

Analizar
programação
estruturada e
orientada a objetos

SEENAC:

MOVIMENTE

O AGORA



Analisar programação estruturada e orientada a objetos



Docente: Rodrigo Neves Ottoboni Dias

Turma: Técnico em Desenvolvimento de Sistemas

Apresentação

- UC4: Analisar programação estruturada e orientada a objetos
- Carga Horária: 48 Horas (12 aulas)
- Conhecimentos a serem desenvolvidos
 - a. Introdução aos conceitos de programação estruturada e orientada a objetos.
 - b. Princípio básico da programação estruturada (sequência, seleção e iteração).
 - c. Pilares da programação orientada a objetos (Abstração, Encapsulamento, Herança, Polimorfismo).
 - d. Comparação entre os Paradigmas.

Semana 1

- Programação estruturada: conceitos e aplicabilidade.
- Introdução aos conceitos de programação estruturada e orientada a objetos.
- Vantagens e desvantagens da programação estruturada.
- Exemplo Prático de programação estruturada.



Existem dois tipos de programação bem usuais: Orientada a Objetos (OO) e Estruturada. Em cada uma delas iremos conhecer os seus principais conceitos e suas respectivas utilizações e, por fim, mostraremos um breve comparativo entre esses dois tipos e qual o melhor a ser utilizado quando um projeto de desenvolvimento de software for iniciado. Utilizaremos como linguagem de programação base para os nossos exemplos as linguagens Java e C#.



Senac

Conceito de Programação Estruturada

- **Paradigma que organiza o código em sequências de comandos, decisões e repetições.**
- **Enfatiza a divisão em funções/procedimentos para melhorar a organização do código.**
 - O princípio básico da programação estruturada é que um programa pode ser dividido em três partes que se interligam: **sequência, seleção e iteração.**



Conceito de Programação Estruturada

- A programação estruturada é um **paradigma de programação que se baseia na divisão de um programa em partes menores, chamadas de blocos de código, que executam tarefas específicas de forma sequencial, condicional e repetitiva**. Caracterizada pela clareza, simplicidade e organização do código, a programação estruturada torna mais fácil a **compreensão e manutenção do software**.
- Alguns exemplos de linguagens de programação que seguem o paradigma da programação estruturada são C, Pascal e Fortran. Suas vantagens incluem a facilidade de depuração de erros, a reutilização de código e a modularidade, o que facilita a manutenção e evolução do programa ao longo do tempo.



Senac

Conceito de Programação Estruturada

- A programação estruturada é amplamente utilizada em sistemas de informação, desenvolvimento de aplicativos, jogos e softwares de gestão, entre outras aplicações. Sua abordagem organizada e lógica é essencial para garantir a eficiência e robustez dos programas desenvolvidos.



Benefícios de Programação Estruturada

- Um dos grandes benefícios da programação estruturada é a facilidade de manutenção do código. Como o código é dividido em blocos mais simples e bem definidos, fica mais fácil identificar e corrigir possíveis erros. Além disso, a programação estruturada facilita a compreensão do código, tanto para o programador que está desenvolvendo o programa quanto para outros programadores que precisam dar manutenção no código.



Senac

Benefícios de Programação Estruturada

- Algumas linguagens de programação que seguem o paradigma da programação estruturada incluem o C, Pascal e Fortran. Essas linguagens são amplamente utilizadas em diferentes áreas, como desenvolvimento de sistemas, análise de dados e computação científica.
- Essas características tornam a programação estruturada uma abordagem eficiente e robusta para o desenvolvimento de software.



Principais características da programação estruturada

- Um dos princípios fundamentais da programação estruturada é a utilização de estruturas de controle bem definidas, como if, else, while e for, que permitem que o programador controle o fluxo da execução de forma clara e organizada. Além disso, a programação estruturada preza pela modularização do código, ou seja, pela divisão do programa em funções ou procedimentos que realizam tarefas específicas.



Senac

Programação Estruturada : Seleção

- Na seleção o fluxo a ser percorrido depende de uma escolha. Existem duas formas básicas para essa escolha.
- A primeira é através do condicional “Se”, onde se uma determinada condição for satisfatória o fluxo a ser corrido é um e, caso contrário, o fluxo passa a ser outro. Ou seja, se o fluxo só percorre apenas um caminho, apenas uma ação é processada.
- A outra forma de escolha é onde o número de condições se estende a serem avaliadas. Por exemplo, se a Condição 1 for verdade faça Processamento 1, caso contrário, se a Condição 2 for verdade faça Processamento 2, caso contrário, se a Condição 3 for verdade faça Processamento 3, e assim por diante.



Senac

Programação Estruturada : Iteração

- Na iteração é permito a execução de instruções de forma repetida, onde ao fim de cada execução a condição é reavaliada e enquanto seja verdadeira a execução de parte do programa continua.



Senac

Programação Estruturada : Modularização

- A medida que o sistema vai tomando proporções maiores, é mais viável que o mesmo comece a ser dividido em partes menores, onde é possível simplificar uma parte do código deixando a compreensão mais clara e simplificada. Essa técnica ficou conhecida como Subprogramação ou Modularização. No desenvolvimento utilizamos essa técnica através de procedimentos, funções, métodos, rotinas e uma série de outras estruturas. Com essa divisão do programa em partes podemos extrair algumas vantagens, como:



Programação Estruturada : Modularização

- Cada divisão possui um código mais simplificado;
- Facilita o entendimento, pois as divisões passam a ser independentes;
- Códigos menores são mais fáceis de serem modificados;
- Desenvolvimento do sistema através de uma equipe de programadores;
- Reutilização de trechos de códigos.
- Hoje as linguagens estruturadas, como Algol, Pascal e C, ainda são encontradas em muitos sistemas, apesar das linguagens Orientadas a objetos serem mais usuais.



Senac

Programação Estruturada : Exemplo

```
1 # include <stdio.h>
2
3     int main()
4
5         int soma, n=1;
6
7         soma = 0; // inicialização da variável soma
8
9         for (n=1; n<=100; n++)
10
11             soma= soma + n; // atualização da variável soma
12
13
14         printf("O valor da soma = %d\n",soma);
15
16         return 0;
17
18 }
```



Programação Estruturada : Exemplo

Problema: Calcular a média de um aluno e informar se está aprovado.

Pseudocódigo Estruturado:

plaintext

Copiar

Editar

```
início
    ler nota1, nota2
    média ← (nota1 + nota2) / 2
    se média ≥ 6 então
        escrever "Aprovado"
    senão
        escrever "Reprovado"
    fim
```



Programação Estruturada : Exemplo

Em Python (estruturado):

python

Copiar

Editar

```
def calcular_media():

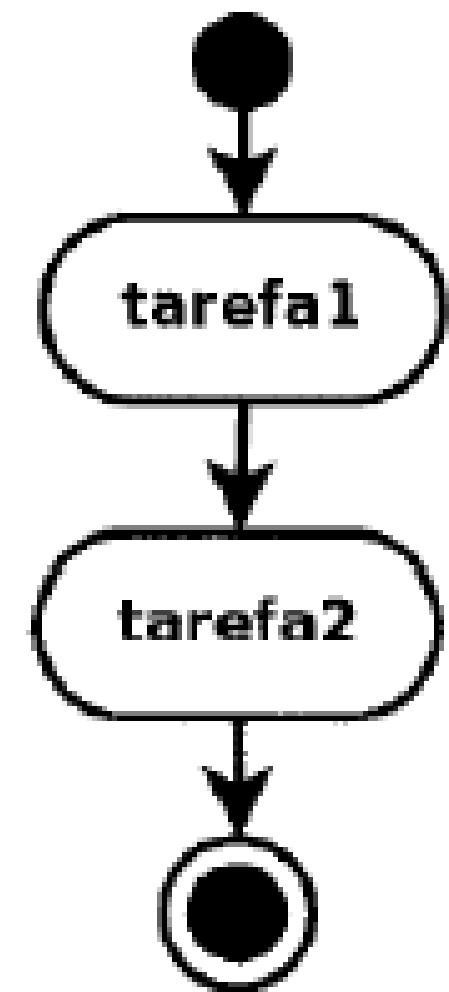
    nota1 = float(input("Digite a primeira nota: "))
    nota2 = float(input("Digite a segunda nota: "))
    media = (nota1 + nota2) / 2

    if media >= 6:
        print("Aprovado")
    else:
        print("Reprovado")

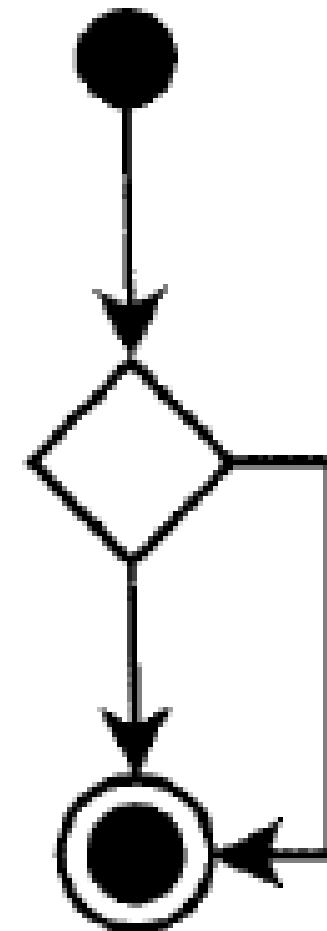
calcular_media()
```



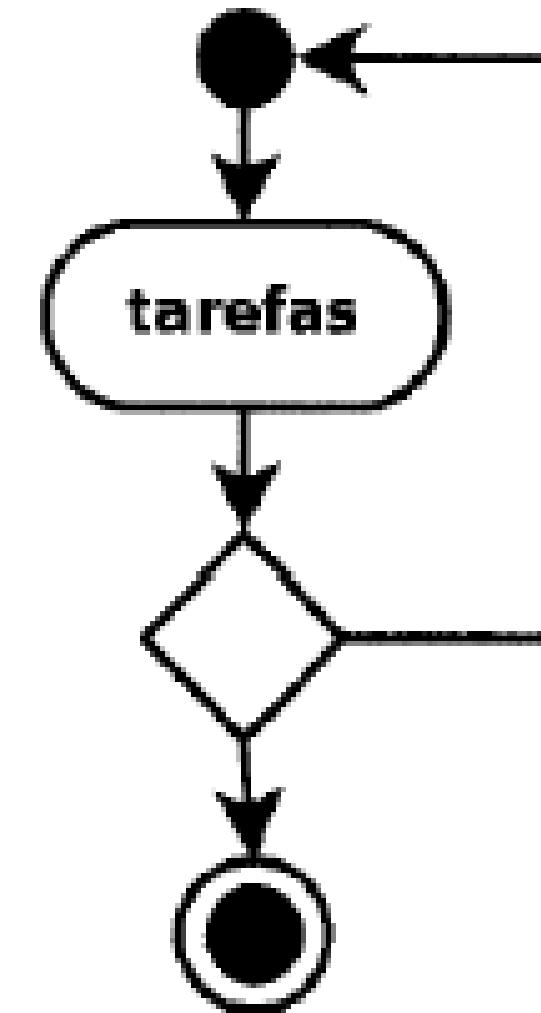
Programação Estruturada : Exemplo



SEQUÊNCIA



DECISÃO



REPETIÇÃO



Conceito de Programação Orientada a Objetos

- A programação orientada a objetos é um modelo de programação onde diversas classes possuem características que definem um objeto na vida real. Cada classe determina o comportamento do objeto definido por métodos e seus estados possíveis definidos por atributos. São exemplos de linguagens de programação orientadas a objetos: C++, Java, C#, Object Pascal, entre outras. Este modelo foi criado com o intuito de aproximar o mundo real do mundo virtual. Para dar suporte à definição de Objeto, foi criada uma estrutura chamada Classe, que reúne objetos com características em comum, descreve todos os serviços disponíveis por seus objetos e quais informações podem ser armazenadas.



Senac

Fundamentos da POO

Programação Orientada a Objetos

- A motivação do paradigma é representar cada elemento do mundo real por um objeto, sendo que um objeto pode ser definido como um conjunto de estados e comportamentos. Os estados são os atributos, ou seja, as características do objeto, e os comportamentos são os métodos ou as ações que o objeto pode apresentar.



Senac

Fundamentos da POO

Programação Orientada a Objetos

Carro

- Velocidade : double
- Modelo : string

- + Frear() : void
- + AcenderFarol() : void



Fundamentos da POO

- No diagrama UML acima, temos a classe Carro que apresenta as características (atributos) velocidade e modelo e que pode realizar as ações (métodos) frear e acender farol. Essa mesma classe pode ser implementada a nível desenvolvimento através do seguinte código:

```
public class Carro{  
    public double Velocidade { get; set; }  
    public string Modelo { get; set; }  
    public void Frear() {  
        //código do carro para frear  
    }  
    public void AcenderFarol() {  
        //código do carro para acender farol  
    }  
}
```

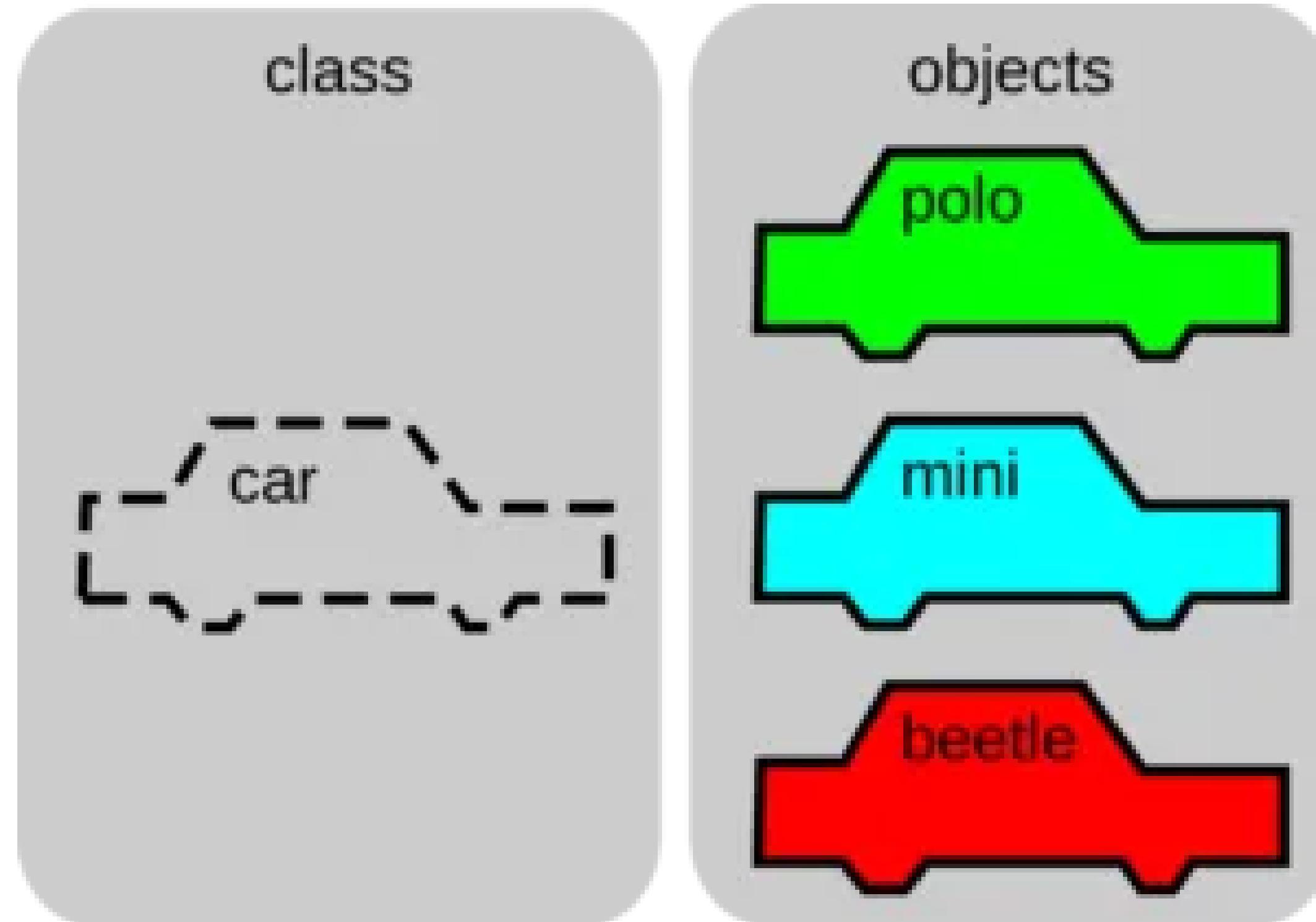
Fundamentos da POO

- A classe Carro pode vir a se tornar um objeto em tempo de execução.
Mas qual é a diferença entre uma classe e um objeto?
- As classes são a definição de uma estrutura que depois irá gerar um objeto em tempo de execução. Já os objetos, são instâncias de uma classe. Para ficar mais simples, imagine a classe como um molde que irá ditar quais características e comportamentos um objeto deve adotar, e os objetos são a formação dessas características já bem definidas. Por exemplo, para a nossa classe Carro, podemos ter o objeto Polo, o objeto Mini, ou até mesmo o objeto Beetle.



Senac

Fundamentos da POO



Exemplo:

Abaixo segue o exemplo de um código utilizado para criar uma instância da classe Carro em tempo de execução:

```
static void Main(string[] args)
{
    var carro = new Carro();
    carro.Velocidade = 200.00;
    carro.Modelo = "polo";
}
```



Exercício 1 : Objetos em JavaScript

💡 Exercício 1 – Criar um Objeto Literal

Objetivo: Entender como criar e acessar objetos em JavaScript.

javascript

Copiar Editar

```
// Criar um objeto literal
let pessoa = {
    nome: "Ana",
    idade: 25,
    saudacao: function () {
        console.log("Olá, meu nome é " + this.nome);
    }
};

// Teste no console
console.log(pessoa.nome); // "Ana"
console.log(pessoa.idade); // 25
pessoa.saudacao(); // "Olá, meu nome é Ana"
```

⭐ Tente adicionar uma nova propriedade `email` e acessar depois.



Senac

Exercício 2: Objetos em JavaScript

Exercício 2 – Criar uma Classe Simples

Objetivo: Entender a sintaxe de classe e instanciar objetos.

javascript

Copiar Editar

```
// Definição da classe
class Carro {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    exibirInfo() {
        console.log(`Carro: ${this.marca} ${this.modelo}`);
    }
}

// Criando instâncias
let carro1 = new Carro("Toyota", "Corolla");
let carro2 = new Carro("Honda", "Civic");

// Testes
carro1.exibirInfo(); // "Carro: Toyota Corolla"
carro2.exibirInfo(); // "Carro: Honda Civic"
```

⭐ Modifique para adicionar um método que diga se o carro é automático ou manual.



Senac

Exercício 3: Objetos em JavaScript

Exercício 3 – Adicionando Métodos Dinamicamente

Objetivo: Mostrar como métodos podem ser adicionados depois da criação do objeto.

```
javascript

let aluno = {
  nome: "Lucas",
  curso: "Informática"
};
```

 Copiar  Editar

```
// Adicionando método depois
aluno.mostrarCurso = function () {
  console.log(`Curso do aluno: ${this.curso}`);
};

// Teste
aluno.mostrarCurso(); // "Curso do aluno: Informática"
```



Senac

Exercício 4: Objetos em JavaScript

Exercício 4 – Classe com Lista (array) de Objetos

Objetivo: Trabalhar com múltiplos objetos usando uma classe.

javascript

Copiar Editar

```
class Produto {
    constructor(nome, preco) {
        this.nome = nome;
        this.preco = preco;
    }

    exibir() {
        console.log(`#${this.nome} custa R$ ${this.preco}`);
    }
}

// Lista de produtos
let listaProdutos = [
    new Produto("Caderno", 15),
    new Produto("Lápis", 2),
    new Produto("Borracha", 1.5)
];

// Mostrar todos
listaProdutos.forEach(p => p.exibir());
```



Senac

Exemplo: Objetos em JavaScript

```
JavaScript

// Definição de uma função construtora
function Pessoa(nome, idade) {
    this.nome = nome;
    this.idade = idade;
}

// Adição de um método ao protótipo
Pessoa.prototype.saudacao = function() {
    return `Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`;
};

// Criação de instâncias da classe Pessoa
let pessoa1 = new Pessoa("João", 30);
let pessoa2 = new Pessoa("Maria", 25);

// Acesso a propriedades e métodos
console.log(pessoa1.nome); // Output: João
console.log(pessoa2.idade); // Output: 25
console.log(pessoa1.saudacao()); // Output: Olá, meu nome é João e tenho 30 anos

// Herança de protótipo
Pessoa.prototype.profissao = "desenvolvedor"; // Adiciona uma propriedade ao protótipo

console.log(pessoa1.profissao); // Output: desenvolvedor
console.log(pessoa2.profissao); // Output: desenvolvedor
```

Exemplo: Objetos em JavaScript

```
JS

var pessoa = {
    nome: ["Bob", "Smith"],
    idade: 32,
    sexo: "masculino",
    interesses: ["música", "esquiar"],
    bio: function () {
        alert(
            this.nome[0] +
            " " +
            this.nome[1] +
            " tem " +
            this.idade +
            " anos de idade. Ele gosta de " +
            this.interesses[0] +
            " e " +
            this.interesses[1] +
            ".",
        );
    },
    saudacao: function () {
        alert("Olá! Eu sou " + this.nome[0] + ".");
    },
};
```



Exemplo: Objetos em JavaScript

```
JS

var pessoa = {
    nome: ["Bob", "Smith"],
    idade: 32,
    sexo: "masculino",
    interesses: ["música", "esquiar"],
    bio: function () {
        alert(
            this.nome[0] +
            " " +
            this.nome[1] +
            " tem " +
            this.idade +
            " anos de idade. Ele gosta de " +
            this.interesses[0] +
            " e " +
            this.interesses[1] +
            ".",
        );
    },
    saudacao: function () {
        alert("Olá! Eu sou " + this.nome[0] + ".");
    },
};
```



Exemplo: Objetos em JavaScript

JS

```
pessoa.nome;  
pessoa.nome[0];  
pessoa.idade;  
pessoa.interesses[1];  
pessoa.bio();  
pessoa.saudacao();
```



Pilares da programação Orientada a Objetos

Abstração

- Determina características do objeto como identidade, propriedade e métodos do objeto

Encapsulamento

- Confere segurança à aplicação em uma programação orientada a objetos, pois deixa as propriedades difíceis de serem alcançadas.

Herança

- Os objetos filhos herdam as características e ações de seus ancestrais, otimizando a produção da aplicação em tempo e linhas de código.

Polimorfismo

- faz a alteração do funcionamento interno de um método herdado, atribuindo uma nova implementação para o método pré-definido.



Programação Orientada a Objetos : Abstração

- Este princípio é uma forma de abstrair o quanto complexo é um método ou rotina de um sistema, ou seja, o usuário não necessita saber todos os detalhes de como sua implementação foi realizada, apenas para que serve determinada rotina e qual o resultado esperado da mesma. Sendo assim, podemos também dividir internamente problemas complexos em problemas menores, onde resolvemos cada um deles até encontrarmos a solução do problema inteiro. Um exemplo da vida real para ilustrar esse conceito seria o conceito de carro a abstração de um veículo, que é utilizado como meio de transporte por várias pessoas para mover-se de um ponto a outro. Não é necessário que a pessoa informe que irá se locomover com a ajuda de um veículo movido a combustível, contendo rodas e motor. Basta a pessoa informar que utilizará um carro para tal, pois esse objeto é conhecido por todos e abstrai toda essa informação por trás disso.



Programação Orientada a Objetos : Abstração

Abstração foca em esconder a complexidade e mostrar apenas o essencial.

```
JS teste.js > ...
1  console.log();class Carro {
2    constructor(modelo) {
3      this.modelo = modelo;
4      this.ligado = false;
5    }
6
7    ligar() {
8      this.ligado = true;
9      console.log(`#${this.modelo} está ligado.`);
10   }
11
12   dirigir() {
13     if (this.ligado) {
14       console.log(`#${this.modelo} está em movimento.`);
15     } else {
16       console.log(`Ligue o carro primeiro.`);
17     }
18   }
19 }
20
21 const carro = new Carro("Fusca");
22 carro.dirigir(); // Ligue o carro primeiro.
23 carro.ligar();
24 carro.dirigir(); // Fusca está em movimento.
25 |
```



Programação Orientada a Objetos : Abstração

```
1 public class Conta {  
2     int cod_banco;  
3     int num_conta;  
4     double saldo;  
5     double limite;  
6  
7     void ConsultarSaldo() {  
8         System.out.println("Conta: " + this.num_conta);  
9         System.out.println("Saldo: " + this.saldo);  
10    }  
11  
12    void Depositar(double valor) {  
13        this.saldo = this.saldo + valor;  
14    }  
15  
16    void Sacar(double valor) {  
17        this.saldo = this.saldo - valor;  
18    }  
19}
```



Programação Orientada a Objetos : Abstração

- Para este caso, um cliente só precisa entender que uma conta é um local, em um determinado banco, onde é possível ser depositado e sacado valores. Para exemplificar este caso, criamos uma classe Conta com os atributos: código do banco, número da conta, saldo e limite. Criamos também um método ConsultarSaldo, onde ele retorna qual o saldo da conta naquele momento. Criamos também outro método chamado Depositar onde passamos um valor como parâmetro e ele soma esse ao saldo atual da conta. Outro método chamado Sacar foi criado com um valor passado por parâmetro, onde o mesmo subtrai esse valor do saldo atual da conta.



Programação Orientada a Objetos : Encapsulamento

- O princípio do encapsulamento é a forma pela qual o programa é dividido a ponto de se tornar o mais isolado possível, ou seja, cada método pode ser executado isoladamente e retornar um resultado satisfatório ou não para cada situação. Sendo assim, o objeto não necessita conhecer qual forma cada método foi implementado.



Encapsulamento

Encapsulamento é o ato de esconder os detalhes internos de um objeto e expor apenas o necessário.

```
JS encapsulamento.js > ContaBancaria > depositar
1  class ContaBancaria {
2    #saldo = 0; // propriedade privada
3
4    constructor(cliente) {
5      this.cliente = cliente;
6    }
7
8    depositar(valor) {
9      if (valor > 0) {
10        this.#saldo += valor;
11      }
12    }
13
14    sacar(valor) {
15      if (valor > 0 && valor <= this.#saldo) {
16        this.#saldo -= valor;
17      }
18    }
19
20    consultarSaldo() {
21      return this.#saldo;
22    }
23  }
24
25  const conta = new ContaBancaria("João");
26  conta.depositar(1000);
27  conta.sacar(300);
28  console.log(conta.consultarSaldo()); // 700
29
```

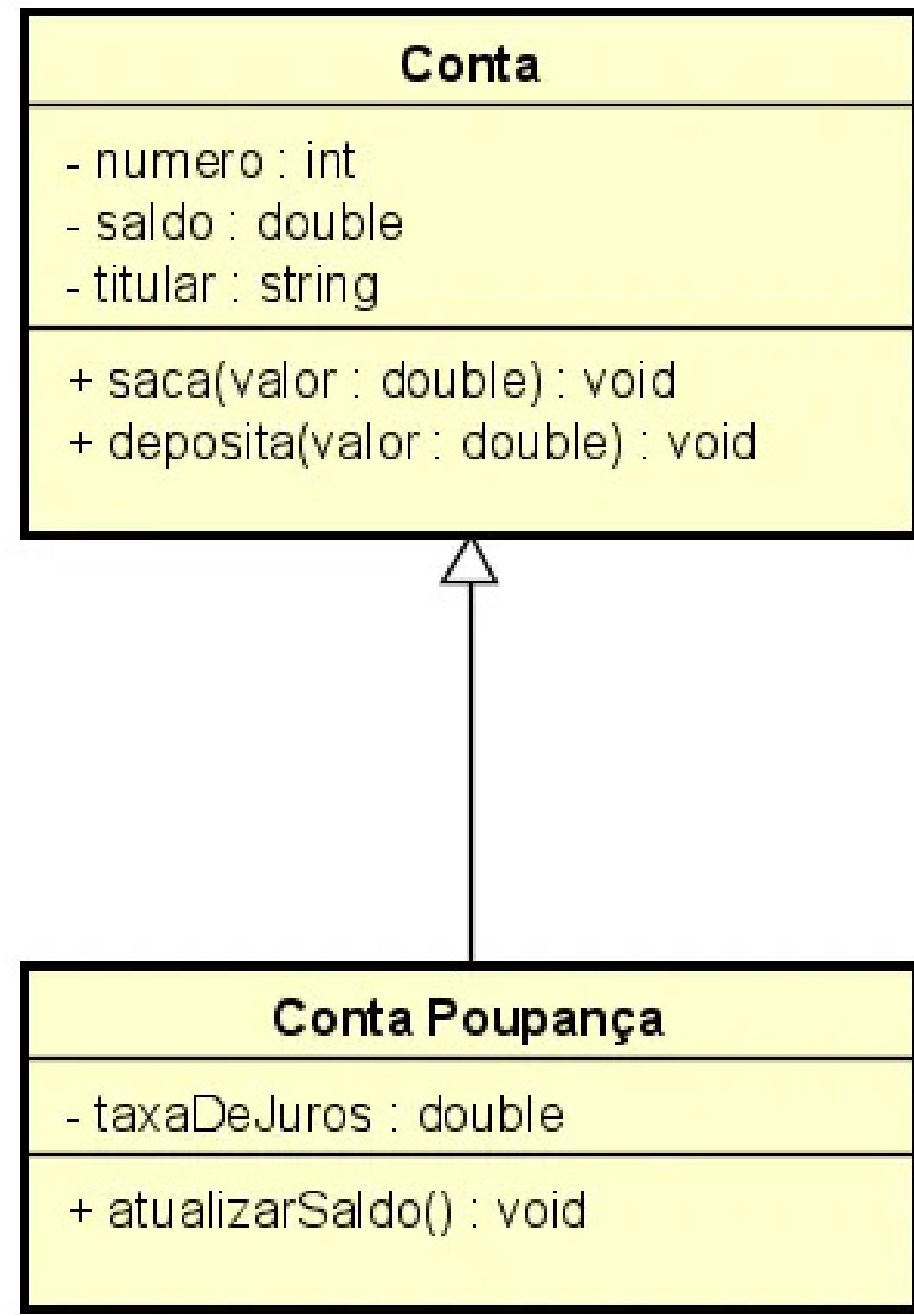


Programação Orientada a Objetos : Herança

- Chama-se de herança a capacidade de um objeto ser idealizado baseado em outro objeto. Nesse sentido, atributos e métodos são estendidos do objeto pai para o objeto filho. Agora, imagine que no nosso sistema bancário, precisamos também de uma conta do tipo Poupança, que terá as mesmas características base da Conta, mas com alguns implementos de acordo com as regras de negócio.



Programação Orientada a Objetos : Herança



Programação Orientada a Objetos : Herança

- Dessa forma, a **Conta Poupança** (classe filha) herda todos os atributos e métodos da **Conta** (classe pai), além de possuir o atributo **Taxa de Juros** e o método **Atualiza Saldo**, responsáveis por calcular o quanto a conta está rendendo e acrescentar ao saldo. Abaixo temos a classe **Conta Poupança** implementada a nível código:



Senac

Programação Orientada a Objetos : Herança

Herança permite que uma classe herde propriedades e métodos de outra.

```
JS heranca.js > ...
1  class Animal {
2    constructor(nome) {
3      this.nome = nome;
4    }
5
6    emitirSom() {
7      console.log(`#${this.nome} fez um som.`);
8    }
9  }
10
11 class Cachorro extends Animal {
12   emitirSom() {
13     console.log(`#${this.nome} latiu.`);
14   }
15 }
16
17 const dog = new Cachorro("Rex");
18 dog.emitirSom(); // Rex latiu.
19
```



Programação Orientada a Objetos : Herança

```
public class Conta Poupança : Conta {  
    public double TaxaDejuros { get; set; }  
  
    public void AtualizaSaldo() {  
        Saldo = Saldo * TaxaDejuros;  
    }  
}
```



Programação Orientada a Objetos : Polimorfismo

- A palavra **polimorfismo** vem do grego e significa aquilo que pode tomar várias formas, ou seja, um único nome representando códigos diferentes. No POO, há quatro tipos diferentes de polimorfismo: a inclusão, o paramétrico, a sobreposição e a sobrecarga.
- **Inclusão:**
- É o **polimorfismo básico**, quando uma classe Pai aponta para um objeto da classe Filha.



Senac

Programação Orientada a Objetos : Polimorfismo

Polimorfismo permite que métodos com o mesmo nome se comportem de maneira diferente em classes diferentes.

```
JS polimorfismo.js > ...
1  class Forma {
2    desenhar() {
3      console.log("Desenhando uma forma genérica.");
4    }
5  }
6
7  class Circulo extends Forma {
8    desenhar() {
9      console.log("Desenhando um círculo.");
10   }
11 }
12
13 class Quadrado extends Forma {
14   desenhar() {
15     console.log("Desenhando um quadrado.");
16   }
17 }
18
19 const formas = [new Circulo(), new Quadrado(), new Forma()];
20 formas.forEach(forma => forma.desenhar());
```



Senac

Programação Orientada a Objetos : Polimorfismo

```
static void Main(string[] args)
{
    Conta conta1 = new Conta(1001, 500, "Alex");
    Conta conta2 = new ContaPoupança (1002, 500, "Ana", 0.01);

    conta1.Saca(10);
    conta2.Saca(10);

    Console.WriteLine(conta1.Saldo);
    Console.WriteLine(conta1.Saldo);
}
```



Programação Orientada a Objetos : Polimorfismo

- No nosso exemplo, temos a classe pai Conta apontando para o seu próprio objeto sendo instanciado (o de uma conta comum), quanto para a instância de um objeto da sua classe filha, do tipo Conta Poupança.



Programação Orientada a Objetos : Polimorfismo

- **Paramétrico:**
- No polimorfismo paramétrico, a definição de um elemento por si só é incompleta, ela precisa parametrizar o tipo para que ele exista. No C# seu uso se dá com o Generics, permitindo que classes, interfaces e métodos possam ser parametrizados por tipo.
- Vamos imaginar que no nosso sistema, precisamos gerar relatórios listando tanto as contas ativas (com todos os seus atributos), quanto os titulares e os atributos relacionados a eles. Utilizando o Generics, podemos criar uma classe Lista<T>, sendo o seu parâmetro genérico. Assim, reutilizamos o código e o utilizamos para gerar os dois tipos de lista.



Senac

Programação Orientada a Objetos : Polimorfismo

- **Paramétrico:**
- No polimorfismo paramétrico, a definição de um elemento por si só é incompleta, ela precisa parametrizar o tipo para que ele exista. No C# seu uso se dá com o Generics, permitindo que classes, interfaces e métodos possam ser parametrizados por tipo.
- Vamos imaginar que no nosso sistema, precisamos gerar relatórios listando tanto as contas ativas (com todos os seus atributos), quanto os titulares e os atributos relacionados a eles. Utilizando o Generics, podemos criar uma classe Lista<T>, sendo o seu parâmetro genérico. Assim, reutilizamos o código e o utilizamos para gerar os dois tipos de lista.



Programação Orientada a Objetos : Polimorfismo

```
public class Lista<T> {
    public T[] itens;
    public int cont;

    public Lista() {
        this.itens = new T[0];
    }

    public void Add(T item) {
        var novosItens = new T[this.count + 1];

        for (var i = 0; i < this.cont; i++) {
            novosItens[i] = this.itens[i];
        }
        novosItens[this.cont] = item;

        this.itens = novosItens;

        this.cont++;
    }

    public void Imprimir() {
        for (int i = 0; i < cont; i++) {
            Console.Write("[" + itens[i] + "]");
        }
    }
}
```



Programação Orientada a Objetos : Polimorfismo

Utilizando o código acima, podemos instanciar listas tanto do tipo Conta quanto do tipo Titular:

```
var lista = new Lista<Conta>();
```

```
var lista = new Lista<Titular>();
```



Vamos criar um mini projeto com JavaScript orientado a objetos, HTML e CSS. A ideia será um sistema simples de gerenciamento de tarefas (To-Do List) com os seguintes recursos:

- **Funcionalidades:**
 - **Adicionar tarefas**
 - **Marcar tarefas como concluídas**
 - **Remover tarefas**
 - **Usar classes JS para representar tarefas e a lista**



Senac

PROJETO

Vamos criar um mini projeto com JavaScript orientado a objetos, HTML e CSS. A ideia será um sistema simples de gerenciamento de tarefas (To-Do List) com os seguintes recursos:

- Funcionalidades:
 - Adicionar tarefas
 - Marcar tarefas como concluídas
 - Remover tarefas
 - Usar classes JS para representar tarefas e a lista



Senac

Programação Orientada a Objetos : Sobreposição e Sobrecarga

- Para entendermos os dois últimos tipos de polimorfismo, precisamos ter clara a definição de assinatura, que representa a quantidade e os tipos de parâmetros. Então, se dois métodos possuem a mesma quantidade e o mesmo tipo de parâmetros, dizemos que eles possuem a mesma assinatura.
- No polimorfismo de sobreposição, os métodos possuem o mesmo nome, a mesma assinatura, mas estão em classes diferentes e apresentam comportamentos diferentes.
- Imaginemos que a regra de negócio do nosso sistema bancário mudou e agora ao sacar na conta Poupança, será cobrado uma taxa de juros de R\$2,00. A classe Poupança irá continuar herdando da classe pai Conta, mas o método Saca será sobreposto (override), e é esse método que irá apresentar o polimorfismo:



Programação Orientada a Objetos : Sobreposição e Sobrecarga

```
public class Conta Poupança : Conta {  
    public double TaxaDejuros { get; set; }  
  
    public void AtualizaSaldo() {  
        Saldo = Saldo * TaxaDejuros;  
    }  
  
    public override void Saca(double valor) {  
        base.Saca(valor);  
        Saldo -= 2.0;  
    }  
}
```



Programação Orientada a Objetos : Sobreposição e Sobrecarga

```
public class Conta Poupança : Conta {  
    public double TaxaDejuros { get; set; }  
  
    public void AtualizaSaldo() {  
        Saldo = Saldo * TaxaDejuros;  
    }  
  
    public override void Saca(double valor) {  
        base.Saca(valor);  
        Saldo -= 2.0;  
    }  
}
```



Programação Orientada a Objetos : Sobreposição e Sobrecarga

- Já no tipo de polimorfismo sobrecarga, os métodos possuem o mesmo nome e estão na mesma classe. No entanto, apresentam assinaturas diferentes e comportamentos também diferentes. Nessa situação, imagine que temos uma Conta Corrente onde é possível realizar empréstimos. E a taxa de juros desse empréstimo pode ser calculada com base no prazo para o pagamento ou no valor que foi emprestado. Por isso, na nossa classe temos dois métodos de Calcula Juros:



Senac

Programação Orientada a Objetos : Sobreposição e Sobrecarga

Conta Corrente

- limiteDeEmprestimo : double
- + empresta(valor : double) : void
- + calculaJuros(prazo : int) : double
- + calculaJuros(valorEmprestimo : double) : double



Comparação dos Paradigmas

| Aspecto | Programação Estruturada | Programação Orientada a Objetos |
|--------------------|------------------------------|-----------------------------------|
| Organização | Funções | Classes e Objetos |
| Facilidade Inicial | Mais simples | Levemente mais complexo |
| Manutenção | Diffícil em projetos grandes | Mais fácil e organizada |
| Reutilização | Limitada | Alta (com herança e polimorfismo) |
| Adequação | Pequenos projetos | Projetos médios e grandes |

 **Pergunta para reflexão:**
→ “O que vocês percebem de mais vantajoso em cada paradigma?”



Senac

Comparação dos Paradigmas

| Aspecto | Programação Estruturada | Programação Orientada a Objetos |
|--------------------|------------------------------|-----------------------------------|
| Organização | Funções | Classes e Objetos |
| Facilidade Inicial | Mais simples | Levemente mais complexo |
| Manutenção | Diffícil em projetos grandes | Mais fácil e organizada |
| Reutilização | Limitada | Alta (com herança e polimorfismo) |
| Adequação | Pequenos projetos | Projetos médios e grandes |

 **Pergunta para reflexão:**
→ “O que vocês percebem de mais vantajoso em cada paradigma?”



Senac

Atividade Prática

► Proposta:

Criar um sistema que cadastre informações de um funcionário e exiba os dados.

- Parte 1: Resolver usando programação estruturada (listas, dicionários, funções simples).
- Parte 2: Resolver usando POO (classe `Funcionario` com métodos e atributos).

Exemplos de atributos: nome, idade, cargo, salário.

Exemplos de métodos: aumentar salário, exibir informações.



Senac

Conclusão

- Utilizar a Programação Orientada a Objetos traz algumas vantagens importantes ao desenvolvimento, inclusive para aqueles que estão iniciando agora no mundo da programação. Por ser dividido em objetos, ele é mais fácil de compreender, é reutilizável e mais simples de dar manutenção, já que modificar um objeto não irá interferir em outro.



Senac



Fim!!!!



CNC | Fecomércio MG
Sindicatos Empresariais | Sesc

Senac, integrado
ao Sistema
Fecomércio MG

Siga o Senac em Minas nas redes sociais:

