N_PROG SIST II_A1 - Texto de apoio

Site: EAD Mackenzie
Tema: PROGRAMACAO DE SISTEMAS II {TURMA 03A} 2023/2
Livro: N_PROG SIST II_A1 - Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: quinta, 3 ago 2023, 10:47

Descrição

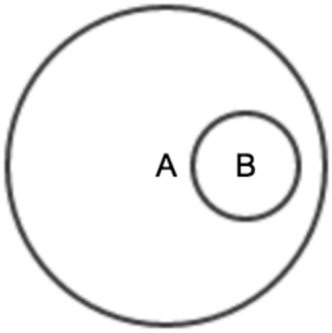
Índice

1. CLASSE ABSTRATA E INTERFACE

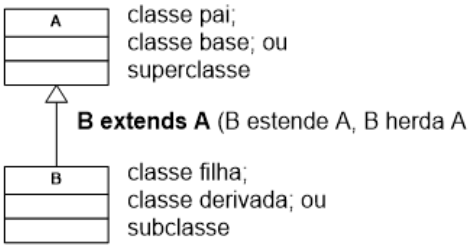
Herança

O conceito de **herança** é um dos pilares da Programação Orientada a Objetos (POO), juntamente com **encapsulamento** e **polimorfismo**.

Herança é um tipo especial de **relacionamento entre classes do tipo generalização-especialização**, onde uma **subclasse (mais especializada B)** herda parte (ou o todo) de uma **superclasse (mais genérica A)**.



No diagrama abaixo, a **subclasse B (mais especializada)** herda parte (ou o todo) da **superclasse A (mais genérica)**.



Um dos objetivos principais da herança é fazer **reuso de um código** já implementado nas superclasses e adicionar os detalhes necessários para que a subclasse que esteja estendendo a superclasse seja mais especializada. Assim, uma classe derivada pode:

- herdar atributos e métodos da classe base;
- redefinir métodos herdados.
- definir novos atributos e métodos;

Em Java, **toda classe** que não estende especificamente uma outra classe é uma **subclasse** da classe **Object**.

Considere as classes A e B definidas abaixo:

```
// subclasse
// classe filha
public class B extends A{
   private int b;
   public B(int a, int b){
      // invoca construtor
      // da classe A
      super( a );
      this.b=b;
   }
}
```

```java
// superclasse
// classe pai
public class A{
   private int a;
   public A(int a){
      this.a=a;
   }
}
```

No exemplo acima a palavra reservada **extends** estabelece a relação de herança entre a **classe A** e **classe B**, ou seja, a **classe B** é derivada (filha) da **classe A**. Note que, no exemplo, o construtor da **classe A** (classe pai) é acessado com o apontador **super** para inicializar o valor do atributo **int a** dentro do construtor da **classe B** (classe filha). O apontador **super** pode ser utilizado de três formas:

- **super(....)** invoca o construtor da superclasse;
- **super.id** acessa o atributo da superclasse identificado por **id**, desde que o atributo tenha sido definido com o modificador de acesso protected.
- **super.m(...)** acessa o método da superclasse identificado por **m**, e o método pode ser público ou privado que poderá ser acessado na classe filha.

Quando não houver confusão entre o nome de identificadores e os métodos da superclasse para subclasse, o operador **super** pode ser suprimido.

1.1. Polimorfismo

O polimorfismo é princípio que permite a reutilização contínua dos códigos, no contexto da programação orientada a objetos, permite também que um método assuma “formas” diferentes das quais foram implementadas inicialmente, ou seja, agir de modo diferenciado; basicamente, existem dois tipos de polimorfismo **sobrecarga** e **sobrescrita**.

1.2. Polimorfismo estático

A **sobrecarga (=overload)** ou o **polimorfismo estático** permite definir no corpo de uma mesma classe mais de um método com o mesmo nome, entretanto, eles devem obrigatoriamente possuir parâmetros e/ou retorno diferentes para funcionar **(assinatura do método)**. O polimorfismo de **sobrecarga** é útil para definir mais de um construtor para uma mesma classe, a fim de oferecer diversas maneiras para instanciar e inicializar os objetos de uma classe.

A escolha de qual método será invocado é definida durante a compilação do programa **(tempo de compilação)**. Em função da assinatura dos métodos sobrecarregados, esse tipo de escolha é denominado ligação prematura ou *early binding*. Abaixo, um exemplo polimorfismo de sobrecarga na **classe A**:

```
public class A{
  int calcula() { return 1; };
  int calcula( int x ) { return 1+x; };
}
```

Note que o método **calcula()** é sobrecarregado, o primeiro não possui parâmetro e o segundo possui como parâmetro **int x**. Considerando esses dois métodos, poderíamos ter as seguintes chamadas :

```
public static void main(String[] args) {
    A objA = new A();
    System.out.println(objA.calcula());
    System.out.println(objA.calcula(10));
}
```

Temos como resultado, na primeira chamada **objA.calcula()**, o valor igual a 1, pois é invocado o método sem parâmetro que retorna 1; na chamada **objA.calcula(10)**, temos como saída o valor **11**, pois é invocado o respectivo método que recebe um valor como argumento e adiciona mais 1 ao argumento. Observe que o compilador da linguagem Java consegue escolher qual método será invocado em tempo de compilação, avaliando o parâmetro do método.

1.3. Polimorfismo dinâmico

O **polimorfismo de sobrescrita (=override)** ou **inclusão** consiste em permitir que classes derivadas de uma mesma superclasse invoquem métodos que têm a mesma identificação **(assinatura)**, mas comportamentos distintos. Nesse tipo de polimorfismo, **é necessário** que os métodos tenham exatamente a mesma identificação **(assinatura do método)**. Esse tipo de polimorfismo acontece na herança, quando a subclasse sobrepõe o método original; nesse caso, a escolha do método se dá em **tempo de execução (ligação tardia ou late binding)** e não mais em tempo de compilação como no polimorfismo estático. Por essa razão o **polimorfismo de sobrescrita** também é denominado **polimorfismo dinâmico**. Abaixo, um exemplo polimorfismo de sobrescrita:

```
public class A{
  int calcula() { return 1; };
  String metodoA() { return "classe A";}
}
```

```
public class B extends A{
  @Override
  int calcula() { return 2; };
  String metodoB() { return "classe B"; }
}
```

Veja que o método **calcula()** da **classe A** foi sobrescrito na **classe B**; para deixar explícito a sobrescrito, é usada a anotação **@Override**.

Na linguagem Java existem diversas anotações e cada uma terá um efeito diferente sobre seu código e elas são utilizadas para passar informações ao compilador ou até gerar automaticamente, no momento da compilação ou da execução, código-fonte, arquivos XML, arquivos de documentação etc.

O método **metodoA()** não foi sobrescrito na **classe B**, mas na **classe B** temos a definição de um outro método **metodoB()**. A partir das classes acima, podemos ter o seguinte trecho de código:

```
System.out.print("digite A ou B para instanciar um objeto das classes:");
String opcao = ler.nextLine();
A objA;
if( opcao.equals("A"))
    objA = new A();
else
    objA = new B();

System.out.println("saida:"+objA.calcula());
```

Você conseguiria dizer qual seria a saída do trecho acima? Daria para prever isso em tempo de compilação?

Resposta: Não, pois a escolha do método que será invocado se dá em tempo de execução (ligação tardia ou late binding) em razão da opção da classe digitada quando o programa estiver executando; consequentemente, a variável de referência objA pode receber tanto um objeto da classe A ou da classe B. Isto só é possível porque a classe B é um subtipo (filha ou derivada) da super classe A, ou seja, uma variável de referência da classe pai suporta objetos de todas as suas classes filhas.

Na hierarquia de classes, os objetos da **classe B** tem acesso a todos os métodos públicos definidos na **classe A**, e a **ligação tardia** define que a busca da implementação de um método é realizada primeiro na subclasse (filha) e depois na superclasse (pai), ou seja, quanto temos o **polimorfismo inclusão**, o método definido na classe filha sobrescreve o método da classe pai.

1.4. Classe Abstrata

Em muitos casos, desejamos definir uma classe geral que representa objetos de maneira genérica, mas que não faz sentido possuir uma instância. Para evitar que o sistema possa instanciar uma classe, podemos declará-la como **abstrata**, assim a classe serve como **um modelo** de como as classes que herdarem características dela devem se comportar. (Herdem os atributos/ as propriedades e seus métodos).

Uma **classe abstrata** é uma classe declarada como **abstrata** e possui **métodos abstratos** e um **método abstrato** é um método que não contém código além de sua assinatura e deve obrigatoriamente ser implementado na classe filha (classe que herda).

Como exemplo, suponha que queiramos desenvolver uma aplicação para manipular **figuras geométricas** planas (2D), a aplicação poderá trabalhar com várias figuras geométricas diferentes, tais como: retas, retângulos, triângulos, quadrados, círculos etc. A classe abstrata que serve como modelo para todas as outras figuras geométricas compartilha uma única característica em comum: a cor (atributo), e define que todas as figuras geométricas devem obrigatoriamente disponibilizar três métodos: um para **calcular sua área**, outro para **calcular seu perímetro**, e o último para **comparar se uma figura é igual a outra**.

Para definir de forma geral como cada uma das figuras geométricas deve se comportar, proporemos uma classe modelo para todas as outras classes que forem figuras geométricas, ou seja, uma **classe FiguraGeometrica**. A classe **abstrata FiguraGeometrica** pode ser usada para:

- definir os atributos comuns a todas as figuras;
- servir como classe base para as subclasses;
- obrigar as classes derivadas a implementar os métodos abstratos.

Assim, a **classe FiguraGeometrica** não poderá ser instanciada, mas suas subclasses poderão.

Por conta das características acima, **a classe FiguraGeometrica** deverá ser uma classe abstrata definida conforme abaixo:

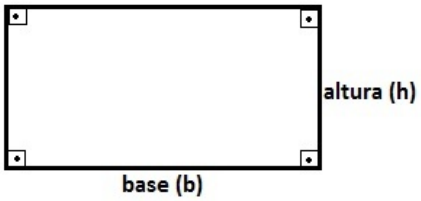
```
public abstract class FiguraGeometrica{
    private String cor;
    public FiguraGeometrica(String cor) {
        this.cor = cor;
    }
    public String getCor() {
        return this.cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    public abstract boolean compare(FiguraGeometrica outra);
    public abstract double Area();
    public abstract double Perimetro();
}
```

Note que a **classe abstrata FiguraGeometrica** possui somente um atributo **private String** cor e **três métodos abstratos:**

- **public boolean compare(FiguraGeometrica o)**: compara se duas figuras geométricas são iguais, com o critério definido na classe filha.
- **public double Area()**: calcula a área.
- **public double Perimetro()**: calcula o perímetro.

No exemplo, a **classe abstrata FiguraGeometrica**, além dos métodos abstratos, possui dois métodos concretos **getCor()** que retornam o valor do atributo cor e o método **setCor()** que atualiza a cor da figura geométrica. Esses métodos podem ser utilizados pelas classes filhas da **classe abstrata FiguraGeometrica**.

A partir da classe abstrata **FiguraGeometrica**, podemos definir agora uma outra figura geométrica como um retângulo. Um **retângulo** é uma figura geométrica plana formada por quatro lados (quadrilátero). Por ser um tipo especial de paralelogramo (seus lados opostos são paralelos), um retângulo possui quatro lados, dois a dois congruentes. Por convenção, chama-se de bases do **retângulo** os lados de maior comprimento, e de altura, os lados de menor comprimento.




```
public class Retangulo extends FiguraGeometrica {
    private double base, altura;
    public Retangulo(String cor, double base, double altura){
        super(cor);
        this.base = base;
        this.altura = altura;
    }
    @Override
    public boolean compare(FiguraGeometrica o) {
          Retangulo r = (Retangulo) o;
          return r.base==this.base && r.altura==this.altura
                && getCor().equals(r.getCor());
    }
    @Override
    public double Area() {
        return this.base * this.altura;
    }
    @Override
    public double Perimetro() {
        return 2 * ( this.base + this.altura );
    }
}
```

A **classe Retangulo** representa a figura geométrica retângulo e é derivada da classe **FiguraGeometrica**. A herança é indicada na declaração da classe **(class Retangulo extends FiguraGeometrica)** e, por conta disso, ela deve implementar todos os métodos abstratos definidos na **classe FiguraGeometrica**. Para que tenhamos o polimorfismo de sobrescrita, o método sobrescrito **compare()** deve continuar recebendo por parâmetro um objeto do tipo **FiguraGeometrica** no **parâmetro** o, ou seja, não podemos mudar a assinatura do método compare().

```
@Override
public boolean compare(FiguraGeometrica o) {
      Retangulo r = (Retangulo) o;
      return r.base==this.base && r.altura==this.altura
            && getCor().equals(r.getCor());
}
```

No corpo do método realizamos uma **coerção** (casting) do **parâmetro** o para que ele seja mapeado internamente para um objeto da **classe Retangulo**, de acordo com relações de equivalência existentes entre as classes.

Considere que agora queremos criar uma **figura geométrica Reta**. Basicamente, a reta terá como atributo seu **comprimento**. Se fizermos a **classe Reta** estender a classe abstrata **FiguraGeometrica**, teríamos que, obrigatoriamente, implementar os métodos abstratos **Area()** e **Perimetro()**, mas uma reta não tem **área** nem **perímetro**.

Como adicionar esses recursos (métodos) somente a algumas classes que são subclasses da **classe abstrata FiguraGeometrica**?

Resposta: Nesse caso, teríamos de definir essas funcionalidades somente para algumas classes por meio de um contrato.

1.5. Interface

Uma **interface** define um **contrato público** que **deve ser seguido** por todas as **classes** que a **implementem**. Neste contrato, especificamos as assinaturas dos métodos que vêm seguidos pelas classes, além de possíveis constantes compartilhadas por elas; assim, uma **interface** define as assinaturas dos métodos sem implementá-los, ou seja, define o que o **objeto deve fazer**, e não como ele faz, sendo uma outra forma de incluir **funcionalidade nas classes**.

As **interfaces** têm similaridades e diferenças em relação a **classes abstratas**, conforme podemos ver a seguir:

INTERFACES	CLASSES ABSTRATAS
Uma **interface** define apenas um conjunto de métodos abstratos.	Uma **classe abstrata**, além de definir métodos abstratos, pode implementar métodos concretos.
Uma interface só pode definir constantes, exemplo: **static final float PI=3.145f**;	Classes abstratas podem possuir **atributos** (variáveis).
Interfaces são definidas de forma independente e uma **classe** pode implementar **diversas interfaces**.	Uma **classe** só pode estender apenas uma **classe abstrata**.

Uma observação muito importante é que uma classe **pode implementar mais de uma interface** e a implementação de uma ou mais interfaces não exclui a possibilidade de herança com classes abstratas ou classes concretas.

Para nosso problema, a melhor solução para descrever a exigência das funcionalidades de **calcular área** e **perímetro** seria utilizar uma interface.

A **interface Calcula** possui dois métodos que deverão ser implementados pelas classes:

```
public interface Calcula {
    public double Area();
    public double Perimetro();
}
```

Considerando a **interface Calcula**, a **classe abstrata FiguraGeometrica** pode ser reescrita da seguinte forma:

```
public abstract class FiguraGeometrica{
    private String cor;
    public FiguraGeometrica(String cor) {
        this.cor = cor;
    }
    public String getCor() {
        return this.cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    // assinatura do único método abstrato
    public abstract boolean compare(FiguraGeometrica o);
}
```

Com isso, conseguimos escrever a implementação da **classe Reta**, com a modificação acima. Agora, é possível estender a classe abstrata **FiguraGeometrica**, uma vez que a classe não possui os métodos abstratos **Area()** e **Perimetro()**.

```
public class Reta extends FiguraGeometrica{
    private double comprimento;
    public Reta(String cor,double comprimento){
        // invoca o construtor da classe Figura
        super(cor);
        this.comprimento = comprimento;
    }
    public boolean compare(FiguraGeometrica o) {
          Reta r = (Reta) o;
          return this.comprimento == r.comprimento &&
                this.getCor().equals(r.getCor());
    }
}
```

E a **classe Retangulo** agora é reescrita considerando a **interface Calcula** e a nova **classe abstrata FiguraGeometrica**, note que a **classe Retangulo** estende a **classe FiguraGeometrica (extends FiguraGeometrica)** e implementa a **interface Calcula (implements Calcula)**.

```
public class Retangulo extends FiguraGeometrica implements Calcula{
    private double base;
    private double altura;
    public Retangulo(String cor, double base, double altura){
        super(cor);
        this.base = base;
        this.altura = altura;
    }
    public boolean compare(FiguraGeometrica o) {
          Retangulo r = (Retangulo) o;
          return r.base==this.base && r.altura==this.altura
                && this.getCor().equals(r.getCor());
    }
    public double Area() {
        return this.base * this.altura;
    }
    public double Perimetro() {
        return 2 * ( this.base + this.altura );
    }
}
```