

## 5 – Polimorfismo

### Sobrecarga (overloading) de métodos:

```
public class x {  
    public void m1( ) {...}  
  
    public void m1 ( int p ) {...}  
}
```



sobrecarga do  
método m1

- Diz-se que o nome de um método foi sobrecarregado (“overloaded”) se dois métodos têm o mesmo nome mas assinaturas diferentes.

Métodos com o mesmo nome e assinaturas diferentes podem:

- ser definidos na mesma classe
- ser herdados por uma dada classe
- um ser herdado e o outro definido na classe.

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

É incorrecto definir

```
public class y {
```

```
    public void m1( ) {...}
```

```
    public void m1( ) {...}
```



*errado*

```
}
```

*Uma classe não pode declarar duas vezes o mesmo método, isto é, dois métodos com a mesma assinatura.*

## **Sobreposição (overriding) de métodos:**

Se uma classe herda um método que “não lhe serve” pode redefinir esse método.

A definição local sobrepõe-se à definição herdada.

Dois métodos

(um numa classe e outro nalguma subclasse da primeira)

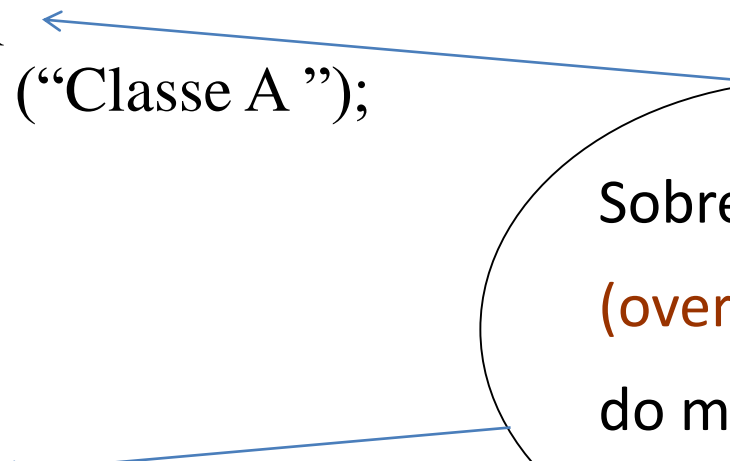
sobrepõem-se se têm a mesma assinatura e (necessariamente) o mesmo tipo de resultado.

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo:

```
public class A {  
    public void m1( ) {  
        System.out.print ("Classe A ");  
    }  
}
```

```
public class B extends A {  
    public void m1( ){  
        System.out.print ("Classe B ");  
    }  
}
```



Sobreposição  
(overriding)  
do método m1

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Qual o output do seguinte programa?

```
public static void main (String args[]) {  
  
    A a = new A();  
  
    B b = new B();  
  
    a.m1();  
  
    b.m1();  
  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Quando redefinimos os métodos clone , equals ou toString da classe Object estamos a sobrepor estes métodos.

*Questão: Perdemos o acesso ao método m1 da classe A?*

Não , “dentro da classe B” podemos aceder ao método m1 da classe A usando a referência super como prefixo.

super.m1()

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo:

Mantendo a classe A suponha a seguinte definição da classe B:

```
public class B extends A{  
    public void m1(){  
        super.m1();  
        System.out.print("classe B ");  
    }  
}
```



**Qual seria agora o output do código anterior?**

...

a.m1();

b.m1();

?

## Ocultação (hiding) de variáveis:

Se uma variável *x* é redefinida numa subclasse essa declaração **oculta (hides)** a variável definida na superclasse.

Exemplo 1:

```
public class ClasseA {  
  
    private int x,y;  
  
    public ClasseA() { x=0;y=0;}
```



## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
public int getX(){
    System.out.println("ClasseA – getX ");
    return x;
}
public int getY() {
    return y;
}
public void setX (int a){
    System.out.println("ClasseA – setX ");
    x=a;
}
public void setY (int b){
    y=b;
}
}
```

# Programação Orientada a Objectos - P. Prata, P. Fazendeiro

```
public class ClasseB extends ClasseA {
```

```
    private int x ;
```

```
    private int z;
```

```
    public ClasseB () {
```

```
        super();
```

*Variáveis  
definidas  
em ClasseB*

```
        x = 100;
```

```
        z = 0;
```

```
    } // fim do construtor
```

oculta (hides)  
a definição de x de  
ClasseA

invoca o construtor  
da superclasse para  
inicializar as  
variáveis definidas  
em ClasseA, x e y

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
public int getZ(){  
    return z;  
}
```

```
public void setZ (int c){  
    z=c;  
}
```

sobreposição (overriding)  
do método getX

```
public int getX(){  
    System.out.println("ClasseB – getX " );  
    return x;  
}
```

Devolve o valor de x  
definido em ClasseB

```
public int getXsuper(){  
    System.out.println("ClasseB – getXsuper " );  
    return super.getX();  
}
```

```
}  
} // fim da classe ClasseB
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
public class Teste{
```

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

```
        ClasseA a = new ClasseA();
```

```
        ClasseB b = new ClasseB();
```

```
        a.setX(10);
```

```
        System.out.println( a.getX()); (1)
```

```
        b.setX(20); // onde está o método setX ???
```

```
        System.out.println(b.getX()); (2)
```

```
        System.out.println(b.getXsuper()); (3)
```

```
    }
```

```
} O output é?
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

ClasseA – setX

ClasseA – getX

10 //valor obtido na instrução (1)

ClasseA – setX // o método setX só está definido na classe ClasseA  
// modifica o x definido na superclasse

ClasseB – getX // obtém o x definido na classe ClasseB

100 //valor obtido na instrução (2)

ClasseB – getXsuper

ClasseA – getX

20 //valor obtido na instrução (3)

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo 2:

```
public class Point {  
    private int x=0, y=0;  
    private int color;  
    public void move (int dx, int dy){  
        x += dx ;  
        y += dy;  
    }  
    public int getY(){  
        return y;  
    }  
    public int getX(){  
        return x;  
    }  
}
```



# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo 2 (cont.):

```
public class RealPoint extends Point {
```

```
    private float x=0.0f, y=0.0f;
```

Ocultam x e y

```
    public void move (float dx, float dy){
```

```
        x += dx;
```

```
        y += dy;
```

```
    }
```

```
    public void move (int dx, int dy){
```

```
        move ( (float) dx, (float) dy);
```

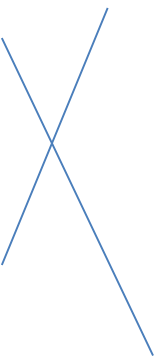
```
    }
```

Sobrecarrega (overloads) o método move da própria classe e o método move da superclasse

Sobrepõe (overrides) o método move da superclasse

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo 2 (cont.):



```
public float getY(){  
    return y;  
}  
public float getX(){  
    return x;  
}
```

Errado.

Sobreposição incorrecta de métodos da classe Point:

. mesmo nome ✓

. mesma assinatura ✓

. diferentes tipos de resultado ✗

## **Utilização de this e super em construtores**

Um construtor de uma dada classe pode invocar  
um construtor da mesma classe,  
usando a referência

this (...)

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo:

*voltando à classe Contador ...*

Em vez de,

```
public Contador () {  
    conta = 0;  
}  
public Contador (int val) {  
    conta = val;  
}
```

podíamos definir o primeiro construtor à custa do segundo:

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
public Contador (){  
    this (0);  
}
```

```
public Contador (int val ){  
    conta = val;  
}
```

## **Construtor de cópia**

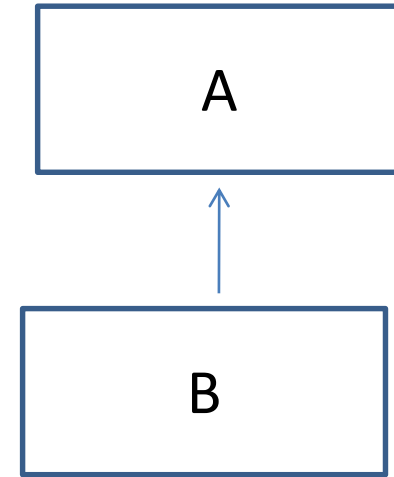
```
public Contador (Contador c ){  
    this ( c.conta );  
    //ou  
    // this.conta = c.conta;  
}
```

Cria um novo objecto do tipo Contador que é uma cópia do objecto que recebe como argumento.

- A invocação de um construtor através da referência this, se existir, tem que ser a primeira instrução do construtor com a excepção de quando exista a invocação do construtor da superclasse (nesse caso será a segunda).

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Dada uma classe A e uma sua subclasse B,



no construtor da subclasse, B,  
temos de inicializar as variáveis

definidas na subclasse B,  
e as variáveis  
definidas na superclasse A.

Todos os construtores de B devem invocar algum construtor de A através da referência `super(...)`.

Se isso não for feito explicitamente, o compilador insere como primeira instrução o construtor da superclasse, `super()`, que garantirá a invocação do construtor por omissão de A.



## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exercícios: Supondo a classe Empregado,

```
public class Empregado{  
    private long nss;  
    private String nome;  
    private double salario;  
    public Empregado () { ...};  
    public Empregado (long nss, String nome) { ...};  
    public long getNss() { ...};  
    public String getNome () { ...};  
    public double getSalario() { ...};  
    public void setNss( long nss) { ...};  
    public void setNome (String nome) { ...};  
    public void setSalario (double salario) { ...};  
    public String toString () { ...};  
}
```

}

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

### Exercícios:

1 – Para a classe empregado, construa:

a) – os métodos equals e clone;

b) – um construtor de cópia

2 – Suponha uma oficina de automóveis em que existem gestores, empregados administrativos e empregados especializados. Um empregado especializado é um Empregado que tem formação numa dada especialidade (por exemplo, electricista, mecânico, etc) e uma dada categoria (supervisor, técnico principal, etc).

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exercícios:

Para a classe empregado, construa:

a) – os métodos equals e clone;

```
public boolean equals (Object obj) {  
    if ( obj != null && this.getClass() == obj.getClass() ) {  
        Empregado e = (Empregado) obj;  
        return this.nome.equals(e.nome) && this.nss == e.nss && this.salario == e.salario;  
    }  
    return false;  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
public Object clone(){
```

```
    Empregado e = new Empregado ( this.nss, this.nome);
```

```
    e.salario = this. salario;
```

```
    return e;
```

```
}
```

b) – um construtor de cópia

```
public Empregado (Empregado e) {
```

```
    this.nome = e.nome;
```

```
    this.nss = e.nss;
```

```
    this.salario = e.salario; }
```

Equivalente a:  
`this ( e.nss, e.nome)`

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

- Deixando para mais tarde, os gestores e os administrativos, pretende-se criar uma classe `EmpregadoEspecializado` ( ou abreviadamente (EE).

- a) Defina os atributos ;
- b) Um construtor com um objecto do tipo `Empregado` e a especialidade como parâmetros;
- c) Um construtor de cópia
- d) Os getters e setters;
- e) O método `toString`
- f) O método `equals`
- g) O método `clone`

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

a) `public class EE extends Empregado{`

`private String esp; private String cat;`

b) Um construtor com um objecto do tipo Empregado e a especialidade como parâmetros;

`public EE (Empregado e, String esp){`

`super (e.getNss(), e.getNome());`

`super.setSalario(e.getSalario());`

`this.esp= esp;`

`this.cat = “”;`

`}`

Inicializar as variáveis  
herdadas da superclasse

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

c) Um construtor de cópia

```
public EE (EE x){  
    super( x.getNss(), x.getNome());  
    super.setSalario(x.getSalario());  
    this.esp = x.esp;  
    this.cat = x.cat;  
}
```

...

e) Método toString

@Override

```
public String toString() {  
    return super.toString() + "esp=" + esp + ", cat=" + cat ;  
}
```

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Main ...

**Qual o output?**

```
Empregado emp1 = new Empregado ();  
EE eesp1 = new EE (emp1, "mecânico");
```

```
System.out.println("1 - " + emp1.getClass());  
System.out.println("2 - " + eesp1.getClass());
```

*1 - class estudo.Empregado  
2 - class estudo.EE*

```
System.out.println("3 - " + ( emp1 instanceof Empregado) );  
System.out.println("4 - " + ( emp1 instanceof EE ) );
```

*3 - true  
4 - false*

```
System.out.println("5 - " + ( eesp1 instanceof Empregado ) );  
System.out.println("6 - " + ( eesp1 instanceof EE ) );
```

*5 - true  
6 - true*



f) Método equals

```
public boolean equals (Object o){  
    if (o!= null && o.getClass()== this.getClass()){  
        EE ee = (EE)o;  
        return super.equals(ee) &&  
            this.cat.equals(ee.cat) &&  
            this.esp.equals(ee.esp);  
    }  
    return false;  
}
```

g) Método clone

```
public Object clone (){  
    EE copia = new EE(this);  
    return copia;  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

f) Método equals

g) Método clone

```
public Object clone (){  
    EE copia = new EE(this);  
    return copia;  
}
```

Equivalente a:

```
public Object clone(){  
    Empregado e = new Empregado (super.getNss(), super.getNome());  
    e.setSalario ( super.getSalario());  
    EE copia = new EE (e, this.esp);  
    copia.cat = this.cat;  
    return copia;  
}
```

## **6 – Polimorfismo (continuação ...)**

### **Princípio da substitutividade:**

Declarada uma variável como sendo de uma dada classe (tipo), é permitido que lhe seja atribuído um valor da sua classe ou de qualquer subclasse desta.

### **Tipo estático versus tipo dinâmico**

A declaração de uma variável é um processo estático (determina o tipo estático da variável em tempo de compilação)

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

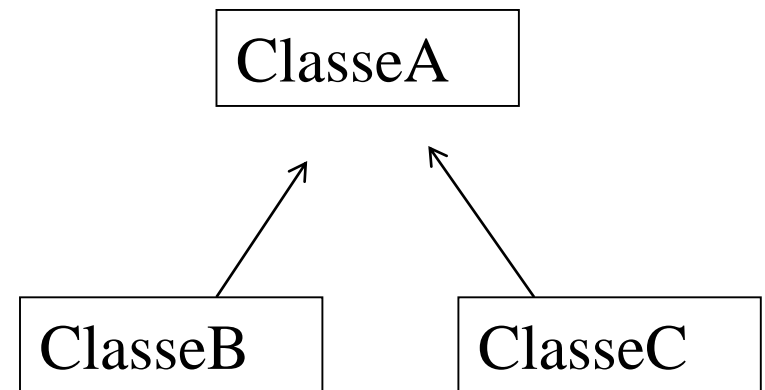
Ex.

...

ClasseA a1, a2, a3;

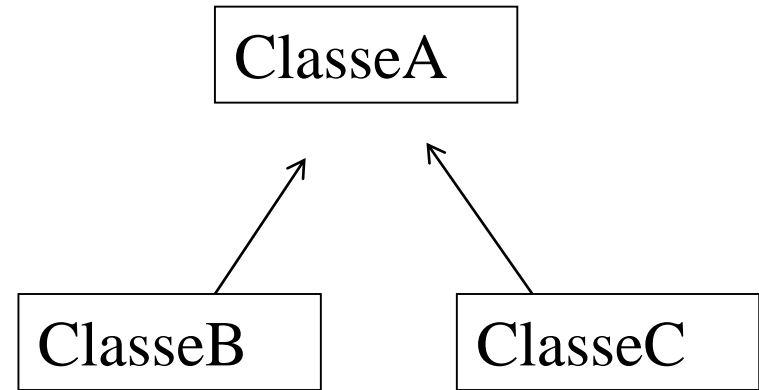
// o tipo estático das variáveis a1,a2 e a3 é ClasseA

Supondo a hierarquia:



# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

São permitidas as atribuições:



`a1 = new ClasseA();`

`a2 = new ClasseB();`

`a3 = new ClasseC();`

O tipo dinâmico, isto é, o tipo em tempo de execução (verificado pelo interpretador) de:

`a1` é ClasseA

`a2` é ClasseB

`a3` é ClasseC

Uma variável cujo tipo dinâmico pode ser diferente do tipo estático diz-se polimórfica.

## **Polimorfismo**

Capacidade de um valor ter mais do que um tipo.

Uma função (método) ou um operador dizem-se polimórficos se podem ser aplicados a vários tipos de valores.

Na generalidade das linguagens existem duas **formas de polimorfismo:**



## **1) De coerção**

Ex.lo - uma variável inteira é tratada como real

*Existe uma relação pré-definida de correspondência entre tipos.*

*Se determinado contexto exige um tipo e recebe outro, é verificado se existe a conversão adequada.*

## **2) De sobrecarga** (overloading)

*O mesmo nome (de um método ou função) pode ser usado mais do que uma vez com diferentes tipos de parâmetros.*

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Exemplo 1 – os operadores aritméticos são uniformemente aplicáveis a reais e inteiros.

Exemplo 2 – Operador “+ ” em Java

...

```
int x = 1;
```

```
int y = 2;
```

```
String output1 = “teste” + x + y;
```

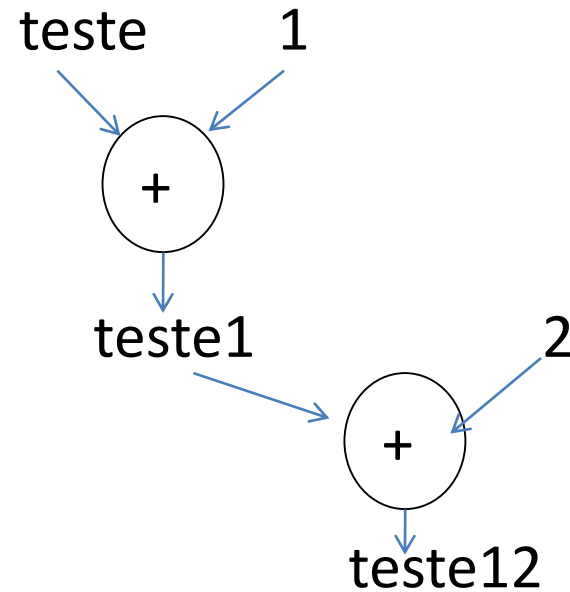
```
String output2 = x + y + “teste”;
```

Qual a diferença ?

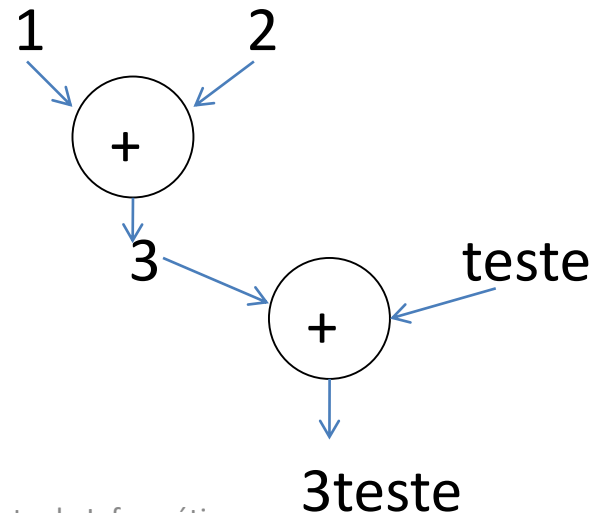


# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

String output1 = "teste" + x + y;



String output2 = x + y + "teste";



Outro tipo mais importante de polimorfismo:

### 3) **Polimorfismo universal**

Capacidade de uma única função (código único) poder ser usado com mais do que um tipo.

Existem duas formas de polimorfismo universal:

### 3.1) Polimorfismo de inclusão

- Uma função definida num determinado tipo pode também operar todos os seus subtipos.

*Resulta directamente do mecanismo de herança – uma operação definida na classe base é também aplicável aos objectos de todas as subclasses*

### 3.2) Polimorfismo paramétrico

Uma única função pode ser aplicada a um conjunto de tipos (sem qualquer relação entre si)

Funções genéricas

(exemplo: - packages em Ada, templates em C++, classes genéricas em Java )

existe implícita ou explicitamente um parâmetro de tipo que determina o tipo de argumento para cada aplicação da função.

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Exercícios**

Suponha uma ArrayList de Empregados:

```
ArrayList<Empregado> lista;
```

- a) Construa um método de classe (public static) que receba uma arrayList de Empregados e conte quantos empregados da lista são empregados especializados.
- b) Construa um método de classe que receba uma ArrayList de Empregados e calcule o total dos salários dos empregados da lista.
- c) Construa um método de classe que receba uma ArrayList de Empregados e conte quantos empregados têm a categoria “supervisor”.
- d) Construa um programa onde teste cada um dos métodos anteriores.

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

a) Construa um método de classe (public static) que receba uma arrayList de Empregados e conte quantos empregados da lista são empregados especializados.

```
public static int contaEE (ArrayList<Empregado> lista){  
    int n = 0;  
    for (int i = 0; i < lista.size(); i++) {  
        if (lista.get(i) instanceof EE )  
            n++;  
    }  
    return n;  
}
```

Como contar os objetos que não são empregados especializados?

## **Exercícios**

b) Construa um método de classe que receba uma ArrayList de Empregados e calcule o total dos salários dos empregados da lista.

```
public static double totalSalario (ArrayList<Empregado> lista){  
    double t = 0;  
    for (int i = 0; i < lista.size(); i++) {  
        t = t + lista.get(i).getSalario();  
    }  
    return t;  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

c) Construa um método de classe que receba uma ArrayList de Empregados e conte quantos empregados têm a categoria “supervisor”.

```
public static int contaSupervisores (ArrayList<Empregado> lista){  
    int n = 0;  
    for (int i = 0; i < lista.size(); i++) {  
  
        if (lista.get(i) instanceof EE )  
  
            if ( ( (EE)lista.get(i) ).getCat().equals("Supervisor") )  
                n++;  
    }  
    return n;  
}
```



# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Exercícios**

d) Construa um programa onde teste cada um dos métodos anteriores.

...