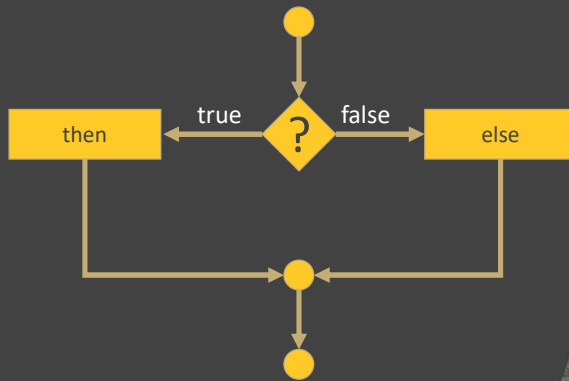
A large yellow square containing the letters 'JS' in a bold, dark grey sans-serif font. The square is positioned in the center of the slide, overlapping a white background on the left and a dark grey background on the right.

JS

@vtaquette  
@rzuquim



## controle de fluxo | condicionais



Corriqueiramente em qualquer programa precisamos executar (ou não) uma rotina dada uma condicional.

JS nos disponibiliza as seguintes construções para fazermos essa escolha:

**if / else**

**switch**

**? (operador ternário)**

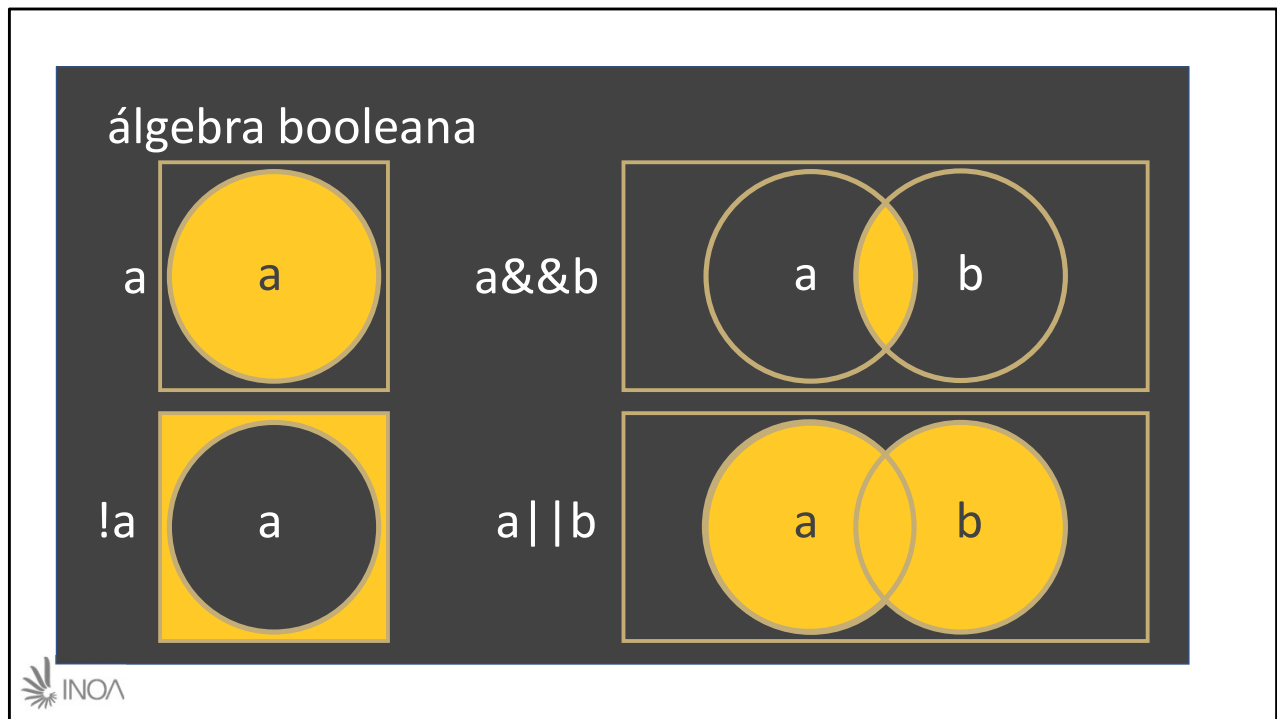
E os seguintes operadores de comparação:

**igualdade: === ou !==**

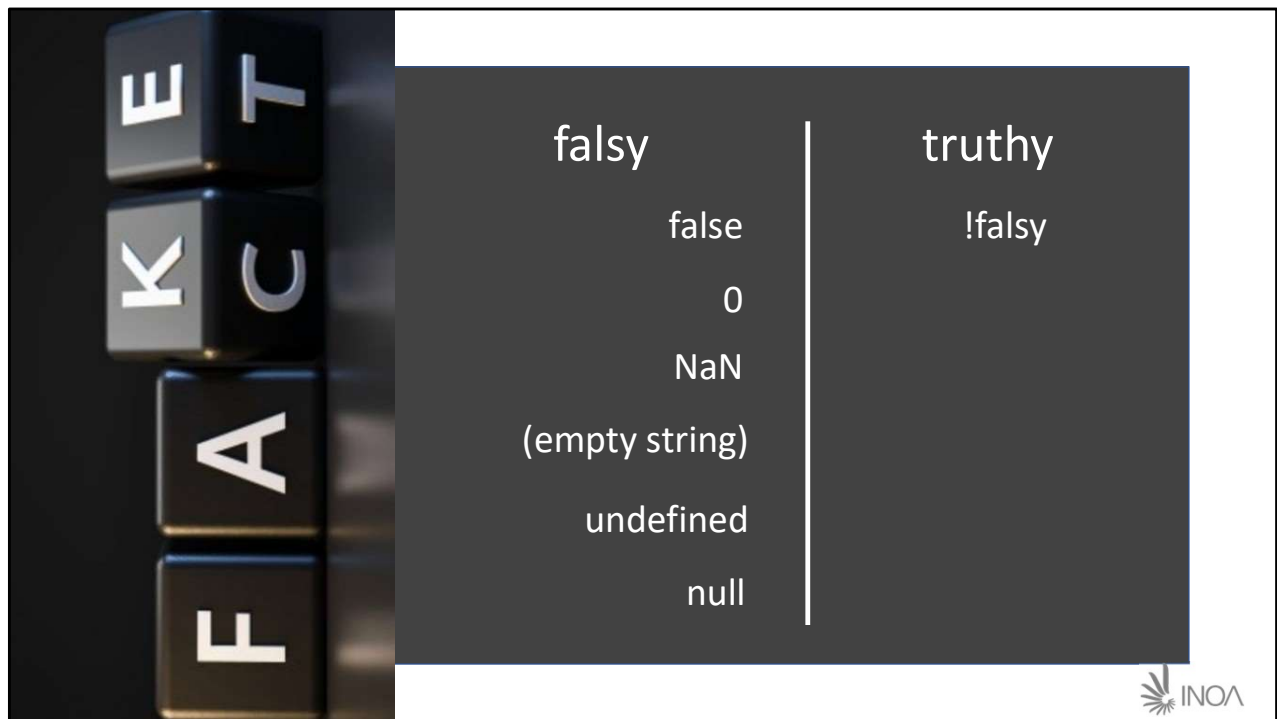
**comparação: >, <, >= e <=**

**Referências:**

<https://www.avatarapi.com/>



Quando utilizamos os operadores de comparação, na verdade estamos executando uma operação que retorna um valor booleano (**true** ou **false**).



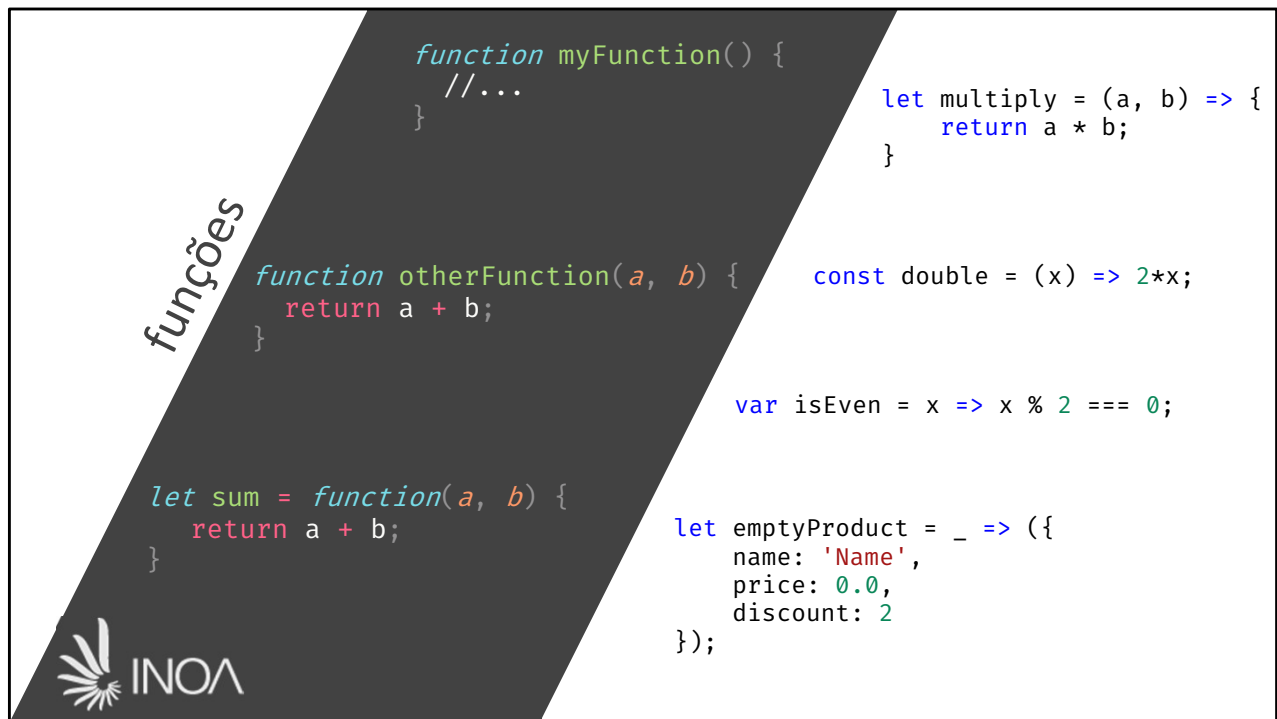
Conversão implícita de tipos, quando avaliamos um objeto como booleano, causa um efeito chamado de 'Truthyness' onde alguns valores especiais são convertidos para **false** para que a linguagem se torne mais expressiva.

Isso causa efeitos inesperados e o mesmo efeito pode ser obtido usando os operadores de equivalência: `==` e `!=`

Quando usamos o operador de **igualdade estrita** (`===`) a conversão implícita não acontecem. Por isso é preferível usar a **igualdade estrita** para evitar ambiguidades e comportamentos inesperados.

**Referencias:**

<https://www.sitepoint.com/javascript-truthy-falsy/>



Uma forma de pensar em funções é pensar em um bloco de código com um nome. Uma vez que nomeamos esse bloco de código, podemos reutilizar ele diversas vezes.

Como todo bloco de código uma função tem um escopo definido, e variáveis declaradas dentro dela não são acessíveis de fora.

A forma de interagir com esses blocos de código é através de seus parâmetros e seu retorno.

Quando uma função tem parâmetros de entrada e retorno, podemos pensar ela também como um função matemática.

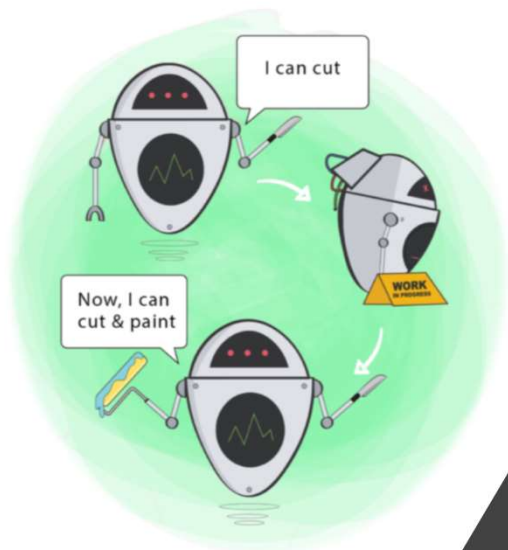
Um procedimento que transforma dados de entrada em dados de saída.

E podemos armazenar

Existem diversas formas de declararmos funções (ainda mais depois da ES6).

### Referência

<https://github.com/getify/You-Dont-Know-JS>



methods++

```

let robot = {
  name: "R2D2",
  cut: function () {
    console.log("cutting");
  },
};

robot.paint = () => {
  console.log("painting...");
  console.log("DONE");
};

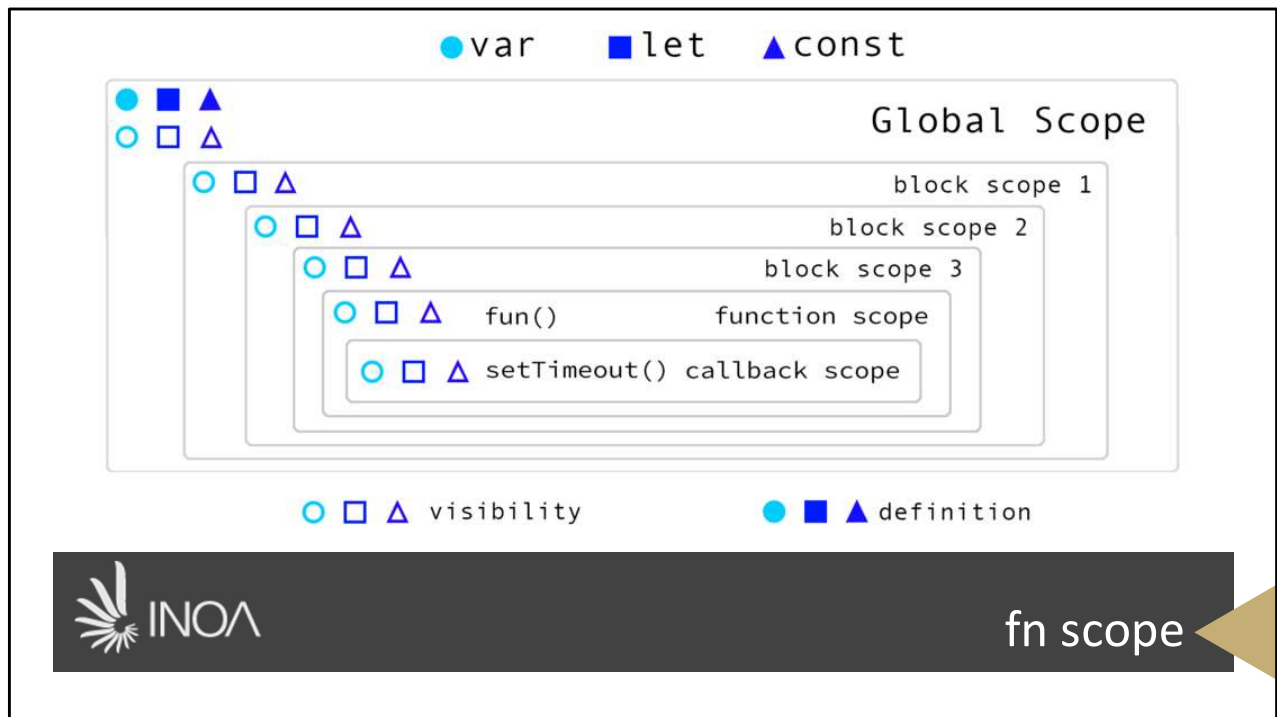
```

Funções podem ser atreladas a objetos, normalmente chamamos essas funções de métodos.

Objetos aceitam atribuições novas sob demanda.

### Referências:

<https://medium.com/backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898>



### Funções tem escopo próprio.

Da mesma forma que blocos de escopo, o acesso a variáveis é um caminho de mão única.

Escopos mais internos tem acesso a escopos mais externo.

Mas vimos que em blocos de escopo, devido ao efeito comumente chamado de içamento (hoisting), variáveis declaradas com `var` num escopo externo passam a ser globais.

Isso não ocorre em escopos de função.

Em funções, acesso a variáveis é um caminho de mão única.

Na verdade o escopo global nada mais é que um escopo da função principal da execução do seu script.

Muitas vezes, para nos protegermos contra sujar o escopo global, vemos um pattern chamado IIFE envolvendo o código de arquivo inteiro.

### Referências:

<https://jstutorial.medium.com/the-visual-guide-to-javascript-variable-definitions-scope-abfb86edad>

[https://en.wikipedia.org/wiki/Immediately\\_invoked\\_function\\_expression](https://en.wikipedia.org/wiki/Immediately_invoked_function_expression)

string  
number  
object  
undefined

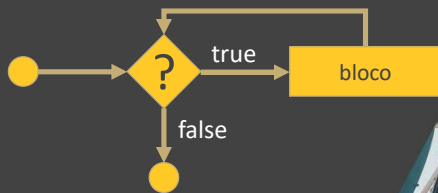
boolean  
function  
array

Resumo e um Chá





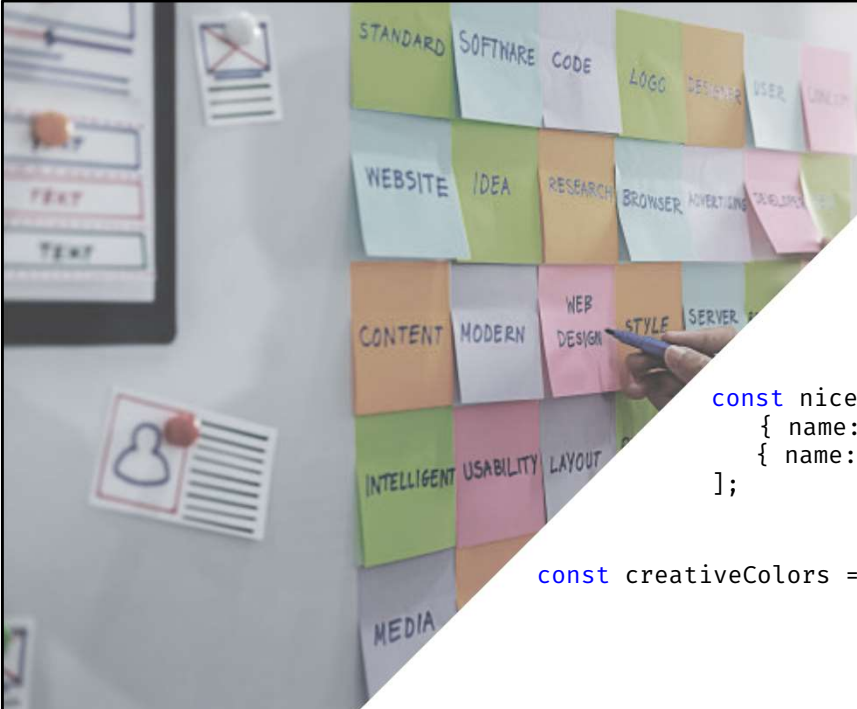
## controle de fluxo | loops



**Loops** são blocos de código que devem ser repetidos enquanto alguma condição for verdade.


A maioria das linguagens, incluindo javascript suportam 3 tipo de loops: while, do...while e for

Dentro do bloco de código que será repetido você pode abortar o loop imediatamente com o commando **break**, ou forçar o reinicio do ciclo com **continue**.



## arrays

```
let someStuff= [];  
  
var numbers = [1,2,3,4];  
  
const nicePeople = [  
  { name: 'Bob' }, { name: Carla' },  
  { name: 'Marta' }  
];  
  
const creativeColors = ['red', 'blue', 'yellow'];
```

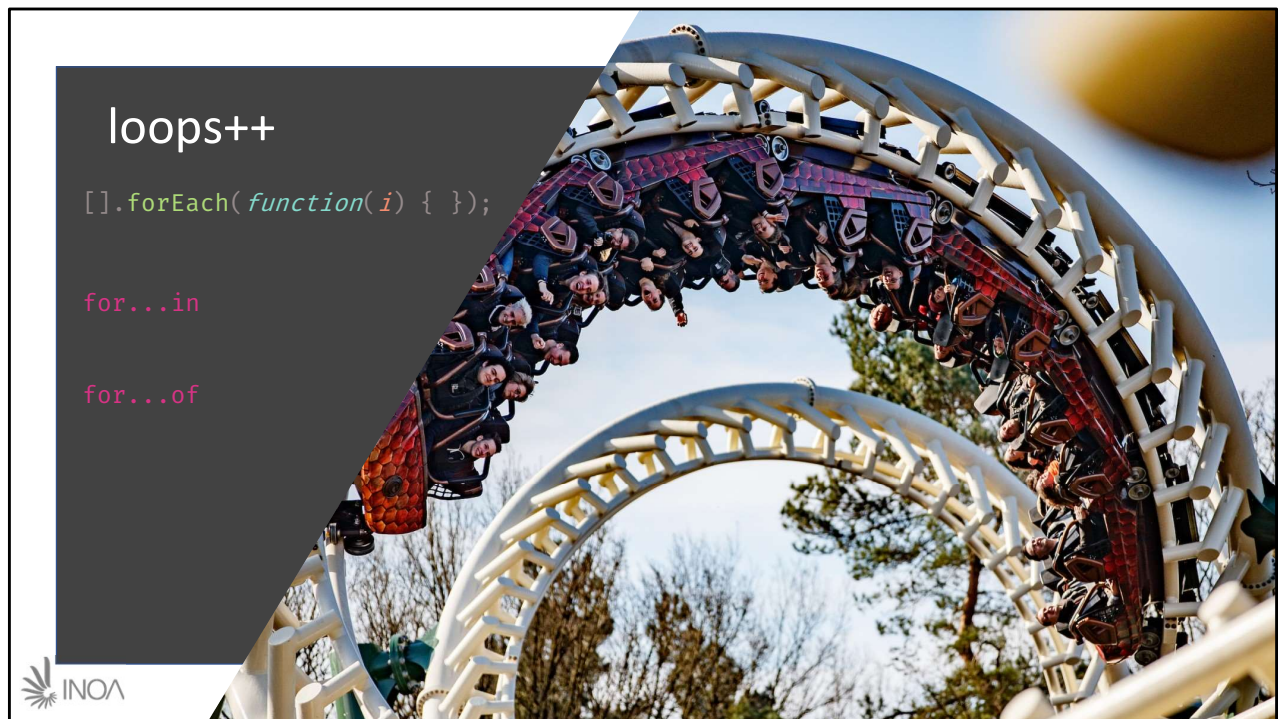


Arrays são objetos especiais que conseguem armazenar diversos itens.

JS por padrão só possui essa estrutura de dados de coleção, então podemos acessar esses itens por índices, colocar e remover novos itens à vontade.

### Referências:

<https://developer.mozilla.org/en-US/docs/Glossary/array>



Conseguimos também iterar sobre enumeráveis por exemplo sobre as propriedades de um objeto ou sobre os itens de um array.

Isso nos protege de loops infinitos.

## arrays++

```
[].slice();  
[].concat();  
[].splice();  
var [a, b, ...rest] =  
    [0,1,2,3,4,5];
```

Bread.slice(0,10)



Alguns métodos importantes de manipulação de arrays são usados corriqueiramente.

**slice:** cria um novo array, fatiando o array inicial partindo de um índice inicial até o índice final

**concat:** junta 2 arrays em novo array

**splice:** remove itens a partir de um índice de referência (e opcionalmente insere outros itens no lugar)

**spread (...):** desconstrói um array capturando sub-coleções dele em variáveis específicas



string            boolean  
number           function  
object            array  
undefined

Resumo e um tchau

THIS IS THE END OF  
THE PRESENTATION

INOVA

**Referências para mais exercícios:**

<https://www.codingame.com/training/easy/the-descent>  
<https://leetcode.com/problemset/all/>