

ORIENTAÇÃO A OBJETOS

JAVA
RESUMÃO

BY RAFAELA P. C. MOREIRA



Sumário

Programação Orientada a Objetos (POO)



É um paradigma.



Representa coisas do mundo real
através de objetos.

Vantagens da POO



Reutilização de código.



Maior nível de abstração.



Facilidade na manutenção do código.

Elementos Básicos da OO

CLASSES

OBJETOS

ATRIBUTOS

MÉTODOS

Elementos Básicos da OO

CLASSES

Coleção de objetos com características similares.

Tipo abstrato de dados (molde).



Uma classe é formada por dados (atributos) e códigos (métodos) que operam sob esses dados.

Elementos Básicos da OO

CLASSES



NOME

Atributos

Raça

Cor

Porte

Métodos

Comer

Latir

Elementos Básicos da OO

CLASSES

Classes em JAVA são criados com a palavra reservada **class**.

```
modificador class Classe1{
    //atributos
    tipo atr1;
    tipo atr2;
    // métodos
    modificador tipo metodo1(listaParam(opcional)){
        // corpo do metodo
    }
    modificador tipo metodo2(listaParam(opcional)){
        // corpo do metodo
    }
}
```

Elementos Básicos da OO

CLASSES

Para acessar os membros de uma classe, deve-se utilizar o operador ponto (.) .

Esse operador é usado para ligar a referência a um objeto para seus membros (atributos ou métodos).

objeto . atributo

Elementos Básicos da OO

CLASSES

Variáveis de Classes

Variáveis de classe são chamadas de variáveis de instância, porque cada instância (objeto) possui sua própria cópia dessas variáveis.

Assim, os dados de um objeto são únicos para aquele objeto e não misturam com dados de outros objetos.

Elementos Básicos da OO

CLASSES

```
public class Cachorro{  
    //atributos da classe  
    String nome;  
    String raca;  
    String cor;  
    String porte;  
    // métodos da classe  
    public void latir(){  
        if(porte.equalsIgnoreCase("pequeno"))  
            System.out.println("Au-au!");  
        else if(porte.equalsIgnoreCase("medio"))  
            System.out.println("Ruff-ruff!");  
        else  
            System.out.println("Woof-woof!");  
    }  
}
```

Elementos Básicos da OO

OBJETOS

Entidade física ou conceitual do mundo real (carros, casas, . . .)
Instância de uma classe.

Possui características próprias e executa determinadas ações.



Belinha
Yorkshire
Preto/Dourado
Pequeno
Latir



Alone
Buldog
Tigrado
Médio
Latir



Luke
Rottweiler
Preto
Grande
Latir

Elementos Básicos da OO

• OBJETOS •

Objetos em **JAVA** são criados com o operador **new**, que aloca a memória dinamicamente para guardar informações desse objeto.

Construindo uma atribuição com operador **new**:

```
Cachorro c1 = new Cachorro();
```

- **c1**: irá receber a referência para o objeto.
- **Cachorro**: o nome da classe especifica o tipo de objeto que vai ser construído.
- **Parênteses ()**: chamada ao construtor da classe.

Elementos Básicos da OO

ATRIBUTOS

Característica dos objetos.

Variável do objeto.

Atributos são declarados dentro da classe.



Belinha
Yorkshire
Preto/Dourado
Pequeno
Latir



Alone
Buldog
Tigrado
Médio
Latir



Luke
Rottweiler
Preto
Grande
Latir

Elementos Básicos da OO

MÉTODOS

Sub rotinas que são executadas por um objeto.

Determinam o comportamento dos objetos (ações).

O método deve ser declarado dentro de uma classe.



Belinha
Yorkshire
Preto/Dourado
Pequeno
Latir



Alone
Buldog
Tigrado
Médio
Latir



Luke
Rottweiler
Preto
Grande
Latir

Elementos Básicos da OO

MÉTODOS

Um método em **JAVA** tem a seguinte forma geral:

```
modificador tipo nomeMetodo(listaParametros(opcional)) {  
    // corpo do metodo  
}
```

- **modificador de acesso:** é a palavra-chave que define um atributo, método ou classe como público, privado ou protegido
- **tipo:** especifica o tipo que o método deve retornar.
 - ao final do método deve usar a palavra **return** para retornar um valor (quando necessário).
 - tipo **void**: o método não possui retorno, logo não necessita do return.
- **nomeMetodo:** especifica o nome pelo qual o método será referenciado no objeto.
- **lista-parametros:** sequência de pares de tipos e identificadores separados por vírgula.

Elementos Básicos da OO

MÉTODOS

Dois termos são frequentemente utilizados ao se referenciar métodos:

Parâmetros: variável definida pelo método que recebe o valor quando o método é chamado.

```
public double soma(double x, double y) {  
    return x + y;  
}
```

Argumentos:

valor passado para o método quando o mesmo é invocado.

```
double s = soma(10, 5);
```

Elementos Básicos da OO

MÉTODOS

Chamada de Métodos

Como os métodos também são membros de uma classe, deve-se utilizar o operador ponto (.) para acessá-los.

```
objeto.método(listaParametros);
```

O método acessa os atributos diretamente.

- O método latir acessa os atributos nome, raça, cor, porte do objeto no qual foi invocado.

Elementos Básicos da OO

MÉTODOS

O exemplo a seguir mostra a chamada do método latir para dois cachorros diferentes. Cada chamada mostra o latido de cada cachorro.

```
public class Principal{  
  
    public static void main(String [] args){  
        // criando objeto c1 da classe Cachorro  
        Cachorro c1 = new Cachorro();  
  
        // atribuindo valores para os atributos de c1  
        c1.nome = "Belinha";  
        c1.raca = "Yorkshire";  
        c1.porte = "Pequeno";  
  
        // chamada do método latir para c1  
        c1.latir();  
    }  
}
```

Elementos Básicos da OO

MÉTODOS

```
// criando objeto c1 da classe Cachorro
Cachorro c2 = new Cachorro();

// atribuindo valores para os atributos de c2
c2.nome = "Alone";
c2.raca = "Rottweiler";
c2.porte = "Grande";

// chamada do método latir para c2
c2.latir();

}
```

CONSTRUTORES

Java permite que objetos inicializem seus atributos quando são criados, através de um **construtor**.

O construtor define o que acontece durante a criação do objeto.

Construtor é sintaticamente similar a um método, mas **deve ter o mesmo nome de sua classe** e não deve ter um tipo de retorno.

Uma vez definido, o construtor é chamado automaticamente quando o objeto é criado.

CONSTRUTORES

```
public class Caixa{
    // declarando os atributos
    public double largura, altura, profundidade;
    //construtor vazio
    public Caixa() {
    }
    //construtor com parâmetro
    public Caixa(double l, double a, double p) {
        largura = l;
        altura = a;
        profundidade = p;
    }
    // metodo para calcular volume
    public double volume() {
        return largura * altura * profundidade;
    }
}
```

CONSTRUTORES

Durante a criação do objeto, é chamado o construtor da classe (por isso dos parênteses)

```
Caixa c1 = new Caixa(10,2,8);  
Caixa c2 = new Caixa(2,4,6);
```

Caso a classe não possua nenhum construtor, Java irá criar um construtor vazio, conhecido como **construtor padrão**.

```
Caixa c1 = new Caixa();
```

Palavra reservada

THIS

Em alguns casos, o método invocado de um objeto pode precisar de uma referência para o próprio objeto.

A palavra-reservada **this** pode ser usada para referenciar o próprio objeto.

Duas variáveis locais não podem ter o mesmo nome, mas essa restrição não é aplicada para variáveis de instância.

Palavra reservada

THIS

Uma variável local pode ter o mesmo nome de uma variável de instância.

Quando isso acontece, a variável local esconde a variável de instância. Porém, pode-se usar a palavra-reservada **this** para referenciar a variável de instância.

```
public Caixa(double largura, double altura,  
double profundidade) {  
    this.largura = largura;  
    this.altura = altura;  
    this.profundidade = profundidade;  
}
```

Princípios da Orientação a Objetos

ABSTRAÇÃO



Abstração



Complexidade

Humanos gerenciam a complexidade
através de abstração.

Princípios da Orientação a Objetos

ABSTRAÇÃO

Uma forma de gerenciar a complexidade é através de classificações hierárquicas.

Pode-se criar camadas semânticas de sistemas complexos subdividindo-o em unidades menores.



Som
Direção
Freios

Princípios da Orientação a Objetos

ABSTRAÇÃO

Na programação de computadores os dados podem ser abstraídos para componentes de objetos.

Abstração é a representação de um objeto real.

Pontos importantes:

- **Identidade:** necessita de um nome para identificar o objeto.
- **Propriedades:** características do objeto.
- **Métodos:** ações que o objeto executará.

Princípios da Orientação a Objetos

ABSTRAÇÃO

Identidade:

Cachorro

Propriedades:

nome, raça, porte

Métodos:

latir



Princípios da Orientação a Objetos

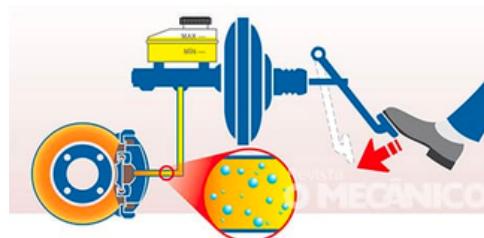
ENCAPSULAMENTO

Técnica que oculta os detalhes internos do funcionamento dos métodos de uma classe.

Uma classe pode ser visualizada de duas formas:

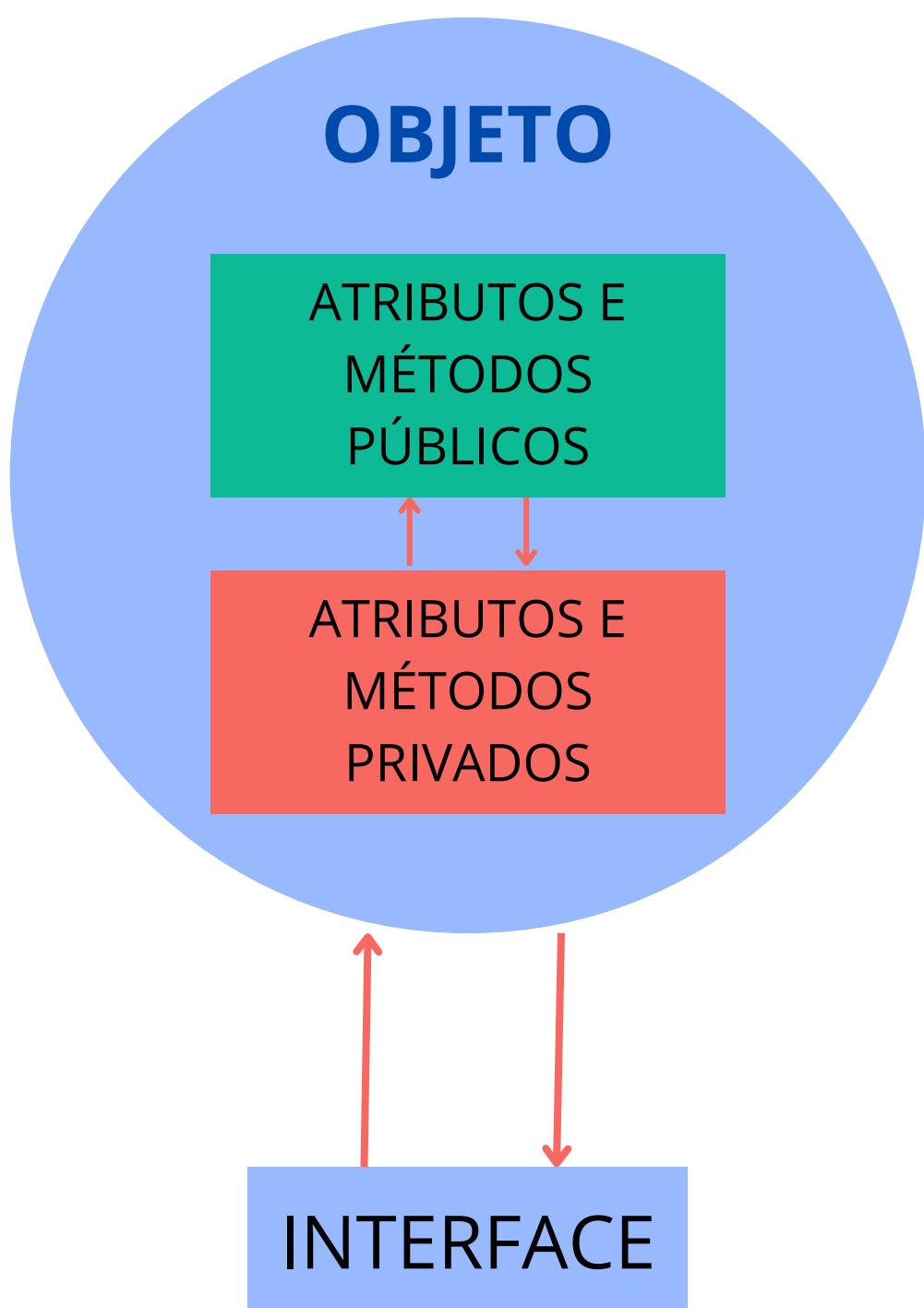
Interface:
pode ser vista e usada por outros objetos.

Implementação:
é escondida do objeto.



Princípios da Orientação a Objetos

ENCAPSULAMENTO



Princípios da Orientação a Objetos

ENCAPSULAMENTO

Modificadores de acesso

Definem como um membro pode ser acessado.

- **público(public)**: os métodos ou atributos são acessíveis por qualquer classe.
- **privado(private)**: os métodos ou atributos são acessíveis apenas pela própria classe.
- **protegido(protected)**: os métodos ou atributos são acessíveis pela própria classe, classes do mesmo pacote ou classes da mesma hierarquia.

```
// modificador público
public int a;
// modificador privado
private double b;
// modificador protegido
protected String c;
```

Princípios da Orientação a Objetos

ENCAPSULAMENTO

Métodos GETTER's e SETTER's

Os métodos GET e SET são técnicas para gerenciar o acesso aos atributos.

O método **GET** retorna o valor do atributo.

O método **SET** altera o valor do atributo de forma segura.

```
public double getPreco() {  
    return preco;  
}  
public void setPreco(double preco) {  
    this.preco = preco;  
}
```

Princípios da Orientação a Objetos

ENCAPSULAMENTO

```
public class Produto{  
    // declarando os atributos  
    private int codigo;  
    private String nome;  
    private double preco;  
    //construtor com parâmetro  
    public Produto(int codigo, String nome, double  
preco) {  
        this.codigo = codigo;  
        this.nome = nome;  
        this.preco = preco;  
    }  
    ...  
    public double getPreco() {  
        return preco;  
    }  
    public void setPreco(double preco) {  
        this.preco = preco;  
    }  
    public String toString() {  
        return "Codigo: " + this.codigo + ", Nome: "  
        + this.nome + ", Preco: " + this.preco;  
    }  
}
```

Palavra reservada

STATIC

Em geral, um membro de uma classe é acessado por um objeto. Porém existem situações em que seja interessante criar um membro independente.

Quando isso acontece, é possível criar um membro que pode ser acessado sem a necessidade de instanciar um objeto, usando o modificador **STATIC**.

Métodos e atributos podem ser marcados com o modificador **static**.

Palavra reservada

STATIC

Variáveis de instância (atributos) declarados com o modificador static são essencialmente variáveis globais.

Restrições de métodos estáticos:

- Só podem chamar outros métodos estáticos.
- Só podem acessar dados estáticos.

Uma classe pode acessar membros estáticos declarados em outras classes.

Palavra reservada

STATIC

Para acessá-los, deve-se chamar o nome da classe.

```
public class Estatica {  
    public static int a = 15;  
    public static int b = 39;  
  
    static void imprimir(){  
        System.out.println("a = " + a);  
    }  
}
```

```
public class EstaticaPrincipal{  
    public static void main(String [] args){  
        Estatica.imprimir();  
        System.out.println("b = " + Estatica.b);  
    }  
}
```

Princípios da Orientação a Objetos

HERANÇA

Herança é uma técnica que permite criar novas classes (subclasses) com as características adicionais de uma classe existente (superclasse).

Herança é uma forma de reutilização de software.

A subclasse (classe filha) “herda” as funcionalidades da superclasse (classe mãe) e adiciona novos aspectos.

Princípios da Orientação a Objetos

HERANÇA

Uma subclasse é mais específica que sua superclasse e representa um grupo especializado de objetos.

A subclasse exibe os comportamentos da superclasse e pode modificá-los de modo que eles.

Superclasse → generalização
Subclasse → especialização.

Para herdar uma classe, basta incorporar a definição de uma classe em outra através da palavra-reservada **extends**.

Princípios da Orientação a Objetos

HERANÇA

Embora uma subclasse herde todos os membros da superclasse, ela não poderá acessar os membros declarados como **privados (private)**.

Um membro declarado como privado continuará privado a sua classe, ou seja, não é acessível por nenhuma outra classe, inclusive subclasses.

Java possui uma solução para esse problema através da palavra-reservada **super**.

Princípios da Orientação a Objetos

HERANÇA

A palavra-reservada **super** chama o construtor da superclasse e acessa um membro da superclasse.

```
super(listaParametros);
```

Para chamar um membro da superclasse deve-se utilizar a palavra-reservada **super**.

```
super.membro;
```

Princípios da Orientação a Objetos

HERANÇA

```
public class Pessoa {  
  
    private String nome;  
    private String cpf;  
    protected String email;  
  
    public Pessoa(String nome, String cpf) {  
        this.nome = nome;  
        this.cpf = cpf;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

Princípios da Orientação a Objetos

HERANÇA

```
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public String getCpf() {  
    return cpf;  
}  
  
public void setCpf(String cpf) {  
    this.cpf = cpf;  
}  
  
public String getEmail() {  
    return email;  
}  
  
public void setEmail(String email) {  
    this.email = email;  
}
```

Princípios da Orientação a Objetos

HERANÇA

```
@Override  
public String toString() {  
    return " Nome: " + nome + ", CPF: " +  
        cpf + ", Email: " + email;  
}  
}
```

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

Princípios da Orientação a Objetos

HERANÇA

```
public void setMatricula(String matricula) {  
    this.matricula = matricula;  
}  
  
@Override  
public String toString() {  
    return super.toString() + ", Matricula:  
" + matricula;  
}  
}
```

```
public class Professor extends Pessoa {  
  
    private double salario;  
    private String titulo;  
  
    public Professor(String nome, String cpf) {  
        super(nome, cpf);  
    }  
}
```

Princípios da Orientação a Objetos

HERANÇA

```
public double getSalario() {  
    return salario;  
}  
  
public void setSalario(double salario) {  
    this.salario = salario;  
}  
  
public String getTitulo() {  
    return titulo;  
}  
  
public void setTitulo(String titulo) {  
    this.titulo = titulo;  
}  
@Override  
public String toString() {  
    return super.toString() + ", Salario: "  
+ salario + ", Titulo: " + titulo;  
}  
}
```

Princípios da Orientação a Objetos

HERANÇA

```
public class PessoaHeranca {  
  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa("Maria",  
"920.287.886-23");  
        Aluno a1 = new Aluno("José",  
"565.241.896-63", "3435334");  
        Professor pf1 = new  
Professor("Rafaela","926.265.106-61");  
        p1.setEmail("maria@gmail.com");  
        pf1.setEmail("rafapcmor@gmail.com");  
        pf1.setSalario(10000.00);  
        pf1.setTitulo("Doutora em MMC");  
        a1.setEmail("jose.tech@gmail.com");  
        System.out.println("---- Pessoa ----  
\n" + p1.toString());  
        System.out.println("---- Aluno ----  
\n" + a1.toString());  
        System.out.println("---- Professor ---  
- \n" + pf1.toString());  
    }  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

Poli = muitas + **Morfos** = formas

Polimorfismo é uma técnica que permite a criação de código com a capacidade de operar sobre valores distintos.

A mesma mensagem enviada para vários objetos tem muitas formas de resultado.

Princípios da Orientação a Objetos

POLIMORFISMO

Tipos de Poliformismo

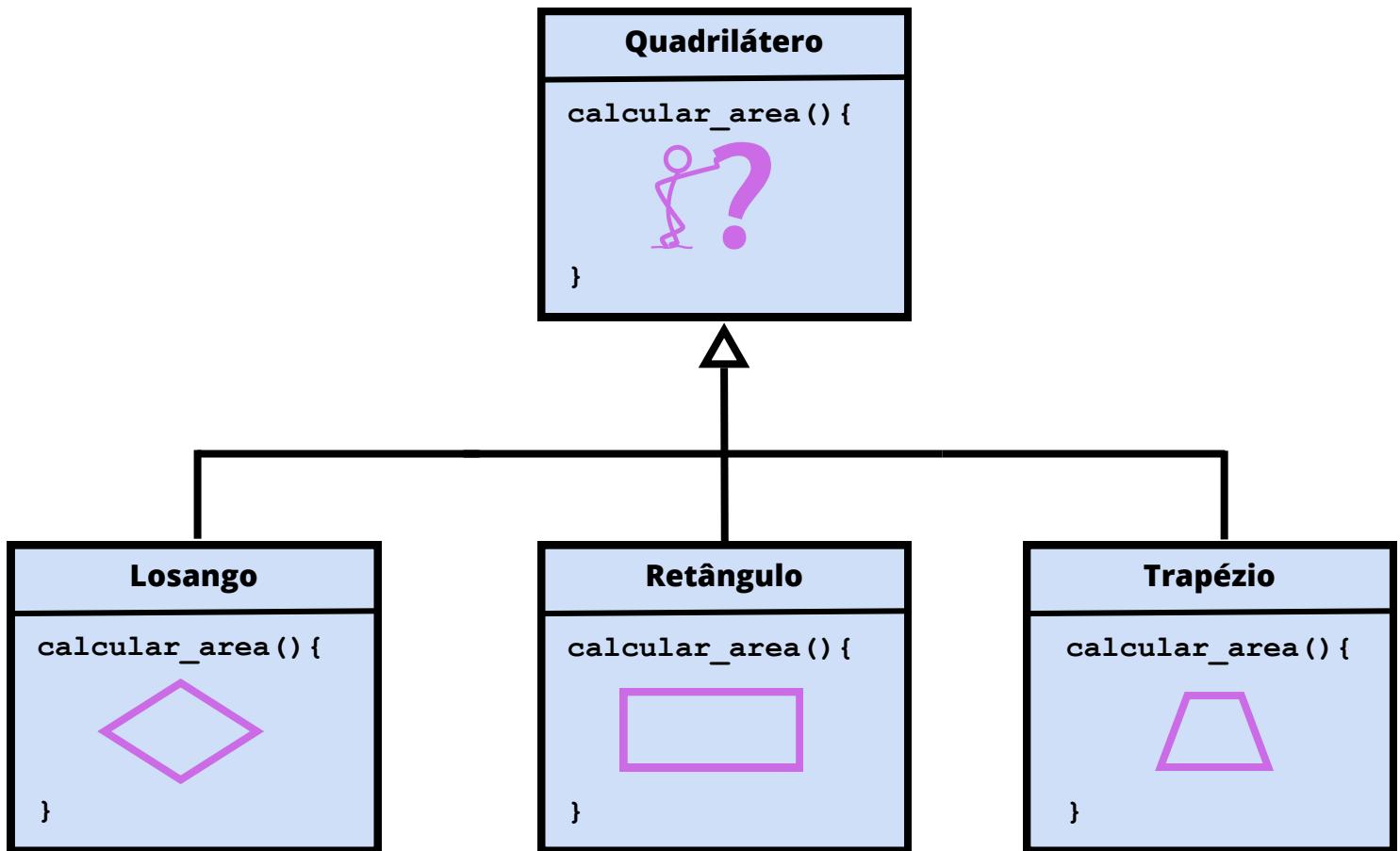
Sobrecarga: permite criar, dentro da mesma classe, métodos com o mesmo nome, mas com parâmetros diferentes.

```
soma(int a, int b);  
soma(double a, double b);  
soma(String a, String b);
```

Sobrescrita: permite reescrever na subclasse os métodos da superclasse, alterando o seu comportamento.

Princípios da Orientação a Objetos

POLIMORFISMO



Princípios da Orientação a Objetos

POLIMORFISMO

Assinatura de métodos: é a identificação do método (nome + quantidade de parâmetros + tipo de parâmetros). O retorno do método não faz parte da assinatura.

Métodos com a mesma assinatura são **sobrescritos**. Caso contrário, são **sobre carregados**.

Princípios da Orientação a Objetos

POLIMORFISMO

```
public abstract class Quadrilatero {  
  
    private double dim1, dim2;  
  
    public Quadrilatero(double dim1, double  
dim2) {  
        this.dim1 = dim1;  
        this.dim2 = dim2;  
    }  
  
    public double getDim1() {  
        return dim1;  
    }  
  
    public void setDim1(double dim1) {  
        this.dim1 = dim1;  
    }  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public double getDim2() {  
    return dim2;  
}  
  
public void setDim2(double dim2) {  
    this.dim2 = dim2;  
}  
  
public abstract double calcular_area();  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public double getDim2() {  
    return dim2;  
}  
  
public void setDim2(double dim2) {  
    this.dim2 = dim2;  
}  
// metodo abstrato  
public abstract double calcular_area();  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public class Losango extends  
Quadrilatero{  
  
    public Losango(double dim1, double  
dim2) {  
        // chamada do construtor da  
superclasse  
        super(dim1, dim2);  
    }  
  
    @Override  
    public double calcular_area() {  
        // formula da area do losango A =  
(D.d)/2  
        return (super.getDim1() *  
super.getDim2()) / 2;  
    }  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public class Retangulo extends Quadrilatero{  
  
    public Retangulo(double dim1, double dim2) {  
        // chamada do contrutor da superclasse  
        super(dim1, dim2);  
    }  
  
    @Override  
    public double calcular_area() {  
        // area do retangulo A = b.h  
        return super.getDim1() * super.getDim2();  
    }  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public class Trapezio extends Quadrilatero{  
    private double dim3;  
  
    public Trapezio(double dim1, double dim2,  
double dim3) {  
        // chamada do contrutor da superclasse  
        super(dim1, dim2);  
        this.dim3 = dim3;  
    }  
  
    @Override  
    public double calcular_area() {  
        // area do trapezio A = ((B+b).h)/2  
        return ((this.dim3 +  
super.getDim1())*super.getDim2())/2;  
    }  
}
```

Princípios da Orientação a Objetos

POLIMORFISMO

```
public class PolimorfismoFormas {  
  
    public static void main(String[] args) {  
        Quadrilatero qd;  
        Retangulo r = new Retangulo(3,7);  
        Losango l = new Losango(3,7);  
        Trapezio t = new Trapezio(3,7,5);  
  
        qd = r;  
        System.out.println("Área: "+  
qd.calcular_area());  
        qd = l;  
        System.out.println("Área: "+  
qd.calcular_area());  
        qd = t;  
        System.out.println("Área: "+  
qd.calcular_area());  
    }  
}
```

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

Existem situações onde é desejável definir a estrutura de uma classe sem necessariamente definir uma implementação concreta para todos os métodos.

As classes abstratas são modelos para outras classes, definem apenas a forma generalizada. O trabalho de preenchimento fica na responsabilidade das subclasses.

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

A superclasse define a natureza dos métodos que a subclasse deve implementar.

Uma classe abstrata não pode ser instanciada por si só.

Para criar um objeto de uma classe abstrata deve-se criar uma classe mais especializada herdando dela.

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

Os métodos da classe abstrata devem então serem sobrescritos nas subclasses.

É muito comum que um método de uma superclasse não tenha uma implementação significativa. Existem duas formas de lidar com isso:

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

Definir uma implementação sem significado (não-apropriada).

- **Exemplo:** retornar 0 no cálculo da área de um quadrilátero.

Deixar sem implementação obrigando que a subclasse sobrescreva os métodos necessários através de **métodos abstratos**.

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

Um método abstrato não possui corpo.

```
modificador abstract tipo  
nomeMetodo (listaParametros) ;
```

Uma classe com métodos abstratos deve ser declarada como abstrata. Mas é possível declarar uma classe abstrata sem que ela possua métodos abstratos.

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

```
public abstract class Quadrilatero{  
    public abstract double calcular_area();  
}
```

Classes abstratas não podem ser instanciadas (com new), ou seja, não existem objetos dela.

Princípios da Orientação a Objetos

CLASSES ABSTRATAS

O método na subclasse tem uma **annotation** conhecida como **@Override**, significando que estamos sobrescrevendo o método da superclasse.

```
@Override  
public abstract double calcular_area() {  
    // corpo do metodo  
};
```

CLASSE OBJECT

Java define um tipo especial de classe: a classe **Object**.

Todas as classes em Java são subclasses de **Object**

A herança é implícita.

Isso significa que uma referência do tipo Object pode referenciar qualquer objeto de outras classes.

CLASSE OBJECT

Os métodos definidos na classe Object são herdados por todas as classes criadas.

Método **toString** é público e pode ser sobrescrito.

```
@Override  
public String toString() {  
    return "Quadrilatero{" + "dim1:  
" + dim1 + ", dim2: " + dim2 + '}';  
}
```

Palavra reservada

FINAL

A palavra-reservada final pode ser utilizada para três finalidades em Java:

- Prevenir que o valor de uma variável seja alterado depois de definido, variável constante.
- Prevenir que um método seja sobreescrito.
- Prevenir que uma classe seja herdada.

```
final double pi = 3.14;  
pi = 10;
```

Error: cannot assing a value to
final variable pi.

Palavra reservada

FINAL

```
public class Quadrilatero{  
    public final void imprimir(){  
        System.out.println("Metodo  
Final.");  
    }  
}
```

```
public class Trapezio{  
    public final void imprimir(){  
  
        System.out.println("Modificando Metodo  
Final.");  
    }  
}
```

Error: imprimir() in Trapezio ca
not override imprimir() in
Quadrilatero overriden method is
final.

INTERFACE

Interface são similares a classes, porém
não possuem variáveis de instância.
Permitem apenas constantes.

Seus métodos são declarados sem corpo
(abstratos).

Para implementar uma interface, a classe
deve criar o conjunto completo de métodos
definidos.

Desconecta a definição de um conjunto de
métodos da hierarquia de herança.

INTERFACE

Isso permite que classes não relacionadas a hierarquia possam implementar a mesma interface.

Interfaces definem tipo de forma abstrata como uma espécie de **contrato**.

Não possui implementação e portanto não podem ser usadas para criar instâncias.

INTERFACE

A ideia por trás de interface é bem simples: fazer com que todos os métodos sejam abstratos, **obrigando** a subclasse a implementar os métodos.

Uma outra vantagem de interfaces é que **uma classe pode implementar várias delas**.

Os métodos são sempre **public** e **abstract**.

Variáveis são sempre **public**, **final** e **static**.

INTERFACE

Depois de definir uma interface, uma ou mais classes podem implementá-la através da palavra-reservada **implements** em uma declaração de classe.

A classe deve prover implementação para todos métodos definidos por essa interface.

```
modificador interface nomeInterface{  
    tipo metodo (parametros) ;  
}
```

```
modificador class nomeClasse implements  
nomeInterface{  
    // corpo da classe  
}
```

INTERFACE

```
public interface Figura {  
    public void getNomeFigura();  
    public double calcularPerimetro();  
}  
  
public class Quadrado extends Quadrilatero implements Figura{  
    public Quadrado(double dim1, double dim2) {  
        // chamada ao construtor da superclasse  
        super (dim1, dim2);  
    }  
    @Override  
    public double calcularArea() {  
        // calcula a area de retangulo  
        return Math.pow(super.getDim1(), 2);  
    }  
    @Override  
    public String getNomeFigura() {  
        return "Quadrado" ;  
    }  
    @Override  
    public double calcularPerimetro() {  
        return super.getDim1() * 4;  
    }  
}
```

JOGO DE BATALHA ENTRE PERSONAGENS

1. Objetivo

O objetivo desta atividade é praticar os conceitos de orientação a objetos em Java, incluindo:

- Classes e objetos
- Métodos e atributos
- Herança e polimorfismo
- Interfaces
- Classes abstratas



2. Descrição do Jogo

O jogo permite uma batalha entre dois personagens de diferentes classes:

- **Guerreiro:** Especialista em combate corpo a corpo; seu principal atributo é Força.
- **Arqueiro:** Especialista em combate de longa distância; seu principal atributo é Agilidade.
- **Mago:** Especialista em combate de média distância; seu principal atributo é Inteligência.

Cada personagem tem habilidades específicas que influenciam seus ataques e defesas na batalha.

JOGO DE BATALHA ENTRE PERSONAGENS

3. Implementação

3.1. Classe Abstrata Personagem

A classe Personagem deve ser abstrata e conter os seguintes atributos:

- nome (String)
- força (int)
- agilidade (int)
- inteligencia (int)
- pontosVida (int)

Métodos:

- Construtor que define apenas o nome.
- Getters e setters para todos os atributos.

Métodos abstratos:

- calcularPontosVida()
- calcularDano(int numeroAleatorio)
- Método `toString()` que exibe os atributos do personagem.

JOGO DE BATALHA ENTRE PERSONAGENS

3.2. Classes Guerreiro, Arqueiro e Mago

Cada uma dessas classes herda de Personagem e implementa as regras de atributos e cálculo de vida e dano.

- **Guerreiro**

- Atributos adicionais (constantes):
- adrenalina = 8% (chance de ataque bônus)
- heroismo = 6% (chance de defesa bônus)
- Implementação dos métodos abstratos:
- Pontos de Vida = $5 * \text{força} + r$
- Dano = $3 * \text{força} + r$

- **Arqueiro**

- Atributos adicionais (constantes):
- tiroCerteiro = 15%
- camuflagem = 3%
- Implementação dos métodos abstratos:
- Pontos de Vida = $4 * \text{agilidade} + r$
- Dano = $2 * \text{agilidade} + r$

JOGO DE BATALHA ENTRE PERSONAGENS

- **Mago**

- Atributos adicionais (constantes):
 - cura = 5%
 - conflagar = 10%
- Implementação dos métodos abstratos:
 - Pontos de Vida = $7 * \text{inteligencia} + r$
 - Dano = $2 * \text{inteligencia} + r$

r representa um número aleatório para adicionar variação nos cálculos

JOGO DE BATALHA ENTRE PERSONAGENS

4. Mecanismo do Jogo

A batalha acontece através da classe JogoBatalha, que implementa a interface Jogo.

4.1. Interface Jogo

- void setarAtributos(List<Personagem> personagens);
- double calcularBonusAtaque(Personagem p, double dano);
- double calcularBonusDefesa(Personagem p, double dano);
- void atacar(Personagem atacante, Personagem defensor);

4.2. Implementação em JogoBatalha

Define os atributos de batalha dos personagens aleatoriamente dentro dos seguintes intervalos:

- **Guerreiro:** Força (0-100), Agilidade (0-50), Inteligência (0-50)
- **Arqueiro:** Força (0-50), Agilidade (0-100), Inteligência (0-50)
- **Mago:** Força (0-50), Agilidade (0-50), Inteligência (0-100)

JOGO DE BATALHA ENTRE PERSONAGENS

- **Bônus de Ataque:**
 - **Guerreiro** (adrenalina): aumenta o dano em 40%
 - **Arqueiro** (tiroCerteiro): aumenta o dano em 25%
- **Mago** (conflagar): aumenta o dano em 55%
- **Bônus de Defesa:**
 - **Guerreiro** (heroismo): reduz o dano recebido em 50%
 - **Arqueiro** (camuflagem): reduz 100% do dano recebido
 - **Mago** (cura): restaura todos os pontos de vida

JOGO DE BATALHA ENTRE PERSONAGENS

4.3. Simulação da Batalha

A classe Principal irá:

- Exibir um menu de opções até que o usuário escolha sair.
- Permitir que o usuário escolha dois personagens, definindo seus nomes.
- Gerar atributos de batalha aleatoriamente para os personagens.
- Escolher um personagem para iniciar o ataque.
- Alternar ataques até que um dos personagens atinja pontosVida ≤ 0 .
- Exibir os dados do vencedor e do perdedor.

4.4. Regras Adicionais

- O jogo só pode iniciar se houver exatamente dois personagens.
- Se um personagem atingir 0 pontos de vida antes de atacar, ele não tem chance de revidar.
- O combate deve ser justo, alternando ataques entre os personagens.

CONTATO



RAFAELA PRISCILA
CRUZ MOREIRA



rafapcmoreira@gmail.com



rafaelapcmoreira