Estruturas de Dados - TypeScript

Tipos Primitivos

Para iniciar é essencial entender sobre tipos primitivos para escrever um código mais seguro, legível e eficiente. Os tipos primitivos são os tipos de dados básicos que compõem qualquer aplicação TypeScript. Eles representam valores imutáveis e são usados para armazenar dados fundamentais, como números, textos e valores lógicos.

- number - Representa números inteiros e decimais.

```
let idade: number = 20;
let preco: number = 19.99;
```

- string - Representa cadeias de caracteres.

```
// É possivel declarar string com aspas simples(''), duplas("")
// ou template strings(``)

let nome: string = "Felipe";
let professor: string = `O professor é ${nome}`;
```

- boolean - Representa valores lógicos **true** ou **false**.

```
const aprovado: boolean = true;
const reprovado: boolean = false;
```

Existem alguns tipos em TypeScript que são utilizados em algumas situações especiais

- any - Repesenta qualquer coisa, ou seja, pode ser qualquer tipo

```
// Deve ser evitado pois tira o sentido de usar TypeScript
// e deixa o código inseguro

let qualquerCoisa: any = 1;
qualquerCoisa = "Sou uma string";
qualquerCoisa = true;
```

- unknown - Representa um valor desconhecido.

```
// semelhante ao any, mas com mais segurança, exigindo
// uma validação antes de utiliza-lo

let valor: unknown;

if (typeof valor === "string") {
  console.log(valor.toUpperCase());
}
```

- null e undefined - Representam, respectivamente, ausência de valor e a falta de declaração

```
const vazio: null = null;
const naoDefinido: undefined = undefined;
```

- void e never - Representam, respectivamente, a ausência de retorno de uma função e um valor que nunca pode ser alcancado

```
const imprimirMensagem = (): void => console.log("Hello World!");

// Use never para indicar que uma função não pode
// ter um valor de retorno válido.

function erro(): never {
  throw new Error("Algo deu errado");
}
```

- symbol - Representa um identificador exclusivo de uma propriedade de um objeto

```
const id = Symbol("id");
```

- bigint - Representa um número inteiro grande, alem dos limites do tipo number

```
const numeroGrande: bigint = 9007199254740991n;
```

Lista - TypeScript

Como funcionam Listas em TypeScript?

No TypeScript, as listas são representadas por arrays, que permitem armazenar múltiplos valores do mesmo tipo. Além disso, podemos utilizar tuplas, que permitem listas com tipos diferentes para cada posição.

Como criar uma lista em TypeScript?

Para criar uma lista em TypeScript, podemos utilizar o array literal, ou o construtor Array.

```
const numeros: number[] = [1, 2, 3, 4, 5];
// podemos criar uma lista com mais de um tipo de dado
const numString: (number | string)[] = [1, "Yan", 2, "Artur", 3, "Marquin"];
const nomes: Array<string> = ["Yan", "Artur", "Marquin"];
```

Como acessar elementos de uma lista em TypeScript?

Para acessar elementos de uma lista em TypeScript, podemos utilizar o operador de colchetes.

```
const numeros: number[] = [1, 2, 3];

console.log(numeros[0]); // 1
console.log(numeros[1]); // 2
console.log(numeros[2]); // 3
```

Tuplas em TypeScript

As tuplas são um tipo especial de lista em TypeScript que permite armazenar diferentes tipos de dados em uma ordem específica.

```
const tupla: [number, string] = [1, "Yan"];
console.log(tupla[0]); // 1
console.log(tupla[1]); // "Yan"

tupla.push("Artur", 2); // Causa um erro por estar em ordem diferente
```

Metodos para trabalhar com listas em TypeScript

```typescript
let alunos: string[] = ["Clara", "Gustavo", "Davi"];

// Adicionar um elemento ao final da lista
alunos.push("Yan");

// Remover o ultimo elemento da lista
const alunoRemovido = alunos.pop();

// Remover ou Adicionar um elemento em uma posição desejada da lista
alunos.splice(1, 0, "Artur");
alunos.splice(1, 1);

// Inverter a ordem dos elementos da lista
alunos.reverse();

// Ordenar os elementos da lista
alunos.sort();

// Buscar um elemento na lista
const alunoEncontrado = alunos.find((aluno) => aluno === "Yan");

// Buscar o indice de um elemento na lista
const indicesAluno = alunos.indexOf("Yan");

// Verificar se um elemento existe na lista
const alunoExiste = alunos.includes("Yan");

// Adicionar um elemento ao inicio da lista
alunos.unshift("Yan");

// Remover o primeiro elemento da lista
alunos.shift();

// interar sobre os elementos da lista
alunos.forEach((aluno) => console.log(aluno));
```

Existem mais metodos que podem ser utilizados para trabalhar com listas em TypeScript, vai da sua curiosidade e vontade de aprender mais!

Pilha - TypeScript

Como usar uma pilha em TypeScript

Para usarmos uma pilha em TypeScript, podemos criá-la de varias formas diferentes, usando simplesmente arrays e as funções da própria linguagem, como push e pop, mas a forma que irei abordar aqui é usando arrays e classes.

Implementando uma pilha em TypeScript

Para implementar uma pilha em TypeScript, precisaremos uma classe que represente a pilha, que terá um array privado para armazenar os elementos da pilha, usando generics para definir o tipo do array.

```
class Pilha<t> {
  private pilha: T[] = [];

  push(item: T): void {
    this.pilha.push(item);
  }

  pop(): T | undefined {
    return this.pilha.pop();
  }

  peek(): T | undefined {
    return this.pilha[this.pilha.length - 1];
  }

  isEmpty(): boolean {
    return this.pilha.length === 0;
  }

  size(): number {
    return this.pilha.length;
  }

  clear(): void {
    this.pilha = [];
  }
}
```

Usando a pilha em TypeScript

```
const pilha = new Pilha<number>();

pilha.push(1); // [1]
pilha.push(2); // [1, 2]
pilha.push(3); // [1, 2, 3]

console.log(pilha.pop()); // Output: 3
console.log(pilha.peek()); // Output: 2
console.log(pilha.isEmpty()); // Output: false
console.log(pilha.size()); // Output: 2
pilha.clear(); // Limpa a pilha
console.log(pilha.isEmpty()); // Output: true
```

Fila - TypeScript

Como usar uma fila em TypeScript

Da mesma forma da pilha, é possível usar uma fila em TypeScript, usando simplesmente arrays e as funções da propia linguagem, como push e shift, mas a forma que irei abordar aqui é usando arrays e classes.

Implementando uma fila em TypeScript

Para implementar uma fila em TypeScript, precisaremos uma classe que represente a fila, que terá um array privado para armazenar os elementos da fila, usando generics para definir o tipo do array.

```
class Fila<t> {
  private fila: T[] = [];

  enqueue(item: T): void {
    this.fila.push(item);
  }

  dequeue(): T | undefined {
    return this.fila.shift();
  }

  peek(): T | undefined {
    return this.fila[0];
  }

  isEmpty(): boolean {
    return this.fila.length === 0;
  }

  size(): number {
    return this.fila.length;
  }

  clear(): void {
    this.fila = [];
  }
}
```

Usando a fila em TypeScript

Para usar uma fila em TypeScript, basta criar uma instância da classe Fila e chamar as funções da mesma, como enqueue, dequeue, peek, isEmpty, size e clear.

```
const fila = new Fila<number>();

fila.enqueue(1); // [1]
fila.enqueue(2); // [1, 2]
fila.enqueue(3); // [1, 2, 3]

console.log(fila.dequeue()); // 1
console.log(fila.peek()); // 2
console.log(fila.isEmpty()); // false
console.log(fila.size()); // 2
fila.clear();
console.log(fila.isEmpty()); // true
```

Estruturas da Linguagem TypeScript

Já foi mostrado as estruturas abordadas na disciplina de Estruturas de Dados, agora vamos abordar algumas estruturas da linguagem TypeScript.

Classes

Uma classe no TypeScript representa um objeto. Elas permitem definir propriedades e métodos, e também podem ser estendidas e instanciadas para criar objetos com comportamentos e estados específicos.

```
class Pessoa {
  // Propriedades
  public nome: string; // public - pode ser acessada de fora da classe
  private cpf: string; // private - não pode ser acessada de fora da classe
  protected idade: number; // protected - apenas pela classe e subclasses

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

  // Metodos
  public exibirInformacoes(): void {
    console.log(`Nome: ${this.nome}`);
    console.log(`Idade: ${this.idade}`);
    console.log(`CPF: ${this.cpf}`);
  }

  // Getters e Setters
  get seuCPF(): string {
    return this.cpf;
  }

  set seuCPF(cpf: string) {
    if (cpf.length === 14) this.cpf = cpf;
  }
}

const pessoa1 = new Pessoa("Yan", "123.456.789-10", 20);
pessoa1.exibirInformacoes();
pessoa1.seuCPF = "123.456.789-11";
```

Interface

Uma interface no TypeScript representa um contrato que uma classe ou objeto deve seguir. Ela serve para definir o formato e o comportamento de um objeto.

```
interface Moto {
  marca: string;
  modelo: string;
  cilindrada: number;
  exibirDetalhes(): void;
}

const moto1: Moto = {
  marca: "Yamaha",
  modelo: "MT-07",
  cilindrada: 689,

  exibirDetalhes(): void {
    console.log(
      `Moto: ${this.marca} - ${this.modelo}, Cilindrada: ${this.cilindrada} cc`
    );
  },
};

// Tambem podemos criar uma classe que implementa a interface
class classeMoto implements Moto {
  marca: string;
  modelo: string;
  cilindrada: number;

  constructor(marca: string, modelo: string, cilindrada: number) {
    this.marca = marca;
    this.modelo = modelo;
    this.cilindrada = cilindrada;
  }

  exibirDetalhes(): void {
    console.log(
      `Moto: ${this.marca} - ${this.modelo}, Cilindrada: ${this.cilindrada} cc`
    );
  }
}

const moto2: classeMoto = new classeMoto("Yamaha", "R15", 155);

moto1.exibirDetalhes(); // Moto: Yamaha - MT-07, Cilindrada: 689 cc
moto2.exibirDetalhes(); // Moto: Yamaha - R15, Cilindrada: 155 cc
```

Type

Os type são similares às interfaces, mas permitem definições mais complexas, como uniões e interseções.

```
type Pessoa = {
  nome: string;
  idade: number;
};

type Programador = {
  linguagem: string;
  experiencia: number;
};

type Desenvolvedor = Pessoa & Programador;

const desenvolvedor1: Desenvolvedor = {
  nome: "Yan",
  idade: 20,
  linguagem: "TypeScript",
  experiencia: 1,
};
```

Enum

Os enum nos permitem definir um conjunto de constantes que representam um conjunto finito de valores. Elas ajudam a manter o código mais organizado e legibilidade.

```
enum DiasDaSemana {
  Segunda = 1,
  Terca = 2,
  Quarta = 3,
  Quinta = 4,
  Sexta = 5,
  Sabado = 6,
  Domingo = 7,
}

console.log(DiasDaSemana.Segunda); // 1
```