

Programação Orientada a Objetos

Unidade 2 – Herança





Bibliografia

- Beginning Java 2 – Ivor Horton – 1999 WROX
- Java2 – The Complete Reference – 7th Edition – Herbert Schildt – Oracle Press
- Core Java Fundamentals – Horstmann / Cornell – PTR- Volumes 1 e 2 – 8th Edition
- Inside the Java 2 – Virtual Machine Venners – McGrawHill
- Understanding Object-Oriented Programming with JAVA – Timothy Budd – Addison Wesley
- Head First Java, 2nd Edition by Kathy Sierra and Bert Bates
- Effective Java, 2nd Edition by Joshua Bloch
- Thinking in Java (4th Edition) by Bruce Eckel
- Java How to Program - 9th Edition by Paul Deitel and Harvey Deitel



Introdução

- ❖ Por meio de **herança**, podemos criar **novas** classes a partir de classes **existentes**.
- ❖ Isto permite o **reuso** de métodos e atributos das classes existentes.
- ❖ Na classe **nova**, pode-se também criar **novos métodos** e campos para adaptar à novas situações.
- ❖ Esta técnica é de extrema importância na Linguagem Java e também na **Programação Orientada a Objetos**.



Usando classes existentes

- ⊗ Este procedimento é conhecido por **derivação**.
- ⊗ A classe nova é chamada classe derivada ou subclasse.
- ⊗ A classe existente é chamada base ou superclasse.

Um gerente é um
funcionário comum em
uma Empresa ?



Gerentes e Empregados certamente têm muitas coisas em comum ...



Ambos têm um salário ...



Ambos têm código funcional ...



Ambos têm dados pessoais...



Mas, gerentes têm algo a mais...



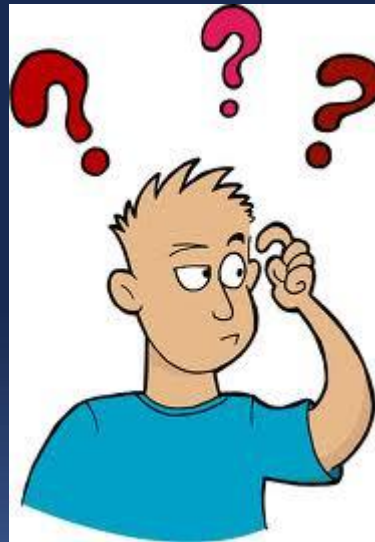
Todo gerente é um empregado...



- ❖ Esta é uma situação **típica** do uso de **herança**;
- ❖ Precisamos definir uma **nova classe** – **Gerente** – e adicionar a ela funcionalidades;
- ❖ Mas, podemos aproveitar o que já está definindo na classe **Empregado**;
- ❖ Os **atributos** e **métodos** da Classe **Empregado** são aproveitados para a classe **Gerente**;
- ❖ Há um relacionamento “**is-a**” entre **Gerente** e **Empregado**;
- ❖ Ou seja, **todo Gerente também é um Empregado**.

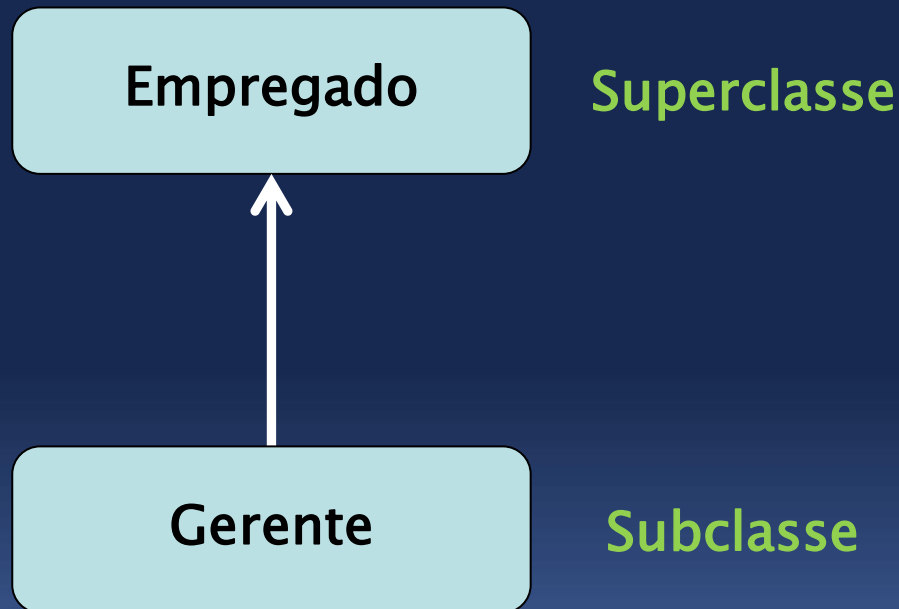


Como definir Herança em Java ?



Por meio da keyword **extends**

- ❏ A keyword **extends** indica que se está criando uma nova classe a partir de outra classe já existente.



Sem herança

Empregado

nome:	String
salario:	double
codfunc:	int

Imprime_Func()
GetDetalhes()

Gerente

nome:	String
salario:	double
codfunc:	int
bonus:	double

GetDetalhes()

Sem herança

```
class Empregado {
    private String nome;
    private double salario;
    private int codfunc;

    public String Get_Detalhes() {
        ...
    }
    public void Imprime_Func() {
        ...
    }
}
```



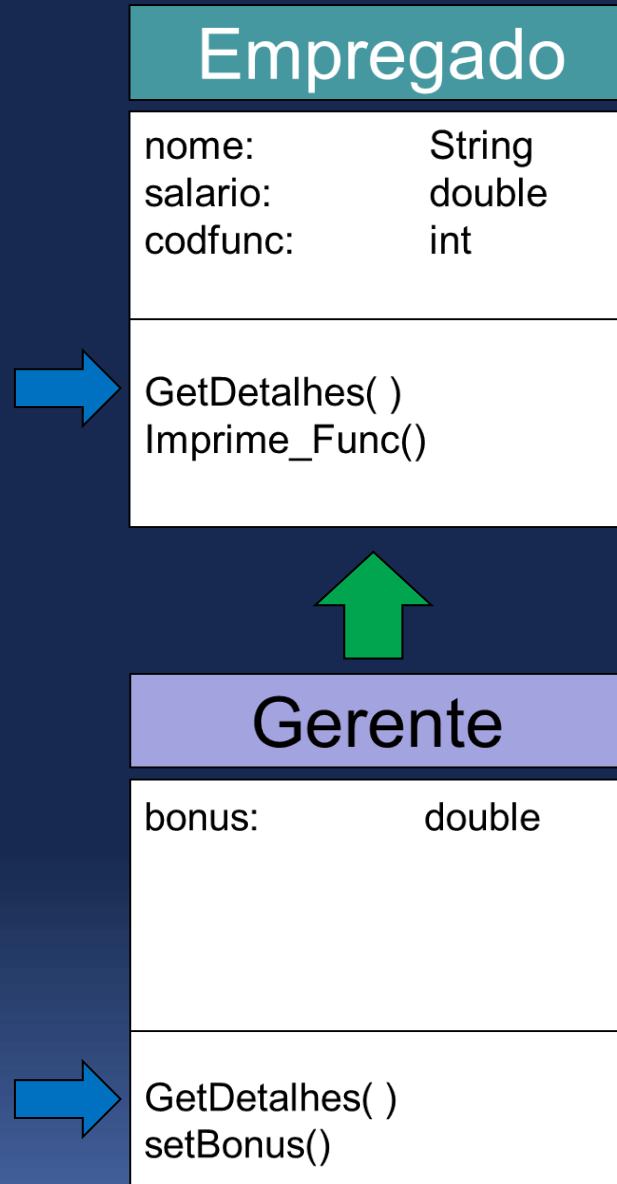


Sem herança

```
class Gerente {  
    private String nome;  
    private double salario;  
    private int codfunc;  
    private double bonus;  
  
    public String Get_Detalhes( ) {  
        ...  
    }  
    public void SetBonus(double b) {  
        bonus = b;  
    }  
}
```



Com herança



Superclasse
Classe pai

Subclasse
Classe filha

Com herança

```
public class Gerente extends Empregado{  
  
    private double bonus;  
  
    public String Get_Detalhes( ) {  
        ...  
    }  
    public void SetBonus(double b) {  
        bonus = b;  
    }  
}
```



Observações

- ❖ A função **Imprime_func()** é da Classe **Empregado**;
- ❖ No entanto, podemos usá-la para objetos do tipo **Gerente**;
- ❖ Esta função é **herdada** da classe **Empregado**;
- ❖ Similarmente, os atributos **nome**, **salario** e **codfunc** são herdados de **Empregado**;
- ❖ Assim, todo gerente tem 4 atributos : **nome**, **salario**, **codfunc** e **bonus**.

Overriding



- ❖ Pode haver funções definidas em **Empregado** que **não** são apropriadas para **Gerente**. Em particular, a função **getSalario()**, pois o cálculo para Gerente é diferente (Gerente tem bonus);
- ❖ Neste caso, definimos um outro método **getSalario()** na classe **Gerente** que sobrepõe a função na superclasse;
- ❖ Nesse caso, a função é reescrita na subclasse;
- ❖ Este conceito é chamado **Overriding**.




Chamada da Superclasse

- ❖ Precisamos indicar para o compilador que desejamos o método `getSalario()` da superclasse e não da classe base.
- ❖ Em Java, isto é indicado pela **keyword** **super**;
- ❖ **super** indica que se está chamando a função do pai, ou seja da **superclasse**.

Keyword super

- ❁ Precisamos indicar para o compilador que desejamos o método `getSalario()` da superclasse e não da classe base.
- ❁ Em Java, isto é indicado pela keyword **super**.

```
public double getSalario() {  
    return (super.getSalario() + this.bonus);  
}
```



Herança de Construtores

- ❖ O construtor de Gerente **não** pode acessar os campos **private** da superclasse (Empregado).
- ❖ Assim, a classe Gerente deve providenciar um **construtor** para inicializar os campos da superclasse.

```
public Gerente(String nome,  
                double salario, int codfunc, double bonus) {  
  
    super(nome, salario, codfunc);  
    this.bonus = bonus;  
}
```

 **super** nesse caso é uma chamada do construtor de Empregado

Keyword **super** no Construtor

```
super(nome, salario, codfunc);
```

- ❖ Aqui a keyword **super** tem um diferente significado.
- ❖ A instrução acima significa uma **chamada** para o **construtor** da **superclasse** (**Empregado**).
- ❖ A chamada **super** deve ser sempre o primeiro **comando**!

Um exemplo...

```
package oop;
```

```
public class TesteGerente02 {
```

```
    public static void main(String[] args) {
```

```
        Gerente x = new Gerente("Paulo",  
                                16700.5, 55320, 2400.0 ); // x é gerente...
```

```
        Empregado[] set = new Empregado[3];
```

```
        set[0] = x;
```

```
        set[1] = new Empregado("Antonietta",  
                                760.5, 49211); // set[1] é empregado
```

```
        set[2] = new Empregado("Aurio",  
                                1220.5, 43678); // set[2] é empregado
```

```
        for (int i=0; i< set.length ; i++)  
            System.out.println (set[i].getNome() +  
                                " " + set[i].getSalario());
```

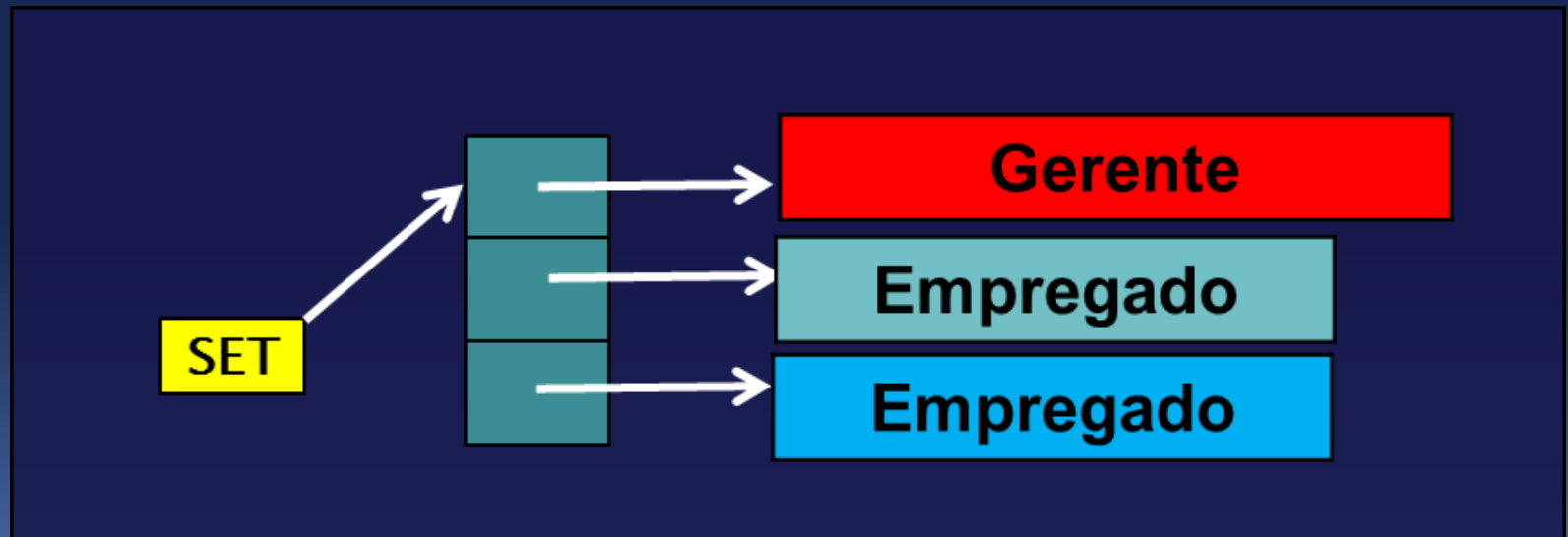
```
    }
```

```
}
```

Um exemplo...

```
Empregado[] set = new Empregado[3];
```

- O tipo declarado de **set** é Empregado, mas o tipo real do objeto para o qual **set** aponta pode ser Gerente ou Empregado.

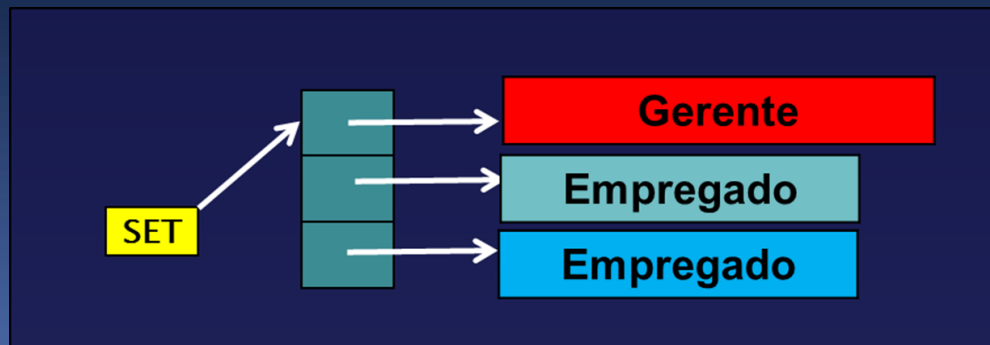


Polimorfismo



```
for (int i=0; i< set.length ; i++)  
    System.out.println (set[i].getNome() + " " + set[i].getSalario());
```

- ❏ `Set[0].getSalario()` efetua chamada do salário de **Gerente**.
- ❏ `Set[1].getSalario()` efetua chamada do salário de **Empregado**.
- ❏ A JVM sabe qual o tipo em tempo de execução e chama o método adequado.
- ❏ Este conceito é chamado **POLIMORFISMO**.



Estrutura de Herança

- ❖ A herança pode se estender em vários níveis.
- ❖ Por exemplo, poderíamos criar uma classe **Executivo** que é filha de **Gerente**.
- ❖ Java **não** suporta múltipla herança. Esta funcionalidade é tratada com o conceito de **interfaces**.

