Características das Linguagens de Programação I

Conteúdo: Herança

Prof. Dsc. Giomar Sequeiros
giomar@eng.uerj.br

Herança

Herança

- Herança: forma de reutilização de software
- Novas classes são criadas a partir de classes já existentes
- Absorvem atributos e comportamentos, e incluem os seus próprios
  - Sobrescrevem métodos: redefinem métodos herdados

- Subclasse herda de uma superclasse
  - Superclasse direta: subclasse herda explicitamente
  - Superclasse indireta: subclasse herda de dois ou mais níveis acima na hierarquia de classes

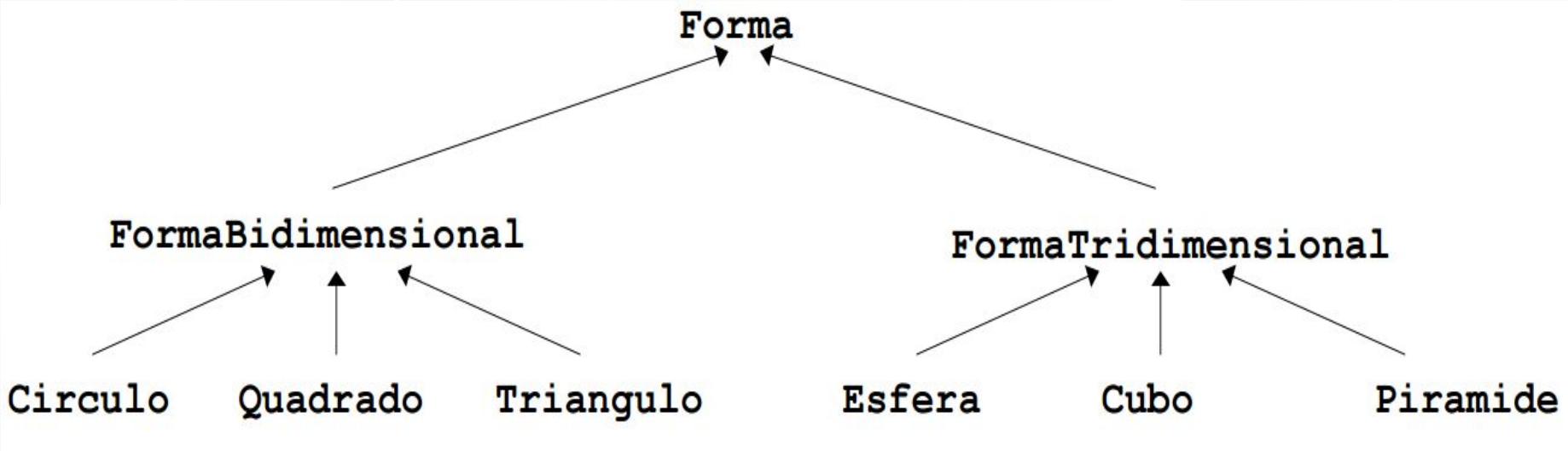
Herança

- Um dos conceitos de orientação a objetos que possibilita a reutilização de código é o conceito de herança. Pelo conceito de herança é possível criar uma nova classe a partir de outra classe já existente.
- Sintaxe

```
class NomeDaSubclasse extends NomeDaSuperclasse {
    //declaração de atributos
    //declaração de construtores e métodos
}
```

- Em Java, quando a superclasse não é indicada, a classe estende java.lang.Object
- Construtores não são herdados, porém, um construtor da superclasse imediata pode ser invocado pela subclasse
- Em Java só existe herança simples

Herança: Exemplo



Herança: Exemplo 1

- Considere a classe **Conta**

Classe base (superclasse)

```
public class Animal {
    String especie;
    int peso;

    public void comer() {
        System.out.println("O animal come.");
    }

    public void emitirSom() {
        System.out.println("O animal emite um som.");
    }
}
```

Classe filha (subclasse)

```
public class Cachorro extends Animal {
    String raca;

    public void latir() {
        System.out.println("Au au!");
    }
}
```

Herança: Exemplo 1

- Teste básico

```
public class Principal {
    public static void main(String[] args) {
        Cachorro meuCachorro  = new Cachorro();
        meuCachorro.especie = "Mamífero"; // Herda de Animal
        meuCachorro.peso = 10;              // Herda de Animal
        meuCachorro.raca = "Labrador";      // Próprio de Cachorro

        meuCachorro.comer();      // Método herdado de Animal
        meuCachorro.emitirSom(); // Método herdado de Animal
        meuCachorro.latir();      // Método próprio de Cachorro
    }
}
```

- Faça que os atributos das classes Animal e Cachorro sejam privados, adicione os métodos getter e setter e refaça o teste.

Herança: Exemplo 2

- Considere a classe **Conta**

Atributos

```
public class Conta {
    private int numero;
    private String titular;
    protected double saldo;
```

Construtores

```
    public Conta(int numero, String titular) {
        this.numero = numero;
        this.titular = titular;
        this.saldo = 0;
    }

    public Conta(int numero, String titular, double saldoInicial) {
        this.numero = numero;
        this.titular = titular;
        this.deposita(saldoInicial);
    }
```

Herança: Exemplo 2

- Considere a classe **Conta**

Getter e setters

```
public int getNumero() {
    return numero;
}

public String getTitular() {
    return titular;
}

public double getSaldo() {
    return saldo;
}
```

Métodos

```
public void deposita(double quantidade) {
    saldo += quantidade;
}

public void saca(double quantidade) {
    if(quantidade <= saldo)
        saldo -= quantidade;
    else
        System.out.println("Saldo insuficiente");
}
```

```
    @Override
    public String toString() {
        return "Conta [numero=" + numero + ", titular=" + titular + ", saldo=" + saldo + "]";
    }
}
```

Herança: Exemplo 2

- Vamos criar uma classe para representar as contas especiais de um banco. Uma conta especial é um tipo de conta que permite que o cliente efetue **saques acima** de seu saldo até um **limite**. Assim, criaremos uma classe ContaEspecial que herdará da classe Conta.

```
public class ContaEspecial extends Conta {
    private double limite;

    public double getLimite() {
        return limite;
    }
    public void setLimite(double limite) {
        this.limite = limite;
    }
    @Override
    public String toString() {
        return  super.toString() + " [limite=" + limite + "]";
    }
}
```

- Nesse caso dizemos que ContaEspecial é uma **subclasse ou classe filha** de Conta. Podemos também dizer que Conta é **ancestral ou classe pai de** ContaEspecial. Note que ContaEspecial define um tipo mais especializado de conta. Assim, ao mecanismo de criar novas classes herdando de outras é dado o nome de **especialização.**

Herança: Exemplo 2

- Agora suponha que tenhamos um outro tipo de conta: a ContaPoupanca. A ContaPoupanca tem tudo o que a Conta tem com um método a mais que permite atribuir um reajuste percentual ao saldo. Agora teríamos duas classes herdando da classe Conta Nesse contexto podemos dizer que a classe Conta **generaliza os conceitos de ContaEspecial e ContaPoupanca.**

```
public class ContaPoupanca extends Conta {

    //métodos
    public void reajustar(double percentual) {
        double reajuste = this.getSaldo()*percentual;
        this.deposita(reajuste);
    }

}
```

Herança: Construtores

- Caso você não tenha definido um construtor em sua superclasse, **não** será obrigado a definir construtores para as subclasses, pois Java utilizará o construtor padrão para a superclasse e para as subclasses.

- Porém, caso haja algum construtor definido na superclasse, **obrigatoriamente** você precisará criar ao menos um construtor para cada subclasse. Vale ressaltar que os construtores das subclasses utilizarão os construtores das superclasses pelo uso da palavra reservada super.

Herança: Construtores

Exemplo

- Criando o construtor para a Classe **ContaEspecial**

```
public ContaEspecial(int numero, String titular, double limite) {
      super(numero, titular);
      this.limite = limite;
 }
```

- Criando o construtor para a Classe **ContaPoupanca**

```
public ContaPoupanca(int numero, String titular) {
      super(numero, titular);
 }
```

Herança: Atributos protected

- Quando estudamos **encapsulamento aprendemos que devemos preferencialmente** manter os atributos com nível de acesso privado (private) de forma que para acessá-los outras classes precisem utilizar métodos.

- Mas, vimos também que há um nível de acesso protegido (**protected**) que faz com que o atributo se comporte como público para classes da mesma hierarquia ou do mesmo pacote e como privado para as demais classes.

- Reajustar a classe conta e considerar o atributo saldo como protected
- Modificar o método reajustar da classe ContaPoupanca

```
public class ContaPoupanca extends Conta {

    public void reajustar(double percentual) {
        double reajuste = this.saldo*percentual;
        this.deposita(reajuste);
    }
}
```

Herança: Atributos protected

- Em uma superclasse:
  - **public**
    - Acessível em qualquer classe
  - **private**
    - Accesível somente nos métodos da própria superclasse
  - **protected**
    - Proteção intermediária entre private and public
    - Somente acessível pelos métodos da superclasse ou de uma subclasse desta

- Métodos na subclasse
  - Podem se referir a membros **public** ou **protected** pelo nome

Herança: Exemplo 3

```java
public class Pessoa {
    private String nome;

    public Pessoa(String nome) {
        this.nome = nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return this.nome;
    }
}
```

```java
public class Aluno extends Pessoa {

    private String curso;

    public Aluno(String nome, String curso) {
        super(nome);
        this.curso = curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    public String getCurso() {
        return this.curso;
    }
}
```

Herança: Exemplo 4

- Classe **Ponto**
  - **Atributos** protected x, y
  - **métodos**: setPonto, getX, getY, sobrescreve toString

- Classe **Circulo** (extends **Ponto**)
  - **Atributos protected** raio
  - **métodos**: setRaio, getRaio, area, sobrescreve toString

- – Classe **Cilindro** (extends Circulo)
  - **Atributos** protected altura
  - métodos: setAltura, getAltura, area (superfície), volume, sobrescreve toString

Herança: Exemplo 4

```
public class Ponto {
    protected float x,y;

    public Ponto() {
        x = y = 0;
    }
    public Ponto(float x,float y) {
        setPonto(x,y);
    }

    public void setPonto(float x,float y) {
        this.x = x; this.y = y;
    }
    public float getX() { return x; }
    public float getY() { return y; }

    // O método toString() retorna uma representação
    //textual de um objeto
    public String toString() {
        return "[" + x + ", " + y + "]";
    }
}
```

```
public class Circulo extends Ponto {
    protected float raio;
    public Circulo() {  // construtor de Ponto é chamado
                        implicitamente!
        setRaio(0);
    }
    public Circulo(float x,float y,float raio) {
        super(x,y); // construtor de Ponto é chamado
                    explicitamente!
        setRaio(raio);
    }

    public void setRaio(float raio) {
        if(raio<=0)
            raio = 0;
        this.raio = raio;
    }
    public float getRaio() { return raio; }
    public float area() {
        return Math.PI * raio * raio;
     }
    public String toString() {
        return "Centro= " + super.toString()+ "Raio= " + raio;
     }
}
```

Herança: Exemplo 4

```
public class Cilindro extends Circulo {
    protected float altura;

    public Cilindro() {   // construtor de Circulo é chamado implicitamente
        setAltura(0);
    }
    public Cilindro(float x,float y,float raio,float alt) { // construtor de Circulo é chamado explicitamente!
        super(x,y,raio);
        setAltura(alt);
    }
    public void setAltura(float altura) {
         if(altura<=0) altura = 0; this.altura = altura;
    }
    public float getAltura() { return altura; }

    public float area() {
        return 2 * super.area() + Math.PI * raio * altura;
    }
    public float volume() {
        return super.area() * altura;
    }
    public String toString() {
        return super.toString()+ " Altura= "+ altura;
    }
}
```

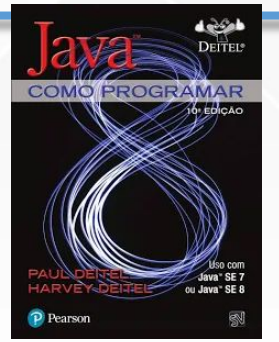
Exercício

- Testar as funcionalidades (criar uma classe Teste) dos projetos três projetos: Conta, Pessoa e Ponto

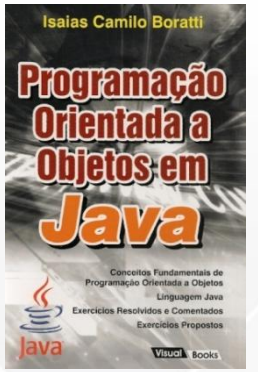
Referências

BIBLIOGRAFIA BÁSICA:

- DEITEL, Harvery M.. Java : como programar. 10ª ed. São Paulo: Pearson - Prentice Hall, 2017.



- BORATTI, Isaías Camilo. Programação Orientada a Objetos em Java : Conceitos Fundamentais de Programação Orientada a Objetos. 1ª ed. Florianópolis: VisualBooks, 2007.



- SIERRA, Kathy; BATES, Bert. Use a Cabeça! Java. 2ª ed. Rio de Janeiro: Alta Books, 2007.