FACVLDADE UNINASSAU ser educacional

PROGRAMAÇÃO ORIENTADA A OBJETOS E ESTRUTURA DE DADOS

PROF ME. MÁRIO MEIRELES FILHO

ORIENTAÇÃO A OBJETO A EVOLUÇÃO

- Os modelos Orientados a Objetos evoluíram a partir da própria evolução das linguagens de programação;
  - Primeiras linguagens de programação
    - Semelhantes à máquina
    - Linguagens imperativas
  - Programas extremamente dependentes de hardware, e de difícil manutenção;
  - Ideia da O.O. ao invés de programar pensando como a máquina, pode-se programar pensando como humano;

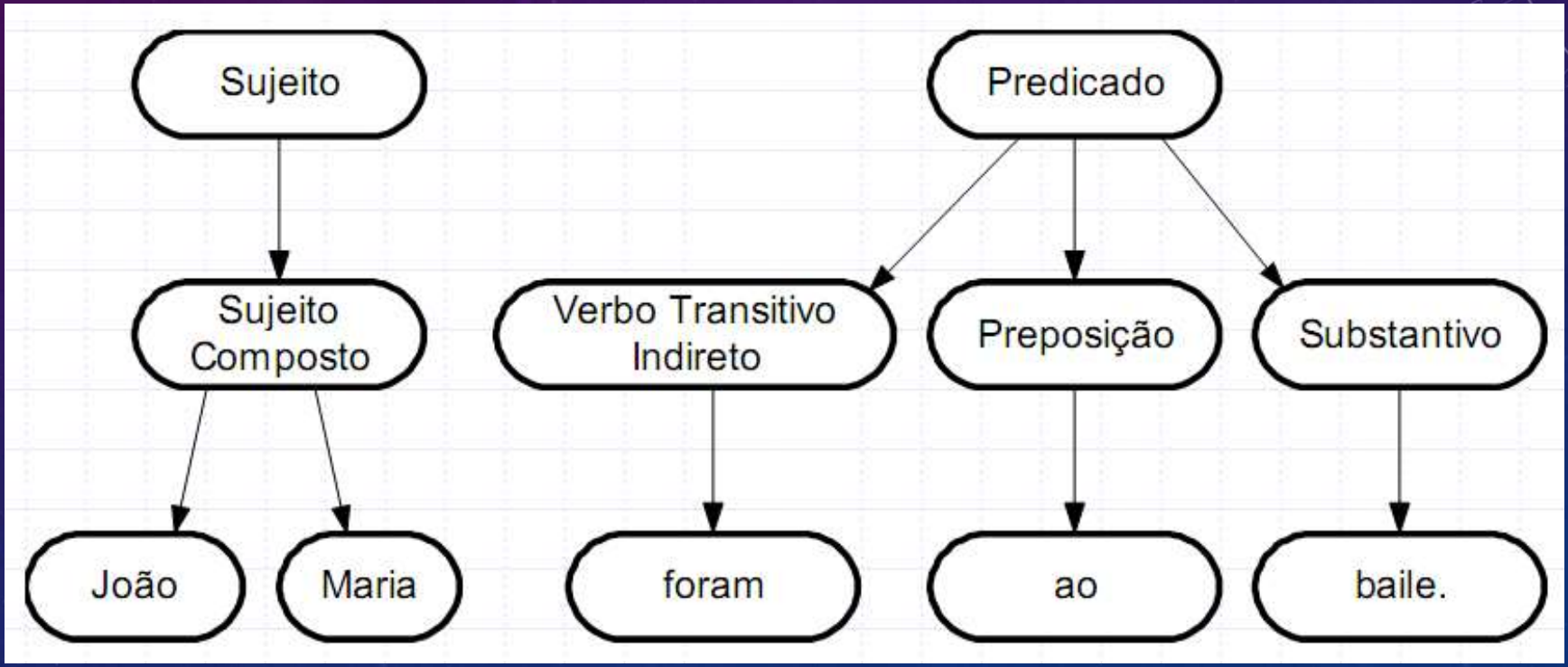
PROGRAMAÇÃO ESTRUTURADA

- Base:
  - Sequência: Uma tarefa é executada após a outra, linearmente.
  - Decisão: A partir de um teste lógico, determinado trecho de código é executado, ou não.
  - Interação: A partir de um teste lógico, determinado trecho de código é repetido por um número finito de vezes.

PROGRAMAÇÃO ESTRUTURADA

- Vantagens
  - É fácil de entender. Ainda muito usada em cursos introdutórios de programação.
  - Execução mais rápida.
- Desvantagens
  - Baixa reutilização de código
  - Códigos confusos: Dados misturados com comportamento

REGRAS DE COMUNICAÇÃO



LINGUAGEM DE PROGRAMAÇÃO

- A comunicação com o computador também é feita através de um conjunto de regras, originando a:

Linguagem de Programação!

- Exemplos: Pascal, C, C++, Delphi, Java, etc.

EXEMPLO PROGRAMA EM C

```
#include <stdio.h>
int main() {
    float num1, num2, resultado;
    printf("Digite o primeiro número: ");
    scanf("%f", &num1);
    printf("Digite o segundo número: ");
    scanf("%f", &num2);
    if (num2 == 0) {
        printf("Erro: Divisão por zero não é permitida.\n");
    } else {
        resultado = num1 / num2;
        printf("O resultado da divisão é: %.2f\n", resultado);
    }
    return 0;
}
```

DEFINIÇÃO DE PROGRAMA

- O Que é um Programa de Computador?
  - **Programas** são sequências finitas de ordens que têm o objetivo de resolver um problema, apresentar uma figura, calcular valores, tomar ou auxiliar decisões.

LÓGICA DE PROGRAMAÇÃO

- Para se programar em uma linguagem é necessário possuir **Lógica de Programação**;
- O que é Lógica de Programação ?
- **Lógica de Programação** consiste em compreender claramente os diversos passos e funções que são realizados na execução de um programa.

LÓGICA DE PROGRAMAÇÃO

- Lógica de Programação – Técnica
  - Técnica de *encadear pensamentos* para atingir determinado *objetivo*
  - Necessária para desenvolver programas e sistemas, pois permite definir a *sequência lógica* para a solução de um problema
  - SEQÜÊNCIA LÓGICA: **?** → 1. → 2. → 3. → **!**
    - Estes pensamentos podem ser descritos como uma *sequência de instruções*, que devem ser seguidas para se cumprir uma determinada tarefa
    - *Passos* executados até se atingir um objetivo ou solução de um problema

DEFINIÇÃO DE ALGORITMO

- **Algoritmo** é uma sequência de instruções organizadas de forma lógica e estruturada (sem desvios), expressas em linguagem natural (Português estruturado), que tem por finalidade resolver um problema ou descrever uma tarefa.

ALGORITMO

- Sequência finita de passos que levam à execução de uma tarefa
- Claro e preciso. Ex. “somar dois números”:
- Passos

Escrever primeiro número no retângulo A

Escrever segundo número no retângulo B

Somar o número do retângulo A com o número do retângulo B e escrever o resultado no retângulo C

ALGORITMO

- FASES para desenvolver o algoritmo:
  - Determinar o problema, defini-lo bem
  - Dividir a solução nas três fases:

- Exemplo:
  - Problema: calcular a média de quatro números
  - Dados de entrada: os números, N1, N2, N3 e N4
  - Processamento: somar os quatro números e dividir a soma por 4
  - Dados de saída: a média final

ALGORITMO

- **Importante**: abordar o máximo de detalhes em um algoritmo.

- **Não esquecer:** como será interpretado o que foi escrito?

ALGORITMO

- Algoritmo:
  - Receber o primeiro número
  - Receber o segundo número
  - Receber o terceiro número
  - Receber o quarto número
  - Somar todos os números
  - Dividir a soma por 4
  - Mostrar o resultado da divisão

VARIÁVEL

- Representa uma posição na memória, onde pode ser armazenado um dado
- Possui um nome e um valor
- Durante a execução do algoritmo, pode ter seu valor alterado (seu valor pode variar)
- Mudanças no valor das variáveis:
  - Por entrada de dados (“Ler N1”)
  - Por atribuição (“MEDIA = <um certo valor>”)

VARIÁVEL

- Exemplo SEQÜENCIAL:
  "Calcular a média de quatro números"
  - PSEUDOCÓDIGO:
    - Ler N1
    - Ler N2
    - Ler N3
    - Ler N4
    - MEDIA = (N1+N2+N3+N4) / 4
    - Mostrar MEDIA

VARIÁVEIS:
mais clareza no pseudocódigo

VARIÁVEL

SE ... ENTÃO ...

- Formato:
- Se <condição> então <ações>
- Significado: Se a <condição> resultar em verdadeiro, então executar as <ações>. Senão, simplesmente ignorar as <ações> e seguir para a próxima instrução no algoritmo.
- Usada para decidir se um conjunto de ações opcionais deve ser executado ou não, dependendo do valor de algum dado ou de algum resultado que já tenha sido calculado no algoritmo.

SE ... ENTÃO ...

- Exemplo da estrutura SE...ENTÃO:
  “Avisar se um número lido é negativo.”
- PSEUDOCÓDIGO:
- Ler N
- Se (N < 0)
- então exibir “É negativo!”
- FLUXOGRAMA:

CONDIÇÃO:

Poderá ser V ou F, depen-dendo do valor de N, que foi lido antes.

SE ... ENTÃO ...SENÃO...

Formato:

Se <condição> **então** <ações 1>
senão <ações 2>

- Significado: Se a <condição> resultar em verdadeiro, então executar <ações 1>. Senão, ignorar <ações 1> e executar <ações 2>.
- Usada para decidir entre *duas alternativas de ações*.
- *Um* dos dois conjuntos de ações será executado e *o outro não*, dependendo do valor de algum dado ou de algum resultado que já tenha sido calculado no algoritmo.
- O valor do dado ou do resultado anterior será testado na condição, determinando qual conjunto de ações será executado.

ESTRUTURAS DE REPETIÇÃO

- São muito comuns as situações em que se deseja repetir um determinado trecho de um programa um certo número de vezes.
- As estruturas de repetição são muitas vezes chamadas de Laços ou também de Loops.
- Classificação:
  - **Laços Contados**
    - Conhecimento previo de quantas vezes o comando no interior da construção será executado;
  - **Laços Condicionais**
    - Não se conhece de antemão o número de vezes que o conjunto de comandos no interior do laço será repetido.
    - Amarrado a uma condição sujeita à modificação pelas instruções do interior do laço.

LAÇOS CONDICIONAIS

- O conjunto de comandos em seu interior é executado até que uma determinada condição seja satisfeita.
- Laços condicionais mais comuns nas linguagens de programação modernas:
  - **Enquanto** - laço condicional com teste no início
  - **Repita** - laço condicional com teste no final
- A variável que é testada dever sempre estar associada a um comando que a atualize no interior do laço.

LAÇOS CONDICIONAIS COM TESTE NO INÍCIO (ENQUANTO ... FACA)

- Caracteriza-se por uma estrutura que efetua um teste lógico no início de um laço, verificando se é permitido ou não executar o conjunto de comandos no interior do laço.

```
enquanto <condição> faca
  <comando_composto>
fimenquanto
```

LAÇOS CONDICIONAIS COM TESTE NO FINAL (REPITA ... ATE)

- Efetua um teste lógico no final de um laço, verificando se é permitido ou não executar novamente o conjunto de comandos no interior do mesmo.
- Na construção **Repita** o comando é executado uma ou mais vezes (pelo menos uma vez). Além disso, a variável pode ser inicializada ou lida dentro do laço.
- Na construção **Enquanto** o comando é executado zero ou mais vezes.

```
repita
  <comando_composto>
ate <condição>
```

LAÇOS CONTADOS

- São úteis quando se conhece previamente o número exato de vezes que se deseja executar um determinado conjunto de comandos.
- Estrutura dotada de mecanismos para contar o número de vezes que o corpo do laço é executado.

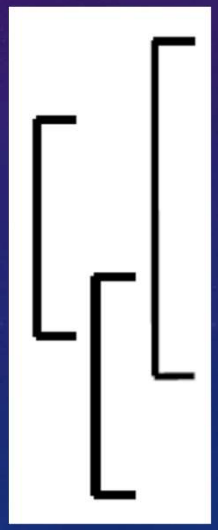
```
para <variável> de <início> ate <final> faca
   <comando_composto>
fimpara
```

ESTRUTURAS DE CONTROLE ENCADEADAS OU ANINHADAS

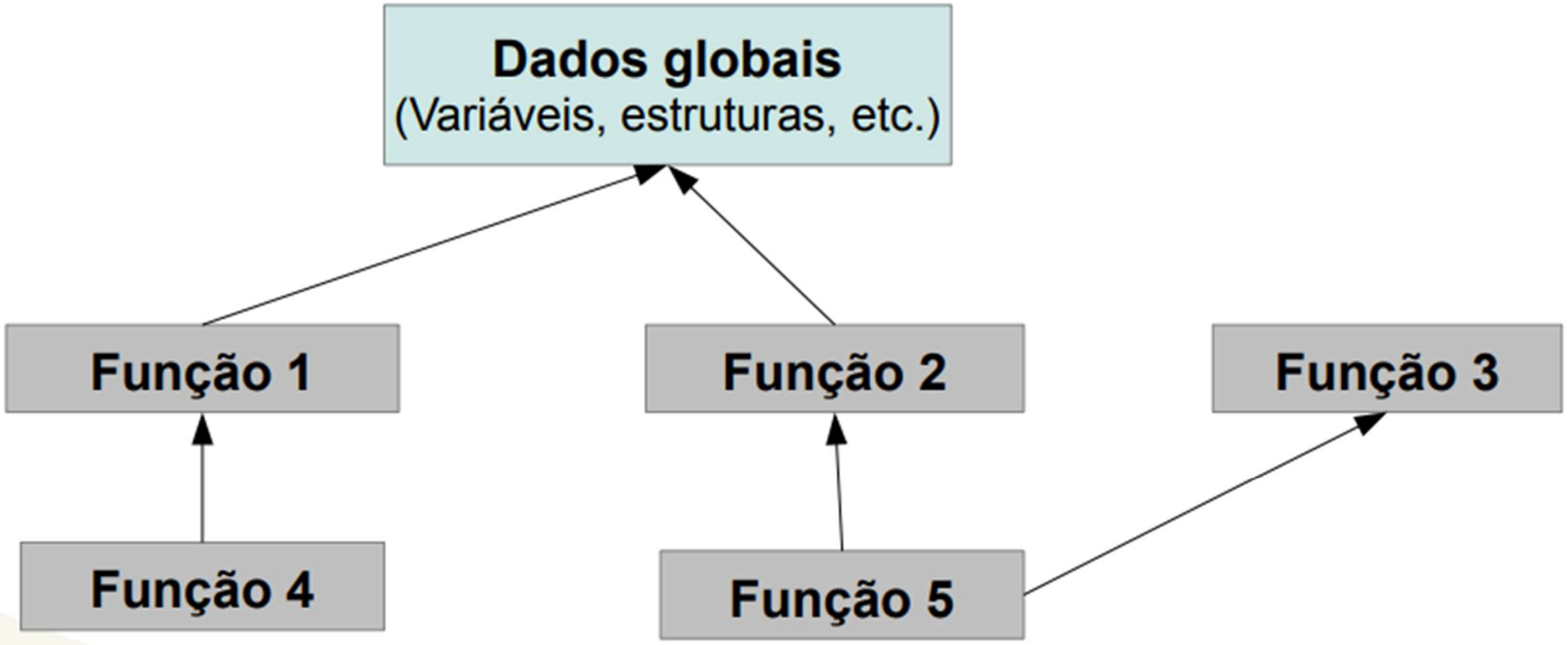
- Um aninhamento ou encadeamento é o fato de se ter qualquer um dos tipos de construção apresentados anteriormente dentro do conjunto de comandos (comando composto) de uma outra construção.
- Em qualquer tipo de aninhamento é necessário que a construção interna esteja completamente embutida na construção externa.

ESTRUTURAS DE CONTROLE ENCADEADAS OU ANINHADAS

Aninhamento válido



Aninhamento inválido


Dados globais
(Variáveis, estruturas, etc.)
Função 1
Função 2
Função 3
Função 4
Função 5

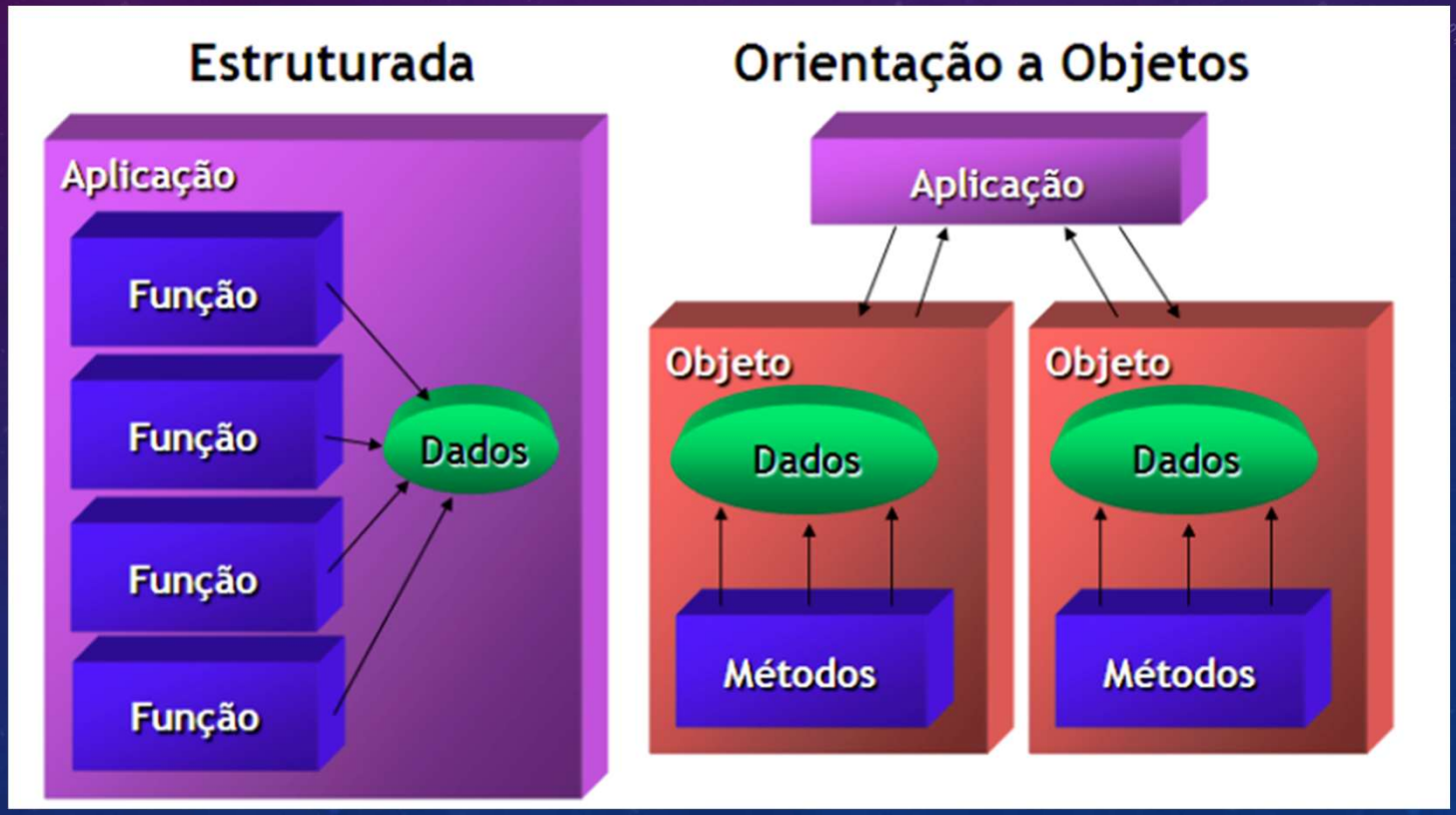
ORIENTAÇÃO A OBJETO A EVOLUÇÃO

- Base
  - Classes e Objetos
  - Métodos e Atributos
- Vantagens
  - Melhor organização do código
  - Bom reaproveitamento de código
- Desvantagens ◦
  - Desempenho mais baixo que o paradigma estruturado
  - Mais difícil compreensão

ORIENTAÇÃO A OBJETO A EVOLUÇÃO

- A Programação Orientada a Objetos (POO) é um paradigma de programação baseado no conceito de 'objetos', que podem conter dados e código: dados na forma de campos (atributos ou propriedades) e código na forma de procedimentos (métodos).

PROGRAMAÇÃO ORIENTADA A OBJETO X PROGRAMAÇÃO ESTRUTURADA



ORIGEM

- Nos anos 70 surge Smalltalk, a primeira linguagem totalmente em Orientação a Objeto (O.O)
- C++, evolução de C, já possuía conceitos O.O
- Na década de 80 praticamente todas as linguagens já usavam conceitos O.O
  - Delphi
  - PASCAL
  - Java

LINGUAGENS TÍPICAS ORIENTADAS A OBJETOS

- Linguagens mais populares no mercado de trabalho e na academia:
- - Java: Amplamente utilizada em aplicações corporativas.
- - Python: Popular em ciência de dados e inteligência artificial.
- - C++: Usada em sistemas embarcados e jogos de alta performance.

TABELA COMPARATIVA: POO VS. PROGRAMAÇÃO PROCEDURAL

Característica	Programação Orientada a Objetos (POO)	Programação Procedural
Foco	Objetos (instâncias de classes) com atributos e comportamentos	Procedimentos (funções) que manipulam dados
Organização do código	Classes e objetos	Funções e módulos
Reutilização de código	Alta, através de herança e polimorfismo	Moderada, através de funções e bibliotecas
Abstração	Forte, através de classes abstratas e interfaces	Fraca, foca em detalhes de implementação
Encapsulamento	Alto, protegendo os dados internos dos objetos	Baixo, dados são frequentemente acessíveis globalmente

TABELA COMPARATIVA: POO VS. PROGRAMAÇÃO PROCEDURAL

Característica	Programação Orientada a Objetos (POO)	Programação Procedural
Herança	Permite a criação de novas classes a partir de classes existentes	Não suporta o conceito de herança
Polimorfismo	Permite que objetos de diferentes classes sejam tratados como se fossem de uma mesma classe	Não suporta o conceito de polimorfismo
Complexidade de projetos grandes	Melhor gerenciabilidade e manutenção	Pode se tornar complexo e difícil de manter em projetos grandes
Exemplo	Um programa de simulação de um sistema bancário, onde cada conta bancária é um objeto com seus próprios atributos e métodos	Um programa que calcula a média de um conjunto de números, onde a lógica é dividida em funções

PROGRAMAÇÃO ORIENTADA A OBJETOS

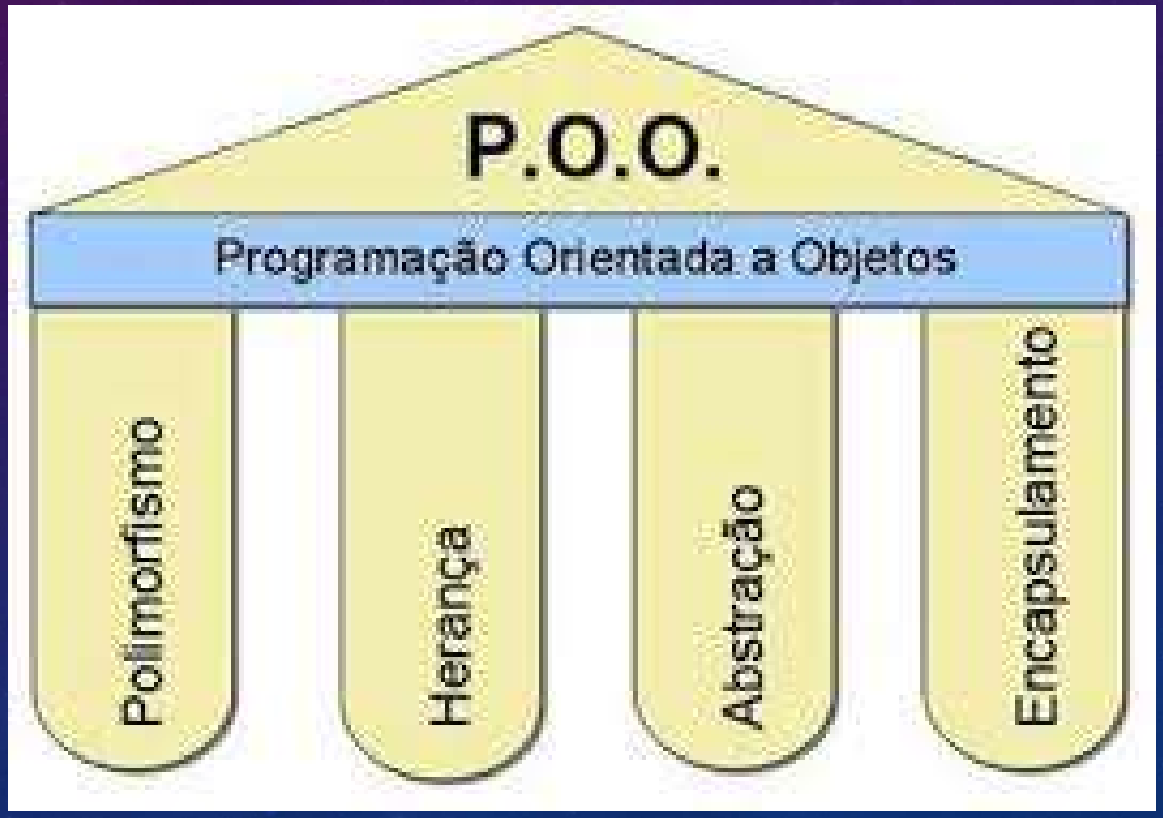
- Conceitos
  - “Uma nova maneira de pensar os problemas utilizando conceitos do Mundo Real. O componente fundamental é o objeto que combina estrutura e comportamento em uma única entidade”

    (Raumbaugh)
  - “Sistema orientado a objetos é uma coleção de objetos que interagem entre si”

    (Bertrand Meyer)
  -

PROGRAMAÇÃO ORIENTADA A OBJETO

- A POO é fundamentada em quatro pilares:
  1. Abstração: Ocultar a complexidade e mostrar a interface essencial.
  2. Encapsulamento: Esconder o estado interno e exigir todas as interações através de métodos.
  3. Herança: Permite que novas classes herdem atributos e métodos de uma classe existente.
  4. Polimorfismo: Capacidade de tratar objetos de diferentes classes de forma uniforme.


P.O.O.
Programação Orientada a Objetos
Polimorfismo
Herança
Abstração
Encapsulamento

OBJETO

- O que é:
  - Representação computacional de algo do mundo real
    - Concreto
    - Abstrato
- Abstração Transformar aquilo que observamos realidade para a virtualidade

OBJETO

- Concretos
  - Cão
  - Moto
  - Casa
- Abstratos
  - Música
  - Transação Bancária
- Modelo
  - Características + Comportamento

OBJETO

- Estado
  - Atributos (Características)
- Operações
  - Métodos (Comportamentos)
- Identidade
  - Dois objetos com estado e operações precisamente idênticos não são iguais
- Operações podem mudar os valores dos atributos assim mudando o estado de um objeto.

MÉTODOS E ATRIBUTOS

- Atributos
  - Raça: Poodle
  - Nome: Rex
  - Peso: 5 quilos
  - Potência: 500cc
  - Modelo: Honda
  - Ano: 1998
- Método
  - Latir
  - Comer
  - Dormir
  - Acelerar
  - Frear
  - Abastecer

EXERCÍCIO

- 1) Cite 4 atributos de um aluno;
- 2) Cite 3 métodos de um aluno;

CLASSES

- Conjunto de objetos:
- Características semelhantes
- Comportamento comum
- Interação com outros objetos
  - Uma classe é a forma para criação de objetos
  - Objetos são representações concretas (instâncias) de uma classe.

CLASSES

Gato

New

Gato
Raça: Savannah
Nome: Gatuno
Peso: 2,5 quilos
Idade: 2 anos

New

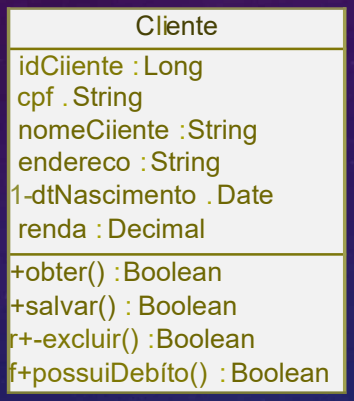
Gato
Raça: Maine Moon
Nome: Listrado
Peso: 3 quilos
Idade: 5 anos

New

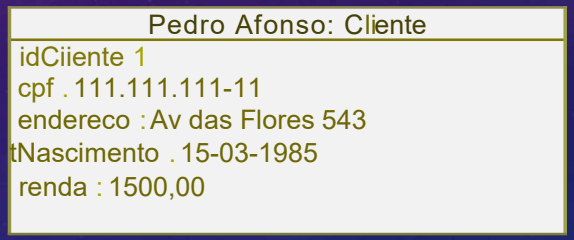
Gato
Raça: Siamês
Nome: Bichano
Peso: 4 quilos
Idade: 3 anos

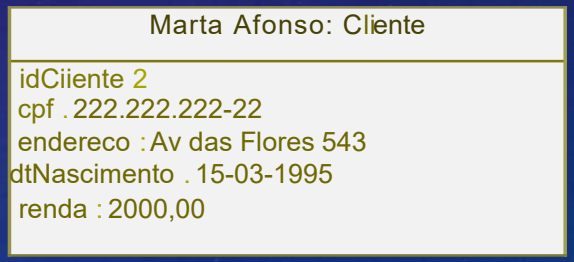
CLASSES X OBJETOS

Classe



Instâncias de um objeto





EXERCÍCIOS

- Quais as classes de um banco?

CLASSES EM JAVA


```
public class Aluno {

    private String matricula;
    private String nome;

     public void setMatricula(String matricula) {
         this.matricula = matricula;
     }
     public String getMatricula() {
          return matricula;
     }
     public void setNome(String nome) {
           this.nome = nome;
     }
     public String getNome() {
           return nome;
     }

}
```

ENCAPSULAMENTO

- O encapsulamento é um dos pilares da Programação Orientada a Objetos (POO). Ele refere-se à prática de esconder os detalhes internos de um objeto e expor apenas o que é necessário para o funcionamento do programa. Isso ajuda a proteger o estado interno do objeto e a reduzir a complexidade do sistema.

ENCAPSULAMENTO

- Um objeto, em um programa, “encapsula” todo o seu estado e o comportamento;
- Os dados e as operações são agrupados e a sua implementação é escondida, protegida dos usuários;

O QUE É ENCAPSULAMENTO?

- Encapsulamento é o princípio que garante que o estado interno de um objeto seja acessado e modificado apenas através de métodos definidos (interfaces públicas), enquanto o estado interno em si é protegido contra acessos diretos.

BENEFÍCIOS DO ENCAPSULAMENTO

- Segurança: Protege dados e evita que o estado interno do objeto seja alterado de maneira inadequada.
- Manutenção: Facilita a manutenção e a modificação do código sem afetar outras partes do sistema.
- Abstração: Permite que você trabalhe com objetos sem precisar entender a implementação interna.

COMO FUNCIONA O ENCAPSULAMENTO?

- **Atributos e Métodos:**
- **Atributos:** São as variáveis de um objeto que armazenam o estado.
- **Métodos:** São funções definidas em uma classe que operam sobre os atributos do objeto.

MODIFICADORES DE ACESSO

- Privado (private): Atributos e métodos que são acessíveis apenas dentro da própria classe.
- Público (public): Atributos e métodos que podem ser acessados de fora da classe.
- Protegido (protected): Atributos e métodos que podem ser acessados pela própria classe e por classes derivadas.

MODIFICADORES DE ACESSO

- O principal benefício de usar modificadores de acesso na programação orientada a objetos é controlar o nível de visibilidade e acesso aos atributos e métodos de uma classe, garantindo a segurança e integridade dos dados.

MODIFICADORES DE ACESSO

- Ao utilizar modificadores como private, protected e public, o desenvolvedor pode:
  - Proteger os dados sensíveis de acessos e modificações não autorizados.
  - Enforco no encapsulamento, permitindo que os atributos e métodos sejam acessados apenas de forma controlada por meio de interfaces públicas (como os métodos getter e setter).
  - Facilitar a manutenção do código, já que a implementação interna da classe pode ser alterada sem afetar outras partes do sistema que dependem dela.

  Obs.: Isso promove uma melhor organização e modularidade do código, além de aumentar a segurança e a flexibilidade.

O QUE SÃO OS MÉTODOS GETTER E SETTER?

- Getter: Método usado para acessar o valor de um atributo privado.
- Setter: Método usado para modificar o valor de um atributo privado, frequentemente com validação para garantir integridade.

BENEFÍCIOS DOS MÉTODOS GETTER E SETTER

- Segurança de dados: Impedem que atributos sejam acessados ou modificados de forma inadequada.
- Abstração: Os detalhes internos da classe são escondidos, permitindo que o comportamento possa ser alterado sem impacto no restante do código.
- Facilidade de manutenção: Mudanças na implementação não afetam as classes que usam os métodos de acesso.

MÉTODOS GETTER E SETTER

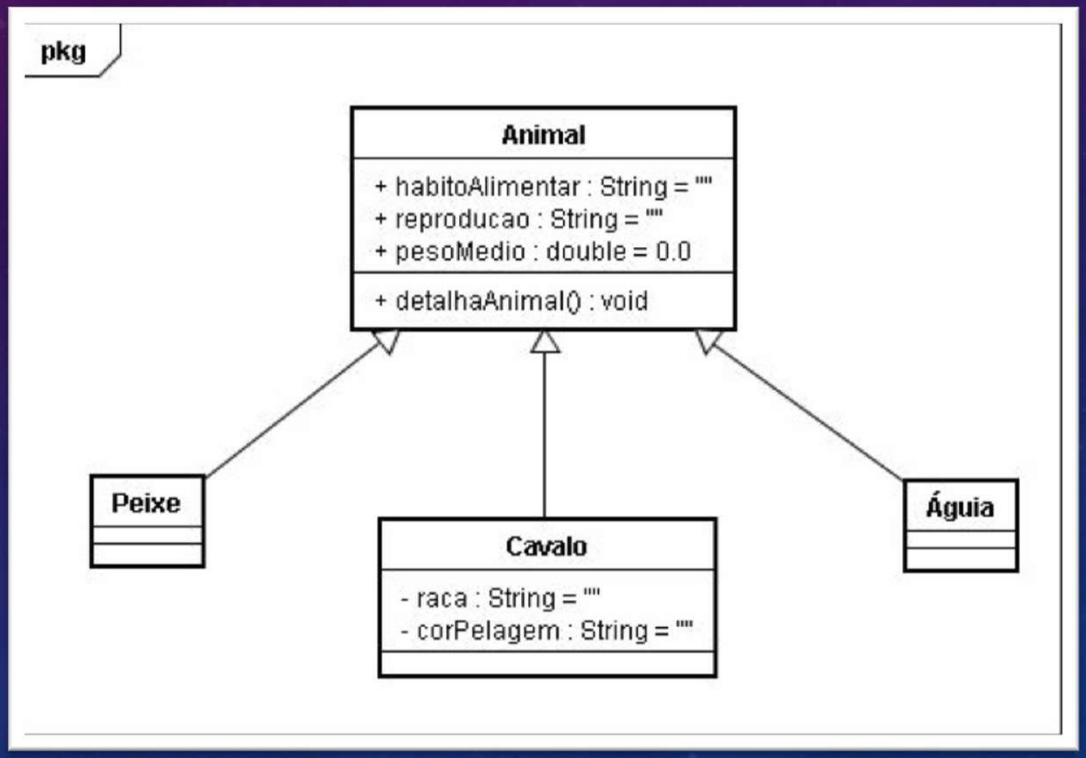
```java
public class ContaBancaria {
    private double saldo;

    // Getter para acessar o saldo
    public double getSaldo() {
        return saldo;
    }

    // Setter para modificar o saldo, com validação
    public void setSaldo(double saldo) {
        if (saldo >= 0) {
            this.saldo = saldo;
        } else {
            System.out.println("Saldo inválido.");
        }
    }
}
```

```
public class ContaBancaria {
    // Atributos privados
    private double saldo;
    private String titular;

    // Construtor
    public ContaBancaria(String titular, double saldoInicial) {
        this.titular = titular;
        this.saldo = saldoInicial;
    }

    // Método público para depósito
    public void depositar(double valor) {
        if (valor > 0) {
            saldo += valor;
        }
    }

    // Método público para saque
    public void sacar(double valor) {
        if (valor > 0 && valor <= saldo) {
            saldo -= valor;
        }
    }

    // Método público para consulta de saldo
    public double getSaldo() {
        return saldo;
    }

    // Método público para consulta de titular
    public String getTitular() {
        return titular;
    }
}
```

HERANÇA

- O que é Herança?
  - Mecanismo da POO que permite criar novas classes a partir de classes já existentes.
  - Permite que uma classe (chamada de **subclasse** ou **classe derivada**) herde atributos e métodos de outra classe (chamada de **superclasse** ou **classe base**)

HERANÇA


pkg
Animal
+ habitoAlimentar : String = ""
+ reproducao : String = ""
+ pesoMedio : double = 0.0
+ detalhaAnimal() : void
Peixe
Cavalo
- raca : String = ""
- corPelagem : String = ""
Águia

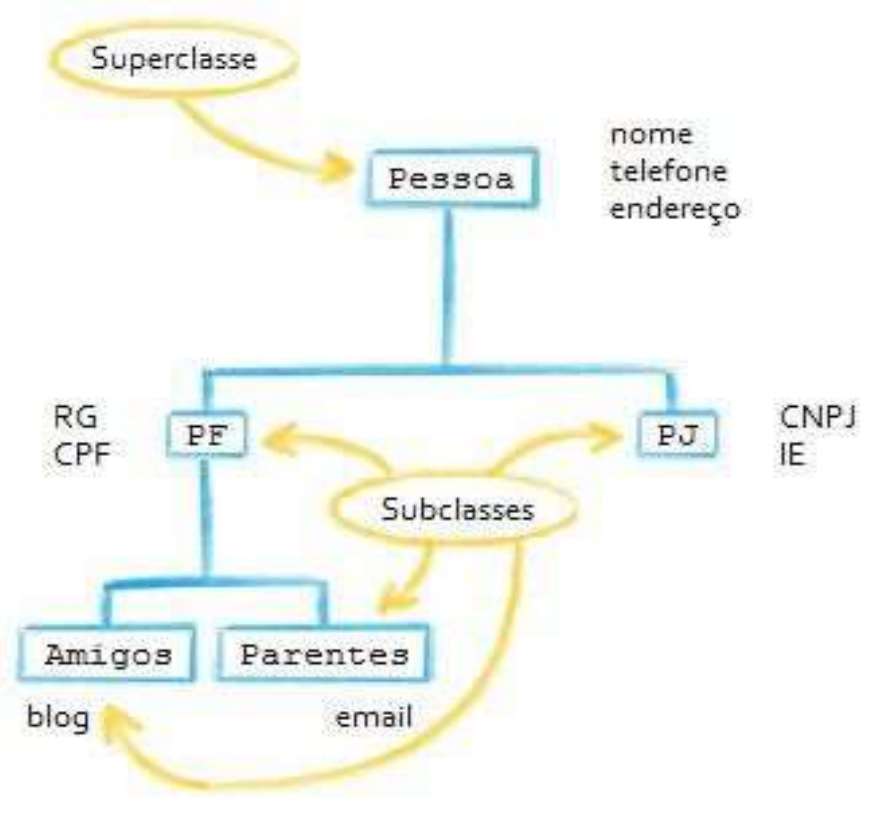
POR QUE UTILIZAR HERANÇA?

- Reutilização de código: Evita a duplicação de código.
- Organização do código: Cria uma hierarquia de classes mais clara e intuitiva.
- Extensibilidade: Permite criar novas classes especializadas a partir de classes mais genéricas.

HERANÇA - CONCEITOS-CHAVE:

- Superclasse (classe base): Classe mais geral, que fornece atributos e métodos comuns.
- Subclasse (classe derivada): Classe mais específica, que herda os atributos e métodos da superclasse e pode adicionar novos ou sobrescrever os existentes.
- Generalização e Especialização: A superclasse representa uma generalização, enquanto as subclasses representam especializações.

HERANÇA


Superclasse
Pessoa
nome
telefone
endereço
RG
CPF
PF
PJ
CNPJ
IE
Subclasses
Amigos
Parentes
blog
email

```java
public class Pessoa {
    private String nome;
    private String endereco;

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
    public String getDocumento() {
        return "";
    }
}
```

```java
public class Pessoa_ficisca extends Pessoa {
    private String cpf;

    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    @Override
    public String getDocumento() {
        return cpf;
    }
}
```

```java
public class Pessoa juridica extends Pessoa {
    private String cnpj;

    public String getCnpj() {
        return cnpj;
    }
    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }
    @Override
    public String getDocumento() {
        return cnpj;
    }
}
```

POLIMORFISMO

- O que é Polimorfismo?
  - Refere-se à capacidade de uma classe de fornecer diferentes implementações para métodos herdados ou compartilhados por várias classes.

POLIMORFISMO

- **Tipos de Polimorfismo:**
  1. **Polimorfismo em Tempo de Compilação (Sobrecarga de Métodos)**
  2. **Polimorfismo em Tempo de Execução (Sobrescrita de Métodos)**

POLIMORFISMO EM TEMPO DE COMPILAÇÃO (SOBRECARGA DE MÉTODOS)

- Ocorre quando múltiplos métodos na mesma classe têm o mesmo nome, mas diferentes parâmetros.
- Não depende de herança.
- Isso é :
  - Ocorre na mesma classe.
  - Os métodos devem ter assinaturas diferentes (quantidade ou tipos de parâmetros diferentes).
  - Não depende de herança ou interfaces.
  - Pode alterar o tipo de retorno, mas isso por si só não diferencia métodos sobrecarregados.

POLIMORFISMO EM TEMPO DE COMPILAÇÃO (SOBRECARGA DE MÉTODOS)

```
class Calculadora {
    // Sobrecarga de método
    public int somar(int a, int b) {
        return a + b;
    }

    public double somar(double a, double b)
{
        return a + b;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();
        System.out.println(calc.somar(5, 3)); //
Saída: 8
        System.out.println(calc.somar(5.0, 3.0)); //
Saída: 8.0
    }
}
```

POLIMORFISMO EM TEMPO DE EXECUÇÃO (SOBRESCRITA DE MÉTODOS)

- Ocorre quando uma subclasse altera o comportamento de um método herdado da superclasse.
- Depende da herança e interfaces.
- Isso é:
  - Ocorre em classes diferentes (entre uma superclasse e sua subclasse).
  - O método sobrescrito deve ter mesma assinatura (nome, parâmetros e tipo de retorno).
  - Requer herança (ou seja, uma relação de superclasse-subclasse).
  - Usado para alterar o comportamento de um método herdado de uma classe pai.
  - O método da superclasse pode ser chamado dentro do método sobrescrito usando a palavra-chave super.

POLIMORFISMO EM TEMPO DE EXECUÇÃO (SOBRESCRITA DE MÉTODOS)

```
class Animal {
    public void fazerSom() {
        System.out.println("O animal faz um som.");
    }
}

class Cachorro extends Animal {     @Override
    public void fazerSom() {
        System.out.println("O cachorro late.");
    }
}
```

```
class Gato extends Animal {       @Override
    public void fazerSom() { System.out.println("O gato mia.");     }
}
public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Animal();
        Animal meuCachorro = new Cachorro();
        Animal meuGato = new Gato();
         meuAnimal.fazerSom(); // Saída: O animal faz um som.
        meuCachorro.fazerSom(); // Saída: O cachorro late.
        meuGato.fazerSom(); // Saída: O gato mia.
    }
}
```

DIFERENÇA ENTRE SOBRECARGA E SOBRESCRITA

- Sobrecarga (Compile Time): Múltiplos métodos com o mesmo nome na mesma classe, mas parâmetros diferentes.

- Sobrescrita (Runtime): Método da superclasse é redefinido em uma subclasse com o mesmo nome, assinatura e parâmetros.

BENEFÍCIOS DO POLIMORFISMO

- Código mais limpo e fácil de manter.
  - O uso de polimorfismo reduz a necessidade de estruturas de controle como **if** e **switch** para escolher comportamentos.
- Facilita a escalabilidade.
  - Novos tipos podem ser facilmente adicionados ao programa sem modificar o código existente.
- Suporte ao princípio do "aberto/fechado" (Open/Closed Principle).
  - As classes estão abertas para extensão, mas fechadas para modificação.

POLIMORFISMO COM INTERFACES

- As interfaces permitem implementar o polimorfismo de forma mais flexível, pois uma classe pode implementar várias interfaces.
  - Ele permite que diferentes classes compartilhem o mesmo comportamento, mesmo quando não estão relacionadas por herança direta
  - uma classe pode implementar várias interfaces, mas herdar de apenas uma classe

POLIMORFISMO COM INTERFACES

- Interface: é um contrato que define um conjunto de métodos que devem ser implementados pelas classes que a adotam.
- Polimorfismo: A capacidade de um objeto de ser referenciado por diferentes tipos, sendo o comportamento definido em tempo de execução.
  - Obs: Quando usamos interfaces, podemos ter várias classes implementando a mesma interface e, portanto, diferentes tipos de objetos podem ser tratados de maneira uniforme, mesmo que não compartilhem uma relação de herança direta

VANTAGENS DO POLIMORFISMO COM INTERFACES

- Flexibilidade: Uma classe pode implementar várias interfaces, o que permite compartilhar comportamento entre classes não relacionadas.
- Desacoplamento: Permite desacoplar o código, pois você pode referenciar objetos por meio de suas interfaces em vez de suas classes concretas. Isso facilita a manutenção e extensão do código.
- Substituibilidade: Qualquer classe que implementa a interface pode ser usada onde quer que a interface seja esperada, promovendo a substituição de implementações de forma simples e eficiente.

POLIMORFISMO COM INTERFACES

```
interface Forma {
      void desenhar();
}
class Circulo implements Forma {
    @Override
       public void desenhar()  {
              System.out.println("Desenhando um círculo.");    }
}
```

POLIMORFISMO COM INTERFACES

```
class Quadrado implements Forma {
    @Override
    public void desenhar() {
        System.out.println("Desenhando um quadrado.");
    }
}
```

POLIMORFISMO COM INTERFACES

```
public class Main {
    public static void main(String[] args) {
        Forma forma1 = new Circulo();
        Forma forma2 = new Quadrado();
        forma1.desenhar(); // Saída: Desenhando um círculo.
        forma2.desenhar(); // Saída: Desenhando um quadrado.
    }
}
```

POLIMORFISMO

Característica	Sobrecarga de Métodos	Sobrescrita de Métodos	Interfaces
Ocorrência	Mesma classe	Subclasse redefine método da superclasse	Classes diferentes implementam a interface
Mudança na assinatura	Diferente (número ou tipo de parâmetros)	Mesma assinatura	Definida pela interface
Herança	Não requer herança	Requer herança	Não requer herança direta
Polimorfismo	Tempo de compilação	Tempo de execução	Tempo de execução
Objetivo	Prover várias versões de um método	Alterar comportamento herdado	Compartilhar comportamento entre classes