CAPACITAÇÃO JAVASCRIPT & TYPESCRIPT

Sumário:

1. Extensões para VsCode

- Prettier
- ESLint
- Error Lens
- Auto Import
- Color Highlight
- vscode-styled-components
- Color Picker
- Material Icon Theme
- JavaScript (ES6) code snippets
- Rocketseat React Native
- Code Spell Checker
- Brazilian Portuguese - Code Spell Checker
- Tailwind CSS IntelliSense
- Console Ninja
- Git Lens - Git supercharged
- Code Runner

2. Overview Java Script

JavaScript (JS) é uma linguagem leve, interpretada e baseada em objetos com funções de primeira classe (funções que podem ser argumentos de outras funções). Utilizada frequentemente de script para páginas Web. É uma linguagem baseada em protótipos (permite a reutilização de objetos existentes), multi-paradigma e dinâmica, suportando estilos de orientação a objetos, imperativos e declarativos.

JS roda no *client side* da web, o que pode ser usado para projetar / programar o comportamento de uma página web a partir da ocorrência de um evento. JavaScript é uma linguagem fácil de se aprender mas que também é poderosa, sendo amplamente utilizada para controlar o comportamento de páginas web.

O ECMAScript é o padrão internacional no qual o JavaScript se baseia, e é mantido pela ECMA International. Enquanto isso, a escolha do nome "JavaScript" pela Netscape em 1995 foi parcialmente uma estratégia de marketing, aproveitando-se da popularidade crescente da linguagem Java na época. Embora os nomes sejam semelhantes, JavaScript não é Java!! Sintaxe básica é intencionalmente similar tanto a Java quanto a C++ (if, for, while, switch, try..catch).

Objetos são criados programaticamente em JavaScript, onde métodos e propriedades são anexados a objetos vazios em tempo de execução, ao invés das definições sintáticas de classe normalmente encontradas em linguagens compiladas como C++ e Java. Assim que um objeto é construído, ele pode ser usado como um esquema (ou protótipo) para se criar objetos similares.

As capacidades dinâmicas de JavaScript incluem a construção de objetos em tempo de execução, listas variáveis de parâmetros, variáveis de funções, criação dinâmica de scripts (através da função eval), introspecção de objetos (através da estrutura for ... in), e recuperação de código fonte (programas escritos em JavaScript podem descompilar funções de volta a seus textos originais).

3. Primeiro Hello World

```
console.log("Hello World!");
```

Essa expressão significa o primeiro contato do desenvolvedor com uma linguagem de programação. Acredita-se que o programador que não utiliza o termo possui longos anos de bugs pela frente.

O console.log() é capaz de imprimir uma frase (string), um número (number) ou qualquer outro tipo de variável no console, não exibe nada na tela. O que a torna excelente para debugar o código.

Para que o código em JS funcione é necessário a instalação do node.js, um sistema de tempo de execução que permite o desenvolvedor a criar servidores, aplicações web, algoritmos que leem inputs e interagem com o sistema operacional, e scripts.

Link de instalação: https://nodejs.org/en/download

4. Lógica de Programação em JS

4.1. Princípios de programação

4.1.1. O que é um algoritmo

Algoritmo é diretamente afetado pela linguagem de programação utilizada, mas ele não é ligado apenas a programação, na verdade ele é um sequência de passos que são necessários para executar determinada ação. Como exemplo, cada pessoa tem um algoritmo ao acordar até estar pronto para atividades do dia, tem aqueles que preferem 1° levantar, 2° lavar o rosto, 3° tomar café, 4° escovar os dentes e 5° arrumar a cama, tem outros que preferem escovar os dentes antes do café, e aqueles que nem arrumam a cama pois vão deitar de novo de noite. Mas, são nestas situações que se consegue observar o que é um algoritmo.

Além disso, também cabe ressaltar que um algoritmo pode conter repetições de tarefas, ou seja, no exemplo dado eu posso tomar café duas vezes, mas também, posso tomar café até acabar o sono. Assim, observa-se que pode-se haver repetições que ocorrem por um número determinado de vezes ou aquelas que acontecem até que determinada condição seja satisfeita.

Outra possibilidade em um algoritmo é a tomada de decisão, ou seja, posso decidir se vou executar algum passo ou não. Como no exemplo já citado, se a pessoa acordar sem fome, ela pode optar por não tomar o café, ou, como já dito, se estiver com ânimo, arruma a cama, se não, deixa sem arrumar mesmo.

Algo que deve-se atentar em um algoritmo é a ordem de execução dos passos, pois estes podem alterar o resultado. Imagine, no exemplo citado, se a pessoa lava o rosto antes de se levantar, isso resultaria no famoso balde de água na cara para acordar. E assim como essa situação não é muito desejada, os problemas gerados por passos em ordem errada trazem o conhecido "bug" para a programação.

Por fim, um último ponto importante a ressaltar de um algoritmo é que ele pode ter entradas, processamento (realizar o passo a passo) e liberar uma saída. Então, seguindo o exemplo, poderia-se considerar que a entrada é a pessoa ao acordar e dependendo de como for o processamento, ou seja, os passos, vai ter como saída uma pessoa feliz ou desanimada para o seu dia.

Como curiosidade deixo que os algoritmos podem ser representados de várias formas, sendo algumas delas:

- Fluxograma
- Linguagem Natural (Inglês, Português, Alemão, etc)
- Linguagem Artificial (Java Script, Java, C++, Python, etc)
- Pseudo-Linguagem

4.1.2. Visão geral de todas estruturas

Primeiramente é importante definir os dados, ou seja, são todas as informações que são utilizadas por um algoritmo, sejam estas, uma palavra, um número, um valor booleano, um objeto, etc. Desta forma, tem-se as Estruturas de Dados que existem para organizar e administrar os dados que são utilizados.

Desta forma algumas das estruturas de dados clássicas são:

- Lista (um conjunto de elementos que pode ter ordem ou não)
- Fila (FIFO - First in First out - primeiro a entrar é o primeiro a sair)
- Pilha (LIFO - Last in First out - último a entrar é o primeiro a sair)
- Árvore (como o sistemas de arquivos do computador)
- Tabelas (organizar os dados em linhas e colunas)

Também tem-se estruturas básica de dados para as linguagens de programação:

- int (números inteiros)
- real (números reais, ou seja, número com casas decimais)
- char (letras e símbolos)
- string (conjunto de letras, existe em apenas algumas linguagens)
- bool (valores lógicos, ou seja, verdadeiro ou falso)

Mas algumas linguagens, assim como o JS, não separam o int do real, tudo é number. E assim, já podemos observar que essas estruturas básicas são utilizadas para criar estruturas mais complexas dentro das linguagens, como exemplo na construção de objetos.

Outro ponto é que nas linguagens de programação os dados podem ser armazenados em variáveis ou constantes, que, como o nome indica, guardarão, respectivamente, valores que variam com o tempo e valores que são fixos.

Por fim, com os dados armazenados em variáveis e organizados pelas estruturas pode-se realizar diversas operações com eles, como as de atribuição, aritméticas, relacionais e lógicas.

4.1.3. O que é um bloco de código

Um bloco de código é definido por um par de chaves "{}", que têm a funcionalidade de separar algumas instruções específicas que compõem um grupo de um funcionalidade. Isso será muito importante para quando chegar na parte de funções, mas, por enquanto, tem-se um exemplo de dois blocos de código.

```
{
   console.log("Bloco de código 1");
}


{
   console.log("Bloco de código 2");
}
```

Vale ressaltar que pode-se encontrar um bloco dentro de outro, o que denominamos como blocos aninhados. Mas não há como ter uma intersecção entre blocos. Sendo assim, segue um exemplo de um bloco aninhado.

```
{
   console.log("Bloco externo");
   {
       console.log("Bloco interno");
   }
}
```

4.1.4. Comentário de um código

Aqui no Ex Machina prezamos pelo clean code, ou seja, um código que é de fácil entendimento só de olhar. Para isso, você deve tentar colocar os nomes de variáveis e de funções sempre da maneira mais clara e correlacionada possível.

Contudo, comentários podem ser feitos para fornecer uma melhor explicação da funcionalidade de um determinado bloco de código. Essa prática de comentários é altamente recomendada na fase de aprendizagem, pois você pode aproveitar deste recurso para anotar algo novo ou até um macete para guardar como utilizar o conteúdo aprendido.

Portanto, para fazer um comentário em JS basta colocar duas barras (//) para comentar somente uma linha e barra-asterisco e asterisco-barra (/* */) para comentar várias linhas. Veja como fazer no exemplo abaixo:

```
//esse é um comentário de uma linha


/*
   esse é um
```

```
   comentário
   de várias linhas
*/
```

OBS: existe um atalho para comentar uma linha ou várias linhas, geralmente basta selecionar as linhas desejadas depois utilizar a teclas CTRL + / (ou CTRL + :) para comentar as linhas desejadas.

4.2. Variáveis (let,var e const)

Variáveis são contêineres que armazenam algum valor. Deve-se declará-la ao colocar o prefixo e um nome. O nome da variável não pode utilizar palavras reservadas da linguagem (Exemplo: if, let, switch), não pode começar com números (Exemplo: 123ExMachiner) e também conter espaços ou traços. É de suma importância que o nome da variável seja significativo.

4.2.1. Let

Declara um tipo de variável que pode ter seu valor atribuído, com escopo de bloco. É preferível seu uso ao 'var'.

```
let nome = "Ex-Machina"; //String

let idade = 10; // Number

let ehValido = true; // Boolean

let lista = [1, 2, 4, 6, 7, 10, 11, 24]; // Array

let Ponto = {
  coordenadaX : 10,
  coordenadaY : 5,
}; // Objeto (vai ser explicadinho mais para frente)
```

4.2.2. Var

Declara um tipo de variável de escopo de função ou, caso esteja fora de uma, de escopo global.

```
var x = 1;

{
  var x = 2;
  console.log(x);
  // Expected output: 2
}

console.log(x);
// Expected output: 2
```

4.2.3. Const

Utilizado para declarar constantes imutáveis de escopo de bloco. Único método de modificar a constante é se ela for um objeto, cujas propriedades podem ser adicionadas, atualizadas e removidas, como será visto mais adiante.

```
const PI = 3.14

const Pessoa = {
    nome:"Jhon",
}
```

4.3. Tipos de dados primitivos (string, boolean, number)

No JavaScript existem diversos tipos de dados nas quais as variáveis podem assumir, como string e number.

4.3.1. String

Representa o formato de texto e deve ser sempre declarado entre aspas simples ou duplas.

```
let sobrenome = "Doe"; //String
```

4.3.2. Number

Representa o formato de números, no JS, diferentemente da linguagem C, não ocorre a diferenciação entre números inteiros e com ponto flutuante.

```
let altura = 1.8; // Number
let altura1 = 180; // Number
```

4.3.3. Boolean

Representa um estado lógico, verdadeiro ou falso (true or false)

```
let ehCasado = false; // Boolean
```

4.3.4. Null

Representa o nulo, não aponta para nenhum lugar na memória.

```
let partner = null; // NULL
```

4.3.5. Undefined

Assim como o nulo, não aponta para nenhum lugar na memória. Entretanto, diferentemente do ‘null’, não é intencional.

```
let profissao; // undefined (indefinido)
```

4.4. Estrutura de dados (array e objetos)

Como já explicitado anteriormente, as estruturas de dados servem para organizar as informações. Sendo assim, duas maneiras comumente utilizadas são o array e os objetos. O primeiro, também chamado de vetor, é uma coleção de dados que podem ser de qualquer tipo, até mesmo objeto ou outros vetores (matriz), e para sua definição utiliza-se os colchetes [], como nos exemplos abaixo.

```
const numeros = [1, 2, 3, 4, 5];
```

```
const letras = ['a', 'b', 'c', 'd', 'e'];
const palavras = ['Olá', 'mundo', 'do', 'JavaScript'];
```

Como pode-se observar, são vários elementos dentro de um vetor, sendo assim, para acessá-los basta colocar o nome do vetor e dentro dos colchetes a posição do elemento que você deseja acessar. É muito importante lembrar que a contagem começa em 0, então se você deseja recuperar o primeiro elemento do array números deve digitar: `numeros[0]`. Com esse comando você receberá como resposta o número 1 presente no vetor.

Esta notação do nome do vetor e colchetes com a posição também pode ser utilizada para trocar o valor em uma posição, como exemplo: `letras[2] = 'c'`. Além disso existe a função `push` para adicionar valores e a `pop` para retirar valores do fim do array. Também, cabe ressaltar que existem várias outras funções específicas para trabalhar com os arrays, elas serão especificadas na seção 4.8.2.

Já quando se trata de objetos diferencia-se um pouco da ideia dos arrays de juntar valores em posições específicas. Os objetos na verdade são uma maneira de você juntar informações, o que permite criar uma estrutura personalizada para o caso em que está trabalhando. Como exemplo um produto do site Extoque tem a seguinte estrutura de objeto:

```
const produto = {
    nome: "Arroz",
    categoria: "Alimento",
    descrição: "Arroz branco",
    validade: "2022-12-31",
    baixoEstoque: 10.0,
    preco: 25.0,
    quantidade: 15.0,
    unidade: "pacotes"
}
```

Assim, é possível notar que você pode usar a criatividade com gosto e fazer um objeto que atenda todas as necessidades, como ter dentro dele um array ou até mesmo outro objeto.

Para acessar um atributo do objeto, ou seja, uma de suas características, basta colocar nome do objeto, ponto e nome do atributo como: `produto.preco`. Outra opção é usar uma notação semelhante a do array: `produto["preco"]`. Então, essas duas sintaxe, assim como no array, podem ser usadas para inserir ou modificar valores.

Por fim, um ponto importante a se destacar sobre objetos é que eles não são a mesma coisa que um JSON. Pois, o JSON foi baseado no objeto do JS, mas ele é um formato textual que é usado para trocar dados entre sistemas distribuídos (sistemas na internet).

4.5. Operadores

Os operadores são símbolos que realizam operações em valores ou variáveis. Alguns exemplos comuns incluem: aritméticos, de atribuição, de desestruturação (destructing), relacionais, lógicos, unário e ternário.

4.5.1. Atribuição

O operador de atribuição (=) é usado para atribuir um valor a uma variável. Por exemplo:

```
const nome = "John Doe";
```

4.5.2. Destructing

O destructuring permite extrair valores de objetos ou arrays de forma concisa. Exemplo:

```
//Para arrays:
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
//Output: 10

console.log(b);
//Output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
//Output: [30, 40, 50]

//Para objetos:
const pessoa = { nome: 'Jhonny', idade: 16 };
console.log(pessoa.nome);
//Output: Jhonny
```

4.5.3. Aritméticos

Representam os operadores matemáticos, note que também é possível somar strings.

```
let a = 5, b = 10, c;
c = a + b; // Soma // Output: 15
c = a - b; // Subtração // Output: 5
c = a * b; // Multiplicação // Output: 50
c = a / b; // Divisão // Output: 2
c = a % b; // Resto da Divisão // Output: 0
```

```
let nome = "Nome", nome2 = "Sobrenome";
console.log(nome + " " + nome2); //Output: Nome Sobrenome
```

4.5.4. Relacionais

Os operadores relacionais comparam valores e retornam um resultado booleano (true ou false). Esses operadores são:

- > (maior)
- < (menor)
- >= (maior igual)
- <= (menor igual)
- == (igual)
- != (diferente)
- === (estritamente igual)

```
let e = 10,
 f = 15;

if (e > f) {
 //do stuff
}

if (e < f) {
 //do stuff
}

if (e == f) {
 //do stuff
}

if (e >= f) {
 //do stuff
}

if (e <= f) {
 //do stuff
}
```

O JavaScript faz conversões automáticas de tipos em algumas situações. Isso pode levar a resultados inesperados, como no exemplo abaixo. É importante entender a diferença entre == (comparação solta) e === (comparação estrita). A comparação solta (==) converte os operandos para o mesmo tipo antes de compará-los, enquanto a comparação estrita (===) não faz conversões de tipo.

```
console.log("10" == 10); //true (conversão de tipos)
```

```
console.log("10" === 10); //false (comparação estrita)
```

4.5.5. Lógicos

Os operadores lógicos representam a álgebra booleana. Operações como OU, E e NÃO. possuem símbolos próprios, sendo: '||' , '&&' e '!', respectivamente.

```
let idade = 25, num = 10;

if(idade >= 18 && idade <= 65){ // && = E, precisa atender ambas condições
      console.log("Você é um adulto!")
}

if(num == 0 || num >= 10 ){   // || = OU, apenas umas das condições é necessária para validar
      console.log("Alguma coisa")
}

if(!idade){   // ! = NÃO (NOT), nesse caso, verifica se idade é diferente de 0, NULL ou undefined
      console.log("Idade Inválida")
}
```

4.5.6. Unário e Ternário

4.5.6.1. Unário

Trata-se de uma operação com apenas um (1) operando, são apresentados como: operadores de incremento (++) e decremento (--), que podem ser posicionados antes ou depois da variável, produzindo efeitos diferentes. Também têm-se os de atribuição (+=, -=, *=, /=, %=).

Exemplo incremento e decremento:

```
let x = 10, y = 15, z;

z = --x + y; //Output: 24

z = x-- + y; //Output: 24
console.log(x); //Output: 8

z = ++x + y; //Output: 24

z = x++ + y; //Output: 24
console.log(x); //Output: 10
```

Exemplo atribuição:

```
let l = 10, m = 5;

l += m // Equivalente: l = l + m // Output: 15
l -= m // Equivalente: l = l - m // Output: 5
l *= m // Equivalente: l = l * m // Output: 50
l /= m // Equivalente: l = l / m// Output: 2
l %= m // Equivalente: l = l % m // Output: 0
```

4.5.6.2. *Ternário*

É comumente utilizado como um atalho para o 'if', avaliando uma expressão como verdadeira ou falsa.

Sintaxe: condição? resultado1 : resultado2

Exemplo:

```
console.log(idade > 18 ? "Você é um adulto!" : "Você é jovem!")
```

4.6. Estruturas de controle

Como já familiarizado anteriormente, estruturas agem no sentido de organizar e as estruturas de controle vão agir justamente na organização da execução do código, ou seja, qual passo vai seguir, assim como foi explicado na introdução a um algoritmo. Sendo assim, a seguir serão explicadas as principais estruturas utilizadas. Vale ressaltar que, assim como os blocos de códigos, elas podem ser aninhadas entre si ou com outras, visando atingir os objetivos desejáveis.

4.6.1. if

Estrutura para fazer o "se", ou seja, ela verifica se um condição é verdadeira para executar certa parte definida do código. Desta forma, utiliza fortemente os operadores relacionais e lógicos. Um exemplo é:

```
if (nota >= 7) {
  console.log("Aprovado emm, só sucesso! Passou com "+nota);
}
```

4.6.2. if else

Estrutura que complementa o if. Ou seja, se algo não for verdadeiro ela executará a parte do "se não". Sendo assim, complementando o exemplo acima:

```
if(nota >= 7){
  console.log("Aprovado emm, só sucesso! Passou com "+nota);
}else{
  console.log("Não foi dessa vez. Sua nota foi "+nota);
}
```

Mas vale ressaltar que pode-se ter uma sequência de ifs e elses, além de aninhar essas estruturas.

4.6.3. switch

Essa estrutura, assim como um if, testa se determinada condição é verdadeira para executar um pedaço de código. Contudo, ela tem uma entrada de qualquer tipo e casos. Para cada caso, ela verifica se a entrada é equivalente à opção que o caso fornece.

Se um caso for verdade ele será executado, senão vai procurar até achar o primeiro que se encaixe. Nota-se, então, que se tiver dois casos iguais apenas o superior será executado.

Para ficar mais fácil a seguir será apresentado um switch para avaliar a quantidade de estrelas fornecida por um usuário:

```
switch(qntEstrelas){
   case 1:
       console.log("Péssimo");
       break;
   case 2:
       console.log("Ruim");
       break;
   case 3:
       console.log("Regular");
       break;
   case 4:
       console.log("Bom");
       break;
   case 5:
       console.log("Excelente");
       break;
   default:
       console.log("Quantidade de estrelas inválida");
       break;
}
```

Como é possível notar o “:” indica o começo do caso e o “break” indica o final. Mas vale ressaltar que o break não é obrigatório e sua omissão pode ser utilizada para quando dois cases diferentes possuem o mesmo retorno, basta colocá-los em seguida e colocar no break apenas no case inferior.

Por fim, no exemplo acima pode-se ver que quando não encontra nenhuma opção para `qntEstrelas` temos o `default` que funciona justamente para suprir quando não há um caso válido. Também, vale pontuar que ele não é obrigatório, mas é uma boa prática de programação.

4.6.4. *while*

Essa estrutura encaixa-se na categoria de repetição, isso porque, assim como o seu nome indica, enquanto uma afirmação for verdadeira ela continuará a executar as instruções de código que se encontram em seu bloco.

Desta forma, ela também utiliza as expressões lógicas seguindo a mesma forma utilizada no if. Em seguida tem-se um exemplo:

```
while (sono === true){
   console.log("A mimir!");
   sono = Math.random() >= 0.5
}
```

Vale ressaltar que a instrução `Math.random() >= 0.5` apenas foi utilizada para sortear aleatoriamente um booleano. O que indica uma importante característica da estrutura while, nela geralmente não se sabe quantas vezes vai repetir, isso depende da expressão que está sendo testada.

Nesse caso, também poderia ter sido utilizada uma sintaxe mais compacta, como no exemplo abaixo.

```
while (sono){
   console.log("A mimir!");
   sono = Math.random() >= 0.5
}
```

Isso acontece porque a variável sono já possui um valor booleano, então a operação `sono === true` sempre terá o valor igual a própria variável.

Obs: essa maneira mais compacta também pode ser utilizada para outros tipos de variáveis (Number, String, etc), que passarão por uma conversão de tipos para Boolean, o que nem sempre será algo previsível porque JavaScript é estranho (e nem é meme).

4.6.5. *do while*

Essa estrutura também é uma dentre as de repetição, mas a sua diferença para o while explicado anteriormente é a verificação da expressão. Ou seja, no while primeiro verifica se a expressão é verdadeira para depois executar as instruções presente no bloco, já o do while é o oposto primeiro executa as instruções e depois verifica a expressão. Esse modelo inverso, garante que pelo menos uma vez ocorrerá a execução das instruções presentes neste bloco. A seguir tem-se o exemplo anterior modificado para o do while:

```
do{
   console.log("A mimir!");
   sono = Math.random() >= 0.5
}while (sono)
```

Então, aqui sabemos que pelo menos uma vez será impresso “A mimir!” o que não acontecia no exemplo anterior.

4.6.6. *for*

Essa estrutura também é uma repetição, mas diferente do while o seu número de repetições é conhecido antes dela iniciar. Ou seja, deve ser utilizada em contexto que o número de repetições é definido.

Esta estrutura possui uma sintaxe semelhante ao while, mas dentro dos parênteses ela apresenta 3 partes principais: sendo a 1° a definição da variável que servirá como um contador para as repetições, a 2° a expressão para verificar se a contagem já chegou ao ponto desejado e, por fim, a 3° a forma como a contagem será realizada.

A seguir será apresentado um exemplo de um for que realiza uma contagem de 1 a 10:

```
for(let i = 1; i <= 10;i++){
   console.log("i:"+i);
}
```

Como podemos observar a variável i é setada para 1 que indica de onde a contagem irá se iniciar e a expressão `i <= 10` verifica se já chegou ao limite que é 10. Já o `i++` é uma expressão para indicar que será acrescida uma unidade em i a cada repetição, utilizando o operador unário. Caso desejasse, por exemplo, contar de 2 em 2 elementos poderia utilizar `i+=2`.

4.6.7. *for in*

Essa estrutura assemelha-se ao for, ou seja, também é utilizada para fazer repetição. Contudo, ela tem como objetivo específico percorrer uma estrutura de dados, como um array ou um objeto.

A seguir um exemplo desta função para cada estrutura de dados citada:

```
const notas = [6.7, 7.4, 9.8, 8.1, 7.7];

for(let i in notas){
   console.log(i, notas[i]);
}

const pessoa ={
   nome: "Ana",
   sobrenome: "Silva",
   idade: 29,
   peso: 64
}

for(let atributo in pessoa){
   console.log(`${atributo} = ${pessoa[atributo]}`);
}
```

4.7. Clonando Objetos, Shallow Copy x Deep Copy

A clonagem nos permite criar versões independentes e isoladas, preservando a integridade dos dados e evitando efeitos colaterais indesejados. Essa prática é especialmente útil em situações em que desejamos modificar apenas uma cópia do objeto, enquanto mantemos o objeto original intacto.

4.7.1. Shallow Copy

Quando utilizamos o ... (spread operator) ou o Object.assign(), estamos efetuando uma shallow copy de um objeto. Isso significa que estamos criando um novo objeto com uma nova referência.

```
let livro = {
  titulo: "Os Miseráveis",
  genero: ["Romance","Drama","Épico","Tragédia","Ficção"],
  autor: "Victor Hugo",
}
console.log(livro)

const clone = {...livro};
console.log(clone)

const clone2 = Object.assign()
console.log(clone2)
```

Propriedades do tipo primitivo (primitive type), como por exemplo string e number, são copiadas gerando um novo valor, ocupando um novo espaço de memória, ou seja, são cópias independentes.

As propriedades do tipo referência (reference type), como arrays e objetos, tem suas referências copiadas, ou seja, não é feita uma cópia do objeto em si, mas sim do endereço de memória onde o objeto está armazenado. Portanto, as propriedades reference type do clone apontarão para o mesmo endereço das propriedades do objeto original, o que, se usado não intencionalmente, pode gerar diversos bugs no programa.

4.7.2. Deep Copy

Uma deep copy cria uma nova instância do objeto original e também cria novas instâncias de todos os objetos referenciados. Isso significa que todas as propriedades e membros do objeto original são copiados, não apenas as referências. Como resultado, as alterações feitas em um objeto não afetam o outro.

```
let livro2 = {
    titulo: "One Piece",
    genero: ["Aventura", "Fantasia", "Ficção"],
    autor: "Eiichiro Oda",
}
```

```
console.log(livro2)

const deepclone = structuredClone(livro2);
console.log(deepclone)
```

A única questão que deve ser analisada na deep copy, é sua demora na execução comparada com a shallow copy, dependendo do tamanho da estrutura a ser copiada.

4.8. Funções

Funções podem ser definidas simplesmente como um conjunto de instruções agrupadas em um bloco de código. Geralmente elas possuem um objetivo específico e podem ou não ter um retorno e parâmetros.

Outro ponto importante sobre funções é que no JS elas podem ser criadas de forma literal, armazenadas em um variável ou em um atributo de um objeto. Como nos exemplos a seguir:

```
//criar de forma literal - sempre retorna undefined por default
function fun1() { }

//armazenar em uma variável
const fun2 = function () { }

//armazenar em um atributo de objeto
const obj = {};
obj.falar = function () { return "Opa"; }
console.log(obj.falar());
```

Para executar (chamar ou invocar) as instruções internas de uma função, basta adicionar o nome da função e um par de parênteses, como foi possível observar no último exemplo: `obj.falar()`. Acredito que nesse momento você deve ter percebido que vem utilizando uma função desde o início desta capacitação e agora que se deu conta. E essa função que já é familiar para você, está armazenado em um atributo do console, ou seja, é a função log que faz todas as informações serem mostradas no terminal.

Outro ponto importante para as funções são seus parâmetros, ou seja, os valores que são passados para que seja possível realizar alguma operação interna, como exemplo clássico temos um função soma que recebe dois parâmetros e retorna o valor da soma entre eles:

```
const soma = function (x, y) {
    return x + y;
}
```

Por último, cabe ressaltar que foi possível observar pelos exemplos que algumas funções possuíam a palavra `return` e outras não. Isso significa que uma função pode

ou não ter um retorno, então no exemplo da função soma apresentada, ela retorna o valor da soma para onde ele é necessário (onde a função foi chamada), mas dependendo da aplicação ele poderia ser simplesmente imprimido por um `console.log()` e a função não retornaria nada.

4.8.1. Função arrow

Aqui no Ex Machina temos nosso jeito preferido de usar as funções do JS e ela é o arrow function. A seguir tem-se um exemplo de como uma função é transformada para uma funções arrow:

```
let dobroA = function (a) {
    return 2 * a;
}

//função arrow eh sempre anônima
//deve sempre ser associada a uma variável
const dobroB = (a) => {
    return 2 * a;
}

const dobroC = a => 2 * a; //return está implícito
```

Como pode-se perceber esse tipo de função tem menos elementos que a função tradicional, o que deixa mais agradável de programar e, também, como o costume mais fácil de entender.

Outro ponto importante para utilização de funções arrow é que funções podem receber funções como um parâmetro e devido ao formato reduzido fornecido pelas arrows elas se encaixam muito bem nesse papel, isso será demonstrado na próxima seção nas Funções do array.

4.8.2. Funções do array

No JavaScript, os arrays possuem diversas funções e métodos para facilitar o manuseio dessa estrutura, que geralmente substituem a necessidade da utilização de uma estrutura for para percorrer pelos itens. Destacamos algumas das funções mais importantes, sendo elas: ‘foreach’, ‘map’, ‘filter’, ‘reduce’.

4.8.2.1. *Foreach*

O método ‘forEach’ executa uma dada função para cada elemento de um array. Essa dada função, assim como os próximos exemplos, possui 3 parâmetros que podem ou não ser utilizados de acordo com a necessidade: o primeiro deles é o elemento atual que está sendo percorrido, o segundo é o índice desse elemento no vetor original e o terceiro é o próprio vetor original.

```
let vetor = [1, 2, 3];
vetor.forEach((elemento) => console.log(elemento));
```

```
let vetor = [1, 2, 3, 4, 5];
vetor.forEach((item, indice, vetor) => {
   console.log((item));
   console.log((indice));
   console.log((vetor));
})
```

4.8.2.2. *Map*

O método 'Map' invoca uma função (callback) passada por argumento para cada elemento do array e retorna um novo array como resultado.

```
let array = [1, 5, 6, 102]

const arrayDobro = array.map((elemento) => elemento*2)
console.log(arrayDobro)
// Output: [ 2, 10, 12, 204 ]
```

4.8.2.3. *Filter*

O método 'Filter' cria um novo array com todos os elementos que passaram no teste de verificação implementado pela função fornecida.

```
let numeros = [1,2,3,4,5,6,7,8]

const pares = numeros.filter((x) => x%2 == 0)
console.log(pares);
// Output: [ 2, 4, 6, 8 ]
```

4.8.2.4. *Reduce*

O método 'Reduce' executa uma função **reducer (**fornecida por você) para cada elemento do array, resultando em um único valor.

```
let LeagueofLegends = ["Nome1", "Nome2", "Nome3",
"Nome4"]

const LoL = LeagueofLegends.reduce((elemento, elemento1)
=> elemento + elemento1);
console.log(LoL)
```

4.9. Promises, Async e Await

Até o momento vimos códigos síncronos, ou seja, códigos que a linha de baixo só é executada depois que a linha de cima termina sua execução. Nesta seção trabalharemos com códigos assíncronos, ou seja, agora uma função é chamada e o código continua sendo executado, mas depois que a resposta chega a função que a chamou a processa. Isso acontece com muita frequência quando está trabalhando com requisições para um banco de dados você pede um dado e ele te retorna um promessa que vai processar a sua requisição e quando terminar ele te retorna a sua resposta.

Desta forma, as Promises vieram justamente para trabalhar com essas promessas de retorno para requisições. A seguir tem-se um exemplo simples de uma promise:

```
function falarDepoisDe(segundos, frase){
    return new Promise((resolve, reject) =>{
        setTimeout(() => {
            resolve(frase, 'abc')
        },segundos*1000)
    })
}

falarDepoisDe(3, 'Que legal!')
    .then((frase ,abc) => {
       console.log(abc)
       //undefined pq só aceita um parametro msm mandando 2
       return frase.concat("?!?")})
    .then(outraFrase => console.log(outraFrase)) //chama a vontade
    .catch(e => console.log(e))
    //para tratar erros quando usa o reject, mas chama uma vez só
```

Este exemplo utiliza a função `setTimeout` para simular a demora por uma resposta, visto que a função só retorna algo depois de passar o tempo definido. Sendo assim, a função falarDepoisDe retorna uma Promise, que tem duas funções internas para tratar o resultado, se o valor requisitado não tiver problemas a função `resolve` será executada, se ocorrer algum erro durante a espera a função `reject` que será executada.

Já sobre a chamada da função percebe-se que segue o mesmo padrão das funções síncronas, a diferença consiste no `then` que passa justamente a função resolve que a promise irá executar. Sobre o then pode-se notar que ele pode ser chamado quantas vezes necessárias, mas que só aceita um parâmetro por vez. Já a outra diferença é o `catch`, o qual se relaciona a função reject da promise para tratar os erros.

Contudo, a utilização de promisses com then e catch, além de ser muito verbosa é complexa de se trabalhar. Por isso, o mais comum e o que utilizamos no Ex é a dupla famosa Async e Await. A seguir tem-se um exemplo de como eles trabalham como promises:

```
const esperarPor = (tempo = 2000) => {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log('Executando promise...')
            resolve()
        }, tempo)
    })
}
```

```
//por trás do async e do await sempre tem que ter um promise
const executar = async () => {
   await  esperarPor(1500)
   console.log('Async/Await 1...')

   await  esperarPor(1500)
   console.log('Async/Await 2...')

   await  esperarPor(1500)
   console.log('Async/Await 3...')
}

//se retornasse um valor e quisesse usá-lo tem que usar o then pq
ela retorna uma promise
executar()
```

Como pode-se perceber, o exemplo é bem semelhante ao anterior, mas dessa vez o Async e Await servem para tratar uma promise como se ela fosse um função síncrona, ou seja, primeiro espera-se o retorno da função para depois continuar a executar o código.

Desta forma, como é possível notar, para usar o async e o await, basta criar uma função que seja assíncrona colocando o async antes do nome da função, como demonstrado no exemplo. E, para esperar a resposta, basta colocar o await na frente da chamada de uma função que retorna uma promise.

Por fim, não necessita o then já que o await faz o papel dele e o catch continua sendo opcional como era anteriormente, a diferença é que ele é feito por meio do tratamento de erros, como será explicado na seção 5.8.

5. Type Script

Agora que você viu vários aspectos do JS, viu como essa é uma linguagem poderosa e que pode atender aos requisitos de projetos. Contudo, como você deve ter reparado em nenhum momento quando as variáveis eram declaradas, tinha um tipo definido para elas, como por exemplo um “nome” poderia ser do tipo string já que sabemos que um nome é um conjunto de letras.

Também cabe ressaltar que as operações do JS podem ou não depender do tipo, como foi explicitado na seção 4.7. Desta maneira, a não declaração de tipos pode ser uma grande fonte para bugs e um dos maiores indicadores das pesquisas científicas que indicam as dores cabeças dos pobres devs.

Sendo assim, o TypeScript (TS) veio ser a nossa dipirona, ele é responsável por tipar as variáveis, ou seja, garantir que um valor esperado em algum lugar do código realmente será do tipo que o desenvolvedor espera. Sendo assim, a sintaxe básica para tipar uma variável é nomeVariavel: tipo.

Então nas próximas seções será apresentado como o TS é utilizado, você perceberá que é a base do JS, com a diferença de adicionar tipos. E é ele que iremos utilizar em nossos desenvolvimento no Ex Machina.

5.1. Tipos de dados básico

Primeiramente serão apresentados como o TS funciona para os tipos primitivos que já foram familiarizados nas seções do JS.

5.1.1. Number

Representa valores numéricos, inteiros ou de ponto flutuante.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

5.1.2. String

Representa valores de texto.

```
let color: string = "blue";
color = 'red';
```

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}.
  I'll be ${age + 1} years old next month.`;
```

```
let sentence: string =
    "Hello, my name is " +
    fullName +
    ".\n\n" +
    "I'll be " +
    (age + 1) +
    " years old next month.";
```

5.1.3. Array

Representa uma lista de valores de um determinado tipo.

```
let numeros: number[] = [1, 2, 3, 4, 5];
```

5.1.4. Tuple

Representa uma lista ordenada de elementos com tipos específicos.

```
let pessoa: [string, number] = ["Xuxa", 60];
```

5.1.5. Enum

É uma forma de criar um conjunto de valores nomeados. Representa valores enumerados.

```
enum rifa {
   "bicicleta",
   "doce de leite",
   "doce de abobora"
}

enum DiaDaSemana {
    Segunda,
    Terca,
    Quarta,
    Quinta,
    Sexta,
    Sabado,
    Domingo
}

let hoje: DiaDaSemana = DiaDaSemana.Segunda;
```

5.1.6. Boolean

Representa valores verdadeiros ou falsos.

```
let ativo: boolean = true;
```

5.1.7. Null e Undefined

Null indica o valor nulo e undefined indica o valor indefinido.

```
let u: undefined = undefined;
let n: null = null;
```

5.1.8. Any

Permite que uma variável tenha qualquer tipo. Seria o “tipo padrão”/do JS, portanto não faz tanto sentido ser usado.

```
let variavelQualquer: any = "Isso pode ser qualquer coisa";
```

5.1.9. Void

Usado para indicar que uma função não retorna nenhum valor.

```
function mostrarMensagem(): void {
   console.log("Olá, mundo!");
}
```

5.1.10. Object

Tipo `object` em TypeScript é uma representação genérica de qualquer valor não primitivo. Embora seja flexível, você deve ter cuidado ao usá-lo, pois perde informações detalhadas sobre a estrutura do objeto, o que pode levar a erros de tempo de execução. É geralmente preferível usar tipos mais específicos sempre que possível para garantir maior segurança de tipos em seu código. Exemplo:

```
let objeto: Object;

objeto = { nome: "Alice" }; // Válido
objeto = [1, 2, 3];         // Válido
objeto = "Olá";              // Válido
objeto = 42;                 // Válido
```

No entanto, você não pode fazer o seguinte sem uma verificação de tipo: objeto.nome;. Isso resultará em um erro: “Erro, propriedade 'nome' não existe em tipo 'object”. Ou seja, a flexibilidade oferecida possibilita bugs.

Para evitar esses bugs e acessar propriedades ou métodos em um objeto com tipo `object`, você pode usar a verificação de tipo ou realizar uma conversão de tipo, como exemplificado a seguir:

```
let objeto: object = { nome: "Alice" };

// Verificação de tipo
if ("nome" in objeto) {
   console.log(objeto["nome"]); // "Alice"
}

// Conversão de tipo
let objetoConvertido = objeto as { nome: string };
console.log(objetoConvertido.nome); // "Alice"
```

5.2. Inferência de Tipos

Uma das vantagens do TypeScript é sua capacidade de inferir tipos automaticamente, o que significa que você não precisa especificar o tipo de variável o tempo todo. Por exemplo:

```
let numero = 42; // Infere o tipo "number" automaticamente
let mensagem = "Olá, TypeScript!"; //Infere "string" sozinho
```

O TypeScript usa a inferência de tipos para determinar o tipo das variáveis com base no valor atribuído a elas. Isso torna o código mais limpo e legível, mas você ainda pode especificar tipos manualmente quando necessário.

Além dos tipos básicos, o TypeScript suporta tipos mais avançados, como uniões, interseções, tipos genéricos e tipos personalizados. Isso permite criar sistemas de tipos

complexos para atender às necessidades específicas do seu projeto, garantindo maior segurança e confiabilidade em seu código TypeScript.

5.3. Union Types

Em Typescript é possível ter mais de um tipo de retorno, como por exemplo:

```
function isNumber(value: string | number) {
   if (typeof value === "number") return true;
   else return false;
}
```

O uso de **Union Types** é comum ao se trabalhar com eventos no **DOM** (HTML), pois o código **Typescript** não tem acesso prévio ao **DOM**, não sendo assim capaz de identificar se esse elemento existe ou não, por isso null é comum como um tipo de retorno.

Um exemplo disso é que ao declarar um **querySelector** os tipos que aparecem no próprio editor de texto são ou um **HTMLButtonElement** ou **null**. Uma outra maneira de realizar isso é através de um Optional chaining (?.). Usando essa ferramenta, ao invés de um retorno nulo que joga um erro é retornado **undefined**, nos casos que seriam **null**.

5.4. Funções em TypeScript

Nesta seção será apresentado os detalhes de como implementar funções em TypeScript e você poderá observar as diferenças e semelhanças com relação ao JavaScript.

5.4.1. Declaração de Funções

Em TypeScript, você pode declarar funções de várias maneiras. A maneira mais comum é usar a sintaxe de função, onde você fornece um nome de função seguido pelos parâmetros e pelo corpo da função:

```
function digaOla(nome: string): void {
   console.log(`Olá, ${nome}!`);
}
```

Você também pode usar as arrow functions:

```
const digaOla = (nome: string): void => {
   console.log(`Olá, ${nome}!`);
};
```

5.4.2. Parâmetros e Tipos

Em TypeScript, você pode especificar os tipos de parâmetros em uma função para melhorar a segurança de tipos. Isso ajuda a evitar que erros de tipo ocorram durante a execução.

```
function adicionar(a: number, b: number): number {
   return a + b;
}
```

Neste exemplo, a função `adicionar` aceita dois parâmetros, ambos do tipo `number`. O TypeScript irá verificar se os argumentos passados para a função são do tipo correto.

5.4.3. Valor de Retorno

Você pode especificar o tipo de valor de retorno de uma função usando `: TipoDeRetorno`. Isso ajuda a indicar qual tipo de valor a função deve retornar.

```
function soma(a: number, b: number): number {
    return a + b;
}
```

Neste caso, a função `soma` deve retornar um valor do tipo `number`.

5.5. Interfaces

5.5.1. Criando e usando interfaces

As interfaces são criadas usando a palavra-chave `interface` e especificam a forma que um objeto deve ter. Você pode usá-las para definir tipos personalizados que são aplicados a variáveis, parâmetros de função ou valores de retorno.

```
interface EstudanteUNIFEI {
    nome: string;
    matricula: number;
    curso: string;
    anoIngresso: number;
}

const estudante: EstudanteUNIFEI = {
    nome: "João",
    matricula: 123456,
    curso: "Engenharia da Computação",
    anoIngresso: 2020
};
```

5.5.2. Extensão de tipos com interfaces

Assim como as classes podem herdar de outras classes, as interfaces podem herdar de outras interfaces. Isso permite que você compartilhe estruturas comuns entre diferentes tipos de objetos.

```
interface Pessoa {
    nome: string;
    idade: number;
}
```

```
interface Funcionario {
    cargo: string;
    salario: number;
}

interface Gerente extends Pessoa, Funcionario {
    departamento: string;
}

const gerente: Gerente = {
    nome: "João",
    idade: 35,
    cargo: "Gerente de Projetos",
    salario: 75000,
    departamento: "Desenvolvimento"
};
```

Você pode estender tipos existentes usando interfaces. Isso é útil quando você precisa adicionar novas propriedades a um tipo existente sem modificar o código original.

```
interface EstudanteUNIFEI {
    nome: string;
    matricula: number;
    curso: string;
    anoIngresso: number;
}

interface ExMachina extends EstudanteUNIFEI {
    anoIngressoProjeto: number;
    anoSaidaProjeto: number;
}

const AlunoDoExMachina: ExMachina = {
    nome: "Ronaldo",
    matricula: 987654,
    curso: "Engenharia Elétrica",
    anoIngresso: 2019,
    anoIngressoProjeto: 2022,
    anoSaidaProjeto: 2023,
};
```

5.6. Type Aliases

Assim como as interfaces, o Type Aliases é utilizado para definição de tipos. A documentação do TypeScript deixa explícito que escolher entre type aliases e interfaces é algo muito relacionado ao gosto pessoal.

Contudo aqui no Ex Machina estamos mais acostumados a trabalhar com o Type Aliases, visto que ele é mais utilizado para tipagem do front-end, enquanto as interfaces são recomendadas para o back-end para trabalhar com objetos node.

Sendo assim, nas próximas subseções serão replicados os exemplos anteriores para ter-se uma comparação clara entre type aliases e interfaces.

5.6.1. Criando e usando type aliases

Para criar um tipo a sintaxe é parecida com o interface, primeiramente tem-se a palavra reservada type seguida do nome do tipo e da atribuição, como no exemplo a seguir:

```
type EstudanteUNIFEI = {
   nome: string,
   matricula: number,
   curso: string,
   anoIngresso: number,
}

const estudante: EstudanteUNIFEI = {
   nome: "João",
   matricula: 123456,
   curso: "Engenharia da Computação",
   anoIngresso: 2020
}
```

5.6.2. Extensão de tipos com type aliases

Os types permitem estender de um tipo só ou de vários por meio da notação &, como no exemplo abaixo:

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

type Funcionario = {
   cargo: string;
   salario: number;
}

type Gerente = Pessoa & Funcionario & {
   departamento: string;
}
```

```
const gerente: Gerente = {
    nome: "João",
    idade: 35,
    cargo: "Gerente de Projetos",
    salario: 75000,
    departamento: "Desenvolvimento"
}
```

E como pode-se perceber, foi possível adicionar novos atributos ao tipo que foi criado por meio da extensão.

5.7. Genéricos

Os genéricos no TypeScript permitem que você escreva código que pode funcionar com uma variedade de tipos, mantendo a segurança de tipos. Eles são especialmente úteis para criar funções e classes flexíveis que podem ser usadas com diferentes tipos de dados.

5.7.1. Introdução aos genéricos

Genéricos são introduzidos usando parâmetros de tipo, que são especificados entre **<** e **>**. Eles podem ser usados em funções, classes e interfaces para tornar o código mais reutilizável e genérico.

```
function qualquerValor<T>(valor: T): T {
    return valor;
}

// Uso da função identidade com diferentes tipos de dados
const numero: number = qualquerValor(42); // retorna 42
const texto: string = qualquerValor("Olá, TypeScript!");
// retorna "Olá, TypeScript!"

const array: number[] = qualquerValor([1, 2, 3]);
// retorna [1, 2, 3]
```

5.7.2. Usando genéricos em funções e classes

Você pode criar funções genéricas que funcionam com uma ampla gama de tipos de entrada. Além disso, você pode criar classes genéricas que aceitam tipos personalizados como argumentos.

5.8. Tratamento de erros

Serão apresentados conceitos importantes para lidar com exceções e erros em TypeScript, permitindo que você crie um código mais robusto e seguro.

5.8.1. Lidando com exceções em TypeScript

O TypeScript oferece suporte ao tratamento de exceções usando as construções `try`, `catch` e `throw`, assim como em outras linguagens de programação. Aqui está um exemplo:

```
try {
  //Código que pode gerar um erro
  const resultado = 10 / 0; //Tentando dividir por zero
  console.log(resultado);   //Esta linha nunca será executada
} catch (erro) {
   // Capturando e lidando com o erro
   console.error("Ocorreu um erro:", erro);
} finally {
   // Bloco opcional que sempre é executado
   console.log("Execução concluída.");
}
```

Neste exemplo, tentou-se fazer uma divisão por zero, o que resulta em um erro de tempo de execução. O bloco `try` tenta executar o código, e se ocorrer um erro, o bloco `catch` é executado para lidar com a exceção.

5.8.2. Tipos de Erro

No TypeScript, você pode criar tipos de erros personalizados estendendo a classe `Error` ou criando suas próprias classes de erro. Isso é útil para identificar erros específicos do domínio e tratá-los adequadamente. Aqui está um exemplo:

```
class ErroDaProtese extends Error {
   constructor() {
       super("Erro da Protese Mãozinha");
       this.name = "ErroDaProtese";
   }
}

try {
   throw new ErroDaProtese();
} catch (erro) {
   if (erro instanceof ErroDaProtese) {
    console.error("Erro capturado:", erro.message);
   } else {
       console.error("Outro erro ocorreu:", erro);
   }
}
```

Neste exemplo, criou-se uma classe de erro personalizado chamada `ErroDaProtese` que estende a classe **`Error`**. Em seguida, lançou uma instância

desse erro e a capturou no bloco `catch`. Foi usada a `instanceof` para verificar se o erro capturado é uma instância do erro personalizado e, em seguida, tratou o erro de acordo.

6. Vídeos de conteúdo complementar

- JavaScript (A linguagem mais AMADA e/ou ODIADA 😁) // Dicionário do Programador
- Curso de JavaScript e TypeScript do básico ao avançado JS/TS
  Utilize o notion do projeto para obter o login de acesso
- Desvendando DEFINITIVAMENTE as Promises em JavaScript // Mão no Código #21
- Async / Await SIMPLES e DESCOMPLICADO no JavaScript // Mão no Código #22
- Curso Web Moderno Completo com JavaScript + Projetos
  Utilize o notion do projeto para obter o login de acesso
- JavaScript Funcional e Reativo - PENSE como um Dev JS
  Utilize o notion do projeto para obter o login de acesso
- TypeScript // Dicionário do Programador
- VOU APRENDER TYPESCRIPT (3 motivos)
- TypeScript - O que é e quais os seus benefícios? | Diego Fernandes

7. Referências Bibliográficas

CÓDIGO FONTE TV. Programação Orientada a Objetos: Classes e Objetos em TypeScript. [S.l.], [s.d.]. 1 vídeo (38min20s). Publicado pelo canal CódigoFonteTV em 11 de setembro de 2020. Disponível em: https://www.youtube.com/watch?v=gmupEp468lY&ab_channel=C%C3%B3digoFonteTV. Acesso em: 04 maio 2024.

CONSOLE.LOG. Clonando Objetos JavaScript: Shallow vs Deep Copy. Console.log, [S.l.], [s.d.]. Disponível em: https://consolelog.com.br/clonando-objetos-javascript-shallow-vs-deep-copy/. Acesso em: 04 maio 2024.

ESTEVAM, Vinicius. Principais diferenças entre Types e Interfaces em TypeScript. Medium, [S.l.], [s.d.]. Disponível em: https://viniciusestevam.medium.com/principais-diferen%C3%A7as-entre-types-e-interfaces-em-typescript-a00c945e5357. Acesso em: 04 maio 2024.

Mozilla Developer Network. JavaScript. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript. Acesso em: 04 maio 2024.

OPENAI. ChatGPT. [S.l.], [s.d.]. Inteligência Artificial. Disponível em: https://openai.com/chatgpt. Acesso em: 04 maio 2024.

RABELO, Eduardo. TypeScript: Entendendo a Notação de Tipos. Medium, 2024. Disponível em: https://oieduardorabelo.medium.com/typescript-entendendo-a-nota%C3%A7%C3%A3o-de-tipos-9e8c1c89ef62. Acesso em: 04 maio 2024.

SILVESTRE, Gabriel. Heranças e Interfaces. Dev.to, [S.l.], [s.d.]. Disponível em: https://dev.to/gabrielhsilvestre/herancas-e-interfaces-k0. Acesso em: 04 maio 2024.

TECNOBLOG. O que é TypeScript? Guia para Iniciantes. Tecnoblog, [S.l.], [s.d.]. Disponível em: https://tecnoblog.net/responde/o-que-e-typescript-guia-para-iniciantes/. Acesso em: 04 maio 2024.

TYPESCRIPT. Basic Types. TypeScript Documentation, [S.l.], [s.d.]. Disponível em: https://www.typescriptlang.org/docs/handbook/basic-types.html. Acesso em: 04 maio 2024.

TYPESCRIPT. Functions. TypeScript Documentation, [S.l.], [s.d.]. Disponível em: https://www.typescriptlang.org/docs/handbook/functions.html. Acesso em: 04 maio 2024.

Udemy. Curso de JavaScript Moderno do Básico ao Avançado. Disponível em: https://www.udemy.com/course/curso-de-javascript-moderno-do-basico-ao-avancado/learn/lecture/16331758?start=0#overview. Acesso em: 04 maio 2024.

Udemy. Curso Web: HTML, CSS, JavaScript, jQuery, Bootstrap, PHP, MySQL. Disponível em: https://www.udemy.com/course/curso-web/. Acesso em: 04 maio 2024.