

# PROGRAMAÇÃO ORIENTADA A OBJETOS

## Polimorfismo



# Polimorfismo



2

- Princípio pelo qual entidades de tipos diferentes podem ser acessadas através de uma única interface<sup>1</sup>

1. <http://www.stroustrup.com/glossary.html#Gpolymorphism>

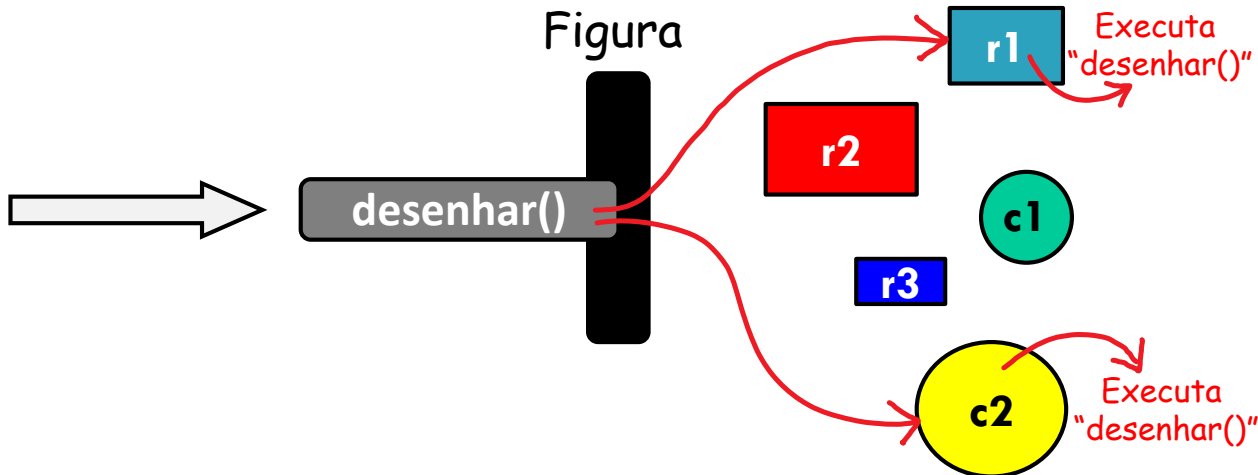
# Polimorfismo

3

- ❑ Diferente objetos “Retangulo” e “Circulo” podem ser acessados através da interface da classe “Figura”

## Programa usuário

```
01411: Private Function CleanUpLine(ByVal sLine As String) As String
01412: Dim iQuoteCount As Long
01413: Dim iCount As Long
01414: Dim sChar As String
01415: Dim sPrevChar As String
01416:
01417: ' Starts with the ' if it is a comment
01418: sLine = Trim(sLine)
01419: If Left(sLine, 1) = "'" Then
01420: CleanUpLine = ""
01421: Exit Function
01422: End If
01423:
01424: ' Starts with ' if it is a comment
01425: If Left(sLine, 1) = "" Then
01426: CleanUpLine = ""
01427: Exit Function
01428: End If
01429:
01430: ' Contains ' any end in a comment, so test if it is a comment or in the
01431: ' body of a string
01432: If InStr(sLine, "'") > 0 Then
01433: iQuoteCount = 0
01434:
01435: For iCount = 1 To Len(sLine)
01436: sChar = Mid(sLine, iCount, 1)
01437:
01438: ' If we found ' then an even number of ' characters in front
01439: ' means it is the start of a comment, and odd number means it is
01440: ' part of a string
01441: If sChar = "'" And sPrevChar = "" Then
01442: If iQuoteCount Mod 2 = 0 Then
01443: sLine = Trim(Left(sLine, iCount - 1))
01444: Exit For
01445: End If
01446: ElseIf sChar = "" Then
01447: iQuoteCount = iQuoteCount + 1
01448: End If
01449: sPrevChar = sChar
01450: Next iCount
01451: CleanUpLine = sLine
01452: End Function
```



```
Figura fig;
```

```
fig = new Retangulo();
```

→ "fig" aponta para um objeto "Retangulo"

```
fig.desenhar();
```

```
fig = new Circulo();
```

→ "fig" agora aponta para um objeto "Circulo"

```
fig.desenhar();
```

Note que o objeto "fig" assumiu mais de um tipo ao longo da execução do programa

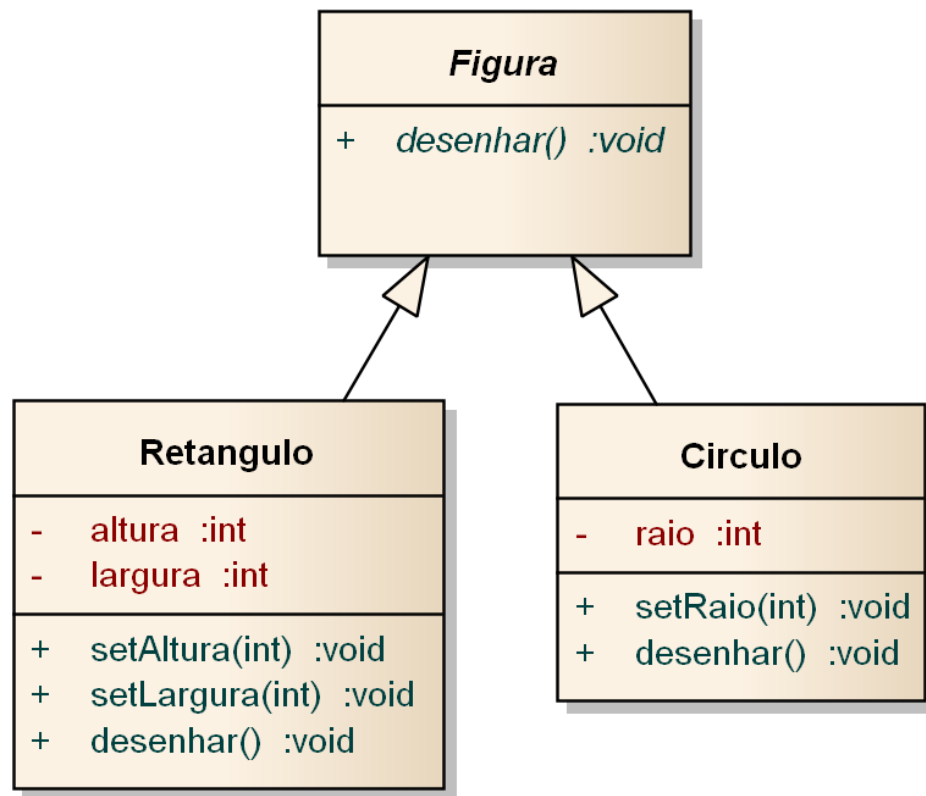
```
Saída - Exemplo04 (run) X
run:
sou um retângulo
sou um círculo
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

```
Figura fig;  
fig = new Retangulo();  
fig.desenhar();  
fig = new Circulo();  
fig.desenhar();
```

Este tipo de polimorfismo  
é conhecido como  
**"polimorfismo de inclusão"**

O **"Polimorfismo de Inclusão"** é obtido a partir do relacionamento de generalização/especialização (herança).

Uma variável declarada como sendo do tipo da superclasse pode então assumir (referenciar) qualquer objeto que tenha sido criado a partir de suas subclasses.

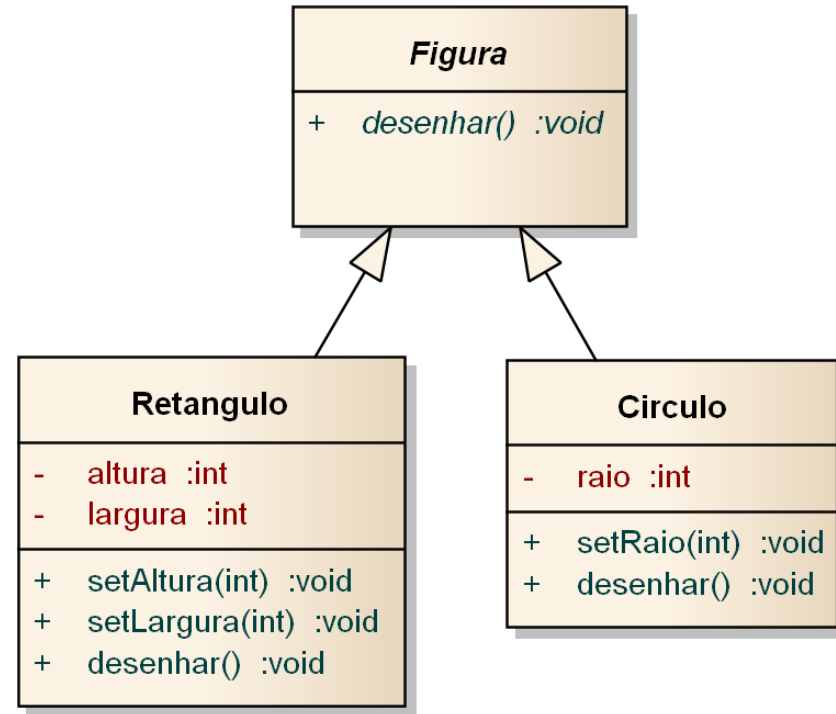


```
Figura fig;  
fig = new Retangulo();  
fig.desenhar();  
fig = new Circulo();  
fig.desenhar();
```

A variável "fig" está limitada pela interface da classe "Figura". Logo, você só poderá acessar as operações disponíveis em "Figura"!!

```
Figura fig;  
fig = new Retangulo();  
fig.setAltura(5); → ERRO!  
fig.desenhar();  
fig = new Circulo();  
fig.desenhar();
```

Você entendeu porque a operação "setAltura" não foi reconhecida pelo compilador?



```
Figura fig;  
fig = new Retangulo();  
(Retangulo) fig.setAltura(5);  
(Retangulo) fig.setLargura(9);  
fig.desenhar();  
fig = new Circulo();  
fig.desenhar();
```

E agora...  
Compila?

Typecasting  
(conversão  
de tipo)



O compilador reconhece  
a operação porque ela  
existe em "Retangulo"




```
Figura fig;  
fig = new Retangulo();  
((Retangulo) fig).setAltura(5);  
((Retangulo) fig).setLargura(9);  
fig.desenhar();  
fig = new Circulo();  
((Circulo) fig).setAltura(5);  
fig.desenhar();
```



**Você saberia explicar  
o motivo do erro de  
compilação?**

```
Figura fig;  
fig = new Retangulo();  
((Retangulo) fig).setAltura(5);  
((Retangulo) fig).setLargura(9);  
fig.desenhar();  
fig = new Circulo();  
(Retangulo) fig).setAltura(5);  
fig.desenhar();
```

Pronto! Agora  
compila, mas...



O erro agora será em  
tempo de execução!!  
Você entende o porquê?

```
Figura fig;  
fig = new Retangulo();  
((Retangulo) fig).setAltura(5);  
((Retangulo) fig).setLargura(9);  
fig.desenhar();  
fig = new Circulo();  
if (fig instanceof Retangulo) {  
    ((Retangulo) fig).setAltura(5);  
}  
fig.desenhar();
```

**E se...**

**Agora compila e não  
há erro na execução!**

**Mas, perdemos a transparência  
do polimorfismo de inclusão!**

# Ligação<sup>1</sup> (biding)



12

- ❑ **Ligação prematura** (*early binding*) ou **ligação estática** (*static binding*)
  - ▣ Quando o método a ser invocado é definido durante a compilação do programa
- ❑ **Ligação tardia** (*late binding*) ou **ligação dinâmica** (*dynamic binding*)
  - ▣ Quando o método a ser invocado é definido somente em tempo de execução do programa

1. Existem autores que utilizam o termo “acomplamento” como tradução de “biding”

# Ligação tardia



13

- ❑ O polimorfismo só pode ser aplicado se a linguagem de programação orientada a objetos suportar este mecanismo
  - ▣ Além de conhecida também como ligação dinâmica, pode ser ainda denominada ligação em tempo de execução (*runtime biding*)

# Ligação tardia em Java

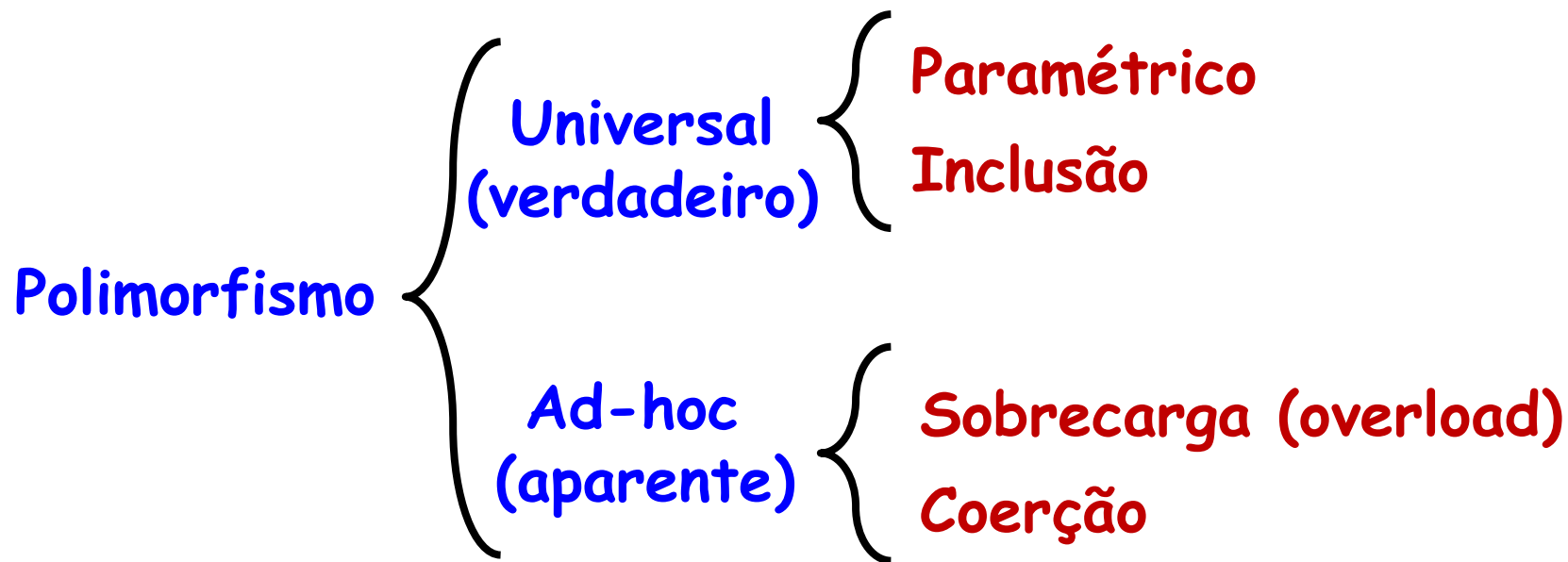


- ❑ Em Java, a ligação tardia é o comportamento padrão
- ❑ Exceções:
  - ▣ Métodos “**final**” não podem ser redefinidos e o polimorfismo não se aplica
  - ▣ Métodos “**private**” são implicitamente declarados como “**final**”

# Existem diferentes tipos de polimorfismo



15





# Tipos de polimorfismo



16

## □ Polimorfismo **universal** ou verdadeiro

- ▣ Quando uma operação ou tipo **trabalha uniformemente** para uma gama de tipos definidos na linguagem

## □ Polimorfismo **ad-hoc** ou aparente

- ▣ Quando uma operação ou tipo **parece trabalhar** para alguns tipos diferentes e pode se comportar de formas diferentes para cada tipo (ex: printf do C)



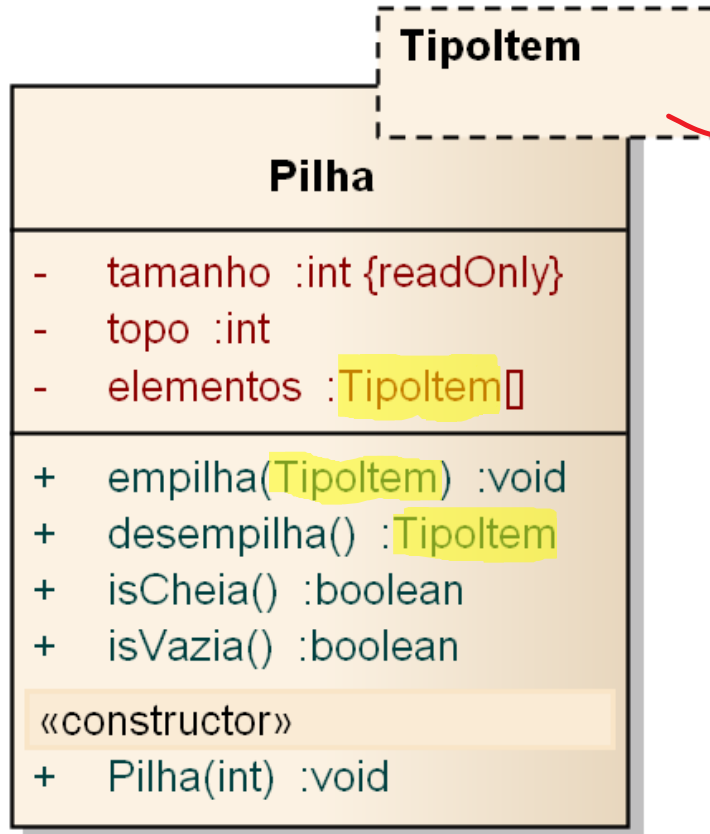
# Polimorfismo paramétrico



## □ Classes paramétricas

- ▣ Permitem que operações e classes **operem sobre dados de diferentes tipos**, sem que elas precisem ser reescritas para cada um dos tipos desejados
- ▣ Uma operação polimórfica tem um **parâmetro de tipo implícito ou explícito**, o qual determina o tipo de argumento para cada aplicação daquela operação

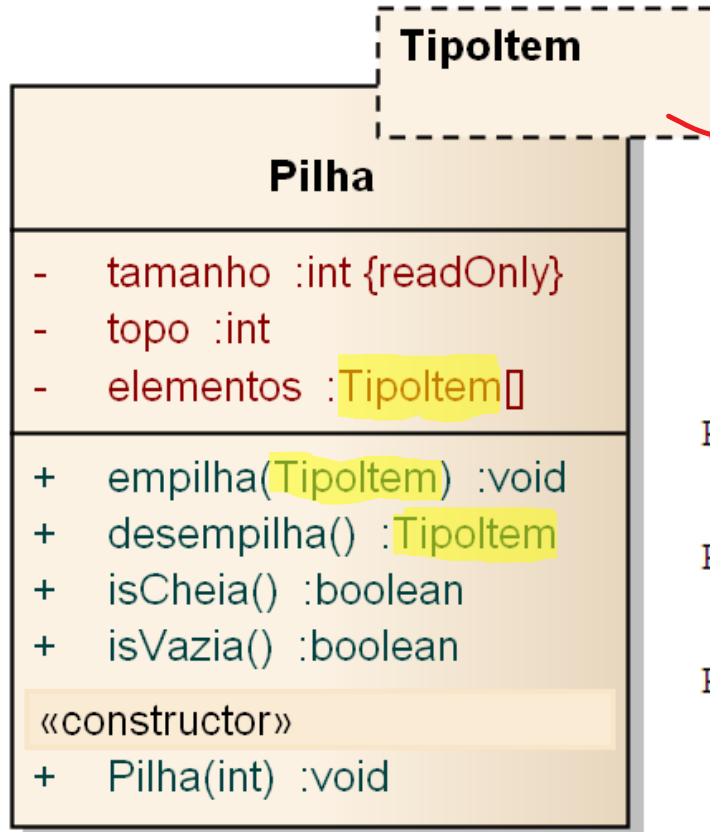
# Template de classe na UML



Parâmetro "Tipo" passado no momento da declaração de variáveis "Pilha"

Objetos desta classe podem ser declarados para diferentes tipos (TipoItem) de modo uniforme: o comportamento será sempre o mesmo

# Template de classe na UML



Parâmetro "Tipo" passado no momento da declaração de variáveis "Pilha"

```
Pilha<Figura> pilhaFig = new Pilha<>();
```

```
Pilha<Retangulo> pilhaRet = new Pilha<>();
```

```
Pilha<Circulo> pilhaCirc = new Pilha<>();
```

```
package br.univali.poo.exemplos.pilha;

public class Pilha<TipoItem> {

    private final int tamanho;
    private int topo;
    private TipoItem[] elementos;

    public Pilha() {
        this(10);
    }

    public Pilha(int tamanho) {
        this.tamanho = tamanho > 0 ? tamanho : 10;
        topo = -1;
        elementos = (TipoItem[]) new Object[tamanho];
    }
}
```

Em Java, templates são  
denominados tipos genéricos:  
**Generics**

```
package br.univali.poo.exemplos.pilha;
```

```
public class Pilha<TipoItem> {
```

```
    private final int tamanho;
```

```
    private int topo;
```

```
    private TipoItem[] elementos;
```

```
    public Pilha() {
```

```
        this(10);
```

```
    }
```

```
    public Pilha(int tamanho) {
```

```
        this.tamanho = tamanho > 0 ? tamanho : 10;
```

```
        topo = -1;
```

```
        elementos = (TipoItem[]) new Object[tamanho];
```

```
    }
```

<TipoItem> pode assumir  
qualquer tipo, mas o  
comportamento será sempre  
o de uma Pilha!

```
package br.univali.poo.exemplos.pilha;
```

```
public class Pilha<TipoItem> {
```

```
    private final int tamanho;
```

```
    private int topo;
```

```
    private TipoItem[] elementos;
```

```
    public Pilha() {
```

```
        this(10);
```

```
    }
```

```
    public Pilha(int tamanho) {
```

```
        this.tamanho = tamanho > 0
```

```
        topo = -1;
```

```
        elementos = (TipoItem[]) new
```

```
    }
```

## Construtores




Você pode implementar vários construtores, mas eles precisam ter assinaturas diferentes!

```
public Pilha() {  
    this(10);  
}
```

Chama o construtor  
que aceita um "int"  
como parâmetro

```
public Pilha(int tamanho) {  
    this.tamanho = tamanho > 0 ? tamanho : 10;  
    topo = -1;  
    elementos = (TipoItem[]) new Object[tamanho];  
}
```

## Operador ternário (condicional)



```
public Pilha(int tamanho) {  
    this.tamanho = tamanho > 0 ? tamanho : 10;  
    topo = -1;  
    elementos = (TipoItem[]) new Object[tamanho];  
}
```

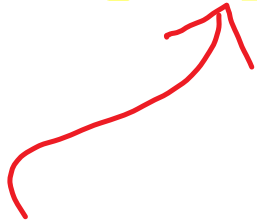


```
public Pilha(int tamanho) {  
    this.tamanho = tamanho > 0 ? tamanho : 10;  
    topo = -1;  
    elementos = (TipoItem[]) new Object[tamanho];  
}
```




O que é isso?

```
public Pilha(int tamanho) {  
    this.tamanho = tamanho > 0 ? tamanho : 10;  
    topo = -1;  
    elementos = (TipoItem[]) new Object[tamanho];  
}
```



**Criamos um vetor de objetos genéricos  
(qualquer objeto criado a partir de qualquer  
classe pode ser colocado neste vetor)**

```
public Pilha(int tamanho) {  
    this.tamanho = tamanho > 0 ? tamanho : 10;  
    topo = -1;  
    elementos = (TipoItem[]) new Object[tamanho];  
}
```



Forçamos então uma conversão  
(typecasting) para o tipo  
passado como parâmetro

# A classe `java.util.ArrayList`



```
ArrayList<Figura> figs = new ArrayList<>();  
Retangulo r = new Retangulo(3, 5);  
figs.add(r);  
Circulo c = new Circulo();  
c.setRaio(10);  
figs.add(c);  
for (Figura f : figs) {  
    f.desenhar();  
}
```

# Outras classes paramétricas em Java



29

- java.util.**Vector**<E>

- <http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html>

- java.util.**HashMap**<K,V>

- <http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

- java.util.**Stack**<E>

- <http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html>

- ...

# Sobrecarga (overload)



30

- Permite diferentes versões de uma mesma operação que aparenta funcionar com diferentes tipos
  - ▣ Os tipos não precisam possuir estrutura comum
  - ▣ A assinatura de cada versão da operação deve ser diferente
  - ▣ As versões das operações não têm relacionamento entre si
  
- Classificada como **polimorfismo ad-hoc** (aparente) ou não verdadeiro

# Sobrecarga (overload)



31

- ❑ Bastante utilizado para construtores
  - ▣ Permite instanciar um objeto de várias formas
    - Oportunidade para diferentes inicializações
- ❑ Mas, pode ser utilizado com qualquer operação
  - ▣ Pode reduzir a legibilidade do código
- ❑ **Você lembra dos dois construtores que implementamos para a classe Pilha?**

## Outro exemplo...

```
package br.univali.poo.exemplos.ponto;
```

```
public class Ponto {
```

```
    private int x, y;
```

```
    public Ponto() {  
        this(0, 0);  
    }
```

```
    public Ponto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }
```

```
    public void plota() {  
        System.out.printf("estou na posição (%d, %d)\n", this.x, this.y);  
    }  
}
```



# Coerção

33

- ❑ Conversão entre tipos diferentes, realizada automaticamente pelo compilador

```
int valor;  
valor = 10 / 3; // resultado float da divisão é convertido para int  
System.out.println(valor); // mostra "3"
```

```
float outroValor = 9.3f;  
outroValor = outroValor + valor; // "valor" é convertido para float  
System.out.println(outroValor); // mostra "12.3"
```

# Coerção



34

- ❑ Conversões implícitas do Java:
  - ❑ **byte** para **short**, **int**, **long** ou **double**
  - ❑ **short** para **int**, **long**, **float** ou **double**
  - ❑ **char** para **int**, **long**, **float** ou **double**
  - ❑ **int** para **long**, **float** ou **double**
  - ❑ **long** para **float** ou **double**
  - ❑ **float** para **double**

# Coerção e Type Casting



35

## □ Coerção

- ▣ Conversão **implícita** realizada pelo compilador

## □ Type Casting

- ▣ Conversão **explícita** realizada pelo programador

### ■ (tipo) variável

```
float outroValor = 9.3f;  
int valor = 10 + (int)outroValor;  
System.out.println(valor); // mostra "19"
```

# Tipos de type casting



36

- ❑ **Upcasting:** indução de uma referência para uma superclasse (tipo acima)
  - ▣ `Person person = new Employee();`
- ❑ **Downcasting:** indução de uma referência para uma subclasse (tipo abaixo)
  - ▣ `Employee employee = (Employee)person;`
  - ▣ Pode disparar a exceção não checada **ClassCastException**

# PROGRAMAÇÃO ORIENTADA A OBJETOS

## Polimorfismo

