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
```

Tipos Objeto

Podem ser classe, interface, arrays [ ]

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

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'
```

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` .