

Programação Orientada a Objetos

Conceitos importantes

Rone Ilídio
Thiago Oliveira



Conceitos

■ Principais

- ☐ Classe
- ☐ Herança
- ☐ Encapsulamento
- ☐ Polimorfismo

■ Adicionais

- ☐ Tipos de classes e membros
- ☐ Sobrecarga de métodos
- ☐ Arquivos e pacotes



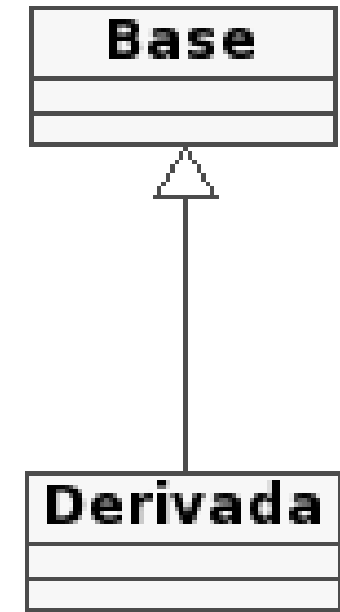
Hierarquia de classes

■ Herança

- ☐ Forma de utilização de *software*;
- ☐ Novas classes são criadas a partir de classes existentes;
- ☐ Absorvendo seus atributos e comportamentos;
- ☐ Adicionando novos recursos que as novas classes exigem.

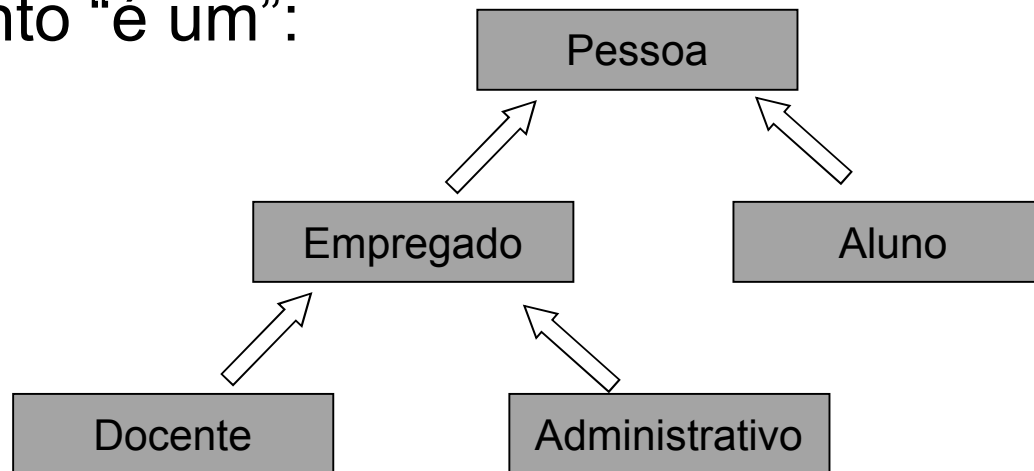
Herança

“Forma de utilização de código onde uma classe é criada absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas.”



Herança

- A classe já existente é chamada superclasse ou classe pai.
- A classe derivada é chamada subclasse ou classe filha.
- Relacionamento “é um”:





Herança

- Inicialmente, considere a classe Pessoa sem construtor ou com construtor vazio.

```
public class Pessoa{  
    private String nome;  
    private int idade;  
    public Pessoa() {    }  
    . . .  
}
```



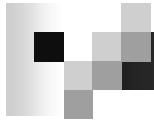
Herança

```
public class Aluno extends Pessoa {  
    private String matricula;  
  
    public String getMatricula() {  
        return matricula;  
    }  
  
    public void setMatricula(String matricula)  
    {  
        this.matricula = matricula;  
    }  
}
```



Herança

```
public class Principal {  
    public static void main(String[] args) {  
        Aluno aluno = new Aluno();  
  
        aluno.setMatricula("11111");  
        aluno.setIdade(18);  
        aluno.setNome("Maria");  
  
        System.out.println("Nome: "+ aluno.getNome());  
        System.out.println("Idade: "+ aluno.getIdade());  
        System.out.println("Matricula : "+ aluno.getMatricula());  
    }  
}
```

Herança e Construtores

- Se a superclasse possui a definição de um método construtor, suas filhas devem:
 - ☐ Também possuir a definição de um construtor;
 - ☐ Chamar o construtor da classe pai na primeira linha de seu construtor.

- Utilização da palavra reservada “super”.



Herança e Construtores

```
public class Pessoa{  
    private String nome;  
    private int idade;  
    public Pessoa(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

Definição do
método construtor

```
    public String getNome() {...}  
    public int getIdade() {...}  
    public void setNome(String nome) {...}  
    public void setIdade(int idade) {...}  
}
```



Herança e Construtores

```
public class Aluno extends Pessoa{  
    private String matricula;  
    public Aluno(String nome, int idade, String matricula){  
        super(nome, idade);  
        this.matricula = matricula;  
    }  
    public String getMatricula() {  
        return matricula;  
    }  
    public void setMatricula(String matricula) {  
        this.matricula = matricula;  
    }  
}
```

← Chama construtor da classe pai



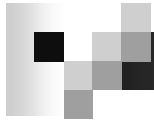
Herança

- **Cada subclasse pode:**

- ☐ Definir novos atributos e/ou operações;
- ☐ Redefinir operações da superclasse;
- ☐ Participar de relacionamentos específicos.

- **Evidências da necessidade de heranças:**

- ☐ Similaridades entre as classes;
- ☐ A subclasse é um tipo da superclasse;
- ☐ Evitar herança de implementação;
- ☐ Herança deve ser total pelas subclasses.




Encapsulamento

- Encapsular consiste em incluir, proteger em uma cápsula, classe.
 - Proteção de dados visa garantir o acesso apenas sobre operações e atributos disponibilizados pela interface da classe;
 - Todos os atributos e operações de uma classe podem ser acessados pelas operações da mesma classe;
 - O acesso aos atributos é, em geral, privado ou protegido.

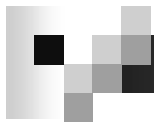


Encapsulamento

- **Acesso público:**
 - ☐ Visível por todos os pacotes.
- **Acesso protegido:**
 - ☐ Visível somente por classes e subclasses da mesma hierarquia.
- **Acesso protegido ao pacote (padrão):**
 - ☐ Visível somente por classes e subclasses do mesmo pacote.
- **Acesso privado:**
 - ☐ Visível somente a própria classe.



```
public abstract class Cliente{
    private String nome;
    private String telefone;
    public Cliente(){
        nome="";
        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 abstract String retornaDados();
}
```



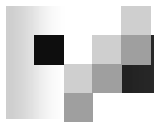
```
public final class PessoaJuridica extends Cliente{
    private String cnpj;

    public PessoaJuridica(){
        super();
        cnpj = "";
    }

    public void setCnpj(String cnpj){
        this.cnpj = cnpj;
    }

    public String getCnpj(){
        return cnpj;
    }

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

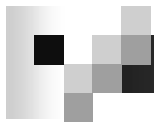
```
public final class PessoaFisica extends Cliente{
    private String cpf;

    public PessoaFisica(){
        super();
        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(){
        Cliente cl;                                // não cria objeto, pois Cliente é classe abstrata!

        PessoaFisica pf = new PessoaFisica();
        pf.setNome("Maria");
        pf.setTelefone("000000000");
        pf.setCpf("111.111.111-11");

        PessoaJuridica pj = new PessoaJuridica();
        pj.setNome("Buteco do Zé");
        pj.setTelefone("999999999");
        pj.setCnpj("0000.000.000-00");


        cl= pf;
        JOptionPane.showMessageDialog(null,"Dados do cliente:" + cl.retornaDados());

        cl= pj;
        JOptionPane.showMessageDialog(null,"Dados do cliente:" + cl.retornaDados());
    }
}
```



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:
 - `nome_classe.nome_metodo();`



```
public class Pessoa{
    private String nome;
    private int idade;
    private static int numPessoas = 0;
    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 p1 = new Pessoa();
        Pessoa p2 = new Pessoa();
        Pessoa p3 = new Pessoa();
        Pessoa p4 = new Pessoa();

        JOptionPane.showMessageDialog(null, "Objetos criados: "
                                         + Pessoa.getNumPessoas());
    }
}
```

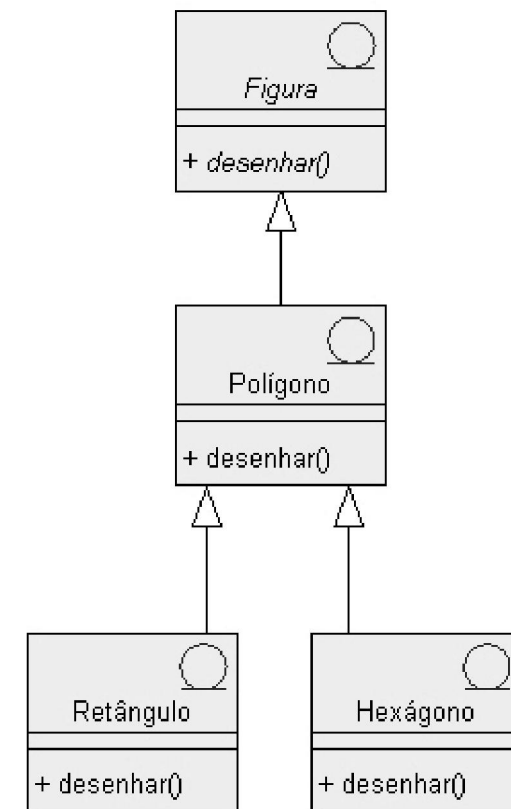


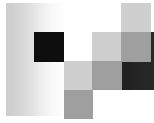
Polimorfismo

- Um mesmo objeto pode ser de vários tipos:
 - Pessoa pode ser um Estudante ou um Professor;
- Instâncias de várias classes são tratadas de forma única em um sistema.
- Não é viável exigir que todos os outros objetos saibam todos os possíveis tipos de um determinado objeto.
- Todos os outros objetos devem reconhecer o objeto através de um único tipo.

Polimorfismo

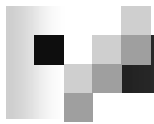
- Cada tipo reimplementa alguma parte da interface em comum.
- Outros objetos do sistema acessam a interface em comum de forma única.
- Polimorfismo é uma técnica para aumentar o grau de reuso.





Sobrecarga de métodos

- Ocorre quando dois ou mais métodos possuem o mesmo nome.
 - Métodos de classes diferentes em uma mesma hierarquia ou dentro de uma mesma classe.
- Para uma classe ter métodos com o mesmo nome, suas assinaturas devem ser diferentes.
 - Ou seja, a passagem de parâmetros deve ser diferente entre eles.



```
public abstract 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();  
    }  
}
```



```
import javax.swing.*;
```

```
public class UsaFigura extends JApplet{
```

```
    public void init()
```

```
{
```

```
    Figura q1, q2;
```

```
    q1 = new Quadrado();
```

```
    q2 = new Quadrado(10,10,50);
```

```
    String saida = "Quadrado 1 tem área: " + q1.area()
```

```
        + "\nE 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 mesmo nome, no caso os métodos construtores.
- Assim, em UsaFigura dois objetos são criados de forma diferente:
 - q1 sem passagem de parâmetro para o construtor;
 - q2 com passagem de parâmetros para o construtor.
- O interpretador Java, através da assinatura, consegue distinguir qual dos dois métodos deverá ser executado.



Duas classes em um mesmo arquivo

- Várias classes podem ser criadas dentro do mesmo arquivo fonte, mas só uma pode ser declarada como *public*.
- As demais só poderão ser acessadas dentro do pacote onde se encontram.
- O arquivo deve ter o nome da classe *public*.
- A compilação deste arquivo gera dois arquivos *.class*.

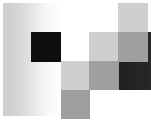


```
import java.awt.Graphics;
public class PosicaoSalas extends JApplet{
    Sala sala;

    public void init(){
        String entrada;
        sala = new Sala();

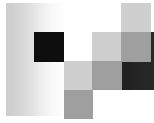
        //Inserindo dados de uma sala
        entrada = JOptionPane.showInputDialog(null, "Informe o nome da sala");
        sala.setNome(entrada);
        entrada = JOptionPane.showInputDialog(null, "Qual o número da sala?");
        sala.setNumero(Integer.parseInt(entrada));
        entrada = JOptionPane.showInputDialog(null, "Em qual bloco?");
        sala.setBloco(Integer.parseInt(entrada));
    }

    public void paint (Graphics g){
        g.drawOval(sala.getNumero(), sala.getBloco()+10, 6, 6);
        g.drawString(sala.getNome(), sala.getNumero(), sala.getBloco());
    }
}
//continua no mesmo arquivo
```



//continuação no mesmo arquivo

```
class Sala{
    private String nome;
    private int numero;
    private int bloco;
    public Sala(){
        nome = "";
        numero = 0;
        bloco = 0;
    }
    public String getNome(){
        return nome;
    }
    public void setNome(String n){
        nome = n;
    }
    public int getNumero(){
        return numero;
    }
    public void setNumero(int valor){
        numero = valor;
    }
    public int getBloco(){
        return bloco;
    }
    public void setBloco(int valor){
        bloco = valor;
    }
}
```



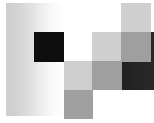
Pacotes

- Pacote é um conjunto de classes relacionadas.
- Os códigos de classes de mesmo pacote encontram-se na mesma pasta.
- Organizam o código e facilitam a reutilização de código.
- Palavras reservadas:
 - *package* : define o nome de um pacote
 - *import* : informa a utilização de um pacote



Pacotes

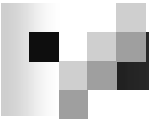
- Passos para criação de um pacote:
 1. Definir a classe como *public* (se ela não for *public* só poderá ser utilizada dentro do pacote);
 2. Escolher um nome de pacote e adicionar uma instrução *package* ao arquivo de código-fonte para a definição da classe reutilizável;
 3. Compilar a classe;
 4.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



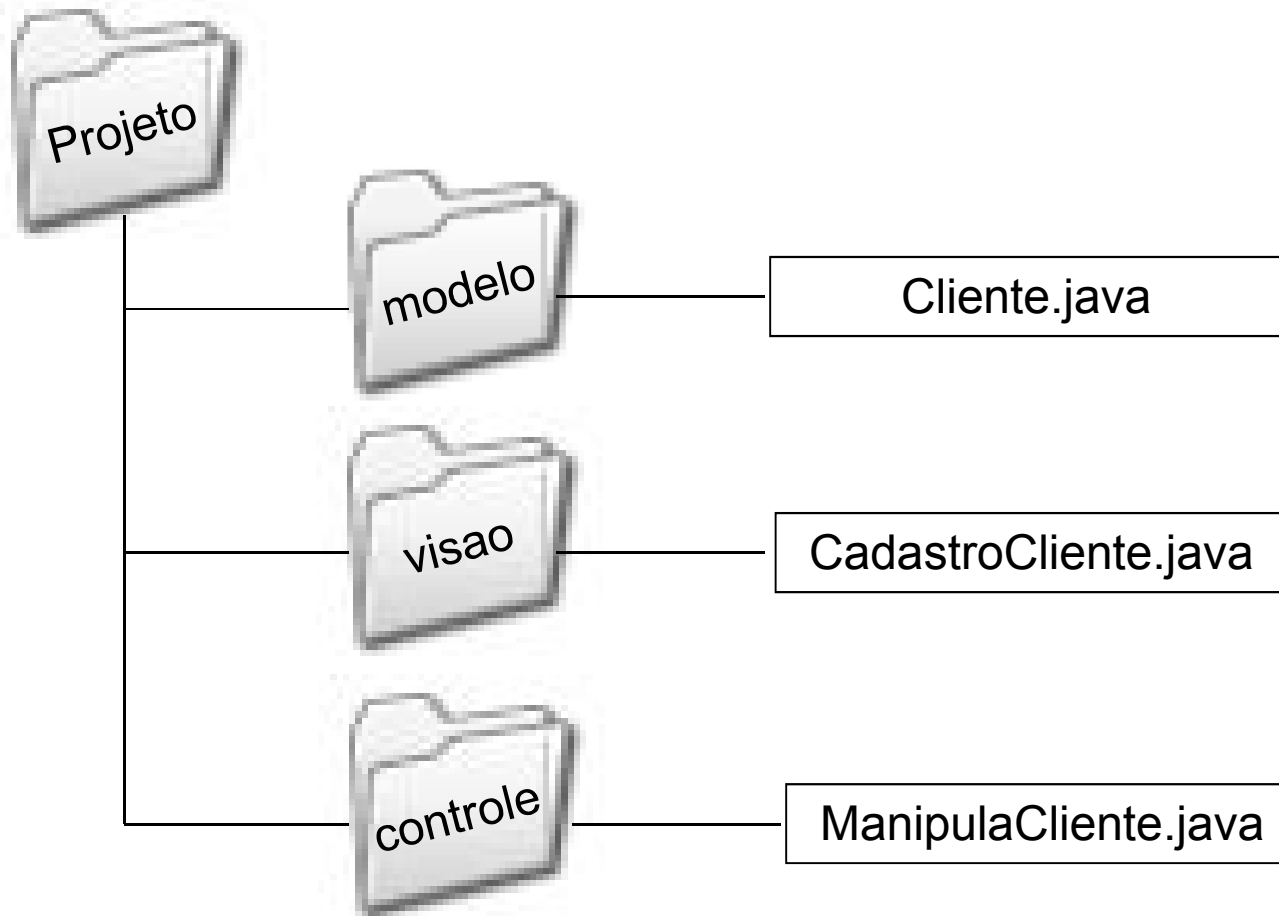
```
package poo.salas
public class Sala{
    private String nome;
    private int numero;
    private int bloco;
    public Sala(){
        nome = "";
        numero = 0;
        bloco = 0;
    }
    public String getNome(){
        return nome;
    }
    public void setNome(String n){
        nome = n;
    }
    public int getNumero(){
        return numero;
    }
    public void setNumero(int valor){
        numero = valor;
    }
    public int getBloco(){
        return bloco;
    }
    public void setBloco(int valor){
        bloco = valor;
    }
}
```

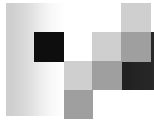


Pacotes

- Para compilar a classe Sala, a partir do diretório onde ela se encontra, utiliza-se a seguinte linha de comando:
 - ☐ `javac -d . Sala.java`
 - ☐ A opção `-d` especifica onde criar (ou localizar) o diretório da instrução *package*
 - ☐ O ponto (.) após `-d` representa o diretório corrente.
 - ☐ Arquivo `Sala.class` será criado dentro de `salas`.
 - ☐ Se estiver em outro diretório, o compilador cria o diretório `poo` e dentro dele o diretório `salas`.

Pacotes





Pacotes

```
package modelo;  
public class Cliente {  
...  
}
```

```
package visao;  
public class CadastroCliente{  
...  
}
```

```
package controle;  
public class ManipulaCliente{  
...  
}
```



Pacotes

- Criação de um objeto Cliente na classe ManipulaCliente.

```
package controle;  
import modelo.Cliente;  
public class ManipulaCliente{  
    ...  
    Cliente cliente = new Cliente();  
    ..  
}
```



Pacotes

- No import pode-se informar a importação de todas as classe de um pacote.

```
package controle;  
import modelo.*;  
public class ManipulaCliente{  
    ...  
    Cliente cliente = new Cliente();  
    ..  
}
```




Pacotes

- O import pode ser substituído se o nome da classe tiver o pacote onde ela se encontra.

```
package controle;  
public class ManipulaCliente{  
    ...  
    modelo.Cliente c = new modelo.Cliente();  
    ..  
}
```



Pacotes

- Divide o programa em módulos.
- Criação de bibliotecas de classe.
- Auxilia a reutilização de código.
- O Java já possui uma vasta biblioteca de classe, exemplos de pacotes.
 - javax.swing: interface gráfica
 - javax.sql: acesso a banco de dados
 - java.net: rede