



FUNDAMENTOS DA PROGRAMAÇÃO WEB

AULA 5

Profª Margarete Klamas



CONVERSA INICIAL

Vamos iniciar nossos estudos em *JavaScript*. O objetivo desta etapa é fazer uma introdução nessa linguagem de programação. Agora, poderemos proporcionar interação aos usuários, pois com ela é possível escrever HTML, inserir mensagens de boas-vindas e até fazer formatações. Vale lembrar a seguinte recomendação: evite gerar marcação HTML com *JavaScript*. Pode-se concluir que pode-se manipular conteúdo e apresentação de páginas *web*.

JavaScript permite até manipular o comportamento do navegador. É possível criar janelas *pop-up*, exibir mensagens, alterar dimensões etc., além de interagir com formulários, realizar cálculos e fazer validações.

Vamos estabelecer como prática o padrão da *web*: utilizar HTML para a estruturação de conteúdo. Ao formatar, utilize CSS e comportamento com *JavaScript*. No Tema 1, abordaremos um pouco de história da linguagem de programação *JavaScript*, hospedagens, ambiente de desenvolvimento, editores de códigos *on-line* e local e uso do console. No Tema 2, estudaremos as variáveis, tipos de dados e comentários, e no Tema 3, os operadores e interação. No Tema 4, veremos as estruturas de controle condicionais e no Tema 5, os laços de repetição.

TEMA 1 – JAVASCRIPT

Nos anos de 1990, as páginas *web* eram estáticas, somente com texto, e posteriormente foi possível inserir imagens. Em 1995, Brendan Eich, contratado pela Netscape, ficou encarregado de desenvolver uma linguagem de programação para incorporar ao seu navegador Netscape. Ele desenvolveu um protótipo em dez dias chamado inicialmente de *LiveScript* e, depois, *JavaScript*. Trata-se de uma linguagem de programação que permite adicionar interação às páginas *web*; em conjunto com HTML e CSS, pode tornar o conteúdo dinâmico.

JavaScript é uma linguagem interpretada. Quando você está lendo uma página na *web* que contém *JavaScript*, o navegador tem um interpretador integrado que executa o código. O que isso significa? É uma linguagem que roda do lado do cliente e depende de funcionalidades hospedadas no navegador do usuário. Precisamos apenas de um navegador para executar *scripts*.

O nome oficial da *JavaScript* é *ECMAScript*, e a versão atual é *ECMAScript 2022*.



1.1 Definições

- **ECMA**: abreviatura de European Computer Manufactures Association, fundada em 1961 com o objetivo de estabelecer padrões. Em 1997, lançou a primeira norma para *JavaScript* (ECMA-262).
- **ECMAScript**: linguagem de programação orientada a objetos com a finalidade de realizar cálculos e manipular objetos computacionais de um ambiente de hospedagem.
- **Linguagem de script**: linguagem de programação usada para manipular, personalizar e automatizar as funcionalidades de um sistema.

1.2 Hospedagens

Um navegador *web* é um exemplo de ambiente de hospedagem *ECMAScript* que funciona do lado do cliente. Não é objeto desta etapa, mas *JavaScript* atua também do lado do servidor, usando *node.js*, que é um interpretador instalado independentemente dos navegadores, em um ambiente no sistema operacional do computador (pode ser *macOS*, *Windows* ou *Linux*). Seu uso possibilita que você escreva programas em *JavaScript* que, por exemplo, vão transformar seu computador em um servidor.

1.3 Ambiente de desenvolvimento

Para o desenvolvimento, na maioria dos casos, é necessário um editor de código, um compilador ou um interpretador da linguagem utilizada; obviamente, utilizar *notebook* ou *desktop* é mais adequado para os estudos. Neste estudo, vamos apresentar duas possibilidades: ambientes *on-line* e programas que podem ser instalados.

1.3.1 Ambiente de desenvolvimento *on-line*

Ambientes de desenvolvimento *on-line* disponibilizam um editor de código e executam o código diretamente no navegador *web*, tornando possível compartilhar os códigos digitados. Os editores *on-line* têm características similares. São normalmente utilizados como plataformas de teste ou treinamento e para postar soluções de problemas de programação. Confira alguns ambientes de programação:



- CodePen: disponível em: <<https://codepen.io/pen/>>. Acesso em: 14 mar. 2023.
- JsBin: disponível em: <<https://jsbin.com/?html,output>>. Acesso em: 14 mar. 2023.
- JSFiddle: disponível em: <<https://jsfiddle.net/>>. Acesso em: 14 mar. 2023.

Observe que são soluções didáticas e para testes e não ambientes de desenvolvimento completos.

1.3.2 Ambiente de desenvolvimento local

Três elementos são necessários para iniciar nossos estudos: um editor de código, um interpretador (navegador) e um depurador. Dependendo do nível de complexidade do projeto, pode ser necessário recorrer a outras ferramentas. Podemos citar, entre outras:

- **Gerenciadores de pacotes:** permitem utilizar bibliotecas que contêm soluções prontas para ser empregadas em programas ou componentes para o desenvolvimento, como por exemplo: Npm ou Yarn. O Npm é o gerenciador-padrão do node.js, e o Yarn foi desenvolvido pelo *Facebook*.
- **Executores de tarefas e empacotadores de módulos:** possibilitam automatizar o processo de desenvolvimento de *software* e agrupar o código resultante de vários arquivos e bibliotecas, como Grunt ou Webpack.
- **Frameworks de teste:** permitem executar testes automáticos; como exemplos, temos Mocha, Jasmine ou Jest.
- **Scanners de segurança:** possibilitam verificar a segurança da solução. Snyk e RetireJs são alguns exemplos.

1.3.2.1 Editores de códigos

Como editor de códigos *JavaScript*, precisamos de editores de texto simples, como o bloco de notas. O mercado oferece editores de códigos profissionais gratuitos e pagos. Existem aqueles que apresentam benefícios como realce de sintaxe, preenchimento automático e verificação de erros, funcionalidades que melhoram a eficiência e ajudam no entendimento do código e na redução de erros de digitação. Alguns exemplos incluem:



- **Visual Studio Code (VS Code)** – bastante utilizado e gratuito, conta com recursos integrados, como depurador *JavaScript* e ferramentas que otimizam projetos; é personalizável, com a possibilidade de baixar extensões.
- **WebStorm** – adequado para grandes projetos, pesado e complexo para pequenos programas; destina-se a uso comercial e não é gratuito.
- **Sublime Text** – rápido e fácil de usar, pode ser utilizado por um período de avaliação; Para uso por tempo mais longo, é necessário adquirir uma licença paga.
- **NotePad++** – de uso gratuito, é leve e rápido.

Ainda há muitos outros editores, tanto pagos quanto gratuitos. Para este estudo, vamos utilizar o *VS Code*.

1.3.2.2 Interpretador

O interpretador é um programa que executa instrução por instrução. Ao desenvolver um aplicativo que rode do lado do servidor, certamente o interpretador escolhido pode ser o *node.js*. Se o programa rodar no lado do cliente, o próprio navegador será o interpretador. Todos os navegadores contam com interpretadores integrados.

1.3.2.3 Depurador

Depuradores são ferramentas que permitem executar instruções de um programa, passo a passo. Quando usamos o navegador *web* como interpretador de *JavaScript*, também obtemos um depurador. Todos os navegadores possuem ferramentas de desenvolvedor. Durante a visualização de páginas *web*, quando você está navegando habitualmente, não os vê; é necessário ativar essas ferramentas.

As ferramentas normalmente disponibilizadas pelos navegadores são:

- **Inspetor de códigos:** permite, por exemplo, verificar os elementos HTML individuais de um *site* aberto.
- **Console *JavaScript*:** exibe informações sobre os erros e nos permite executar comandos *JavaScript* exclusivos no contexto da página atual.
- **Depurador:** permite executar passo a passo as instruções.



Dependendo do navegador, o comando para exibir as ferramentas de desenvolvimento pode ser diferente. Vale também fazer uma pesquisa no *Google*, sempre que esquecer os comandos, ou mudar de navegador.

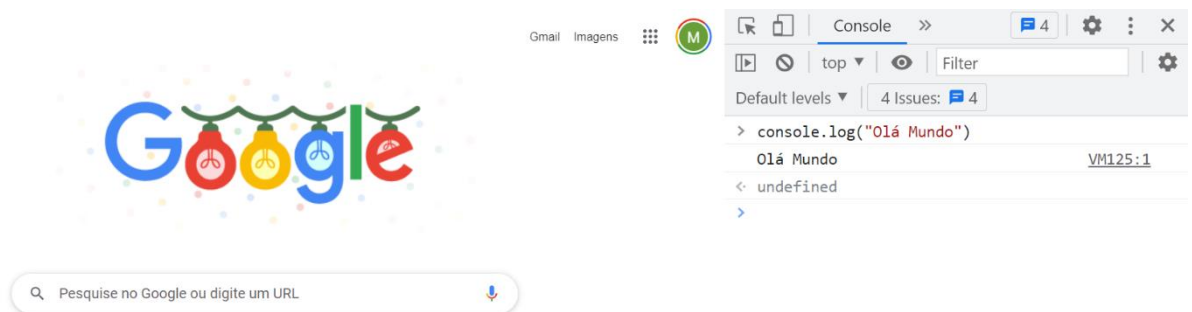
Combinação de tecla(s) para exibir o depurador (console):

- *Internet Explorer e Edge*: tecla F12.
- Demais navegadores: teclas CTRL+SHIFT+J.

1.3.2.4 Exemplo de uso do console

Na Figura 1, recorremos ao navegador *Chrome* e ativamos o console (CTRL+SHIFT+J). Utilizamos a função `console.log()` para digitar Olá Mundo. Ao dar ENTER, ele executou o comando. Como digitamos um texto, ele foi digitado entre aspas.

Figura 1 – Exemplo do uso de console no navegador *Chrome*



TEMA 2 – VARIÁVEIS, TIPOS DE DADOS E OPERADORES

Um dado é nomeado como **variável** quando tem a possibilidade de ter seu valor alterado durante a execução do algoritmo em que é utilizado (por exemplo, o peso de uma pessoa, a idade, a temperatura). Já um dado que não tem razão para sofrer alteração é chamado de **constante** (por exemplo, o valor do π , que é 3,1416).

Existe uma grande liberdade para criar nomes de variáveis/constantes, mas lembre-se de que eles devem se referir ao que armazenaremos na variável (por exemplo, altura, cor, peso, gênero). Em *JavaScript*, não é necessário declarar a variável antes de utilizá-la. Os nomes das variáveis podem consistir em qualquer sequência de letras (maiúsculas e minúsculas), números,



sublinhados e cifrões, mas não devem começar com números. Não podemos utilizar palavras reservadas (Quadro 1).

Quadro 1 – Palavras reservadas *JavaScript*

abstract	Arguments	Await	Boolean
break	Byte	Case	catch
char	Class	Const	continue
debugger	Default	Delete	Do
double	Else	Enum	eval
export	Extends	False	final
finally	Float	For	function
goto	Implements	If	import
in	Instanceof	Int	interface
let	Long	Native	New
null	Package	Private	protected
public	Return	Short	static
super	Switch	synchronized	This
throw	Throws	transient	True
try	Typeof	Var	void
volatile	While	With	yield

Fonte: Flanagan, 2013, p. 24.

O importante também é que o interpretador *JavaScript* diferencia maiúsculas de minúsculas (CASE SENSITIVE) em nomes de variáveis. Então nomes como *teste*, *Teste* ou *TESTE* serão tratados como variáveis distintas.

Para declarar variáveis, podemos utilizar as palavras *var* ou *let*, e para constantes, ***const***. Exemplo:

```
var altura;  
altura=1.7;
```

Observe que declarações, como outras instruções *JavaScript*, deve terminar com um ponto e vírgula. Utilizamos *var* para declarar a variável; o termo provém da sintaxe original *JavaScript*, ao passo que *let* foi introduzida mais tarde. Vamos encontrar *var* em programas mais antigos; recomendamos utilizar *let* para declarar variáveis.

```
let altura;  
altura=1.7;
```



Uma das diferenças entre os termos é que *let* não permite declarar variáveis com o mesmo nome, já *var* permite redeclarar variáveis, o que pode levar a erros.

Antes de fazer a declaração de variáveis, recomenda-se utilizar a expressão *use strict* para forçar que as variáveis utilizadas obrigatoriamente devam ser declaradas antes.

```
"use strict";  
let peso;  
peso=5;  
const ano = 1996;
```

Se utilizarmos *let* e *const* fora de blocos de códigos, que são delimitados por { }, serão variáveis globais. Exemplo:

```
let altura= 180;  
{  
  let peso= 70;  
  console.log(altura); // -> 180  
  console.log(peso); // -> 70  
}  
console.log(altura); // -> 180  
console.log(peso);
```

Observe os resultados: altura foi declarada fora do bloco de códigos { }, e, ao exibi-la no console, ela apresenta o resultado 180. Isso não ocorre com peso, pois é uma variável local e só pode ser acessada dentro do bloco de código. Digite o exemplo acima no console. Abra seu navegador e acesse o console com CTRL+SHIFT+J ou F12 e mãos à obra:



```
Console >> 1 3
Filter
Default levels 3 Issues: 3

> let alturas = 180;
  {
    let peso = 70;
    console.log(alturas);
    console.log(peso);
  }
  console.log(alturas);
  console.log(peso);

180 VM705:4
70 VM705:5
180 VM705:7
✖ ▶ Uncaught ReferenceError: peso is not defined VM705:8
  at <anonymous>:8:13

> |
```

O mesmo não ocorre se utilizarmos a palavra *var*. As variáveis serão globais.

```
> var altura2 = 180;
  {
    var peso2 = 70;
    console.log(altura2);
    console.log(peso2);
  }
  console.log(altura2);
  console.log(peso2);

180 VM867:4
70 VM867:5
180 VM867:7
70 VM867:8
< undefined
>
```

2.1 Tipos de dados

Na terminologia *JavaScript*, a palavra literal designa qualquer dado que se insere no *script*. Para utilizar *scripts* em um arquivo HTML podemos inseri-lo diretamente na página por meio da tag `<script>` e `</script>`, que pode ser inserido no *head* ou *body*. Outra possibilidade é criar o arquivo externamente, salvá-lo como *JavaScript* (*js*) e inseri-lo na página com a tag `<script src="arquivo.js"></script>`.



2.1.1 Dados literais primitivos

Existem **seis tipos de dados literais primitivos**: *boolean*, *number*, *BigInt*, *string*, *symbol*¹ e *undefined*. Vamos explicá-los a seguir.

O tipo de **dado booleano (*boolean*)** é um tipo de dado lógico. Só pode ser **true** ou **false (minúsculas)** e define condições de verdadeiro e falso. Por sua vez, os tipos de **dados numéricos (*number*)** representam os números reais (decimais, por exemplo) e inteiros. Inteiros são números escritos em base decimal (base 10) ou octal (base 8). A base octal está em desuso, segundo a ECMA-262. A base 10 é o nosso sistema de numeração representado pelo conjunto $Z=\{\dots-3,-2,-1,0,1,2,3\dots\}$, o conjunto de números inteiros positivos e negativos.

Já o tipo de dado ***BigInt***, que não é utilizado com frequência, permite escrever números inteiros de qualquer tamanho. Podemos usá-lo em operações matemáticas da mesma maneira que *number*.

O tipo ***string*** representa uma sequência de caracteres. As operações comuns em textos incluem concatenação, extração de *substring* e verificação de comprimento. As *strings* são envolvidas por aspas duplas (" ") ou aspas simples (' '). Exemplos:

```
nomeCliente= "João da Silva";
nomeProduto='Creme dental';
```

Em caso de utilizarmos a operação soma (+) com *strings*, ocorrerá uma concatenação:

```
let teste = "100" + "10";
console.log(teste); // -> 10010
console.log(typeof teste); // -> string
```

Observe nesse caso que, além de exemplificar o concatenar, apresentamos o operador *typeof*, que nos informa o tipo de dado. Nesse exemplo, nos diz que o valor da variável teste é uma *string*. Pode executar no seu console os seguintes testes:

```
let ano = 2022;
console.log(typeof ano); // -> number
console.log(typeof 2023); // -> number

let nome = "Alice";
```

¹ Não é objeto dessa etapa. Trata-se de um tipo complexo.



```
console.log(typeof nome); // -> string
console.log(typeof "João"); // -> string
```

É possível utilizar **interpolação de expressões** para encapsular expressões dentro de *strings*. Com *template string*, é possível utilizar as substituições sintáticas e tornar o código mais legível. Exemplo:

```
let a=6;
let b=10;
console.log(`O resultado da operação é ${a+b} `); // O resultado da operação é 16
```

Observe que foi utilizado o acento crase (``) para tornar possível visualizar o resultado da operação. O símbolo \$ juntamente com { } permite imprimir variáveis, ou resultados de operações diretamente. Outro exemplo:

```
let pais = "Brasil";
let continente = "América do Sul";
let frase = ` ${pais} se localiza na ${continente}.`;
console.log(frase); // Brasil se localiza na América do Sul.
```

Ao utilizarmos *strings*, precisamos conhecer novos conceitos: **métodos** e **autoboxing**. Um método é um tipo especial de função que pertence a um objeto. Objetos são tipos de dados complexos, podem ter muitos valores (de propriedades) e métodos. Para chamar um método de um objeto, escrevemos o nome do objeto, seguido de um ponto (.) e por fim o nome do método. Um exemplo é o objeto `console` que possui vários métodos, sendo um deles o método `log`, quando digitamos `console.log`.

Todos os dados primitivos possuem objetos correspondentes nos quais podem ser convertidos. Cada um desses **objetos terá métodos específicos para seu tipo de dados**. Esse é o **conceito** de **autoboxing**. Se houver um ponto depois de um literal que representa um dado primitivo, ou variável, o interpretador o converte em objeto instantaneamente. Na verdade, parece complexo, mas é muito simples, conforme mostra o exemplo:

```
let rio = "Amazonas";
let caracter = rio.charAt(2);
console.log(caracter); // -> a
```

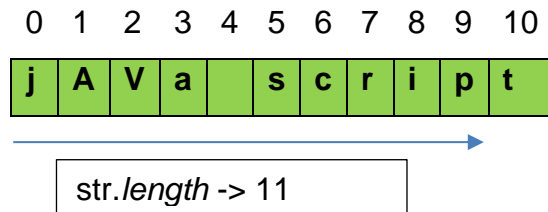
Veja que utilizamos o método `charAt()`, que exibe uma única letra da posição especificada (as letras são contadas a partir da posição zero).

Os métodos normalmente utilizados são:

- **length**: propriedade que informa o número de caracteres de uma *string*.

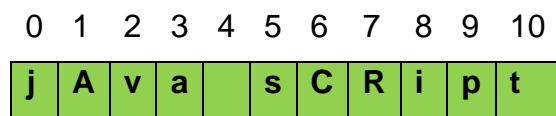


str= "java script"



- **charAt(index)**: informa a posição "index" da *string* (começam com zero).

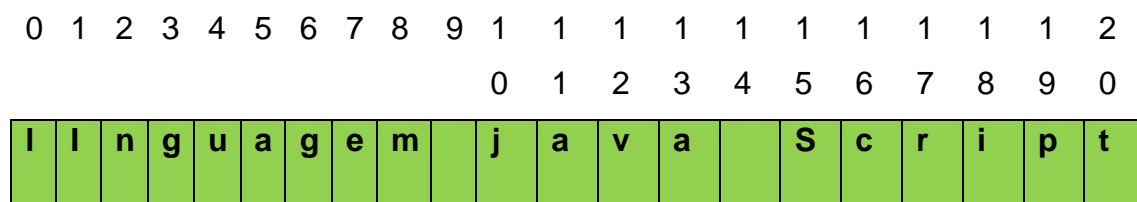
str= "java script"



str.charAt(6) -> "c"

- **slice(beginIndex, [opcional] endIndex)**: método que retorna uma nova *string* que é criada a partir dos caracteres entre beginIndex (incluído) e endIndex (excluído); se endIndex for omitido, a nova *string* será de beginIndex até o final dela.

str1="linguagem java script"

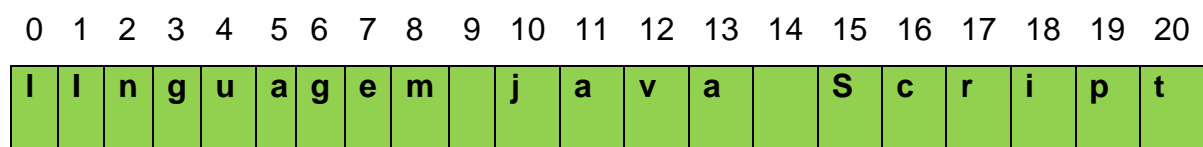


str1.slice(0,9) -> "linguagem"

str1.slice(10,14) -> "java"

- **split(separator, [opcional] limit)**: divide a *string* em *substrings* sempre que um separador é encontrado nessa *string* e retorna um *array* dessas *substrings* (trataremos mais adiante sobre *arrays*), enquanto um limite opcional *limit* limita o número de *substrings* adicionadas à lista.

str1="linguagem java script"





```
str1.split(" ") -> "linguagem", "java", "script"

console.log('192.168.1.1'.split('.')); // ['192', '168', '1', '1']
```

- **undefined**: é o valor-padrão que todas as variáveis possuem após uma declaração, se nenhum valor for atribuído a elas. Exemplo:

```
let declaraVar;
console.log(typeof declaraVar); // undefined
```

- **null**: é um tipo específico. Essa é uma categoria separada, associada a tipos complexos, como objetos. O valor nulo é usado para indicar que a variável não contém nada, e, na maioria dos casos, é uma variável que se destina a conter valores de tipos complexos.

2.2 Comentários em JavaScript

Comentários de uma linha: //

Comentários de múltiplas linhas: /* ...*/

TEMA 3 – OPERADORES E INTERAÇÃO

Assim como outras linguagens de programação, *JavaScript* também implementa vários operadores, conforme mostra o Quadro 2.

Quadro 2 – Operadores *JavaScript*

Grupo	Operador	Descrição
Atribuição	=	Atribuição simples
	+=	Atribuição de adição
	-=	Atribuição de subtração
	*=	Atribuição de multiplicação
	/=	Atribuição de divisão
	%=	Atribuição de resto
	**=	Atribuição de exponenciação
Relacional	==	Igual
	===	Exatamente igual (conteúdo e tipo de dado)
	!=	Diferente
	!==	Exatamente diferente (conteúdo e tipo de dado)
	<	Menor
	<=	Menor ou igual
	>	Maior
	>=	Maior ou igual
Aritméticos	+	Adição



	-	Subtração
	*	Multiplicação
	/	Divisão
	%	Resto da divisão
	**	Exponenciação
	++	Incremento
	--	Decremento
Lógicos	&&	E (AND)
		OU (OR)
	!	NÃO (NOT)

Fonte: Flanagan, 2013, p. 62.

A seguir, temos alguns exemplos de operadores:

```
var x = 4;
x -= 2; // 2 subtraiu 2 de 4
var y = x ** 2; // Elevou 2 (valor de x da linha anterior por 2 -
Potência)

console.log (2 + 2 * 2); // -> 6
console.log(2 + (2 * 2)); // -> 6
console.log((2 + 2) * 2); // -> 8
```

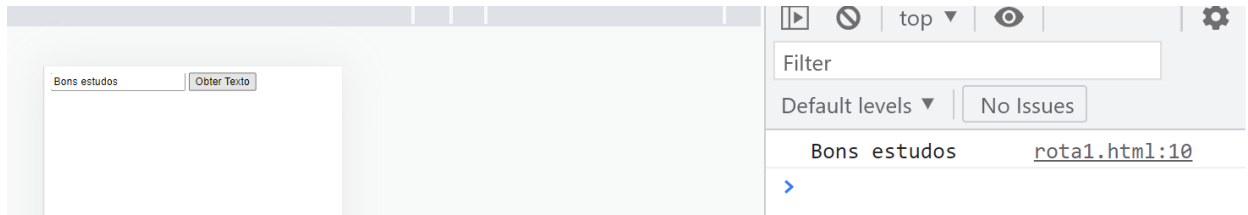
3.1 Interação com o usuário

No *front end*, a interação é fundamental, e podemos receber informações dos usuários e processar esses dados. Quando um usuário utiliza uma calculadora, ele informa números e a operação que deseja realizar. É um exemplo de interação. Utilizamos para interagir com o usuário componentes HTML. Usá-los pode não ser muito difícil, mas requer pelo menos uma compreensão sobre fundamentos do *Document Object Model* (DOM), também chamado de *Árvore do Documento*, que é usado em páginas da *web* e nos fundamentos do próprio HTML. Confira o seguinte exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
  <input id="meuId" type="text"></input>
  <button
onclick="console.log(document.getElementById('meuId').value)">Ob
ter Texto</
button>
</body>
</html>
```



Ao executarmos esse código, temos:



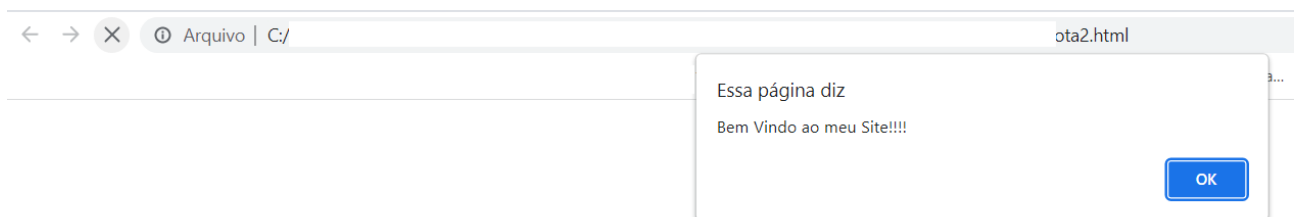
Observamos uma caixa de texto e um botão. Na caixa de texto, o usuário pode digitar qualquer texto. Ao clicar no botão, o texto digitado pode ser recuperado no console (CTRL+SHIFT+J).

Podemos interagir com o usuário por meio de caixas de diálogo que podem ser inseridas em todos os navegadores. São as janelas *pop-up* (ou janelas modais), que, quando exibidas, fazem com que os usuários primeiramente interajam com elas e as fechem para poder ter acesso à página em exibição. Observamos que essas janelas incomodam os usuários, por isso seu uso deve ser moderado. Vamos utilizá-las aqui para fins didáticos. Os tipos de caixas de diálogos incluem:

- **Alert:** usamos o método **alert()**, o qual tem como parâmetro um texto que aparecerá para o usuário. Exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Exemplo de Alert</title>
  <script>
    alert("Bem Vindo ao meu Site!!!!");
  </script>
</head>
<body>
</body>
</html>
```

Ao abrir o arquivo no navegador, teremos:





Podemos utilizar também o método **window.alert()**:

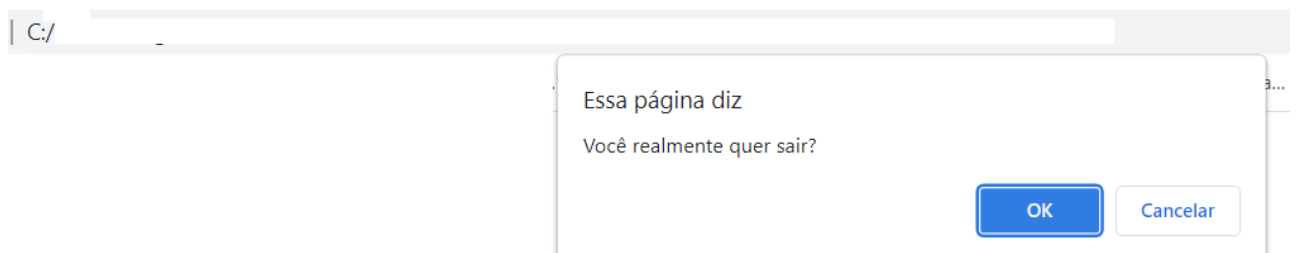
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Exemplo de Alert</title>
  <script>
window.alert("Olá Mundo!");
</script>
</head>
<body>
</body>
</html>
```

Observe, nos códigos acima, que se usarmos o *JavaScript* no navegador, precisamos do HTML, e o código deve estar entre as *tags script*. Nos exemplos que apresentamos antes desse, utilizamos o console para digitar; agora, estamos utilizando um arquivo.html.

- **Confirm:** a diferença entre alerta e confirmação é que a caixa de diálogo de confirmação mostra dois botões: “OK” e “Cancelar”. Exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Exemplo de Confirm</title>
  <script>
window.confirm("Você realmente quer sair?")
</script>
</head>
<body>
</body>
</html>
```

Ao visualizar o código acima no navegador, teremos:

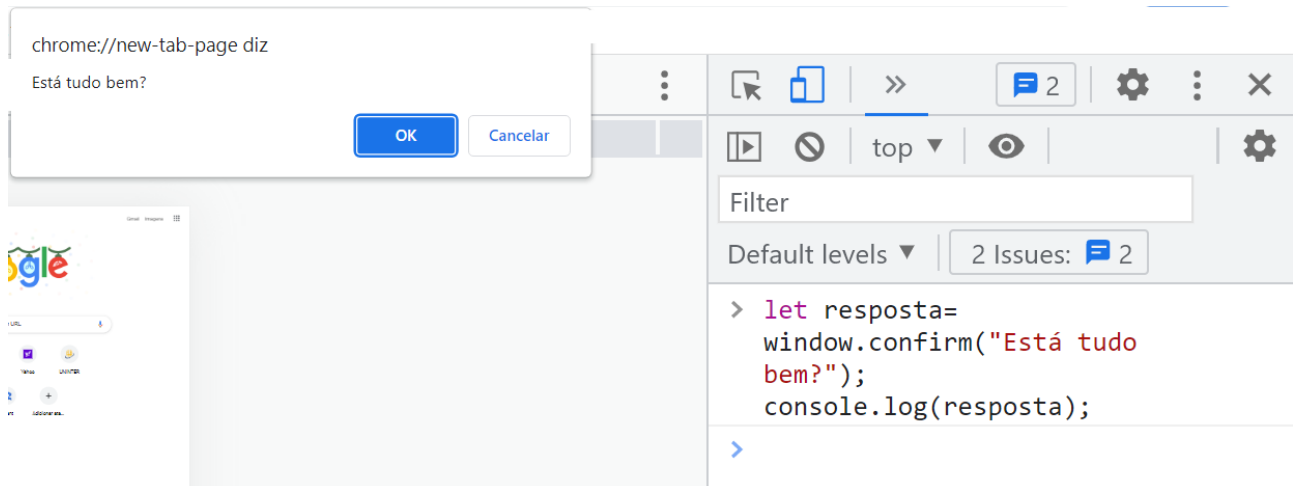




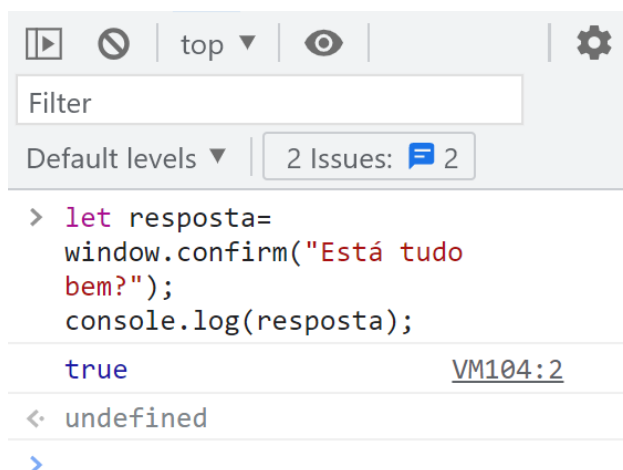
Vamos ver outro exemplo, agora testando no console:

```
let resposta= window.confirm("Está tudo bem?");  
console.log(resposta);
```

Ao abrir o navegador, digite CTRL+SHIFT+J e em seguida o código acima no console, e terá:



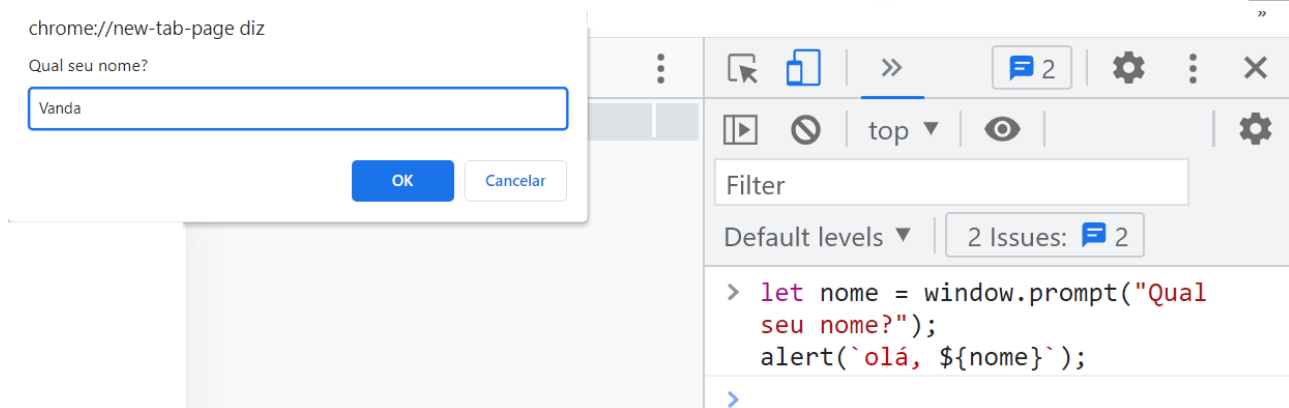
Após clicar no botão “OK”, aparecerá no console -> true.



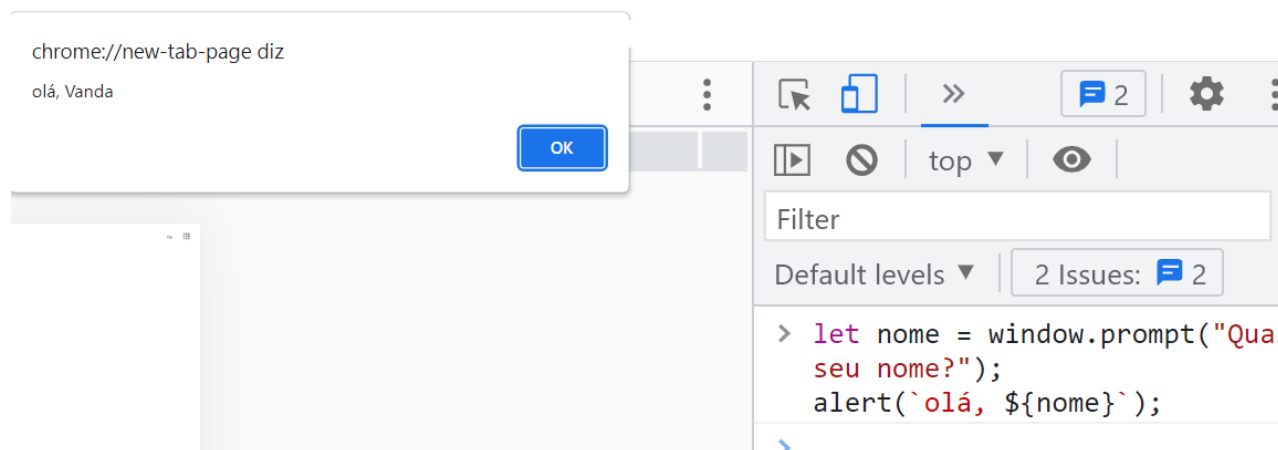
A caixa de diálogo **prompt** contém os botões “OK” e “Cancelar”, mas também um campo de texto de linha única que permite ao usuário inserir texto. Exemplo (execute no console):

```
let nome = window.prompt("Qual seu nome?");  
alert(`olá, ${nome}`);
```

Ao visualizar o código acima no navegador, teremos:



Após o usuário digitar o nome:



TEMA 4 – ESTRUTURAS DE CONTROLE DE FLUXO

Sabemos que os códigos são executados em uma sequência linear de cima para baixo e da esquerda para a direita, ou seja, na ordem em que foram escritos. Se tivéssemos somente a opção linear, eles seriam muito extensos, porém, há outras interessantes para utilizarmos em programação. São as estruturas de controle de fluxo e os *loops*.

4.1 Estruturas de controles de fluxo

Em programação, podemos testar algumas condições com as estruturas de controle, ou as estruturas de seleção. É possível escolher um grupo de ações para serem executadas conforme as condições testadas, representadas por expressões lógicas ou relacionais.



4.1.1 Condicional Simples (estrutura *if*)

A **instrução *if*** permite verificar determinada condição e, dependendo de seu valor *booleano*, executa ou pula um bloco de código. A sintaxe fica assim:

```
if(condição) {  
    //bloco de código a ser executado se a condição for verdadeira  
}
```

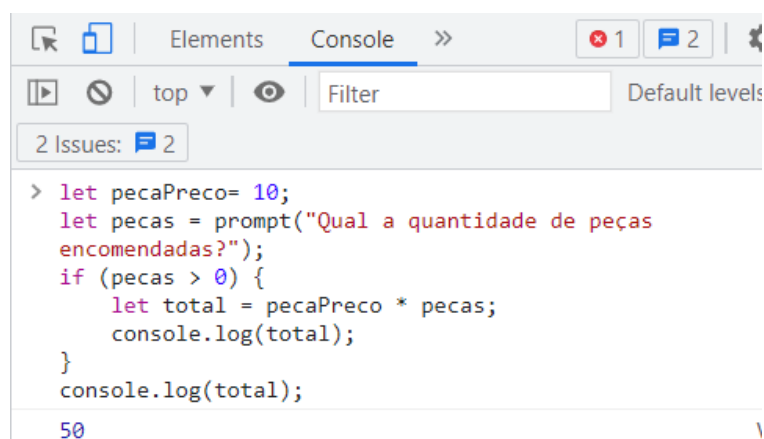
Exemplo:

```
let idade=3;  
if(idade<=11){  
    alert("Você ainda é uma criança");  
}
```

Outro exemplo (executar diretamente no console):

```
let pecaPreco= 10;  
let pecas = prompt("Qual a quantidade de peças encomendadas?");  
if (pecas > 0) {  
    let total = pecaPreco * pecas;  
    console.log(total);  
}  
console.log(total);
```

Ficará assim:



Será exibido o resultado da variável total, que multiplicou o preço da peça pela quantidade de peças encomendadas. Nesse exemplo, digitamos 5 peças.



4.1.2 Condicional composta (estrutura *if – else*)

A palavra-chave **else** é uma parte opcional da instrução **if** e permite adicionar um segundo bloco de código que será executado somente quando o resultado do teste condicional for falso. Ambos os blocos de código são separados pela palavra-chave **else**. Portanto, a sintaxe **if ... else** é a seguinte:

```
if (condição) {  
    condição - código verdadeiro  
} else {  
    condição - código falso  
}
```

Exemplo (testar no console):

```
let seEstaPronto= confirm("Está pronto?");  
if (seEstaPronto) {  
    console.log("Usuário pronto!");  
} else {  
    console.log("Usuário não está pronto!");  
}
```

4.1.3 Condicionais aninhadas

Juntamente com a instrução **else**, podemos utilizar **if** ou **if...else** dentro dele, e qualquer número de instruções **if...else** pode ser aninhado, se necessário. A sintaxe é:

```
if (condição_1) {  
    código  
} else if (condição_2) {  
    código  
} else if (condição_3) {  
    código  
} else {  
    código  
}
```

Exemplo:

```
let idade = prompt("Qual sua idade? ");  
if(idade<=11){  
    alert("Você é criança");  
}  
else if(idade >11 && idade <=14){  
    alert("Você é pré-adolescente");  
}  
else if(idade >14 && idade<=21){  
    alert("Você é adolescente");  
}  
else if(idade >21 && idade<=60){
```



```
    alert("Você é adulto");
}
else{
    alert("Você é idoso!");
}
```

4.1.4 Condicional de múltipla escolha (*switch-case*)

A declaração **switch** permite criar uma estrutura de testes condicionais, avaliar uma expressão condicional e tentar corresponder seu valor a um dos casos fornecidos. A sintaxe da instrução *switch* é:

```
switch (expressão) {
    case resultado 1:
        código 1
        break;
    case resultado 2:
        código 2
        break;
    default:
        código
}
```

Exemplo (digitar no console):

```
let estado = prompt("Qual seu estado? ");
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");
}
```

Observe que a declaração **case** seguida de uma expressão é a chave de testes da estrutura; se for verdadeira, executa o bloco de código que se segue; caso seja falsa, pula para o bloco seguinte, e assim sucessivamente. Por sua vez, a declaração **break** serve para interromper os testes se a condição for verdadeira, e o bloco executado. Já a declaração **default** contém o bloco a ser executado se todos os blocos forem falsos.



TEMA 5 – LAÇOS DE REPETIÇÃO (*LOOPS*)

Os *loops* são a segunda forma de instruções de fluxo de controle. Eles permitem repetir qualquer trecho de código, determinando a quantidade de vezes da repetição ou estipulando uma condição para parar a repetição.

5.1 *for*

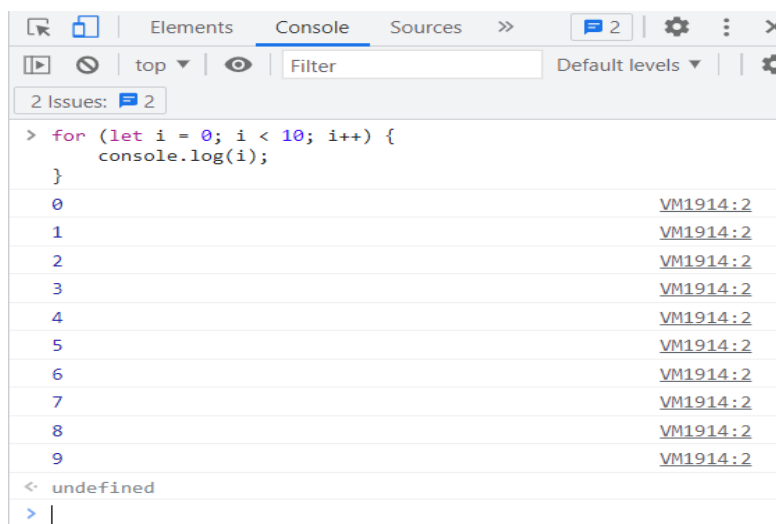
Fazemos uso de ***for*** em situações em que sabemos quantas vezes executar o *loop*. Por exemplo, se desejarmos imprimir na tela uma sequência de números entre 0 e 9, podemos utilizar a estrutura de repetição *for* para facilitar nosso trabalho. A sintaxe é:

```
for (inicialização; condição; incremento) {  
    bloco de código (faça isso)  
}
```

Exemplo (digitar no console):

```
for (let i = 0; i < 10; i++) {  
    console.log(i);  
}
```

No código apresentado acima, inicializamos a variável *i* com zero, informamos a condição (*i*<10) e incrementamos a cada *loop* “1” (*i*++). A visualização no navegador ficará:



5.2 *while*

O loop *while* é um dos que normalmente usamos quando não sabemos exatamente quantas vezes repetir algo, mas sabemos quando parar. A sintaxe é:



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

Por exemplo, se desejarmos imprimir os números menores do que 91, dez em dez:

```
let num = 0;  
while(num < 90) {  
    console.log(num);  
    num += 10;  
}
```

Nesse código, iniciamos a variável `num` com valor zero. Colocamos a condição enquanto (`num<90`). Exibimos no console o valor de cada iteração. `num +=10` significa que a cada *loop* (iteração) a variável `num` receberá +10. Ao visualizarmos no console:

```
> let num = 0;  
  while(num < 90) {  
    console.log(num);  
    num += 10;  
  }  
0  
10  
20  
30  
40  
50  
60  
70  
80  
90  
> |
```

Um exemplo para você testar no console:

```
let continua = false; //atribuímos a variável continua o valor false  
let contador = 1; //inicializamos a variável contador com 1  
  
while (!continua) { //enquanto a variável continua for true (!  
  Diferente do valor declarado false  
    continua = !confirm(`[${contador++}] Mais um loop?`);  
  } //executa o bloco de código acima e soma +1 ao valor da  
  variável contador
```



5.3 do/while

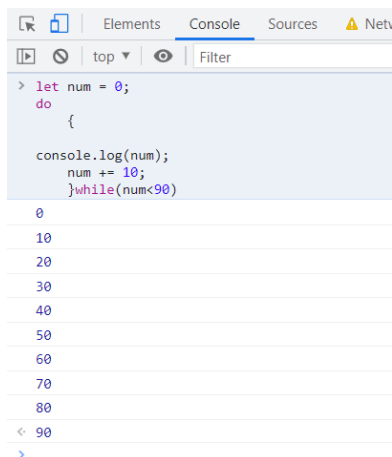
O *loop do/while* é muito semelhante ao *loop while*, a principal diferença é que em um *loop while* a condição é verificada antes de cada iteração, e no *do/while*, ela ocorre após cada iteração. A sintaxe é:

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

Reescrevemos o código:

```
let num = 0;  
do  
{  
    console.log(num);  
    num += 10;  
}  
while(num<90)
```

A execução no console será:



FINALIZANDO

Chegamos ao final desta etapa, na qual apresentamos uma introdução em *JavaScript*, que é uma linguagem bastante utilizada em páginas web. *JavaScript* é uma linguagem poderosa e nos proporciona uma liberdade muito grande para interagirmos com os usuários. Também selecionamos os principais pontos fundamentais para entender outros conceitos da linguagem.

Execute no console os exemplos aqui mostrados; assim você vai se familiarizando com a sintaxe da linguagem. Bons estudos!



REFERÊNCIAS

FLANAGAN, D. **JavaScript** – o guia definitivo. 6. ed. Porto Alegre: Bookman, 2013.

MORRISON, M. **Use a cabeça! JavaScript**. Rio de Janeiro: Alta Books, 2020.

MOZILLA DEVELOPER NETWORK. [S.d.]. Disponível em: <<https://developer.mozilla.org/en-US/docs/Web/JavaScript>>. Acesso em: 7 mar. 2023.

SILVA, M. S. **JavaScript** – guia do programador. São Paulo: Novatec, 2010.

YANK, K.; ADAMS, C. **Só JavaScript**: tudo o que você precisa saber sobre *JavaScript* a partir do zero. Porto Alegre: Bookman, 2009.