Curso Superior de Tecnologia em Sistemas para Internet
Desenvolvimento front-end I (front 1)
Rafael Cunha Cardoso
rafaelcardoso@ifsul.edu.br

Declarações e Estruturas

Declarações

Entende-se por **declaração**, **uma linha ou um bloco de comando** que compõem um script.

Declaração	Finalidade
`var, let, const`	Declarar uma variável
`function`	Declarar uma função
`return`	Retornar um valor
`if/else`	Criar estrutura condicional
`switch`	Criar estrutura condicional
`case`	Usado na estrutura switch
`break`	Usado na estrutura switch
`default`	Usado na estrutura switch

Declaração	Finalidade
`for`	Cria um loop
`continue`	Reiniciar um loop
`while`	Cria estrutura de repetição
`do/while`	Cria estrutura de repetição
`for/in`	Cria loop em objeto
`throw`	Sinalizar erros
`try/catch/finally`	Tratar erros
`with`	Alterar o escopo
`;`	Declaração vazia

var

Declara **explicitamente** uma ou mais variáveis.

Sintaxe

```
var nome1 [=valor1], [nome2 = valor2], …, [nomeN = valorN]
```

Exemplos

```
var x;

var x = 6;

var x, y, z;

var x = 2, y = 5, z = 10;
```

Escopo de variáveis declaradas como var

Escopo, essencialmente, significa **onde essas variáveis poderão ser utilizadas.**

Declarações com **var** tem **escopo global** ou **escopo de função/local**.

O escopo é **global** quando uma variável **var** é declarada **fora de uma função**.

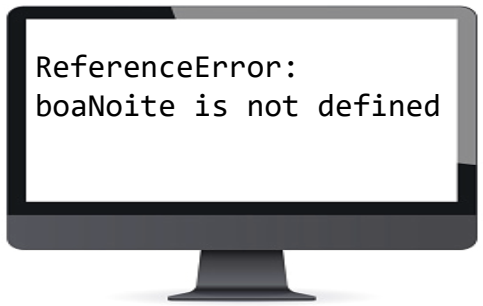
- **Significa que qualquer variável que seja declarada com var fora de um bloco de função pode ser utilizada na janela inteira.**

var tem escopo de **função** quando é declarada **dentro de uma função**.

- **Significa que a variável está disponível e pode ser acessada somente de dentro daquela função.**


```
var bomDia = "Bom dia!";

function funcao() {
    var boaNoite = "Boa Noite!";
}

console.log(boaNoite);
```

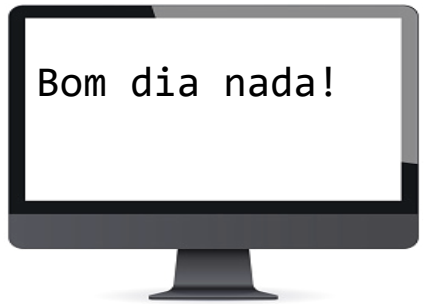
Exemplo_01.js



Variáveis declaradas com var podem ser redeclaradas e atualizadas.

Assim, é possível fazer o seguinte dentro do mesmo escopo e não gerar um erro:

```
var bomDia = "Bom dia!";
var bomDia = "Bom dia nada!";

console.log(bomDia);
```



Exemplo_02.js

hoisting de var

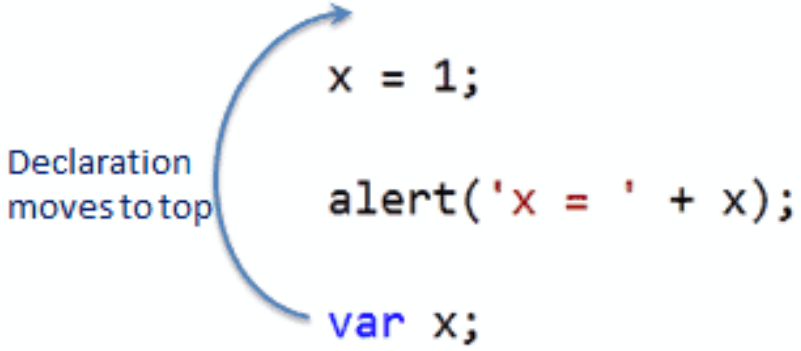
Mecanismo do JavaScript que faz com que as **declarações de variáveis e de funções** sejam **movidas (içadas) para o topo de seu escopo antes da execução do código**.

```
console.log(bomDia);
var bomDia = "Bom dia!";
```

A interpretação será:

```
var bomDia;
console.log(bomDia); // bomDia is undefined

bomDia = "Bom dia!";
```

Desse modo, **variáveis de tipo var** sofrem o **hoisting** (içamento) e vão para o topo do escopo, sendo inicializadas com um valor **undefined**.



Exemplo.js

Problemas com var

1) A variável x é definida globalmente, fora da função com o valor 2.
2) Depois, x é redefinida dentro da função com o valor 3.

Código original

```
var x = 2;
//global

function funcao() {
    //local
    console.log(x);
    var x = 3;
    console.log(x);
}
funcao();
```

O que é impresso?

A interpretação será:

```
var x = 2;
//global
function funcao() {
    //variável “içada"
    var x;
    console.log(x); //undefined
    x = 3;
    console.log(x); //3
}
funcao();
```

Exemplo.js

O código JavaScript é executado em duas fases.

1. É feito o ***parsing***, vasculhando declarações de variáveis, funções e parâmetros.
2. Só então é feita a execução de fato.

A consequência é que as **declarações de variáveis são içadas** para o topo da função ou arquivo em que estão definidas (***variable hoisting***).

let

`let` - mantendo o escopo de bloco

Para trazer o **escopo de bloco ao JavaScript**, o **`ECMAScript 6`** buscou disponibilizar essa mesma flexibilidade (e uniformidade) para a linguagem.

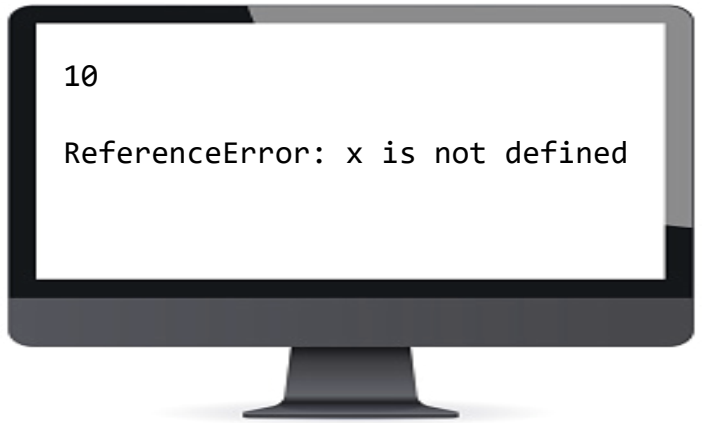
A palavra-chave **`let`** permite declarar **variáveis com escopo de bloco**.

Exemplo:

```
if (true) {
    let x = 10;
    console.log(x); // 10
}
console.log(x);
```

Ao acessar uma variável que foi declarada através da palavra-chave **`let`** fora do seu escopo, ocorre o
`ReferenceError: escopoBloco is not defined`

>> `let` garante o escopo de bloco.



`let` - mantendo o escopo de bloco

Para trazer o **escopo de bloco ao JavaScript**, o **`ECMAScript 6`** buscou disponibilizar essa mesma flexibilidade (e uniformidade) para a linguagem.

A palavra-chave **`let`** permite declarar **variáveis com escopo de bloco**.

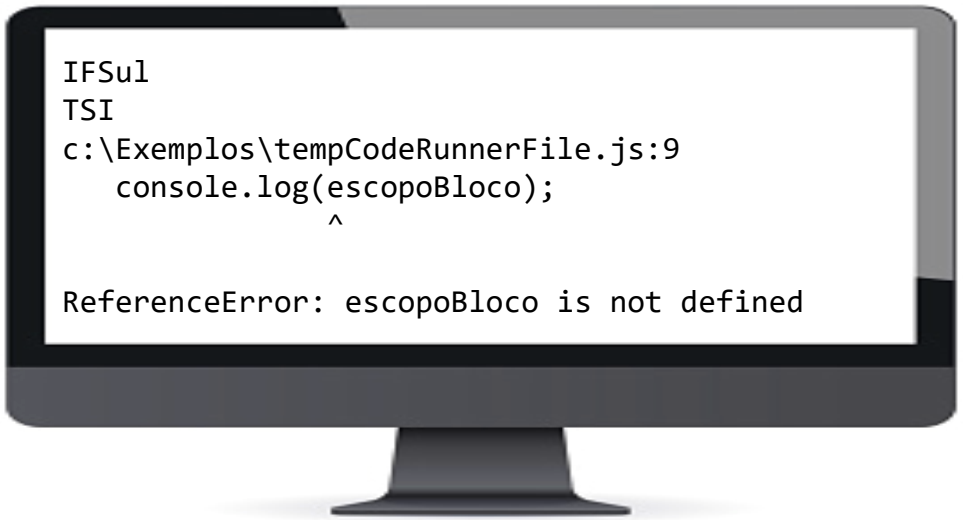
O que imprime este código?

```
var imprime = function() {
    if(true) {
        var escopoFuncao = 'TSI';
        let escopoBloco = 'IFSul';

        console.log(escopoBloco);
    }
    console.log(escopoFuncao);
    console.log(escopoBloco);
}

imprime();
```

Exemplo_03.js

Ao acessar uma variável que foi declarada através da palavra-chave **`let`** fora do seu escopo, ocorre o **`ReferenceError: escopoBloco is not defined`**

`>>` **`let`** **garante o escopo de bloco**.



`let` – evitando *hoisting* problemático

Hoisting é o comportamento do JavaScript que move as declarações de variáveis e funções para o topo do seu contexto de execução antes de executar o código.

Com var, a declaração da variável é "elevada" (hoisted) para o início do contexto de execução, **mas a inicialização não é**.

Isso pode levar a comportamentos inesperados, **pois o código pode tentar usar uma variável antes dela ser inicializada**.

```
console.log(a); // undefined
var a = 5;
console.log(a); // 5
```

É interpretado como

```
var a;
console.log(a); // undefined
a = 5;
console.log(a); // 5
```

Com var, a declaração é elevada, mas a inicialização não é.
O acesso à variável antes da inicialização retorna `undefined`.

`let` – evitando hoisting problemático

Hoisting é o comportamento do JavaScript que move as declarações de variáveis e funções para o topo do seu contexto de execução antes de executar o código.

Com **`let`**, a declaração da variável também é "elevada", mas a variável permanece em uma "zona morta" (*temporal dead zone*) até que a declaração seja processada.

Isso significa que a variável não pode ser acessada antes da sua declaração.

```
console.log(b); // ReferenceError: Cannot access 'b' before initialization
let b = 10;
console.log(b); // 10
```

A declaração também é elevada, mas a variável está em uma "zona morta" até ser inicializada, resultando em um erro se acessada antes da inicialização.

`let` - Proibição de declaração dupla no mesmo escopo

Enquanto `var` permite que você declare a mesma variável múltiplas vezes no mesmo escopo, **`let`** não permite isso, ajudando a evitar erros que podem surgir de **declarações acidentais duplicadas**.

```
var x = 1;
var x = 2; // Não gera erro

let y = 1;
let y = 2; // SyntaxError: Identifier 'y' has already been
declared
```

`let` - Melhoria na legibilidade e manutenção do código

Ao fornecer escopo de bloco e evitar declarações duplicadas, **`let`** contribui para um **código mais limpo**, **menos propenso a erros e mais fácil de entender e manter**.

const

const

Embora o **let** garanta o escopo, ainda assim, existe a possibilidade de declarar uma variável com **let** e ela ser **undefined**.

O que imprime este código?

```
void function(){
    let mensagem;
    console.log(mensagem); // Imprime undefined
}();
```

Os parênteses no final da função, servem para invocar a função imediatamente após sua definição. Essa técnica é comumente chamada de **IIFE (*Immediately Invoked Function Expression*)**.

A principal razão para usar essa abordagem é criar um escopo isolado para as variáveis definidas dentro da função. Isso ajuda a evitar conflitos de nomes de variáveis e mantém o código organizado.

Além disso, a função pode ser usada para executar um bloco de código imediatamente, sem a necessidade de chamá-la explicitamente em outro lugar.

Se tivermos uma variável que precisa ser inicializada com um valor, como podemos fazer isso no **JavaScript** sem causar uma inicialização **default** com **undefined**?

Para garantir esse comportamento em uma variável no **JavaScript**, pode-se declarar constantes por meio da palavra-chave **const**

```
void function(){
    const mensagem = 'IFSul';
    console.log(mensagem);
    mensagem = 'TSI';
}();
```

O que imprime este código?

```
IFSul
\tempCodeRunnerFile.js:4
    mensagem = 'TSI';
             ^

TypeError: Assignment to constant variable.
```

Exemplo_04.js

const

A declaração **const** em JavaScript foi introduzida com o ECMAScript 6 (ES6) e é usada para criar variáveis cujo **valor não pode ser reatribuído**.

Características:

1) Imutabilidade da referência: Variáveis declaradas com **const** são imutáveis no sentido de que não é possível reatribuir um novo valor à variável.

```
const x = 10;
x = 20; // TypeError: Assignment to constant
variable

const obj = { a: 1 };
obj.a = 2; // Isso é permitido
console.log(obj.a); // 2
```

Isso não significa que o valor em si seja imutável.

Se o valor é um objeto (incluindo arrays), suas **propriedades ou elementos ainda podem ser modificados**.

const

2) Escopo de bloco

- Assim como let, const tem escopo de bloco. Isso significa que `const` é visível apenas dentro do bloco em que é definido.

```
if (true) {
    const y = 5;
    console.log(y); // 5
}
console.log(y); // ReferenceError: y is not defined
```

3) Declaração duplicada

- Não é permitido declarar a mesma variável com **`const`** no mesmo escopo, o que ajuda a evitar erros de duplicação acidentais.

```
const z = 1;
const z = 2; // SyntaxError: Identifier 'z' has already been declared
```

const

4) *Hoisting* e zona morta temporal

- Assim como **let**, a declaração de uma variável **const** é elevada para o topo do bloco, mas a variável permanece em uma "zona morta" até que a declaração seja processada.

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization
const a = 30;
console.log(a); // 30
```

5) Uso com objetos e arrays

- Embora não seja permitido reatribuir um objeto ou array declarado com const, é possível pode modificar seu conteúdo.

```
const arr = [1, 2, 3];
arr.push(4); // Isso é permitido
console.log(arr); // [1, 2, 3, 4]

arr = [1, 2, 3, 4]; // TypeError: Assignment to constant variable
```

Resumindo `let` e `const`

`const` e **`let`** são duas palavras-chave introduzidas no ECMAScript 6 (ES6 ou ES2015) para **declaração de variáveis** em JavaScript.

`let` (*block-scoped*)

a) Variáveis declaradas com `let` são mutáveis, o que significa que seu valor pode ser reatribuído.

b) Elas também devem ser inicializadas no momento da declaração, mas essa inicialização não é obrigatória (a variável ficará com valor `undefined` até que seja atribuído um valor).

c) O escopo de uma variável `let` é o bloco em que ela está definida (*block-scoped*), o que a torna mais restrita do que as variáveis declaradas com `var`.

`const` (Constante)

1. Variáveis declaradas com `const` são constantes, o que significa que seu valor não pode ser reatribuído após a inicialização.

2. Elas devem ser inicializadas no momento da declaração, e essa inicialização é obrigatória.

3. O escopo de uma variável `const` é o bloco em que ela está definida (*block-scoped*).

function

function

Define uma **função JavaScript**.

Sintaxe

```
function nomeFuncao([arg1, arg2, …, argn]){
        //script
};
```

Exemplo

```
function multiplica(x, y){
        var resultado = x * y;
        alert(resultado);
};
```

- Admite **zero ou mais argumentos**;
- Define um **bloco de código** a ser executado em determinados pontos do script;
- Útil para **não reescrever códigos repetidamente**;
- Permite que a **função seja "chamada"** sempre que necessário.

return

Permite **retornar um valor** resultante de uma `function`.

Exemplos

```
function multiplica(x, y){

    var resultado = x * y;
    alert(resultado);

};
```

```
function multiplica(x, y){

    return x * y;

};

var a = multiplica(3, 2);
```

Estruturas condicionais

if/else, else if e switch

Destinam-se a criar **estruturas de testes condicionais**.

Sintaxe

```
// ...
if(expressao){
        //executa o bloco
}
// ...
```

if

Testa uma expressão.

Caso seja **true**, executa o bloco que a segue.

Exemplo

```
...
var idade = 3;
if(idade < 18){
        alert("Proibido para menores de 18 anos.");
}
...
```

if/else, else if e switch

Destinam-se a criar **estruturas de testes condicionais.**

Sintaxe

```
// ...
if(expressao){
        //se true, executa este bloco
} else{
        // se false, executa este bloco
}
// ...
```

else

Usada em conjunto com o **if**.

Permite a tomada de decisão **caso o teste resulte em falso**.

Exemplo

```
...
var idade = 21;
if(idade < 18){
        alert("Proibida entrada para menores de 18 anos.");
}else{
        alert("Entrada liberada.");
}
...
```

if/else, else if e switch

Destinam-se a criar estruturas de testes condicionais.

if-else

```
function verifiqueDados() {
   if (document.form1.tresCaracteres.value.length == 3) {
      return true;
   } else {
      alert("Informe exatamente três caracteres. "
               + document.form1.tresCaracteres.value + " não é válido.");
      return false;
   }
}
```

if/else, else if e switch

Sintaxe

```
// ...
if(expressao){
    //se true, executa este bloco
} else if {
    // se false, executa este bloco
} else{
    // senão executa esse bloco
}
// ...
```

else if

Usada em conjunto com o **if/else**.

Permite o **encadeamento de dois ou mais testes.**

Exemplo

```
...
var idade = 40
if(idade < 16){
    alert("Você é um criança/adolescente.");
}else if(idade>=16 && idade<21){
    alert("Você é um jovem adulto.");
} else{
    alert("Você é um adulto.")
}
...
```

if/else, else if e switch

Sintaxe

```
switch(expressao){
    case resultado1:
        script1;
        break;

    case resultado2:
        script2;
        break;

    case resultado3:
        script3;
        break;
    ...
    default:
        script_padrão;
}
```

Exemplo

```
var estado = "SP";
switch(estado){
  case "RS":
    alert("Rio Grande do Sul");
    break;

  case "SC":
    alert("Santa Catarina");
    break;

 case "PR":
    alert("Paraná");
    break;
 default:
    alert("Não é um estado do Sul");
}
```

switch

Cria uma estrutura de testes semelhante a **um sistema de chaveamento** (switch em inglês).

Permite que um programa avalie uma expressão e tente associar o valor da expressão ao **rótulo de um case**.

Se uma correspondência é encontrada, o programa executa a declaração associada.

Operador Ternário


```
> let media = 6;

> let nota = 5;

> (nota >= media) ? "APROVADO" : "REPROVADO“;
‘REPROVADO'
```

Estruturas de repetição

for, while, do-while, for in

Destinam-se a criar **estruturas de repetição**.

Sintaxe

```
for(inicializacao, condicao, incremento){
        //executa o bloco
};
```

for

Cria um ***loop***

Exemplo

```
var msg = "";
for(var i = 0; i < 10; i++){
    msg += i + ",";
};
alert(msg);
```

for, while, do-while, for in

Destinam-se a criar **estruturas de repetição**.

Sintaxe

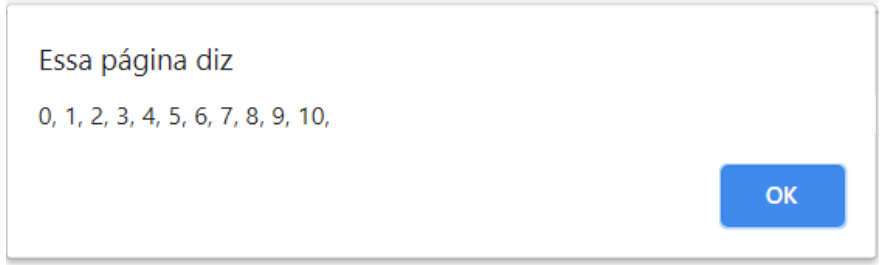
```
while (expressao) {
        //executa o bloco
};
```

while

Cria um *loop* com **teste no início do laço**.

Exemplo

```
var msg = "", j = 0;
while(j<=10){
    msg = msg + j + ", ";
    j++;
};
alert(msg);
```



for, while, do-while, for in

Destinam-se a criar **estruturas de repetição**.

Sintaxe

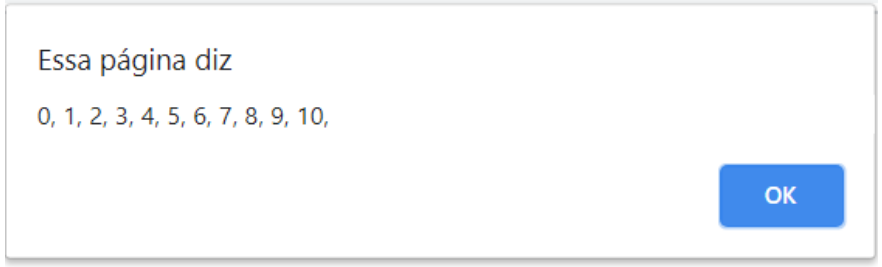
```
do{
    //executa o bloco
} while(expressao);
```

do-while

Cria um loop com **teste no fim do laço**.

Exemplo

```
var msg = "", j = 0;
do{
    msg = msg + j + ", ";
    j++;
}while(j<=10);
```



for, while, do-while, for in

Destinam-se a criar **estruturas de repetição**.

Sintaxe

```
for(variavel in objeto){
    //executa o bloco
};
```

Exemplo

```
var msg = "", j = 0;

var Carro = {
    marca: "Jeep",
    modelo: "Renegade",
    ano: 2016,
    combustivel: "gasolina"
}

var msg = "", i;
for (i in Carro) {
    msg = msg + ": " + Carro[i] + "\n";
}
alert(msg);
```

for in

Cria um *loop* para **percorrer as variáveis ou índices de um objeto**.

Estruturas para Exceções

Exceções

Indicam que que **ocorreu uma situação excepcional ou erro no programa**.

Exemplo: É solicitada a entrada de um dado numérico, e o usuário envia uma **`string`**.

Essa situação **gera uma exceção na aplicação** e, possivelmente, a interrupção do script.

Error

Objeto em **`JavaScript`** que armazena **propriedades e métodos relacionados a erros** ocorridos no script.

Declaração	Finalidade
`message`	Propriedade que **retorna uma mensagem com detalhes do erro** ocorrido.
`name`	Propriedade que **retorna o tipo de erro** ocorrido.
`toString`	Método que **retorna uma `string` composta pelos valores de `message` e `name`**.

Exceções - throw

A declaração **throw** é usada para **lançar uma exceção**.

Ao encontrar um erro, o interpretador **JavaScript** **automaticamente cria uma referência para o erro**.

Usando **throw**, é possível criar referências personalizadas para o erro.

throw expressão;

```
throw "Error2"; // tipo string

throw 42; // tipo numérico

throw true; // tipo booleano

throw {toString: function() {
  return "Eu sou um objeto!";
  }
};
```

```
// Cria um objeto do tipo UserException

function UserException(mensagem) {
  this.mensagem = mensagem;
  this.nome = "UserException";
}

// Cria uma instância de um tipo de objeto e lança ela
  throw new UserException("Valor muito alto");
```

Exceções – try/catch/finally

Fornecem um mecanismo para o **tratamento de exceções**.

A declaração **try...catch** coloca um bloco de declarações em **try.**

E especifica **uma ou mais respostas** para uma exceção lançada.

Sintaxe

```
try{
    //script que pode lançar uma exceção
} catch(exc){
    // bloco para tratamento do erro
} finally{
    // bloco que é executado acontecendo
    // ou não a exceção
}
```

É composta por um bloco **try**, que contém uma ou mais linhas de código que podem gerar uma exceção.

E zero ou mais blocos **catch**, contendo declarações que especificam o que fazer se uma exceção é lançada no bloco **try**

Se qualquer declaração do bloco **try** (ou em uma função chamada dentro do bloco **try**) lança uma exceção, o controle é imediatamente mudado para o bloco **catch**.

Se nenhuma exceção é lançada no bloco **try**, o bloco **catch** é ignorado.

O bloco **finally** executa após os blocos **try** e **catch** executarem, mas antes das declarações seguinte ao bloco **try**...**catch**.

Exemplo 1

```
var leitura = prompt("Digite uma nota entre 0 e 10.");
var x = parseInt(leitura);
try{
    if(x < 0){
        throw("ErroMenor");
    }

    if(x>10){
        throw("ErroMaior");
    }

    if(isNaN(parseInt(x))){
        throw("ErroNaN");
    }

}catch(e){
    if(e == "ErroMenor"){
        alert("O número deve ser maior ou igual a 0.");
    }

    if(e == "ErroMaior"){
        alert("O número deve ser menor ou igual a 10.");
    }

    if(e == "ErroNaN"){
        alert("Digite somente números entre 0 e 10");
    }
}
```

Exemplo 2

```
var leitura = prompt("Digite uma nota entre 0 e 10.");
var x = parseInt(leitura);
try{
    if(x < 0){
        throw new Error("O número deve ser maior ou igual a 0");
    }

    if(x>10){
        throw new Error("O número deve ser menor ou igual a 10.");
    }

    if(isNaN(parseInt(x))){
        throw new Error("Digite somente números entre 0 e 10.");
    }

}catch(e){
    alert(e.message);
}
```

Curso Superior de Tecnologia em Sistemas para Internet
Desenvolvimento front-end I (front 1)
Rafael Cunha Cardoso
rafaelcardoso@ifsul.edu.br

Declarações e Estruturas