Módulo 6

Programação Orientada a Objetos – Herança

Programação Orientada a Objetos I

Java

(Rone Ilídio)

Programação Orientada a Objetos

- Principais conceito
  - Classe
  - Herança
  - Encapsulamento
  - Polimorfismo
- Adicionais
  - Classes e membros final
  - Sobrecarga de métodos

Programação Orientada a Objetos

- Herança

  É a forma de utilização de *software* em que novas classes são criadas a partir de classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos que as novas classes exigem.

```
Public class Cliente{
    private String nome;
    private String telefone;
    public void setNome(String nome){
        this.nome = nome;
    }
    public String getNome(){
        return nome;
    }
    public void setTelefone(String telefone){
        this.telefone = telefone;
    }
    public String getTelefone(){
        return telefone;
    }
}
```

```
Public class PessoaJuridica extends Cliente{
    private String cnpj;
    public void setCnpj(String cnpj){
        this.cnpj = cnpj;
    }
    public String getCnpj(){
        return cnpj;
    }
}
```

```
Public class PessoaJuridica extends Cliente{
    private String cnpj;
    public void setCnpj(String cnpj){
        this.cnpj = cnpj;
    }
    public String getCnpj(){
        return cnpj;
    }
}
```

```
public final class PessoaFisica extends Cliente{
private String cpf;

public PessoaFisica(){
	cpf = "";
}

public void setCpf(String cpf){
	this.cpf = cpf;
}

public String getCpf(){
	return cpf;
}

public String retornaDados(){
	return "Nome=" + super.getNome() + "\nCPF=" + getCpf();
}
}
```

```
import javax.swing.*;
public class ControlaCliente extends JApplet{
   public void init(){
       PessoaFisica pf = new PessoaFisica();
      PessoaJuridica pj = new PessoaJuridica();

      pf.setNome("Maria");
      pf.setTelefone("00000000");
      pf.setCpf("111.111.111-11");

      pj.setNome("Butecu do Zé");
      pj.setTelefone("99999999");
      pj.setCnpj("0000.000.000-00");

     JOptionPane.showMessageDialog(null,"Nome:" + pj.getNome() +
                                  " Telefone:"+pj.getTelefone() + " CNPJ:"+pj.getCnpj());

     JOptionPane.showMessageDialog(null,"Nome:" + pf.getNome() +
                                  " Telefone:"+pf.getTelefone() + " CPF:"+pf.getCpf());
   }
}
```

Membros *static*

- Atributos static possuem somente um valor para todos os objetos de sua classe, ou seja, possuem escopo de classe
- Métodos static podem ser chamados sem a criação de objetos, utilizando-se somente o nome_classe.nome_metodo();

```
public class Equacoes {
  public static double valor= 3.14;
  public static double delta(double a,     double b, double c){
      return b*b - 4*a*c;
   }
  public static double raiz1(double a, double b, double c){
    return (-b + Math.sqrt(delta(a,b,c)))/(2*a);
  }
  public static double raiz2(double a, double b, double c){
    return (-b - Math.sqrt(delta(a,b,c)))/(2*a);
  }
}
```

```
import javax.swing.*;
public class Calcular extends JApplet {
  public void init(){
    double a=4, b = 16, c = 2;
    if(Equacoes.delta(a, b, c)<0)
      JOptionPane.showMessageDialog(null, "Não existe raizes reais");
   else
      JOptionPane.showMessageDialog(null, "X1:"+Equacoes.raiz1(a, b, c)+"\nX2:"+
                                                    Equacoes.raiz2(a, b, c));
    Equacoes e1 = new Equacoes();
    Equacoes e2 = new Equacoes();
    e1.valor = 3.0;
    JOptionPane.showMessageDialog(null, ""+e2.valor);
  }
}
```

```
public class Pessoa{
    private String nome;     private int idade;
    private static int numpessoas;
    public Pessoa(){
            nome = "";
            idade = 0;
            numpessoas++;
    }
    public static int getNumPessoas(){
            return numpessoas;
    }
    public String getNome(){
            return nome;
    }
    public int getIdade(){
            return idade;
    }
    public void setNome(String nome){
            this.nome = nome;
    }
    public void setIdade(int idade){
            this.idade = idade;
    }
}
```

```
public class UsaPessoa
{
   public static void main(String[] args)
   {
      Pessoa p = new Pessoa();
      Pessoa p1 = new Pessoa();
      Pessoa p2 = new Pessoa();
      Pessoa p3 = new Pessoa();

      System.out.println("Idade="+
   Pessoa.getNumPessoas());

   }
}
```

Pacotes

- Um pacote é um conjunto de classe relacionadas
- As classes de um pacote podem ser utilizadas se tal pacote for importado (comando *import*)
- Pacotes ajudam os programadores a administrar a complexidade dos componentes de um aplicativo e facilitam a reutilização de código

Pacotes

- Passos para criação de um pacote
  1. Definir a classe como *public* (se ela não for *public* só poderá ser utilizada dentro do pacote)
  2. Escolher um nome de pacote e adicionar uma instrução *package* ao arquivo de código-fonte para a definição da classe reutilizável
  3. Compilar a classe
  4. Importar a classe reutilizável para dentro de um programa e utilizá-la

Pacotes

Importante

- Só pode existir uma instrução *package* em um arquivo de código-fonte Java
- Essa instrução deve ser a primeira linha de código
- Fora do bloco da classe só podem existir dois comando: *package* e *import*

```
package AcessoDados;
  public class ConexaoDados {
  }
}

package Negocio;
public class ManipulaCliente {
}

package Interface.saida;
public class TelaImpressora {

}


package Interface;
import AcessoDados.*;
import Negocio.*;
import Negocio.ManipulaCliente;
  public class TelaPrincipal {
  public static void main(String args[]){
     ConexaoDados cd = new ConexaoDados();
     ManipulaCliente mc = new ManipulaCliente();
  }
}
```

Pacotes

- Para compilar a classe sonda, a partir do diretório onde ela se encontra, utiliza-se a seguinte linha de comando:

javac -d . Sonda.java

- A opção *-d* especifica onde criar (ou localizar) o diretório da instrução *package*.
- O ponto (.) após do *-d* representa o diretório corrente (Windows, Linux e Unix)

Pacotes

- Para a classe Sonda, se a compilarmos como mostrado, o compilador cria o diretório rone e dentro dele o diretório teste1
- O arquivo Sonda.class se encontrará dentro de teste1.

Sobrecarga de métodos

- Ocorre quando dois ou mais métodos possuem o mesmo nome
- Isso acontece entre métodos de classes diferentes mas em uma mesma hierarquia ou dentro de uma mesma classe
- Importante: quando uma classe possui métodos com o mesmo nome suas assinaturas devem ser diferentes, ou seja, a passagem de parâmetros deve ser diferente entre eles

```
public class Figura {
    private int x;
    private int y;
    public Figura(){
        setX(0);
        setY(0);
    }
    public void setX(int x){
        this.x = x;
    }
    public int getX(){
        return x;
    }
    public void setY(int y){
        this.y = y;
    }
    public int getY(){
        return y;
    }
    public int area(){
        return 0;
    }
}
```

```
public class Quadrado extends Figura{
    private int lado;
    public Quadrado(){
        super();
        setLado(0);
    }
    public Quadrado(int x, int y, int lado){
        setX(x);
        setY(y);
        setLado(lado);
    }
    public void setLado(int lado){
        this.lado = lado;
    }
    public int getLado(){
        return lado;
    }
    public int area(){
        return getLado() * getLado();
    }
}
```

Sobrecarga de métodos

```
import javax.swing.*;
public class UsaFigura extends JApplet{
    Quadrado q1, q2;
    public void init(){
        q1 = new Quadrado();
        q2 = new Quadrado(10,10,50);
    String saida = "O quadrado 1 tem área " + q1.area()+
                   "\nO quadrado 2 tem área " + q2.area();
    JOptionPane.showMessageDialog(null,saida);
}
}
```

Sobrecarga de métodos

- O método *area* sofreu sobrecarga de métodos pois foi criado em Figura e em Quadrado.
- Na classe UsaFigura, quando o método q1.area() é chamado, o método executado é o método criado em Quadrado. Em outras palavras, o método *area* da subclasse Quadrado sobrescreveu o método *area* da superclasse Figura.

Sobrecarga de métodos

- A classe quadrado possui dois métodos com o mesmos nome, no caso são os métodos construtores
- Contudo, em UsaFigura dois objetos são criados de forma diferente: q1 sem passagem de parâmetro para o construtor e q2 com passagem de parâmetros para o construtor
- O interpretador Java, através da assinatura, consegue distinguir qual dos dois métodos deverá ser executado

Importante

- Variáveis de superclasses podem receber objetos de subclasses
- Ex:

```
Figura f;
Quadrado q = new Quadrado();
f = q;
```

```
import javax.swing.JOptionPane;
public class TesteObject {
    public static void main(String[] args) {
        Object generico;
        String e = JOptionPane.showInputDialog("1 - PF \n2 -  PJ");
        if(e.equals("1")){
            PessoaFisica pf = new PessoaFisica();
            pf.setNome("Zé Mané");
            pf.setTelefone("12345678");
            pf.setCpf("11111111111");
            generico = pf;
        }
        else{
            PessoaJuridica pj = new PessoaJuridica();
            pj.setNome("Bar do Mané");
            pj.setTelefone("87654321");
            pj.setCnpj("2222222222");
            generico = pj;
        }
        if(generico instanceof PessoaFisica)
            JOptionPane.showMessageDialog(null, "É Pessoa Física!");
        else
            JOptionPane.showMessageDialog(null, "É Pessoa Jurídica!");
    }
}
```

Classe Object

- É a mãe de todas as classes em Java
- É chamada de tipo genérico
- Uma variável do tipo Object pode receber objetos de qualquer tipo
- Nos exemplos anteriores, as classes Figura e Cliente extends Object
- No exemplo a seguir, considera as classes PessoaFisica, PessoaJuridica e Cliente

Class Vector

- Coleção de objetos
- Implementa um vetor que não possui um tamanho definido
- O tipo é Object
- Possui vários métodos
- Exemplo

```
import java.util.*;
import javax.swing.*;
public class TestaVector {
	public static void main(String[] args) {
		Vector<Float> v = new Vector<Float>();
		String s;
		while(true){
		   s = JOptionPane.showInputDialog("Informe seu salário. Enter para sair.");
		   if(s.equals("")) break;
		   v.add(Float.parseFloat(s));
		}
		for(int i=0; i<v.size(); i++)
		   s=s+"\n" + i + " - " +v.elementAt(i);
		String e = JOptionPane.showInputDialog(null, s + "\nQual você deseja
		   remover:");
		v.remove(Integer.parseInt(e));
		s = "";
		for(int i=0; i<v.size(); i++)
		   s=s+"\n" + i + " - " + v.get(i);
		JOptionPane.showMessageDialog(null, s);
	}
}
```

Parametrização de Tipos

- Utiliza-se <tipo>
- Foi utilizado em:

  Vector<Float> v = new Vector<Float>();
- Fala que o vetor utilizará somente objetos do tipo Float
- Se não colocado, deve-se fazer a seguinte mudança:

  float aux = (Float) v.get(i);
- **Obs: os tipos predefinidos devem ser substituídos pelas usa classes correspondentes**

Classe e Métodos Abstratos

- Definidos pela palavra reservada abstract
- Classes abstratas → não podem gerar objetos
- Métodos abstratos → não possui declaração do seu corpo, somente do cabeçalho

Classe e Métodos Abstratos

- Obs1: um método abstrato só pode ser criado dentro de uma classe abstrata.
- Obs2: os filhos de uma classe abstratas são obrigados a implementar os métodos abstratos definidos na mãe.
- Para que serve classes e métodos abstrata?
  - Servem para obrigar as classes descendentes a implementar determinados métodos.

```
public abstract class Figura {
    private String cor;
    public String getCor() {
        return cor;
    }
    public void setCor(String cor) {
        this.cor = cor;
    }
    public abstract double area();
}
```

```
public class Quadrado extends Figura {
    private double lado;
    public double getLado() {
        return lado;
    }
    public void setLado(double lado) {
        this.lado = lado;
    }
    public double area() {
        return lado * lado;
    }
}
```

```
public class Circulo extends Figura{
    private double raio;
    public double area() {
        return Math.PI * raio * raio;
    }
    public double getRaio() {
        return raio;
    }
    public void setRaio(double raio) {
        this.raio = raio;
    }
}
```

```
public class ControleFiguras {
	public static void main(String args[]){
		int t = 3; Figura v[] = new Figura[t];
		for(int i=0; i<t; i++){
			String e = ent("1-Quadrado\n2-Círculo");
			if(e.equals("1")){
				Quadrado aux = new Quadrado();
				aux.setCor(ent("Cor"));
				aux.setLado(Integer.parseInt(ent("Lado")));
				v[i] = aux;
			}else{
				Circulo aux = new Circulo();
				aux.setCor(ent("Cor"));
				aux.setRaio(Integer.parseInt(ent("Raio")));
				v[i] = aux;
			}
		}
		String s = "Áreas das figuras";
		for(int i=0; i<t; i++) s = s + "\n" + v[i].area();
		JOptionPane.showMessageDialog(null, s);
	}
	public static String ent(String e){
		return JOptionPane.showInputDialog(e);
	}
}
```

Interfaces

- Equivalentes a classes abstratas
- Somente podem ter métodos abstratos e constantes (final)
- Define o comportamento das filhas
- Utiliza a palavra implements e não extends
- Aceita herança múltipla

```
public interface Projeto {
	public String nome = "Projeto";
	public abstract double ValorFinal();
}
public class ProjSoftware implements Projeto{
	private double numhoras;
	private double valorhora;
	public double getNumhoras() {
		return numhoras;
	}
	public void setNumhoras(double numhoras) {
		this.numhoras = numhoras;
	}
	public double getValorhora() {
		return valorhora;
	}
	public void setValorhora(double valorhora) {
		this.valorhora = valorhora;
	}
	public double ValorFinal() {
		return numhoras * valorhora;
	}
}
```

```
public class ProjHardware implements Projeto{
    private double valormaterial;
    private double valormaoobra;
    public double getValormaterial() {
        return valormaterial;
    }
    public void setValormaterial(double valormaterial) {
        this.valormaterial = valormaterial;
    }
    public double getValormaoobra() {
        return valormaoobra;
    }
    public void setValormaoobra(double valormaoobra) {
        this.valormaoobra = valormaoobra;
    }
    public double ValorFinal() {
        return valormaterial + valormaoobra;
    }
}
```

```
import javax.swing.*;
public class ControleProjeto {
	public static void main(String[] args) {
		int t = 3;
		Projeto v[] = new Projeto[t];
		for(int i=0; i<t;i++){
			String e = ent("1-Software\n2-Hardware");
			if(e.equals("1")){
				ProjSoftware aux = new ProjSoftware();
				aux.setNumhoras(Double.parseDouble(ent("Horas")));
				aux.setValorhora(Double.parseDouble(ent("Valor hora")));
				v[i] = aux;
			}
			else{
				ProjHardware aux = new ProjHardware();
				aux.setValormaoobra(Double.parseDouble(ent("Mão-de-obra")));
				aux.setValormaterial(Double.parseDouble(ent("Material")));
				v[i] = aux;
			}
		}
```

```
        String s = "Áreas das figuras";
        for(int i=0; i<t; i++){
            if(v[i] instanceof ProjHardware)
                s = s + "\n" +((ProjHardware)v[i]).getValormaterial()+ "+" +
                ((ProjHardware)v[i]).getValormaoobra()+"="+ v[i].ValorFinal();
            else {
                s = s + "\n" +((ProjSoftware)v[i]).getValorhora()+ "*" +
                  ((ProjSoftware)v[i]).getNumhoras()+"="+ v[i].ValorFinal();
            }
    }
        JOptionPane.showMessageDialog(null, s);
    }
    public static String ent(String e){
        return JOptionPane.showInputDialog(e);
    }
}
```

instanceof

- O operador instanceof verifica que um objeto é do tipo de uma classe.
- No exemplo anterior:

```
if(v[i] instanceof ProjHardware)

else {

}
```

Encapsulamento de Métodos e Atributos

- Vem de encapsular – esconder
- Define quais atributos e métodos de uma classe estarão disponíveis para acesso.

	Local	**Subclasse**	**Objeto**
public	Sim	Sim	Sim
protected	Sim	Sim	Sim/Não
private	Sim	Não	Não

- Obs:
  - protected em objeto do mesmo pacote → sim
  - protected em objeto de pacotes diferentes → não

Encapsulamento de Classes

- Classe public: são acessíveis a partir de qualquer objeto, independentemente do package. Uma classe pública deve ser a única classe desse tipo no arquivo em que está declarada e o nome do arquivo deve ser igual ao da classe.
- Classe friendly: Apenas os objetos integrantes do mesmo package podem utilizar uma classe friendly. se nenhum modificador de classe for especificado, então a classe será considerada friendly
- Ainda existem os modificadores final e abstract, descritos aos se tratar polimorfismo

```
package primeiro;
public class Parent {
    public int a;
    protected int b;
    private int c;
    public Parent(){
        a=1;
        b=2;
        c=3;
    }
    public void parentDados(){
       System.out.println("\nparentDados:  A=" + a + " B=" + b
       + " C= " + c);
    }
}
```

```
package primeiro;

public class Son extends Parent{
    public void sonDados(){
        System.out.println("\nsonDados: A=" + a + " B=" + b );
    }
}
```

```
package primeiro;
public class MesmoPacote {
    public static void main(String[] args) {
        Son obj = new Son();
        System.out.println("obj.a= "+ obj.a + " obj.b="+obj.b);
        obj.sonDados();
        obj.parentDados();
    }
}
```

```
package segundo;
import primeiro.*;
public class OutroPacote {
    public static void main(String[] args) {
        Son obj = new Son();
        System.out.println("obj.a= "+ obj.a);
        obj.sonDados();
        obj.parentDados();
    }
}
```