



A linguagem de programação JavaScript

Prof. Luiz Henrique de Angeli
 luizdeangeli@gmail.com
 luiz.angeli@unicesumar.edu.br

Quem sou?



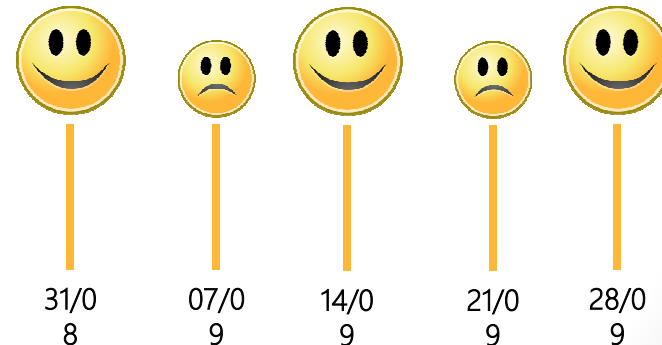
- **Luiz Henrique de Angeli**
- Graduado em Processamento de Dados
Unicesumar – Centro Universitário Cesumar (2006)
- Pós-Graduado em Desenvolvimento de Sistemas Orientado a Objetos Java
Unicesumar – Centro Universitário Cesumar (2010)
- Professor na Unicesumar desde 2007
Ministrando disciplinas na área de desenvolvimento: PHP, HTML, CSS, JS, DELPHI, SHELL SCRIPT, C, PASCAL
- Head de Desenvolvimento Ensino EAD
- Desenvolvimento para WEB desde 2005

Conteúdo da Disciplina



- | | |
|---------------------------|------------------------|
| • O que é | • Funções |
| • História | • DOM |
| • Como programar | • Objetos |
| • Funcionalidades | • Tratamento de Erros |
| • Caixa de Diálogo | • Expressões Regulares |
| • Erros e Depuração | • ECMAScript 6+ |
| • Tipos de dados | • Testes unitários |
| • Operadores | • TypeScript |
| • Variáveis | • JQuery |
| • Estrutura condicional | • Tópicos especiais |
| • Estrutura de repetições | |
| • Arrays | |

Nossas aulas....

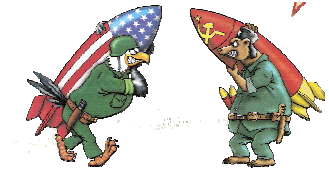


Um pouco da história da internet

é rapidinho.....

- Durante a guerra fria, a **Darpa** criou uma rede de dados chamada **Arpanet** para comunicar 4 computadores e criaram os protocolos de comunicação:

- NCP : Protocolo de Controle de Rede
- FTP: Transferência de arquivos
- DNS: Identificação de maquinas



- Objetivo da rede era proteger os dados em casos de ataques;
- Dinheiro do Governo dos EUA + Conhecimento dos Universitários

1969

- **Ray Tomlinson** criou um sistema para troca de mensagens online o **email**;



1972

- **Robert Kahn**: Modificiou o NCP e criou o TCP (Transfer Control Protocol) e criou o termo Internetting;

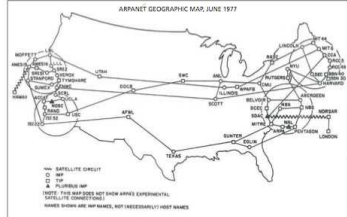


- **Vint Cerf** e criou o (Internetwork Protocol) que juntos criaram o TCP/IP



1973

- Governo dos EUA e os militares não quiseram mais tomar conta do projeto e foi criada a **Internet**



1977

- **Timothy John Berners-Lee** criou:
 - O protocolo **http** (hipertext transfer protocolo);
 - **A linguagem html**: HyperText Markup Language;
 - **WWW**: World Wide Web
 - Criou –se então a primeira versão do HTML



1990

- A NCSA criou Navegador Mosaic que ficou muito famoso na época;



1992

- Desenvolvedores do NCSA saíram do órgão e criaram a empresa Netscape e também criaram o navegador com o mesmo nome;



1994

- Foi criado os navegadores:

- IBM WebExplorer
- UDI WWW
- Internet Explorer 1.0

- Briga entre Microsoft e Netscape.

- Microsoft teve que retirar o IE do Windows



1995

- Netscape foi vendida para America Online
- Fundadores do Netscape criaram a Fundação Mozilla e o navegador Mozilla Firefox



1999

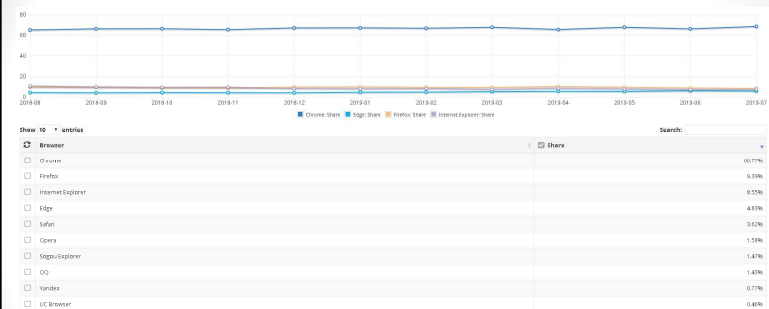
- A Google resolveu entrar na briga e criou o seu próprio navegador;
- Em 2 anos de criação passou a ser o 3º navegador mais utilizado

Google



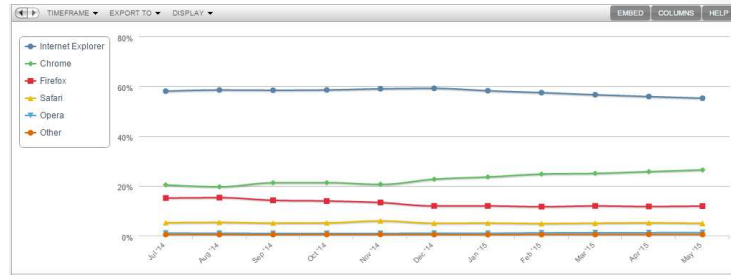
2008

- Como esta o uso dos navegadores em geral?



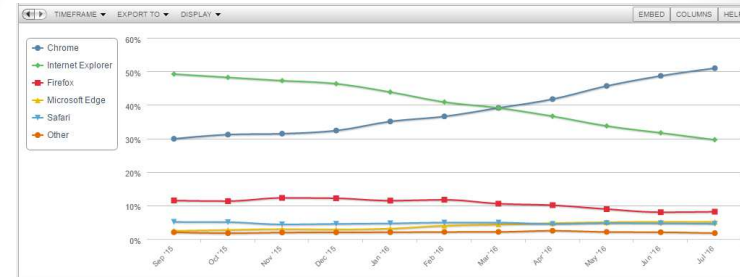
2019

- www.netmarketshare.com



2015

- www.netmarketshare.com



2016

W3C



- No mês de outubro de **1994**, foi criado o World Wide Web Consortium (W3C);
- O Consórcio World Wide Web (W3C) é um consórcio internacional no qual organizações filiadas, uma equipe em tempo integral e o público trabalham juntos para desenvolver padrões para a Web.
- Liderado pelo inventor da web **Tim Berners-Lee** e o CEO **Jeffrey Jaffe**, o W3C tem como missão **Conduzir a World Wide Web para que atinja todo seu potencial, desenvolvendo protocolos e diretrizes que garantam seu crescimento de longo prazo.**
- <http://www.w3.org>
- <http://www.w3c.br>

O WHATWG (The Web Hypertext Application Technology Working Group) é um grupo de desenvolvedores que não se conformavam com a velocidade que as coisas aconteciam no W3C. Portanto eles são um grupo separado do W3C.

Esse grupo foi fundado por membros da Mozilla, Apple e Opera por volta de 2004. Eles resolveram iniciar a escrita de uma nova recomendação para o HTML, já que o W3C iria abandonar o HTML e focar seus esforços em linguagens baseadas em XML.

É por isso que algum dia no seu passado você começou a fechar as tags BR com uma barra no final (
).

19 19 19 19 20 20 20

As Três Camadas da Web



- HTML: [Conteúdo]
 - Camada de base que qualquer visitante deve ser capaz de ver
 - Tudo que é necessário para entender o conteúdo de uma página web.
- CSS: [Apresentação]
 - Camada para apresentar uma melhor aparência para o site
- JAVASCRIPT: [Comportamento]
 - Camada de interatividade e comportamento dinâmico



As Três Camadas da Web



Riqueza na
experiência
com o usuário

E o JavaScript???

O que é JavaScript



- É uma **linguagem de programação** interpretada;
- Esta presente nos principais navegadores;
- É atualmente a principal linguagem de programação *cliente/servidor* em navegadores web;
- É leve e expressivo;
- Curva de aprendizagem pequena;
- É uma linguagem com muitas contradições, contém muitos erros e pontas soltas;

Timeline do JavaScript

- A empresa **Nombas** criou um projeto cmm (c--), que seria uma linguagem mais simples, por questões de aprendizagem;
- Foi comprada em pouco tempo pela **Openwave**;
- Os desenvolvedores resolveram rebatizar a linguagem para **ScriptEase**

1992



- Em seguida o ScriptEase despertou um interesse em uma grande empresa da época, a **Netscape**;
- Assim a Netscape pegou o projeto do ScriptEase para utilizar em seu produto, o navegador Netscape;

1994

- O projeto foi parar na mão de **Brendan Eich** que rebatizou o projeto para **Mocha**;
- Posteriormente antes do lançamento a linguagem foi rebatizada para **LiveScript**
- O LiveScript foi lançado no Netscape 2.0



1995




- No mesmo ano do lançamento a **Netscape** estava em negociação com a **SUN** para desenvolver novas tecnologias para navegadores;
- A **SUN** estava acabando de fazer o grande o lançamento, a linguagem **JAVA**;
- E fizeram uma grande “jogada de marketing” mudando o nome da linguagem para **JavaScript**;

1995

JavaScript	Java
Orientada a objeto. Sem distinção entre tipos e objetos. A herança é feita através do protótipo e as propriedades e métodos podem ser adicionadas a qualquer objeto dinamicamente.	Baseada em classes. Objetos são divididos em classes e instâncias com toda a herança através da hierarquia da classe. Classes e instâncias não podem ter propriedades ou métodos adicionados dinamicamente.
Os tipos de dados das variáveis não precisam ser declarados (tipagem dinâmica)	Os tipos de dados das variáveis devem ser declarados (tipagem estática).
Não pode escrever automaticamente no disco rígido.	Pode escrever automaticamente no disco rígido.
Linguagem não compilada	Linguagem compilada

1995

- Microsoft resolveu criar uma linguagem para o seu navegador **Jscript**;

Microsoft 

1996

- A **Netscape** submeteu o JavaScript a **ECMA** internacional como candidato a padrão industria e o trabalho subsequente resultou na versão padronizada chamada **ECMAScript**;
- ECMA é uma associação dedicada a padronização de sistemas de informação;
- Surgiu o ECMA-262 – ECMAScript: uma versão padronizada do JavaScript. Foi uma jogada para ter o reconhecimento do padrão do mercado.

1997

EcmaScript (versões)

Edição	Publicação	Principais Aspectos
1	Junho, 1997	Primeira versão.
2	Junho, 1998	Pequenas modificações para manter a especificação alinhada com a ISO/IEC 16262.
3	Dezembro, 1999	Foram adicionadas expressões regulares, melhorias no tratamento de strings, definição de erros, tratamento de exceções com try/catch, formatação para output de valores números e outras melhorias.
4	Abandonada	A quarta versão propunha modificações grandes, como a inclusão de classes, modularização, generators e afins. Infelizmente, acabou sendo abandonada devido ao avanço lento e complicações políticas entre os membros da Ecma's Technical Committee 39 (TC39).

EcmaScript (versões)

Edição	Publicação	Principais Aspectos
5	Dezembro, 2009	Tornou-se mais claro muitas ambiguidades presentes na terceira edição da especificação. Além disso, adicionou algumas novas features, tais como: getters e setters, suporte a JSON e um mecanismo mais robusto de reflexão nas propriedades dos objetos.
5.1	Junho, 2011	Adaptações para que a especificação ECMA ficasse totalmente alinhada com o padrão internacional ISO/IEC 16262:2011.
6	Junho, 2015	Versão ES6
	2016	Versão ES2016
	2017	Versão ES2017
	2018	Versão ES2018
	2019	Versão ES2019

Visão Geral

- Não há como fazer funcionar um formulário HTML com o uso de elementos HTML.
 - A HTML limita-se a criar os rótulos e campos de um formulário para serem preenchidos pelo usuário e nada mais.

Visão Geral

- **O lado cliente do JavaScript** fornece objetos para controlar um navegador web e seu *Document Object Model* (DOM).
- Por exemplo, as extensões do lado do cliente permitem que uma aplicação coloque elementos em um formulário HTML e responda a eventos do usuário, como cliques do mouse, entrada de formulário e de navegação da página.

Visão Geral

- **O lado do servidor do JavaScript** fornece objetos relevantes à execução do JavaScript em um servidor.
- Por exemplo, as extensões do lado do servidor permitem que uma aplicação comunique-se com um banco de dados, garantindo a continuidade de informações de uma chamada para a outra da aplicação, ou executar manipulações de arquivos em um servidor.

Funcionalidades gerais da JavaScript

- Manipular conteúdo e apresentação
- Manipular o navegador
- Interagir com formulários
- Interagir com outras linguagens dinâmicas

Orientação a objetos

- A linguagem JavaScript suporta programação orientada a objetos – Object-Oriented Programming (OOP)
- É mais apropriado dizer que JavaScript é uma linguagem capaz de simular muitos dos fundamentos de OOP, embora não plenamente alinhada com todos os conceitos de orientação a objetos

Popularização do Javascript

- O Javascript começou a ser visto com outros olhos após o crescimento da utilização do Ajax e do Single Page.
- **Ajax**: Forma para atualizar um conteúdo de uma página sem atualizar a página toda.
- **Single Page**: Aplicações de página única, utilizando o conceito do Ajax

Resumo

- **Javascript**:
 - não é Java;
 - não foi criado pelo W3C;
 - é uma linguagem de programação client-side;
 - é utilizada para controlar o HTML e o CSS para manipular comportamentos na página;
 - não é mantido pelo W3C, ele é uma linguagem mantida pela ECMA;

chega de bla bla bla Vamos começar a programar

abra o seu editor de preferência

Executando um programa JavaScript

- O motivo pelo qual o JavaScript é tão popular é ser relativamente fácil de ser acrescentado a uma página WEB.
- Tudo o que você precisa fazer é incluir pelo menos um elemento HTML script na página para especificar "text/javascript" para o atributo `type` e adicionar qualquer JavaScript que quiser.
- JavaScript funciona imediatamente, incluindo um bloco de scripting e já está em ação.

A tag script

- Geralmente o elemento `script` é adicionado ao elemento `head` de uma página por que é mais fácil de realizar a manutenção.
- Alguns autores recomendam a colocação dos elementos `script` na parte de baixo de um documento, para permitir que o resto da página web seja carregada primeiro antes do `script`.
- Independente da abordagem que você usar: coloque seus scripts sempre no elemento `head` ou na parte de baixo do elemento `body`.

Executando um programa JavaScript

- Você pode colocar a quantidade de código que quiser dentro desta tag. O navegador executará assim que tiver feito o download;

```
<!DOCTYPE html>
<html>
<head>
    <title> Exemplo de JavaScript </title>
    <script type="text/javascript">
        </script>
</head>
<body>
</body>
</html>
```

Comentários

- A linguagem JavaScript aceita dois tipos de comentários:
 - Comentário em **linha** ou em **bloco**;

```
<script type="text/javascript">
    //comentário em uma linha

    /*
        comentário
        em
        linhas
    */
</script>
```

Atenção: todos os comentários JavaScript são visíveis no código fonte da página!

Executando um programa JavaScript

- Nem todos os scripts existentes dentro de páginas web são JavaScript.
- A tag de abertura do elemento `script` contém um atributo definindo o tipo do script.
- É uma forma de identificar o conteúdo codificado.
- Entre outros valores permitidos para o atributo `type` estão:
 - `text/ecmascript`;
 - `text/jscript`;
 - `text/vbscript`
 - `text/vbs`

Executando um programa JavaScript

- text/ecmascript**: Especifica que o script será interpretado como ECMAScript baseado no padrão de scripting ECMA-262;
- text/jscript**: Uma variação de ECMAScript que a Microsoft interpreta no Internet Explorer.
- text/vbscript** e **text/vbs**: São interpretados como VBScript da microsoft, uma linguagem scripting completamente diferente.

Executando um programa JavaScript

- Todos esses valores de `type` descrevem o tipo MIME do conteúdo.
- MIME (Multipurpose Internet Mail Extension) é uma forma de identificar como o conteúdo está codificado (`text`) e seu formato específico (`javascript`).
- Versões anteriores da tag `script` recebiam um atributo `language` que era usado para especificar a versão da linguagem (`javascript1.2`). Entretanto este atributo ficou obsoleto em HTML 4.01.

JavaScript x ECMAScript X JScript

- Embora o nome JavaScript tenha se tornado onipresente para scripting baseados em navegadores do lado cliente, apenas **Mozilla** seu popular navegador, o **Firefox**, implementam JavaScript.
- A ECMAScript é, na verdade, uma especificação de scripting do lado cliente que abrange todo o mercado.
- Contudo a maioria dos navegadores reverencia o tipo `text/javascript`.

Criando arquivos externos

- O uso do JavaScript está se tornando mais orientado a objetos e completo.
- Para simplificar desenvolvedores estão criando objetos JavaScript reutilizáveis que podem ser incorporados em muitas aplicações.
- A única forma de compartilhar esses objetos é criá-los em arquivos separados e fornecer um link para cada arquivo na página WEB.

Criando arquivos externos

- Se o código precisa ser alterado posteriormente, é realizado somente em um lugar.
- Atualmente, mesmo o JavaScript mais simples é criado em arquivos scripts separados.
- Para carregar um arquivo js externo na página vamos utilizar o atributo `src` da tag `script`.

Criando arquivos externos

- Para fazer referência a um arquivo JavaScript externo, você precisa usar o atributo **src** na tag `<script>`;

```
<!DOCTYPE html>
<html>
<head>
    <title> Exemplo de JavaScript </title>

    <script type="text/javascript" src="exemplo.js"></script>

</head>
<body>

</body>
</html>
```

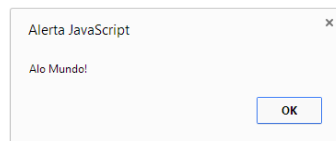
Caixas de diálogo

Caixa de diálogo

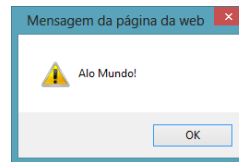
- Caixa de diálogo é uma janela do tipo pop-up que se destina a fornecer informações ou coletar dados do usuário.
- A linguagem JavaScript possui três métodos (ou funções), para o objeto Window, destinadas a criar três tipos de caixa de diálogo.
 - Caixa de alerta
 - Caixa de diálogo de confirmação
 - Caixa de diálogo para entrada de string

Caixa de diálogo

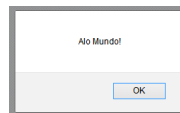
- As caixas de diálogo criadas com JavaScript tem sua apresentação determinada pelo fabricante do navegador:



Chrome



Internet Explorer



Firefox

Caixa de diálogo - Alerta

- `alert()` Box, ou caixa de alerta, destina-se a colocar na interface do usuário uma caixa de diálogo contendo uma mensagem a ele.

```
<script type="text/javascript">  
    alert ("Alo Mundo!");  
    alert ("Alo\nMundo!");  
</script>
```

Caixa de diálogo - Confirmação

- O método `confirm()` do objeto Window destina-se a colocar na interface do usuário uma caixa de diálogo contendo dois botões, normalmente denominados OK e Cancelar, e, ao clicar um deles, ele confirma ou cancela uma determinada ação.
- O retorno do método `confirm()` será:
 - **true**: para escolha ok;
 - **false**: para o cancelar;

```
<script type="text/javascript">
  confirm("Você tem certeza que quer apagar o arquivo? ");
</script>
```

Caixa de diálogo - Confirmação

- O método `prompt()` do objeto Window destina-se a colocar na interface do usuário uma caixa de diálogo contendo um campo para que ele digite uma string. Essa função admite dois argumentos; `prompt("arg1", ["arg2"]);`
 - **arg1**: é uma instrução ao usuário para o que se espera que ele digite no campo de texto
 - **arg2**: é um valor *default*, facultativo, inserido no campo de texto, normalmente para fornecer uma dica da forma de preenchimento do campo.

```
<script type="text/javascript">
  prompt("Informe a data do seu nascimento:", "dd/mm/aaaa");
</script>
```

Erros e Depuração

Espaços em branco

- Quebras de linhas e espaços em branco, quando inseridos entre nomes de variáveis, nomes de funções, números e entidades similares da linguagem, **são ignorados** na sintaxe JavaScript.
- Contudo, para strings e expressões regulares, tais espaçamentos são considerados.

Sintaxe Válida	Sintaxe Inválida
<code>a=27; e a = 27;</code>	<code>a = 2 7;</code>
<code>document.write("<p> Eu sou \ uma string</p>")</code>	<code>document.write \ ("<p> Eu sou uma string</p>")</code>
<code>document.write ("<p> Eu sou uma string</p>")</code>	<code>document.write("<p> Eu sou uma string</p>")</code>
<code>alert("Olá") e alert ("Olá")</code>	
<code>function(){...} e function () {...}</code>	

Erros e Depuração

- Como alguns dos problemas são quase invisíveis à olho nú, podemos utilizar uma ferramenta que está disponível no Mozilla Firefox, Google Chrome e no Internet Explorer;
- O console de erros não faz parte da linguagem e sim do ambiente.
- **Console de erros;**
 - **Mozilla Firefox:** Ctrl+Shift+J
 - **Google Chrome:** Ctrl+Shift+J ou F12
 - **Internet Explorer:** F12

Atenção: O Console exibe erros de JavaScript, CSS e também erros de cromagem gerados internamente pelo navegador. (extensão defeituosa)

Console de Erros

O console exibe mais que apenas erros:



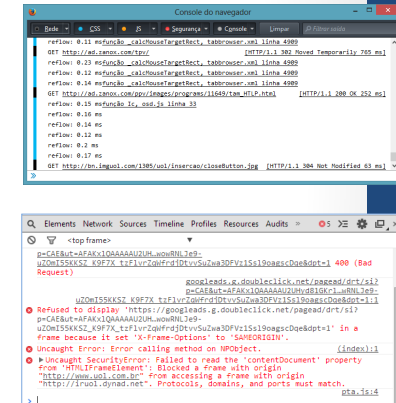
Errors: problemas no seu código que impediram o navegador de continuar o script;



Warnings: Problemas no seu código que o navegador foi capaz de contornar;



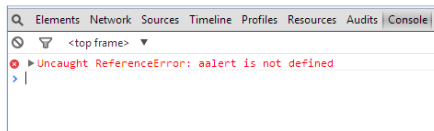
Messages: Anotações de seu código que indicam que esta fazendo normalmente;



Erros e Depuração

- Exemplo de Erro:

```
<script type="text/javascript">
  aalert("Alo Mundo!");
</script>
```



Erros e Depuração - Console

- Você pode usar comandos para enviar mensagens e executar outras tarefas na janela Console do JavaScript;

Comando	Descrição	Exemplo
assert (expressão, mensagem)	Envia uma mensagem se expressão for avaliada como false.	console.assert ((x == 1), "assert message: x != 1");
clear ()	Limpa as mensagens da janela do console, incluindo mensagens de erro de script, e limpa também o script exibido na janela do console. Não limpa o script inserido no prompt de entrada do console.	console.clear ();
count (título)	Envia o número de vezes que o comando count foi chamado para a janela do console. Cada chamada do comando count é identificada exclusivamente pelo título opcional.	console.count ();

Erros e Depuração - Console

Comando	Descrição	Exemplo
<code>debug (message)</code>	Os objetos transmitidos usando o comando são convertidos em um valor de cadeia de caracteres.	<code>console.log("logging message");</code>
<code>dir (object)</code>	Envia o objeto especificado para a janela do console e o exibe em um visualizador de objetos. Você pode usar o visualizador para inspecionar propriedades na janela do console.	<code>console.dir(obj);</code>
<code>dirxml (object)</code>	Envia o nó XML object especificado para a janela do console e o exibe como uma árvore de nós XML.	<code>console.dirxml(xml Node);</code>
<code>error (message)</code>	Envia message para a janela do console. O texto da mensagem está em vermelho e antecedido por um símbolo de erro.	<code>console.error("error message");</code>

Erros e Depuração - Console

Comando	Descrição	Exemplo
<code>group (title)</code>	Os comandos <code>group*</code> podem facilitar a exibição da saída da janela do console em alguns cenários, como quando um modelo de componente está em uso.	<pre>console.log("This is the outer level"); console.group("Level 2 Header"); console.log("Level 2"); console.group(); console.log("Level 3"); console.warn("More of level 3"); console.groupEnd(); console.log("Back to level 2"); console.groupEnd(); console.debug("Back to the outer level");</pre>
<code>info (message)</code>	Envia message para a janela do console. A mensagem é prefaciada por um símbolo de informação.	<code>console.info("info message");</code>

Erros e Depuração - Console

Comando	Descrição	Exemplo
<code>log (message)</code>	Os objetos transmitidos usando o comando são convertidos em um valor de cadeia de caracteres.	<code>console.log("logging message");</code>
<code>profile (reportName)</code>	chamando esta função inicia um perfil JavaScript CPU	<code>function processPixels() { console.profile("Processing pixels"); pixels console.profileEnd(); }</code>
<code>profileEnd ()</code>	Interrompe a sessão atual de perfis JavaScript CPU,	
<code>time (name)</code>	Inicia um temporizador que é identificado pelo parâmetro name opcional.	<code>console.time("app start"); app.start(); console.timeEnd("app start");</code>
<code>timeEnd (name)</code>	Interrompe um temporizador que é identificado pelo parâmetro name opcional. Consulte o comando	

Erros e Depuração - Console

Comando	Descrição	Exemplo
<code>trace ()</code>	Envia um rastreamento de pilha à janela do console. O rastreamento inclui a pilha de chamadas completa e informações como o nome do arquivo, o número da linha e o número da coluna.	<code>console.trace();</code>
<code>warn (message)</code>	Envia message para a janela do console, prefaciada por um símbolo de aviso.	<code>console.warn("warning message");</code>

Referências

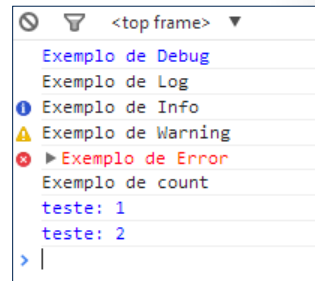
- <https://developers.google.com/chrome-developer-tools/docs/console-api?hl=pt-br>
- <http://msdn.microsoft.com/pt-br/pt-b/library/windows/apps/hh696634.aspx>

Erros e Depuração - Console

- Exemplos

```
<script type="text/javascript">
  console.debug("Exemplo de Debug");
  console.log("Exemplo de Log");
  console.info("Exemplo de Info");
  console.warn("Exemplo de Warning");
  console.error("Exemplo de Error");

  console.log("Exemplo de count");
  console.count("teste");
  console.count("teste");
</script>
```



Variáveis

Variáveis

- Variável é um nome qualquer ao qual se atribui um valor ou dado;
- Uma variável pode conter uma string, número, array, booleano, função, objeto, etc;

Declaração

- Existem três tipos de declarações em JavaScript.
- var:** Declara uma variável, opcionalmente, inicializando-a com um valor.
- let:** Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.
- const:** Declara uma constante apenas de leitura.

Classificação

- Uma variável declarada usando a declaração **var** ou **let** sem especificar o valor inicial tem o valor undefined.
- Uma tentativa de acessar uma variável não declarada resultará no lançamento de uma exceção ReferenceError.

```
<script type="text/javascript">

var a;
console.log("O valor de a é " + a); //"O valor de a é undefined"
console.log("O valor de b é " + b); //exception de erro de referência

</script>
```

Variáveis

- O nome da variável é de livre escolha do programador, ressalvadas as seguintes restrições sintáticas:
 - Podem começar com **letra**, **\$** ou **_**
 - Não pode começar com **números**;
 - É possível usar **letras** ou **números**;
 - É possível usar **acentos** e **símbolos**;
 - Não podem conter **espaços**;
 - Não podem ser **palavras reservadas**;
- Escolha nomes que transmitam uma dica do conteúdo da variável.

Escopo de variável

- Quando você declara uma variável fora de qualquer função, ela é chamada de variável **global**, porque está disponível para qualquer outro código no documento atual.
- Quando você declara uma variável dentro de uma função, é chamada de variável **local**, pois ela está disponível somente dentro dessa função.

Escopo de variável

- JavaScript antes do ECMAScript 6 não possuía escopo de declaração de bloco; pelo contrário, uma variável declarada dentro de um bloco de uma função é uma variável local do bloco que está inserido a função.

```
<script type="text/javascript">

{
  var x = 5;
}
console.log(x); // 5

</script>
```

Escopo de variável

- Esse comportamento é alterado, quando usado a declaração `let` introduzida pelo ECMAScript 6.

```
<script type="text/javascript">
{
  let y = 5;
}
console.log(y); // ReferenceError: y não está definido
</script>
```

Palavras Chaves

- Uma restrição para os nomes de variáveis são as palavras-chave JavaScript.

Tabela 1 - Palavras-Chave JavaScript

break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	try	
do	instanceof	typeof	

Fonte: Aprendendo JavaScript – Shelley Powers

Palavras Chaves

- Devido às extensões propostas à especificação ECMA-262, as palavras da tabela abaixo também são consideradas reservadas.

Tabela 2 – Palavras reservadas da especificação ECMA-262

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

Fonte: Aprendendo JavaScript – Shelley Powers

Palavras Chaves

- Determinadas palavras específicas de JavaScript implementadas na maioria dos navegadores são consideradas reservadas.

Tabela 3 – Palavras reservadas típicas de navegadores (mais comum)

alert	eval	location	open
array	focus	math	outerHeight
blur	function	name	parent
boolean	history	navigator	parseFloat
date	image	number	RegExp
document	isNaN	object	status
escape	length	onload	string

Fonte: Aprendendo JavaScript – Shelley Powers

Diretrizes de nomenclatura

- Funções e variáveis frequentemente começam com letras minúsculas e incorporam uma representação verbal do que a função faz. Ex: `function validaNomeNoRegistro(nome)`
- Muitas vezes, variáveis e funções tem uma ou mais palavras concatenadas em um único identificador. Podemos utilizar um formato chamado **CamelCase**.
- As bibliotecas JavaScript mais novas invariavelmente usam a notação *CamelCase*.
- Também podemos utilizar traços (-) e sublinhados (_) para separar, embora o *CamelCase* torna muito mais legível.

Diretrizes de nomenclatura

- Diversas práticas de nomenclatura, muitas herdadas do Java e outras linguagens de programação podem tornar o código mais fácil de seguir e manter;
- Utilize palavras significativas em vez de algo que tenha juntado rapidamente. Ex: `taxaDeJuros` e não `txJu` ou `tj`;
- Usar plural para coleções; Ex: `var nomesClientes`
- Reservar letras maiúsculas para objetos os torna mais fáceis de diferenciar.

Tipos de Dados

Literais

- Na terminologia JavaScript, a palavra literal designa qualquer dado;
- Existem seis tipos de dados literais conforme descritos a seguir:
 - inteiros;
 - decimais;
 - booleanos;
 - strings;
 - arrays;
 - objetos.

Atenção: a linguagem JavaScript é vagamente tipada, ela não se preocupa com o que as variáveis contém!

Literais

- Seis tipos de dados são os chamados primitivos:
 - Números
 - Strings
 - Booleanos
 - Nulo (null)
 - Indefinido (undefined)
 - Symbol: Um tipo de dado cuja as instâncias são únicas e imutáveis. (ES6)
- **Todos os demais são objetos**
 - Matrizes (arrays)
 - Funções
 - Expressões Regulares
 - Objetos

Declaração

- Mesmo não sendo necessário declarar o tipo de dados antecipadamente, ainda é fundamental saber o tipo de dados que a variável irá armazenar;
- Ao contrário da maioria das linguagens de programação, as variáveis da JavaScript podem conter qualquer tipo de dado;
- **Para declarar uma variável que pertence ao:**
 - **escopo local:** usa-se a palavra-chave **var**.
 - **escopo global:** sem uso da palavra-chave **var**;
- É importante declarar as variáveis na primeira linha de código da região para a qual ela será válida;

Declaração e Atribuição

- Exemplo de declaração de variável e atribuição de dados;

```
<script type="text/javascript">

  idade=0;
  media=0;
  nome="";

  idade= 28;
  media= 40.5;
  nome = "Luiz Henrique";
  nome = "Luiz Henrique" + "de Angeli"; //concatenando

  //utilizando template string
  sobrenome = "de Angeli"
  nome = "Luiz Henrique ";
  console.log(`O nome dele é ${nome} ${sobrenome}`);
</script>
```

Convertendo tipos de dados

- Exemplo de conversões de tipos de dados

```
<script type="text/javascript">

  inteiro = parseInt("10");
  decimal = parseFloat("10.45");
  texto = decimal.toString();

  inteiro = Number("10");
  decimal = Number("10.45");
  texto = String(decimal);

</script>
```

Alguns recursos de String e Number

```
<script type="text/javascript">

nome = "Luiz Henrique de Angeli";
console.log(nome.length);
console.log(nome.toUpperCase());
console.log(nome.toLowerCase());

</script>
```

```
<script type="text/javascript">

s = 1575.7;
console.log(s.toFixed(2));
console.log(s.toFixed(2).replace(".",","));
console.log(s.toLocaleString('pt-br', {style: 'currency', currency: 'BRL'}));
console.log(s.toLocaleString('pt-br', {style: 'currency', currency: 'USD'}));
console.log(s.toLocaleString('pt-br', {style: 'currency', currency: 'EUR'}));

</script>
```

Constantes

- Às vezes você desejará definir um valor e tratar apenas como leitura;
- A **constante** pode ter qualquer valor e, por não poder receber um valor posteriormente, é inicializada com seu valor constante quando é definida.
- Da mesma forma que variáveis, uma constante JavaScript tem escopo global e local.

```
<script type="text/javascript">

const MES_ATUAL = 3.5;

</script>
```

Constantes

- “Pontas soltas”

```
<script type="text/javascript">

const PI = 3.14;
PI = 4;

</script>
```

```
<script type="text/javascript">

const objeto = {
  nome : "Luiz",
  idade : 25
};
Object.freeze(objeto);
objeto.idade = 35;
</script>
```

Operadores

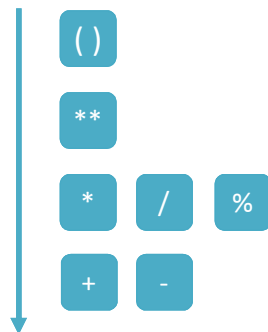
Operadores

- São símbolos, em geral de um ou poucos caracteres, que permitem operações aritméticas, lógicas, etc. Em outras palavras, são usados basicamente para modificar valores de variáveis.
 - Aritméticos;
 - Atribuição;
 - Relacionais;
 - Lógicos;
 - Ternários;

Operadores Aritméticos

Operador	Descrição	Simulação
+	Soma	$3 + 2 \rightarrow 5$
-	Subtração	$3 - 2 \rightarrow 1$
*	Multiplificação	$3 * 2 \rightarrow 6$
/	Divisão	$3 / 2 \rightarrow 1,5$
%	Resto da Divisão	$3 \% 2 \rightarrow 1$
**	Potência	$3 ** 2 \rightarrow 9$
()	Precedência de Operação	$(3 + 2) / 5 \rightarrow 1$

Ordem de Precedência



Operadores de Atribuição

Atribuição	Simplificada
<code>var n = 5</code>	
<code>n = n + 1</code>	<code>n++</code>
<code>n = n - 1</code>	<code>n--</code>
<code>n = n + 5</code>	<code>n+=5</code>
<code>n = n - 5</code>	<code>n-=5</code>
<code>n = n * 5</code>	<code>n*=5</code>
<code>n = n ** 5</code>	<code>n**=5</code>
<code>n = n % 5</code>	<code>n%=5</code>

Operadores Relacionais

Operador	Descrição	Simulação
>	Maior	3 > 2 → true
<	Menor	3 < 2 → false
>=	Maior ou Igual	3 >= 2 → true
<=	Menor ou Igual	3 <= 2 → false
==	Igual	3 == 2 → false
!=	Diferente	3 != 2 → true
===	Identidade ou Igualdade Restrita	2 === 2 → true 2 === '2' → false
!==	Identidade ou Igualdade Restrita	2 !== 2 → false 2 !== '2' → true

Comparação do operador "===" com "=="

Operação	Resultado	Operação	Resultado
' ' == '0'	false	' ' === '0'	false
0 == ''	true	0 === ''	false
0 == '0'	true	0 === '0'	false
false == 'false'	false	false === 'false'	false
false == '0'	true	false === '0'	false
false == undefined	false	false === undefined	false
false == null	false	false === null	false
null == undefined	true	null === undefined	false

Operador "==" não leva em consideração o tipo de dados; e o operador "==="

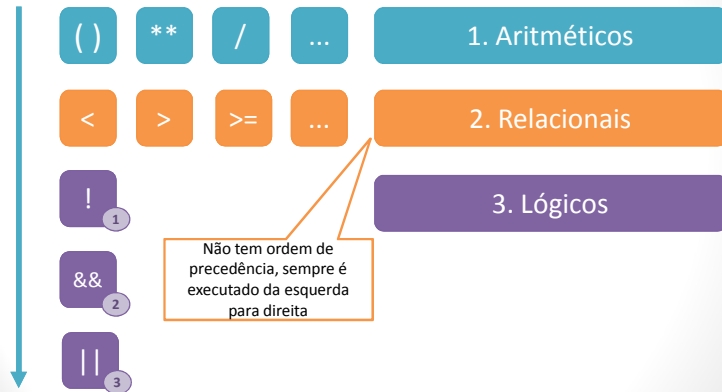
Operadores Lógicos

Operador	Descrição	Simulação
!	Negação	! true → false ! false → true
&&	E ou Conjunção	true && true → true true && false → false false && true → false false && false → false
	OU ou Disjunção	true && true → true true && false → true false && true → true false && false → false

Operadores Lógicos

Operador	Descrição	Simulação
!	Negação	! 3 == 2 → true
&&	E ou Conjunção	3 > 2 && 2 > 1 → true 3 > 2 && 2 < 1 → false
	OU ou Disjunção	3 > 2 2 > 1 → true

Ordem de Precedência



Operadores Ternários

Operador	Descrição	Simulação
?:	Ternário	TESTE ? TRUE : FALSE

Operador	Descrição	Simulação
?:	Ternário	media > 6 ? "Aprovado": "Reprovado"
		numero % 2 == 0 ? "Par": "Ímpar"

Exercícios

1. Criar um programa de calculadora onde o usuário digite 2 números de entrada e apresente em uma caixa de alerta as seguintes operações: Soma, Subtração, Multiplicação, Divisão, Média dos Números.
2. Criar um programa que solicite ao usuário digitar um número, apresente a tabuada deste número.
3. Criar um programa para efetuar o cálculo e a apresentação do valor de uma prestação em atraso, utilizando a fórmula:

$$\text{PRESTAÇÃO} = \text{VALOR} + (\text{VALOR} * (\text{TAXA}/100) * \text{TEMPO})$$
4. Faça um programa que leia a área de uma sala a ser pintada, o custo de uma lata de tinta e a área coberta por uma lata de tinta e calcule o custo da pintura da sala.

$$\text{CUSTO} = (\text{ÁREA TOTAL} / \text{ÁREA LATA}) * \text{CUSTO LATA}$$

Estruturas Condicionais

Estruturas Condicionais

- A maneira de tomarmos decisões é usando estruturas especiais chamadas condições e loops;
- As quais ajudam a controlar quais partes de seu programa será executada e quantas vezes esta parte será executada;
- Uma declaração condicional** é um conjunto de comandos que são executados caso uma condição especificada seja verdadeira. O JavaScript suporta duas declarações condicionais: *if...else* e *switch*.

Estruturas Condicionais

- A instrução mais comum é o **IF**, que verifica uma condição e permite a execução de algum código se a condição for satisfeita.

```
if(condição){
    código condicional
}
```

```
<script type="text/javascript">
    var numero = 10;
    if (numero == 10){
        alert("Número é igual a 10!");
    }
</script>
```

Estruturas Condicionais

- A instrução IF não oferece nenhuma alternativa para caso a condição não é satisfeita. Esse é o objetivo do **ELSE**

```
if(condição){
    código condicional
}else{
    código alternativo
}
```

```
<script type="text/javascript">
    var numero = 11;
    if (numero == 10){
        alert("Número é igual a 10!");
    }else{
        alert("Número é diferente de 10!");
    }
</script>
```

Estruturas Condicionais

O comando **switch** é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos

```
switch(valor){
    case constante_1:
        código condicional
        break;
    case constante_2:
        código condicional
        break;
    case constante_3:
        código condicional
        break;
    default:
        código condicional padrão
}
```

```
<script type="text/javascript">
    numero = 2;
    switch(numero)
    {
        case 1:
            alert("Um");
            break;
        case 2:
            alert("Dois");
            break;
        default:
            alert("Não encontrado");
    }
</script>
```

Exercício 1



1. Criar um programa de calculadora onde o usuário digite 2 números de entrada e escolha a operação:
 1. Soma
 2. Subtração,
 3. Multiplicação,
 4. Divisão,
 5. Média dos Números

Apresente o resultado da operação escolhida pelo usuário

Exercício 2



- Faça um programa que, dada a idade de um nadador pelo usuário (prompt), classifique-o em uma das seguintes categorias:
 - **Não classificado**: menor que 5 anos
 - **Infantil A**: 5 - 7 anos
 - **Infantil B**: 8 - 10 anos
 - **Juvenil A**: 11 - 13 anos
 - **Juvenil B**: 14 - 17 anos
 - **Sênior**: maiores de 18 anos

Exercício 3



- Faça um programa que leia os valores de 4 notas escolares de um aluno.
 - Calcular a média e apresentar a mensagem “aprovado” se a média for maior ou igual a 7, caso contrário deve solicitar a nota de exame do aluno e calcular a nova média entre a primeira média e a nota do exame.
 - Se a nova média for maior ou igual a 5, apresente “aprovado em exame”, caso contrário apresente a mensagem “reprovado”. Informar junto com a mensagem a média obtida.

Estruturas de repetições

Estruturas de Repetições: Loops

- Existe algumas instruções de loop diferentes, mas elas fazem basicamente o mesmo:
 - Repetir um conjunto de ações até uma condição específica seja true.
- Loops:
 - While
 - Do-While
 - For
 - For-in

Estruturas de Repetições: **while**

- O loop `while` é o mais simples de todos;
- Tudo o que ele precisa é de uma condição e algum código condicional;

```
while(condição){
    código condicional
}
```

```
<script type="text/javascript">
var contador=10;
while(contador < 10){
    alert(contador);
    contador++;
}
</script>
```

Estruturas de Repetições: **do-while**

- O loop `do-while` se comporta de modo quase idêntico a um loop `while`, com uma diferença importante: o código condicional é colocado antes da condição. Portanto o código condicional é executado pelo menos uma vez;

```
do{
    código condicional
}while(condição);
```

```
<script type="text/javascript">
var contador=1;
do{
    alert(contador);
    contador++;
}while(contador < 10);
</script>
```

Estruturas de Repetições: **for**

- O loop `for` é recomendado para utilizar com números fixos de repetições especificado;

```
for(início;condição;incremento)
{
    código
}
```

```
<script type="text/javascript">
for(var i=0;i<10;i++){
    alert(i);
}
</script>
```

Estruturas de Repetições: **for-in**

- Loops `for-in` são usados para iterar objetos que não sejam arrays;
- O laço `for-in` interage sobre propriedades enumeradas de um objeto, na ordem original de inserção. O laço pode ser executado para cada propriedade distinta do objeto.
- Quando iterar é importante usar o método `hasOwnProperty()` para excluir propriedades que tenham sido herdadas.

```
for(var index in objetos){
  código
}
```

```
<script type="text/javascript">
  pessoa = {nome:"Luiz",idade:28};
  for(var p in pessoa){
    if(pessoa.hasOwnProperty(p))
      console.log(p + " - " + pessoa[p]);
  }
</script>
```

Estruturas de Repetições: **for-in**

- Tecnicamente você também pode utilizar o loop `for-in` em arrays (por que no JavaScript o array também é um objeto) **mas não é recomendado**;
- Índices de arrays somente se tornam propriedades enumeradas com inteiros (integer).
- Não há garantia de que utilizando o laço `for...in` os índices de um array serão retornados em uma ordem particular ou irá retornar todas as propriedades enumeráveis.
- É recomendável utilizar o laço `for` com índices numéricos ou `Array.prototype.forEach()` ou ainda **`for...of`** quando iteragir sobre arrays onde a ordem é importante.

Estruturas de Repetições: **for-of**

- O loop `for...of` percorre objetos iterativos (incluindo Array, Map, Set, o objeto), chamando uma função personalizada com instruções a serem executadas para o valor de cada objeto distinto.

```
for(var valor of lista){
  código
}
```

```
<script type="text/javascript">
var numeros = [10, 20, 30];

for (let valor of numeros) {
  console.log(valor);
}
</script>
```

Exercício

1. Criar um programa de calculadora onde o usuário digite 2 números de entrada e escolha a operação:
 1. Soma
 2. Subtração,
 3. Multiplicação,
 4. Divisão,
 5. Média dos Números

Apresente o resultado da operação escolhida pelo usuário, Faça a leitura das opções até que o usuário escolha a opção sair;



Exercício de Estrutura Condicional e Repetição



- Gerar um número aleatório de 0 a 100 utilizando a função `random()`. Solicite ao usuário para digitar um número até que ele acerte o número ou ultrapasse 10 tentativas;
- A cada número digitado pelo usuário informe com `alert` como esta seu resultado conforme a diferença entre o número gerado e o número digitado pelo usuário:
 - **Gelado:** diferença de 50 ou superior
 - **Frio:** diferença de 40 a 49
 - **Morno:** diferença de 30 a 39
 - **Quente:** diferença de 20 a 29
 - **Fervendo:** diferença de 10 a 19
 - **Queimando:** diferença de 1 a 9

```
<script type="text/javascript">
numero = parseInt(Math.random() * 100);
alert(numero);
</script>
```

Arrays []

`isObject = true`

Arrays []

- Os literais arrays, na sintaxe JavaScript, são os conjuntos de zero ou mais valores, separados por vírgula e envolvidos por colchetes (`[]`).
- Os valores contidos em um array recebem um índice sequencial começando com zero.

```
var variável = [
  "value1",
  "value2"
];

ou

variável[0] = "value1";
variável[1] = "value2";
```

```
<script type="text/javascript">

var frutas = [
  "laranja",
  "pera",
  "goiaba",
  "morango"
];

alert(frutas[0]);

</script>
```

Arrays []

- Os arrays podem ser estruturas muito rápidas;
- **Infelizmente**, o JavaScript não possui nada parecido com esse tipo de array;
- Em vez disso, o JavaScript possui um objeto que tem algumas características semelhantes ao array;
- Ele converte índices de arrays em sequências de caracteres que são usados para criar as propriedades; (Isso é mais lento);
- O JavaScript em si é confuso quanto a diferenciar arrays e objetos. O operador `typeof` (tipo de) diz que o tipo de um array é `object`

Arrays []

Os valores do array que mostramos são strings e devem ser escritos entre aspas. Um array pode conter qualquer tipo de dado da JavaScript, incluindo expressões, objetos e outros arrays.

```
<script type="text/javascript">

  a = 4; b = 12;
  ArrayMisto = [ "laranja", 34, "casa", true, [1,3,5], a+b ];

  alert(ArrayMisto[0]);
  alert(ArrayMisto[1]);
  alert(ArrayMisto[3]);
  alert(ArrayMisto[4][1]);
  alert(ArrayMisto[5]);

</script>
```

Arrays [] – propriedades e métodos

- Variáveis JavaScript podem ser objetos. Arrays são tipos especiais de objetos.
- Devido a isso, você pode ter variáveis de tipos diferentes na mesma matriz.
- O objeto Array tem propriedades e métodos pré-definidos:

Arrays [] – propriedades

Propriedade	Descrição
constructor	Retorna a função que criou o protótipo do objeto Array
length	Define ou retorna o número de elementos em uma matriz
prototype	Permite adicionar propriedades e métodos para um objeto Array

Arrays [] – métodos

Método	Descrição
concat ()	Junta-se duas ou mais matrizes, e retorna uma cópia das matrizes unidas
indexOf ()	Procure a matriz para um elemento e retorna a sua posição
join ()	Junta-se todos os elementos de uma matriz em uma string
lastIndexOf ()	Procure a matriz de um elemento, a partir do final e retorna a sua posição
pop ()	Remove o último elemento de uma matriz e retorna esse elemento
push ()	Acrescenta novos elementos ao final de uma matriz e retorna o novo comprimento
reverse ()	Inverte a ordem dos elementos em uma matriz

Arrays [] – métodos

Método	Descrição
shift ()	Remove o primeiro elemento de uma matriz e retorna esse elemento
slice ()	Seleciona uma parte de uma matriz e retorna a nova matriz
sort ()	Classifica os elementos de uma matriz
splice ()	Adiciona / remove elementos de uma matriz
toString ()	Converte uma matriz em uma string, e retorna o resultado
unshift ()	Adiciona novos elementos para o início de uma matriz, e retorna o novo comprimento
valueOf ()	Retorna o valor primitivo de um array

Veja mais métodos em:

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array#M%C3%A9todos

Arrays [] – propriedades e métodos

```
<script type="text/javascript">
```

```
frutas = ["laranja", "pera", "goiaba", "morango"];
console.log("Tamanho: " + frutas.length);
```

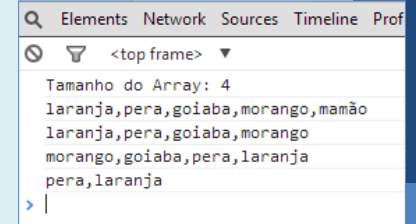
```
frutas.push("mamão");
console.log(frutas.toString());
```

```
frutas.pop();
console.log(frutas.toString());
```

```
frutas.reverse();
console.log(frutas.toString());
```

```
frutas.splice(0, 2);
console.log(frutas.toString());
```

```
</script>
```



Exercício 1

- Criar um programa que solicite ao usuário digitar 10 números. Utilize a função `prompt`. Adicione os números em array utilizando as regras abaixo.
 - Caso o número digitado seja par, remova o número que esta na ultima posição do array e adicione o número digitado no final array.
 - Caso seja impar adicionar o número digitado ao final do array multiplicado por 5 e inverta a ordem dos elementos da matriz.
 - Apresente ao final da leitura de 10 números:
 - A soma dos números contidos no Array;
 - A média dos números contidos do Array
 - O tamanho do array

Números: 1,2,3,4,5,6,7,8,9,10
 1 - Soma: 73
 2 - Média: 14,6
 3 - Tamanho: 5

Exercício 2

- Crie um game de jo-ken-po.
- A cada rodada, o jogador vê o menu: Escolha sua jogada: 1 - Papel 2 - Pedra 3 - Tesoura
- O jogo lê a opção do jogador e verifica se é válida. Se for inválida, o jogador perde a rodada e o jogo acaba. Se for válida, o computador escolhe uma resposta aleatória, que é mostrada ao jogador. Se o jogador ganhar, ele pode jogar mais uma rodada e sua pontuação aumenta. O jogo acaba quando o jogador perde uma rodada. A pontuação total é mostrada no fim do jogo.

Funções ()

isObject = true

Funções : “escrevendo código para depois”

- As funções são como pequenos pacotes de códigos JavaScript prontos para entrar em ação.
- Basta dar um nome ao bloco de código e você poderá chamá-lo a partir de outras áreas de seu programa quando desejar.
- As funções em JavaScript são **objetos**;

```
function função() {
}
```

```
<script type="text/javascript">
function meuNome() {
  alert("Luiz Henrique");
}
</script>
```

Funções : Argumentos

- As funções podem ser projetadas para receber quantos argumentos você quiser de qualquer tipo de dados.

```
function função(argumento1, argumento2){
}
```

```
<script type="text/javascript">
function soma(num1, num2){
  alert(num1, num2);
}
</script>
```

Funções : Instruções de retorno

- Uma função pode **retornar** dados para a instrução que a chamou.
- Para fazer uma função retornar um valor, usamos a palavra-chave **return** seguida do valor que desejamos que ela retorne.

```
function função(argumento1, argumento2){
  return argumento1 + argumento2;
}
```

```
<script type="text/javascript">
function subtracao(num1, num2){
  return num1 - num2;
}
</script>
```

Funções : mantendo suas variáveis separadas

- As variáveis declaradas no **escopo global** podem ser acessadas a partir de qualquer outro código JavaScript que esteja na página web atual.
- Podemos declarar uma variável no **escopo local**, essa variável só existe dentro dos limites da função.

```
variavelGlobal = 10;

function função(){
    var variavelLocal = 10;
}
```

Relembrando: variáveis do **escopo global** declaramos sem a palavra-chave **var**

Funções : mantendo suas variáveis separadas

escopo global

```
<script type="text/javascript">

    function soma(){
        num1=2;
        num2=3;
        return num1 - num2;
    }

    num1 = 10;
    resultado = soma();
    alert(num1);

</script>
```

escopo local

```
<script type="text/javascript">

    function soma(){
        var num1=2;
        var num2=3;
        return num1 - num2;
    }

    num1 = 10;
    resultado = soma();
    alert(num1);

</script>
```

Funções : com parâmetros infinitos

- Uma função pode **receber** parâmetros infinitos .

```
<script type="text/javascript">

function somaTudo(){
    var soma = 0;
    for(var i=0;i<arguments.length;i++){
        soma+=arguments[i];
    }
    return soma;
};

console.log(somaTudo(20));
console.log(somaTudo(20,50,90,35,76,65,50,50,50,14));

</script>
```

Exercício de Funções

- Faça um programa que solicite ao usuário digitado o salário bruto e retorne o salário líquido:
 - Faça uma função para calcular o valor do desconto do INSS
 - Faça uma função para calcular o valor do desconto do IRRF
 - Faça uma função que retorne o salário líquido;



Salário Bruto	Alíquota	Salário Base Calculado	Alíquota	Dedução
até 1.317,07	8%	Até 1.787,77	0%	
de 1.317,08 até 2.195,12	9%	De 1.787,78 até 2.679,29	7,5%	134,08
de 2.195,13 até 4.390,24	11%	De 2.679,30 até 3.572,43	15%	335,03
Acima de 4.390,24	R\$ 482,93	De 3.572,44 até 4.463,81	22,5%	602,96
		Acima de 4.463,81	27,5%	826,15
		R\$ 179,71 por dependente legal		

Exercício de Funções

- **INSS:**
 - $\text{INSS} = \text{SalarioBruto} - \text{ValorTabelaINSS}$
- **IRRF**
 - $\text{ValorBaseIRRF} = \text{SalarioBruto} - \text{INSS} - (\text{Dependentes} * 179.71);$
 - $\text{IRRF} = (\text{ValorBaseIRRF} * \text{PercentualDescontoTabela}) - \text{Dedução}$
- **LÍQUIDO**
 - $\text{SalarioLiquido} = \text{SalarioBruto} - \text{INSS} - \text{IRRF}$

Salário Bruto	Alíquota	Salário Base Calculado	Alíquota	Dedução
até 1.317,07	8%	Até 1.787,77	0%	
de 1.317,08 até 2.195,12	9%	De 1.787,78 até 2.679,29	7,5%	134,08
de 2.195,13 até 4.390,24	11%	De 2.679,30 até 3.572,43	15%	335,03
Acima de 4.390,24	R\$ 482,93	De 3.572,44 até 4.463,81	22,5%	602,96
		Acima de 4.463,81	27,5%	826,15
		R\$ 179,71 por dependente legal		

Funções Conceitos

- **Funções são objetos que podem:**
 - Ser usados dinamicamente no tempo de execução;
 - Ser atribuído a variáveis e ter suas referências copiadas a outras variáveis;
 - Ser passadas como argumentos a outras funções e também ser retornados por outras funções
 - Ter métodos e propriedades próprios

Funções anônimas

- É possível criar uma função usando o construtor `Function` e atribuir a uma variável;

```
<script type="text/javascript">
    soma = new Function("x","y","z","return x + y + z");
    console.log(soma(2,3,1));
</script>
```

Expressões de Função

- Funções literais também são conhecidas como expressões de funções, por que a função é criada como parte de uma expressão;
- Elas lembram *funções anônimas*, pelo fato de não ter nome;

```
<script type="text/javascript">
    add = function (num1, num2){
        return num1 + num2;
    }
    console.log(add(2,4));
</script>
```

Função de Callback

- Funções podem ser passadas como argumentos a outras funções, isso é possível por ser objeto;

```
<script type="text/javascript">

function soma(num1,num2,callback){
    var soma = num1+num2;
    callback();
}

function callback(){
    alert("Função de callback");
}

soma(1,2,callback);

</script>
```

```
<script type="text/javascript">

function soma (num1,num2,callback){
    var soma = num1+num2;
    callback();
}

soma(1,2,function(){
    alert("Função de callback");
});

</script>
```

Função de Callback

- Exemplo utilizando função com variável ou direta.

```
soma = function(numero1, numero2, funcao){
    var resultado = numero1 + numero2;
    funcao(resultado);
};

callback1 = function(resultado){
    console.log("O Resultado é: " + resultado);
};

callback2 = function(resultado){
    console.log("The Result is: " + resultado);
}

soma(15,20, callback1);
soma(10,20, callback2);

soma(50,60,function(res){
    console.log("O resultado da soma é: " + res);
});
```

Função de Callback

- Exemplo de Função de Callback com temporizador:
 - setInterval**: Executa a função infinitamente;
 - setTimeout**: Executa a função somente uma vez;

```
<script type="text/javascript">

function contador(){
    console.count("contador");
}

setInterval(contador, 1000);

</script>
```

```
<script type="text/javascript">

interval = setInterval(function(){
    console.count("contador");
}, 1000);

</script>

clearInterval(contador);
```

Exercício de Funções



1. Criar um programa que funcione como um cronometro regressivo, o usuário digitar o tempo que deseja em segundos e o programa diminui o tempo até chegar a zero, mostrar a cada segundo o tempo que falta para zero.
2. Criar um programa que tenha um lista de pessoas (array) com o nome e idade. Escreva uma função que receba como parâmetro esta lista e uma função de callback.
 - a) Na função, percorra a lista de pessoa verificando se a idade é menor que 18 anos.
 - b) Quando encontrar pessoa com menor de 18 anos, execute a função de callback, passando a lista e o índice desta pessoa;
 - c) Na função de callback remova o índice recebido do array e informe com uma mensagem o nome da pessoa excluída;
 - d) Apresente a lista de pessoas atualizada;

Retornando Funções

- Funções também podem ser usadas como valor de retorno;

```
<script type="text/javascript">

    instalador = function(){
        var passo = 0;
        return function(){
            passo++;
            return passo;
        }
    }

    proximo = new instalador();
    console.log(proximo());
    console.log(proximo());

</script>
```

Funções autodefiníveis

- Funções podem ser definidas dinamicamente e ser atribuídas a variáveis;
- Se você escrever uma função e atribuí-la a uma variável que já armazenava outra função, você sobrescreverá a função antiga;

```
<script type="text/javascript">

    meuNome = function(){
        alert("Luiz Henrique");
        meuNome = function(){
            alert("Luiz Henrique de Angeli");
        }
    }
    meuNome();
    meuNome();

</script>
```

Funções imediatas

- O padrão de função imediata é uma sintaxe que lhe permite executar uma função tão logo ela seja definida;
- Ajuda a encapsular certa quantidade de trabalho que você queira realizar, eliminando qualquer traços de variáveis globais;

```
<script type="text/javascript">

    (function(){
        var dias = ["Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sab"];
        var hoje = new Date();
        var mensagem = "Hoje é " + dias[hoje.getDay()];
        alert(mensagem);
    })();

</script>
```

Funções imediatas - Parâmetros

- Passando parâmetros para a função imediata;

```
<script type="text/javascript">

    (function(nome){
        var dias = ["Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sab"];
        var hoje = new Date();
        var mensagem = "Olá " + nome + " Hoje é " +
            dias[hoje.getDay()];
        alert(mensagem);
    })("Luiz Henrique");

</script>
```

Funções imediatas - Retorno

- Recebendo retorno de uma função imediata;

```
<script type="text/javascript">

    soma =(function(num1, num2) {
        return num1 + num2;

    } (1,2));

    console.log(soma);

</script>
```

Exercício de Funções



- Criar um programa onde o usuário deve digitar uma idade (prompt). Criar uma função que receba esta idade:
 - Caso a idade seja maior ou igual a 18 anos, retorne uma função que solicite o nome e cpf da pessoa;
 - Caso a idade seja menor que 18 anos, retorne uma função que solicite o nome, nome do responsável e RG.

DOM

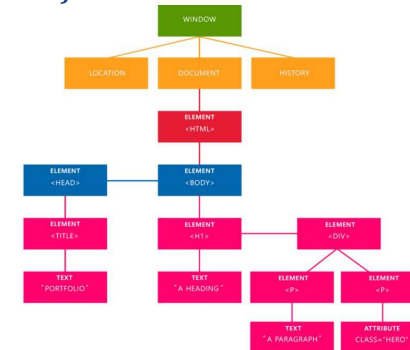
O que é o DOM?

- O DOM (Document Object Model) é uma interface que representa como os documentos HTML e XML são lidos pelo seu browser.
- Após o browser ler seu documento HTML, ele cria um **objeto** que faz uma representação estruturada do seu documento e define meios de como essa estrutura pode ser acessada.
- É possível acessar e manipular o DOM utilizando JavaScript.

O que é o DOM?

- O objeto *document* que é responsável por conceder ao código Javascript todo o acesso a **árvore DOM** do navegador Web.
- Portanto, qualquer coisa criado pelo navegador Web no modelo da página Web poderá ser acessado através do objeto *document*.
- Usa-se o DOM principalmente para atualizar uma página Web ou quando se quer construir uma interface de usuário avançada.
- Com o DOM pode-se mover itens dentro de uma página ou criar efeitos CSS bastante interessantes sem precisar nem mesmo recarregar a página.

Representação



Fonte: <https://tableless.com.br/entendendo-o-dom-document-object-model/>

Manipulando o DOM

Window

- O objeto *window* representa uma janela que contém um elemento DOM; a propriedade *document* aponta para o documento DOM document carregado naquela janela.
- Acessando link é possível verificar a documentação do *window*.

<https://developer.mozilla.org/pt-BR/docs/Web/API/Window>

DOM

- O DOM possui muitos métodos, são eles que fazem a ligação entre os nodes (elementos) e os eventos;
- Através do objeto *document* pode-se ter acesso a um grande número de propriedades.
- Acessando link é possível verificar a documentação do *document*.
<https://developer.mozilla.org/en-US/docs/Web/API/Document>

Exemplo 1

window.document.write()

```
<script type="text/javascript">
window.document.write("Olá Mundo");
window.document.write(window.document.charset);
window.document.write(window.navigator.appname);
window.document.write(window.document.URL);

</script>
```

Acessando os elementos no DOM

Métodos de Acessos

- Por Marca
- Por ID
- Por Nome
- Por Classe
- Por Seletor

Acessando os elementos no DOM

Por Marca

```
<p> Primeiro Paragrafo </p>
<p> Segundo Paragrafo </p>

<script type="text/javascript">

var p1 = window.document.getElementsByTagName('p')[0];
window.document.write(p1.innerText);

var p2 = window.document.getElementsByTagName('p')[1];
p2.style.color = 'blue';
p2.style.backgroud = 'red';
p2.innerHTML = "novo <b>texto</b>";
window.document.write(p2.innerHTML);

</script>
```

Acessando os elementos no DOM

Por ID

```
<p id='p1'> Primeiro Paragrafo </p>
<p id='p2'> Segundo Paragrafo </p>
<div id='divMsg'></div>

<script type="text/javascript">

var divMsg = window.document.getElementById('divMsg');
divMsg.style.background = 'green';
divMsg.innerHTML = 'mensagem na div';

</script>
```

Acessando os elementos no DOM

Por Nome

```
<p name='p1'> Primeiro Paragrafo </p>
<p name='p2'> Segundo Paragrafo </p>
<div name='divMsg'></div>

<script type="text/javascript">

var divMsg = window.document.getElementsByName('divMsg')[0];
divMsg.style.background = 'green';
divMsg.innerHTML = 'mensagem na div';

</script>
```

Acessando os elementos no DOM

Por Classe

```
<p class='paragrafo'> Primeiro Paragrafo </p>
<p class='paragrafo'> Segundo Paragrafo </p>
<div name='box'></div>

<script type="text/javascript">

var divMsg = window.document.getElementsByClassName('paragrafo')[0];
divMsg.style.background = 'green';
divMsg.innerHTML = 'mensagem na div';

</script>
```

Acessando os elementos no DOM

Por Seletor (novo)

```
<p class='paragrafo'> Primeiro Paragrafo </p>
<p class='paragrafo'> Segundo Paragrafo </p>
<div id='box'></div>

<script type="text/javascript">

var p1 = window.document.querySelector('#p1');
p1.style.background='red';

var p2 = window.document.querySelectorAll('p')[1];
p2.style.background='blue';

</script>
```

Eventos

- Eventos Dom (Dom Events) são utilizados para notificar o código de novidades durante a navegação do usuário.
- Cada evento é representado por um objeto que é baseado na interface *Event*.
- Eventos podem representar desde interações básicas do usuário (cliques, rolagem da página...) até notificações automáticas de novidades no DOM.

Lista dos Eventos: <https://developer.mozilla.org/pt-BR/docs/Web/Events>

Eventos: on....

```
<style> .... </style>
<div id="divArea" onclick="clicar()" onmouseenter="entrar()"
onmouseout="sair()"> Div de Teste </div>
<script type="text/javascript">
var a=window.document.getElementById('divArea');
function clicar(){
    a.innerText = "clicou";
}

function entrar(){
    a.innerText = "entrou";
    a.style.background = "red";
}

function sair(){
    a.innerText = "saiu";
    a.style.background = "blue";
}
</script>
```

Eventos: listener

```
<style> .... </style>
<div id="divArea"> Div de Teste </div>
<script type="text/javascript">

var a=window.document.getElementById('divArea');
a.addEventListener('click', clicar);
a.addEventListener('mouseenter', entrar);
a.addEventListener('mouseout', sair);

function clicar(){}

function entrar(){}

function sair(){}

</script>
```

Acessando elementos de formulário

```
<input type="number" id="numero1">
<input type="number" id="numero2">
<button type="button" onclick="somar()">S</button>
<div id="resultado"></div>
<script type="text/javascript">

function somar(){
    var numero1 = document.getElementById("numero1").value;
    var numero2 = document.getElementById("numero2").value;
    var resultado = numero1 + numero2;
    document.getElementById("resultado").innerHTML = resultado;
}

</script>
```

Exercício

1. Agora refaça o exercício da calculadora, porém, utilizando os recursos HTML, CSS e JavaScript.

128			
	C	BKS	/
7	8	9	*
4	5	6	-
1	2	3	+
	0	.	=

Exercício

1. Faça o jogo do campo minado, a matriz deve ser 10 x 10 e deve conter 10 bombas;

Tentativas: 10
Recorde: 10
Nome: Luiz Henrique

Minas: 100 Bombar: 20 Iniciar

Mais do objeto Window

Objeto History

- Este objeto mantém um histórico de páginas carregadas no navegador;
- Assim, seus métodos e propriedades ter a ver com os botões avançar e voltar do navegador;
- Você pode percorrer history os métodos
 - **back ()** : Volta para a página anterior;
 - **forward ()** : Avança para a próxima página;
 - **go (n)** : Avança ou volta passando o número da página (-3) (+3)

Objeto Screen

- Este objeto contém as informações sobre a tela de exibição;
- Incluindo largura, altura, cor, profundidade;
- As propriedades exatas suportadas pode variar de navegador para navegador e versão para versão:
 - availTop e availLeft : A posição mais alta e mais a esquerda onde uma janela pode ser posicionada;
 - availWidth e availHeight: Largura e Altura;
 - colorDepth: Profundidade da cor da tela;
 - pixelDepth: Profundidade de bit da tela;

Objeto Navigator

- Esse objeto fornece informações sobre o navegador ou outro agente que acesse a página;
- Sistema Operacional, Navegador, Política de segurança, linguagem e se os cookies estão habilitados;
- Alguns navegadores também fornecem um array de plugins instalados;
 - appName: O nome da base de código do navegador;
 - appName: O nome do navegador;
 - appMinorVersion: O número da menor versão do navegador;
 - appVersion: O número principal da versão

Objeto Navigator

- cookieEnabled: Se os cookies estão habilitados;
- mimeType: Os MIME suportados;
- onLine: se o usuário está online;
- platform: A plataforma que o navegador está operando;
- plugins: Plugins suportados no navegador;
- userAgent: Descrição integral do agente do navegador;
- userLanguage: Linguagem suportada pelo navegador;

Objeto Location

- As propriedades dos objetos estão relacionadas a localização da página.
 - hash: Para url com formato #algumHash
 - host: nome do host e porta da url
 - hostname: o nome do host apenas
 - href: URL inteira
 - pathname: O nome do caminho que segue o domínio;
 - port: A porta da URL;
 - protocol: O protocolo usado com a url
 - search: String de consulta, qualquer coisa após ?
 - target: Se dado, o nome do alvo da url;

Geolocalização

- Geolocalização é funcionalidade que foi lançada em 2008;
- A especificação que trata da geolocalização define uma API destinada a oferecer acesso via script as informações relacionadas a localização geográfica do dispositivo que hospeda a funcionalidade;
- A captação dos parâmetros de localização se faz com uso de GPS;
- A API esta implementada no objeto `navigator` como filho deste, em um objeto chamado `geolocation`;

Geolocalização

- Métodos:
 - **`getCurrentPosition`**: Obtem a posição do dispositivo do usuário;
 - **`watchPosition`**: Obtem a posição do dispositivo continuamente;
 - **`clearWatch`**: Encerra a leitura continua da posição, funciona igual o `setInterval` e `clearInterval`;

Geolocalização

```
<script type="text/javascript">

    function minhaPosicao() {
        if (navigator.geolocation) {
            navigator.geolocation.getCurrentPosition(sucesso);
        } else {
            alert("Navegador não suporta geolocalização!");
        }
    }

    function sucesso(posicao) {
        console.log(posicao);
        console.log("Latitude: " + posicao.coords.latitude);
        console.log("Longitude: " + posicao.coords.longitude);
    }

    minhaPosicao();

</script>
```

Atenção: Navegador pode bloquear o acesso;

Acelerômetro

- O navegador mobile tem a possibilidade de ler os dados dos sensores, como acelerômetro e giroscópio diretamente de Javascript.
- Manipulador `window.ondevicemotion`
- Objeto de evento:
 - **`event.accelerationGravity`**: com x, y, z atributos
 - **`event.accelerationIncludingGravity`**: com x, y, z atributos
 - **`event.rotationRate`**: com alfa, beta, gama atributos

<http://www.albertosarullo.com/demos/accelerometer/>

Canvas

Canvas

- <canvas> é um elemento HTML que pode ser usado para desenhar usando linguagem de "script" (normalmente JavaScript).
- Isto pode ser usado, por exemplo, para desenhar gráficos, fazer composições de fotos ou simples (e não tão simples) animações.
- Canvas fornece uma API muito poderosa utilizada pelo script para criar desenhos.

Canvas

- O elemento <canvas> tem apenas dois atributos - width e height. Ambos são opcionais e podem ser aplicados utilizando as propriedades DOM respectivas. Se não forem especificados, o canvas será iniciado com 300 pixels de largura por 150 pixels de altura.
- <canvas> cria uma superfície de desenho de tamanho fixo que expõe um ou mais contextos de renderização, que são usados para criar e manipular o conteúdo mostrado.

Canvas

- Inicialmente o canvas é branco e para mostrar algo, primeiro um script precisa acessar o contexto de renderização e desenhar sobre ele.
- O elemento <canvas> tem um método chamado getContext(), usado para obter o contexto de renderização e suas funções de desenho.
- getContext() recebe o tipo de contexto como parâmetro, para gráficos 2D, deverá ser especificado "2d".

```
<script type="text/javascript">  
    var canvas = document.getElementById('canvas');  
    var ctx = canvas.getContext('2d');  
</script>
```

Prototype

Objetos JavaScript

Objetos JavaScript

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

Objetos browser:

- Window
- Navigator
- Screen
- History
- Location

HTML DOM

- Document
- Element
- Attributes
- Events

HTML Elements Object:

- Form
- Input
 - Checkbox
 - Radio
 - Text
- Label
- Textarea

<http://www.w3schools.com/jsref/>

Protótipos – (prototype)

- Todo objeto é ligado a um protótipo de onde ele pode herdar propriedades.
- Todos os objetos criados de literais são associados a Object.prototype
- O protótipo fornece uma propriedade prototype que permite estender qualquer objeto, como String, Number, etc;
- Você pode usar essa prototype para derivar novos métodos e novas propriedades para um objeto, em vez de herança;

Protótipos – (prototype)

- Exemplo String

```
<script type="text/javascript">

  console.log(String);
  console.log(new String());
  console.log(String.prototype);

</script>
```


Protótipos – (prototype)

- Exemplo de alteração do protótipo com String

```
<script type="text/javascript">

    String.prototype.tamanho = function(){
        return this.length;
    }

    nome = "Luiz Henrique";
    console.log(nome.tamanho());

</script>
```

Protótipos – (prototype)

- Exemplo de alteração do protótipo com DATE

```
<script type="text/javascript">

Date.prototype.diaSemana = function(){
    var dias = ["Dom", "Seg", "Ter", "Qua", "Qui", "Sex", "Sab"];
    return "Hoje é " + dias[this.getDay()];
};

Date.prototype.dataFormatada = function(){
    return this.getDate() + "/" + this.getMonth() + "/" + this.getFullYear();
};

data = new Date();
console.log(data.diaSemana());
console.log(data.dataFormatada());

</script>
```

Exercício de Prototype

- Criar um programa que faça a leitura de um texto utilizando o prompt.
- Altere o protótipo do objeto String e:
 - Adicione uma função que percorra a String e retorne a quantidade de vogais.
 - Adicione uma função que percorra a String e retorne a quantidade de consoantes;



Objetos { }

isObject = true

Objetos {}

- Objetos no JavaScript são grupos mutáveis com índices;
- No JavaScript, matrizes, funções, expressões regulares são objetos; e objetos são objetos;
- Um objeto é um contêiner de propriedades, e elas possuem:
 - nome: Pode ser qualquer sequência de caracteres, incluindo vazia
 - valor: Pode ser qualquer um novo JavaScript, exceto undefined;
- Objetos no JavaScript são livres de classe;

Objetos Literais

- Um objeto literal é um par de chaves contendo zero ou mais pares nome/valor;
- Podem aparecer em qualquer lugar onde uma expressão possa ser usada;

```
<script type="text/javascript">

var objeto_vazio = {};
var pessoa = {
    nome      : "Luiz Henrique",
    sobrenome  : "de Angeli"
};

</script>
```

Objetos Literais

- As aspas no nome da propriedade são opcionais se ela for legal no JavaScript e não uma palavra reservada;
- Então aspas são necessárias para "primeiro nome";
- Mas são opcionais para primeiro_nome;

```
<script type="text/javascript">

var pessoa = {
    nome      : "Luiz Henrique",
    sobrenome  : "de Angeli",
    "data nasc" : "06/10/1985"
};

</script>
```

Objetos Literais – [Recuperando e Atualizando]

- Você pode acessar o valor com um par de [];
- Ou a notação se a propriedade não é uma palavra reservada e é uma expressão literal;

```
<script type="text/javascript">

var pessoa= {
    nome      : "Luiz Henrique",
    sobrenome  : "de Angeli",
};

pessoa["nome"]
pessoa.nome

</script>
```

```
<script type="text/javascript">

var pessoa= {
    nome      : "Luiz Henrique",
    sobrenome  : "de Angeli",
};

pessoa["nome"] = "Luiz";
pessoa.nome = "Luiz";

</script>
```

Objetos Literais

- Objetos podem ser aninhados;

```
<script type="text/javascript">

var voo= {
  nome      : "Luiz Henrique",
  sobrenome  : "de Angeli",
  numero    : 1092,
  partida : {
    cidade : "Maringá",
    hora   : "06:05"
  },
  chegada : {
    cidade : "Curitiba",
    hora   : "06:55"
  }
};

</script>
```

Objetos Literais – [Referência]

- Objetos são passados por referência. Eles nunca são copiados.;

```
<script type="text/javascript">

  a = {id : 1};
  b = a;

  b.id = 2;

  console.log(a); //2
  console.log(b); //2

  console.log(a == b); //true
  console.log({} == {}); //false

</script>
```

Objetos Literais – [Reflexão]

- É fácil inspecionar um objeto para determinar quais propriedades ele possui;
- O operador `typeof` pode ser muito útil;
- Tomar cuidado com o protótipo herdado, você pode usar o método `hasOwnProperty` que retorna `true` ou `false`;

```
<script type="text/javascript">
  console.log(typeof voo.nome); //string
  console.log(typeof voo.numero); //number
  console.log(typeof voo.partida); //object
  console.log(typeof voo.teste); //undefined
  console.log(typeof voo.toString); //function
  console.log(voo.hasOwnProperty("numero")); //true
  console.log(voo.hasOwnProperty("toString")); //false
</script>
```

Objetos Literais – [Enumeração]

- O comando `for-in` pode iterar todos os nomes de propriedades de um objeto;
- Quando iterar é importante usar o método `hasOwnProperty()` ou operador `typeof` para excluir propriedades que tenham sido herdadas ou funções.
- `hasOwnProperty`: determina se um objeto tem uma propriedade com o nome especificado.

```
<script type="text/javascript">
  for(v in voo){
    if(voo.hasOwnProperty(v))
      console.log(v + " - " + voo[v]);
  }
</script>
```

Objetos Literais – [delete]

- O operador `delete` pode ser usado para remover uma propriedade de um objeto.

```
<script type="text/javascript">

    delete voo.nome;

    console.log(voo);

</script>
```

exercício de objetos



- Criar um programa que faça a leitura de um time de vôlei (6 jogadores):
 - Time
 - Nome do Time
 - Estado
 - Cidade
 - Jogadores (6)
 - Nome
 - Idade
 - Numero
 - Titular (true/false)
 - Apresentar a média de idades do time;

Objetos [invocação]

- Quando uma função é armazenada como uma propriedade de um objeto, ela é chamada de **método**;
- Quando um método é invocado, `this` é associado àquele objeto;

```
<script type="text/javascript">

    meuObjeto = {
        valor : 0,
        incrementa : function(){
            this.valor+=1;
        }
    };

    meuObjeto.incrementa();
    console.log(meuObjeto.valor); //1
    meuObjeto.incrementa();
    console.log(meuObjeto.valor); //2
```

exercício de objetos



- Criar um programa que tenha um objeto calculadora com:
 - Atributos:**
 - Número 1
 - Número 2
 - Operação (1,2,3 ou 4)
 - Métodos**
 - Soma
 - Subtração
 - Multiplicação
 - Divisão
 - Resultado
 - Solicite ao usuário 2 números e escolher a operação, apresente o resultado da operação ;

exercício de objetos



• Criar um objeto carro com os:

• Atributos:

- Velocidade
- Distancia
- Ligado
- Tempo
- Alerta Velocidade

• Métodos

- Ligar
- Desligar
- Andar
- Acelerar
- Frear
- getVelocidadeMedia

```

• carro.ligar();
• carro.acelerar();
• carro.acelerar();
• carro.acelerar();
• carro.andar();
• carro.andar();
• carro.acelerar();
• carro.acelerar();
• carro.andar();
• carro.andar();
• carro.frear();
• carro.frear();
• carro.andar();
• carro.andar();
• carro.desligar();
• carro.getVelocidadeMedia()
  // 36.66
  
```

exercício de objetos



- Não permitir andar com o carro se ele não estiver ligado
- O método acelerar deve aumentar a velocidade em 10km
- O método frear deve diminuir a velocidade em 10km
- O método andar deve aumentar a distancia conforme a velocidade e o tempo de viagem;
- Emitir um aviso quando o carro ultrapassar o limite estipulado de velocidade;

JSON

JavaScript Object Notation
(Notação de Objetos JavaScript)

`isObject = true`

JSON {}

- É um formato de transferência de dados;
- É baseado um subconjunto da Linguagem JavaScript
- Foi criado por Douglas Crockford [RFC4627](https://tools.ietf.org/html/rfc4627)
- Ele é leve e conveniente de usar em várias linguagens, especialmente JavaScript;

<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

JSON {}

- Ele é apenas uma combinação das notações literais de:
 - OBJETOS {} e ARRAYS [];
 - Exemplo: {nome: "Luiz", notas : [8,9,7,6]}
- Em strings JSON você não pode usar funções literais nem expressões regulares literais;
- A simplicidade de JSON tem resultado em seu uso difundido, especialmente como uma alternativa para XML em AJAX.

<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

Padrões

Padrão de namespace

- Os namespaces ajudam a reduzir os números de globais pelos nossos programas;
- Ao mesmo tempo também ajuda a evitar colisões de nomeação o prefixação excessiva de nomes;
- O JavaScript não possui namespaces nativamente na sintaxe da linguagem, mas é fácil de conseguir;
- Em vez de poluir o escopo global com variáveis e funções, você pode criar um objeto global para sua aplicação o biblioteca;
- Em seguida você pode adicionar todas as funcionalidades a esse objeto;

Padrão de namespace

- Normalmente os desenvolvedores usam uma convenção de criar a variável global EM MAIÚSCULAS;

Antipadrão

```
<script type="text/javascript">
    function soma(num1, num2){
    }

    function subtracao(num1, num2){
    }

    pessoa = {};
    modulo = {};
</script>
```

Com Padrão Aplicado

```
<script type="text/javascript">
    MYAPP = {};

    MYAPP.soma = function(num1, num2){
    };

    MYAPP.subtracao = function(num1, num2){
    };

    MYAPP.pessoa = {};
    MYAPP.modulo = {};
</script>
```

Funções construtoras customizadas

- Além do padrão de objeto literal você pode criar objetos usando suas funções construtoras;
- A utilização se parece muito com a criação de um objeto Java.
- A sintaxe é semelhante, mas na verdade não há classes no JavaScript;
- Quando é invocado a função construtora `new`, acontece o seguinte:
 - Um objeto vazio é criado e referencia pela variável `this`, herdando o protótipo da função;
 - Propriedades e métodos são adicionados ao objetos referenciado por `this`;
 - O objeto recebido criado é retornado no final;

Objetos Notações [personalizados]

- Para criar um novo objeto personalizado em JavaScript, comece com uma função.

```
<script type="text/javascript">
  function MeuObjeto() {
    this.valor = 0;

    this.getValor = function() {
      return this.valor;
    };
    this.setValor = function(valor) {
      this.valor=valor;
    };
  };

  obj = new MeuObjeto();
  obj.setValor(10);
  console.log(obj.getValor());
</script>
```

Objetos [this e var]

- `this` atua como uma referência para o objeto pai;
- O que `this` faz é criar uma propriedade pública que é acessível de fora do objeto;
- Usar `var` em vez de `this` torna a propriedade ou método privado. Acessível internamente ao objeto;

Objetos [propriedades e métodos privados]

- O JavaScript não possui um sintaxe para denotar propriedades e métodos privados, protegidos ou públicos.
- Todos os membros de objeto são públicos;
- Você pode utilizar um closure e qualquer variável que seja parte do escopo do closure não é exposta;

Objetos [propriedades e métodos privados]

```
<script type="text/javascript">
function MeuObjeto() {
    var valor = 0; //propriedade privada

    this.metodoPublico = function(num) {
        console.info("Isto é um método PÚBLICO");
        metodoPrivado(); //acessando o método privado
    };

    var metodoPrivado = function() {
        console.info("Isto é um método PRIVADO");
    }
}

obj = new MeuObjeto();
obj.metodoPublico(); //funciona
obj.metodoPrivado(); //não funciona
console.log(obj.valor); //não funciona
</script>
```

Objetos Literais e Privacidade

- No casos dos objetos literais, podemos utilizar o closure criado por uma função imediata anônima adicional;

```
<script type="text/javascript">
    pessoa = {};

    (function() {
        var nome = "";
        pessoa = {
            getNome : function () { return nome; },
            setNome : function (n) { nome = n; }
        };
    })();

    pessoa.setNome("Luiz");
    console.log(pessoa.getNome()); //Luiz
    console.log(pessoa.nome); //undefined
</script>
```

Protótipos e Privacidade

- Uma desvantagem de membros privados é que quando usados com construtores, é que eles são recriados toda vez que o construtor é invocado;
- Para evitar a duplicação de esforços e economizar memória, você pode adicionar propriedades e métodos comuns à propriedade `prototype` do construtor;
- Desta forma, as partes comuns são compartilhadas entre todas as instâncias;
- Como a propriedade `prototype` é apenas um objeto, ela pode ser criada com objetos literais

Protótipos e Privacidade

```
Aluno = function (ra, nome, curso, n1, n2, n3, n4) {
    this.ra = ra;
    this.nome = nome;
    this.curso = curso;
    this.n1 = n1;
    this.n2 = n2;
    this.n3 = n3;
    this.n4 = n4;
};

Aluno.prototype.media = function () {
    let soma = this.n1 + this.n2 + this.n3 + this.n4;
    return soma / 4.0;
};

aluno1 = new Aluno("Luiz", 25, "Maringá", 5, 6, 7, 8);
console.log(aluno1);
console.log(aluno1.media());
```


Objetos [getters e setters]

- Uma das modificações que foram adicionadas ao ECMAScript é o conceito de getters e setters;

```
<script type="text/javascript">
var pessoa = {
  cidade : "",
  get localizacao () {
    return this.cidade;
  },
  set localizacao(val) {
    this.cidade = val;
  }
};
pessoa.localizacao = "Brasil";
console.log(pessoa.localizacao);
</script>
```

```
<script type="text/javascript">
function Pessoa () { //constructor }

Pessoa.prototype = {
  get localizacao () {
    return this.loc;
  },
  set localizacao (val){
    this.loc = val;
  }
};
var pessoa = new Pessoa();
pessoa.localizacao = "Brasil";
console.log(pessoa.localizacao);
</script>
```

Exercício de Objetos



- Criar um programa que faça a leitura dos dados de algumas pessoas:
- Cada pessoa deve ser um objeto com os seguintes atributos:
 - Nome
 - Idade
 - Salario
 - Estado
 - Cidade
- Os atributos deve ser público e ter métodos de getter e setter para recuperar e atribuir informações
- Criar um método show que apresente todos os dados de uma pessoa;

Membros Estáticos

- No JavaScript não há uma sintaxe para denotar membros estáticos;
- Mas você pode ter a mesma sintaxe de uma linguagem com Classes usando uma função construtora e adicionando propriedades a ela;

```
<script type="text/javascript">
Aplicativo = function(){
  this.name = "Facebook";
};

Aplicativo.getVersao = function(){
  return 1.2;
};
console.log(Aplicativo.getVersao());
;
</script>
```

Constantes de Objetos

- Não há constantes no JavaScript, alguns ambientes oferecem a declaração `const`;
- Como alternativa é comum usar convenção de nomeação e destacar as variáveis que não devem mudar usando apenas maiúscula;
- Essa convenção é adotada nos objetos JavaScript:
 - `Math.PI`; //3.141592....
 - `Math.SQRT2`; // 1.41421....
 - `Number.MAX_VALUE` //1.7976931....
- A mesma convenção pode ser aplicada a objetos literais;

Constantes de Objetos

- Se realmente quiser um valor imutável, você pode criar uma propriedade privada e fornecer um método getter, mas não um setter;

```
<script type="text/javascript">
  aplicativo = {
    nome : "Facebook",
    get VERSAO () {
      return 1.3;
    },
  };

  console.log(aplicativo.VERSAO);
</script>
```

Constantes de Objetos

- No ECMAScript 5 podemos utilizar o objeto `Object.defineProperty()` que define uma nova propriedade para um objeto;

```
<script type="text/javascript">
  aplicativo = {
    nome : "Facebook",
  };

  Object.defineProperty(aplicativo, "VERSAO", {
    value: 1.3,
    writable: false
  });

  console.log(aplicativo.VERSAO);
</script>
```

Encadeamento

- O padrão de encadeamento permite você chamar métodos em um objeto, um após o outro, sem atribuir os valores de retorno das operações anteriores a variáveis;
 - `Objeto.metodo1().metodo2().metodo3("param");`
- Ao criar métodos que não tenham valor de retorno significativo, você pode fazê-lo retornar `this`;
- `this` será a instancia do objeto que esta trabalhando;
- Benefícios:
 - Economizar na digitação e criar códigos melhores;
 - Dividir algumas funções e melhorar a manutenção;

Encadeamento

```
<script type="text/javascript">
  objeto = {
    valor : 0,
    incrementa : function () {
      this.valor++;
      return this;
    },
    adiciona : function(numero){
      this.valor+=numero;
      return this;
    },
    apresenta : function(){
      alert(this.valor);
    }
  };

  objeto.incrementa().incrementa().adiciona(10).apresenta(
);
</script>
```

Exercício de Encadeamento



- Criar uma calculadora que realize as 4 operações básicas com os métodos encadeados
- **Calculadora**
 - Operação
 - Calcular
 - Soma
 - Subtração
 - Multiplicação
 - Divisão
 - Apresentar

Method

- JavaScript pode ser confuso para quem já está acostumado com classes;
- Uma tentativa de tentar deixar o JavaScript mais próximo de orientações de classes é o método `method()`, introduzida por Douglas Corckford;
- Para criar métodos desta forma precisamos encadear a chamada ao construtor;
- O `method` aceita dois parâmetros:
 - O nome do novo método;
 - A implementação do método;

Method

```
<script type="text/javascript">

  if(typeof Function.prototype.method !== "Function"){
    Function.prototype.method = function (name, func) {
      this.prototype[name] = func;
      return this;
    };
  }

  Pessoa = function () {
    this.nome = "";
  }.method("getNome", function () {
    return this.nome;
  }).method("setNome", function (nome) {
    this.nome = nome;
    return this;
  });

  pessoa = new Pessoa();
  console.log(pessoa.setNome("Luiz Henrique").getNome());

</script>
```

Exercício de Method



- Utilizando o `method`, crie os getters e setter de um programa que faça a leitura dos dados de uma cidade:
 - Atributos
 - Nome
 - População
 - Estado
 - País
- Criar também os métodos:
 - Para realizar a leitura dos dados da cidade;
 - Para apresentar os dados de uma cidade;

Reutilização de Códigos

Funções

- Todas as funções do **javascript** possuem alguns métodos como `toString()`, `call()` e `apply()`.

Call()

O método **`function.call()`** permite você dizer em qual escopo uma função deve ser executada. Ou seja quem será o **`this`** da função.

```
<script type="text/javascript">
  var url = 'unicesumar.edu.br';
  var site = { url : 'globo.com' };
  function f(parametro) {
    console.log(this.url);
  }
  f.call(site, "Um texto qualquer");
</script>
```

Apply()

O método **`function.apply()`** tem o mesmo funcionamento do **`function.call()`** com uma diferença na forma dos parâmetros, o segundo parâmetro sempre deverá ser um **`array`**, contendo todos os parâmetros que serão enviados para a função.

```
<script type="text/javascript">
  var url = 'unicesumar.edu.br';
  var site = { url : 'globo.com' };
  function f(p1, p2) {
    console.log(this.url);
  }
  f.apply(site, ["Um texto qualquer", 10]);
</script>
```

Pode ser utilizado em funções que chamam funções

Herança Clássica

- Em JavaScript, como não existe classes, a noção de instancia não faz muito sentidos;
- Porém o JavaScript possui funções construtoras, e a sintaxe do operador new lembra muito o uso de classes;
- O objetivo de implementar a herança clássica é fazer objetos criados por uma função construtora (Filho) receber as propriedades que venham de outro construtor (Pai);

ECMAScript

- Na ECMAScript 5, o padrão de herança prototípica torna-se oficialmente parte da linguagem;
- Esse padrão é implementado por meio do método `Object.create()`;
- O método aceita dois parâmetros:
 - Um objeto que será herdado;
 - Um objeto com as propriedades particulares do objeto filho

ECMAScript

```
<script type="text/javascript">

  Animal = {
    idade : 0,
    nascer : function () {},
    morrer : function () {},
    respirar : function () {}
  };

  gato = Object.create(Animal, {
    miar : {
      value: function() { alert(1) }
    }
  });
  console.log(gato.miar());
  console.log(gato);
</script>
```

Como era feito anteriormente...

Padrão Clássico 1

- O método mais comum é criar um o objeto usando o construtor Pai e atribuir esse objeto ao protótipo do Filho;
- `inherit = herdar`

```
<script type="text/javascript">
    herdar = function(Filho, Pai){
        Filho.prototype = new Pai();
    }
</script>
```

Padrão Clássico 1

```
<script type="text/javascript">
    Animal = function(){}
    Animal.prototype.idade = 0;
    Animal.prototype.nascer = function (){};
    Animal.prototype.morrer = function (){};
    Animal.prototype.respirar = function (){};

    Gato = function(){}
    Gato.prototype.miar = function (){};

    herdar(Gato, Animal);

    gato = new Gato();
    console.log(gato);
</script>
```

Padrão Clássico 1

- Ao usar este padrão, você herda tanto as propriedades particulares (this) como propriedades e **métodos de protótipo**;
- Na maior parte das vezes você não vai querer as propriedades particulares, por que elas tendem a ser específicas a uma instância, não sendo reutilizáveis;

Padrão Clássico 2

- Este padrão resolve o problema de passar argumentos do filho;
- Mas isso pode ser um problema, pois, caso o protótipo do Pai seja alterado o filho será alterado também;

```
<script type="text/javascript">
    herdar = function(Filho, Pai){
        Filho.prototype = Pai.prototype;
    }
</script>
```

Padrão Clássico 3

- Este padrão resolver o problema, quebrando a ligação direta entre o protótipo do pai do filho;

```
<script type="text/javascript">
    herdar = function(Filho, Pai){
        var F = function(){};
        F.prototype = Pai.prototype;
        Filho.prototype = new F();
    }
</script>
```

Padrão Clássico 3

- Uma otimização é evitar a criação de um construtor temporário toda vez que necessite de herança, usando uma função imediata;

```
<script type="text/javascript">
    herdar = (function () {
        var F = function () {};
        return function (Filho, Pai) {
            F.prototype = Pai.prototype;
            Filho.prototype = new F();
            Filho.prototype.constructor = Filho;
        }
    })();
</script>
```

Padrão Clássico 3

- Executando o construtor da superclasse.

```
<script type="text/javascript">
    Gato = (function(__extends){
        herdar(Gato, __extends);
        function Gato(){
            __extends.call(this);
        }

        Gato.prototype.miar = function (){}

        return Gato;
    })(Animal);
</script>
```

Simplificando

- Executando o construtor da superclasse.

```
Pessoa = function (nome, idade, cidade) {
    this.nome = nome;
    this.idade = idade;
    this.cidade = cidade;
};

Pessoa.prototype.apresentar = function () {
    console.log("Nome: " + this.nome);
    console.log("Idade: " + this.idade);
    console.log("Cidade " + this.cidade);
};

Professor = function (nome, idade, cidade, curso) {
    Pessoa.call(this, nome, idade, cidade);
    this.curso = curso;
};

Professor.prototype = Object.create(Pessoa.prototype);
Professor.prototype.constructor = Professor;
```

Exercício de Herança



- Criar um programa que tenha 3 “classes”:
 - **Pessoa:** Telefone, Endereço, Estado, Cidade;
 - **Física:** Nome, CPF;
 - **Jurídica:** CNPJ, Razão Social, Nome Fantasia;
- Criar os métodos getter e setter para os atributos
- Realizar a entrada de dados para a pessoa conforme a escolha do usuário.
 - Usuário deve escolher o tipo de pessoa: Física ou Jurídica

Herança prototípica

- Neste padrão, não há classes envolvidas;
- Os objetos herdam de outros;

```
<script type="text/javascript">
  herdar = function(o){
    function F(){};
    F.prototype = o;
    return new F();
  };

  Animal = {
    idade : 0,
    nascer : function () {},
    morrer : function () {},
    respirar : function () {}
  };

  gato = herdar(Animal);
  gato.miar = function () {};
  console.log(gato);
</script>
```

Padrões de Projeto

Singleton

- A ideia do padrão Singleton é ter apenas uma instância de uma classe específica;
- No JavaScript não existe classes para criar um novo objeto;
- Quando criamos um objeto literal no JavaScript não existe nenhum outro objeto igual a ele e o novo objeto já um Singleton;
- Mesmo criando objetos com atributos iguais os objetos serão diferentes;
- Então sempre que criar um objeto literal, pode-se dizer que é um Singleton;

Singleton

- Usando o **new** você pode:
 - usar uma variável global para armazenar a instância;
 - Esta opção **não é recomendada** pois os princípios de globais são ruins e qualquer um pode sobrescrever essa variável, mesmo por acidente;
 - alocá-lo em um propriedade estática do construtor.
 - Funções no Javascript são objetos, então ela pode ter propriedades;
 - encapsular a instância em um closure;
 - Desta forma você mantém a instância privada e indisponível para modificações fora de seu construtor;

Singleton [instância em uma propriedade estática]

```
<script type="text/javascript">

    function Singleton(){
        if(typeof Singleton.instancia === "object"){
            return Singleton.instancia ;
        }
        this.usuario = "Luiz";
        Singleton.instancia = this;
    };

    singleton1 = new Singleton();
    singleton2 = new Singleton();
    console.log(singleton1 === singleton2);

</script>
```

Singleton [instância em um closure]

```
<script type="text/javascript">

    Singleton = function(){

        var instancia = this;
        this.usuario = "Luiz";

        Singleton = function(){
            return instancia;
        }
    };

    singleton1 = new Singleton();
    singleton2 = new Singleton();
    console.log(singleton1 === singleton2);

</script>
```

Singleton [instância em um closure]

- No exemplo anterior podemos ter alguns problemas como mudanças no protótipo, uma solução alternativa é utilizar função imediata:

```
<script type="text/javascript">

    var Singleton;

    (function(){
        var instancia;

        Singleton = function Singleton(){
            if(instancia)
                return instancia;

            instancia = this;
            this.usuario = "Luiz";
        };

        Singleton.prototype = {};
    })();

    singleton1 = new Singleton();
    singleton2 = new Singleton();
    console.log(singleton1 === singleton2);

</script>
```

Iterador (Iterator)

- Neste padrão você tem um objeto contendo algum tipo de dado agregado.
- O consumidor do objeto não precisa saber como você estrutura seus dados;
- Ele precisa saber como trabalhar com os elementos individuais;
- O objeto precisa fornecer alguns métodos:
 - `next()`: para chamar o próximo elemento em sequência;
 - `hasNext()`: para determinar se chegou a fim de seus dados;
 - `rewind()`: para redefinir o ponteiro;
 - `current()`: para retornar o elemento atual;

Iterador (Iterator)

```
<script type="text/javascript">

    lista = (function(){
        var indice = 0, numeros = [1,2,3,4,5];

        return {
            next : function(){
                if(!this.hasNext()) return null;

                var elemento = numeros[indice];
                indice++;
                return elemento;
            },
            hasNext : function(){
                return indice < numeros.length;
            }
        };
    })();

    while(lista.hasNext()){
        console.log(lista.next());
    }

</script>
```

Exercício de Iterador



- Utilizando o padrão de iterador, criar um programa com um iterador que armazene pessoas:
 - Nome;
 - Idade;
 - Cidade;
 - Estado;
- Criar os métodos:
 - `add(pessoa)`;
 - `next()`;
 - `hasNext()`;
 - `rewind()`;
 - `current()`;

Fábrica (Factory)

- O objetivo do fábrica é criar objetos;
- Ela costuma ser implementada em uma classe com um método estático com o seguintes propósitos:
 - Realizar operações repetidas quando se definem objetos semelhantes;
 - Oferecer uma forma ao cliente de a fábrica criar objetos sem conhecer o tipo específico;

Fábrica (Factory)

```
<script type="text/javascript">

function Carro() {}

Carro.prototype.dirigir = function(){
    return "Dirigindo com um carro de " + this.portas + " portas!";
}

Carro.factory = function(tipo){
    var tipo;
    if(typeof Carro[tipo].prototype.dirigir !== "function")
        Carro[tipo].prototype = new Carro();
    return new Carro[tipo]();
}

Carro.Compacto = function(){ this.portas = 4; };
Carro.Conversivel = function(){ this.portas = 2; };

carroCompacto = Carro.factory("Compacto");
console.log(carroCompacto.dirigir());

</script>
```

Decorador (Decorator)

- No padrão decorar, funcionalidades extras podem ser adicionadas dinamicamente a um objeto em tempo de execução.
- No JavaScript os objetos são mutáveis e este processo não é um problema;

Fachada (Façade)

- É um padrão simples: ela fornece apenas uma interface alternativa para um objeto;
- É uma boa prática para manter seus métodos curtos e não forçá-los a realizar muito trabalho;

```
<script type="text/javascript">

evento = {
    stop : function(){
        funcao1();
        funcao2();
    }
}

</script>
```

ER – Expressões Regulares

isObject = true

Expressões Regulares

- Basicamente servem para você dizer algo abrangente de forma específica. Definindo seu padrão de busca.
- É uma composição de símbolos, caracteres com funções especiais, que agrupados forma uma expressão.
- Um exemplo rápido, `[rqp]ato` pode casar com rato, gato e pato;

Expressões Regulares - Metacaracteres

- Metacaracteres são símbolos que tem sua função específica, que pular dependendo do contexto no qual está inserido.
- Podemos agrega-los uns aos outros, combinando suas funções.

Metacaractere	Nome	Metacaractere	Nome
.	Ponto	^	Circunflexo
[]	Lista	\$	Cifração
[^]	Lista Negada	\b	Borda
?	Opcional	\	Escape
*	Asterisco		Ou
+	Mais	()	Grupo
{}	Chaves	\1	Retrovisor

Expressões Regulares - Metacaracteres

- Representantes

Metacaractere	Nome	Função
.	Ponto	Um caractere qualquer
[]	Lista	Lista de caracteres permitidos
[^]	Lista Negada	Lista de caracteres proibidos

- Quantificadores

Metacaractere	Nome	Função
?	Opcional	Zero ou um
*	Asterisco	Zero um ou mais
+	Mais	Um ou mais
{n,m}	Chaves	De n até m

Expressões Regulares - Metacaracteres

- Âncoras

Metacaractere	Nome	Função
^	Circunflexo	Início da linha
\$	Cifração	Fim da linha
\b	Borda	Início ou fim da palavra

- Outros

Metacaractere	Nome	Função
\c	Escape	Torna literal o caractere c
	Ou	Ou um ou outro
(...)	Grupo	Delimita um grupo
\1...\9	Retrovisor	Texto casado nos grupos 1...9

Expressões Regulares em JavaScript

- Há um pouco de confusão com relação aos métodos que usam expressões regulares, pois algumas estão no objeto RegExp e outros estão no String.

Função	Ação
RegExp .test()	Testa se casou ou não true / false
RegExp .exec()	Retorna array com trecho casado ou null
String .search()	Testa se casou e retorna o index ou -1
String .match()	Retorna um array com o trecho casado ou null
String .replace()	Faz substituições, retorna a string
String .split()	Faz divisões, retorna array

Expressões Regulares em JavaScript

- Validando um formato de entrada de texto;

Exemplo de validação de telefone

```
<script type="text/javascript">
  er = new RegExp("....-....");
  console.log(er.test("3027-6360"));
</script>
```

Exemplo de validação de número

```
<script type="text/javascript">
  er = new RegExp("^ [0-9]+ $");
  console.log(er.test("123456"));
</script>
```

Expressões Regulares em JavaScript

- Utilizando as funções de String

.search

```
<script type="text/javascript">
  texto = "JavaScript JavaScript";
  console.log(texto.search("^Java")); //0
  console.log(texto.search("JavaScript$")); //11
</script>
```

.match

```
<script type="text/javascript">
  texto = "31/12/2014";
  resultado = texto.match("^(..)/(..)/(....)$");
  console.log(resultado);
  //[ "31/12/2014", "31", "12", "2014", index: 0, input: "31/12/2014" ]
</script>
```

Expressões Regulares em JavaScript

- Utilizando as funções de String

.replace

```
<script type="text/javascript">
  texto = "JavaScript";
  resultado = texto.replace(/[a-z]/, "*");
  console.log(resultado); //J*vaScript
</script>
```

.split

```
<script type="text/javascript">
  texto = "JavaScript que tal?";
  resultado = texto.split(/\s+/);
  console.log(resultado); //["JavaScript", "que", "tal?"]
</script>
```

Expressões Regulares [não seja afobado]

- Dê um passo após o outro:

hh:mm

- ...
- `[0-9]{2}:[0-9]{2}`
- `[012][0-9]:[0-9]{2}`
- `[012][0-9]:[0-5][0-9]`
- `([01][0-9])|2[0-3]:[0-5][0-9]`

Expressões Regulares [não seja afobado]

- Dê um passo após o outro:

dd/mm/aaaa

- ...
- `[0-9]{2}/[0-9]{2}/[0-9]{4}`
- `[0123][0-9]/[01][0-9]/[0-9]{3}`
- `[0123][0-9]/[01][0-9]/[12][0-9]{3}`
- `([012][0-9]|3[01])/[01][0-9]/[12][0-9]{3}`
- `([012][0-9]|3[01])/([01-9]|1[012])/[12][0-9]{3}`
- `([01-9]|12[0-9]|3[01])/([01-9]|1[012])/[12][0-9]{3}`

Expressões Regulares

- <http://www.regular-expressions.info/>

Exercício de ER



- Utilizando o comando `prompt` e o laço de repetição `do-while` solicite ao usuário digitar uma data. Faça a solicitação enquanto o usuário não digitar a data em formato correto.

- ER:

• `^(0[1-9]|12[0-9]|3[01])/([01-9]|1[012])/([12][0-9]{3})$`

Tratamento de Erros

`try, catch e finally`

Tratamento de Erros

- O JavaScript possui um mecanismo para lidar com exceções;
- No JavaScript 1.5 o `try...catch...finally` foi incorporado a linguagem;
 - **try**: delimita um bloco de código que fica dentro do mecanismo de manipulação de erros;
 - **catch**: fica no final do bloco e captura qualquer exceção e permite a você processá-la como quiser;
 - **finally**: não é obrigatório, mas é necessário se alguma operação tiver que ser executado caso a exceção ocorra ou não;

Tratamento de Erros

- Seis tipos de erros são implementados no JavaScript 1.5:
 - **EvalError**: Gerado por eval quando usado de forma incorreta;
 - **RangeError**: Gerado quando o valor numérico excede sua faixa
 - **ReferenceError**: Gerado quando uma referência inválida é usada;
 - **SyntaxError**: Usado com sintaxe inválida;
 - **TypeError**: Gerado quando uma variável não for do tipo esperado;
 - **URIError**: Gerado quando encodeURI() ou decodeURI() é usado incorretamente;

Tratamento de Erros

- Usar o `instanceof` ao capturar o erro permite saber se o erro é de um desses tipos internos;

```
<script type="text/javascript">

    try{
        clientes = null;
        alert(clientes[1]);
    }catch(e){
        if(e instanceof TypeError)
            alert("Erro do Tipo" + e.message);
    }

</script>
```

Tratamento de Erros

- Se você tiver qualquer funcionalidade que precise ser processada independente de sucesso ou fracasso, pode incluí-la no bloco finally;

```
<script type="text/javascript">

    try{
        clientes = [];
        alert(clientes[1]);
    }catch(e){
        if(e instanceof TypeError)
            alert("Erro do Tipo" + e.message);
    }finally{
        clientes = null;
    }
    console.log(clientes);

</script>
```

Tratamento de Erros

- Você pode criar erro personalizados usando o objeto Error;

```
<script type="text/javascript">

    comErro = function() {
        var error = new Error('Um erro muito grave para ser tratado');
        error.name = 'CustomError';
        throw error;
    };

    try {
        comErro();
    }
    catch(e) {
        console.log(e.name + ' - ' + e.message);
    }

</script>
```

Cookie

Cookie

- O nome original para um cookie veio do termo magic cookie – um símbolo passado entre dois programas;
- Embora acessível pelo JavaScript, cookeis não são baseados em JavaScript;
- Eles são um mecanismo do servidor HTTP, são acessíveis pelo cliente e pelo servidor;
- Cookies são pequenos pares de chave/valor associados a uma data de expiração e um domínio específico;
- Cookie são acessíveis pelo objeto document;

Cookie

- Para atribuir um cookie, apenas atribua ao valor de `document.cookie` um string com o seguinte formato:
`document.cookie = "cookieName=value; expires=date; path=path";`
 - cookieName:** O nome do cookie e seu valor é você que define;
 - expires:** Data de expiração em formato GMT (UTC), criar um objeto `Date` e depois usar `toGMTString`
 - path:** O caminho do cookie, é comparado com a solicitação da página e, se não forem sincronizados, o cookie não pode ser acessado ou configurado. Isso evita que outros sites acessem quaisquer e todos cookies;

Atenção: Para assegurar resultados consistentes use apenas tipos primitivos que sejam convertidos totalmente em string;

GMT - Greenwich Mean Time
UTC - Universal Time Coordinated

Cookie [gravando]

- Usa-se o `encodeURIComponent` para colocar caracteres de escape em quaisquer caracteres especiais que possam fazer parte do valor do cookie.

```
<script type="text/javascript">

function setCookie(chave, valor){
    var data = new Date(2015,07,11,23,30,00);
    document.cookie = chave + "=" + encodeURIComponent(valor) +
        "; expires=" + data.toGMTString() +
        "; path="/;
}

setCookie("UNICESUMAR", "2015");

</script>
```

Atenção: O Chrome ignora cookies locais;

Cookie [leitura]

- Pegar o cookie não é tão fácil, por que todos os cookies são concatenados em uma string por ponto e vírgula no objeto cookie.

```
<script type="text/javascript">

function getCookie(key){
    var index = document.cookie.indexOf(key+"=");

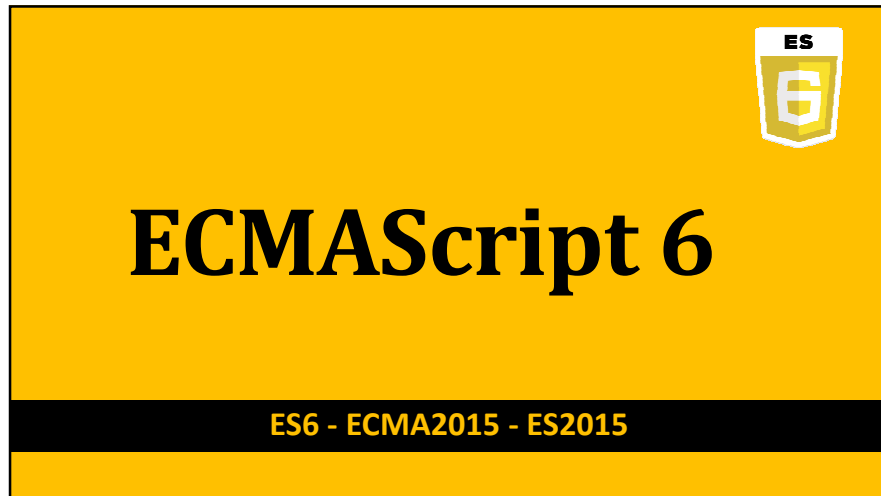
    if(index >= 0){
        var str = document.cookie.substring(index, document.cookie.length);
        var ultimo = str.indexOf(";");
        if(ultimo < 0) ultimo = str.length;
        str = str.substring(0,ultimo).split("=");
        return decodeURI(str[1]);
    }
    return null;
}

</script>
```

Exercício de Cookie



- Criar um programa que solicite os dados abaixo para o usuário:
 - Nome
 - Idade
 - CPF
 - RG
- Armazene as informações no Cookie;
- Ao abrir a página novamente apresente os dados registrados.



EcmaScript 6

- Versão lançada em junho de 2015;
- A partir da versão ES6, será adotado o versionamento por ano e não mais por número. É por isso que em muitos lugares encontramos o ES6 como ECMA2015 ou ES2015. São a mesma coisa.
- Esta nova atitude se deve ao fato da pretensão de termos uma nova atualização da especificação a cada ano.

O que mudou?

- Vamos conhecer algumas funcionalidades do ES6

Métodos auxiliares para Array

- Existe uma infinidade de métodos auxiliares para array, vamos conhecer os principais e mais utilizados.

forEach

O *forEach* é uma mão na roda para quando precisamos passar por todos os elementos de dentro de um Array

```
var nomes = ['maria', 'josé', 'joão'];
nomes.forEach(function(nome) {
  console.log(nome);
});

var canais = ["Globo", "Sbt", "Record"];
canais.forEach(function(canal) {
  canais.push("RedeTV"); // este item será ignorado
  console.log(canal);
})
console.log(canais); //['Globo', 'Sbt', 'Record', 'RedeTV', 'RedeTV', 'RedeTV']
```

map

O método *map* é muito útil quando precisamos não somente passar por todos os elementos de um Array.

```
var numeros = [1,2,3];
var dobro = numeros.map(function(numero) {
  return numero * 2;
});

console.log(numeros); // [1,2,3]
console.log(dobro); // [2,4,6]
```

filter

Como o próprio nome já pode induzir, este método é deve ser utilizado quando temos a necessidade de filtrar nossa lista de acordo com algum critério.

```
var alunos = [
  {nome:'joão', idade:15},
  {nome:'josé', idade:18},
  {nome:'maria', idade:20}
];

var alunosDeMaior = alunos.filter(function(aluno) {
  return aluno.idade >= 18;
});

console.log(alunosDeMaior);
// [{nome:'josé', idade:18}, {nome:'maria', idade:20}]
```

find

Esta função auxiliar é particularmente interessante quando o objetivo é encontrar um item específico dentro de um Array

```
var alunos = [
  {nome:'joão', idade:15},
  {nome:'josé', idade:18},
  {nome:'maria', idade:20}
];

var aluno = alunos.find(function(aluno) {
  return aluno.nome === 'josé';
});

console.log(aluno); // {"nome":"josé"}
```

every

Esta é uma função auxiliar bem interessante. Ao contrário das outras que vimos até então, esta não retorna uma cópia do Array, mas sim um valor booleano.

```
var alunos = [
  {nome:'joão', idade:15},
  {nome:'josé', idade:18},
  {nome:'maria', idade:20}
];

var todosAlunosDeMaior = alunos.every(function(aluno) {
  return aluno.idade > 18;
});

console.log(todosAlunosDeMaior); // true
```

some

Se a tarefa é validar se, pelo menos, um dos elementos de um Array satisfaz uma dada condição, o *some* é o método perfeito para o trabalho.

```
var pesoDasMalas = [12,32,21,29];

var temMalaAcimaDoPeso = pesoDasMalas.some(function(pesoDaMala) {
  return pesoDaMala > 30;
});

console.log(temMalaAcimaDoPeso); // true
```

reduce

A ideia por trás da função *reduce* é pegar todos os valores de um Array e condensá-los em um único.

Reduce aceita dois parâmetros: função de iteração e valor inicial.

```
var numeros= [1,2,3,4,5];

var soma=0;
soma = numeros.reduce(function(soma, numero) {
  return soma + numero;
}, 0);

console.log(soma); // 15
```

Iteração com Iteradores

- O ES6 introduziu um novo mecanismo para esta tarefa: iteração. O conceito de iteração não é novo e já é utilizado em muitas linguagens de programação como o Java, Python e C#, mas somente agora foi padronizado no JavaScript.
- O seu objetivo é prover uma forma de sequencialmente acessar os elementos de um iterável sem expor sua representação interna, retirando a responsabilidade dele de saber como acessar e caminhar sobre sua estrutura. Vamos entender exatamente o que isso quer dizer

next e done

Vamos utilizar iteradores e iteráveis. Assumiremos que todos os bruxos estão em um Array chamado *bruxos* e que recebemos essa coleção para iterá-la. Podemos obter o iterador da coleção de bruxos e usar a propriedade *done* em um laço de repetição *do...while*

```
bruxos = ["Harry Potter", "Hermione Granger", "Rony Weasley"]
var iterador = bruxos[Symbol.iterator]();
var proximo = iterador.next();

do {
  var bruxo = proximo.value;
  console.log(bruxo);
  proximo = iterador.next();
} while (!proximo.done);
```

Estruturas Map e Weakmap

- Mapas são estruturas de dados em que é possível associar uma chave a um valor, como em um dicionário, onde há um significado correspondente para cada palavra. Cada uma das chaves é única e possui apenas um valor associado.
- Essas novas estruturas permitem:
 - Adicionar elemento pelo par (chave e valor);
 - Remover elementos pela chave;
 - Acessar elementos dada um chave
 - Pesquisar elementos;

map

Em um Map do Javascript, qualquer valor (tanto objetos, funções ou valor primitivos), podem ser usados como chave ou valor.

```
var map = new Map();
function funcao(){};
var objeto = {};

map.set("string", "sou uma string");
map.set(objeto, "sou um objeto");
map.set(funcao, "sou uma função");

console.log(map.get("string")); // sou uma string
console.log(map.get(objeto)); // sou um objeto
console.log(map.get(funcao)); // sou uma função
```

map

Outras funcionalidades

```
console.log("tamanho: " + map.size); // tamanho: 3

console.log(map.has("string")); // true
console.log(map.has("abc")); // false

map.delete("string");
console.log(map.has("string")); // false

map.clear();
console.log("tamanho: " + map.size); // tamanho: 0
```

map

Iteração

```
var mapa = new Map();
mapa.set('um', 1);
mapa.set('dois', 2);
mapa.set('três', 3);

for(var chave of mapa.keys()){
  console.log(chave); // um dois três
}

for(var valor of mapa.values()){
  console.log(valor); // 1 2 3
}

for(var entrada of mapa.entries()){
  console.log(entrada); // ['um', 1] ['dois', 2] ['três', 3]
}
```

weakmap

Um WeakMap é uma coleção de pares de chave/valor na qual as chaves só podem ser objetos. As referências do objetos nas chave são fracamente mantidas.

```
var weakMap = new WeakMap();

var elemento1 = window;
var elemento2 = document.querySelector('body');

weakMap.set(elemento1, 'sou o elemento1');
weakMap.set(elemento2, 'sou o elemento2');
```

set

O Set é uma estrutura de dados que nos permite ter listas com valores que nunca se duplicam e que mantém a ordem de inserção dos seus itens.

```
var set = new Set();

set.add(2);
set.add(1);
set.add(2);

for (const valor of set) {
  console.log(valor); // 2, 1
}
```

let

Podemos considerar o let como o verdadeiro substituto do var. Idealmente ele deve ser usado para declarar variáveis que esperamos que mudem com o valor e com o tempo/execução do programa.

```
let resultado = 0;
resultado = soma(2,3);
console.log(resultado); // 5
```

Arrow Function

- Uma expressão arrow function possui uma sintaxe mais curta quando comparada a uma expressão de função;

```
var textos = ["Luiz Henrique", "Unicesumar"];
var retorno = textos.map(function (texto) {
  return texto.length;
});
```

```
var textos = ["Luiz Henrique", "Unicesumar"];
retorno = textos.map(texto => texto.length);
```

Fetch

- A API Fetch fornece uma interface JavaScript para acessar e manipular partes do pipeline HTTP, tais como os pedidos e respostas. Ela também fornece o método global fetch() que fornece uma maneira fácil e lógica para buscar recursos de forma assíncrona através da rede.

```
fetch("http://www.mocky.io/v2/5d783a6b320006d7f9241a4")
  .then(function (response) {
    response.json().then(function (data) {
      console.log(data);
    });
  })
  .catch(function (error) {
    console.log(error);
  });
```

Executar em um servidor Apache

Modelagem de classe

- Uma das maiores dificuldades de estudar Javascript é entender a maneira com a herança funciona, pois ela não funciona da maneira como aprendemos em Orientação Objetos, ou em linguagens como Java ou C#.
- O Javascript possui herança por prototipagem.
- A ideia por trás da funcionalidade de classes do ES6 é que possamos criar uma hierarquia de objetos que, por debaixo dos panos ainda funciona em cima de herança por prototipagem.

Modelagem de classe: Antes

- Vamos fazer um exemplo sem Classes e depois o mesmo exemplo usando Classes.

```
function Carro(modelo, chassi, qtdePortas) {
  this.modelo = modelo;
  this.chassi = chassi;
  this.qtdePortas = qtdePortas;
}

Carro.prototype.andar = function () {
  console.log("vrum vrum");
};

prototipo = new Carro('protótipo', '1234567890', 2);
console.log(prototipo.modelo);
console.log(prototipo.chassi);
console.log(prototipo.qtdePortas);
prototipo.andar();
```

Modelagem de classe: Agora

- Vamos fazer um exemplo sem Classes e depois o mesmo exemplo usando Classes.

```
class Carro {
  constructor(modelo, chassi, qtdePortas) {
    this.modelo = modelo;
    this.chassi = chassi;
    this.qtdePortas = qtdePortas;
  }

  andar() {
    console.log("vrum vrum");
  }
}

prototipo = new Carro('protótipo', '1234567890', 2);
console.log(prototipo.modelo);
```

Métodos estáticos

- A nova sintaxe de classes do ES6 nos permite declarar métodos estáticos na definição de nossas classes.

```
class Casa {
  static abrirPorta(){
    console.log('abrindo porta');
  }
}

Casa.abrirPorta(); // abrindo porta
```

Atributos privados

Atualmente, a sintaxe de classe do JavaScript ES6 não dá suporte a propriedades privadas. Mas como vimos no capítulo de Mapas, podemos utilizar a estrutura de WeakMap para nos ajudar

```
const propriedades = new WeakMap();

class VideoGame {
  constructor(nome, controles, saida, midia) {
    propriedades.set(this, {
      nome, controles, saida, midia
    });
  }
}

const xbox360 = new VideoGame("XBOX360", 4, 'hdmi', 'dvd');
console.log(xbox360.nome); // undefined
```

Atributos privados

Recuperando propriedades

```
const propriedades = new WeakMap();

class VideoGame {
  constructor(nome, controles, saida, midia) {...}

  recuperaPropriedade(propriedade) {
    return propriedades.get(this)[propriedade];
  }
}

const xbox360 = new VideoGame("XBOX360", 4, 'hdmi', 'dvd');
console.log(xbox360.nome); // undefined
console.log(xbox360.recuperaPropriedade('nome')); // XBOX360
```


Atributos privados

Outra maneira

```
class VideoGame {
  constructor() {
    this._nome = '';
  }
  get nome() {
    return this._nome;
  }
  set nome(str) {
    this._nome = str;
  }
}

const xbox360 = new VideoGame();
xbox360.nome = 'XBOX360';
console.log(xbox360.nome);
```

Herança de Objetos - Array

```
class Exemplo extends Array {
  constructor() {
    super();
  }

  top() {
    return this[this.length - 1];
  }
}

var exemplo = new Exemplo();
exemplo.push('Linha 1');
exemplo.push('Linha 2');
exemplo.push('Linha 3');
console.log(exemplo.top());
console.log(exemplo.length);
```

Herança de Objetos - Date

```
class MinhaData extends Date {
  constructor() {
    super();
  }

  getDataFormatada() {
    var meses = ['Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun',
      'Jul', 'Ago', 'Set', 'Out', 'Nov', 'Dez'];
    return this.getDate() + '/' + meses[this.getMonth()] +
      '/' + this.getFullYear();
  }
}

var data = new MinhaData();
console.log(data.getTime());
console.log(data.getDataFormatada());
```

Proxy

Proxy é um objeto que representa um outro objeto. Vamos ver o que isso significa na prática.

Imagine que estamos na etapa de implementação de um sistema e estamos tendo certas dificuldades para trabalhar no desenvolvimento de uma funcionalidade. O programa está dando problemas em certa etapa e não sabemos exatamente o porquê. Para facilitar o nosso debug, vamos criar um mecanismo de log nos objetos.

Proxy

```
class Usuario {
  constructor(login, senha) {
    this.login = login;
    this.senha = senha;
  }
}

const usuario = new Usuario('SuperJS', '123');
console.log(usuario.login); // SuperJS
console.log(usuario.senha); // 123

const proxy = new Proxy(usuario, {
  get(alvo, propriedade) {
    console.log(propriedade + ' foi solicitada!');
  }
});

console.log(proxy.login); //login foi solicitada // SuperJS
console.log(proxy.senha); //senha foi solicitada // 123
```

Proxy

Podemos utilizar o proxy para fazer validações de inputs

```
class Usuario {}
const usuario = new Usuario();

const proxy = new Proxy(usuario, {
  set(alvo, propriedade, valor) {
    if (propriedade == "idade") {
      if (!Number.isInteger(valor)) {
        throw new TypeError("A idade não é número");
      }
    }
  }
});

proxy.idade = 10;
proxy.idade = "aaaa";
```

ES6

- O ES6 trouxe para a linguagem JavaScript o que ela precisava para finalmente ser levada a sério como tecnologia.
- A especificação permitiu que a linguagem se tornasse mais próxima das demais linguagens focadas em Orientação a Objetos
- Por debaixo dos panos, tudo continua como antes, mas temos uma linguagem mais flexível, limpa, objetiva e convidativa para novos programadores.

Exemplo

- Exemplo Cadastro de Cidades

Exercício Desafio

Carrinho de compras

- Criar um programa que apresente o seguinte menu para o usuário com o `prompt`:
 - **1 – Escolher produto:** Apresenta os produtos disponíveis em um `prompt` e solicita o código do produto. Adicionar o código do produto em um array de pedido;
 - **2 – Ver produtos do carrinho:** Apresenta um `alert` com os produtos que o usuário já escolheu;
 - **3 – Remover produto do carrinho:** Solicita o código do produto para remover e remover o código informado;
 - **4 – Finalizar compra:** Apresenta o valor total da compra e os produtos escolhidos e encerrar o looping. Solicite a confirmação do usuário;

