APLICAÇÕES INTERATIVAS

TYPESCRIPT II

Codificação do Terminal do VSCode

- Para corrigir problemas de acentuação no terminal do VSCode, execute:
  - **`chcp 65001`**

Comentários de código

- Comentários não são processados pelo compilador e permitem anotar código

```
//Comentário
```

```
/*
 * Comentário
 * multinha
 */
```

Operadores aritméticos e atribuições

- Todas as operações básicas podem ser realizadas em TypeScript da mesma forma como em JavaScript:
  - **+** Adição
  - **-** Subtração
  - ***** Multiplicação
  - **/** Divisão
  - **%** Módulo
  - **++** Incrementar
  - **--** Decrementar

Operadores

- Operadores relacionais e operadores lógicos em TypeScript são idênticos aos em JavaScript

Operadores relacionais

Operador	Nome
==	Igual
!=	Não igual
===	Estritamente igual
!==	Estritamente não igual
>	Maior que
>=	Maior que ou igual
<	Menor que
<=	Menor que ou igual

Operadores lógicos

Operador	Nome
&&	E
||	OU
!	NÃO

Condicionais

- Assim como em JavaScript, é possível especificar desvios condicionais
- “if” como dos principais comandos de desvio

if

- Permite especificar uma ou mais condições
- Executa os comandos em seu interior apenas caso a condição seja verdadeira

if

```
if (condicao) {
    //Código a executar
}
```

if

Condição a validar

Comando if `if (condicao) {` *Início do bloco do if*

Código a executar caso a condição seja verdadeira

`}` *Fim do bloco do if*

if

```
import * as rs from 'readline-sync';

const idade: number = Number(rs.question('Sua idade? '));

if (idade >= 18) {

    console.log('Você é maior de idade');

}
```

if...else

- Permite especificar comandos para quando a condição for verdadeira e para quando a condição for falsa (else)
- Não pode ser utilizado separadamente (apenas em conjunto com if)

if...else

```
if (condicao) {
    //Código a executar
    //caso verdadeiro
}
else {//Opcional
    //Código a executar
    //caso falso
}
```

if...else

Condição a validar

Comando if `if (condicao) {` *Início do bloco do if*

Código a executar caso a condição seja verdadeira

`}` *Fim do bloco do if*

Comando else `else {` *Início do bloco do else*

Código a executar caso a condição seja falsa

`}` *Fim do bloco do else*

if...else

```
import * as rs from 'readline-sync’;

const idade: number = Number(rs.question('Sua idade? ’));

if (idade >= 18) {
    console.log('Você é maior de idade');
}
else {
    console.log('Você é menor de idade');
}
```

if...else if...else

- Permitem efetuar diversas verificações de condição sequenciais
- A primeira condição será verificada. Se for falsa, a segunda condição será verificada. Isso procede até uma condição ser verdadeira ou não existirem mais condições a verificar

if...else if...else

```
if (condicao1) {
        //Código a executar caso
        //condição1 seja verdadeira
}
else if (condicao2) {//Opcional
        //Código a executar caso condicao1
        //seja falsa, mas condicao2 seja verdadeira
} else if (condicao3) {//Opcional
        //Código a executar caso condicao1 e condicao2
        //sejam falsas, mas condicao3 seja verdadeira
}
//...
else { //Opcional
        //Código a executar caso nenhuma das
        //condições sejam verdadeiras
}
```

if...else if...else

```
import * as rs from 'readline-sync';

const idade: number = Number(rs.question('Sua idade? '));

if (idade < 18) {
    console.log('Você é menor de idade');
}
else if (idade < 60) {
    console.log('Você é maior de idade');
}
else {
    console.log('Você é idoso');
}
```

Condicionais aninhadas

- É possível inserir condicionais dentro de outros blocos de condicionais, criando blocos de condicionais aninhados

Condicionais aninhadas

```
if (condicao1) {
        //Comandos a executar quando condicao1 for verdadeira
        if (condicao2) {
                //Comandos a executar quando condicao1 e a
                //condicao2 forem verdadeiras
        } else {
                //Comandos a executar quando condicao1 for
                //verdadeira, mas a condicao2 for falsa
        }
}
else {
        //Comando a executar quando a condicao1 for falsa
}
```

Uso de operadores relacionais

- Em todos os casos de if, é possível validar múltiplas condições utilizando operadores relacionais

```
if ((condicao1 && condição2) || condicao3) {
      //Código a executar
}
```

switch...case

- Permite validar determinada variável quanto a determinados valores
- Útil quando os valores possíveis são pré-definidos e em menor quantidade

switch...case

```
import * as rs from 'readline-sync';

let day: number = Number(rs.question('Entre com o número do dia da semana: '));

switch (day) {
    case 0:
        console.log('Hoje é domingo.');
        break;
    case 1:
        console.log('Hoje é segunda-feira.');
        break;
    case 2:
        console.log('Hoje é terça-feira.');
        break;
    case 3:
        console.log('Hoje é quarta-feira.');
        break;
    case 4:
        console.log('Hoje é quinta-feira.');
        break;
    case 5:
        console.log('Hoje é sexta-feira.');
        break;
    case 6:
        console.log('Hoje é sábado.');
        break;
    default:
        console.log('Dia inválido!');
        break;
}
```

Inferência de Tipos

- Apesar de ser fortemente tipado, TypeScript é capaz de “adivinhar” tipos de dados, principalmente quando ocorrem atribuições
- Nestas situações, não é necessário especificar explicitamente tipos de dados

Inferência de Tipos

```
import * as rs from 'readline-sync';

const idade: number = Number(rs.question('Sua idade? '));
const mensagemMaior: string = 'Você é maior de idade';

if (idade >= 18) {
    console.log(mensagemMaior);
}
```

Inferência de Tipos

```
import * as rs from 'readline-sync';

const idade = Number(rs.question('Sua idade? '));
const mensagemMaior = 'Você é maior de idade';

if (idade >= 18) {
    console.log(mensagemMaior);
}
```

Inferência de Tipos

- Evite especificar explicitamente tipos em TypeScript sempre que a inferência de dados for possível
- Lembre-se! Ainda há tipagem forte! Ela só está sendo declarada de forma mais concisa
- Continua não sendo possível mudar o tipo de uma variável

Problema de Exemplo

1. Escreva um programa em TypeScript que leia o valor de dois números inteiros e a operação aritmética desejada (adição, subtração, multiplicação e divisão) e calcule a resposta adequada

Repetições

- É possível repetir a execução de comandos de acordo com determinadas condições, assim como em JavaScript
- TypeScript também possui tipos especiais de for, que estudaremos mais adiante

while

- Permite repetir determinado bloco de comandos enquanto uma condição for verdadeira
- A condição é verificada antes da primeira execução, de forma que o bloco de repetição pode nunca ser executado

while

- Requer uma condição de saída, ou a execução ficará presa num loop infinito

while

```
while (condicao) {
    //Comandos a repetir
}
```

while

```
let numero = 2;

while (numero <= 12) {
    numero = numero + 2;
}

console.log(numero);
```

do...while

- Semelhante ao while, permite repetir determinado bloco de comandos enquanto uma condição for verdadeira
- Garante a execução do código a ser repetido pelo menos uma vez, já que a verificação da condição ocorre **após** sua execução

do...while

```
do {
    //Comandos a repetir
} while (condicao);
```

do...while

```
import * as rs from 'readline-sync';

let nome: string;

do {

    nome = rs.question('Digite seu nome: ');

} while (nome === '');
```

for (simples)

- Permite repetir determinado comando de forma pré-definida
- Útil para quando já se sabe antecipadamente a quantidade de repetições

for (simples)

```
for (let i = 0; i < qtdRepeticoes; i++) {
     //Comandos a repetir
}
```

for (simples)

```
import * as rs from 'readline-sync';

let repeticoes = Number(rs.question('10 elevado a? '));
let resultado = 1;

for (let i = 0; i < repeticoes; i++) {

    resultado = resultado * 10;

}

console.log('Resultado: ' + resultado);
```

Problema de Exemplo

2. Faça um programa capaz de somar dois números digitados pelo usuário
    - Importante: O programa deverá solicitar continuamente a entrada de números até que um número válido seja digitado para cada um dos valores

Funções

- Blocos de construção fundamentais de TypeScript
- Assim como em JavaScript, uma função é um procedimento - um conjunto de instruções que executa uma tarefa ou calcula um valor
- Para usar uma função, você deve defini-la ou utilizar uma função já definida (outro script, lib, Node ou navegador)

Utilizando uma Função

- Quando utilizamos a instrução “log” ou a instrução “question” anteriormente, estávamos fazendo uso de uma função definida pelo Node e por bibliotecas
- Para chamar uma função, basta declarar seu nome + abre e fecha parênteses.

Executando uma função

```
funcao();
```

Nome da função

Declarando uma Função

- Assim como em JavaScript, podemos declarar nossas próprias funções
- Para isto, basta utilizar a palavra chave “function”, seguida do nome da função, “( )” e de seu retorno (se houver)
- O corpo da função fica dentro de um bloco de código

Executando uma função

```
function nomeFuncao() {
      //Comandos da função
}
```

Executando uma função

Palavra-chave para definição de uma função

Nome da função definida

```
function nomeFuncao() {
    //Comandos da função
}
```

Corpo da função

Equivalente a função sem retorno

Sem retorno

```
function nomeFuncao() : void {
      //Comandos da função
}
```

Função com retorno

Retorno

```
function nomeFuncao() : number {
      //Comandos da função
}
```

Função com parâmetro

Parâmetro

```
function nomeFuncao(param1: string) {
    //Comandos da função
}
```

Função com parâmetros

Parâmetros

```
function nomeFuncao(param1: string, param2: number) {
      //Comandos da função
}
```

Função com retorno e parâmetros

Parâmetros *Retorno*

```
function nomeFuncao(param1: string, param2: number): string {
     //Comandos da função
}
```

Parâmetros opcionais

Parâmetro obrigatório *Parâmetro opcional*

```
function nomeFuncao(param1: string, param2?: number) {
     //Comandos da função
}
```

Execução de funções

- **Funções são apenas declaradas.** Para serem executadas, precisam ser chamadas dentro de um arquivo TypeScript, fora de uma função (ou em outra função que seja chamada no corpo do arquivo)

Execução de comandos

```
/*
* Nome: logInutil
* Descrição: Registra uma mensagem importante no console
*/
function logInutil(): void {
     console.log("Esta função não é muito útil...");
}

//Chama a função
logInutil();
```

Problema de Exemplo

3. Crie uma função para calcular se um número é positivo, negativo ou zero e permita chama-la com um número digitado pelo usuário

Vetores

- Variável única usada para armazenar diferentes elementos
- Usado quando queremos armazenar uma lista de elementos e acessá-los por uma única variável

Declarando um vetor

- É possível declarar um vetor em TypeScript de duas formas

```
let vetor1: number[]; //Forma 1

let vetor2: Array<number>; //Forma 2
```

Inicializando um vetor (forma 1)

```
//Inicialização junto a declaração
const animais = ['leão', 'zebra',
'avestruz', 'macaco'];

//Inicialização após declaração
let animais: string[] = [];
animais[0] = 'leão';
animais[1] = 'zebra';
animais[2] = 'avestruz';
animais[3] = 'macaco';
```

Inicializando um vetor (forma 2)

```
//Inicialização junto a declaração
const animais = new Array('leão',
'zebra', 'avestruz', 'macaco');

//Inicialização após declaração
let animais: string[] = new Array();
animais[0] = 'leão';
animais[1] = 'zebra';
animais[2] = 'avestruz';
animais[3] = 'macaco';

//Cria um vetor com 5 elementos quaisquer
let animais = new Array<any>(5);
```

Acessando elementos

- Vetores são indexados partindo de 0
- Podemos acessar elementos da seguinte forma:

```
console.log( animais[0] );

const animal = animais[2];
console.log( animal );
```

Tamanho de vetores

- A propriedade “length” (comprimento) de um vetor retorna seu tamanho
- O length de um vetor é sempre +1 em relação a seu maior índice

```
console.log (animais.length);

const tamanhoAnimais = animais.length;
console.log(tamanhoAnimais);
```

Percorrendo vetores

- Utilizando o “length” e uma repetição, é possível percorrer vetores

```
const animais = ['leão', 'zebra', 'avestruz', 'macaco'];

const tamanho = animais.length;

for (let i = 0; i < tamanho; i++) {

    console.log('Animal ' + i + ': ' + animais[i]);

}
```

for...of (for especial)

- Permite percorrer um vetor através de uma declaração mais concisa

```
for (let elemento of vetor) {
      //Comando a repetir para cada item do
      //vetor, onde a variável elemento terá
      //o valor de cada item a cada iteração
}
```

for...of (for especial)

```
const animais = ['leão', 'zebra', 'avestruz', '
macaco'];

for (let animal of animais) {
    console.log("Animal: " + animal);
}
```

Problema de Exemplo

4. Leia valores numéricos do usuário enquanto este digitar elementos positivos ou zero. Quando um número negativo for digitado, exiba todos os números pares inseridos


"That's all Folks!"