

6 – Polimorfismo

Sobrecarga (overloading) de métodos:

```
public class x {
```

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

sobrecarga do método m1

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

```
}
```

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.

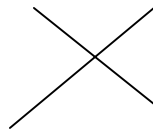
É incorrecto definir

```
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.

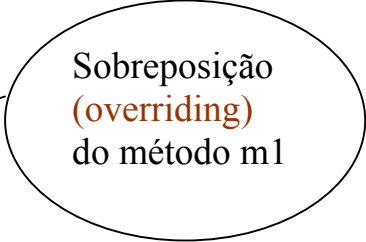
Sobreposição (overriding) de métodos:

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

A definição local sobrepõe-se à definição herdada.

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 " )  
    }  
}
```



Dois métodos (de classes diferentes) sobrepõem-se se têm a **mesma assinatura** e (necessariamente) o mesmo tipo de resultado.

Qual o output do segmento de código:

```
A a = new A ( ) ;  
B b = new B ( ) ;  
a.m1 ( ) ;  
b.m1 ( ) ;
```



Classe A Classe B

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

`super.m1()`

Exemplo:

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

```
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;}

    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;
    }
}
```

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

```
    private int x ;
```

```
    private int z;
```

```
    public ClasseB () {
```

```
        super () ;
```

oculta (hides)
a definição de x de ClasseA

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

Variáveis
definidas em
ClasseB

$$\left\{ \begin{array}{l} x = 100; \\ z = 0; \end{array} \right.$$

```
// fim do construtor
public int getZ(){
    return z;
}
public void setZ(int c){
    z=c;
}

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

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

}

} // fim da classe ClasseB
```

sobreposição (overriding) do método getX

devolve o valor de x definido na classe ClasseB

```
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);
        System.out.println(b.getX()); (2)
        System.out.println(b.getXsuper()); (3)
    }
}
```

Qual o output deste programa?

Resposta:

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)

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;
    }
}
```

```
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;
```

```
    }
```

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

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

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

```
    }
```

sobrepõe (overrides) o método move da superclasse

```
public float getY(){  
    return y;
```

```
}
```

```
public float getX(){  
    return x;
```

```
}
```

Errado.

Sobreposição incorrecta:

. 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 (... , ...)

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:

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

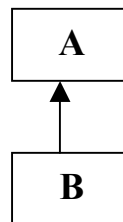

Construtor de cópia

```
public Contador (Contador c ) {  
    this( c.getConta() );  
}
```

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).

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.

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)

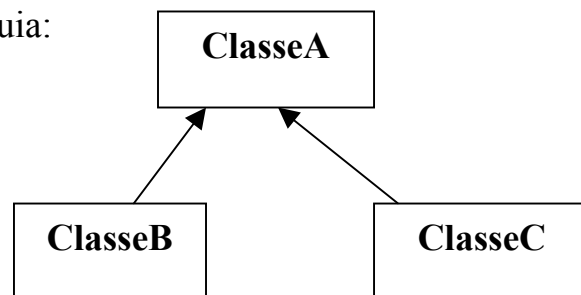
Ex.

...

```
ClasseA a1, a2, a3;
```

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

Supondo a hierarquia:



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. - 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.

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

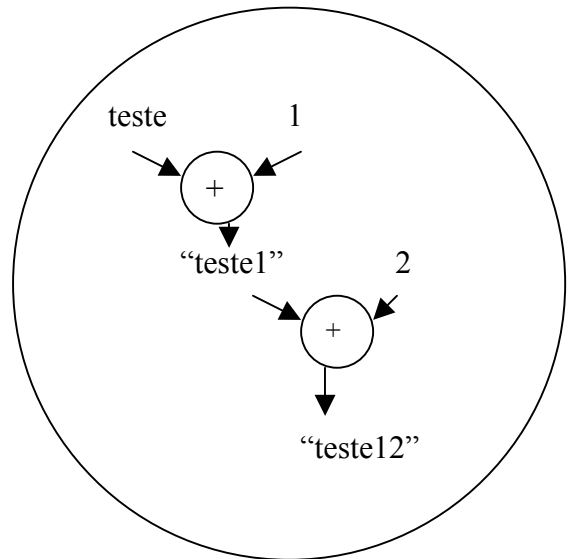
Exemplo 2 – Operador “+ “ em Java

...

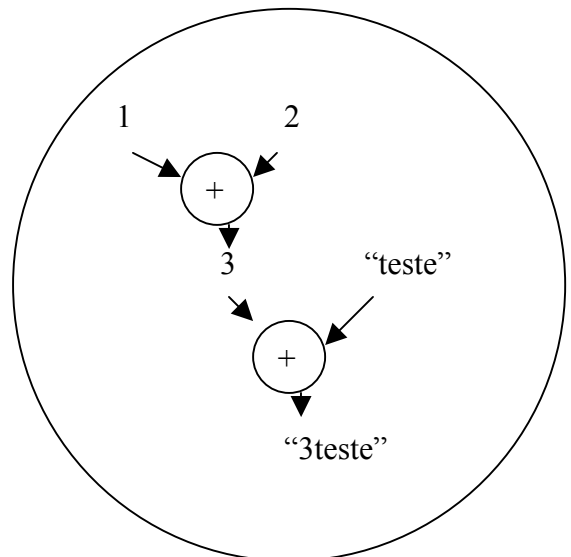
```
int x = 1;
```

```
int y = 2;
```

```
String output1 = "teste" + x + y;
```



```
String output2 = x + y + "teste";
```



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 (ex. - ver linguagem Ada) – existe implícita ou explicitamente um parâmetro de tipo que determina o tipo de argumento para cada aplicação da função.