Polimorfismo e abstração de classes

Prof. Renato Pimentel

2023/2

Sumário

Polimorfismo I

Termo originário do grego *poly* (muitas) + *morpho* (formas).

O **polimorfismo** em POO é a habilidade de objetos de uma ou mais classes em responder a uma mesma mensagem de *formas diferentes*.

Polimorfismo II

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:
  - Processar diversos tipos de dados;
  - Processar os dados de formas distintas;
  - Podem fazer um mesmo objeto ter comportamentos diferentes para uma mesma ação/solicitação.

Polimorfismo III

O polimorfismo pode ocorrer de duas maneiras:

- **Sobrecarga** (*Overloading*)
- **Sobreposição** ou **sobreescrita** (*Overriding*)

Alguns autores não classificam a sobrecarga como um tipo de polimorfismo.

Sobrecarga I

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- **Assinatura**: relacionada ao *número* e *tipo* dos parâmetros.

Resolvido pelo compilador em tempo de compilação:

- A assinatura diferente permite ao compilador dizer qual dos *sinônimos* será utilizado.

Exemplo: quando definimos diferentes construtores em uma classe, cada um recebendo parâmetros diferentes.

Sobrecarga II

Atenção

Mudar o nome dos parâmetros não é uma sobrecarga, o compilador diferencia o tipo, e não o nome dos parâmetros.

Exemplo: métodos
`f(int a, int b)` e
`f(int c, int d)`
numa mesma classe resultam em *erro de redeclaração*.

Sobrecarga III

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?
A **assinatura** de um método é composta pelo **nome do método** e pelos **tipos dos seus argumentos**, *independente dos nomes dos argumentos e do valor de retorno da função*.
Ex.: 2 assinaturas iguais:

```
float soma(float a, float b);
void soma(float op1, float op2);
```

Ex.: 2 assinaturas diferentes:

```
float soma(float a, float b);
double soma(double a, double b);
```

Sobrecarga IV

É implementada, normalmente, para métodos que devem executar operações semelhantes, usando uma lógica de programação diferente para diferentes tipos de dados.

```
1  public class Funcoes{
2     public static int quadrado( int x ) {
3        return x * x;
4     }
5
6     public static double quadrado( double y ) {
7        return y * y;
8     }
9  }
10 // ...
11 System.out.println("2 ao quadrado: " + Funcoes.quadrado(2));
12 System.out.println("PI ao quadrado: " + Funcoes.quadrado(Math.PI));
```


Sobrecarga V

Em muitos casos é necessário criar métodos que precisam de mais ou menos parâmetros, ou até precisem de parâmetros de tipos diferentes.

```
1 Fracao f1 = new Fracao();  // exemplo início curso: num=0,den=1
2 Fracao f2 = new Fracao(f1); // copy constructor: copia conteúdo de f1
3 Fracao f3 = new Fracao(5);  // num=5, den=1 (inteiro)
4 Fracao f4 = new Fracao(5,2);
5
6 // Exercício: como seria a implementação dos construtores das linhas 2
      e 3 dentro da classe Fracao?
```

Sobrescrita I

Conceito já visto em **herança**:

- Permite a redefinição do funcionamento de um método herdado de uma *classe base*.
- A *classe derivada* tem uma função **com a mesma assinatura** da classe base, mas **funcionamento diferente**;
- O método na classe derivada **sobrepõe** a função na classe base.

Sobrescrita II

Polimorfismo estático × Polimorfismo dinâmico

Sobrescrita: **polimorfismo dinâmico** – envolve 2 classes (classe derivada herda e redefine método da classe base); **Polimorfismo estático** – métodos com mesmo nome e assinaturas diferentes na mesma classe (sobrecarga).

Sobrescrita III

Indicações para uso da sobrescrita:

- A implementação do método na classe base não é adequada na classe derivada;
- A classe base não oferece implementação para o método, somente a declaração;
- A classe derivada pretende estender as funcionalidades da classe base.

Sobrescrita IV

Exemplo: considere as seguintes classes:

```
public class Figura {
   ...
   public void desenha(Graphics g) {
      ...
   }
}
```

```
public class Retangulo extends Figura {
...
   public void desenha(Graphics g) {
      g.drawRect(x, y, lado, lado2);
   }
}
```

Sobrescrita V

```
public class Circulo extends Figura {
   ...
   public void desenha(Graphics g) {
      g.drawCircle(x, y, raio);
   }
}
```


Sobrescrita VI

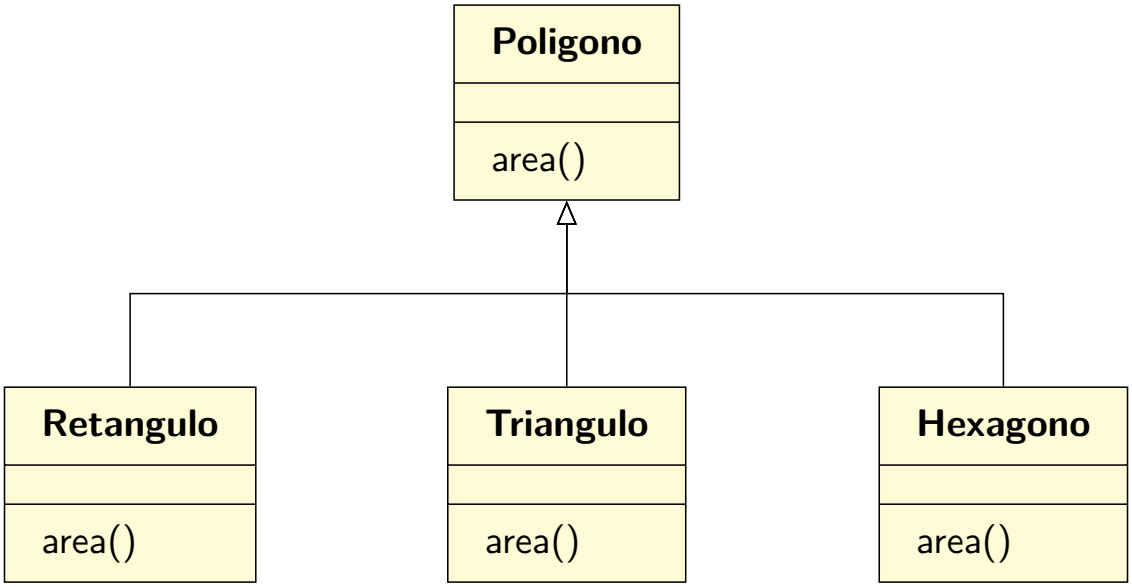
Na classe principal:

```
...
for (int i = 0; i < desenhos.size(); ++i) {
   Figura x = desenhos[i];
   x.desenha(g);
}
...
```

Na ocasião desta chamada, será decidido *automaticamente* qual implementação será invocada, dependendo do objeto: esta decisão é denominada **ligação dinâmica** (**dynamic binding**).

Exemplo de polimorfismo I

Considere o polimorfismo a seguir, *em métodos*:



Exemplo de polimorfismo II

Seguindo o exemplo, podemos observar o polimorfismo *nas variáveis*:

- Uma variável do tipo `Poligono` pode assumir a forma de `Poligono`, `Triangulo`, `Retangulo`, etc.

```
1 Poligono p1, p2, p3;
2
3 p1 = new Poligono();
4 ...
5 p2 = new Triangulo();
6 ...
7 p3 = new Retangulo();
```

Exemplo **sem** polimorfismo I

Área total de um *array* de polígonos, usando exemplo anterior, mas **sem sobrescrita de** `area()`:

```
1  double areaTotal() {
2     double areaTotal = 0;
3     for (int i = 0; i < MAXPOLIG; ++i) {
4        if (pol[i] instanceof Poligono)
5           areaTotal +=
6           pol[i].areaPoligono();
7        else if (pol[i] instanceof Triangulo)
8           areaTotal +=
9           pol[i].areaTriangulo();
10       else if (pol[i] instanceof Retangulo)
11          areaTotal +=
12          pol[i].areaRetangulo();
```


Exemplo **sem** polimorfismo II

```
13       else if (pol[i] instanceof Hexagono)
14          areaTotal +=
15          pol[i].areaHexagono();
16       return areaTotal;
17    }
18 }
```

`instanceof`: palavra reservada para testar se objeto é de determinada classe, retornando `true` quando for o caso, e `false` caso contrário.

Exemplo **com** polimorfismo

Usamos polimorfismo de `area()` como no diagrama de classes visto:

```
1 double areaTotal() {
2    double areaTotal = 0;
3    for (int i = 0; i < MAXPOLIG; ++i) {
4       areaTotal += pol[i].area();
5    return areaTotal;
6 }
```

Rápido, enxuto e fácil de entender:

- O acréscimo de uma nova subclasse de `Poligono` **não altera nenhuma linha** do código acima.

Benefícios do polimorfismo

Legibilidade do código:

- O mesmo nome para a mesma operação (método) facilita o aprendizado e melhora a legibilidade.

Código de menor tamanho:

- Código mais claro, enxuto e elegante.

Flexibilidade:

- Pode-se incluir novas classes sem alterar o código que a manipulará.

Classe abstrata I

Uma classe abstrata – ou **classe virtual** – é uma classe incompleta onde alguns ou todos os seus métodos não possuem implementação

Todas as classes vistas até este ponto não são abstratas, são *classes concretas*.

Classe abstrata II

Quando usamos herança, em diversas ocasiões as classes base são bastante *genéricas* (principalmente se houver *vários níveis de herança*);
Neste caso, pode-se implementar classes que definem *comportamentos genéricos* – as **classes abstratas**:

- A essência da superclasse é definida e pode ser *parcialmente* implementada;
- Detalhes são definidos em subclasses especializadas;
- **Não** podem ser instanciadas (servem apenas para reunir características comuns dos descendentes).

Java

Palavra reservada `abstract`

Classe abstrata III

Exemplo:

```java
public abstract class Conta {
    private int num;
    private float saldo;
}
```

```java
public class Poupanca extends Conta {
    ...
}
```

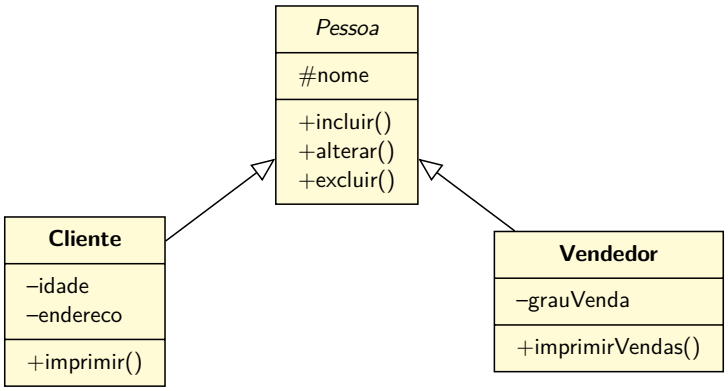
```java
public class ContaEspecial extends Conta {
    ...
}
```


Classe abstrata IV

```java
Conta c; // esta linha está ok
c = new Conta(); // ERRO: não posso criar objeto de
    classe abstrata
c = new ContaEspecial(); // ok
```

Classe abstrata V

Outro exemplo (em UML, nome de classe abstrata é escrito em *itálico*):



- A classe `Pessoa` existe para reunir as características.
- Um objeto efetivo dentro de uma loja deve ser cliente ou vendedor. *Não existe apenas pessoa*.

Métodos abstratos I

Métodos abstratos são métodos definidos exclusivamente dentro de classes abstratas, mas *não são* implementados nas mesmas (apenas sua *assinatura* é especificada).

- Os métodos abstratos devem ser **obrigatoriamente** implementados em **toda classe herdeira (concreta)** da classe abstrata em que são definidos.

Declarar um método como abstrato é uma forma de obrigar o programador a redefinir esse método em todas as subclasses para as quais se deseja criar objetos.

Métodos abstratos II

Exemplo:

```java
public abstract class Forma {
    public abstract double perimetro();
    public abstract double area();
}
```

```java
public class Circulo extends Forma {
    public double perimetro() {
        return 2.0*Math.PI*this.raio;
    }
    public double area() {
        return Math.PI*Math.pow(this.raio,2.0);
    }
}
```


Métodos abstratos III

```java
public class Quadrado extends Forma {
    public double perimetro() {
        return 4.0*this.lado;
    }
    public double area() {
        return Math.pow(this.lado,2.0);
    }
}
```

Exercício

- Crie um algoritmo para instanciar os objetos `BemTevi`, `Papagaio`, `Cachorro` e `Vaca`. Na prática, nunca iremos instanciar um `Animal`. A classe serve apenas para a definição de mamíferos e pássaros (subclasses). Da mesma forma, não instanciamos `Mamifero` nem `Passaro`. Somente instanciamos objetos `BemTevi`, `Papagaio`, `Cachorro` e `Vaca`. Logo, `Animal`, `Mamifero` e `Passaro` são classes abstratas.
- Crie um método abstrato em `Animal`, implementando-o nas classes concretas.

Referências

1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.

Os slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU