

# DBStart

JavaScript e TypeScript

Instrutora: Profa. Andréa Aparecida Konzen ([andrea.konzen@pucrs.br](mailto:andrea.konzen@pucrs.br))

*Material cedido gentilmente pelo Prof. Júlio Pereira Machado*



# JavaScript

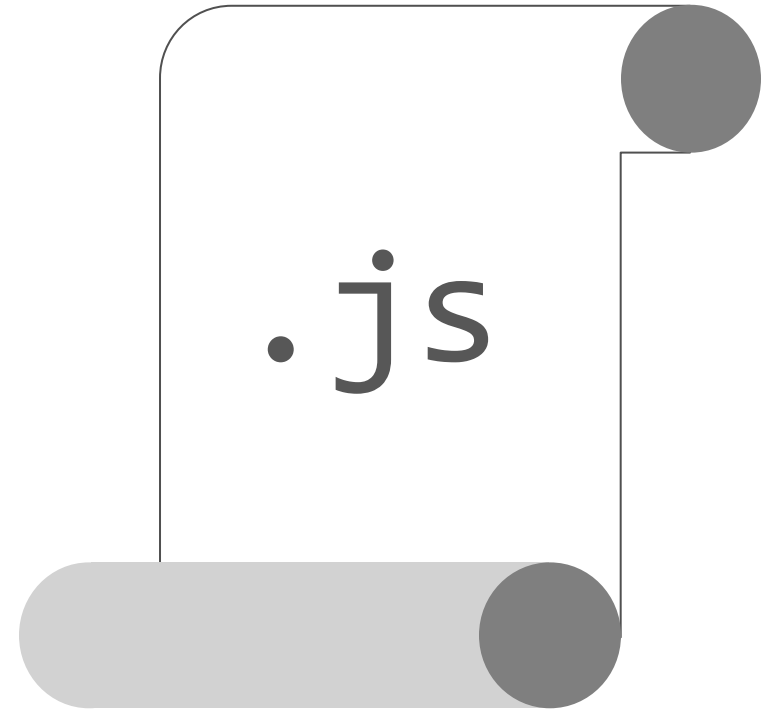


# Aplicações Interativas na Web

- 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

- 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

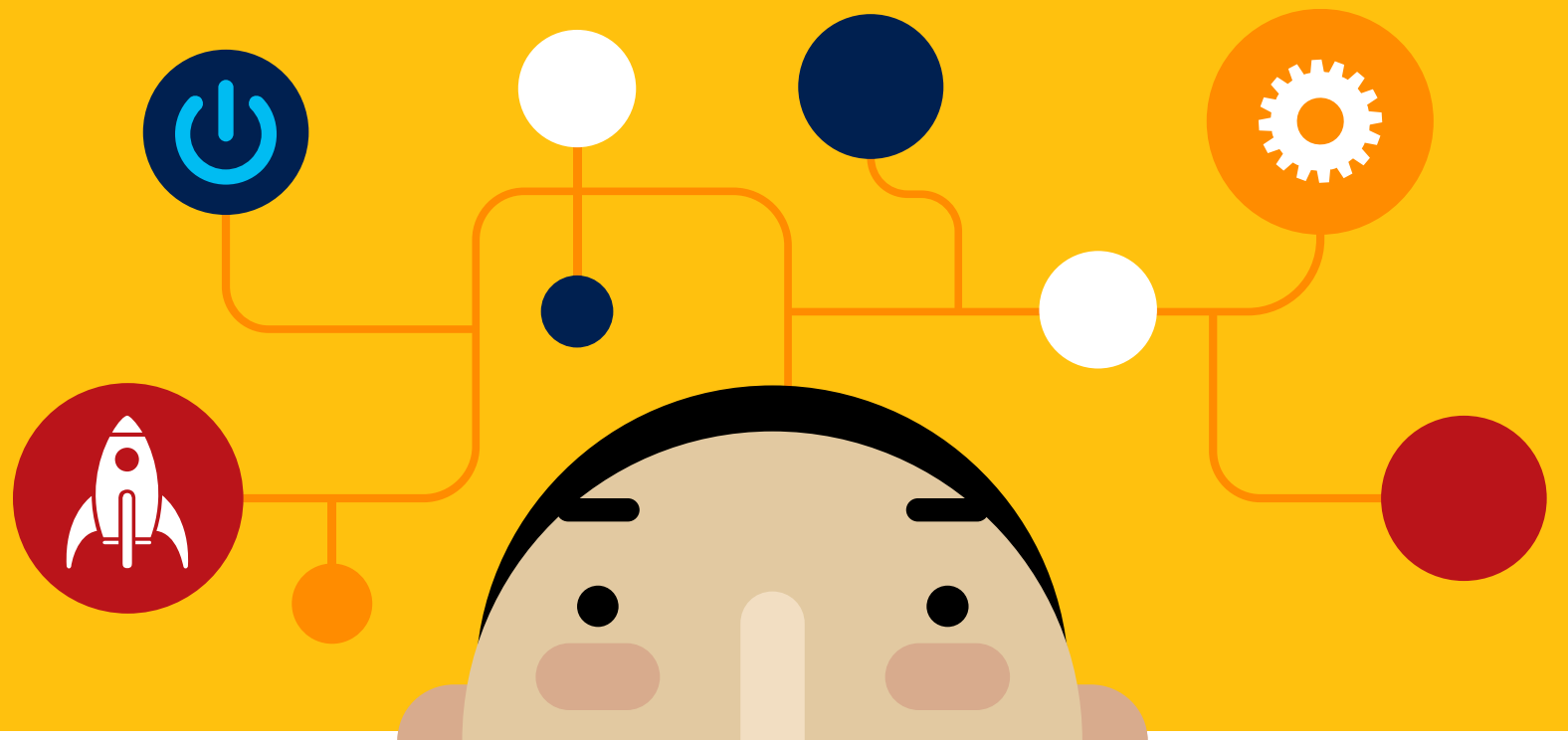


# TypeScript

- 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

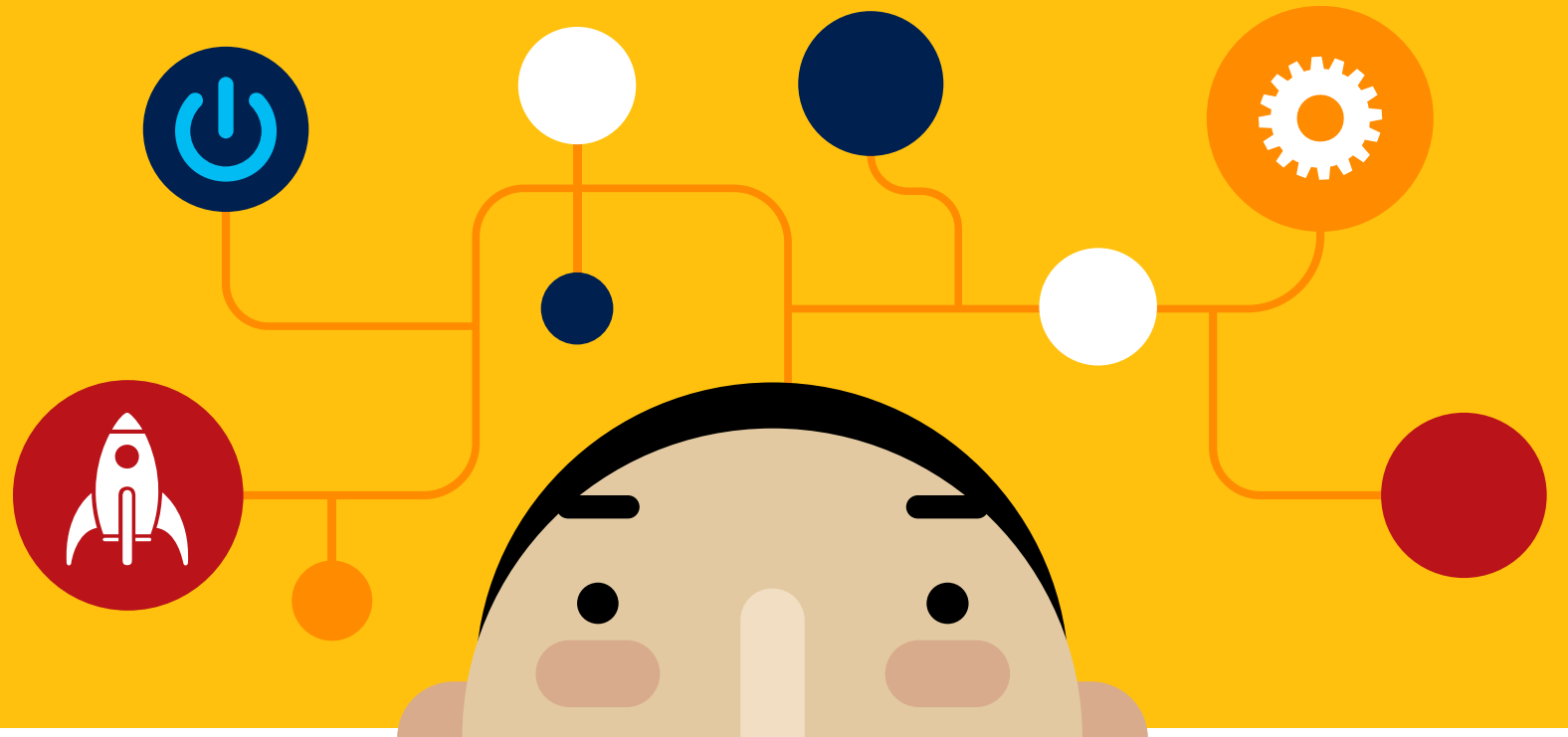
- Abra as instruções do arquivo Lab00\_Ambiente



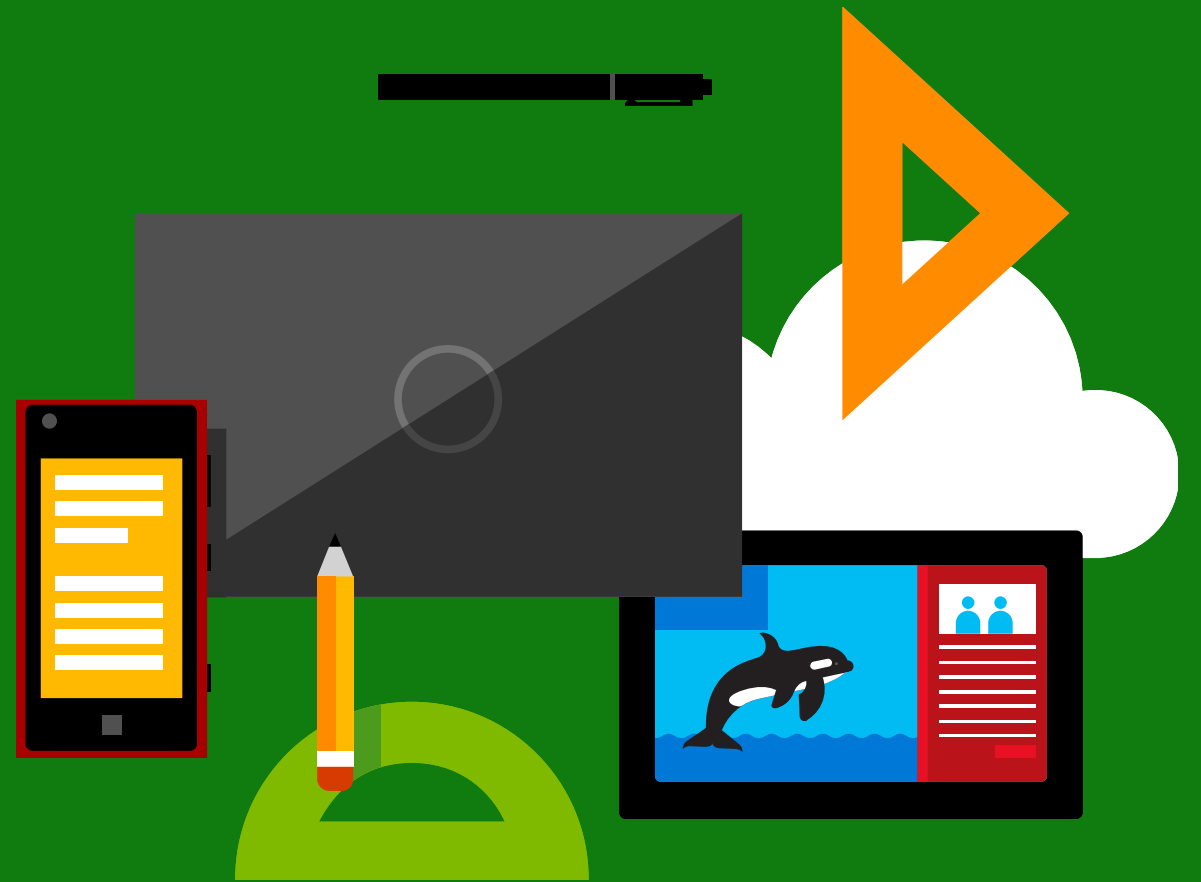


# Laboratório

- 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';

palavra-chave   nome da constante   tipo   operador de atribuição   valor da constante

# 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{NNNNNNNNNN}	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

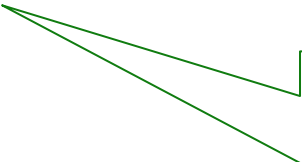
```
for (let i = 0; i<3; i++) {  
  console.log(i);  
}
```

# 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, fornece um iterador.

# Comandos – FOR..OF

## Exemplo

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

# Funções



# Funções

- Uma **função** é um agrupamento de comandos que realizam uma tarefa específica
- Se diferentes partes de um script repetem a mesma tarefa, então o código pode ser estruturado para utilizar uma função contendo os comandos que se repetem

# Funções

- Funções são definidas com a palavra reservada *function*
- Uma função pode possuir um nome
  - Existe o conceito de função anônima
- Uma função pode ter argumentos e retornar valor
  - 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!

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

# Funções

palavra-chave                      nome da função                      tipo da função

```
function helloWorld(): void {  
    alert('Hello, World!');  
}
```

comando

bloco de definição da função

The diagram illustrates the components of a JavaScript function definition. The code is: `function helloWorld(): void { alert('Hello, World!'); }`. Labels with leader lines point to specific parts: 'palavra-chave' (keyword) points to 'function'; 'nome da função' (function name) points to 'helloWorld'; 'tipo da função' (function type) points to ': void'; 'comando' (command) points to the line 'alert('Hello, World!');'; and 'bloco de definição da função' (function definition block) points to the entire function block from 'function' to the closing brace '}'.



# Funções

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

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

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

# Funções

```
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));
```

# Definição e Execução de Funções

- O modo como uma função é definida é diferente do modo como é executada
- A definição da função descreve seu nome, parâmetros e comandos
  - **NOTA:** a definição não executa nenhum dos comandos
- Quando a função é chamada, irá executar o bloco de comandos definidos na função

## Defining the Function

```
function doSomethingAwesome() {  
    var name = prompt("What is your name?");  
    alert(name + ", you just did something awesome!");  
}
```

# Definição e Execução de Funções

- O modo como uma função é definida é diferente do modo como é executada
- A definição da função descreve seu nome, parâmetros e comandos
  - **NOTA:** a definição não executa nenhum dos comandos
- Quando a função é chamada, irá executar o bloco de comandos definidos na função

## Calling the Function

```
<input type="button" value="Click Me"  
      onclick="doSomethingAwesome()">
```

# Funções Manipuláveis

- Em JavaScript, funções podem ser manipuladas assim como valores
- Pode-se atribuir a variáveis, passar a função como parâmetro para outra função, retornar uma função como valor de retorno de outra função, etc
- Uma forma alternativa de definir funções é através de funções anônimas:

```
let identificador: tipo = function (lista de parâmetros): tipo { bloco de comandos };
```

- Outra forma alternativa de definir funções é através de “expressões lambda”:

```
let identificador = (lista de parâmetros) => expressão;
```

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

# Funções Manipuláveis

## Exemplo

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

# Funções Manipuláveis

## Exemplo

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

# Funções Manipuláveis

Exemplo (closure)

```
function multiplicar (fator: number): (f: number) => number {  
  return numero => numero * fator;  
}
```

```
let dobrar = multiplicar(2);  
console.log(dobrar(5));
```



# 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.find(function(item){
  return item >= 0 && item <= 2;
});
```

# Arrays - Métodos

- Métodos:

- `forEach(funçãoAplicação)` – 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;  
});
```

# 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



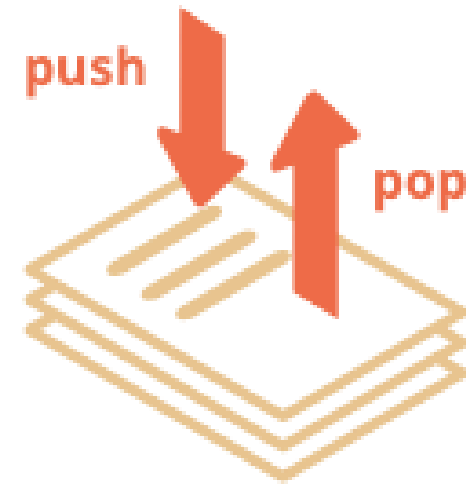
# Listas

- 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());
```

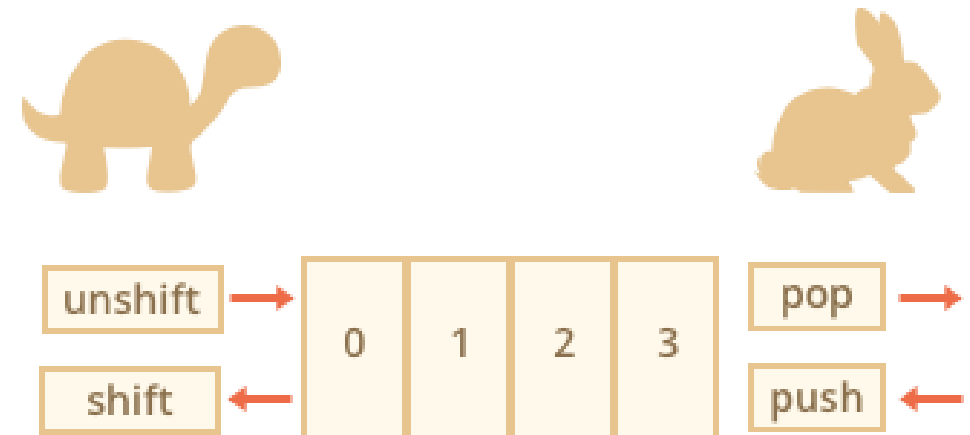


# Filas de Extremidade Dupla

• 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
  - O 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

