

Polimorfismo

Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira
oliveira.edmar@ufjf.edu.br

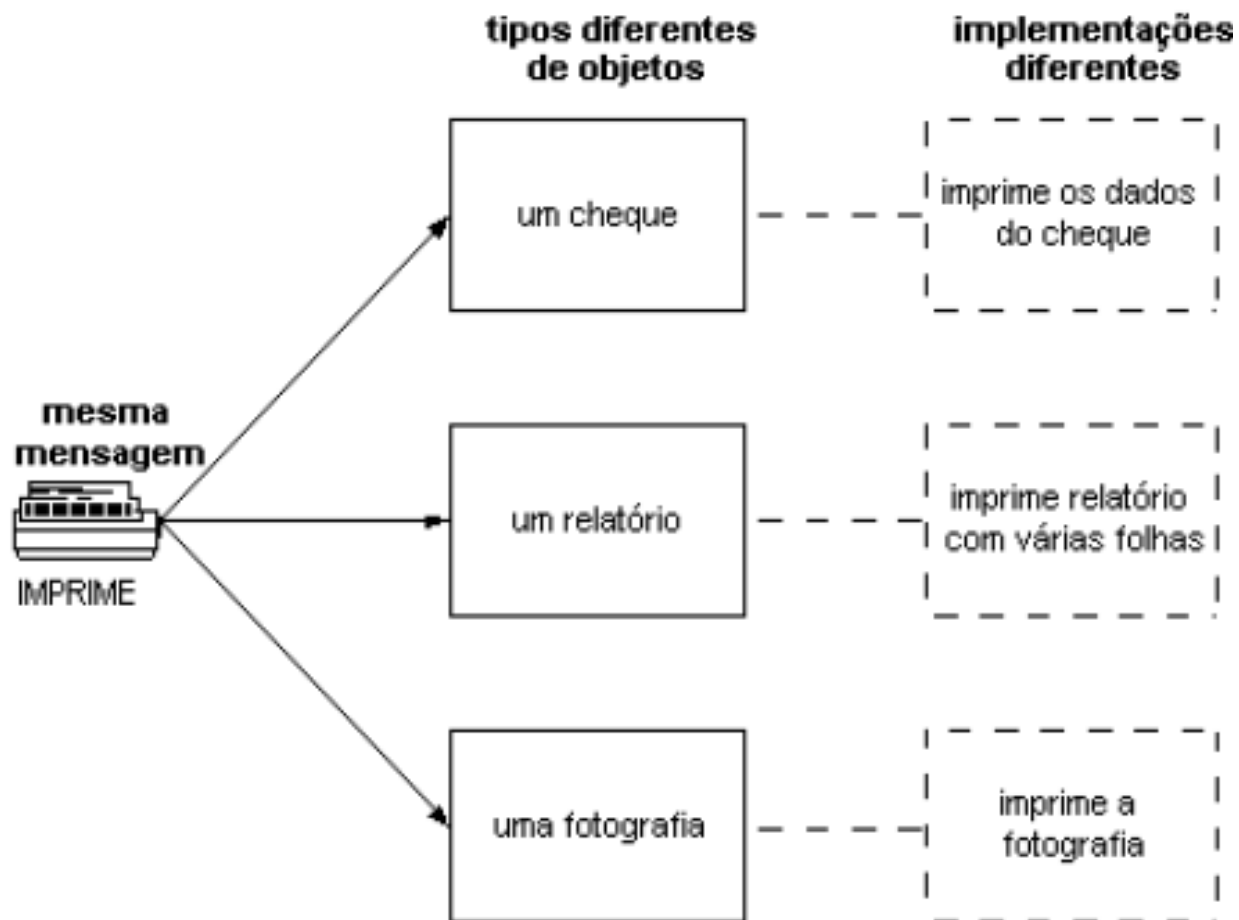
Universidade Federal de Juiz de Fora - UFJF
Departamento de Ciência da Computação - DCC

Polimorfismo

■ Definição

- Polimorfismo: várias formas
- Em OO, é uma propriedade segundo a qual pode existir, em um programa, vários métodos com o mesmo nome operando em classes diferentes (ou mesmo dentro de uma mesma classe). Ao receber uma mensagem para efetuar uma operação, é o objeto que determina como a operação deve ser efetuada
- Polimorfismo permite a criação de várias classes com interfaces idênticas, porém com objetos e implementações diferentes.
- Herança: se refere às classes e sua hierarquia
- Polimorfismo: se refere aos métodos dos objetos

Polimorfismo



Polimorfismo

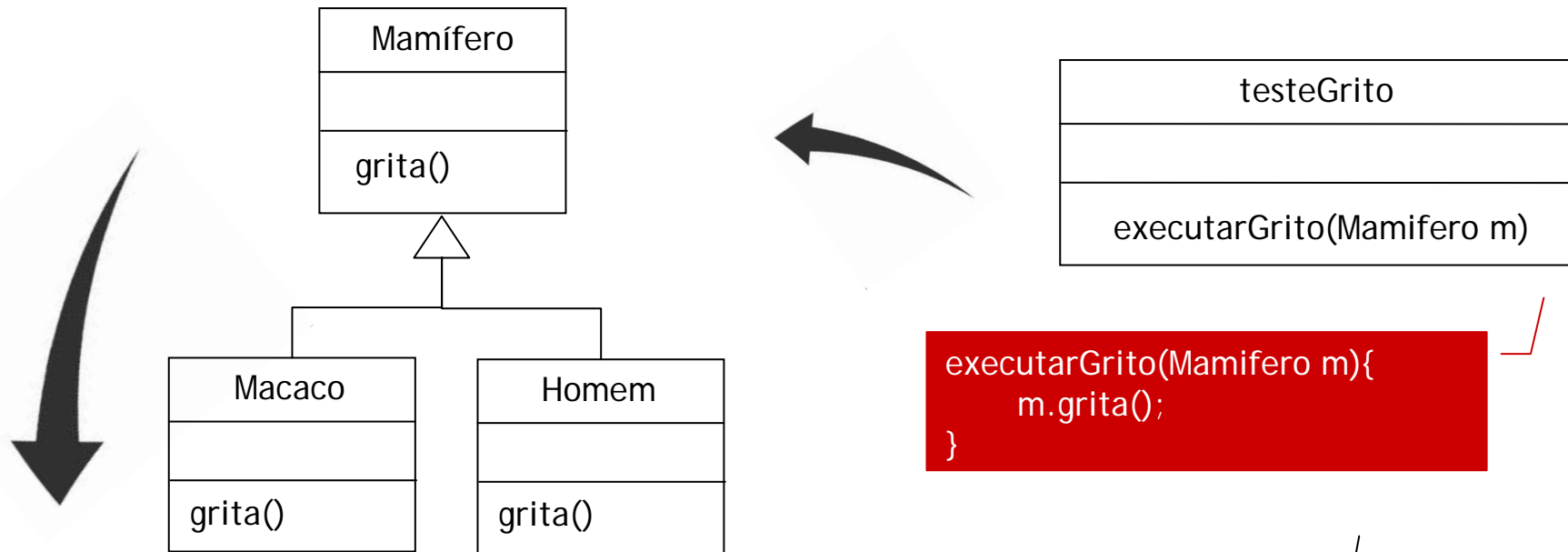
■ Definição

- Em uma linguagem de programação, isso significa que pode haver várias formas de fazer uma "certa coisa". Pergunta: que coisa é essa? O que precisa ficar na mente é que quando se fala em polimorfismo, se fala em **chamada de métodos**. Portanto, em Java, o polimorfismo se manifesta apenas em chamadas de métodos
- Sendo um pouco mais específico
 - Polimorfismo significa que uma chamada de método pode ser executada de várias formas (ou polimorficamente). Quem decide "a forma" é o objeto que recebe a chamada.

Polimorfismo

- Quem decide "a forma" é o objeto que recebe a chamada
 - Se um objeto "a" chama um método `metodo1()` de um objeto "b", então o objeto "b" decide a forma de implementação do método. Mais especificamente ainda, é o tipo do objeto "b" que importa.
 - Considere que `metodo1()` seja `grita()`. Então a chamada `b.grita()` vai ser um grito humano se "b" for um humano e será um grito de macaco, se o objeto "b" for um macaco. O que importa portanto, é o tipo do objeto receptor "b".

Polimorfismo



```
executarGrito(Mamifero m){
    m.grita();
}
```

Aplicação do polimorfismo - chamada do método `executarGrito(Mamifero m)`

```
executarGrito(Homem h);
executarGrito(Macaco m);
```

Chamadas de Métodos diferentes
Nomes iguais, mas parâmetros diferentes

Código

```
3 public class Mamifero {  
4  
5     public void grita(){  
6         System.out.println("Grito Mamífero");  
7     }  
8 }
```

```
3 public class Homem extends Mamifero{  
4  
5     public void grita(){  
6         System.out.println("Grito Homem");  
7     }  
8 }
```

```
3 public class Macaco extends Mamifero{  
4  
5     public void grita(){  
6         System.out.println("Grito Macaco");  
7     }  
8 }
```



Código

Quem decide "a forma" é o objeto que recebe a chamada - ou seja, "m" é quem decide de qual classe será chamado o método grita()

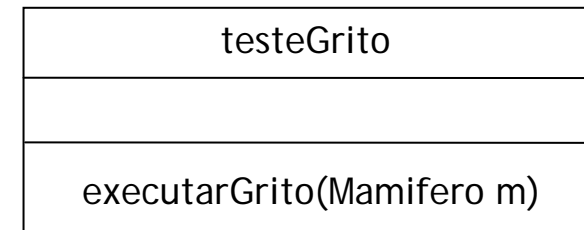
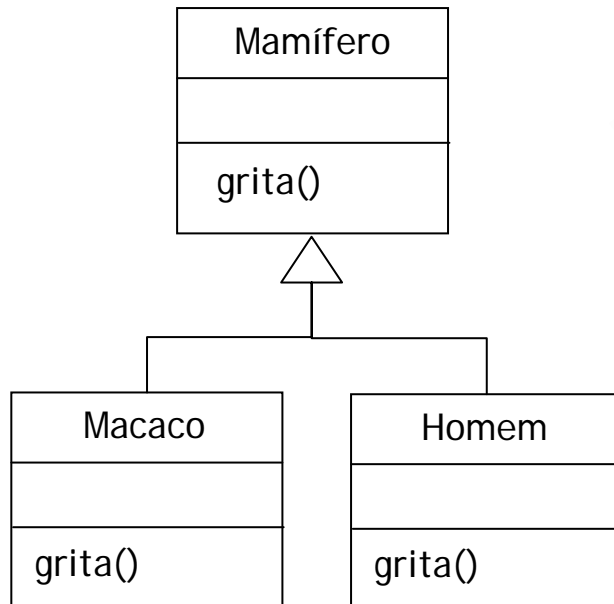
```
3 public class TesteGrito {  
4  
5     public void executarGrito(Mamifero m) {  
6         m.grita();  
7     }  
8 }
```

```
3 public class Principal {  
4  
5     public static void main(String args[]) {  
6  
7         TesteGrito teste = new TesteGrito();  
8  
9         Mamifero mamifero = new Mamifero();  
10        Homem homem = new Homem();  
11        Macaco macaco = new Macaco();  
12  
13        teste.executarGrito(mamifero);  
14        teste.executarGrito(homem);  
15        teste.executarGrito(macaco);  
16    }  
17 }
```


Problema

```
3 public class TesteGrito {  
4  
5     public void executarGrito(Mamifero m) {  
6         m.grita();  
7     }  
8  
9     public void executarGrito(Homem m) {  
10        m.grita();  
11    }  
12  
13    public void executarGrito(Macaco m) {  
14        m.grita();  
15    }  
16 }
```

Polimorfismo



```
executarGrito(Mamifero m){
    m.grita();
}
```

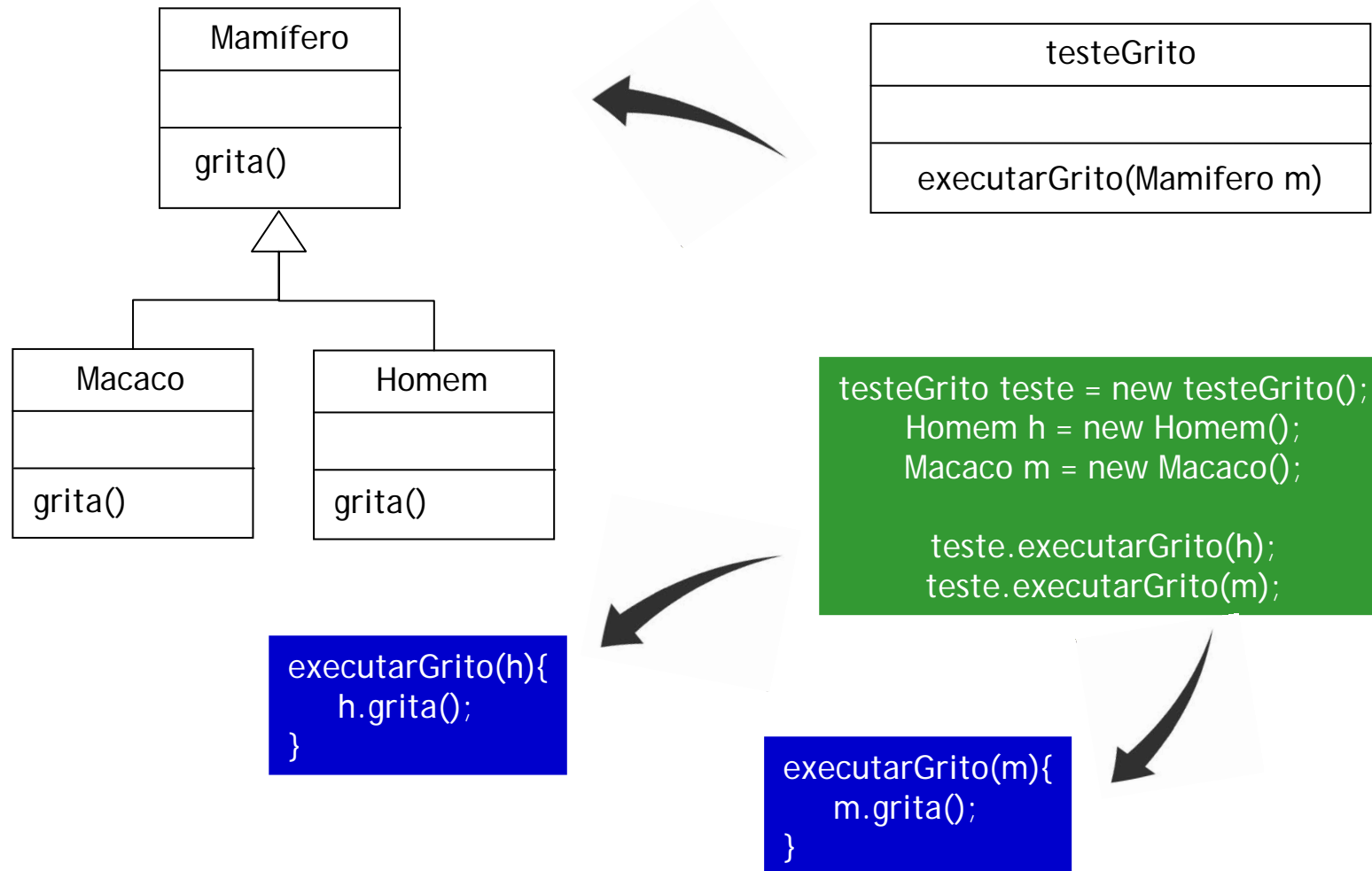
```
executarGrito(homem){
    homem.grita();
}

executarGrito(macaco){
    macaco.grita();
}
```

Essa representação ocorre apenas em tempo de execução. Não existe esse código na implementação

O código que existe é o representado no quadro em vermelho

Polimorfismo



Polimorfismo Paramétrico

■ Definição

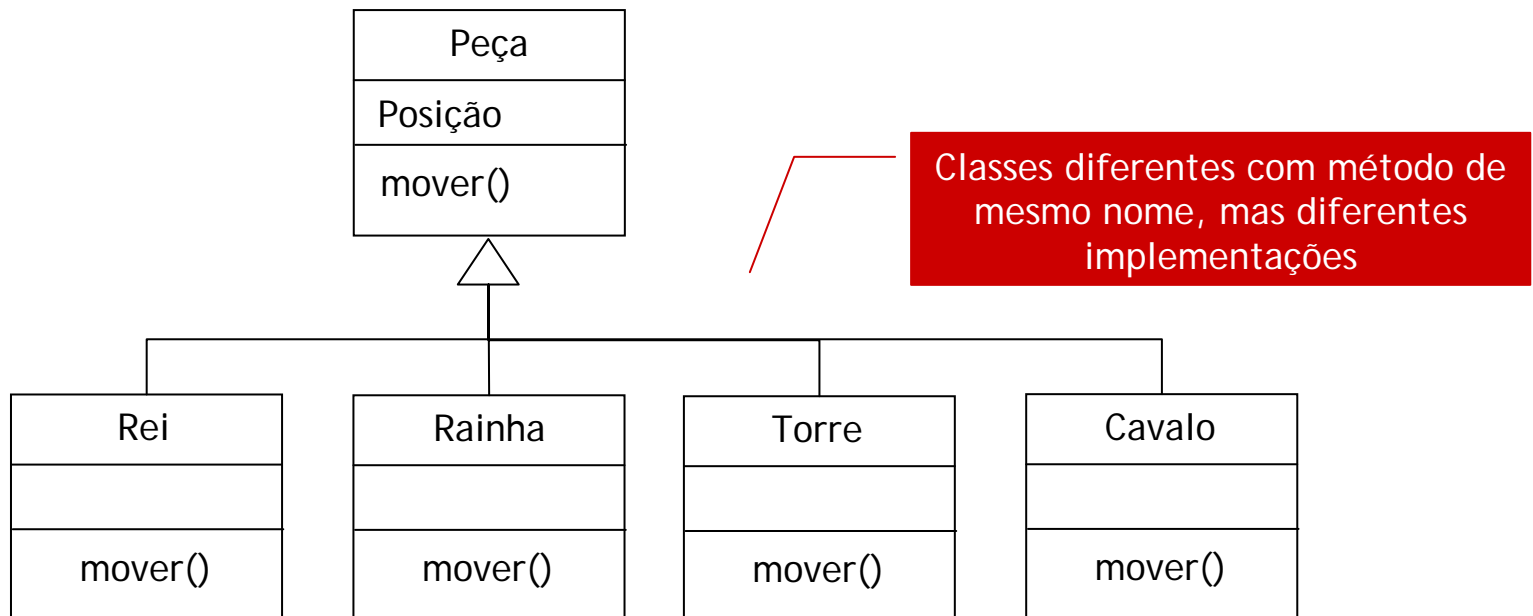
- Representa a possibilidade de definir várias funções do mesmo nome mas possuindo parâmetros diferentes (em número e/ou tipo) - realiza-se **sobrecarga**. Fica possível então, escolher, automaticamente, o método a ser executado - isso dependerá do tipo de dado(s) passado(s) como parâmetro(s).
- Pode-se ter métodos com mesmo nome (e diferentes parâmetros) dentro de uma mesma classe e entre classes distintas. OBS: métodos com nomes/parâmetros iguais geram erro quando dentro de uma mesma classe.

```
public void aumento() {salario= salario*10.0;}  
public void aumento(float percent)  
{  
    salario= salario*percent;  
}
```

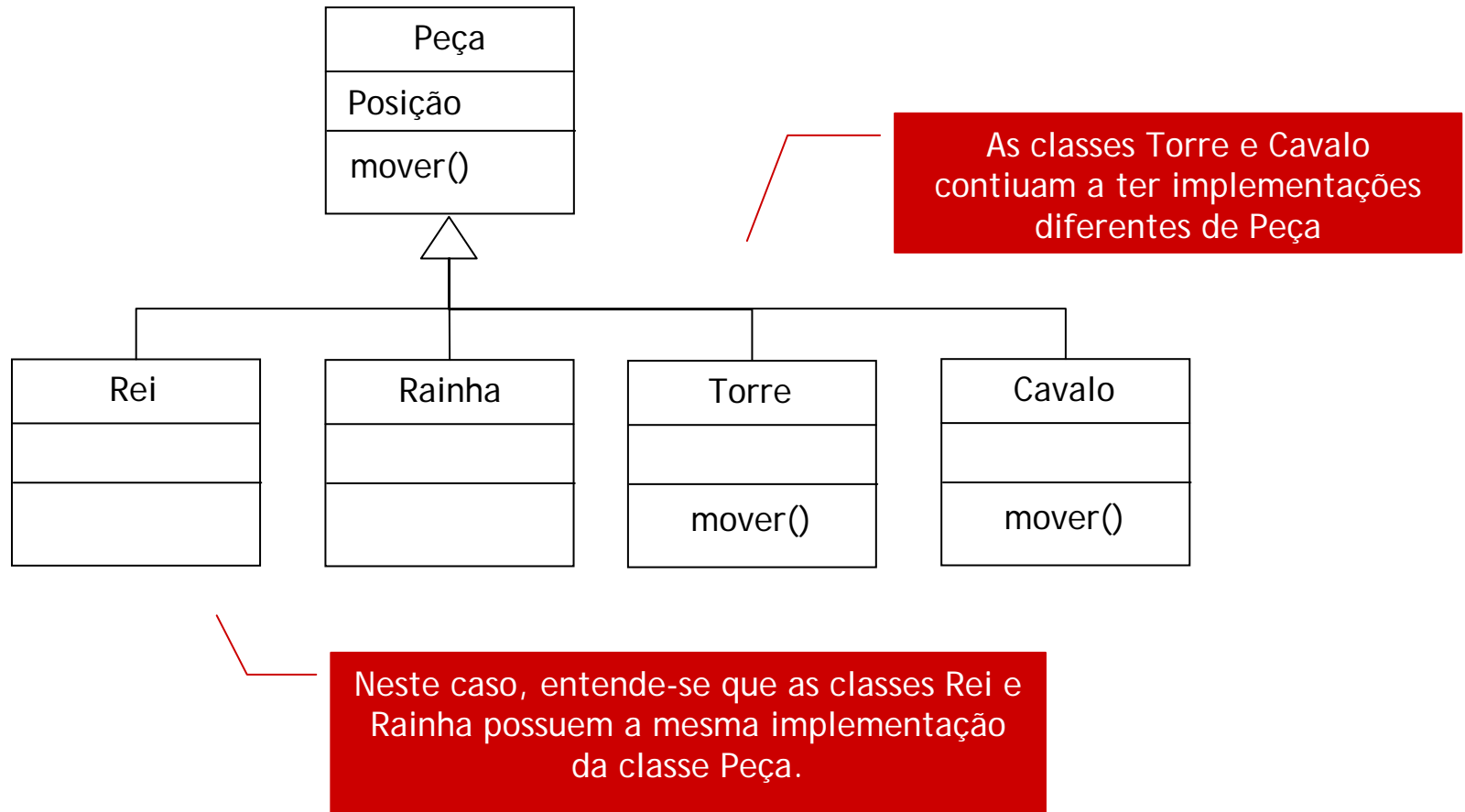
Polimorfismo de Herança

■ Definição

- Refere-se à possibilidade de redefinir um método em classes que são herdeiras de uma classe. É a velha conhecida especialização - **sobrescrita**.



Polimorfismo de Herança



Polimorfismo

■ Reforçando

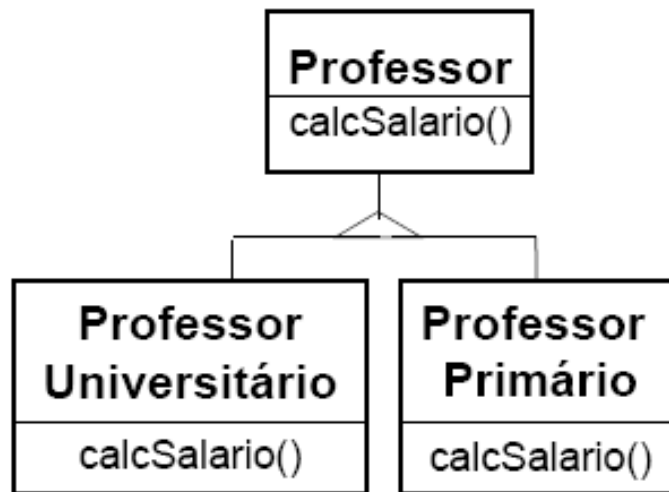
- Possibilidade de uma mesma operação atuar de forma diferente em classes diferentes. Isto é possível quando uma operação é declarada em classes diferentes, porém com o mesmo nome, executando processamentos diferentes para atender os requisitos semânticos de sua classe

Janela
mover()

PeçaXadrez
mover()

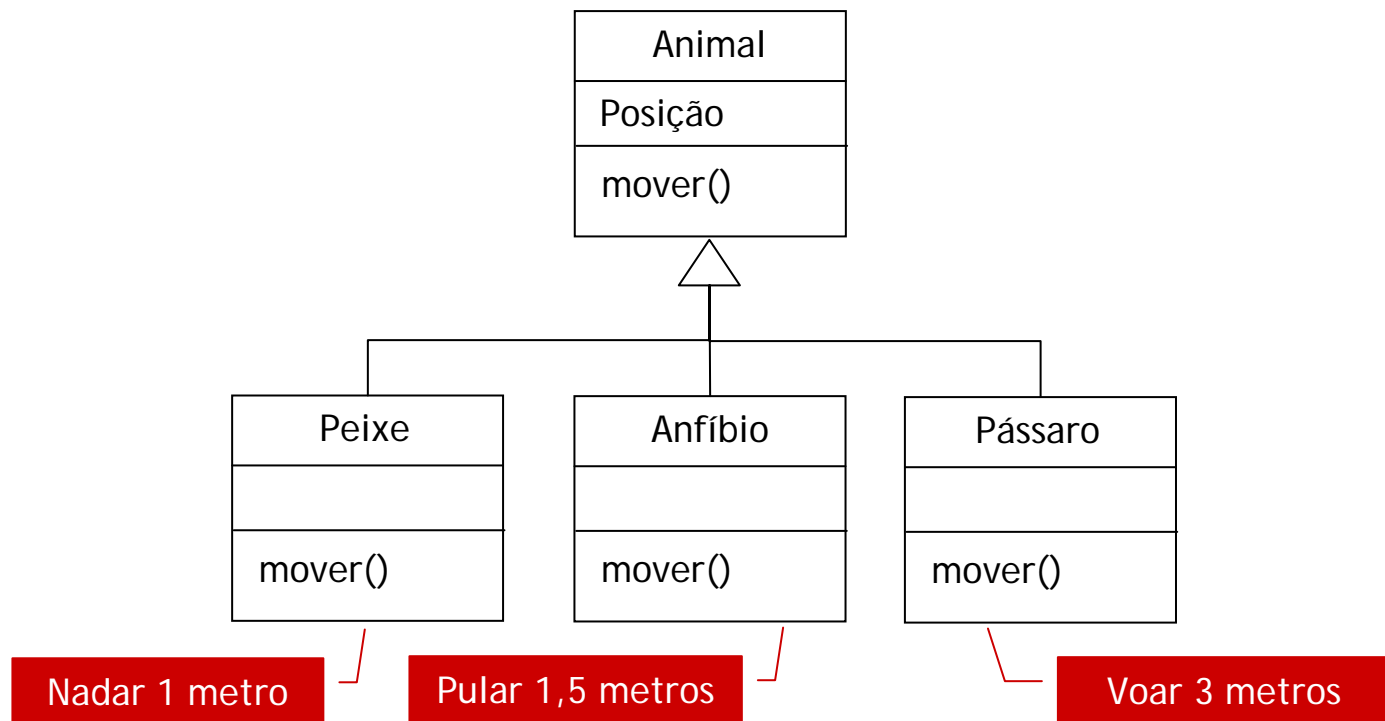
A operação "mover()" para um objeto Janela executa um processo diferente da operação "mover()" de um objeto PeçaXadrez

Exemplo



pode-se adicionar um comportamento específico (implementação) às subclasses de uma hierarquia de generalização/especialização.

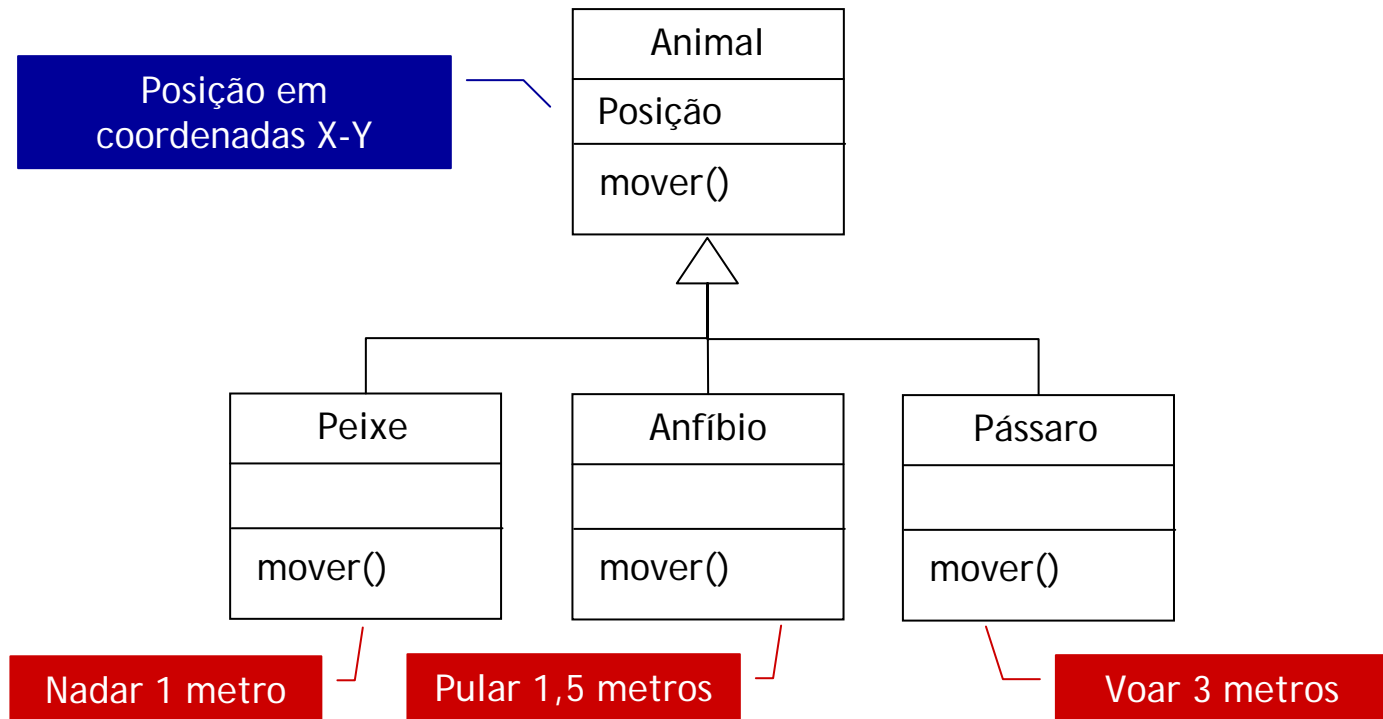
Exemplo



Programa - Simulador

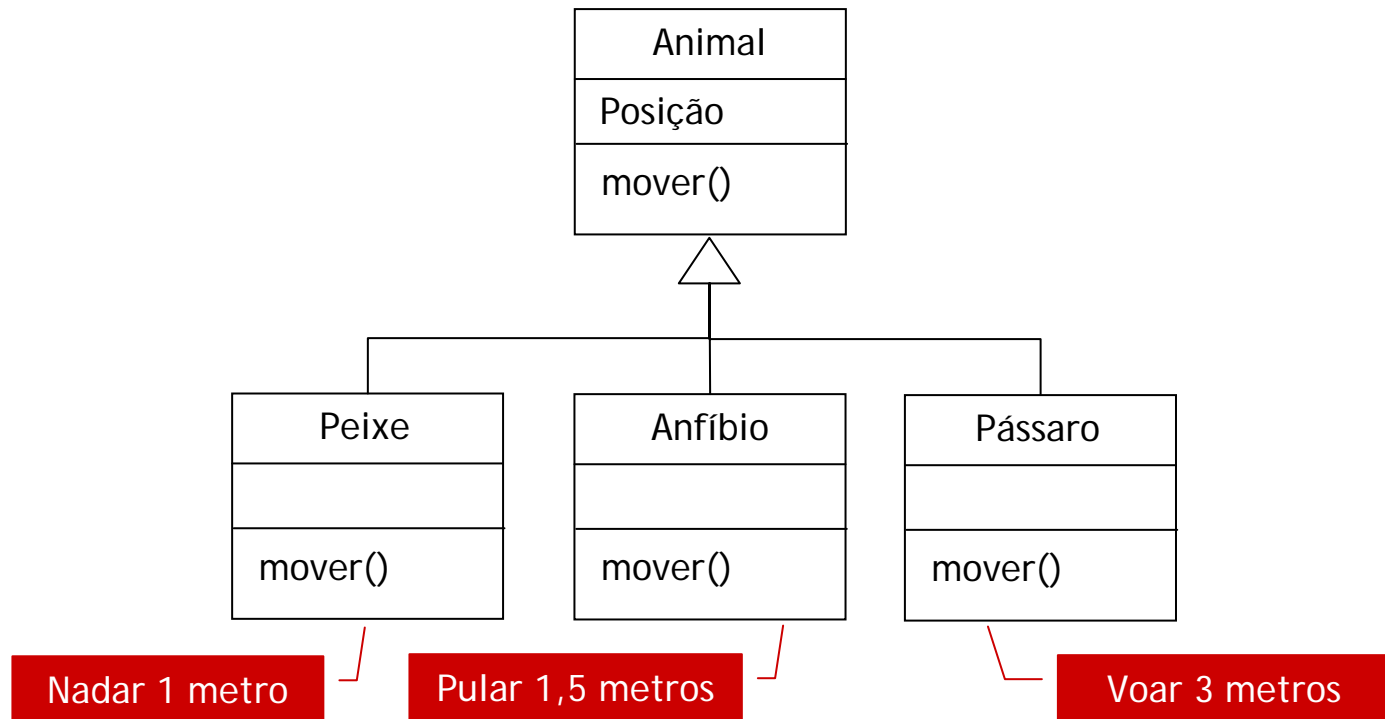
Envia a mesma mensagem a todos os objetos,
para simular o movimento dos animais

Exemplo



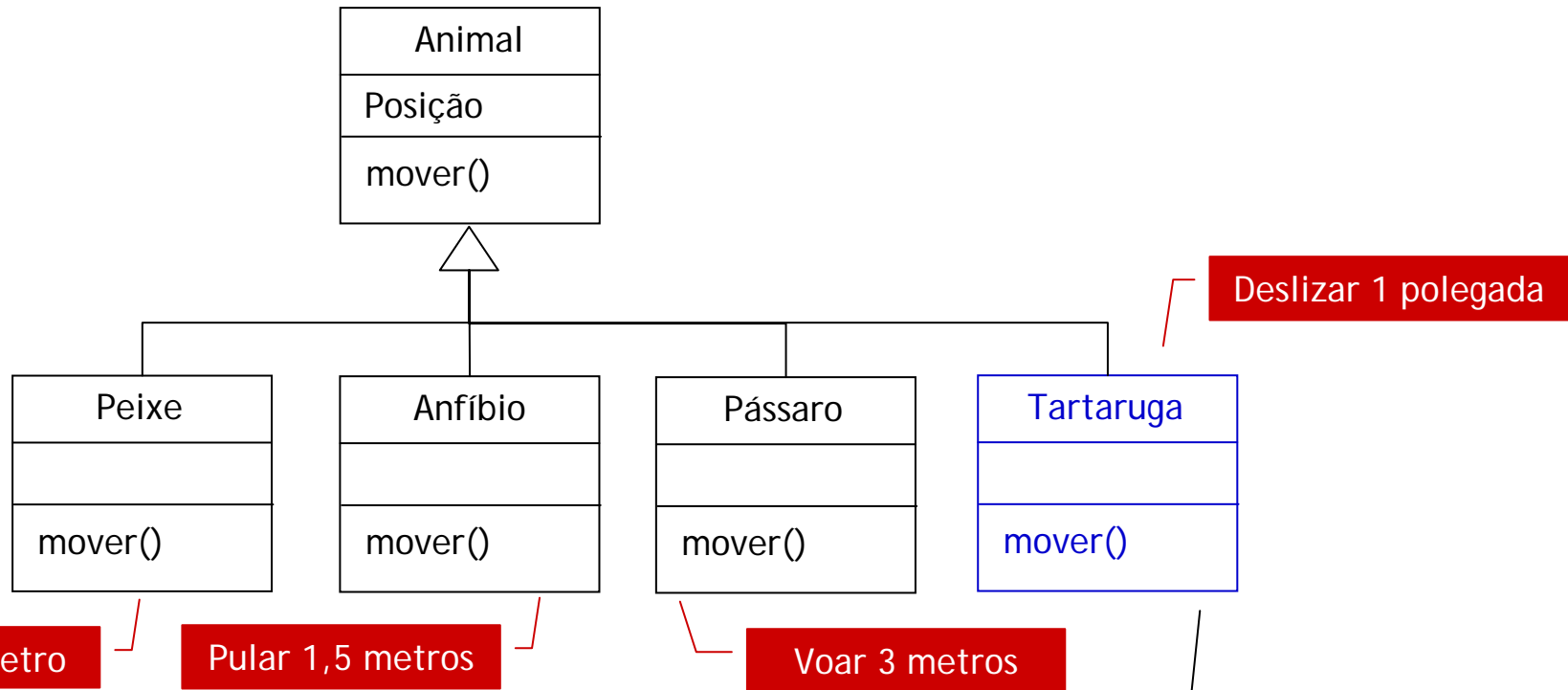
OBS: cada objeto irá interpretar a mensagem de uma forma diferente. O peixe nada, o anfíbio pula e o pássaro voa - uma certa quantidade de metros, alterando sua posição X-Y

Exemplo



Conceito chave de Polimorfismo: contar com o fato de que cada objeto sabe "fazer a coisa certa" em resposta a uma mesma chamada de método

Extensibilidade



Vantagem: pode-se adicionar uma nova classe no esquema sem qualquer modificação nas demais classes. Deve-se apenas implementar o método **mover()** de acordo com as necessidades da nova classe

Exemplo

```
class GeradorDeExtrato {  
  
    public void imprimeExtratoBasico(ContaPoupanca cp) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cp.getSaldo());  
    }  
}
```

Os clientes de um banco podem consultar algumas informações das suas contas através de extratos impressos nos caixas eletrônicos.

Eventualmente, poderíamos criar uma classe para definir um tipo de objeto capaz de gerar extratos de diversos tipos.

Exemplo

Imagine vários tipos de conta. Resultado: criação de um método `imprimeExtrato` para cada uma

```
class GeradorDeExtrato {  
  
    public void imprimeExtratoBasico(ContaPoupanca cp) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cp.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaCorrente cc) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cc.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaSalario cs) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cs.getSaldo());  
    }  
}
```

Exemplo

Supondo a criação de outros tipos de conta: contaSalario, contaCorrente, etc. seria viável generalizar o método imprimeExtratoBasico?

```
class GeradorDeExtrato {  
  
    public void imprimeExtratoBasico(ContaPoupanca cp) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cp.getSaldo());  
    }  
}
```

```
imprimirExtratoBasico(Conta conta){  
}
```

Exemplo

```
class GeradorDeExtrato {  
  
    public void imprimeExtratoBasico(ContaPoupanca cp) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cp.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaCorrente cc) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cc.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaSalario cs) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cs.getSaldo());  
    }  
}
```


Sobrecarga

■ Sobrecarga

- Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, **desde que em suas assinaturas, o tipo e/ou a quantidade de parâmetros sejam diferentes**. Quando se fala em sobrecarga, deve-se ter em mente o fato dos métodos terem assinaturas diferentes.
 - É diferente de sobrecrita de método: neste caso, uma mesma assinatura é utilizada - vimos isso quando estudamos herança. Quando um método A sobreescreve método B, A utiliza a mesma assinatura de B.
- Tal situação (de sobrecarga) não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da **análise dos argumentos do método**.

Sobrecarga - Java

```
1 abstract public class Classe1 {
2
3     public int calcular(int valor){
4         return 0;
5     }
6
7     public int calcular(int valor){
8         return 0;
9     }
10 }
11
12
```

Duplicate method calcular(int) in type Classe1

1 quick fix available:

Rename method 'calcular' (Ctrl+2, R)

```
1 abstract public class Classe1 {
2
3     public int calcular(int valor){
4         return 0;
5     }
6
7     public boolean calcular(int valor){
8         return 0;
9     }
10 }
11
12
```

Duplicate method calcular(int) in type Classe1

1 quick fix available:

Rename method 'calcular' (Ctrl+2, R)

```
1 abstract public class Classe1 {
2
3     public int calcular(int valor){
4         return 0;
5     }
6
7     public int calcular(float valor){
8         return 0;
9     }
10 }
```

Sobrecarga e Sobrescrever

- Sobrecarga
 - Métodos de nomes iguais, mas assinaturas diferentes
 - São métodos diferentes, que apenas possuem o mesmo nome
- Sobrescrever
 - Métodos de assinaturas iguais
 - É o mesmo método (em termos de assinatura)
 - Mas podem possuir implementações diferentes
- O que usar com polimorfismo?
 - Ambos podem ser utilizados em termos de polimorfismo
 - Com assinaturas iguais = uso do conceito de “classes abstratas”

Sobrescrita

```
class Empregado
{
    protected float salario;
    public float getSalario() {return salario;}
}

class Vendedor extends Empregado
{
    protected float comissao;
    public float getSalario() {return salario+comissao;}
}
```

Podemos definir mais de um método com o mesmo nome na mesma classe ou subclasses. Caso o método possua a mesma assinatura (como é o exemplo) que outro método, então o método não pode pertencer à mesma classe do anterior. Se ambos os métodos estiverem na mesma linha hierárquica (classe/subclasse), **como no exemplo**, dizemos que o método da subclasse sobrescreve o método da superclasse.

Sobrecarga

- Quando ocorre sobrecarga
 - Se a assinatura do método for diferente de outro método com o mesmo nome definido anteriormente na mesma classe ou em outra classe da mesma linha hierárquica, então estamos realizando uma sobrecarga sobre o identificador do método.

```
class Empregado
{
    protected float salario;
    public void aumento() {salario= salario*10.0;}
    public void aumento(float percent)
    {
        salario= salario*percent;
    }
}
```

identificador aumento pode referenciar dois métodos distintos. Um aumenta o salário em 10% e no outro o aumento depende do valor da porcentagem passado como parâmetro. Note que as assinaturas dos métodos diferem entre si.

Sobrescrita de Método

```
1 public class Classe1 {  
2  
3     private int resultado1;  
4  
5     public int calcular(int valor){  
6         resultado1 = valor * 2;  
7  
8         return resultado1;  
9     }  
10 }
```

```
1 public class Classe2 extends Classe1{  
2  
3     private int resultado2;  
4  
5     public int calcular(int valor){  
6         resultado2 = valor * 10;  
7  
8         return resultado2;  
9     }  
10 }
```

Sobrescrita de Método

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         Classe2 Obj2 = new Classe2();  
6  
7         Obj2.  
8     }  
9 }  
10
```

- calcular(int valor) : int - Classe2
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Sobrescrita de método - o método calcular() a ser chamado é aquele definido na classe2 e não na classe 1, embora classe 2 seja subclasse de classe 1

Sobrescrita de Método

```
1 public class Teste {  
2  
3     public static void main(String args[]) {  
4  
5         Classe2 Obj2 = new Classe2();  
6  
7         Obj2.  
8     }  
9 }  
10
```

- calcular(int valor) : int - Classe2
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Declarando o método calcular() como "protected" na classe 1 - continuamos a ver somente o método calcular() da classe 2

Sobrecarga de Método

```
1 public class Clase1 {  
2  
3     private int resultado1;  
4  
5     protected int calcular(int valor){  
6         resultado1 = valor * 2;  
7  
8         return resultado1;  
9     }  
10 }
```

```
1 public class Clase2 extends Clase1{  
2  
3     private float resultado2;  
4  
5     protected float calcular(float valor){  
6         resultado2 = valor * 10;  
7  
8         return resultado2;  
9     }  
10 }
```

Sobrecarga de Método

```
1 public class Teste {  
2  
3     public static void main(String args[]) {  
4  
5         Classe2 Obj2 = new Classe2();  
6  
7         Obj2.  
8     }  
9 }  
10
```

- ◆ calcular(float valor) : float - Classe2
- ◆ calcular(int valor) : int - Classe1
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Métodos com assinaturas diferentes

Exemplo

```
class GeradorDeExtrato {  
  
    public void imprimeExtratoBasico(ContaPoupanca cp) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cp.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaCorrente cc) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cc.getSaldo());  
    }  
  
    public void imprimeExtratoBasico(ContaSalario cs) {  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");  
        Date agora = new Date();  
  
        System.out.println("DATA: " + sdf.format(agora));  
        System.out.println("SALDO: " + cs.getSaldo());  
    }  
}
```

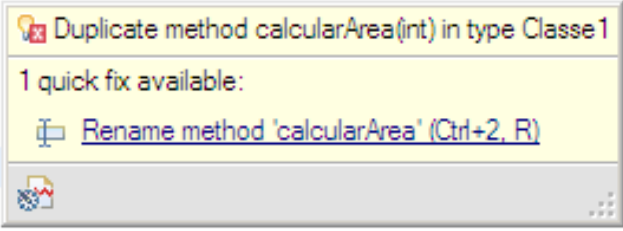
Tipos diferentes

Polimorfismo

■ Definição dos métodos

- Um mesmo nome de método é usado em diferentes classes , OU
- Usado na mesma classe com quantidade/tipo de parâmetros diferentes

```
1 public class Classe1 {  
2  
3     private int valor;  
4  
5     public int calcularArea(int valor){  
6         return 0;  
7     }  
8  
9     public int calcularArea(int valor){  
10        return 0;  
11    }  
12 }  
13  
14
```



Erro: métodos idênticos

Os parâmetros devem ser diferentes, ou deve-se renomear um dos métodos (não faz sentido se ambos realizam a mesma coisa)

Polimorfismo

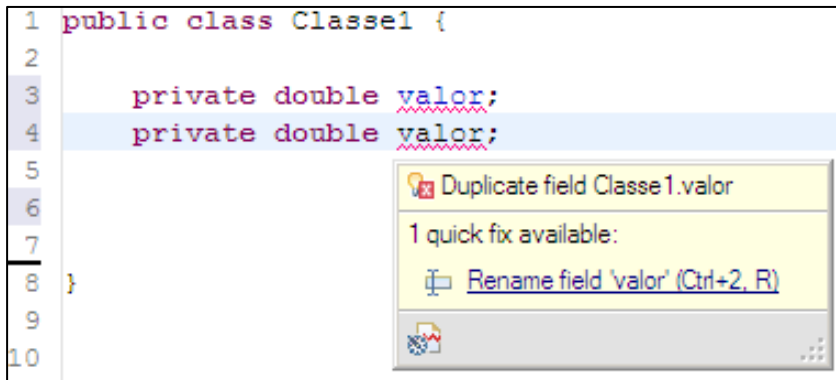
```
1 public class Classe1 {  
2  
3     private int valor;  
4  
5     public int calcularArea(int valor){  
6         return 0;  
7     }  
8  
9     public float calcularArea(float valor){  
10        return 0;  
11    }  
12 }
```

Assinatura do método modificado - tipo do parâmetro alterado. A quantidade continua a mesma (mas poderia ser alterado - funcionaria)

Sobrescrita de Atributo

- Relembrando: Sobrecrita de atributos
 - Não se sobreescreve atributos

```
1 public class Classe1 {  
2  
3     private double valor;  
4     private double valor;  
5  
6  
7  
8 }  
9  
10
```



Declarar duas variáveis idênticas em uma classe é um erro de programação

Há algum problema se um atributo idêntico for declarado em classes diferentes?

Sobrescrita de Atributo

```
1 public class Classe1 {  
2  
3     protected double valor;  
4 }
```



```
1 public class Classe2 extends Classe1 {  
2  
3     protected double valor;  
4 }
```



```
1 public class Classe3 extends Classe2 {  
2  
3     protected boolean valor;  
4 }
```

```
1 public class Teste {  
2  
3     public static void main(String args[]) {  
4  
5         Classe3 c13 = new Classe3();  
6  
7         c13.  
8     }  
9 }  
10
```

- valor : boolean - Classe3
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

De onde vem o atributo
do objeto c13?

Deveria ter sido dado
outro nome

Sobrescrita de Atributo

```
1 public class Teste {  
2  
3     public static void main(String args[]) {  
4  
5         Classe3 c13 = new Classe3();  
6  
7         c13.  
8     }  
9 }  
10
```

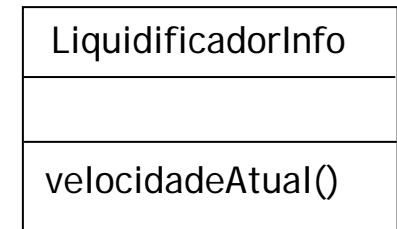
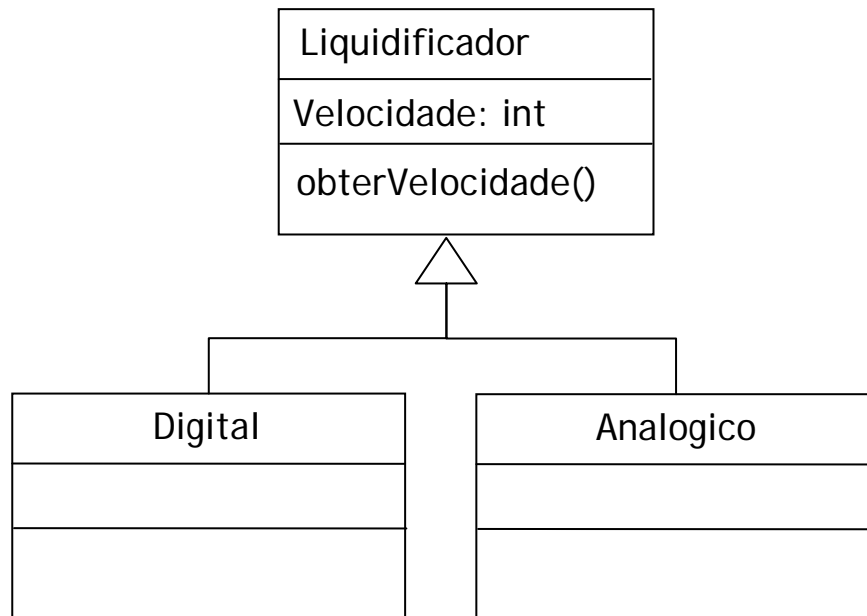
- ◇ outroNomeClasse2 : double - Classe2
- ◇ outroNomeClasse3 : boolean - Classe3
- ◇ valor : double - Classe1
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

```
1 public class Teste {  
2  
3     public static void main(String args[]) {  
4  
5         Classe3 c13 = new Classe3();  
6
```

c13.

- ◇ outroNomeClasse3 : boolean - Classe3
- ◇ valor : double - Classe2
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- toString() : String - Object
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

Exercício 1



Imaginemos que seja necessário implementar numa classe **LiquidificadorInfo** um método que seja capaz de imprimir a velocidade atual de objetos liquidificador os quais podem ser tanto do tipo digital como analógico.

Exercício 1

```
public class LiquidificadorInfo {  
  
    public void velocidadeAtual(LiquidificadorAnalogico l) {  
        System.out.println("Veloc. Atual: "+l.obterVelocidade());  
    }  
  
    public void velocidadeAtual(LiquidificadorDigital l) {  
        System.out.println("Veloc. Atual: "+l.obterVelocidade());  
    }  
}
```

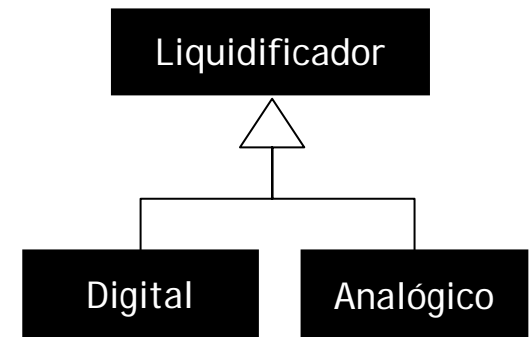
Forma possível para implementar o método `velocidadeAtual()` na classe `LiquidificadorInfo`

Embora esteja correto, existe uma desvantagem nesta implementação.

Qual desvantagem?
Como corrigi-la?

Solução

```
1 public class Liquidificador {  
2  
3     public int velocidade;  
4  
5     public int obterVelocidade() {  
6         return velocidade;  
7     }  
8 }
```

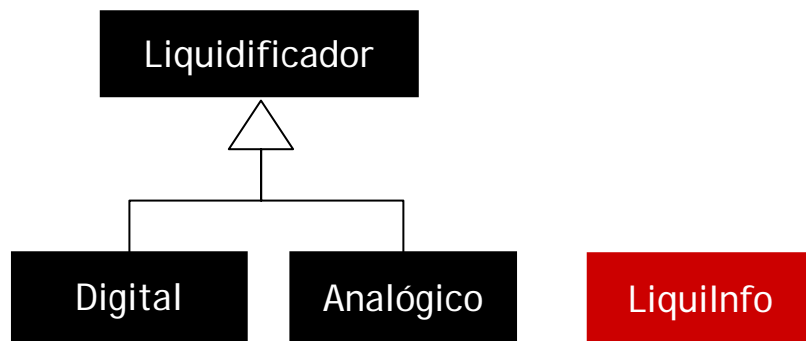


```
1 public class Digital extends Liquidificador{  
2 }
```

```
1 public class Analogico extends Liquidificador{  
2 }
```

Solução

```
1 public class LiquiInfo {  
2  
3     public void velocidadeAtual(Liquidificador l) {  
4         System.out.println("Veloc. Atual: "+l.obterVelocidade());  
5     }  
6 }
```



Solução - generalizar o método VelocidadeAtual. Neste caso, o método obterVelocidade irá depender do tipo do objeto passado como parâmetro

Solução

```
1 public class Teste {  
2  
3 public static void main(String args[]){  
4  
5     LiquiInfo li = new LiquiInfo();  
6     Digital d = new Digital();  
7     Analogico a = new Analogico();  
8  
9     d.velocidade = 20;  
10    a.velocidade = 30;  
11  
12    li.velocidadeAtual(d);  
13    li.velocidadeAtual(a);  
14 }  
15 }
```



```
1 public class LiquiInfo {  
2  
3 public void velocidadeAtual(Liquidificador l) {  
4     System.out.println("Veloc. Atual: "+l.obterVelocidade());  
5 }  
6 }
```

Solução

```
1 public class Teste {
2
3     public static void main(String args[]){
4
5         LiquiInfo li = new LiquiInfo();
6         Digital d = new Digital();
7         Analogico a = new Analogico();
8
9         d.velocidade = 20;
10        a.velocidade = 30;
11
12        li.velocidadeAtual(d);
13        li.velocidadeAtual(a);
14    }
15 }
```

< Problems @ Javadoc Declaration Console Progress

<terminated> Teste [Java Application] C:\Program Files\Java\jdk1.6.0_21\jre\bin\

Veloc. Atual: 20

Veloc. Atual: 30

Solução

```
1 public class Teste {  
2  
3 public static void main(String args[])  
4  
5     LiquiInfo li = new LiquiInfo();  
6     Digital d = new Digital();  
7     Analogico a = new Analogico();  
8  
9     d.velocidade = 20;  
10    a.velocidade = 30;  
11  
12    li.velocidadeAtual(d);  
13    li.velocidadeAtual(a);  
14 }  
15 }
```

```
1 public class Digital extends Liquidificador{  
2 }
```

```
1 public class Analogico extends Liquidificador{  
2 }
```

```
1 public class Liquidificador {  
2  
3     public int velocidade;  
4  
5 public int obterVelocidade() {  
6         return velocidade;  
7     }  
8 }
```

2

```
public class LiquiInfo {  
2  
3 public void velocidadeAtual(Liquidificador l) {  
4     System.out.println("Veloc. Atual: " + l.obterVelocidade());  
5 }  
6 }
```

3

Resumo - Polimorfismo

■ Polimorfismo

■ De herança - sobrescrita

- Métodos com mesma assinatura em classes diferentes
- Entende-se que, neste caso, suas implementações serão diferentes
- OBS: métodos de mesma assinatura em uma mesma classe não é permitido

- Posso ter métodos idênticos em classes diferentes?
- Posso ter métodos idênticos em uma mesma classe?
- Posso ter métodos idênticos em uma relação de herança (super/subclasse)?

Resumo - Polimorfismo

```
1 public class Classe1{
2
3     public int calcular(int x){
4         System.out.println("Oi");
5         return 0;
6     }
7
8     public int calcular(int x){
9         System.out.println("Oi");
10        return 0;
11    }
12 }
13
```

2

Duplicate method calcular(int) in type Classe1

1 quick fix available:

Rename method 'calcular'

```
1 public class Classe1{
2
3     public int calcular(int x){
4         System.out.println("Oi");
5         return 0;
6     }
7 }
```

```
1 public class Classe2{
2
3     public int calcular(int x){
4         System.out.println("Oi");
5         return 0;
6     }
7 }
```

```
1 public class Classe2 extends Classe1{
2
3     public int calcular(int x){
4         System.out.println("Oi");
5         return 0;
6     }
7 }
```

3

1

Resumo - Polimorfismo

■ Polimorfismo

■ Paramétrico - sobrecarga

- Métodos com diferentes assinaturas em uma mesma classe (ou diferentes)
- Nada impede que as implementações idênticas (não faz muito sentido)

```
1 public class Classe1 {  
2  
3     public int calcular(int x){  
4         System.out.println("Oi");  
5         return 0;  
6     }  
7  
8     public double calcular(double x){  
9         System.out.println("Oi");  
10        return 0;  
11    }  
12 }
```

```
1 public class Teste {  
2  
3     public static void main(String args[]){  
4  
5         Classe1 c1 = new Classe1();  
6  
7         c1.calcular(50);  
8         c1.calcular(50.7);  
9     }  
10 }
```

Problems Javadoc Declaration Console Progress

<terminated> Teste [Java Application] C:\Program Files\Java\jdk1.6.0_21\jre\bin\j
Oi
Oi

Resumo - Polimorfismo

■ Polimorfismo

■ Paramétrico - sobrecarga

- Se os dois métodos recebem parâmetros diferentes, entende-se que os mesmos serão usados na implementação. Logo, elas serão diferentes
- Pode-se tentar “esconder” o polimorfismo paramétrico. De início teríamos, pelo menos, dois métodos com nomes iguais, mas parâmetros diferentes (embora no mesmo número). As implementações seriam diferentes, visto que trabalham com parâmetros diferentes. Pode-se pensar em gerar um único método (geral) que substitua os dois anteriores - a assinatura é única e o que difere é o que se passa como parâmetro.
- Analogia do processador de suco de frutas

Polimorfismo

Orientação a Objetos - DCC025

Prof. Edmar Welington Oliveira
oliveira.edmar@ufjf.edu.br

Universidade Federal de Juiz de Fora - UFJF
Departamento de Ciência da Computação - DCC