

Classes e Encapsulamento

Labenu_



Sumário

Labenu_



O que vamos ver hoje? 🙄

- Introdução à Programação Orientada a Objetos
- Modelagem com **tipos abstratos de dados**
- Classes
- Encapsulamento



Motivação

- No backend, temos muito mais **liberdade** para estruturar o código
- É necessário fazer isso de maneira inteligente por uma série de motivos:
 - Legibilidade
 - Consistência
 - Testabilidade
 - Fácil manutenção
- Os paradigmas de programação indicam formas de **pensar** e **organizar** o código, determinando **regras** que buscam garantir os pontos acima



Paradigmas de Programação



Paradigmas de Programação

- Estruturada
 - Utiliza estruturas condicionais e loops para controlar o fluxo de execução do programa
 - Impede controle absoluto do fluxo de execução do programa
- Funcional
 - Utiliza composição de funções que operam em cima dos dados, transformando-os
- Orientada a objetos
 - Utiliza objetos para modelar e controlar os dados e suas manipulações
 - Restringe acesso a variáveis e rotinas a partes específicas do código



Paradigmas e as linguagens

- Linguagens de programação costumam suportar um ou mais paradigmas de programação específicos
 - Java → Orientação a Objetos
 - LISP → Funcional
- Algumas são linguagens **multi-paradigma**, que permitem a escrita de código seguindo vários paradigmas
 - **Javascript/Typescript**
 - Python
- Mais liberdade gera mais responsabilidade. Ao usar linguagens multi-paradigma, pode ser interessante adotar práticas de paradigmas específicos



Programação Orientada a Objetos

- A **P**rogramação **O**rientada a **O**bjetos é um dos paradigmas mais difundidos e conhecidos
- Criada por volta dos anos 60 para ajudar a lidar com a complexidade crescente dos programas
- Utiliza principalmente o conceito de **objetos** como unidade central de estruturação de dados e rotinas
- Os objetos podem ser criados de forma a **abstrair** conceitos do mundo real, simplificando o entendimento do código por pessoas



Objetos

- No contexto da POO, cada objeto é definido por um conjunto de:
 - **Atributos:** informações (dados) do objeto
 - **Métodos:** rotinas (funções) que operam sobre os atributos do objeto, ou contextualmente relacionadas
- Os objetos são criados a partir de uma **classe**, que define quais atributos ele deve ter, e quais são os métodos que ele terá quando for criado

Atenção: nesse contexto, o conceito de objetos é um pouco diferente do que conhecemos como “objeto” no Javascript. Os objetos do JS podem ter atributos e métodos, mas não precisam ser criados a partir de algum “template” (como as classes).



Classes

- Define o **formato** de um objeto
 - Atributos (propriedades)
 - Métodos (funções)



- Possuem as implementações dos métodos e podem possuir valores padrão para os atributos
- No Typescript, por definir as propriedades e métodos, representa também um **tipo**, que pode ser usado para definir variáveis
- Dessa forma, podemos dizer que representam **Tipos Abstratos de Dados**



Abstração 🤔

- Forma de criar representações mais simples de coisas complexas, escondendo essa complexidade com interfaces mais simples
- Esconde detalhes de implementação, facilitando o uso e garantindo consistência, mas limitando possibilidades
- É possível criar múltiplos níveis de abstração
- Boas abstrações tornam o código mais legível, reaproveitável, consistente e expansível, mas são difíceis de fazer e requerem muito estudo e iteração



Abstração - Exemplo

Site / Notebook



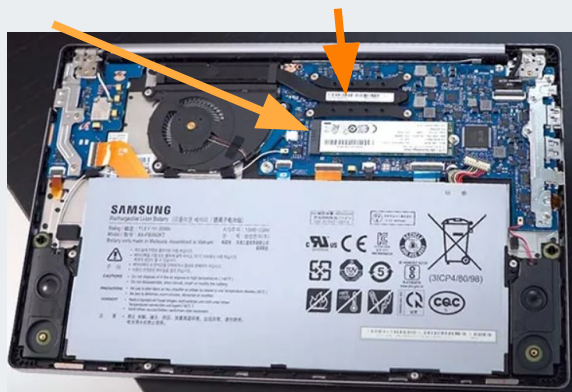
REACT

```
1 import { MouseEvent, Component } from 'react'
2 import React from 'react'
3
4 type Props = {
5   onClick?: MouseEvent<HTMLElement>: void
6   color?: 'blue' | 'green' | 'red'
7   type?: 'button' | 'submit'
8 }
9
10 class Button extends Component<Props> {
11   static defaultProps = {
12     color: 'blue',
13     type: 'button',
14   }
15   render() {
16     const { onClick: handleClick, color, type, children } = this.props
17
18     return (
19       <button type={type} style={{ color }} onClick={handleClick}>
20         {children}
21       </button>
22     )
23   }
24 }
```

Windows - C#

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Diagnostics;
6 using System.IO;
7 using System.Linq;
8 using System.ServiceProcess;
9 using System.Text;
10 using System.Threading.Tasks;
11 using System.Timers;
12
13 namespace MyFirstService
14 {
15   [author: "Basil170, 1 hour ago", 1 change]
16   public partial class Service1 : ServiceBase
17   {
18     Timer timer = new Timer(); // name space(using System.Timers);
19     [author: "Basil170, 1 hour ago", 1 author, 1 change]
20     public Service1()
21     {
22       InitializeComponent();
23     }
24     [author: "Basil170, 1 hour ago", 1 author, 1 change]
25     protected override void OnStart(string[] args)
```

SSD processador



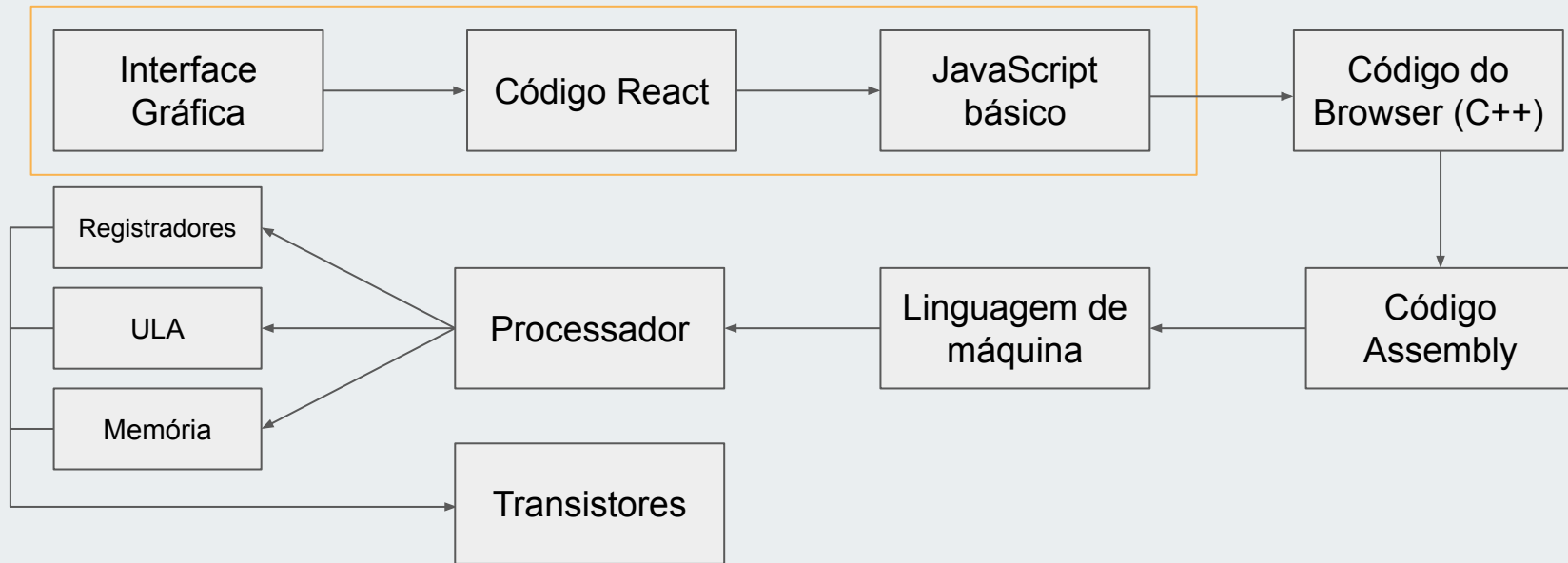
Código de máquina

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

Assembly

```
AX=FFFF BX=0000 CX=00EC DX=0000 SP=FFFF BP=0000 SI=0000 DI=
DS=0743 ES=0743 SS=0743 CS=0743 IP=0100 NU UP EI PL NZ NA PO
0743:0100 EB16 JMP 0118
0743:0100 EB16 JMP 0118
0743:0102 0000 ADD [BX:SI],AL
0743:0104 0000 ADD [BX:SI],AL
0743:0106 9C PUSHF
0743:0107 80FC82 CMP AH,02
0743:010A 7504 JNZ 0110
0743:010C B490 MOV AH,90
0743:010E 3D POPF
0743:010F CF IRET
0743:0110 9D POPF
0743:0111 9C PUSHF
0743:0112 2E CS:
0743:0113 FF1E0201 CALL FAR [0102]
0743:0117 CF IRET
0743:0118 B409 MOV AH,09
0743:011A B40E91 MOV DX,013E
0743:011D CD21 INT 21
0743:011F B01735 MOV AX,3517
```

Abstração - Exemplo



Tipos Abstratos de Dados

- Tipos nativos do TypeScript já são simples abstrações de conceitos computacionais simples: *booleanos, números, strings, arrays, objetos*
- A ideia da POO é criar novos tipos de dados - classes - criando novas abstrações, mais próximas do entendimento humano
- Podemos chamá-los de *Tipos **Abstratos** de **Dados** (TAD)*
- Até aqui, utilizamos os **types** para isso
- Agora, vamos começar a usar **classes**



TADs - Exemplo 🎲



Cachorro

nome: string
peso: number

latir(): void
comer(quantidade: number): void



TADs - Exemplo 🎲



Coruja

nome: string
peso: number

chirriar(): void
comer(quantidade: number): void
voar(tempo: number): void



TADs - Exemplo 🎲

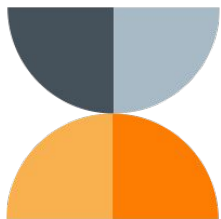


User

nome: string
tarefas: Tarefa[]

adicionarTarefas(): void
removerTarefa(): void





- Programação Orientada a Objetos é um paradigma de programação que incentiva o uso de **objetos** para criar **abstrações**
- Objetos possuem **atributos** e **métodos**
- **Classes** determinam **formato** dos objetos



Classes

Labenu_



Declarando uma classe

- Utilizamos a *keyword* **class** para declarar uma classe, que será nosso modelo para construir objetos (instâncias)

```
class NomeDaClasse {  
  
}
```



Declarando uma classe

- O corpo da classe pode conter atributos e uma função chamada **constructor**, que recebe como parâmetros os atributos dessa classe, referenciados pela *keyword* **this**

```
class NomeDaClasse {  
    nomeDoAtributo: tipoPrimitivo  
  
    constructor ( parametro: tipoPrimitivo ){  
        this.nomeDoAtributo = parametro  
    }  
}
```

Vamos ver na prática! 



Declarando uma classe

- O corpo da classe pode conter também **métodos**:
 - Podem receber parâmetros ou não
 - podem alterar os atributos dessa classe, referenciados pela *keyword* **this**

```
class NomeDaClasse {  
    nomeDoAtributo: tipoPrimitivo  
  
    constructor (parametro: tipoPrimitivo){  
        this.nomeDoAtributo = parametro  
    }  
    nomeDoMetodo (parametro: tipoPrimitivo): tipoPrimitivoDeSaída {  
        this.nomeDoAtributo = parametro  
    }  
}
```




Declarando uma classe

- A criação de novas instâncias(objetos) é feita com a *keyword* **new**

```
class NomeDaClasse {  
    nomeDoAtributo: tipoPrimitivo  
  
    constructor (parametro: tipoPrimitivo){  
        this.nomeDoAtributo = parametro  
    }  
    nomeDoMetodo (parametro: tipoPrimitivo): tipoPrimitivoDeSaída {  
        this.nomeDoAtributo = parametro  
    }  
}
```

const nomeDoSeuObjeto = **new** NomeDaClasse(parametro)

Vamos ver na prática! 



Declarando uma classe

- A classe pode ser usada como um **tipo** no TypeScript

const nomeDaVariavel: **NomeDaClasse** = valorDaVariável

Tipagem



Vamos ver na prática! 



Declarando uma classe

- É possível criar diversas **instâncias** da classe. Essas instâncias são objetos que necessariamente possuem todos aqueles atributos e métodos da classe.
- A declaração da classe representa o TAD. Ela é um **modelo** para objetos daquele tipo

```
class Cachorro {  
  nome: string  
  constructor(novoNome: string){  
    this.nome = novoNome  
  }  
  latir ():void{  
    console.log("Auau! Auau!")  
  }  
}
```

const pluto = **new** Cachorro("Pluto")

const scooby = **new** Cachorro("Scooby")

const pateta = **new** Cachorro("Pateta")

Vamos ver na prática! 



Inicializando (Construindo) 🚧

- Podemos ter ações que devem ser executadas ao criar uma instância da classe
- Para isso, existe o **construtor**
- Deve ser declarado com a função **`constructor()`**
- Pode receber parâmetros, que devem ser passados no momento de criar a instância
- Toda classe possui um constructor. Quando não o explicitamos, ele existe como **construtor vazio**



This

- A referência **this** é usada para **referenciar** os **atributos** e **métodos** daquela **instância**
- Normalmente usada no construtor e nos métodos para acessar e manipular os atributos do objeto



Exemplo

```
export class Cachorro {
  nome: string;
  peso: number;

  constructor(nome: string, peso: number) {
    this.nome = nome;
    this.peso = peso;
  }

  latir(): void {
    console.log("au au 🐶");
  }

  comer(quantidade: number): void {
    console.log("a coruja comeu " + quantidade);
  }
}
```

```
export class Coruja {
  nome: string;
  peso: number;

  constructor(nome: string, peso: number) {
    this.nome = nome;
    this.peso = peso;
  }

  pio(): void {
    console.log("Hu Hu 🦉");
  }

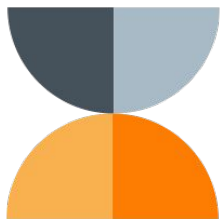
  comer(quantidade: number): void {
    console.log("a coruja comeu " + quantidade);
  }

  voar(quantidade: number): void {
    console.log("A coruja voou por " + quantidade + " minutos");
  }
}
```



Exercício 1

- Transforme o *type* **Pessoa** em uma *classe* **Pessoa**



- Programação Orientada a Objetos é um paradigma de programação que incentiva o uso de objetos para criar abstrações
- Objetos possuem atributos e métodos
- Classes determinam formato dos objetos
- Classes podem ser usadas como tipos
- Construtor serve para inicializar variáveis
- Para criar uma instância, usa-se a *keyword* **new**



Encapsulamento

Labenu_

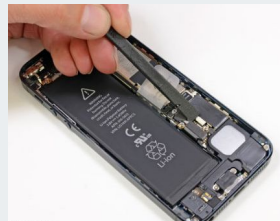


Conceito

- Boas abstrações escondem detalhes da implementação, disponibilizando somente o necessário para o uso correto do objeto
- Isso significa expor somente métodos específicos
- Os atributos, em geral, são todos escondidos e disponibilizados por meio de métodos



Public vs private



- É possível declarar atributos e métodos de classe com as *keywords* **public** e **private**
- Por padrão, as variáveis são públicas, o que significa que elas podem ser acessadas fora da classe
- Variáveis privadas só podem ser acessadas de dentro da própria classe (usando a *keyword* `this`)



Exemplo

```
export class Dog {
  private name: string;
  private weight: number;

  constructor(name: string, weight: number){
    this.name = name;
    this.weight = weight;
  }

  bark(): void {
    console.log("au au 🐶");
  }

  eat(quantity: number): void {
    console.log("the dog has eaten " + quantity);
  }
}

const dog1 = new Dog("Snoopy", 10);
dog1.name = "Snoopy";
```

Property 'name' is private and only accessible within class 'Dog'. ts(2341)

[Peek Problem](#) No quick fixes available



Getters e Setters



- É comum tornar todos os atributos privados e controlar o acesso por métodos públicos
- Esses métodos são chamados de **getters** (para pegar o atributo) e **setters** (para definir)
- Isso garante consistência e extensibilidade

```
export class Dog {  
  private name: string  
  private weight: number  
  
  constructor(name: string) {  
    this.name = name  
    this.weight = 10  
  }  
  
  public getName() {  
    return this.name  
  }  
  
  public setName(  
    newName: string  
  ) {  
    this.name = newName  
  }  
}
```



Exercício 2

- Torne as propriedades da classe Estudantes **privadas**.
- Adicione os **getters** para pegar o nome ou matrícula do estudante e **setters** para alterar a matrícula do estudante.
- Crie uma instância da classe Estudante e imprima o nome e matrícula do estudante criado. **Labenu_**

Resumo

Labenu_



Resumo

- Programação Orientada a Objetos é um paradigma de programação que incentiva a **modularização** e a **abstração** do código
- Módulos são blocos que cuidam de partes específicas do sistema
- Abstração expõe comportamentos específicos de módulos
- Tipos Abstratos de dados permitem criar módulos e abstrações em código. Possuem atributos (variáveis) e métodos (funções)



Resumo

- TADs podem ser representados por classes em TypeScript
- Classes podem ser usadas como tipos
- Construtor serve para inicializar variáveis
- Para criar uma instância, usa-se a *keyword* **new**
- **Atributos** e **métodos** podem ser **públicos** ou **privados**
- Os privados só podem ser acessados de dentro da própria classe
- **Getters** e **Setters** são usados para manipular atributos privados





Obrigado!