Typescript

next-gen JavaScript

let & const

`let` : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

`const` : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
`let` e `const` basicamente substituem `var` . Você usa `let` no lugar de `var` e `const` no lugar de `var` se você não vai alterar ou re-atribuir o valor da variável (efetivamente ira se tornar uma constante).

Type (Tipos)

Tipado estático opcional, mas recomendado.
Sintaxe post-fix :T

```
let soma: number;
let cidade: string = 'São Paulo';
```

O tipo do retorno da função pode ser inferido.

```
function somar(a: number, b: number) {
  return a + b; //retorna :number
}
```

Aceita tipos opcionais com o símbolo `?`

Tipos primitivos

```
number
bool
string
null
undefined
```

Quando atribuímos as variáveis a outras variáveis usando =, uma COPIA do valor é atribuída para a nova variável.
Tipos primitivos são copiados por valor

```
let x: number = 10;
let y: string = 'abc';
let a = x;
let b = y;
console.log(x, y, a, b); // -> 10, 'abc', 10, 'abc'
```

Ambos `a` e `x` agora contém 10. Ambos `b` e `y` agora contém 'abc'. Os valores foram copiados, eles estão separados

Variables	Values
x	10
y	'abc'
a	10
b	'abc'

Alterar uma variável, não ira alterar a outra. Elas não têm relação umas com outras.

```
let x: number = 10;
let y: string = 'abc';
let a = x;
let b = y;

a = 5; //alterando o valor de a
b = 'def'; //alterando o valor de b
console.log(x, y, a, b); // -> 10, 'abc', 5, 'def'
```

Tipos Objeto

Podem ser classe, interface, arrays [ ]

```
let funcionario: Pessoa;
let funcionários : Pessoa[] = [];
```

Variáveis atribuídas com valores não primitivos contêm uma *referência* a esse valor. Essa referencia aponta ao endereço de objeto em memória. As variáveis não contem um 'valor' de verdade.

```
let arr: number[] = [];
```

Variables	Values	Addresses	Objects
arr	<#001>	#001	[]

```
arr.push(1);
```

Variables	Values	Addresses	Objects
arr	<#001>	#001	[1]

O valor, o endereço armazenado pela variável arr é estático. O que muda é o array em memória.

Quando um objeto é atribuído para outra variável usando =, o que é copiado é o endereço desse valor. **Objetos são copiados por referência**.

```
let reference: number[] = [1];
let refCopy = reference;
```

Variables	Values	Addresses	Objects
reference	<#001>	#001	[1]
refCopy	<#001>		

Cada variável agora contem uma referência ao mesmo array. Isso significa que se alterarmos reference, refCopy também será alterado:

```
reference.push(2);
console.log(reference, refCopy); // -> [1, 2], [1, 2]
```

Variables	Values		Addresses	Objects
reference	<#001>		#001	[1, 2]
refCopy	<#001>			

Outro exemplo

```
let funcionario: Pessoa = new Pessoa();
funcionario.nome = 'Max';

let funcionarioCopy = funcionario;
funcionarioCopy.nome = 'Ben';

console.log(funcionario.nome); //Ben
```

ES6 Arrow Functions (Funções seta)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

São outra forma de escrever funções em javascript.
Manter escopo com a palavra `this`.
Exemplo.

```
function callMe(name) {
  console.log(name);
}
```

Pode ser escrito como:

```
const callMe = function(name) {
  console.log(name);
}
```

se converte em:

```
const callMe = (name) => {
  console.log(name);
}
```

Importante:

Quando **não existem argumentos**, é preciso usar parêntesis vazios:

```
const callMe = () => {
  console.log('Max!');
}
```

Quando **existe só um argumento**, você pode omitir os parêntesis:

```
const callMe = name => {
  console.log(name);
}
```

Quando a função **somente retorna um valor (uma instrução)**, pode ser usada a sintaxe:

```
const returnMe = name => name
```

Equivale a:

```
const returnMe = name => {
  return name;
}
```

Objeto Javascript

Pode ser representado com a sintaxe. (par chave: valor)

```
let contato = {
  nome: 'Ana',
  telefone: '11958521452',
  principal: true
}
Console.log(contato.nome); //prints Ana
```

Classes

São abstrações para objetos JavaScript.
Ex:

```
class Person {
  name: string;

  constructor () {
    this.name = 'Max';
  }
}

const person = new Person();
console.log(person.name);  // prints 'Max'
```

Também é possível declarar métodos (funções):

```
class Person {
  name: string = 'Max';
  printMyName () {
    console.log(this.name); // this para referir a classe!
  }
}

const person = new Person();
person.printMyName();
```

Ou:

```
class Person {
  name: string = 'Max';
  printMyName = () => {
     console.log(this.name);
  }
}

const person = new Person();
person.printMyName();
```

Também pode ser usada **herança**:

```
class Human {
  species: string = 'human';
}
```

```
class Person extends Human {
  name: string = 'Max';
  printMyName = () => {
    console.log(this.name);
  }
}

const person = new Person();
person.printMyName();
console.log(person.species); // prints 'human'
```

Operadores

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators

Operadores de atribuição

O operador de atribuição básico é o "igual" (=), que atribui um valor da direita para o operando (variável) que está a sua esquerda. Dessa forma: x = y está ocorrendo a atribuição do valor de y para x. Os operadores de atribuição são geralmente utilizados em operações padrões como mostrado nos exemplos a seguir.

Nome	Shorthand operator	Significado
Atribuição	`x = y`	`x = y`
Atribuição com adição	`x += y`	`x = x + y`
Atribuição com subtração	`x -= y`	`x = x - y`
Atribuição com multiplicação	`x *= y`	`x = x * y`
Atribuição com divisão	`x /= y`	`x = x / y`
Atribuição com resto	`x %= y`	`x = x % y`
Atribuição de exponenciação	`x **= y`	`x = x ** y`
Left shift assignment	`x <<= y`	`x = x << y`
Right shift assignment	`x >>= y`	`x = x >> y`
Unsigned right shift assignment	`x >>>= y`	`x = x >>> y`
Bitwise AND atribuição	`x &= y`	`x = x & y`

Bitwise XOR atribuição	`x ^= y`	`x = x ^ y`
Bitwise OR atribuição	`x |= y`	`x = x | y`

Operadores lógicos

Operadores lógicos são tipicamente usados com valores Booleanos (lógicos).

Operador	Utilização	Descrição
Logical AND (`&&`)	*`expr1`* `&&` *`expr2`*	Retorna `expr1` se essa pode ser convertido para `falso`; senão, retorna `expr2`. Dessa forma, quando usado para valores Booleanos, `&&` retorna `verdadeiro` se ambos os operandos forem `verdadeiro` ; senão, retorna `falso`.
Logical OR (`||`)	*`expr1`* `||` *`expr2`*	Retorna `expr1` se essa pode ser convertido para `verdadeiro`; senão, retorna `expr2`. Dessa forma, quando usado para valores Booleanos, `||` retorna `verdadeiro` se qualquer dos operandos for `verdadeiro`; se ambos são `falso`, retorna `falso`.
Logical NOT (`!`)	`!`*`expr`*	Retorna `falso` se o seu operando pode ser convertido para verdadeiro; senão, retorna `verdadeiro`.

Precedência de operadores

operação	operador
multiplicação / divisão / resto ou módulo	* / %
adição / subtração	+ -
relacional	< <= > >=
igualdade	== != === !==
E	&&
OU	||

Controle de Fluxo

if..else

```
if (condicao) {
  declaracao_1;
} else {
  declaracao_2;
}
```

switch

```
switch (expressao) {
   case rotulo_1:
      declaracoes_1
      [break;]
   case rotulo_2:
      declaracoes_2
      [break;]
   ...
   default:
      declaracoes_padrao
      [break;]
}
```

ex.

```
switch (tipoEndereco) {
   case 'R':
      console.log('Residencial');
      break;
   case 'C':
      console.log('Comercial');
      break;
   default:
      console.log('Outro');
}
```

Laços

For

```
for ([expressaoInicial]; [condicao]; [incremento])
  declaração
```

ex.

```
let arr: number[] = [1,3,5,7,9];

for (let i=0; i < arr.length; i++){
  console.log(arr[i]);
}
```

Foreach

```
arr.forEach(a => {
  console.log(a);
});
```

Do...while

do
  declaracao
while (condicao);

ex.

```
let i: number = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

While

```
while (condicao)
  declaração
```

ex.

```
let n: number = 0;
let x: number = 0;
while (n < 3) {
  n++;
  x += n;
}
```

Continue (Reiniciar um laço)
ex.

```
i = 0;
n = 0;
while (i < 5) {
  i++;
  if (i == 3) {
    continue;
  }
  n += i;
}
```

Break (Terminar um laço)
ex.

```
for (i = 0; i < a.length; i++) {
  if (a[i] == theValue) {
    break;
  }
}
```

Exports & Imports

O código pode ser dividido em muitos arquivos JavaScript também chamados módulos.
Essa prática permite manter cada módulo/arquivo focado em suas tarefas e fácil de manter.
Para o acesso as funcionalidades, são usadas as palavras `export`
(disponibiliza o módulo) e `import` (para acessar)
Existem dois tipos de exports: **default** (sem-nome) e exports **nomeados**:

default => `export default ...;`
nomeado => `export const someData = ...;`

É possível importar **default exports**:

```
import someNameOfYourChoice from './path/to/file.js';
```

O nome, `someNameOfYourChoice` pode ser escolhido por você.

Exports nomeados precisam ser importados pelo seu nome:

```
import { someData } from
'./path/to/file.js';
```

Um arquivo pode conter somente um export default e um ou mais exports nomeados (podem existir os dois no mesmo arquivo).

Quando são importados **exports nomeados**, você pode importar todos eles de uma vez com a sintaxe:

```
import * as qualquerNome from
'./path/to/file.js';
```

`qualquerNome` é usado para acessar objetos dentro da classe importada, ex:

`qualquerNome.Objeto` .