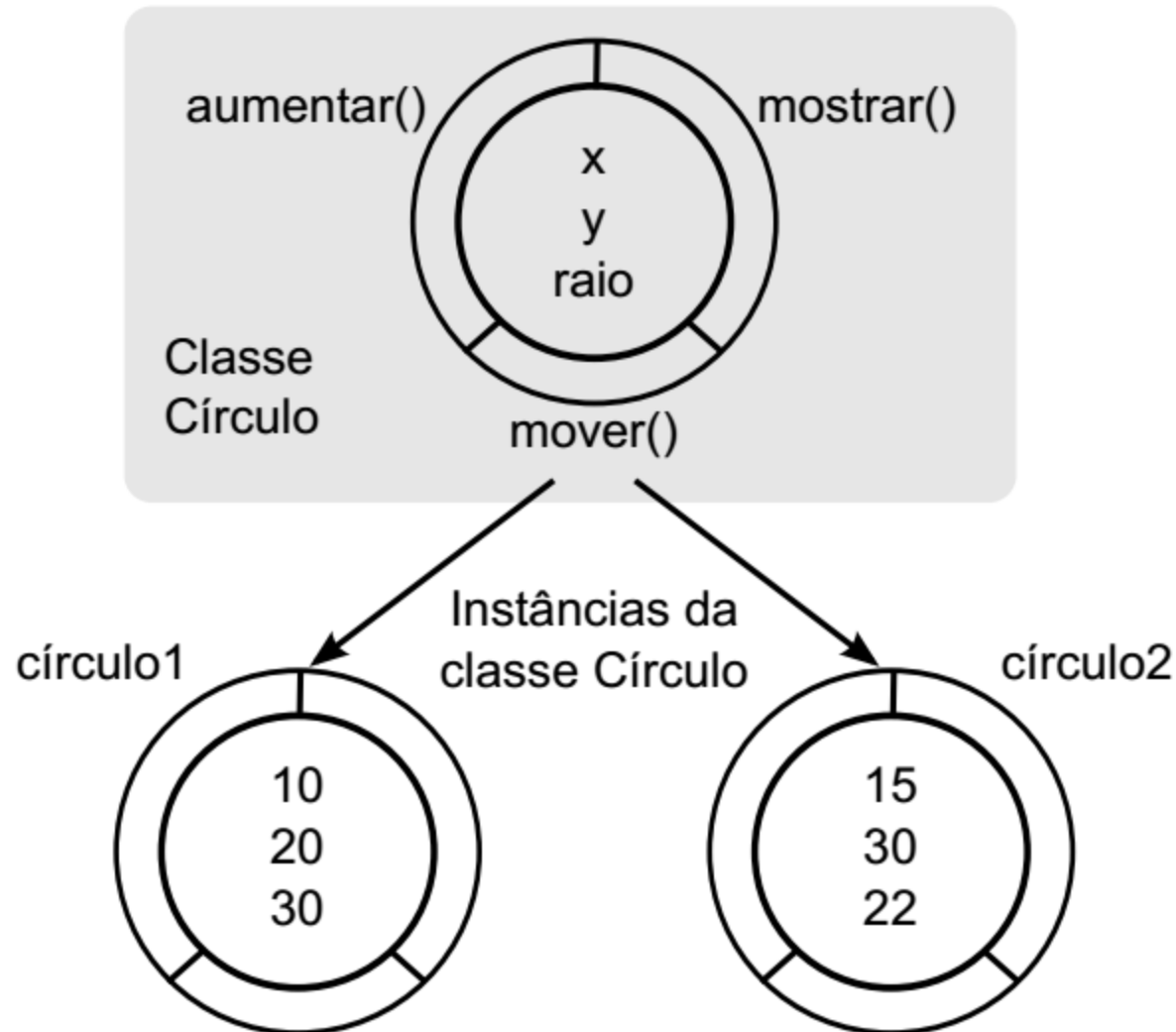
## ■ Conceitos:

- Classe: categoria de objetos, abstração (Ex.: Pessoa)
- Objeto: elemento concreto de uma classe, instância (Ex.: o Pelé)
- Atributo: propriedade de uma classe (Ex.: altura)
- Método: comportamento de uma classe (Ex.: chutar)

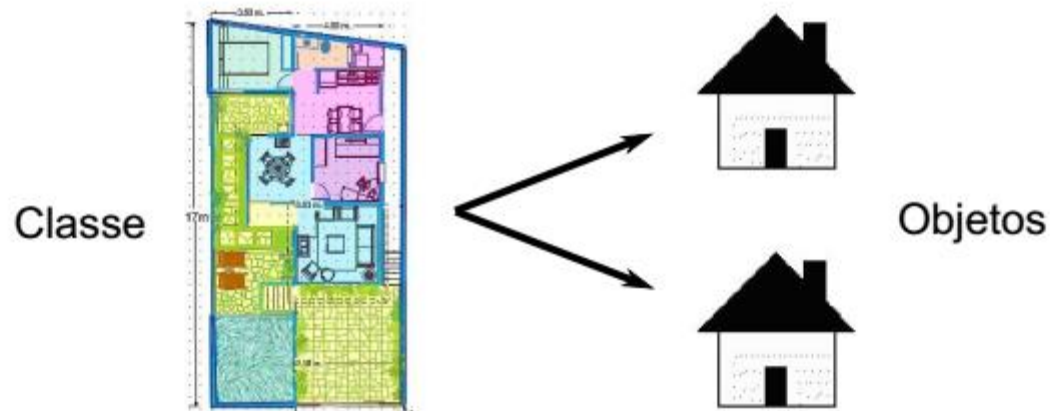
- Um objeto é um indivíduo de uma classe
- Instanciação: processo de criação de um indivíduo



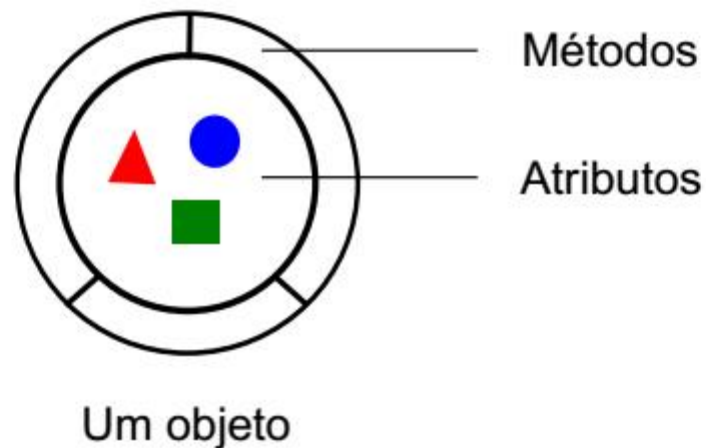
- Classe: como se fosse uma *forma* para criação de objetos



- Objetos são concretizações de uma classe



- Objetos de uma classe têm métodos e atributos



- Instanciação – ato de criar um objeto a partir de uma classe
- Ponto de vista computacional – alocação de memória
  - Reserva de uma porção de memória para guardar valores dos atributos que descrevem um objeto de uma certa classe

- Métodos – implementação de processos disponibilizados pelos objetos (instâncias) da classe
- Permitem que objetos de uma classe realizem tratamento de dados, cálculos, comunicação com outros objetos e todo procedimento necessário

Carro
<pre>-cor: String -ano: String -combustível: String -airBag: boolean</pre>
<pre>+ligar() +deligar() +acelerar(intensidade:int) +frear(intensidade:int) +virar(direcao:int) -checarNivelCombustivel() -injetarCombustivel(qtde:int)</pre>

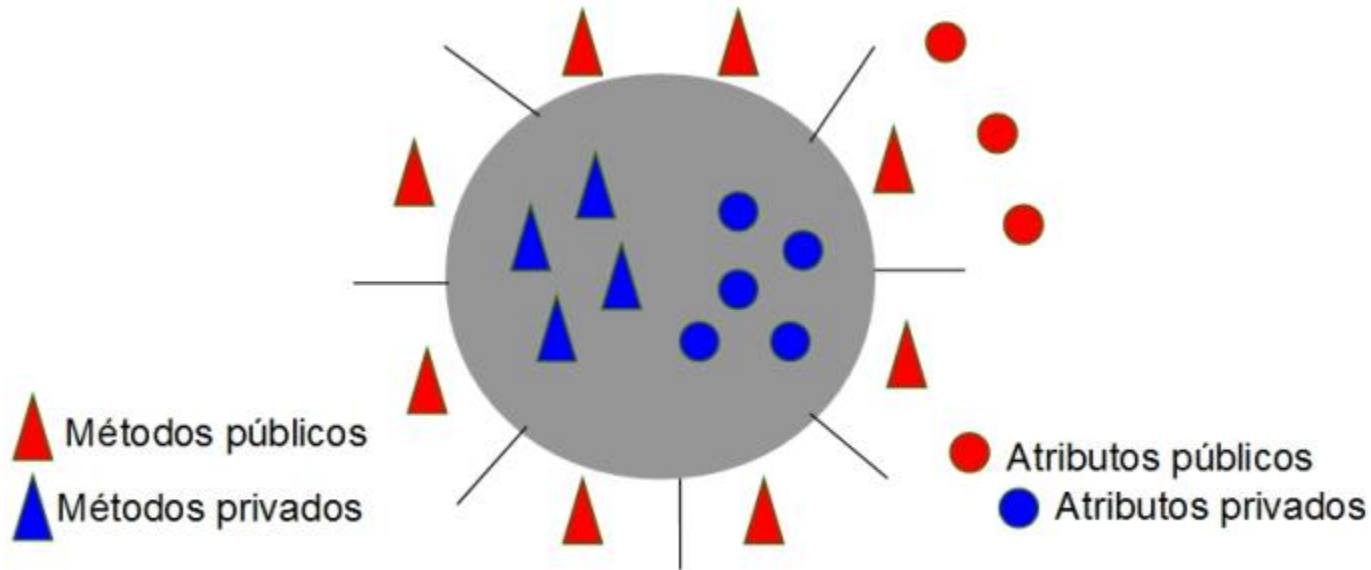


2.

# Encapsulamento

- Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos
- A linguagem Java fornece mecanismos de controle de acessibilidade (visibilidade)
- Implementado através dos modificadores de acesso – palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos
- Modificadores de acesso: public, private, protected, friendly (default)

- Os dois modificadores de acesso extremos são:
  - 1 **public**: permite acesso a partir de qualquer classe. O elemento é visível a partir de qualquer classe
  - 2 **private**: permite acesso apenas na própria classe. O elemento é visível apenas dentro da classe onde está definido



- Mas então como acessar atributos, ao menos para consulta (leitura)?
- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar dois métodos *acessores*
- Os dois métodos são definidos na própria classe onde o atributo se encontra
  - 1 Método que acessa valor – `getXXXXX()`, onde *XXXXXX* é o nome do atributo
  - 2 Método que altera valor – `setXXXXX(tipo parametro)`, onde *XXXXXX* é o nome do atributo e *tipo* é o tipo do atributo

- Métodos `get/set` – cuidado para não quebrar o encapsulamento
- Se uma classe faz `objeto.getAtrib()`, manipula o valor do atributo e depois faz `objeto.setAtrib()`, o atributo é essencialmente público
- Mesmo assim, melhor do que atributo público...

# Exemplo

```
class Conta {  
    private double limite;  
    private double saldo;  
    public double getSaldo() {  
        return saldo;  
    }  
    public void setSaldo(double x) {  
        saldo = x;  
    }  
    public double getLimite() {  
        return limite;  
    }  
    public void setLimite(double y) {  
        limite = y;  
    }  
}
```

# Exemplo

```
class Banco {  
    public static void main(String args[]) {  
        Conta c1 = new Conta();  
        c1.setSaldo(1000000);  
        c1.setLimite(1000000);  
    }  
}
```

# Exemplo - versão 2

```
class Conta {  
    private double saldo = 0;  
    private double limite;  
    public void deposita(double x) {  
        saldo = saldo + x;  
    }  
    public void saca(double x) {  
        if ( saldo + limite >= x )  
            saldo = saldo - x;  
    }  
    public double getSaldo() {  
        return saldo;  
    }  
}
```



# Exemplo - versão 2

```
class Banco {  
    public static void main(String args[]) {  
        Conta c1 = new Conta();  
        c1.deposita(500);  
        c1.saca(200);  
    }  
}
```

3.

Inicializando objetos com  
construtores

# Construtores



São os responsáveis por criar o objeto em memória, ou seja, instanciar a classe que foi definida.



Possuem o mesmo nome que a classe.



Pode existir mais de um construtor em uma classe, porém com diferentes parâmetros.



O Java já cria um construtor default para nós.



Na declaração do Objeto o new é o responsável de chamar o construtor.



É o valor default dos seus objetos, do mesmo modo que 0 é o valor default para int.



Não possui um tipo de retorno.

# Exemplos

```
public class Carro{  
  
    /* CONSTRUTOR DA CLASSE Carro */  
    /*modificadores de acesso (public nesse caso) + nome da classe (Carro  
nesse caso) + parâmetros (nenhum definido neste caso).*/  
  
    public Carro(){  
        //Faça o que desejar na construção do objeto  
    }  
}  
  
public class Aplicacao {  
    public static void main(String[] args) {  
        //Chamamos o construtor sem nenhum parâmetro  
        Carro fiat = new Carro();  
    }  
}
```

# Exemplos

```
public class Carro{  
  
    private String cor;  
    private double preco;  
    private String modelo;  
  
    /* CONSTRUTOR PADRÃO */  
    public Carro(){  
  
    }  
  
    /* CONSTRUTOR COM 2 PARÂMETROS */  
    public Carro(String modelo, double preco){  
        //Se for escolhido o construtor sem a COR do veículo  
        // definimos a cor padrão como sendo PRETA  
        this.cor = "PRETA";  
        this.modelo = modelo;  
        this.preco = preco;  
    }  
}
```

# Exemplos

```
/* CONSTRUTOR COM 3 PARÂMETROS */
public Carro(String cor, String modelo, double preco){
    this.cor = cor;
    this.modelo = modelo;
    this.preco = preco;
}
}
public class Aplicacao {
    public static void main(String[] args) {
        //Construtor sem parâmetros
        Carro prototipoDeCarro = new Carro();

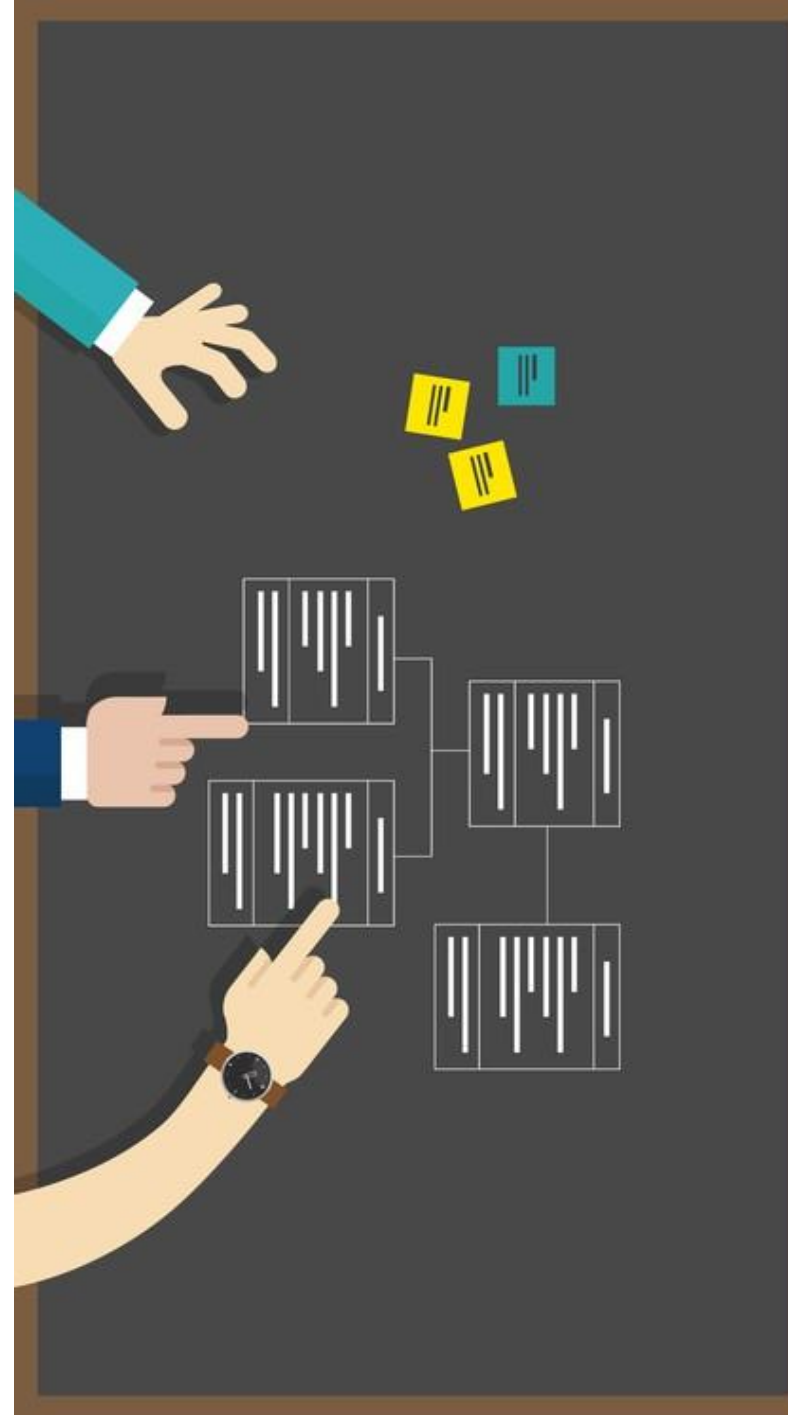
        //Construtor com 2 parâmetros
        Carro civicPreto = new Carro("New Civic", "40000");

        //Construtor com 3 parâmetros
        Carro golfAmarelo = new Carro("PRATA", "Golf", "38000");
    }
}
```

4.  
this

# this

A palavra reservada **this**, no corpo do nosso novo construtor, significa que nós estamos acessando um membro ou um atributo do próprio objeto que está sendo instanciado.





# Exemplo:

```
public class Funcionario {  
    private String nome;  
    private int ID;  
    private double salario;  
  
    public Funcionario( String nome, int ID, double salario){  
        this.nome = nome;  
        this.ID = ID;  
        this.salario = salario;  
    }  
}
```

- ❑ Sempre que colocarmos 'this.' antes de uma variável, fica implícito ao Java que estamos nos referindo aos atributos daquela Classe.

5.

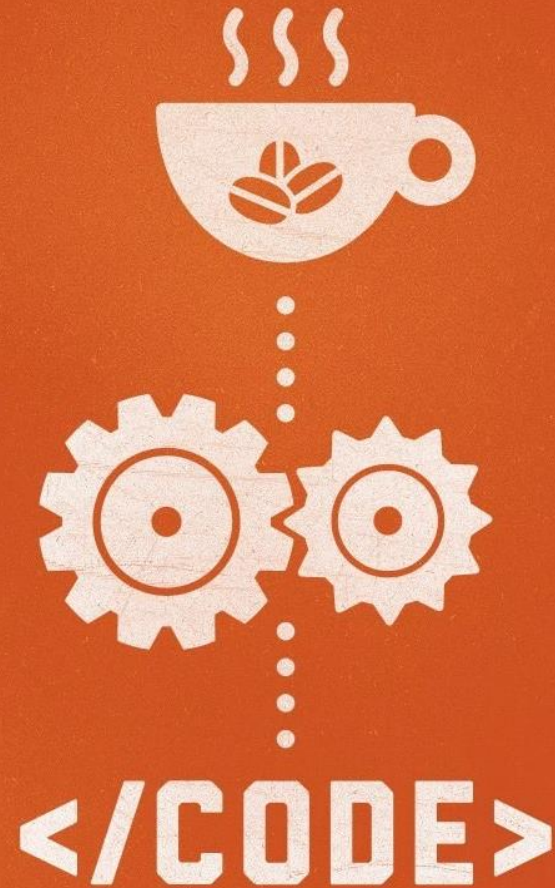
Herança

# Herança

A herança é um mecanismo da Orientação a Objeto que permite criar novas classes a **partir de classes já existentes**, aproveitando-se das características existentes na classe a ser estendida.

Este mecanismo é muito interessante, pois promove um grande **reuso** e **reaproveitamento** de código existente.

# PROGRAMMER

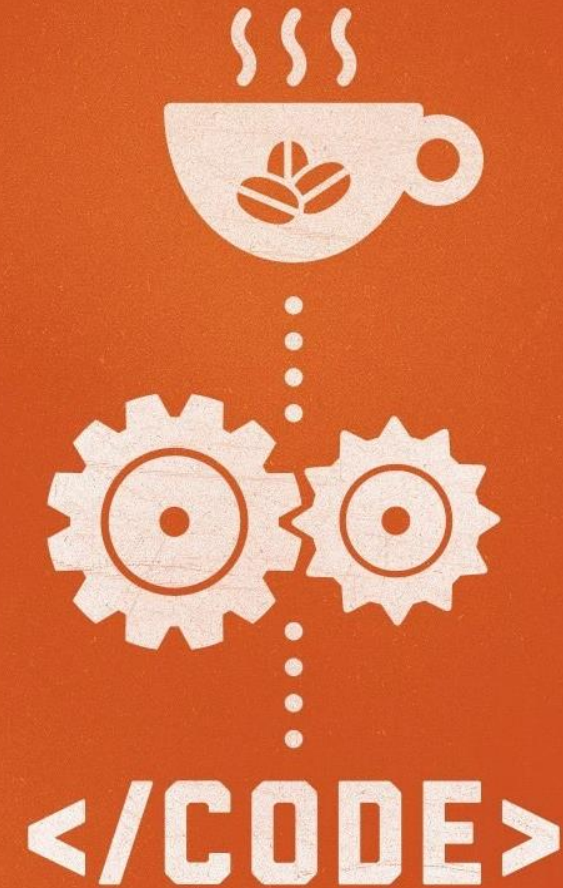


# Herança

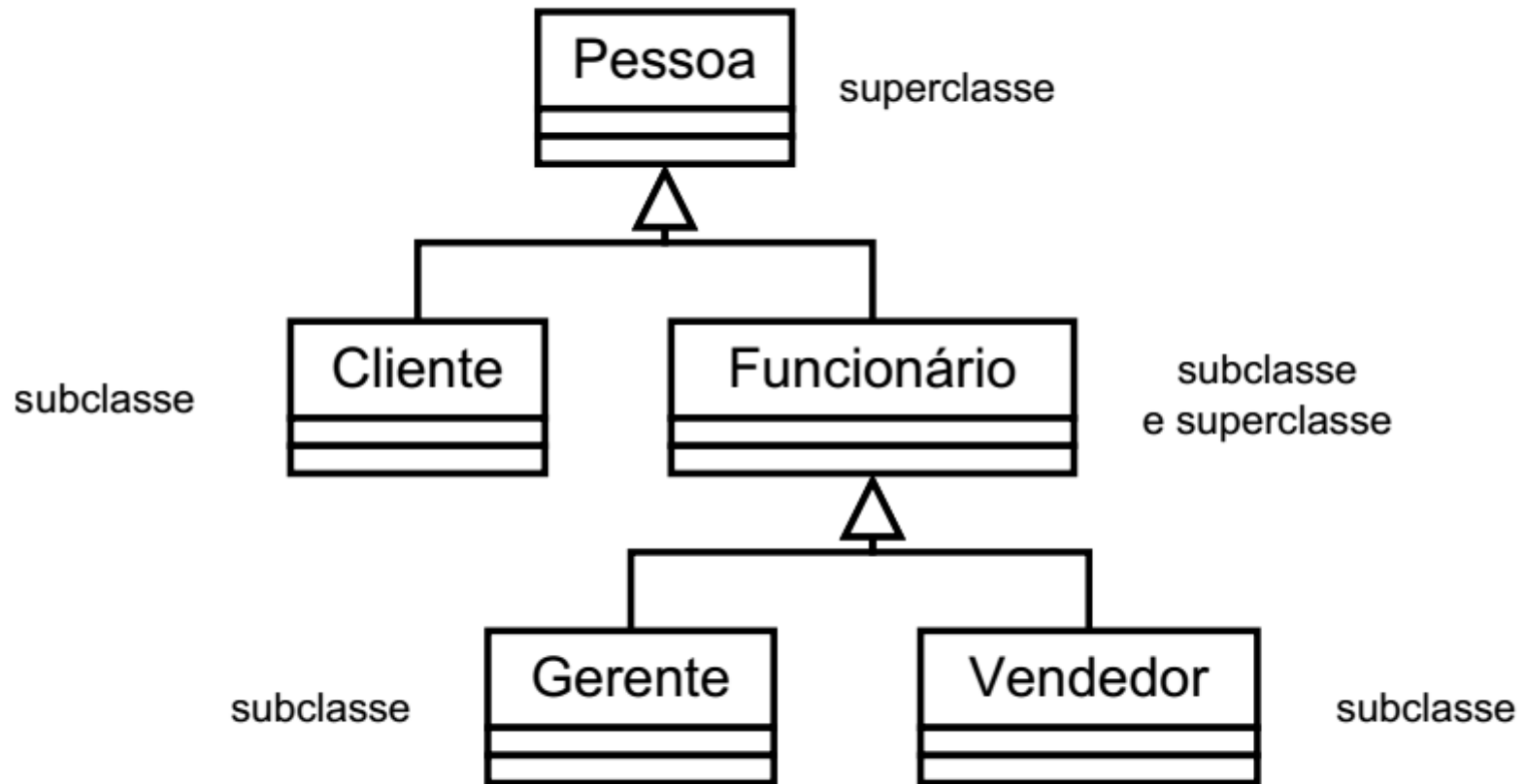
Com a herança é possível criar **classes derivadas**, **subclasses**, a partir de classes bases, **superclasses**. As **subclasses** são mais **especializadas** do que as suas **superclasses**, mais **genéricas**. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos.

A linguagem Java permite o uso de herança simples, mas não permite a implementação de herança múltipla.

# PROGRAMMER



# Herança - Hierarquia de Classes



# Herança - Hierarquia de Classes

- Relacionamentos de herança: estrutura parecida com uma árvore
- Cada classe é:
  - Superclasse: Oferece dados/comportamentos para outras classes
  - Subclasse: Herda dados/comportamentos de outras classes

# Membros Protected

- Acesso protegido
- Nível de acesso intermediário entre public private
- Membros protected acessíveis por
  - Membros da superclasse
  - Membros da subclasse
  - Membros de classes no mesmo pacote
- Subclasse acessa membros da superclasse
  - Palavra reservada super e um ponto (.)

# Exemplo

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe Funcionário:

```
class Funcionario {  
    String nome;  
    String cpf;  
    double salario;  
    // métodos devem vir aqui  
}
```



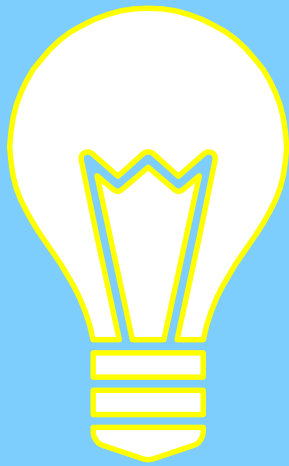
# Exemplo

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes.

```
class Gerente {
    String nome;
    String cpf;
    double salario;
    int senha;
    int numeroDeFuncionariosGerenciados;

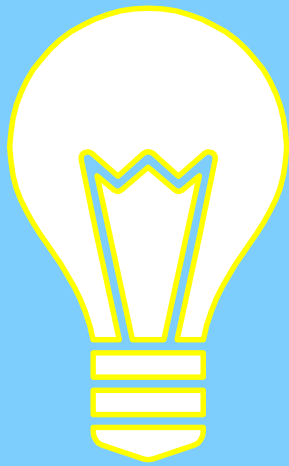
    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // outros métodos
}
```

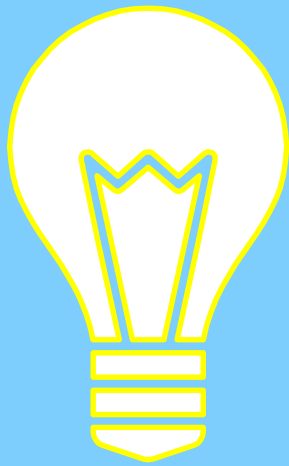


## Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe Funcionario mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios



Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe Gerente tem o método autentica, que não faz sentido existir em um funcionário que não é gerente.



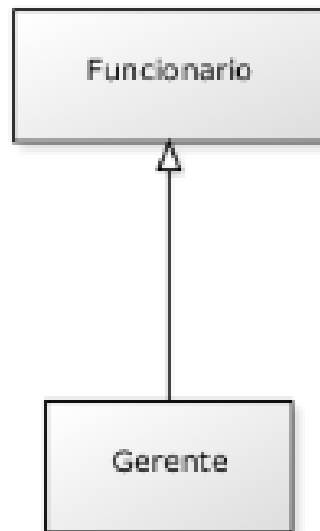
Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma **relação de classe mãe e classe filha**. No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem, gostaríamos que ela fosse uma extensão de Funcionario. Fazemos isto através da palavra-chave **extends**.

# Exemplo

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public boolean autentica(int senha) {  
        if (this.senha == senha) {  
            System.out.println("Acesso Permitido!");  
            return true;  
        } else {  
            System.out.println("Acesso Negado!");  
            return false;  
        }  
    }  
}  
  
// setter da senha omitido  
}
```

# Exemplo

Em todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos definidos na classe Funcionario, pois um Gerente é um Funcionario:



```
class TestaGerente {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();

        // podemos chamar métodos do Funcionario:
        gerente.setNome("João da Silva");

        // e também métodos do Gerente!
        gerente.setSenha(4231);
    }
}
```

# Exemplo

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de `Funcionario` public, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o `protected`, que fica entre o `private` e o `public`.

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
    // métodos devem vir aqui  
}
```

# Reescrita de método

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%.

Vamos ver como fica a classe Funcionario:

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.10;  
    }  
    // métodos  
}
```



# Reescrita de método

Se deixarmos a classe Gerente como ela está, ela vai herdar o método getBonificacao, no entanto ele terá bonificação de 10% como os funcionários comuns.

No Java, quando herdamos um método, podemos alterar seu comportamento. Para consertar isso, podemos **reescrever** este método:

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // ...  
}
```

# Invocando o método reescrito

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porém adicionando R\$ 1000.

O Java nos permite resolver isso de uma maneira simples, o `getBonificacao` do Gerente pode chamar o do Funcionario utilizando a palavra-chave **super**.

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    // ...  
}
```

Obs: Podemos mais chamar o método antigo apenas dentro da subclasse.

6.

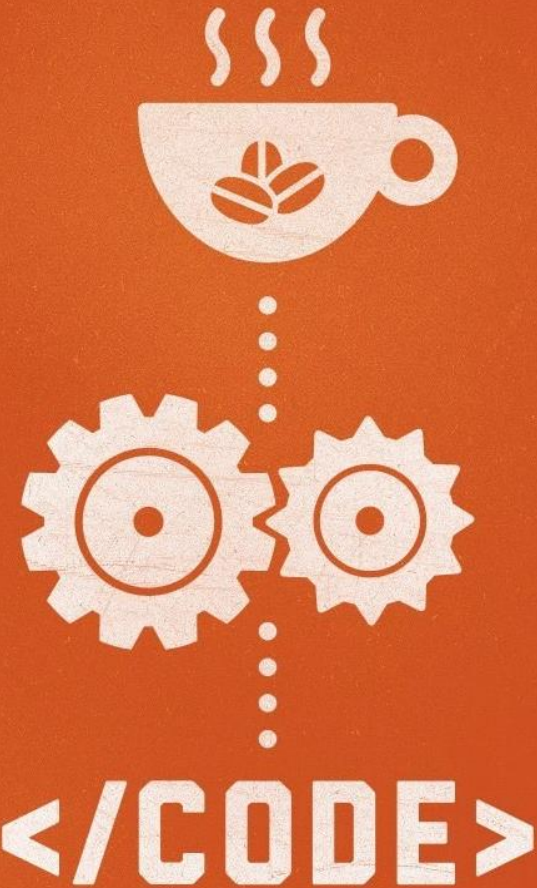
Classe Object

## Classe Object

**Object** é a raiz da hierarquia de classes do Java, a superclasse de todas as classes, direta ou indiretamente.

Sendo a base para todas as classes, **Object** define alguns comportamentos comuns que todos objetos devem ter, como a habilidade de serem comparados uns com os outros, utilizando **equals()**, poderem ser representados como texto, com o método **toString()**, e possuírem um número que identifica suas posições em coleções baseadas em hash, com o **hashCode()**.

# PROGRAMMER





# DESAFIO

E aí, vamos praticar?

# Exercício 1

O valor de  $xy$  pode ser calculado como sendo  $x$  multiplicado por si mesmo  $y$  vezes (se  $y$  for inteiro).

Escreva uma classe chamada "SeriesMatemáticas" que contenha um construtor para inicializar  $x$  e  $y$ , um método chamado "elevadoA" que calcule e retorne o resultado de  $xy$ , e um método chamado "imprimeResultado" que mostre o resultado obtido. Obs: Use o comando while.

**Entrada:**

4  
7

**Saída:**

16384

# Exercício 2

Acrescente a classe "SeriesMatematicas" o método "piQuadradoSobre8" que calcule a série  $(1/1^2) + (1/3^2) + (1/5^2) + (1/7^2) + (1/9^2) + \dots$ . Evidentemente a série não poderá ser calculada infinitamente, devendo parar depois de N termos, sendo que o valor de N deve ser fornecido como parâmetro ao método. Obs: Use o comando do-while.

**Entrada:**

67

**Saída:**

1.2300

# Exercício 3

Considere, como subclasse da classe Pessoa, a classe Empregado. Considere que cada instância da classe Empregado tem, para além dos atributos que caracterizam a classe Pessoa, os atributos `codigoSetor` (inteiro), `salarioBase` (vencimento base) e `imposto` (porcentagem retida dos impostos).

Implemente a classe Empregado com métodos seletores e modificadores e um método `calcularSalario`.

Escreva um programa de teste adequado para a classe Empregado.



# Exercício 4

Implemente a classe Administrador como subclasse da classe Empregado. Um determinado administrador tem como atributos, para além dos atributos da classe Pessoa e da classe Empregado, o atributo ajudaDeCusto (ajudas referentes a viagens, estadias, ...).

Note que deverá redefinir na classe Administrador o método herdado calcularSalario (o salário de um administrador é equivalente ao salário de um empregado usual acrescido da ajuda de custo).

Escreva um programa de teste adequado para esta classe.

# Obrigado!

## **Alguma pergunta?**

Você pode me contatar em:  
[ywassef@hotmail.com](mailto:ywassef@hotmail.com)