

# Dell IT Academy

JavaScript e TypeScript

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)



# JavaScript

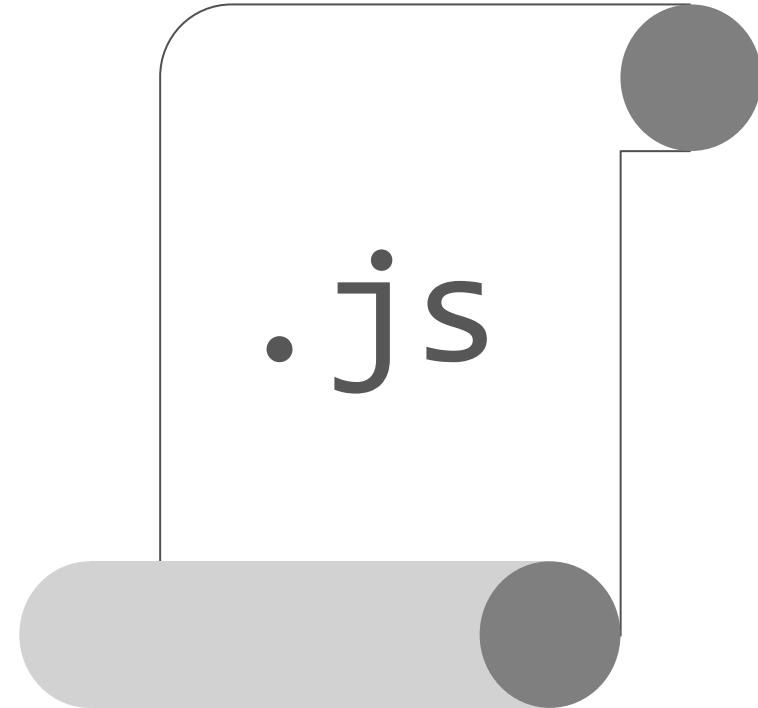


# Aplicações Interativas na Web

- HTML5 e CSS3 permitem a criação de páginas web
- Entretanto, a experiência é relativamente sem interatividade
  - **Interatividade** permite que o usuário realize uma ação e receba uma resposta
- Implementar a interatividade requer código na linguagem de programação JavaScript

# O que é JavaScript?

- JavaScript é uma linguagem multiparadigma, baseada em objetos via protótipos, dinâmica, fracamente tipada e usualmente interpretada por navegadores
  - **WebAssembly é compilado**
  - Com a linguagem, criam-se **scripts** que manipulam o HTML e CSS
  - A extensão do arquivo de script é usualmente **.js**



# JavaScript e os paradigmas

- Programação imperativa
- Programação estruturada
- Programação orientada a objetos
- Programação funcional

# JavaScript

- JavaScript possui múltiplas versões, suportadas ou não pelos diversos navegadores
- JavaScript é o nome “comum” de versões da linguagem que seguem a especificação ECMAScript
- Versões:
  - ECMAScript 2011, ECMAScript 5.1
  - ECMAScript 2015, ECMAScript 6
  - ECMAScript 2016, ECMAScript 7
  - ECMAScript 2017, ECMAScript 8
  - ECMAScript 2018, ECMAScript 9
  - ...
  - ECMAScript 2023, ECMAScript 14
- <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
- <https://tc39.es/ecma262/>

# JavaScript

- Como identificar a compatibilidade de versão?
  - <https://compat-table.github.io/compat-table/>
  - <https://caniuse.com/>

# JavaScript

- Para escrever código que se comunica com os elementos dos navegadores, JavaScript faz uso de diversas APIs
  - <https://developer.mozilla.org/en-US/docs/Web/API>

# Conectando JavaScript com HTML

- Conecta-se JavaScript ao documento HTML de diversas formas:

1. Embutindo código com a tag <script>
2. Referenciando um arquivo separado

1

```
<script>
  document.write("Hello World Wide Web");
</script>
```

2

```
<head>
  <script src="Script.js"></script>
</head>
```

# Conectando JavaScript com HTML

- Elemento SCRIPT:

- Pode aparecer múltiplas vezes dentro dos elementos HEAD e BODY
  - No HEAD usualmente colocam-se funções
  - No BODY usualmente colocam-se código e chamada a funções que geram conteúdo dinamicamente
- O script pode ser definido dentro do conteúdo do elemento ou através de referência via atributo src
- A linguagem de script definida via atributo type (JavaScript é o valor padrão)

- Elemento NOSCRIPT:

- Deve ser avaliado no caso de scripts não suportados ou desabilitado no navegador
- Conteúdo do elemento é utilizado ao invés do elemento SCRIPT

# Exemplo

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title></title>
  </head>
  <body>
    <h1>This is a boring website!</h1>
    <script>
      1      document.write("Hello, World!");
    </script>
  </body>
</html>
```

# Modo Estrito

- Modo estrito é o modo de execução restrito do JavaScript
  - Tornam explícitos erros “silenciosos” do JavaScript “normal”
  - Aplica correções semânticas que permite otimização de código
  - Proibi certas construções sintáticas
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)
- Habilita-se através do comando "**use strict**"; como primeira linha do script
- Observação: módulos são por padrão “strict”, então não necessitam de configuração explícita

# TypeScript



# TypeScript

- Superconjunto da linguagem JavaScript
  - JavaScript + Sistema de tipos + Analisador estático
- Código aberto



<https://www.typescriptlang.org/>

# TypeScript

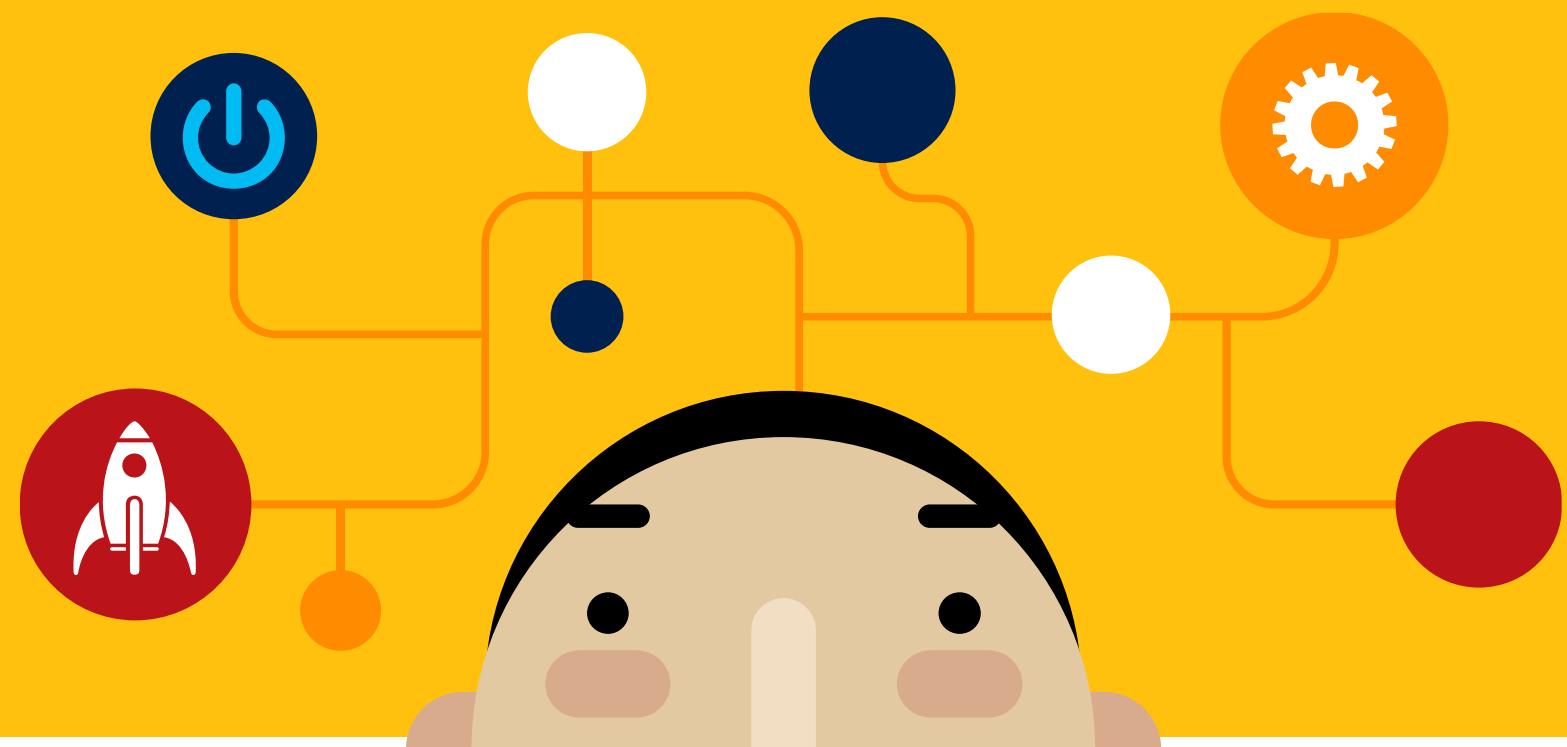


# TypeScript

- Compilador tsc pode ser configurado via arquivo de configuração
  - Arquivo tsconfig.json
  - Documentação oficial:
    - <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>
    - <https://www.typescriptlang.org/tsconfig>
    - <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

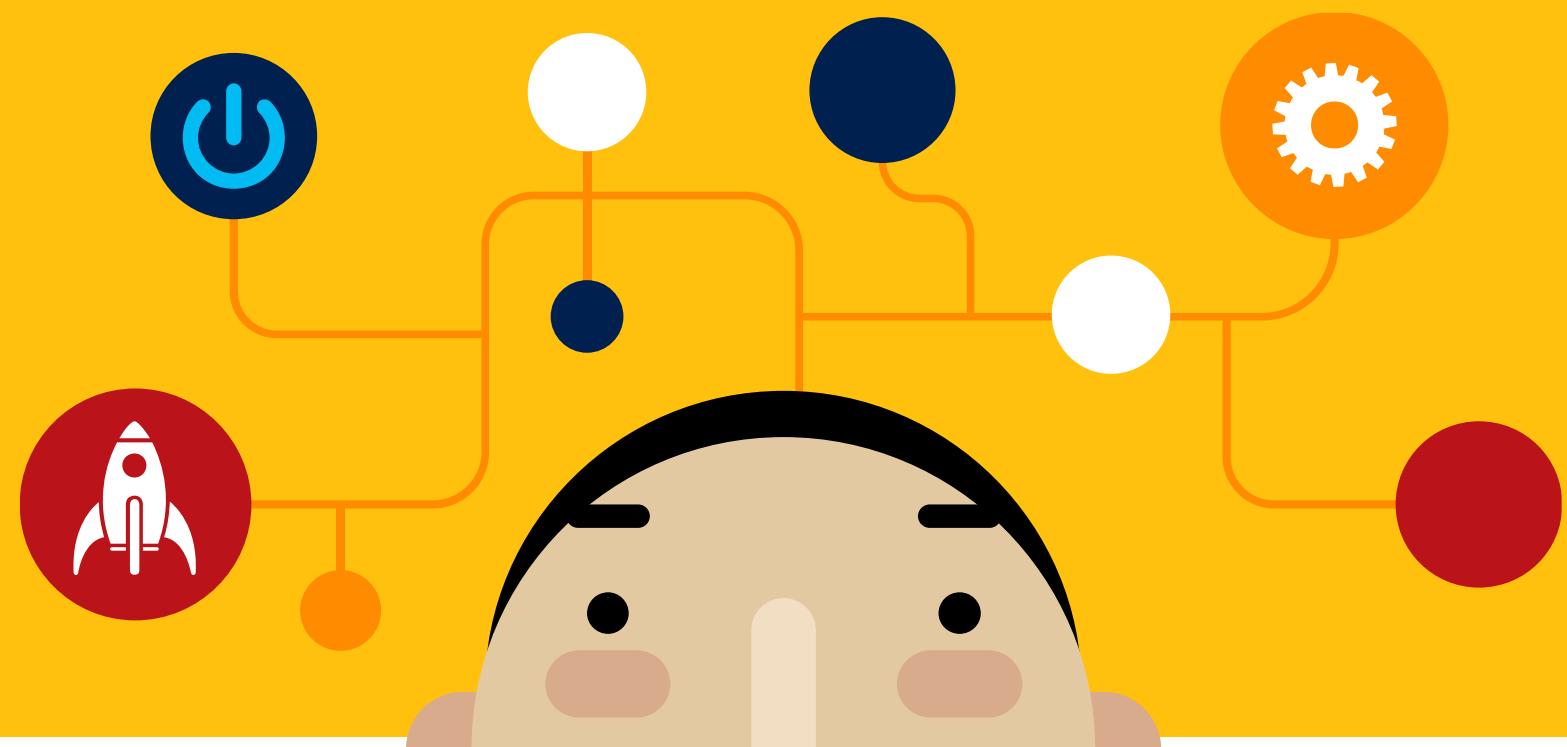
# Laboratório

- Abra as instruções do arquivo Lab00\_Ambiente



# Laboratório

- Abra as instruções do arquivo Lab01\_TypeScript\_Node



# Variáveis, Tipos de Dados, Operadores



# Variáveis

- Variáveis são definidas através da palavra-chave **var** ou **let** seguida do nome que se deseja e do tipo associado
  - Não são totalmente equivalentes
  - Var é uma construção “mais antiga”
    - Não possui escopo de bloco (pode ser referenciada fora do bloco de declaração)
  - Let é uma construção “mais nova”
    - Possui escopo de bloco
- Cuidado! Variáveis declaradas fora do escopo de uma função são chamadas de variáveis globais e podem ser acessadas de qualquer ponto do script
  - Seu uso não é recomendado

# Variáveis

```
var height: number = 6;
```

palavra-chave

nome da variável

tipo da variável

operador de atribuição

valor da variável

```
let height: number = 6;
```

palavra-chave

nome da variável

tipo da variável

operador de atribuição

valor da variável

# Constantes

- Scripts muitas vezes necessitam valores que não mudam
- Esses valores são armazenados em **constantes**
- Constantes são definidas através da palavra-chave **const** seguida do nome que se deseja, do tipo associado e do valor

```
const RED: string = '#F00';
```



# Tipos de Dados

- O sistema de tipos de TypeScript é chamado de “static structural typing with erasure”
  - Possui características diferentes do sistema de tipos estático usual de linguagens como Java ou C#
  - Um tipo é definido pela sua estrutura ao invés do “tipo” em si
  - “Type erasure” se refere ao processo de compilação para JavaScript remover qualquer informação de tipo anotado
- TypeScript possui uma seleção de tipos básicos e diversos “mecanismos de composição” para tipos definidos pelo usuário
- CONVENÇÃO:
  - verificação de tipos é opcional no TypeScript
  - neste curso iremos utilizar tipos explícitos nas variáveis e funções

# Tipos de Dados JavaScript

## Tipos primitivos:

- boolean
  - Valores *true* ou *false*
- string
  - Valores de sequência de caracteres
- number
  - Valores numéricos inteiros ou de ponto-flutuante
- bigint
  - Valores numéricos inteiros de precisão arbitrária
- undefined
  - Valor único *undefined*, representa um valor de variáveis que não receberam nenhuma atribuição
- null
  - Valor único *null*, representa a ideia de “nada” ou “vazio”
- symbol
  - Representa um tipo imutável e único utilizada para chaves de propriedades de objetos

# Tipos de Dados JavaScript

Tipos não-primitivos:

- object
  - Representa o conceito de objeto

Vários tipos de objetos:

- Math, Date, Array, Map, Set, etc

# Tipos de Dados TypeScript

## Tipos da linguagem:

- any
  - Representa qualquer tipo
  - Utilizado para uma variável que pode receber quaisquer valores
  - Significa desabilitar a verificação de tipo
- unknown
  - Representa qualquer tipo
  - Mais seguro do que *any*, pois não é permitido realizar qualquer computação sobre um valor do tipo *unknown*
- void
  - Representa a ausência completa de tipo
  - Utilizado para indicar funções que não retornam um valor, ou seja, são funções que retornam tipo *void*
  - Uma variável do tipo *void* somente pode receber valores *undefined* ou *null*

# Tipos de Dados TypeScript

Tipos da linguagem:

- enum
  - Representa enumerações
- tuple
  - Representa o conceito de tuplas

# Number

- Valores numéricos de ponto flutuante 64 bits padrão IEEE754
  - 64 bit de precisão dupla IEEE 754
  - Valores especiais *NaN*, *+Infinity* e *-Infinity*
- Valores em hexadecimal inicial por 0x
- Valores em octal iniciam por 0o
- Valores em binário iniciam por 0b
- Propriedades:
  - *MAX\_VALUE*, *MIN\_VALUE*, etc
- Métodos:
  - *toExponential()*, *toFixed()*, *toPrecision()*, *toString()*, *valueOf()*

# String

- Sequência imutável de caracteres Unicode UTF-16
- Representada por caracteres entre " ou '
- Quando representadas entre ` , permitem embutir valores de variáveis ou expressões via \${}
- Exemplo: `Alo \${nome}`
- É possível acessar caracteres por posição (inicia em 0) via [ ]
- Propriedades:
  - *length* – informa o número de caracteres
- Métodos:
  - *charAt()*, *indexOf()*, *split()*, *substring()*, *toUpperCase()*, etc

# String

- Caracteres especiais (de escape)

CARACTERE	DESCRIÇÃO
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Tab
\uNNNN	Símbolo Unicode com valor hexadecimal
\u{NNNNNNNN}	Símbolo Unicode com valor hexadecimal

# String

- CUIDADO! Comparação de strings para ordenação não é trivial
  - Alfabetos diferentes em linguagens diferentes
- Padrão ECMA 402 busca resolver a questão
- Método *localeCompare()*

# Enums

- Enumerações são um conjunto de constantes nomeadas
- TypeScript suporta enumerações com base em string e number
- Ex.:

```
enum Direcao {  
    Acima,  
    Abaixo,  
    Direita,  
    Esquerda  
}  
  
let dir: Direcao = Direcao.Direita;
```

# Objetos

- Conjuntos diferentes de objetos são disponibilizados:
  - Intrínsecos ao JavaScript
  - Fornecidos pelo navegador
  - Fornecidos pela API DOM
- Cada objeto possui métodos e propriedades

# Objetos

- JavaScript:

- Array, Boolean, Date, Math, Number, String, RegExp, Global

- Navegador:

- Window, Navigator, Screen, History, Location, Console

- DOM:

- Document, Event, etc

# Math

- Objeto que possui a definição de constantes e operações matemáticas de uso geral
- Propriedades:
  - *E, PI, LN2, etc*
- Métodos:
  - *abs(), sin(), exp(), max(), pow(), random(), etc*

# Date

- Suporta a manipulação de tempo e data

- Construtor:

- Ano possui 4 dígitos
- Mês de 0 a 11

```
let hoje = new Date();
let dia = new Date(2017,4,2);
```

- Comparação:

- Suporta comparação via >, <, etc

```
hoje < d
```

- Métodos:

- getFullYear(), getMonth(), getDate(), getDay(), getHours(), getMinutes(), getSeconds(),  
getMilliseconds(), getTime(), toString(), toDateString(), setFullYear(), setMonth(), setDate(), setHours(),  
setMinutes(), setSeconds(), setMilliseconds(), setTime(), etc

# Global

- Objeto que possui várias propriedades e métodos de uso geral

- Em um navegador, recebe o nome de *window*
- No NodeJS, recebe o nome de *global*

- Propriedades:

- Infinity, NaN, undefined

- Métodos:

- `parseFloat(string)` e `parseInt(string)` – convertem uma string para número
- `escape(string)` e `unescape(string)` – codifica/decodifica uma string
- `eval(string)` – avalia e executa o conteúdo da string com código de script
- etc

# Operadores

- Aritméticos:

- + - \* / % \*\*

- Unitários:

- ++ -- - +

- Comparação:

- < <= > >= == != === != ==

- Lógicos:

- && || !

- Bits:

- & | ^ ~ << >> >>>

- Atribuição:

- = += -= \*= /= %= <<= >>= >>>= &= |= ^=

# Operadores - Igualdade

## `==` e `!=`

- Tentam converter os operandos para um mesmo tipo e em seguida testam se são iguais

```
console.log(5 == "5"); // true , TS Error  
console.log(5 === "5"); // false , TS Error
```

## `====` e `!====`

- Se os tipos dos operandos forem diferentes, retorna *false*
- São chamados de operadores de igualdade restritos

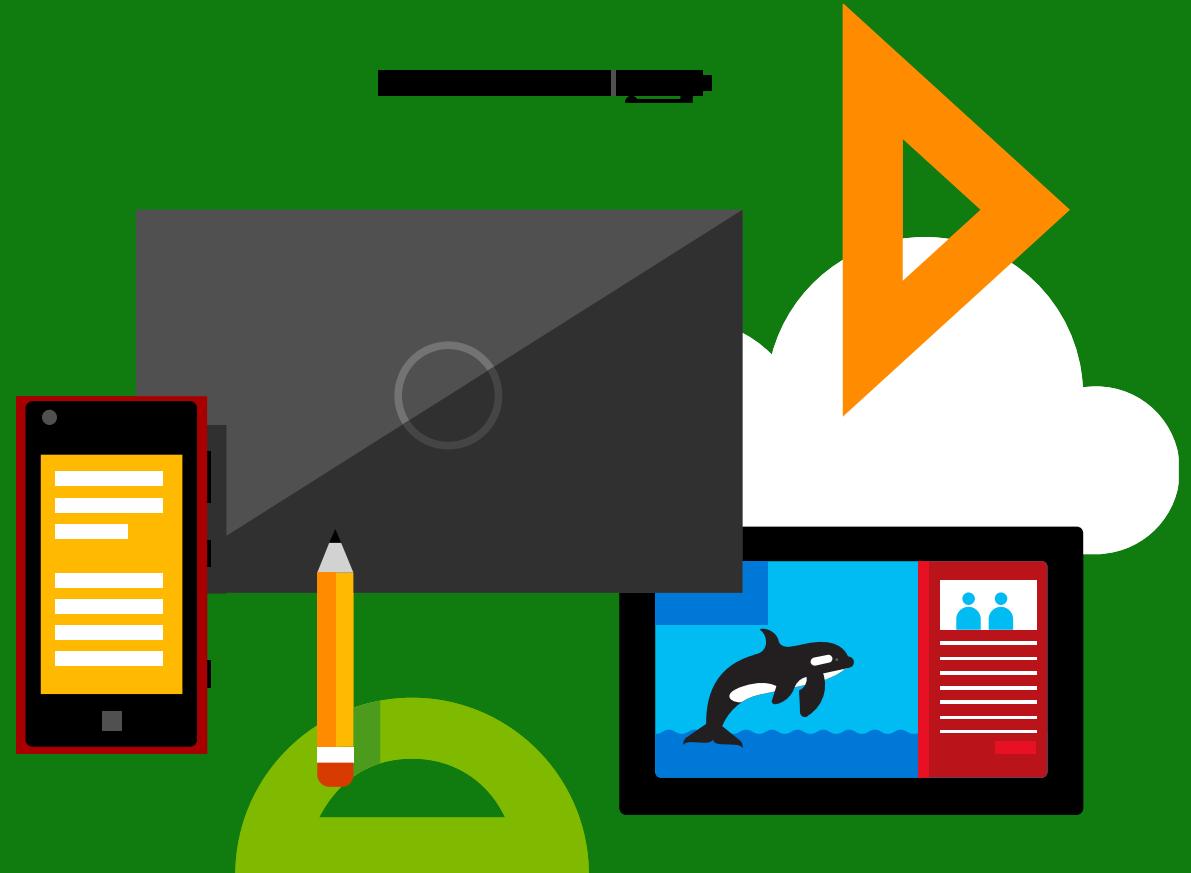
# Asserções de Tipos

- Uma asserção de tipo deve ser utilizada quando o compilador TypeScript não conseguir realizar uma inferência sobre o tipo de uma expressão
  - É tarefa do programador indicar o tipo correto
- Duas sintaxes:
  - <tipo>
    - Ex.:

```
let umvalor: any = 'um texto';
let tamanho: number = (<string>umvalor).length;
```
  - as
    - Ex.:

```
let umvalor: any = 'um texto';
let tamanho: number = (umvalor as string).length;
```

# Comandos



# Comandos - IF

## Estrutura

```
if (condição) comando;
```

```
if (condição) comando;  
else comando;
```

```
if (condição) { bloco de comandos }
```

```
if (condição) { bloco de comandos }  
else { bloco de comandos }
```

# Comandos - IF

## Exemplo

```
if (hora < 12) {  
    saudacao = "bom dia";  
}
```

```
if (hora < 12) {  
    saudacao = "bom dia";  
} else {  
    saudacao = "boa tarde";  
}
```

# Comandos - SWITCH

## Estrutura

```
switch (expressão) {  
    case expressão : comandos; break;  
    case expressão: comandos; break;  
    default : comandos;  
}
```

# Comandos - SWITCH

## Estrutura

```
switch (valor) {  
    case 0:  
    case 1:  
        console.log("zero ou um");  
        break;  
    case 2:  
        console.log("dois");  
        break;  
    default:  
        console.log("outro valor");  
}
```

# Comandos - WHILE

## Estrutura

```
while (condição) comando;
```

```
while (condição) { bloco de comandos }
```

# Comandos - WHILE

## Exemplo

```
let i = 0;
while (i < 3) {
    console.log(i);
    i++;
}
```

# Comandos - DO WHILE

## Estrutura

```
do comando while (condição);
```

```
do { bloco de comandos } while (condição);
```

# Comandos - DO WHILE

## Exemplo

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i<3);
```

# Comandos - FOR

## Estrutura

```
for (inicialização; condição; passo) comando;
```

```
for (inicialização; condição; passo) { bloco de comandos }
```

# Comandos - FOR

## Exemplo

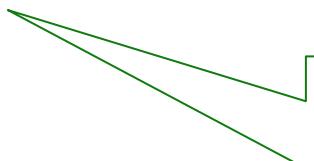
```
for (let i = 0; i<3; i++) {  
    console.log(i);  
}
```

# Comandos – FOR..OF

## Estrutura

```
for (variável of objeto) comando;
```

```
for (variável of objeto) { bloco de comandos }
```



O objeto alvo deve ser um  
objeto iterável, ou seja,  
fornecer um iterador.

# Comandos – FOR..OF

## Exemplo

```
const a = [3,5,7];
for (const i of a) {
  console.log(i);
}
```

# Funções



# Revisitando funções...

- Funções atuam como subprogramas/subrotinas que são úteis para a modularização e reutilização de código.
- Em JavaScript:
  - Funções são objetos do tipo Function;
  - Sempre retornam um valor
    - Explicitamente via comando return;
    - Ou valor padrão undefined;
    - Ou, para construtores via comando new, um valor associado a this.
  - Podem ser criadas através de diferentes formas
    - Via declaração
    - Via expressão
    - etc

# Função via declaração

- Uma função criada via declaração (*function declaration*) possui a seguinte sintaxe:
  - A palavra-chave *function*;
  - O nome da função;
  - Uma lista de parâmetros entre parênteses e separados por vírgulas;
  - Um bloco de comandos entre colchetes, chamado de corpo de definição da função.

```
function nome (lista de parâmetros) { bloco de comandos }
```

# Função via declaração

```
palavra-chave      nome da função   lista de parâmetros  
function          ↗             ↗             ↗  
function somar(a,b) {  
  
    return a + b;  
  
}                                |  
                                comando  
  
-----|  
-----|  
           bloco de definição da função
```

# Função via declaração

```
function somar(a,b) {  
    return a + b;  
}  
  
console.log(somar(1,2));
```

# Função via expressão

- Uma função criada via expressão (*function expression*) possui sintaxe semelhante a uma função via declaração mas é criada como parte de uma expressão maior.
- O nome da função passa a ser opcional, e quando não informado, dá origem a uma *função anônima*.

# Função via expressão

```
const somar = function (a,b) {  
    return a + b;  
}  
  
|  
| comando  
|  
| bloco de definição da função
```

palavra-chave

lista de parâmetros

# Função via expressão

```
const somar = function (a,b)
{
    return a + b;
}

console.log(somar(1,2));
```

# Função via expressão

- Um padrão interessante do uso de funções declaradas via expressão são as chamadas *IIFE* (do inglês *Immediately Invoked Function Expression*) ou *função via expressão imediatamente invocada*.

```
(function (a,b) {  
    console.log(a + b);  
})(1,2);
```

# Função arrow

- Além do mecanismo de função via expressão, uma função anônima também pode ser criada em JavaScript a partir de uma **função arrow**, também conhecida como **função lambda**.
- Possui uma sintaxe curta através do símbolo `=>` separando a definição dos parâmetros da função do corpo de definição da função.

`(lista de parâmetros) => { bloco de comandos }`

`(lista de parâmetros) => expressão`

# Função arrow

```
const somar = (a,b) => {  
    return a + b;  
}
```

---

|  
corpo de definição da função

lista de parâmetros      arrow

# Função arrow

```
const somar = (a,b) => {  
    return a + b;  
}  
  
console.log(somar(1,2));
```

# Função arrow

# Função arrow

```
const somar = (a,b) => a + b;  
console.log(somar(1,2));
```

# Função arrow

- Qual a utilidade?
  - A API de JavaScript possui muitas funções e métodos que recebem outras funções como argumentos
  - Por exemplo:
    - O método *forEach()* do array é capaz de aplicar a função recebida como argumento sobre cada elemento do array;
    - Passar uma função arrow como argumento é uma forma simples e direta de indicar a função que será executada.

# Função arrow

```
const nomes = ["Ana", "Beatriz", "Carlos"];
nomes.forEach(nome => console.log(`Alo
${nome}!`));
```

# Funções

- Características importantes:

- A passagem de parâmetros é por valor
- Cuidado: JavaScript não verifica o número de parâmetros passados (se não recebe um valor, o parâmetro tem valor `undefined`; parâmetros podem possuir valores padrão via símbolo de atribuição), mas TypeScript verifica!

# Funções em TypeScript

```
function aloMundo(): void {  
    console.log('Alô Mundo!');  
}
```

```
function aloMundo(): string {  
    return 'Alô Mundo!';  
}
```

```
function aloMundo(nome: string): void {  
    console.log('Alô' + nome + '!');  
}
```

# Funções em TypeScript

```
function potencia(base: number, expoente: number = 2): number {  
    let resultado = 1;  
    for (let cont = 0; cont < expoente; cont++) {  
        resultado *= base;  
    }  
    return resultado;  
}  
  
console.log(potencia(4));  
console.log(potencia(2,6));
```

# Funções em TypeScript

## Exemplo

```
const somar: (x: number, y: number) => number = function(x: number, y:  
number): number {  
    return x + y;  
}  
console.log(somar(1, 2));
```

# Funções em TypeScript

## Exemplo

```
const somar = (x: number ,y: number) => x + y;  
console.log(somar(1,2));
```

# Funções em TypeScript

## Exemplo (closure)

```
function multiplicar (fator: number): (f: number) => number {  
    return numero => numero * fator;  
}  
  
let dobrar = multiplicar(2);  
console.log(dobrar(5));
```

# Funções de alta-ordem

- Funções de alta-ordem (ou ordem superior) são caracterizadas por:
  - Receber por argumento outra função e/ou
  - Retornar como resultado outra função.
- Esse é o motivo porquê em linguagens como JavaScript se diz que funções são “cidadãos de primeira classe” e podem ser referenciados, manipulados através de variáveis e passados como argumentos para outras funções.

# Arrays



# Arrays

- Objeto que provê uma estrutura de dados que permite armazenar uma coleção indexada de qualquer tipo de elemento
- Também funcionam como base para implementação de outras estruturas de dados, como listas, filas e pilhas

# Arrays

- Declaração de arrays:

- Literal

```
let a: string[] = ["A", "B", "C"];
console.log(a);
```

```
let a: Array<string> = ["A", "B", "C"];
```

- Indexação de elementos:

- Começa no índice 0
  - Operador [ ]

```
let c = a[1];
```

```
a[2] = "c";
```

# Arrays

- Comprimento do array definido na propriedade **length**

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

- Comprimento do array poder ser aumentado atribuindo-se um valor a uma posição de índice maior ao tamanho atual

```
a[3] = "D";
```

- Cuidado: Aumentar o tamanho do array gera posições intermediárias com valores indefinidos!
- Os array são esparsos, isto é, somente reservam espaço para os elementos definidos

# Arrays - Iteração

- O meio mais tradicional de iterar sobre os elementos de um array é através de laços de repetição do tipo **for**

```
let a = [1, 2, 3];
for (let i=0; i<a.length; i++) {
  console.log(a[i]);
}
```

# Arrays - Iteração

- Comando para laços de repetição do tipo **for .. of**
  - Itera sobre os elementos do array, sem expor os índices
  - Na verdade, o comando funciona com qualquer objeto iterável, ou seja, que fornece um iterador
    - Por exemplo, strings também são objetos iteráveis

```
let a = [1, 2, 3];
for (let e of a) {
  console.log(e);
}
```

# Arrays - Métodos

- Métodos:

- `toString()` – retorna uma string com os valores do array separados por vírgula

```
console.log(a.toString());
```

- `join()` – retorna uma string com os valores do array separados pelo símbolo fornecido

```
console.log(a.join(" - "));
```

- `concat()` – retorna um novo array resultante da concatenação dos arrays passados por parâmetro

```
let a2 = a.concat(["X","Y"]);
```

- `slice(indice, fim)` – particiona um array e retorna um novo array com a partição, sem alterar o array original

```
let a = ["A", "B", "C"];
let a3 = a.slice(1); //a[1], a[2]
let a4 = a.slice(0,2); //a[0], a[1]
```

# Arrays - Métodos

- **Métodos:**

- `indexOf(item, inicio)` – retorna o índice do array que contem o elemento *item*, opcionalmente a partir da posição *inicio*; ou -1 caso não encontre
- `lastIndexOf(item, inicio)` – retorna o índice da última ocorrência do elemento *item* no array (ou seja, realiza a busca de trás para frente), opcionalmente a partir da posição *inicio*; ou -1 caso não encontre
- `includes(item, inicio)` – retorna true caso o array contenha o elemento *item*, opcionalmente a partir da posição *inicio*; ou false caso contrário

```
let a = [1,2,2];
console.log(a.indexOf(2));
console.log(a.indexOf(2,2));
console.log(a.lastIndexOf(2));
console.log(a.includes(0));
```

# Arrays - Métodos

- Métodos:

- `findIndex(funçãoPredicado)` – retorna o index do primeiro elemento do array de acordo com a função de predicado; retorna -1 caso contrário
  - A função de predicado é uma função que retorna true ou false
  - A função de predicado tem a assinatura `function(item,index,array)`, onde *item* é o elemento, *index* é a posição atual do elemento, *array* é o próprio array; usualmente utiliza-se somente o primeiro parâmetro
- `find(funçãoPredicado)` – retorna a primeira ocorrência de um elemento no array de acordo com a função de predicado; retorna *undefined* caso contrário

```
let a = [1,2,3];
let i = a.findIndex(function(item){
    return item >= 2;
});
let e = a.findIndex(function(item){
    return item >= 0 && item <= 2;
});
```

# Arrays - Métodos

- Métodos:

- `forEach(funcãoAplicação)` – itera sobre cada elemento do array e chama a função de aplicação sobre cada um
  - A função de aplicação tem a assinatura `function(item,index,array)`, onde *item* é o elemento, *index* é a posição atual do elemento, *array* é o próprio array; usualmente utiliza-se somente o primeiro parâmetro

```
let a = [1,2,3];
a.forEach(function(item,index){
  console.log(` ${item} na posição ${index}`);
});
```

# Arrays - Ordenação

- Métodos:

- sort() – ordena um array de forma ascendente, de acordo com o tipo string (ordem lexicográfica)

```
let a = ["A", "B", "C"];
a.sort();
```

- reverse() – ordena um array de forma descendente, de acordo com o tipo string (ordem lexicográfica)

```
let a = ["A", "B", "C"];
a.reverse();
```

# Arrays - Ordenação

- Métodos:

- `sort(funçãoDeComparação)` – ordena um array de acordo com a função de comparação fornecida
  - A função de comparação deve comparar dois valores e retornar um número negativo (primeiro menor que segundo), número positivo (primeiro maior que segundo), zero (caso contrário)

```
let a = [3, 1, 2];
a.sort(function(x,y){
  if (x<y) return -1;
  if (x>y) return 1;
  return 0;
});
```

# Padrões de codificação

- A utilização de funções puras e funções de alta-ordem junto com a recursão, dão origem a vários padrões de codificação de soluções funcionais.
- Muitos padrões de solução que envolvem iteração podem ser resolvidos através da recursão.
- Bibliotecas de funções de JavaScript já implementam esses padrões.

# Padrões de codificação

Map

Filter

Reduce



## Exemplo

- Multiplicar por 10 todos os números pares entre 1 e 10 e depois somá-los.

```
let numeros = [1,2,3,4,5,6,7,8,9,10];
let somatorio = numeros.filter(n => n % 2 === 0)
                        .map(a => a * 10)
                        .reduce((a,b) => a + b);
console.log(somatorio);
```

[1,2,3,4,5,6,7,8,9,10]



filter(n => n % 2 === 0)

[2,4,6,8,10]



map(a => a \* 10)

[20,40,60,80,100]



reduce((a,b) => a + b)

300

# Map

- O método *map()* cria um novo array populado com o resultado proveniente da execução da função recebida como argumento sobre cada elemento do array.
- Observação:
  - A função recebida é executada somente sobre índices do array que possuem valores associados dentro do limite de tamanho do array quando da execução de *map()*;
  - Não utilize função com efeito colateral como argumento para *map()*;
  - *Map()* não altera o array original, mas retorna um novo array.

# Map

- A função a ser executada possui os seguintes parâmetros:
  - *elemento* atualmente sendo processado
  - *índice* do elemento atualmente sendo processado
  - *array* sobre o qual *map()* foi executado

## Funções arrow

```
map((element) => { })  
map((element, index) => { })  
map((element, index, array) => { })
```

## Funções anônimas

```
map(function (element) { })  
map(function (element, index) { })  
map(function (element, index, array) { })
```

## Função de callback

```
map(callbackFn)
```



## Exemplo

- Mapeando um array de números para um array de números elevados à potência 2.

```
let numeros = [1,2,3,4,5];
let resto = numeros.map(n => Math.pow(n,2));
console.log(numeros);
console.log(resto);
```



## Exemplo

1	2	3	4	5
---	---	---	---	---

--	--	--	--	--



## Exemplo

1	2	3	4	5
---	---	---	---	---

$n \Rightarrow$   
`Math.pow(n,2)`

2				
---	--	--	--	--



## Exemplo

1	2	3	4	5
---	---	---	---	---

$n \Rightarrow$   
`Math.pow(n,2)`

2	4			
---	---	--	--	--



## Exemplo

1	2	3	4	5
---	---	---	---	---

2	4			
---	---	--	--	--

# Filter

- O método *filter()* cria uma cópia rasa (copia as referências e não clona o objeto) de uma porção do array filtrada aos elementos de teste verdadeiro de acordo com o predicado fornecido como argumento.
- Observação:
  - Dá-se o nome de predicado à função recebida pois ela deve retornar um valor que possa ser interpretado como verdadeiro ou falso;
  - A função recebida é executada somente sobre índices do array que possuem valores associados dentro do limite de tamanho do array quando da execução de *filter()*;
  - Não utilize função com efeito colateral como argumento para *filter()*;
  - *Filter()* não altera o array original, mas retorna um novo array.

# Filter

- A função a ser executada possui os seguintes parâmetros:
  - *elemento* atualmente sendo processado
  - *índice* do elemento atualmente sendo processado
  - *array* sobre o qual *filter()* foi executado

## Funções arrow

```
filter((element) => { })  
filter((element, index) => { })  
filter((element, index, array) => { })
```

## Funções anônimas

```
filter(function (element) { })  
filter(function (element, index) { })  
filter(function (element, index, array) { })
```

## Função de callback

```
filter(callbackFn)
```



## Exemplo

- Obtendo os números ímpares a partir de um array de números.

```
let numeros = [1,2,3,4,5,6,7,8,9,10];
let impares = numeros.filter(n => n%2 === 1);
console.log(impares);
```



## Exemplo

1	2	3	4	5
---	---	---	---	---



## Exemplo

1	2	3	4	5
---	---	---	---	---

$n \Rightarrow n \% 2 == 1$  ✓

1
---



## Exemplo

1	2	3	4	5
---	---	---	---	---

$n \Rightarrow n \% 2 == 1$

1
---



## Exemplo

1	2	3	4	5
---	---	---	---	---

$n \Rightarrow n \% 2 == 1$  ✓

1	3
---	---



## Exemplo

1	2	3	4	5
---	---	---	---	---

1	3
---	---

# Reduce

- O método *reduce()* executa uma função de redução (chamada *reducer*) sobre cada elemento do array, em ordem ascendente, passando o valor retornado sobre o cálculo do elemento anterior à execução do *reducer* sobre o elemento atual; o valor final retornado é um único elemento.
- Observação:
  - A função recebida é executada somente sobre índices do array que possuem valores associados dentro do limite de tamanho do array quando da execução de *reduce()*;
  - Não utilize função com efeito colateral como argumento para *reduce()*;
  - *Reduce()* não altera o array original.

# Reduce

- A função a ser executada possui os seguintes parâmetros:
  - *acumulador* utilizado para receber o valor do *reducer* sobre o elemento anterior
    - Na primeira chamada recebe *valor inicial* ou *array[0]*
  - *valor atual* do elemento sendo processado
    - Na primeira chamada recebe *array[0]* se *valor inicial* foi fornecido ou *array[1]* caso contrário
  - *índice atual* do elemento sendo processado
    - Na primeira chamada recebe 0 se *valor inicial* foi fornecido ou 1 caso contrário
  - *array* sobre o qual *reduce()* foi executado
  - *valor inicial* opcional para inicializar o *acumulador*

## Funções arrow

```
reduce((accumulator, currentValue) => {})  
reduce((accumulator, currentValue, currentIndex) => {})  
reduce((accumulator, currentValue, currentIndex, array) => {})
```

## Funções anônimas

```
reduce(function (accumulator, currentValue) {})  
reduce(function (accumulator, currentValue, currentIndex) {})  
reduce(function (accumulator, currentValue, currentIndex, array)  
{})
```

## Função de callback

```
reduce(callbackFn)
```

# Reduce

- A função a ser executada possui os seguintes parâmetros:
  - *acumulador* utilizado para receber o valor do *reducer* sobre o elemento anterior
    - Na primeira chamada recebe *valor inicial* ou *array[0]*
  - *valor atual* do elemento sendo processado
    - Na primeira chamada recebe *array[0]* se *valor inicial* foi fornecido ou *array[1]* caso contrário
  - *índice atual* do elemento sendo processado
    - Na primeira chamada recebe 0 se *valor inicial* foi fornecido ou 1 caso contrário
  - *array* sobre o qual *reduce()* foi executado
  - *valor inicial* opcional para inicializar o *acumulador*

## Funções arrow

```
reduce((accumulator, currentValue) => {}, initialValue)  
reduce((accumulator, currentValue, currentIndex) => {},  
initialValue)  
reduce((accumulator, currentValue, currentIndex, array) => {},  
initialValue)
```

## Funções anônimas

```
reduce(function (accumulator, currentValue) {}, initialValue)  
reduce(function (accumulator, currentValue, currentIndex) {},  
initialValue)  
reduce(function (accumulator, currentValue, currentIndex, array)  
{}, initialValue)
```

## Função de callback

```
reduce(callbackFn, initialValue)
```



## Exemplo

- Calcular o somatório de um array de números.

```
let numeros = [1,2,3,4,5];
let somatorio = numeros.reduce((a,b) => a + b, 0);
console.log(somatorio);
```



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
------------	-------	--------	-----------------



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
0	1	0	1



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
0	1	0	1
1	2	1	3



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
0	1	0	1
1	2	1	3
3	3	2	6



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
0	1	0	1
1	2	1	3
3	3	2	6
6	4	3	10



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b, 0)
```

acumulador	valor	índice	valor retornado
0	1	0	1
1	2	1	3
3	3	2	6
6	4	3	10
10	5	4	15



## Exemplo

- Calcular o somatório de um array de números.
- O quê acontece se não for informado um valor inicial?

```
let numeros = [1,2,3,4,5];
let somatorio = numeros.reduce((a,b) => a + b);
console.log(somatorio);
```



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b)
```

acumulador	valor	índice	valor retornado
------------	-------	--------	-----------------



## Exemplo

1	2	3	4	5
---	---	---	---	---

```
numeros.reduce((a,b) => a + b)
```

acumulador	valor	índice	valor retornado
1	2	1	3



## Exemplo

1	2	3	4	5
---	---	---	---	---

`numeros.reduce((a,b) => a + b)`

acumulador	valor	índice	valor retornado
1	2	1	3
3	3	2	6



## Exemplo

1	2	3	4	5
---	---	---	---	---

`numeros.reduce((a,b) => a + b)`

acumulador	valor	índice	valor retornado
1	2	1	3
3	3	2	6
6	4	3	10



## Exemplo

1	2	3	4	5
---	---	---	---	---

`numeros.reduce((a,b) => a + b)`

acumulador	valor	índice	valor retornado
1	2	1	3
3	3	2	6
6	4	3	10
10	5	4	15



## Exemplo

- Calcular o somatório de um array de números.
- O quê acontece se o array for vazio e com um valor inicial informado?

```
let numeros = [];
let somatorio = numeros.reduce((a,b) => a + b, 0);
console.log(somatorio);
```





## Exemplo

- Calcular o somatório de um array de números.
- O quê acontece se o array for vazio e sem um valor inicial informado?

```
let numeros = [];
let somatorio = numeros.reduce((a,b) => a + b);
console.log(somatorio);
```



# Arrays - Desestruturando

- A operação de atribuição pode utilizar um modo de “desestruturação” que permite funcionalidades interessantes
  - A ideia é “desempacotar” um array em vários “pedaços”

```
let arr = ['Julio', 'Machado'];
let [primeiroNome, segundoNome] = arr;
console.log(primeiroNome);
```

# Arrays - Desestruturando

- Exemplo:

- Ignorar elementos do início

```
let arr = ['Julio', 'Machado'];
let [ , ultimoNome] = arr;
```

- Desestruturar em sub-pedaços com "..."

```
let arr = ['Julio', 'Machado', 'Professor', 'PUCRS'];
let [ primeiroNome, ultimoNome, ...info] = arr;
console.log(info.length); //2
```

# Arrays - Espalhando

- A operação de atribuição pode utilizar um modo de “espalhamento” que permite inserir um array em outro array

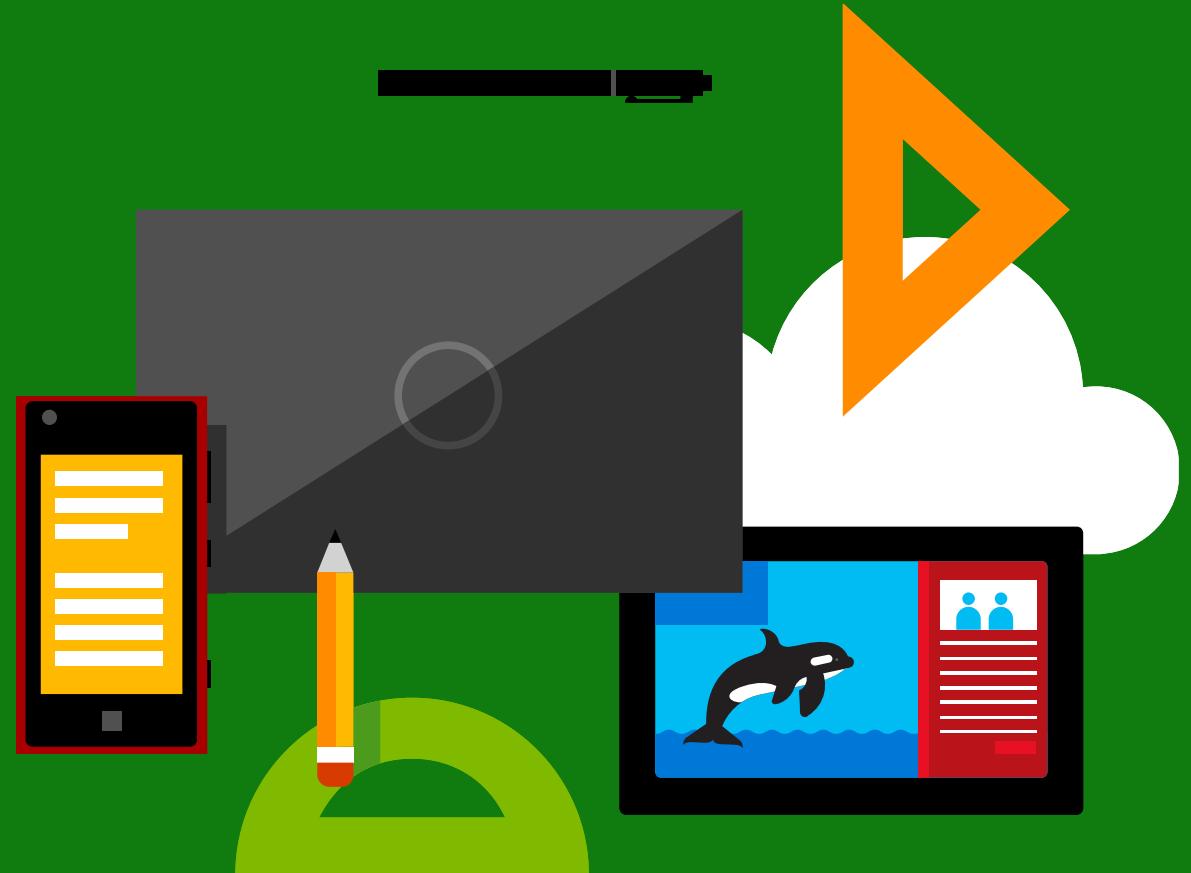
```
let arr1 = [1, 2];
let arr2 = [3, 4];
let arr3 = [0, ...arr1, ...arr2, 5];
console.log(primeiroNome);
```

# Arrays Multidimensionais

- Arrays podem armazenar qualquer tipo de objeto, inclusive outros arrays
- Arrays multidimensionais são arrays de arrays
  - Útil para implementar o conceito matemático de matriz

```
let m: number[][] = [
  [1,2,3],
  [4,5,6],
  [7,8,9]
];
console.log(m.length);
console.log(m[0].length);
console.log(m[1][2]);
```

# Tuplas



# Tuplas

- TypeScript suporta o conceito de tuplas através da sintaxe de array
- Declara-se o tipo de cada elemento da tupla
- Ex.:

```
let tupla: [string, number];  
tupla = ['TypeScript', 1];
```

# Coleções



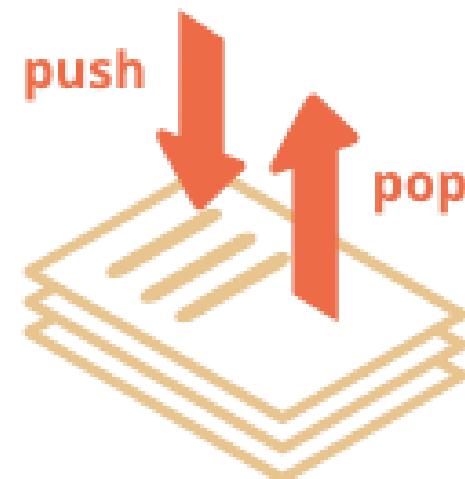
# Listas

- Listas são implementadas diretamente sobre Arrays e seus métodos
- Basta não permitir que posições *undefined* estejam presentes, de forma que a propriedade **length** seja equivalente à contagem do número de elementos dentro do array
  - Lembre-se que a propriedade length é de leitura/escrita e corresponde sempre ao índice de maior valor somado a um

# Pilhas

- Arrays fornecem métodos para manipular seus elementos como uma coleção do tipo pilha (*LIFO – last in, first out*)
  - push(item) – adiciona elemento ao final do array e retorna o novo tamanho do array
  - pop() – remove e retorna o último elemento do array, diminuindo o tamanho do array de uma posição

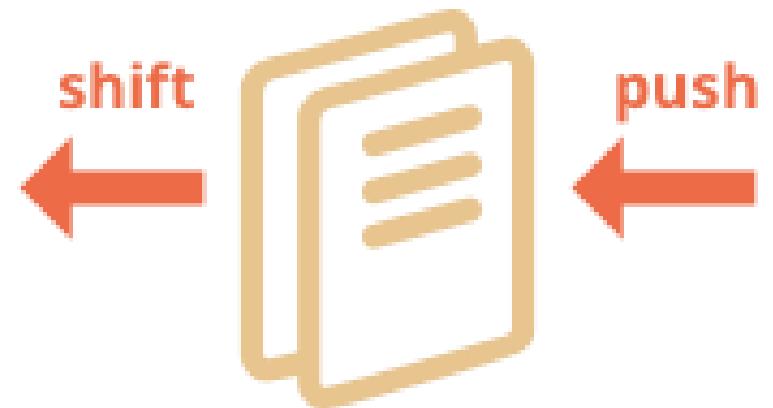
```
let pilha = [1, 2, 3];
console.log(pilha.push(4));
console.log(pilha.pop());
```



# Filas

- Arrays fornecem métodos para manipular seus elementos como uma coleção do tipo fila (*FIFO – first in, first out*)
  - push(item) – adiciona elemento ao final do array e retorna o novo tamanho do array
  - shift() – remove e retorna o primeiro elemento do array, reposicionando os demais elementos através de deslocamento, diminuindo o tamanho do array de uma posição

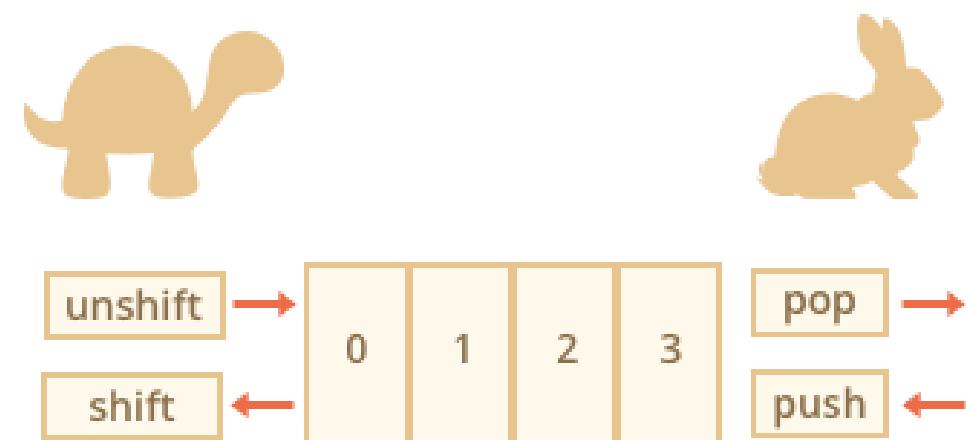
```
let fila = [1, 2, 3];
console.log(fila.push(4));
console.log(fila.shift());
```



# Filas de Extremidade Dupla

- Arrays fornecem métodos para manipular seus elementos como uma coleção do tipo fila de extremidade dupla (*DEQUE – double-ended queue*)
  - push(item) – adiciona elemento ao final do array e retorna o novo tamanho do array
  - pop() – remove e retorna o último elemento do array, diminuindo o tamanho do array de uma posição
  - unshift(item) – adiciona elemento ao início do array e retorna o novo tamanho do array
  - shift() – remove e retorna o primeiro elemento do array, reposicionando os demais elementos através de deslocamento, diminuindo o tamanho do array de uma posição

```
let deque = [1, 2, 3];
console.log(deque.push(4));
console.log(deque.pop());
console.log(deque.unshift(0));
console.log(deque.shift());
```



# Mapas

- São coleções de valores indexadas por chaves, implementadas no objeto **Map<K, V>**
- Abstraem o armazenamento de pares “chave/valor”, onde ambos podem ser qualquer tipo do TypeScript
  - A chave deve ser um valor único na coleção para cada par
    - Os valores de chaves são comparados através do algoritmo [SameValueZero](#), semelhante ao comparador estrito de igualdade ===

# Mapas

- Propriedades:

- size – informa a contagem de elementos

- Métodos:

- new Map() – construtor de mapas
  - set(chave,valor) – armazena o par chave/valor
  - get(chave) – retorna o valor armazenado na chave
  - has(chave) – retorna true se existe a chave, false caso contrário
  - delete(chave) – remove o par chave/valor
  - clear() – esvazia o mapa
  - keys() – retorna um objeto iterável sobre a coleção de chaves
  - values() – retorna um objeto iterável sobre a coleção de valores
  - entries() – retorna um objeto iterável sobre pares [chave,valor]

# Mapas

## Exemplo

```
let mapa = new Map<string,string>();
mapa.set("RS", "Rio Grande do Sul");
mapa.set("SC", "Santa Catarina");
mapa.set("PR", "Paraná");
console.log(mapa.get("RS"));
for(let sigla of mapa.keys()) {
    console.log(sigla);
}
for(let estado of mapa.values()) {
    console.log(estado);
}
for(let entrada of mapa.entries()) {
    console.log(entrada);
}
```

# Conjuntos

- O conceito matemático de conjunto (uma coleção sem elementos repetidos) é fornecida pelo objeto **Set<T>**
- Propriedades:
  - size – informa a contagem de elementos
- Métodos:
  - new Set() – construtor de conjuntos
  - add(item) – adiciona um elemento ao conjunto, retornando o próprio conjunto
  - delete(item) – remove o elemento do conjunto, retornando true se o elemento estava no conjunto ou false caso contrário
  - has(item) – retorna true se o elemento pertence ao conjunto, false caso contrário
  - clear() – esvazia o conjunto
  - values() – retorna um iterador sobre a coleção

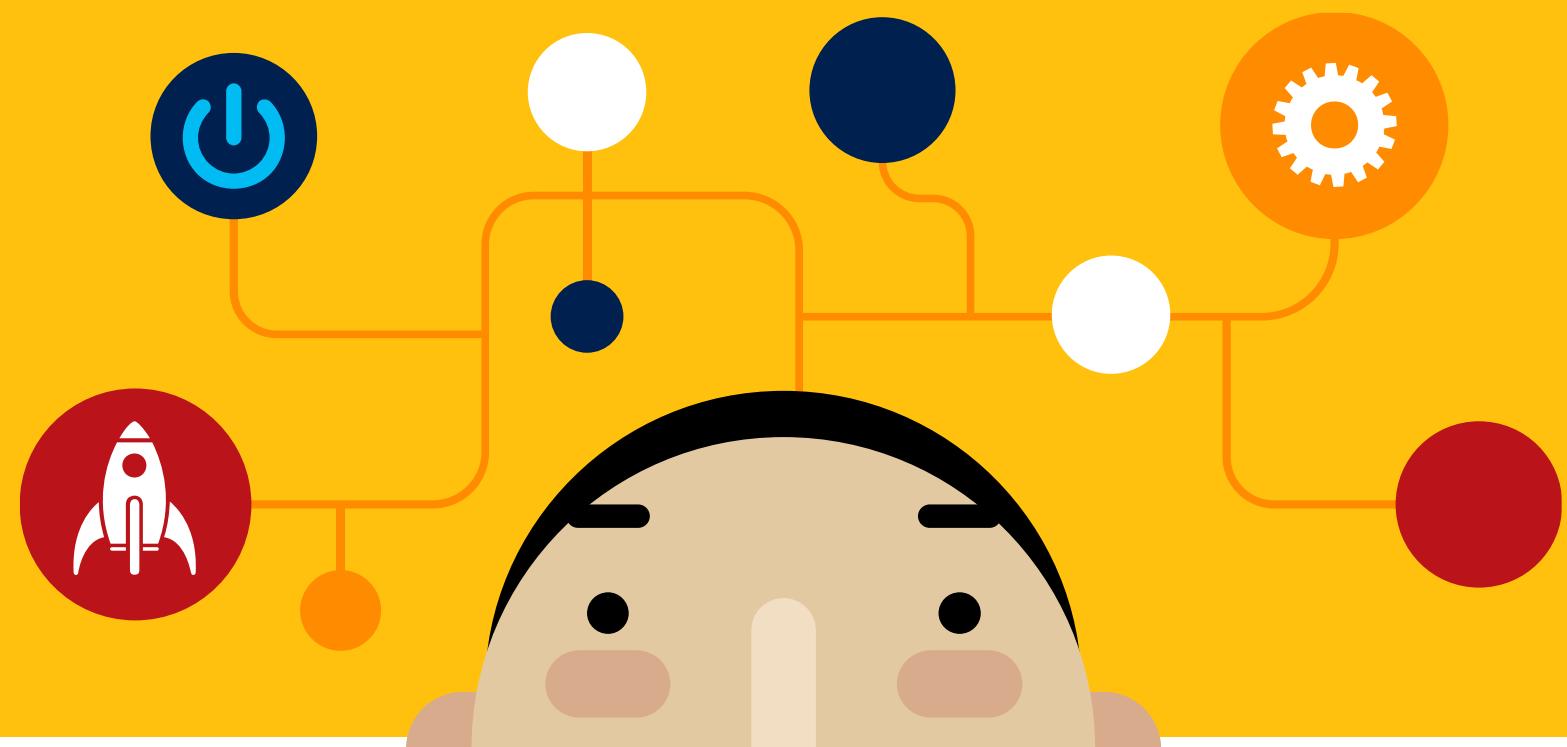
# Conjuntos

## Exemplo

```
let conjunto = new Set<number>();
conjunto.add(1);
conjunto.add(1);
conjunto.add(2);
console.log(conjunto.size);
conjunto.forEach(x => console.log(x));
```

# Laboratório

- Abra as instruções do arquivo Lab02\_TypeScript\_Introducao



# Iteradores e Geradores



# Iteradores

- Iteradores são objetos que mantém um “cursor” para elementos de uma coleção de dados.
- Quando um iterador é criado, aponta para o primeiro elemento da coleção.
- A cada solicitação “next” retorna o valor armazenado pelo próximo elemento da coleção e avança o cursor.
- Iteradores abstraem o acesso sequencial a qualquer tipo de estrutura de dados.
- Permitem acessar os valores armazenados sem liberar o acesso para que a coleção seja modificada ou que o tipo de coleção seja exposto.



- A próxima operação `next()` irá retornar o valor 10 e avançar o cursor para a próxima posição.
- Quando chegar ao final da coleção irá retornar **`undefined`**.

# Iteradores

- Mas de onde vem a ideia de utilizar iteradores?
- Solução catalogada como um padrão de projeto!
- Saiba mais:
  - GAMMA, E.; JOHNSON, R.; VLISSIDES, J.M.; HELM, R.; FOWLER, M. Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2007.

# Padrão *Iterator*

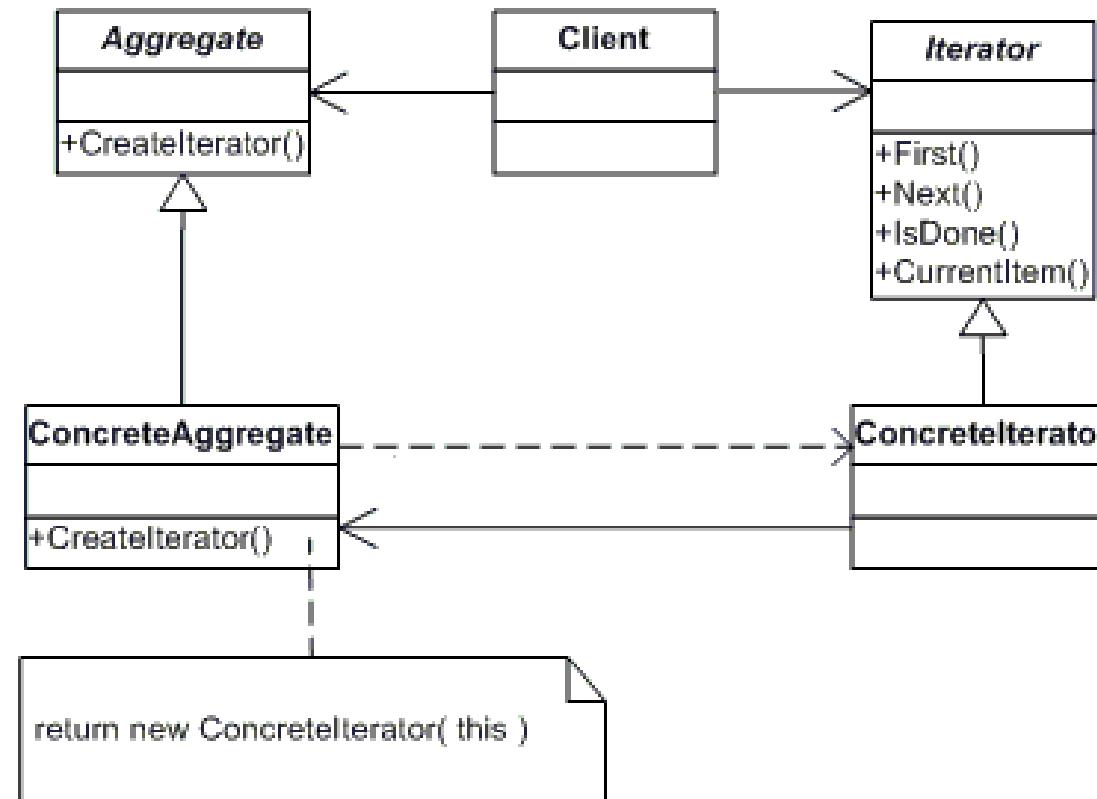
- Provê uma forma de acessar os elementos de um objeto agregado sequencialmente sem expor a sua implementação interna.

# Padrão *Iterator*: definição

- Classificação: comportamental de objeto
- Contexto:
  - Um objeto (chamado de agregador) contém outros objetos (chamados de elementos).
  - Clientes necessitam acessar os elementos.
  - O agregador não deve expor sua estrutura interna.
  - Podem haver vários clientes que necessitam acesso simultâneo.
- Solução:
  - Definir uma classe iteradora que retorna um elemento de cada vez.
  - Cada objeto iterador necessita manter a posição do próximo elemento a retornar.
  - Se existirem diferentes variações de agregadores e iteradores, é melhor que todos implementem uma interface comum.

# Padrão Iterator: diagrama

UML class diagram



# Padrão *Iterator*: JavaScript

- A especificação ECMAScript define a noção de iteradores através de um conjunto de interfaces (embora, concretamente, interfaces não existam como elementos programáveis na linguagem):
  - *Iterable*
  - *Iterator*
  - *IteratorResult*
- Conceitualmente, elas definem um protocolo que deve ser seguido na implementação do padrão em JavaScript.

# Padrão *Iterator*: JavaScript

```
<<Interface>>  
Iterable
```

```
+ @@iterator(): Iterator
```

- `@@iterator` representa a chave de uma propriedade definida pelo símbolo global `[Symbol.iterator]`.
- O objeto *Iterator* deve ser fornecido por uma função responsável por criar o iterador adequado.
- Vários objetos da API já implementam *Iterable*:
  - Strings, arrays, maps, sets.

- **Símbolo (Symbol)** é um tipo primitivo da linguagem JavaScript que representa um identificador único.
  - Nenhum símbolo é igual a outro símbolo.
  - Usualmente são utilizados para descrever nomes de propriedades de forma única.

# Padrão Iterator: JavaScript

## Consumidor de dados

Comando for...of

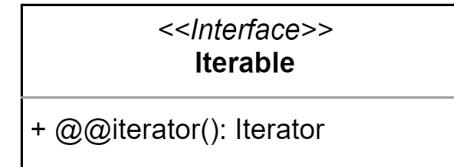
Operador spread ...

Desestruturação de array

Construtor de Set e Map

Método Array.from()

etc



## Fonte de dados

Arrays

Strings

Maps

Sets

etc





## Exemplo

- Iterando sobre um array com *for...of*:

```
const umArray = ['a', 'b', 'c'];
for(const c of umArray) {
  console.log(c);
}
```



## Exemplo

- Iterando sobre um conjunto com *for..of*:

```
const umConjunto = new  
Set().add('a').add('b').add('c');  
for(const c of umConjunto) {  
    console.log(c);  
}
```



## Exemplo

- Iterando sobre um dicionário com *for...of*:

```
const umDicionario = new  
Map().set('a','Argentina').set('b','Brasil').set('c','Chile');  
for(const [chave,valor] of umDicionario) {  
    console.log(` ${chave}: ${valor}`);  
}
```



## Exemplo

- Iterando sobre uma string com *for...of*:

```
const umaString = 'abc';
for(const c of umaString) {
  console.log(c);
}
```



## Exemplo

- Desestruturação do padrão de array:

```
const umaString = 'abc';
const [c1,c2] = umaString;
console.log(c1);
console.log(c2);
```



## Exemplo

- Operador spread ...:

```
const umConjunto = new Set().add('a').add('b').add('c');
const umArray = [...umConjunto];
console.log(umArray);
```



## Exemplo

- Método `Array.from()`:

```
const umConjunto = new Set().add('a').add('b').add('c');
const umArray = Array.from(umConjunto);
console.log(umArray);
```



## Exemplo

- Construtores de coleções a partir de outras coleções:

```
const umArray = ['a', 'a', 'b', 'c'];
const umConjunto = new Set(umArray);
console.log(umConjunto);
```



## Exemplo

- Outros métodos que retornam iterable:

```
const umDicionario = new  
Map().set('a','Argentina').set('b','Brasil').set('c','Chile');  
for(const c of umDicionario.keys()) {  
    console.log(c);  
}  
for(const v of umDicionario.values()) {  
    console.log(v);  
}
```

# Padrão *Iterator*: JavaScript

```
<<Interface>>  
Iterator
```

```
+ next(valor): IteratorResult  
+ return(valor): IteratorResult  
+ throw(exceção): IteratorResult
```

- *next()*
  - Função obrigatória.
  - É a responsável por “avançar” a iteração passo a passo.
  - Aceita zero ou mais argumentos.
    - Não passar argumentos é o mais usual.
    - Significado dos argumentos é dependente da implementação do iterador.
  - Retorna um objeto *IteratorResult* contendo o resultado do passo de iteração.

# Padrão *Iterator*: JavaScript

<<Interface>>  
**Iterator**

- + next(valor): IteratorResult
- + return(valor): IteratorResult
- + throw(exceção): IteratorResult

- *return()*
  - Função opcional.
  - Utilizada para “fechar” o iterador de forma explícita, finalizando a iteração.
  - Aceita zero ou um argumento.
    - Não passar argumento é o mais usual.
    - Significado do argumento é dependente da implementação do iterador.
  - Retorna um objeto *IteratorResult* contendo o resultado do passo de iteração indicando que o processo de iteração terminou.

# Padrão *Iterator*: JavaScript

<<Interface>>  
**Iterator**

+ next(valor): IteratorResult  
+ return(valor): IteratorResult  
+ throw(exceção): IteratorResult

- *throw()*
  - Função opcional.
  - Utilizada para indicar uma “condição de erro” para o iterador.
  - Aceita zero ou um argumento.
    - Significado do argumento é dependente da implementação do iterador mas usualmente é um objeto de exceção.
    - Retorna um objeto *IteratorResult* contendo o resultado do passo de iteração indicando que o processo de iteração terminou ou lança uma exceção.

# Padrão *Iterator*: JavaScript

```
<<Interface>>
IteratorResult
```

```
+ done: boolean
+ value: object
```

- *done*
  - Valor *false* indica que o iterador foi capaz de produzir um novo valor.
    - O valor estará disponível na propriedade *value*.
  - Valor *true* indica que o iterador finalizou a produção de valores.
- *value*
  - Identifica o valor produzido pelo iterador.
  - Se o valor de *done* for *true*, é usualmente omitido (ou *undefined*) ou é o valor passado ao método *return()* do iterador.



## Exemplo

- Iterando sobre um array de forma explícita:

```
const umArray = ['a', 'b', 'c'];
const iterator = umArray[Symbol.iterator]();
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

Acessando o método de fábrica do iterador através do símbolo `Symbol.iterator`.

```
{ value: 'a', done: false }
{ value: 'b', done: false }
{ value: 'c', done: false }
{ value: undefined, done: true }
```



## Exemplo

- Iterando sobre um array de forma explícita com laço:

```
const umArray = ['a', 'b', 'c'];
const iterator = umArray[Symbol.iterator]();
let resultado = iterator.next();
while(!resultado.done) {
  console.log(resultado.value);
  resultado = iterator.next();
}
```

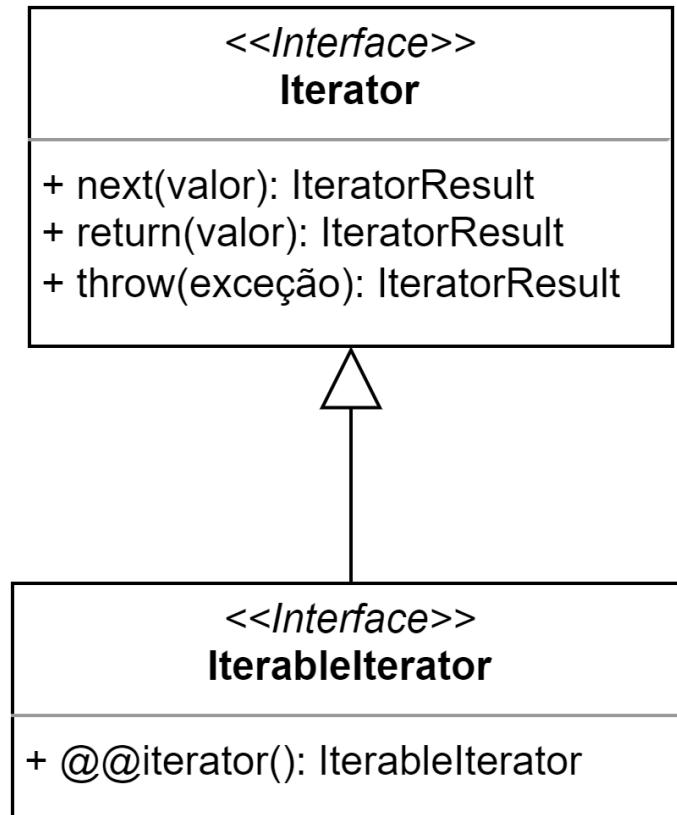


## Exemplo

- Iterando sobre um array de forma explícita com laço:

```
const umArray = ['a', 'b', 'c'];
const iterator = umArray[Symbol.iterator]();
let resultado;
while(!(resultado = iterator.next()).done) {
  console.log(resultado.value);
}
```

# Padrão *Iterator*: JavaScript



- A maioria dos iteradores presentes na API de JavaScript implementam tanto a interface *Iterator* quanto *Iterable*.
- Recebem o nome de “iteradores iteráveis”.



## Exemplo

- Iterando sobre um iterador iterável:

```
const umArray = ['a', 'b', 'c'];
const iteratorIterable = umArray[Symbol.iterator]();
for(const c of iteratorIterable) {
  console.log(c);
}
```

# Padrão Iterator: TypeScript

```
interface IteratorYieldResult<TYield> {
    done?: false;
    value: TYield;
}

interface IteratorReturnResult<TReturn> {
    done: true;
    value: TReturn;
}

type IteratorResult<T, TReturn = any> = IteratorYieldResult<T> |  
IteratorReturnResult<TReturn>;  
  
interface Iterator<T, TReturn = any, TNext = undefined> {  
  
    // NOTE: 'next' is defined using a tuple to ensure we report the cor  
    // rect assignability errors in all places.  
    next(...args: [] | [TNext]): IteratorResult<T, TReturn>;  
    return?(value?: TReturn): IteratorResult<T, TReturn>;  
    throw?(e?: any): IteratorResult<T, TReturn>;
}
```

# Padrão Iterator: TypeScript

```
interface Iterable<T> {  
    [Symbol.iterator](): Iterator<T>;  
}
```

```
interface IterableIterator<T> extends Iterator<T> {  
    [Symbol.iterator](): IterableIterator<T>;  
}
```

# Geradores

- Geradores são objetos que estão de acordo com o protocolo das interfaces *Iterable* e *Iterator*, que adicionalmente possuem a característica de permitir sair e entrar novamente em um bloco de código sem perder o contexto.
- Além disso, enquanto iteradores somente produzem valores, geradores são de mão-dupla: podem produzir e consumir valores.

# Geradores

- Uma função geradora, denota pelo modificador `*`, é responsável pela criação de objetos geradores.
  - Um função *arrow* não pode ser uma função geradora.
  - Todas as demais construções de funções/métodos podem definir uma função geradora.

```
function* genFunc1() { /*...*/ }
```

```
const genFunc2 = function* () { /*...*/ };
```

```
const obj = {
  * generatorMethod() { /*...*/ }
};
```

```
class MyClass {
  * generatorMethod() { /*...*/ }
}
```

# Geradores

- O corpo de uma função geradora faz uso de operadores especiais: *yield* e *yield\**
- Protocolo de funcionamento:
  - Geradores iniciam em um estado de “suspenção”;
  - O operador *yield* no corpo da função geradora implementa o ponto de controle da “suspenção”, pois o gerador suspende sua execução ao encontrar o operador *yield*;
  - Método *next()* instrui o gerador a iniciar ou retomar a execução;
    - O valor associado ao operador *yield* é o valor retornado por *next()* do iterador.
  - Função geradora “saindo” com:
    - *yield* – configura *done=false*
    - *return* – configura *done=true*



## Exemplo

```
function* funcaoGeradora() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
const gerador = funcaoGeradora();  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());
```

```
{ value: 1, done: false }  
{ value: 2, done: false }  
{ value: 3, done: false }  
{ value: undefined, done: true }
```



## Exemplo

```
function* funcaoGeradora() {  
    yield 1;  
    yield 2;  
    return 3;  
}  
const gerador = funcaoGeradora();  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());
```

```
{ value: 1, done: false }  
{ value: 2, done: false }  
{ value: 3, done: true }  
{ value: undefined, done: true }
```



## Exemplo

```
function* funcaoGeradora() {  
    let valor = 1;  
    while(valor <= 3) {  
        yield valor++;  
    }  
}  
  
for(const i of funcaoGeradora()) {  
    console.log(i);  
}
```



A white rectangular box with a thin gray border, containing three large, bold, colored numbers stacked vertically: '1' (brown), '2' (blue), and '3' (brown).

1  
2  
3



## Exemplo

- Números pseudoaleatórios podem ser gerados com base em um algoritmo bastante simples, conhecida como *Gerador Congruente Linear Multiplicativo*.
- O gerador é definido pela relação de recorrência:

$$X_{n+1} = a * X_n \text{ mod } m$$

- Onde:
  - $a$  é uma constante chamada multiplicador, sendo  $0 < a < m$
  - $m$  é uma constante chamada módulo, sendo  $0 < m$ , usualmente um número primo ou uma potência de um número primo
  - $\text{mod}$  é a operação de resto da divisão
  - $X_0$  é a semente do gerador



## Exemplo

```
function* pseudoRandom(seed) {  
    let value = seed;  
    const a = 48271;  
    const m = 2147483647;  
    while (true) {  
        yield value = value * a % m;  
    }  
}
```

```
const gerador = pseudoRandom(123456);  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());  
console.log(gerador.next());
```

# Geradores

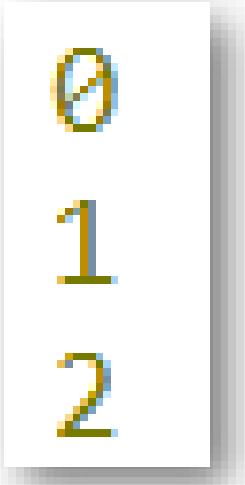
- Geradores podem consumir valores:
  - Um valor pode ser passado para o gerador através do método *next()*;
  - O valor recebido é utilizado como resultado do próximo operador *yield* no fluxo de código da função geradora.



## Exemplo

```
function* geradorLogs() {  
    console.log(0);  
    const valor1 = yield;  
    console.log(valor1);  
    const valor2 = yield;  
    console.log(valor2);  
}  
  
const gerador = geradorLogs();  
gerador.next();  
gerador.next(1);  
gerador.next(2);
```

Yield recebe um valor através do método next().



```
0  
1  
2
```



## Exemplo

```
function* somaUltimosTres() {
  const ultimosTres = [];
  let soma = 0;
  while (true) {
    const valor = yield soma;
    ultimosTres.push(valor);
    soma += valor;
    if (ultimosTres.length > 3) {
      soma -= ultimosTres.shift();
    }
  }
}
```

```
const gerador = somaUltimosTres();
console.log(gerador.next());
console.log(gerador.next(1));
console.log(gerador.next(2));
console.log(gerador.next(3));
console.log(gerador.next(4));
console.log(gerador.next(5));
```

```
{ value: 0, done: false }
{ value: 1, done: false }
{ value: 3, done: false }
{ value: 6, done: false }
{ value: 9, done: false }
{ value: 12, done: false }
```

# Geradores

- Geradores podem delegar execução a outros geradores/iteradores:
  - Utilizar o operador *yield*\*;
  - O operador *yield*\* atua sob um objeto iterável qualquer, quer seja um gerador obtido a partir de uma função geradora ou não;
  - Pode ser utilizado recursivamente sobre si mesmo!



## Exemplo

```
function* nVezes(n) {  
  if (n > 0) {  
    yield* nVezes(n-1);  
    yield n - 1;  
  }  
}  
  
for (const v of nVezes(3)) {  
  console.log(v);  
}
```



0  
1  
2

# Geradores

- Geradores criados a partir de funções geradoras já implementam as funções opcionais `return()` e `throw()` da interface `Iterator`.
- Ambas funções gerenciam a capacidade de “finalizar/fechar” um gerador.
  - `return()` irá colocar o gerador em um estado de finalizado e o valor fornecido será o valor retornado na última iteração com `done=true`;
  - `throw()` irá injetar uma exceção no bloco de código da função geradora no ponto em que ela foi suspensa e, se não tratado, irá colocar o gerador em um estado de finalizado, além de fazer com que a exceção migre para o código que está consumindo o gerador.

# Padrão *Iterator*: JavaScript

<<Interface>>  
**Iterator**

- + next(valor): IteratorResult
- + return(valor): IteratorResult
- + throw(exceção): IteratorResult

- *return()*
  - Função opcional.
  - Utilizada para “fechar” o iterador de forma explícita, finalizando a iteração.
  - Aceita zero ou um argumento.
    - Não passar argumento é o mais usual.
    - Significado do argumento é dependente da implementação do iterador.
  - Retorna um objeto *IteratorResult* contendo o resultado do passo de iteração indicando que o processo de iteração terminou.

# Padrão *Iterator*: JavaScript

<<Interface>>  
**Iterator**

+ next(valor): IteratorResult  
+ return(valor): IteratorResult  
+ throw(exceção): IteratorResult

- *throw()*
  - Função opcional.
  - Utilizada para indicar uma “condição de erro” para o iterador.
  - Aceita zero ou um argumento.
    - Significado do argumento é dependente da implementação do iterador mas usualmente é um objeto de exceção.
    - Retorna um objeto *IteratorResult* contendo o resultado do passo de iteração indicando que o processo de iteração terminou ou lança uma exceção.



## Exemplo

```
function* funcaoGeradora() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
const gerador = funcaoGeradora();  
console.log(gerador.next());  
console.log(gerador.return(22));  
console.log(gerador.next());
```

```
{ value: 1, done: false }  
{ value: 22, done: true }  
{ value: undefined, done: true }
```



## Exemplo

```
function* funcaoGeradora() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
const gerador = funcaoGeradora();  
console.log(gerador.next());  
try {  
    console.log(gerador.throw(new Error('Erro')));  
} catch (error) {  
    console.error(error.message);  
}  
console.log(gerador.next());
```

```
{ value: 1, done: false }  
Erro  
{ value: undefined, done: true }
```



## Exemplo

```
function* funcaoGeradora() {
  for (const v of [1,2,3]) {
    try {
      yield v;
    } catch (error) {
      console.error(error.message);
    }
  }
}

const gerador = funcaoGeradora();
console.log(gerador.next());
console.log(gerador.throw(new
Error('Erro')));
console.log(gerador.next());
```

```
{ value: 1, done: false }
Erro
{ value: 2, done: false }
{ value: 3, done: false }
```

# Geradores: TypeScript

```
interface Generator<T = unknown, TReturn = any, TNext = unknown> extends Iterator<T, TReturn, TNext>
{
    // NOTE: 'next' is defined using a tuple to ensure we report the correct assignability errors in all places.
    next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
    return(value: TReturn): IteratorResult<T, TReturn>;
    throw(e: any): IteratorResult<T, TReturn>;
    [Symbol.iterator](): Generator<T, TReturn, TNext>;
}
```