

# Orientação a Objetos com Java

## Apresentação do básico

Alisson Chiquitto

<sup>1</sup>Sistemas para Internet

Umuarama, 2016

“O paradigma de orientação a objetos, se comparado à programação estrutural, adota formas mais próximas do mecanismo humano para gerenciar a complexidade de um sistema. Neste paradigma, o mundo real é visto como sendo constituído de objetos, cada um com seu próprio estado (atributos) e comportamento (métodos), semelhante ao correspondente no mundo real.”

## 1 Classes & Objetos

- Classe
- Objeto
- Encapsulamento
- Sobrecarga (Overloading)

## 2 Herança

## 3 Classes Abstratas

## 4 Interfaces

Classe é um **molde** que será usado para construir objetos que representam elementos da vida real. Classes determinam:

- Características (**propriedades**);
- Comportamentos (**métodos**);
- Relacionamentos com outros objetos;

# Classe Produto

Produto
+ ref : String + nome : String

```
public class Produto {  
    private String ref;  
    private String nome;  
}
```

Objeto é a materialização (**instância**) de uma **Classe**.

Pode ser conceitual ou físico:

Conceitual	Físico
Consulta médica	Receita médica
Conta bancária	Cliente
Conexão com BD	Computador

**Tabela:** Objetos conceituais e físicos

# Classe Produto e instâncias

<b>Produto</b>
+ ref : String + nome : String

ref	nome
AAA	Computador
BBB	Teclado
CCC	Mouse

Tabela: Instâncias da classe Produto

# Exercício 1.1

1. Escolha um **objeto físico** de sua preferência;
2. Descreva os atributos;
3. Descreva os métodos;
4. Instanciar a classe;
5. Atribuir valores iniciais ao objeto criado;
6. Utilizar os métodos definidos para o objeto;
7. Exibir no console o valor de todos os atributos;



# Resolução: Exercício 1.1 I

```
public class Garrafa {  
    public int volumeTotal;  
    public int volumeAtual;  
    public String cor;  
    public boolean aberta;  
  
    public void abrir() {  
        aberta = true;  
    }  
    public void fechar() {  
        aberta = false;  
    }  
}
```

# Resolução: Exercício 1.1 II

```
public class GarrafaTest {  
    public static void main(String[] args) {  
        Garrafa g = new Garrafa();  
        g.cor = "Vermelho";  
        g.volumeAtual = 250;  
        g.volumeTotal = 500;  
        g.abrir();  
        System.out.println(g.cor);  
        System.out.println(g.volumeAtual);  
        System.out.println(g.volumeTotal);  
        System.out.println(g.aberta);  
    }  
}
```

# Resultado:

Vermelho

250

500

true

## Exercício 1.2

1. Escolha um **objeto conceitual** de sua preferência;
2. Descreva os atributos;
3. Descreva os métodos;
4. Instanciar a classe;
5. Atribuir valores iniciais ao objeto criado;
6. Utilizar os métodos definidos para o objeto;
7. Exibir no console o valor de todos os atributos;

# Resolução: Exercício 1.2 I

```
public class Arquivo {  
    public String conteudo;  
    public boolean aberto;  
    public void abrir() {  
        aberto = true;  
    }  
    public void fechar() {  
        aberto = false;  
    }  
    public boolean salvar(String c) {  
        if (!aberto) { return false; }  
        conteudo = c;  
        fechar();  
        return true;  
    }  
}
```

# Resolução: Exercício 1.2 II

```
public class ArquivoTest {  
    public static void main(String[] args) {  
        Arquivo o = new Arquivo();  
        o.abrir();  
        o.salvar("LoremIpsum");  
        System.out.println(o.aberto);  
        System.out.println(o.conteudo);  
    }  
}
```

```
# Resultado:  
false  
LoremIpsum
```

## Exercício 1.3

Dado uma classe Conta:

<b>Conta</b>
+ saldo : double
+ deposito(double valor) : void + saque(double valor) : void

1. Implementar a classe Conta;
2. Implementar os métodos:  
    deposito()  
    saque()
3. Testar a classe;

# Resolução: Exercício 1.3 I

```
public class Conta {  
    public double saldo = 0;  
    public void deposito(double valor) {  
        saldo += valor;  
    }  
    public void saque(double valor) {  
        saldo -= valor;  
    }  
}
```

# Resolução: Exercício 1.3 II

```
public class ContaTest {  
    public static void main(String[] args) {  
        Conta c = new Conta();  
        c.deposito(1000);  
        c.saque(250);  
  
        System.out.println(c.saldo);  
    }  
}
```



## Exercício 1.4

A partir da resolução do exercício anterior: Acrescente um método na classe Conta para implementar a lógica de transferência de valores entre contas.

Conta
+ saldo : double
+ deposito(double valor) : void + saque(double valor) : void + transferir(double valor, Conta conta) : void

# Resolução: Exercício 1.4 I

```
public class Conta {  
    public double saldo = 0;  
    public void deposito(double valor) {  
        saldo += valor;  
    }  
    public void saque(double valor) {  
        saldo -= valor;  
    }  
    public void transferir(double valor, Conta conta) {  
        saque(valor);  
        conta.deposito(valor);  
    }  
}
```

# Resolução: Exercício 1.4 II

```
public class ContaTest {  
    public static void main(String[] args) {  
        Conta c1 = new Conta();  
        Conta c2 = new Conta();  
        c1.deposito(1000);  
        c1.transferir(300, c2);  
        System.out.println(c1.saldo);  
        System.out.println(c2.saldo);  
    }  
}
```

# Resultado:

700.0

300.0

Garante (obriga) que um determinado trecho de código seja executado toda vez que uma classe seja instanciada.

- Força a execução de um trecho de código;
- Evita erros pela falta da execução de códigos;

<b>Produto</b>
+ ref: String + nome: String
+ Produto(String ref) : void

# Encapsulamento

Técnica para minimizar as interdependências entre objetos, através da definição de interfaces externas (definição de métodos que manipulam os dados internos da classe).

<b>Produto</b>
- ref : String - nome : String
+ Produto(String ref) : void + getNome() : String + setNome(String nome) : void

- Facilitar a identificação de erros;
- Controle centralizado dos valores dos atributos.

# Sobrecarga (Overloading)

Existência de vários métodos com o mesmo nome, na mesma Classe, porém com variações em:

- Quantidade de argumentos;
- Tipo dos argumento;
- Tipo de retorno.

Soma
+ somar(int a, int b) : int + somar(String a, String b) : String + somar(double a, double b) : double

## Exercício 1.5

Implementar a classe Soma de acordo com a figura a seguir. Fique atento as assinaturas dos métodos.

Soma
+ somar(int a, int b) : int + somar(String a, String b) : String + somar(double a, double b) : double

# Resolução: Exercício 1.5 I

```
public class Soma {  
    public String somar(String a, String b) {  
        return a + b;  
    }  
    public int somar(int a, int b) {  
        return a + b;  
    }  
    public double somar(double a, double b) {  
        return a + b;  
    }  
}
```



# Resolução: Exercício 1.5 II

```
public class SomaTest {  
    public static void main(String[] args) {  
        Soma s = new Soma();  
        System.out.println(s.somar("A", "B"));  
        System.out.println(s.somar(1, 2));  
        System.out.println(s.somar(1.5, 2.5));  
    }  
}
```

# Resultado:

AB

3

4.0

## Exercício 1.6

Crie uma classe chamada `Funcionario` para definir funcionários de uma empresa. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

# Resolução: Exercício 1.6 I

```
class Funcionario {  
    public double salario;  
  
    public void aumentaSalario() {  
        aumentaSalario(0.1);  
    }  
  
    public void aumentaSalario(double taxa) {  
        salario += salario * taxa;  
    }  
}
```

# Resolução: Exercício 1.6 II

```
public class FuncionarioTest {  
    public static void main(String[] args) {  
        Funcionario f = new Funcionario();  
        f.salario = 1000;  
        f.aumentaSalario();  
        System.out.println(f.salario);  
        f.salario = 1000;  
        f.aumentaSalario(0.2);  
        System.out.println(f.salario);  
    }  
}
```

# Resultado:

1100.0

1200.0

## Exercício 1.7

Uma pizzeria possui varias mesas, cada uma com um saldo da conta (valor consumido). Ao final da consumação o cliente tem a possibilidade de pagar toda a conta, ou então pagar a conta parcialmente. Sendo assim:

1. Criar uma classe *Mesa*, com o atributo *saldo*;
2. Criar o método *pagar(valor)*, que subtrai *valor* do saldo da mesa;
3. Criar o método *pagar()* que subtrai todo o saldo devedor da mesa.

# Resolução: Exercício 1.7 I

```
public class Mesa {  
    public double conta;  
  
    public void pagar() {  
        pagar(conta);  
    }  
    public void pagar(double valor) {  
        conta -= valor;  
    }  
}
```

# Resolução: Exercício 1.7 II

```
public class MesaTest {  
    public static void main(String[] args) {  
        Mesa m = new Mesa();  
        m.conta = 100;  
        m.pagar(60);  
        System.out.println(m.conta);  
        m.pagar();  
        System.out.println(m.conta);  
    }  
}
```

# Resultado:

40.0

0.0

## 1 Classes & Objetos

## 2 Herança

- Herança
- Polimorfismo
- Sobrescrita (Override)

## 3 Classes Abstratas

## 4 Interfaces

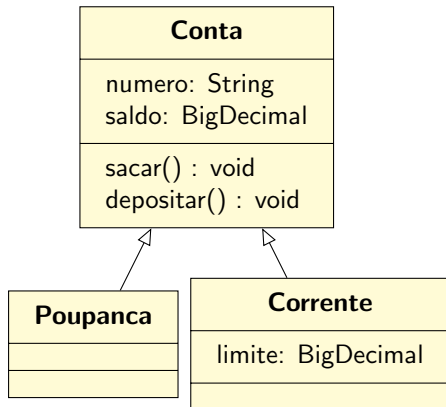


Herança é um princípio que permite a criação de classes como especializações de classes já existentes.

- Super-classe: A classe base ou mais genérica;
- Sub-classe: A especialização;

A sub-classe herda os atributos e métodos da super-classe.

# Exemplo de Herança

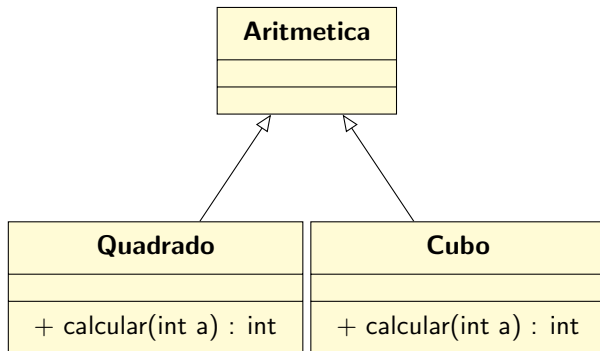


- Super-classe: Conta
- Sub-classes: Poupanca/Corrente

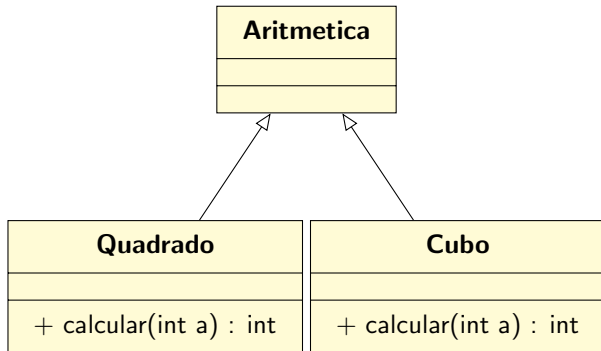
## Polimorfismo

Do grego, " muitas formas" (poli = muitas, morphos = formas).

Princípio a partir do qual, Classes derivadas de uma única classe base são capazes de invocar métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente.



## Exercício 2.1



Faça:

1. Implementar todas as Classes exibidas;
2. Criar mais duas classes a seu gosto, dentro do contexto Polimorfismo.
3. Criar uma Classe para testar as Classes criadas;

# Resolução: Exercício 2.1 I

```
public class Aritmetica {}  
public class Quadrado extends Aritmetica {  
    public int calcular(int a) {  
        return a * a;  
    }  
}  
public class Cubo extends Aritmetica {  
    public int calcular(int a) {  
        return a * a * a;  
    }  
}
```

# Resolução: Exercício 2.1 II

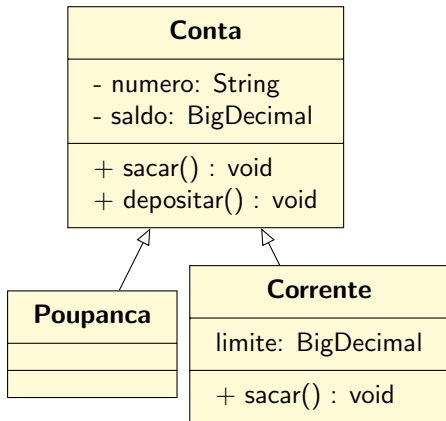
```
public static void main(String[] args) {  
    Quadrado q = new Quadrado();  
    System.out.println(q.calcular(2));  
    Cubo c = new Cubo();  
    System.out.println(c.calcular(2));  
}
```

# Resultado:

4

8

# Sobrescrita (Override)



Capacidade de especializar métodos herdados durante a herança de classes, alterando o comportamento de métodos nas subclasses por um mais específico.

## Exercício 2.2

1. Criar uma classe Conta, que possua um saldo os métodos para depositar e sacar;
2. Adicione um método na classe Conta, que atualiza o saldo de acordo com uma taxa percentual fornecida.
3. Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa.
4. Além disso, a ContaCorrente deve reescrever o método deposita, a fim de retirar uma taxa bancária de dez centavos de cada depósito.



## Resolução: Exercício 2.2 I

```
public class Conta {  
    protected double saldo;  
    public void atualiza(double taxa) {  
        saldo += saldo * taxa;  
    }  
    public void deposita(double valor) {  
        saldo += valor;  
    }  
    public double saldo() {  
        return saldo;  
    }  
    public void saque(double valor) {  
        saldo -= valor;  
    }  
}
```

## Resolução: Exercício 2.2 II

```
public class ContaCorrente extends Conta {
    @Override
    public void atualiza(double taxa) {
        saldo += saldo * taxa * 2;
    }
    @Override
    public void deposita(double valor) {
        saldo += valor - 0.1;
    }
}

public class ContaPoupanca extends Conta {
    @Override
    public void atualiza(double taxa) {
        saldo += saldo * taxa * 3;
    }
}
```

## Resolução: Exercício 2.2 III

```
public class ContaCorrenteTest {  
    public static void main(String[] args) {  
        ContaCorrente cc = new ContaCorrente();  
        cc.deposita(1000.1);  
        System.out.println(cc.saldo());  
        cc.atualiza(0.2);  
        System.out.println(cc.saldo());  
        cc.saque(100);  
        System.out.println(cc.saldo());  
    }  
}
```

# Resultado:

```
1000.0  
1400.0  
1300.0
```

## Resolução: Exercício 2.2 IV

```
public class ContaPoupancaTest {  
    public static void main(String[] args) {  
        ContaPoupanca cc = new ContaPoupanca();  
        cc.deposita(1000);  
        System.out.println(cc.saldo());  
        cc.atualiza(0.2);  
        System.out.println(cc.saldo());  
        cc.saque(100);  
        System.out.println(cc.saldo());  
    }  
}
```

# Resultado:

```
1000.0  
1600.0  
1500.0
```

## Exercício 2.3

Escreva a classe `LampadaFluorescente` como sendo herdeira da classe `Lampada`. A classe `LampadaFluorescente` deve ter um campo que represente o comprimento da lampada em centímetros. Crie nesta classe um construtor para inicializar os seus dados.

# Resolução: Exercício 2.3 I

```
public class Lampada {}  
public class LampadaFluorescente  
    extends Lampada {  
    public int comprimento;  
  
    public LampadaFluorescente(int c) {  
        comprimento = c;  
    }  
}
```

## Resolução: Exercício 2.3 II

```
public class LampadaTest {  
    public static void main(String[] args) {  
        LampadaFluorescente l = new LampadaFluorescente(10);  
        System.out.println(l.comprimento);  
    }  
}
```

# Resultado:  
10

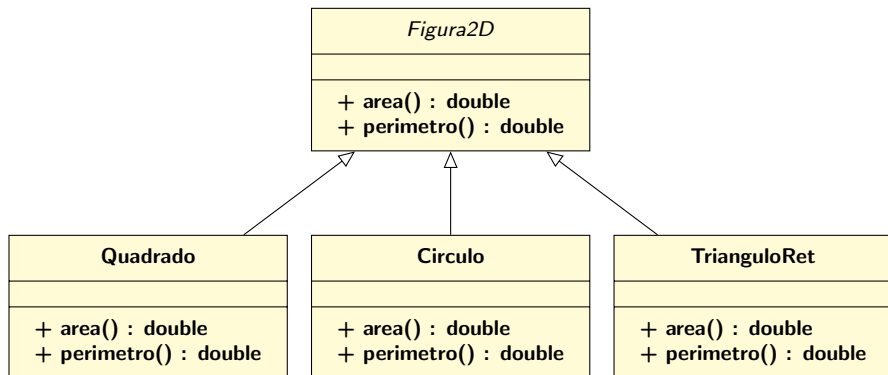
# Outline - Seção

- 1 Classes & Objetos
- 2 Herança
- 3 Classes Abstratas**
- 4 Interfaces

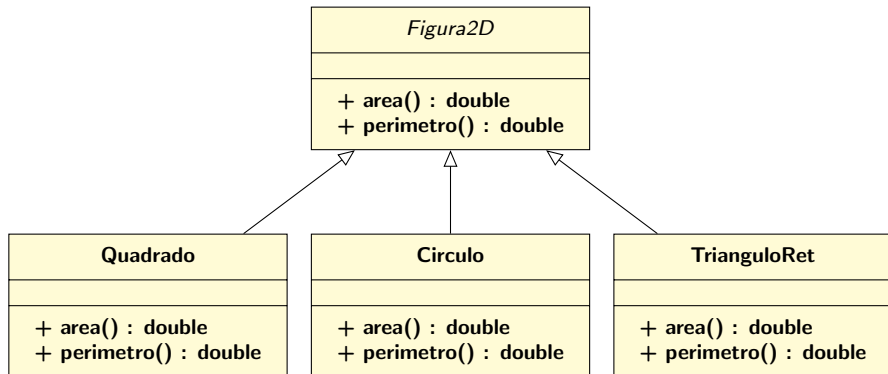


- **Classe Abstrata:** Classes exclusivamente criadas para servirem de modelos para classes derivadas, e não podem ser instanciadas;
- **Método Abstrato:** Devem estar contidos apenas em Classes Abstratas, e não possuem implementação, ou seja, apenas a assinatura.
- **Classes derivadas** devem implementar todos os métodos abstratos definidos na classe abstrata;

# Classes & Métodos abstratos



## Exercício 3.1



Implementar em Java as classes exibidas na figura

# Resolução: Exercício 3.1 I

```
abstract public class Figura2D {
    public abstract double area();
}

public class Quadrado extends Figura2D {
    private double base;

    public Quadrado(double base) {
        this.base = base;
    }

    @Override
    public double area() {
        return base * base;
    }

    @Override
    public double perimetro() {
        return base * 4;
    }
}
```

# Resolução: Exercício 3.1 II

```
public class Circulo extends Figura2D {  
    private double raio;  
  
    public Circulo(double raio) {  
        this.raio = raio;  
    }  
  
    @Override  
    public double area() {  
        return Math.PI * raio * raio;  
    }  
  
    @Override  
    public double perimetro() {  
        return 2 * Math.PI * raio;  
    }  
}
```

# Resolução: Exercício 3.1 III

```
public class TrianguloRet extends Figura2D {
    private double altura;
    private double base;
    public TrianguloRet(double altura, double base) {
        this.altura = altura;
        this.base = base;
    }

    public double hipotenusa() {
        return Math.sqrt((altura + altura) + (base * base));
    }
    @Override
    public double area() {
        return base * altura / 2;
    }
    @Override
    public double perimetro() {
        return base + altura + hipotenusa();
    }
}
```

# Resolução: Exercício 3.1 IV

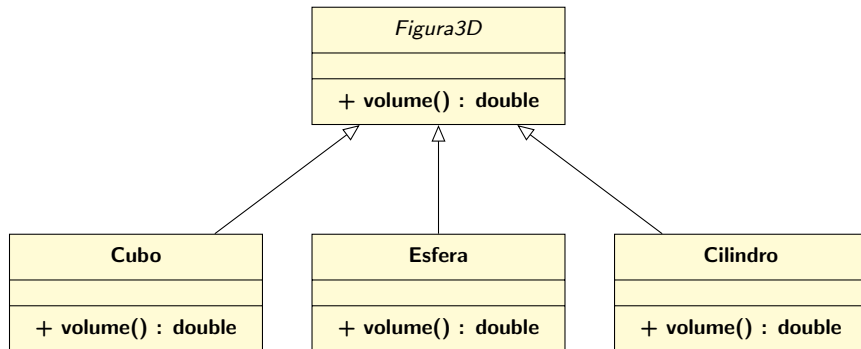
```
public class Figura2DTest {  
    public static void main(String[] args) {  
        Figura2D q = new Quadrado(3);  
        Figura2D c = new Circulo(3);  
        Figura2D t = new TrianguloRet(3,4);  
  
        System.out.println("Areas");  
        System.out.println("Quadrado:" + q.area());  
        System.out.println("Circulo:" + c.area());  
        System.out.println("Triangulo:" + t.area());  
  
        System.out.println("Perimetros");  
        System.out.println("Quadrado:" + q.perimetro());  
        System.out.println("Circulo:" + c.perimetro());  
        System.out.println("Triangulo:" + t.perimetro());  
    }  
}
```

# Resolução: Exercício 3.1 V

```
# Resultado  
Areas  
Quadrado:9.0  
Circulo:28.27  
Triangulo:6.0  
Perimetros  
Quadrado:12.0  
Circulo:18.84  
Triangulo:11.69
```



## Exercício 3.2



Implementar em Java as classes exibidas na figura.

# Resolução: Exercício 3.2 I

```
abstract public class Figura3D {  
    public abstract double volume();  
}  
  
public class Cubo extends Figura3D {  
    private double base;  
    public Cubo(double base) {  
        this.base = base;  
    }  
  
    @Override  
    public double volume() {  
        return Math.pow(base, 3);  
    }  
}
```

## Resolução: Exercício 3.2 II

```
public class Cilindro extends Figura3D {
    private double altura;
    private double raio;
    public Cilindro(double altura, double raio) {
        this.altura = altura;
        this.raio = raio;
    }

    @Override
    public double volume() {
        double area = Math.PI * raio * raio;
        return area * altura;
    }
}
```

## Resolução: Exercício 3.2 III

```
public class Piramide extends Figura3D {  
    private double altura;  
    private double base1;  
    private double base2;  
  
    public Piramide(double a, double b1, double b2) {  
        this.altura = a;  
        this.base1 = b1;  
        this.base2 = b2;  
    }  
  
    @Override  
    public double volume() {  
        double area = base1 * base2 / 2;  
        return area * altura / 3;  
    }  
}
```

## Resolução: Exercício 3.2 IV

```
public class Figura3DTest {  
    public static void main(String[] args) {  
        Figura3D cil = new Cilindro(3, 4);  
        Figura3D cub = new Cubo(3);  
        Figura3D pir = new Piramide(1, 2, 3);  
  
        System.out.println("Cilindro:" + cil.volume());  
        System.out.println("Cubo:" + cub.volume());  
        System.out.println("Piramide:" + pir.volume());  
    }  
}
```

```
# Resultado  
Cilindro:150.79  
Cubo:27.0  
Piramide:1.0
```

# Outline - Seção

- 1 Classes & Objetos
- 2 Herança
- 3 Classes Abstratas
- 4 Interfaces**

# Interfaces (Contratos)

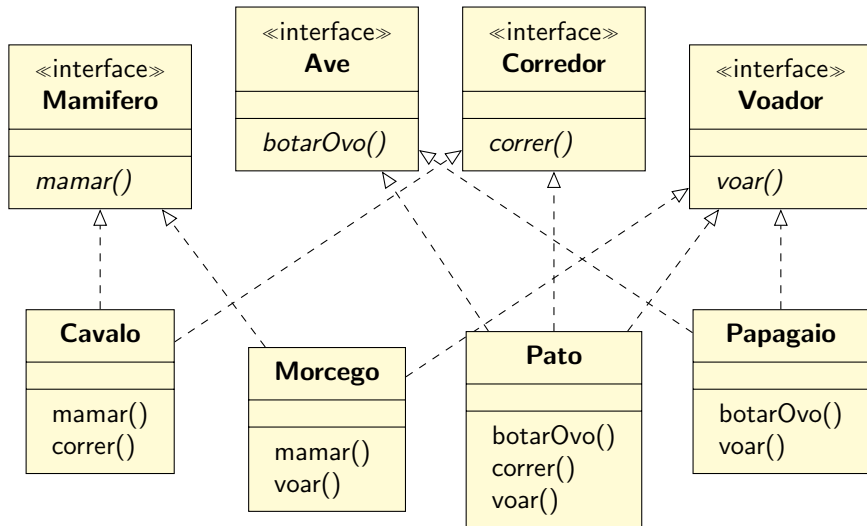
## Interface

Um padrão é definido através de especificações ou contratos. Nas aplicações orientadas a objetos, podemos criar um "contrato" para definir um determinado conjunto de métodos que deve ser implementado pelas classes que "assinarem" este contrato.

## Compilação


A referência (interface) é conhecida em tempo de compilação mas o objeto a que ela aponta (implementação) não é

# Interfaces (Contratos)





 HORSTMANN, Cay. Big Java. Bookman, 2009.

 MENDES, Douglas Rocha. Programação Java com Ênfase em Orientação a Objetos. Novatec, 2009.