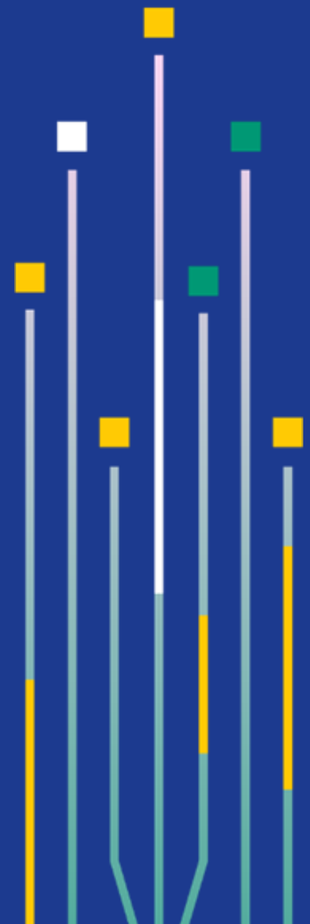
Profª Lucília Ribeiro

Pilares da POO
- Polimorfismo -

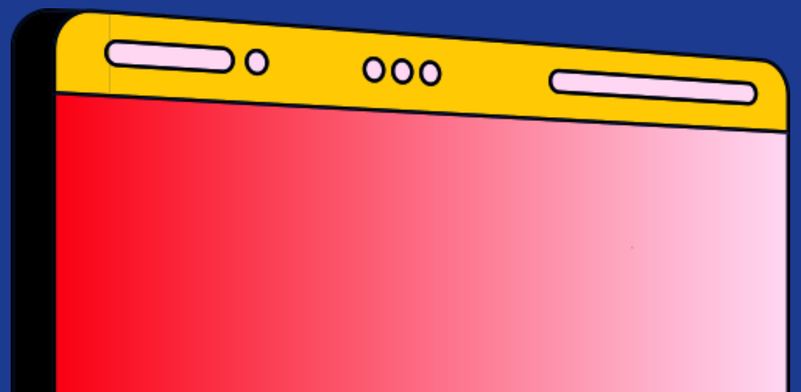
"Antes do software poder ser reutilizável ele primeiro tem de ser utilizável.”
(Ralph Johnson)



OS TRÊS PILARES

ENCAPSULAMENTO

É a característica da OO de ocultar partes independentes da implementação.
O encapsulamento permite que você construa partes ocultas da implementação do mundo exterior.



HERANÇA

É um mecanismo que permite a uma classe (chamada de classe filha ou subclasse) herdar propriedades e comportamentos de outra classe (chamada de classe pai ou superclasse). A herança promove a reutilização de código e estabelece uma hierarquia entre classes.

POLIMORFISMO

É a capacidade de um objeto de tomar muitas formas diferentes. Capacidade de um método ter diferentes implementações, dependendo do objeto que o chama, ou de um método ser chamado de maneira uniforme em diferentes tipos de objetos.

INTERFACE

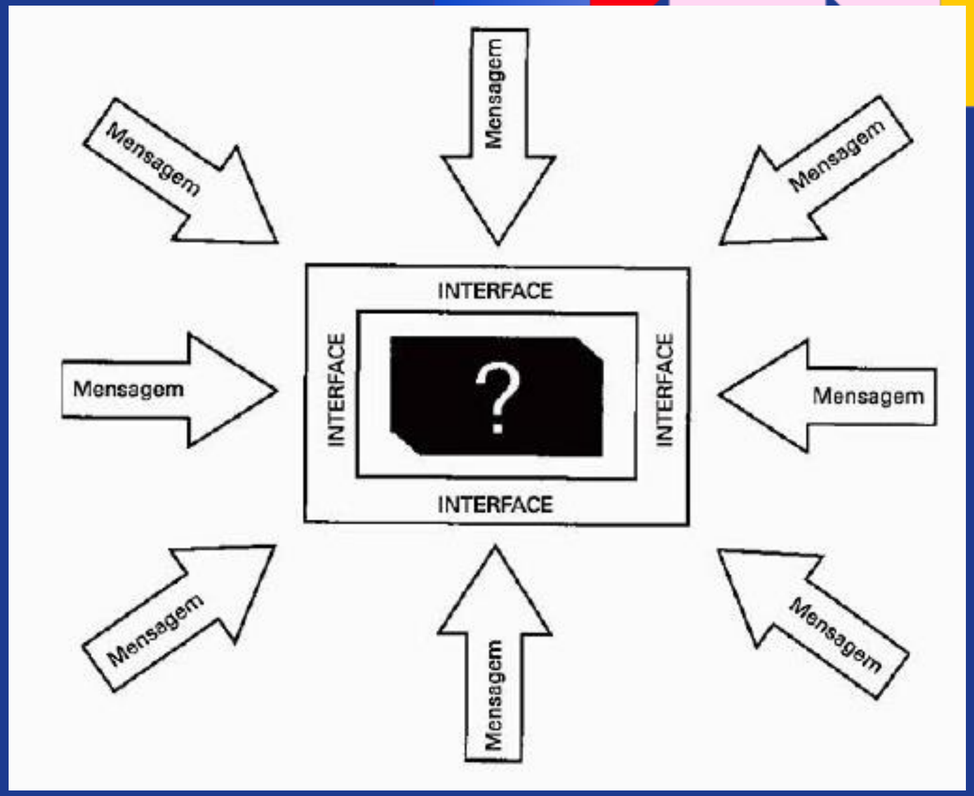
Lista os serviços fornecidos por um componente. A interface é um contrato com o mundo exterior, que define exatamente o que uma entidade externa pode fazer com o objeto. Uma interface é o painel de controle do objeto

API

Uma interface é semelhante a uma API (Interface de Programa Aplicativo) para um objeto. A interface lista todos os métodos e argumentos que o objeto entende

IMPLEMENTAÇÃO

Define como um componente realmente fornece um serviço. A implementação define todos os detalhes internos do componente.



```java
public class Log {
    public void debug(String msg) {
        print("DEBUG ", msg);
    }
    public void info(String msg) {
        print("INFO ", msg);
    }
    public void warning(String msg) {
        print("WARNING ", msg);
    }
    public void error(String msg) {
        print("ERROR ", msg);
    }
    public void fatal(String msg) {
        print("FATAL ", msg);
        System.exit(0);
    }
    private void print(String msg, String gravidade) {
        System.out.println(gravidade + ": " + msg);
    }
}
```

Interface pública

```
public void debug(String msg)
public void info(String msg)
public void warning(String msg)
public void error(String msg)
public void fatal(String msg)
```

NÍVEIS DE ACESSO

01

PÚBLICO

Garante acesso a todos os objetos

02

PROTEGIDO

Garante acesso à instância, ou seja, para aquele objeto, e para todas as subclasses

03

PRIVADO

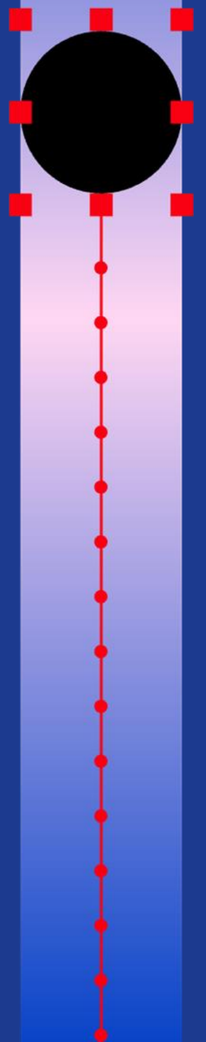
Garante o acesso apenas para a instância, ou seja, para aquele objeto .

Características do Encapsulamento

01 Abstração

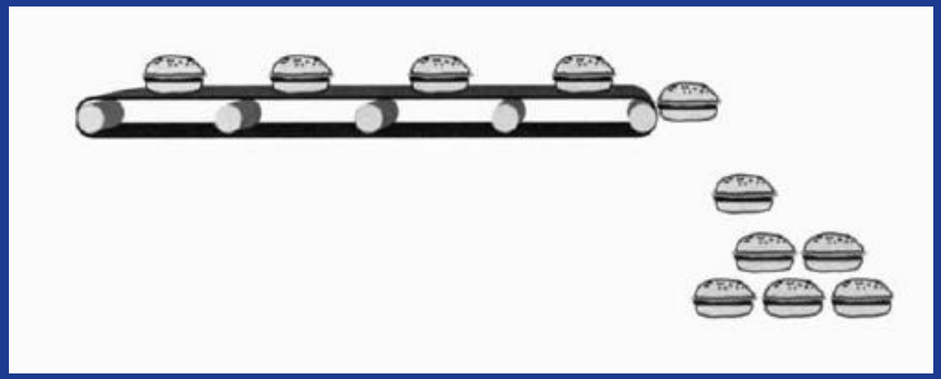
02 Ocultação da implementação

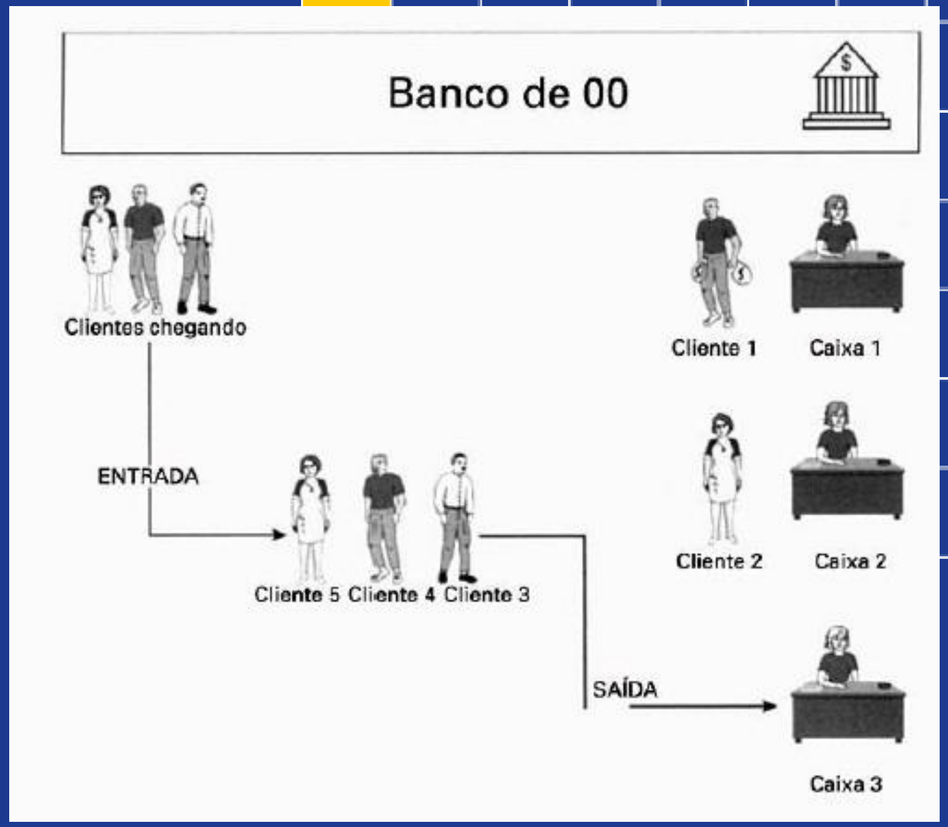
03 Divisão de responsabilidade

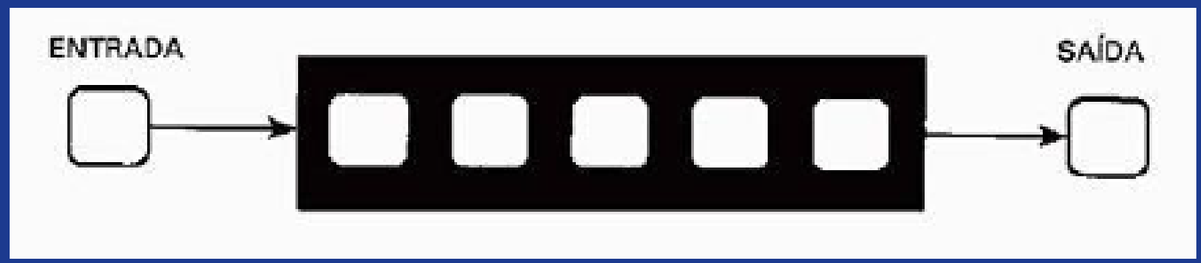


ABSTRAÇÃO

É o processo de simplificar um problema difícil. Quando começa a resolver um problema, você não se preocupa com cada detalhe. Em vez disso, você o simplifica, tratando apenas dos detalhes pertinentes a uma solução.







TAD

Tipo Abstrato de Dados

É um conjunto de dados e um conjunto de operações sobre esses dados. Os TADs permitem que você defina novos tipos na linguagem, ocultando dados internos e o estado, atrás de uma interface bem definida. Essa interface apresenta o TAD como uma única unidade atômica

TIPO

Os tipos definem as diferentes espécies de valores que você pode usar em seus programas. Um tipo define o domínio a partir do qual seus valores válidos podem ser extraídos. Além do domínio, a definição de tipo inclui quais operações são válidas no tipo e quais seus resultados.

exemplo

```java
public class ItemNE {
    public float precoUnitario;
    public float desconto;
    public int quantidade;
    public String descricao;
    public String id;
```

```
Total incorreto: R$ -747.5
Total correto: R$ 2990.0
```

```java
public class ItemNaoEncapsulado {
    public static void main(String[] args) {
        ItemNE monitor = new ItemNE("eletronic-012", "35 polegadas", 1, 2990f);
        float preco;
        monitor.desconto = 1.25f;   //inválido, desconto deve ser menor que 1
        preco = monitor.getReajusteTotal();
        System.out.println("Total incorreto: R$ " + preco);
        monitor.setDesconto(1.25f);
        preco = monitor.getReajusteTotal();
        System.out.println("Total correto: R$ " + preco);
    }
}
```

Tipos de Códigos

FRACAMENTE ACOPLADO

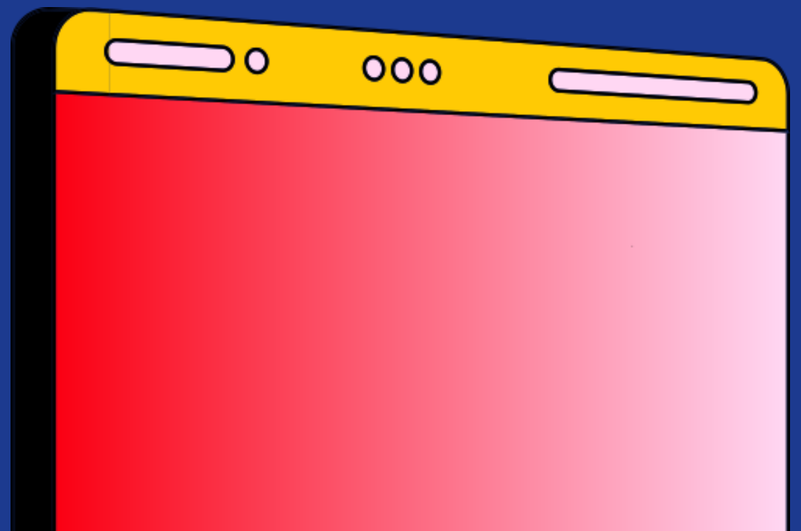
É independente da implementação de outros componentes.

FORTEMENTE ACOPLADO

É fortemente vinculado à implementação de outros componentes.

DEPENDENTE

É dependente da existência de determinado tipo. O código dependente é inevitável



ENCAPSULAMENTO EFETIVO

Encapsulamento Efetivo = Abstração + Ocultação da implementação + Responsabilidade

- Retire a ABSTRAÇÃO e você terá um Código que não é reutilizável
- Retire a OCULTAÇÃO da implementação e você ficará com um Código fortemente acoplado e frágil
- Retire a RESPONSABILIDADE e você ficará com um Código centrado nos dados, procedural, fortemente acoplado e descentralizado

```java
public class ChaveDupla {
    private String chave1, chave2;
    //um construtor sem argumentos
    public ChaveDupla() {
        chave1 = "chave1";
        chave2 = "chave2";
    }
    //um construtor com argumentos
    public ChaveDupla(String chave1, String chave2) {
        this.chave1 = chave1;
        this.chave2 = chave2;
    }
    //acessor
    public String getChave1() {
        return chave1;
    }
    //mutante
    public void setChave1(String chave1) {
        this.chave1 = chave1;
    }
    //acessor
    public String getChave2() {
        return chave2;
    }
    //mutante
    public void setChave2(String chave2) {
        this.chave2 = chave2;
    }
}
```

exemplo

No NOOBank, os clientes entram na fila, enquanto esperam por um caixa.

Programe uma classe de conta. Seja essa uma conta corrente, conta poupança ou conta de mercado financeiro, todas elas têm algumas características compartilhadas. Todas as contas têm um saldo. Uma conta também permitirá que você deposite valores, saque valores e consulte o saldo.

A classe **Caixa** tem um método **main**() que você usará para testar a implementação de sua conta. A classe **Caixa** espera uma interface específica. Aqui estão as regras:

- Você deve chamar a classe conta de **Conta**
- A classe deve ter os dois construtores a seguir:
  - public Conta()
  - public Conta(float depositoInicial)
  - o construtor noargs configurará o saldo inicial como 0.00. O segundo construtor configurará o saldo inicial como depositoInicial
- A classe deve ter os métodos a seguir. O primeiro método credita na conta o valor de fundos.
  - public void deposita(float valor)
- O método seguinte debita na conta o valor de fundos. Entretanto saque() não deve permitir um saque a descoberto. Em vez disso, se valor for maior que o saldo, apenas debita o resto do saldo. Saque() deve retornar a quantidade real que foi retirada da conta.
  - public float saque(float valor)
- O terceiro método recupera o saldo corrente da conta:
  - public float getSaldo()

Além dessas regras, você pode adicionar quaisquer outros métodos que possa considerar úteis. Entretanto, certifique-se de implementar cada um dos métodos exatamente como listado anteriormente.

NooBank

NooBank

```java
public class Conta {
    private float saldo;
    public Conta(float depositoInicial) {
        saldo = depositoInicial;
    }
    public Conta() {
        saldo = 0f;
    }
    public void deposito(float montante) {
        saldo = saldo + montante;
    }
    public float getSaldo() {
        return saldo;
    }
    public float saque(float montante) {
        if (montante > saldo) {
            montante = saldo;
        }
        saldo = saldo - montante;
        return montante;
    }
}
```

PÔQUER

01

BARALHO

52 cartas

02

CARTA

Naipe (ouros, copas, espadas, paus)
Valor (2 a 10, valete, dama, rei, ás)

03

OCULTAÇÃO

Nunca verá o que está no maço, até receber uma carta.

04

RESPONSABILIDADE

- Uma carta exibe seu naipe e seu valor
- Uma carta tem um estado: face pra cima ou pra baixo
- O jogador embaralha e distribui as cartas
- O baralho contém cartas

PÔQUER

Use as classes de projeto de descrição para representar as cartas, o baralho e o jogador. Então, escreva um pequeno método main(), que instancie o jogador e seu baralho, embaralhe as cartas e, em seguida, imprima o baralho.

Ao pensar nas classes, certifique-se de considerar a ocultação da implementação e a divisão da responsabilidade. Coloque a responsabilidade apenas onde ela pertence e, quando você a colocar, certifique-se de que ela não “vaze”

Carta

```java
public class Carta {
    private int valor;
    private int naipe;
    private boolean face;

    public static final int OURO = 4;
    public static final int CORACAO = 3;
    public static final int ESPADA = 6;
    public static final int PAUS = 5;
    public static final int DOIS = 2;
    public static final int TRES = 3;
    public static final int QUATRO = 4;
    public static final int CINCO = 5;
    public static final int SEIS = 6;
    public static final int SETE = 7;
    public static final int OITO = 8;
    public static final int NOVE = 9;
    public static final int DEZ = 10;
    public static final int VALETE = 74;
    public static final int DAMA = 81;
    public static final int REI = 75;
    public static final int AS = 65;

    public Carta(int valor, int naipe) {
        //necessário validar os argumentos
        this.valor = valor;
        this.naipe = naipe;
    }

    public int getNaipe() {
        return naipe;
    }
    public int getValor() {
        return valor;
    }
    public void faceCima() {
        face = true;
    }
    public void faceBaixo() {
        face = false;
    }
    public boolean estaVirada() {
        return face;
    }
    public String display() {
        String display;
        if (valor > 10) {
            display = String.valueOf((char)valor);
        } else {
            display = String.valueOf(valor);
        }
        switch(naipe) {
            case OURO: return display + String.valueOf((char)OURO);
            case CORACAO: return display + String.valueOf((char)CORACAO);
            case ESPADA: return display + String.valueOf((char)ESPADA);
            default:return display + String.valueOf((char)PAUS);
        }
    }
}
```

Baralho

```java
import java.util.LinkedList;
public class Baralho {
    private LinkedList<Carta> baralho;
    public Baralho() {
        construirCartas();
    }
    public Carta get(int indice) {
        if (indice < baralho.size()) {
            return (Carta)baralho.get(indice);
        }
        return null;
    }
    public void troca(int indice, Carta carta) {
        baralho.set(indice, carta);
    }
    public int size() {
        return baralho.size();
    }
    public Carta removaDoTopo() {
        if (baralho.size() > 0) {
            Carta carta = (Carta)baralho.removeFirst();
            return carta;
        }
        return null;
    }
    public void devolvaParaFinal(Carta carta) {
        baralho.add(carta);
    }
    private void construirCartas() {
        baralho = new LinkedList<Carta>();
        baralho.add(new Carta(Carta.PAUS, Carta.QUATRO));
        baralho.add(new Carta(Carta.CORACAO, Carta.SETE));
        baralho.add(new Carta(Carta.OURO, Carta.SETE));
        baralho.add(new Carta(Carta.ESPADA, Carta.AS));
        baralho.add(new Carta(Carta.PAUS, Carta.VALETE));
        //inserir todas as cartas
    }
}
```

Distribuidor

```java
public class Distribuidor {
    private Baralho baralho;
    public Distribuidor(Baralho b) {
        baralho = b;
    }
    public void embaralhar() {
        int numCartas, indice;
        Carta cartaI, cartaIndice;
        numCartas = baralho.size();
        for (int i = 0; i < numCartas; i++) {
            indice = (int)(Math.random() * numCartas);
            cartaI = (Carta)baralho.get(i);
            cartaIndice = (Carta)baralho.get(indice);
            baralho.troca(i, cartaIndice);
            baralho.troca(indice, cartaI);
        }
    }
    public Carta distribuir() {
        if (baralho.size() > 0) {
            return baralho.removaDoTopo();
        }
        return null;
    }
}
```

OBRIGADA