



FERRAMENTAS DE DESENVOLVIMENTO WEB

AULA 3

Prof. Yanko Costa

CONVERSA INICIAL

Nesta aula, trataremos da linguagem de programação Javascript: partindo de sua sintaxe básica, formas de controle de fluxo e tipos de dados compostos. Verificaremos, com exemplos práticos, como o Javascript funciona para, mais adiante, poder integrá-lo a tecnologias e conceitos vistos em aulas passadas (HTML e CSS) e como essa linguagem poderá ser utilizada no contexto de desenvolvimento de aplicações web. Para podermos trabalhar com todas essas tecnologias em conjunto, é importante que você acompanhe e teste os exemplos mostrados na presente aula. Assim, ficará mais tranquilo quando for necessária a incorporação de mais informações e tecnologias nas aulas seguintes.

O Javascript é um importante componente de sistemas SPA, pois essa linguagem é executada no navegador e permite que o processamento no lado cliente aconteça conforme verificaremos a seguir.

TEMA 1 – JAVASCRIPT

Muitos aplicativos são utilizados por usuários no mundo todo para executar tarefas administrativas. À medida que o usuário vai percebendo que existem tarefas que são repetitivas dentro do aplicativo, vem o anseio de deixar para o próprio computador a execução dessas tarefas. É neste ponto que os desenvolvedores desses aplicativos percebem a possibilidade de aumentar a automação interna de seu software, incluindo nele uma linguagem de *script*.

A maioria das linguagens de *script* tem como característica ser interpretada, ter uma sintaxe mais flexível, não ser tipada[1] e, quando adaptadas ao ambiente interno de uma aplicação, permitir o acesso aos dados de forma segura por meio de APIs (Spinellis, 2005).

Por exemplo: o que aplicativos tão diversos como Word (editor de texto), Blender (modelador 3D), Firefox (navegador de Internet), Autocad (projetos CAD) têm em comum? Todos incluíram uma linguagem para automatizar algumas tarefas internas. Não importa qual linguagem, desde que ela possa acessar partes do software nas quais ela foi inserida para poder repetir, alterar ou apagar alguma informação de forma automatizada, podendo, inclusive, estender suas funcionalidades.

Em nosso contexto, os navegadores também identificaram essa necessidade e, em 1995, a empresa Netscape inseriu, dentro de seu software, uma linguagem criada por ela mesma chamada de Javascript. Essa linguagem, criada por Brendan Eich enquanto era funcionário da Netscape Corporation, foi padronizada internacionalmente pela ECMA International[2] em 1996, e seu nome oficial passa a ser **ECMAScript**, mas ainda é conhecida popularmente como Javascript. Nas primeiras revisões da linguagem ECMAScript, usou-se a referência ES1, ES2... até ES6. A partir do ES6 (lançada em 2015), as versões passaram a se chamar *ECMAScript 2015*, *ECMAScript 2016*, usando o ano de lançamento ao final.

Atualmente, a linguagem Javascript está incorporada em praticamente todos os principais navegadores da Internet (Chrome, Edge, Opera, Safari), e com a utilização de técnicas de compilação *just-in-time* e a evolução da otimização de compilação, o desempenho do Javascript tem aumentado e tornado a linguagem uma competidora viável para aplicações em servidores e desktop (Selakovic; Pradel, 2016).

O Javascript é uma linguagem de *script* não tipada e interpretada, portanto, facilita a construção de códigos de forma incremental: pequenos testes e protótipos podem ser criados e testados sem a necessidade do ciclo de compilação. A falta de tipagem permite uma curva de aprendizado mais curta e a construção de protótipos mais enxutos, mas a manutenção posterior de grandes códigos desenvolvidos com esse mecanismo pode ser mais difícil (Spinellis, 2005).

1.1 MULTIPARADIGMA

Um programa normalmente está resolvendo um problema e, para isso, o desenvolvedor usará alguma estratégia para essa resolução. As linguagens de programação também são criadas com base em alguma forma de abordar a situação a ser resolvida, e essa abordagem é chamada de ***paradigma***.

Nem sempre uma linguagem se limita a apenas um paradigma e, por isso, podemos ter diferentes programas em uma mesma linguagem usando um paradigma diferente ou diferentes paradigmas sendo utilizados em trechos de um mesmo programa. A linguagem Javascript é uma linguagem **orientada a objetos**, mas pode utilizar outros paradigmas, além da orientação a objetos para a construção de projetos, como o **funcional** e o **imperativo**. Por esse motivo, o Javascript é considerado **multiparadigma** (Developer Mozilla, 2020).

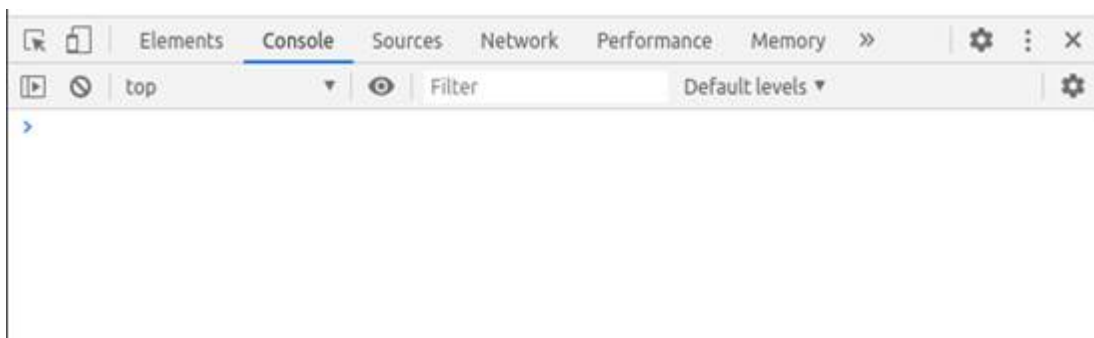
1.2 AMBIENTE DE TESTE

Para iniciar a apresentação da sintaxe do Javascript, vamos proceder a montagem de um ambiente de teste que poderá auxiliar nos primeiros contatos com a linguagem. Para esse ambiente, bastará utilizar um dos navegadores que possuem um console Javascript, como o Chrome ou Firefox.

No próximo quadro, temos em (a) o console do navegador Chrome e em (b) o console do navegador Firefox. Nesse espaço, poderão ser digitadas expressões em Javascript e poderemos ver o resultado dessas expressões sem a necessidade de um editor de código.

Quadro 1 – Console dos navegadores

(a) Chrome



b) Firefox



Em muitos casos, mesmo que o desenvolvedor esteja utilizando um editor de códigos para o desenvolvimento de seu projeto, a utilização do console pode ser útil para testar novas opções ou verificar erros em programas.

Na Figura 1, temos um exemplo de utilização do console do navegador Firefox, mas o mesmo código pode ser feito no navegador Chrome.

Figura 1 – Exemplo de execução de Javascript em console



No exemplo mostrado na Figura 1, é feita uma declaração da variável "custo" com um valor "10" (não se preocupe com a mensagem "undefined" que aparecerá após algumas expressões, pois ela apenas indica que o último comando foi executado corretamente, mas tem retorno indefinido). Essa variável ficará em memória e será utilizada na linha seguinte, na expressão "console.log" ("O novo valor é:", custo * 100), que informa ao navegador que será impresso na área de console pelo comando "console.log" a string **O novo valor é:** juntamente com o resultado de uma operação matemática "custo * 100" que irá multiplicar o conteúdo da variável por 100.

Várias expressões em Javascript poderão ser inseridas diretamente no console, e iremos utilizar essa alternativa diversas vezes durante esta aula. Você pode testar esses mesmos exemplos e criar novas variações dos testes para certificar que entendeu corretamente, assim já vai praticando com a linguagem.

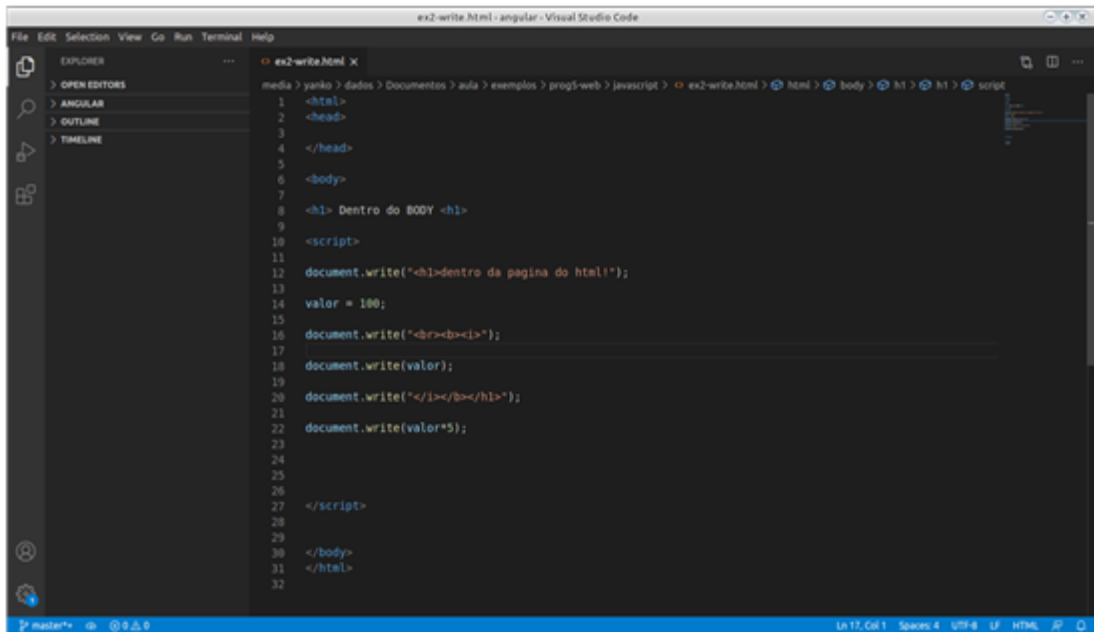
1.3 EDITOR DE CÓDIGO

Digitar uma expressão ou trecho de código diretamente no console do navegador pode ser útil durante um teste ou para verificar algum erro, mas não é produtivo ao desenvolver um aplicativo completo. Nesses casos, um editor de texto é a melhor opção. Mas o editor de texto usado para documentos formatados não é utilizado em programação devido aos controles de formatações (alinhamento, cores, espaçamentos) inseridos dentro do documento.

Uma linguagem de programação precisa ser avaliada (interpretada ou compilada) sem nenhum tipo de caractere ou controle que interfira em sua sintaxe. Por esse motivo, são utilizados editores de texto puro (que não inserem formatação) ou editores de código (que também são editores de texto puro, mas com facilitadores para a montagem de programas).

Nesta disciplina, utilizaremos como opção o editor *Visual Studio Code* (VSC) para o desenvolvimento das aplicações com o console do navegador. O VSC é desenvolvido pela Microsoft, é gratuito e pode ser utilizado em Windows, Linux e Mac.

Figura 2 – Tela do Visual Studio Code



Na Figura 2, temos um exemplo de código Javascript aberto no VSC. A aparência pode variar conforme o sistema operacional utilizado e a configuração de temas visuais da interface gráfica em uso. No caso da Figura 2, o VSC está sendo utilizado num *Linux Mint* com tema escuro.

O próprio VSC é desenvolvido em Javascript, o que permite ter uma visão da abrangência que a linguagem Javascript tem atualmente, podendo ser utilizada além do navegador.

TEMA 2 – VARIÁVEIS E OPERADORES

Uma linguagem de programação possui regras rígidas para a sua utilização. Essas regras estão relacionadas à forma como as suas instruções são digitadas e estruturadas para que o computador possa interpretar e executar essas instruções. O Javascript foi criado para automatizar páginas HTML e quando é inserido neste tipo de documento, é posicionado dentro da *tag* <script>. O código Javascript pode estar em um arquivo externo e ser carregado no documento HTML, facilitando o uso de grandes bibliotecas de códigos.

Por ser uma linguagem interpretada, poderemos testar essa parte básica no console do navegador, sem a necessidade de criar um documento HTML ou utilizar um editor de código.

Nesta aula, serão apresentadas as principais instruções da linguagem Javascript, mas mesmo estas possuem diversas variações e alternativas de uso. Os exemplos trarão os usos mais comuns e que permitem que você possa já utilizar em seus códigos. Trazer todas as variações das instruções pode tornar a leitura desnecessariamente cansativa e pouco produtiva agora no começo. Mais adiante, a partir do momento que estiver mais confiante no uso da linguagem, o leitor poderá pesquisar e testar novas instruções, novas opções e novos usos para as instruções atuais.

2.1 VARIÁVEIS

Quando precisamos armazenar dados para utilização nos algoritmos, fica mais compreensível se, em vez de um endereço na memória, utilizarmos um nome que seja fácil de perceber a sua utilização ou conteúdo armazenado.

No Javascript, podemos declarar uma variável usando *"var"*. Uma variável, porém, pode ser declarada sem essa palavra-chave, mas não é uma boa prática ao desenvolver códigos maiores, pois o *"var"* tem impacto no **escopo** da variável (o conceito de escopo e outras formas de declaração serão vistos mais adiante). No console, podemos criar pequenos trechos de código sem a utilização de *"var"* na declaração da variável, caso seja para simplificação da demonstração.

Figura 3 – Exemplos de declarações de variáveis



```
>> var msg_erro = "Alerta! operação não permitida";
< undefined

>> var valor_atual = 12.5;
< undefined

>> var impostos = 0.2;
< undefined

>> var produto = "MONITOR 19 - 1024x768";
< undefined

>> console.log("O produto ", produto, " tem valor de ", valor_atual)
O produto MONITOR 19 - 1024x768 tem valor de 12.5
```

Nas variáveis mostradas na Figura 3, temos diferentes tipos de dados que foram armazenados. No exemplo, números de ponto flutuante e *strings* foram associados a um nome, como "msg_erro" ou "impostos". O Javascript trabalha com os tipos de dados: números, *string* (sequência de caracteres), booleanos (verdadeiro ou falso) e objetos. O tipo objeto será detalhado em outra aula.

Internamente, para o Javascript, um número é sempre de ponto flutuante, ou seja, 12.0 e 12 são iguais, mas podem ser visualizados de diferentes maneiras conforme são utilizados em funções e operações. Um cuidado adicional deve ser tomado ao nomear as variáveis fazendo o leitor do código assumir algum tipo de dado que pode não ser verdadeiro. Veja exemplo da Figura 4. Ao ser nomeada como "*numero_inteiro*", o leitor do código pode inferir que, ao utilizar a variável mais adiante em seu programa, ela conterá um valor inteiro o que **pode não ser mais verdade**. Podemos ver pelo exemplo que, ao executar a expressão "*numero_inteiro* = *numero_inteiro* * 3.7", o resultado da operação de multiplicação "*numero_inteiro* * 3.7" passa a ser um valor de ponto flutuante (18.5) e será retornado para a mesma variável "*numero_inteiro*". Mais adiante, ao ser impressa a variável "*console.log (numero_inteiro)*", será mostrado o número de ponto flutuante (18.5).

Figura 4 – Nome de variável e tipos



```
>> var numero_inteiro = 5;
< undefined

>> numero_inteiro = numero_inteiro * 3.7;
< 18.5

>> console.log(numero_inteiro)
18.5
```

No Javascript, as *strings* podem ser criadas com a utilização de aspas duplas ou simples. Quando for preciso incluir uma *string* que contém também algum caractere de controle, deve ser utilizado o caractere "\" antes do controle. Além disso, as *strings* podem ser manipuladas, alteradas, divididas ou concatenadas usando operadores e funções específicas.

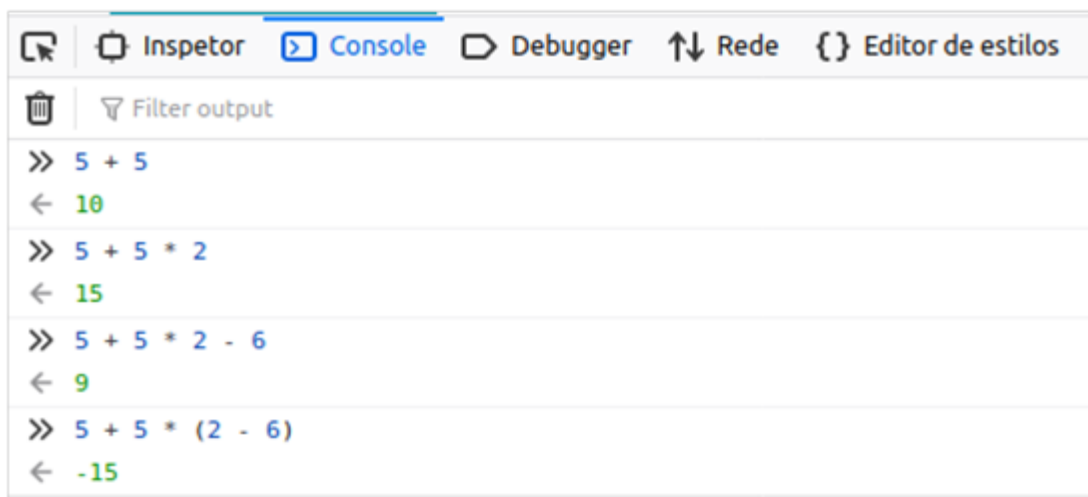
No caso de tipo de dado booleano, este representa um valor verdadeiro ou falso. Pode ser o resultado de uma comparação ou de uma operação de conversão.

2.2 OPERADORES

Os operadores são comandos que vão executar um processamento nos dados usados dentro da linguagem e irão produzir um resultado. Esse processamento pode ser uma operação aritmética, comparativa, lógica ou binária, e o resultado pode ser utilizado numa expressão ou armazenado em variáveis para uso posterior.

Os operadores **aritméticos** vão retornar valores diferentes conforme a ordem em que são executados e, no Javascript, obedecem a mesma ordem usada na matemática, inclusive executando o que está entre parênteses primeiro. Na Figura 5, é apresentada uma sequência de operações aritméticas.

Figura 5 – Operações aritméticas e precedência de cálculo

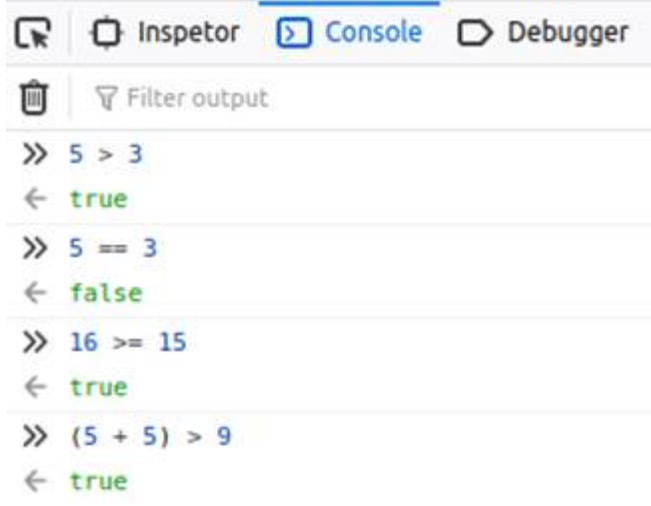


No exemplo da Figura 5, pode ser verificada uma sequência de operações iniciando com a soma (5 + 5) acrescentando uma nova operação que muda a ordem em que a expressão é executada.

Os operadores **comparativos** estão relacionados à comparação entre os operandos e sempre retornam um resultado booleano. No quadro a seguir, temos uma sequência de exemplos de comparações entre números.


Quadro 2 – Comparações entre números

	As operações de comparação ao lado mostram o resultado true (verdadeiro) ou
--	--

 <pre> >> 5 > 3 < true >> 5 == 3 < false >> 16 >= 15 < true >> (5 + 5) > 9 < true </pre>	<p>false (falso) sendo apresentados após cada processamento no console.</p> <p>Na última expressão, antes de fazer a comparação com o número 9, é feita a operação aritmética (5 + 5) e o resultado é depois comparado com o outro operando. Como resultado, temos que 10 é maior que 9 (<i>true</i>).</p>
--	---

Na utilização de operadores **lógicos**, usamos os símbolos **&&** para a operação E (ou “AND”), o símbolo **||** para a operação OU (ou “OR”) e **~** para a negação (ou “NOT”). Os operadores lógicos também são associados à teoria dos conjuntos: união (OU) e interseção (E) e podem ser utilizados para combinar mais de uma operação no mesmo teste. No quadro a seguir, temos um exemplo de utilização do operador && no Javascript.

Quadro 3 – Utilização do operador lógico &&

 <pre> >> true && true < true >> true && false < false >> true && true && false < false >> (5 > 3) && (16 >= 15) < true >> (5 > 3) && (5 == 3) < false </pre>	<p>No exemplo ao lado, temos a aplicação de operadores lógicos entre valores booleanos. Na primeira expressão, temos o resultado do operador E (&&) o qual só será verdadeiro se todos os operandos forem verdadeiros.</p> <p>As expressões lógicas também podem ser trabalhadas com o resultado de outra operação, desde que resulte num valor booleano, como (5 > 3), visto no exemplo anterior, que resulta em verdadeiro.</p>
--	--

No próximo quadro, temos o resultado de operações executadas com o operador lógico OU (||). Esse operador vai retornar falso apenas se todos os operandos forem falsos.

Quadro4 – Utilização do operador lógico ||

	<p>Da mesma forma que no operador && visto no exemplo anterior, os resultados de outras operações podem ser utilizados dentro da expressão.</p>
--	---

```

>> true || true
< true

>> true || false
< true

>> false || false
< false

>> true || false || false
< true

>> (5 > 3) || (5 == 3)
< true

>> (10 > 20) || (5 == 3)
< false

```

No exemplo ao lado, temos as expressões comparativas usadas anteriormente sendo utilizadas numa expressão maior, conectada pelo operador lógico ||.

Basta que uma das expressões seja verdadeira para que o resultado de toda a expressão seja verdadeiro.

Os operadores **binários** executam também as operações OU, E e NOT nos operandos, mas neste caso, essas operações são feitas com cada um dos bits dos operandos. O resultado é um novo valor.

A seguir, são apresentadas as operações usando o operador E (&) e o operador binário OU (|). Para compreender melhor o cálculo que é feito, é importante saber como converter o número decimal para seu equivalente binário.

Quadro 5 – Operação binária com E (&) e OU (|)

```

>> 5 & 1
< 1

>> 5 | 1
< 5

>> 6 & 2
< 2

>> 6 | 1
< 7

```

Para entender os exemplos ao lado, é preciso analisar as operações de forma **binária**. Vamos usar os valores em **4 bits**: o número 5 pode ser representado como 0101, o 6 como 0110, o 1 como 0001 e o valor 2 como 0010.

Na primeira operação temos:

0101 &

0001

O resultado foi **0001**, pois apenas os primeiros bits de cada número (da direita para a esquerda) eram verdadeiros.

Na última operação, temos:

0110 |

0001

O resultado foi **0111**, pois apenas os dois últimos bits de cada número (da direita para a esquerda) eram falsos.

Os operadores podem ser utilizados em conjunto em uma mesma expressão. Podemos criar uma operação aritmética que vai resultar num valor, que será comparado com outro e o resultado dessa comparação será logicamente testada com outros resultados. Usando mecanismos de controle que existem em todas as linguagens, instruções do algoritmo podem ser executadas ou não conforme o resultado de algumas operações.

Essas condições permitem que o programa se adapte e responda a diferentes dados que são coletados durante a sua execução, permitindo certo grau de inteligência ao algoritmo.

TEMA 3 – ESTRUTURAS DE CONTROLE DE FLUXO

O código de um programa inicia na primeira linha e segue sendo executado até a última (do topo para baixo) sequencialmente. Mas, ao criar seu algoritmo para tratar determinada situação, o desenvolvedor poderá criar desvios ou laços de repetição que poderão gerar códigos mais flexíveis e reaproveitáveis em outras situações. Esses novos controles de fluxo são as estruturas de **decisão** e **repetição**.

3.1 CONTROLES DE DECISÃO

As estruturas de controle de decisão são usadas para criar alternativas ao algoritmo. O código normalmente é executado de forma linear, da primeira instrução até a última. Mas, em alguns casos, algumas instruções devem ser executadas apenas em determinadas condições, seja conforme uma opção selecionada pelo usuário, a coleta de um valor por um sensor de temperatura, a posição do dispositivo ou outras variações de informações que são recebidas pelo programa.

A instrução *"if"* (que em português significa "se") permite ao programador verificar uma condição (verdadeira ou falsa) e, conforme a resposta, executar ou não determinadas instruções.

Uma forma de utilização da instrução *"if"* pode ser vista na Figura 6. Neste exemplo, foram criadas duas variáveis: *"dia_da_oferta"* e *"hoje"*. Depois, foi inserido um teste comparativo como primeira parte da instrução *"if"*, que em caso de resultado verdadeiro, o *"if"* executará a instrução entre { }.

Figura 6 – Controle com "if" simples



```
>> var dia_da_oferta = 15;
<< undefined

>> var hoje = 25;
<< undefined

>> if (hoje == dia_da_oferta) { console.log("Valor com 20% de desconto."); }
<< undefined

>> var hoje = 15;
<< undefined

>> if (hoje == dia_da_oferta) { console.log("Valor com 20% de desconto."); }
Valor com 20% de desconto.
```

Na Figura 6, na primeira comparação, conforme os valores iniciais, tivemos o resultado falso: ou seja, 15 não é igual a 25. Ao ser alterado o valor da variável *hoje*, e repetido o teste com o "if", o resultado passa a ser verdadeiro (15 é igual a 15) e o texto dentro do "console.log" é impresso na tela (Valor com 20% de desconto).

Numa situação mais próxima da realidade, esse código seria executado todos os dias e a informação da variável *hoje* seria coletada diretamente do dispositivo. Assim, quando o número do dia fosse o mesmo do "dia_da_oferta", o programa emitiria a mensagem.

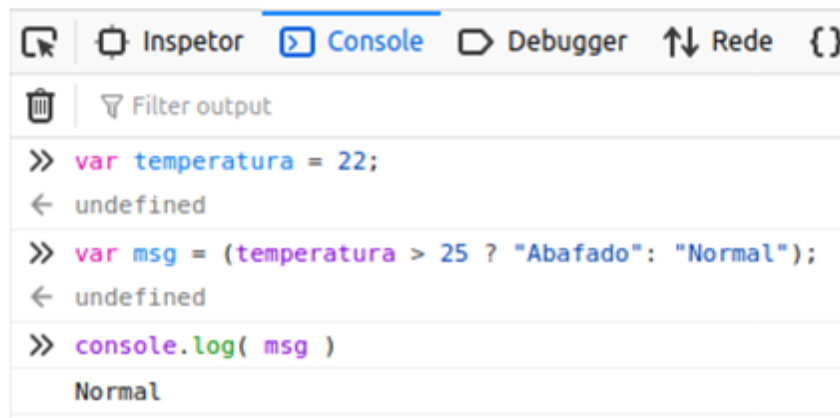
O leitor, ao ler o código de exemplo, pode ter pensado em outras situações que também poderiam ser criadas, e é isso que torna a programação mais interessante. Cada vez que vem o "e se..." na cabeça do desenvolvedor, novas opções podem ser criadas nos programas, tornando-os mais sofisticados e úteis: *e se* tiver um desconto para cada dia da semana? *E se* a oferta for apenas durante o horário comercial? *E se* o desconto for com base na quantidade de pedidos? *E se* o desconto for proporcional ao valor da compra? Esses casos vão tornando o programa mais flexível e com maior quantidade de decisões.

A instrução "if" também tem um complemento opcional que é o "else" (senão) que conterà uma instrução caso a condição seja falsa. Ao pensar na condição, o desenvolvedor estrutura da seguinte forma: "**se** a condição é verdadeira, será feito XXX **senão** será feito YYYY".

Existem outras formas de criar uma decisão dentro do algoritmo. Na Figura 7, temos o operador "?:", que pode ser usado como uma expressão. Nesse exemplo, a variável "msg" vai receber a *string*

"Abafado" se o resultado da comparação (temperatura > 25) for verdadeira ou "Normal", caso contrário.

Figura 7 – Decisão com operador "? :"



```


>> var temperatura = 22;
< undefined

>> var msg = (temperatura > 25 ? "Abafado": "Normal");
< undefined

>> console.log( msg )
Normal
  
```

Quando temos muitas condições para serem testadas na sequência, podemos também utilizar a instrução "switch...case". No próximo quadro, temos um exemplo de utilização do "switch". Sua estrutura monta uma condição na qual o primeiro termo da comparação fica na instrução "switch" e o outro, na instrução "case". Podem existir várias instruções "case" com diferentes opções para teste.

Quadro 6 – Utilização do "switch..case"

 <pre> >> var opcao = "consulta"; < undefined >> switch(opcao) { case 'cadastro': console.log(" abrir tela de cadastro"); break; case 'consulta': console.log(" abrir tela de consulta"); break; } abrir tela de consulta </pre>	<p>O exemplo ao lado pode ser construído no console utilizando <i>shift + enter</i> ao finalizar uma linha.</p> <p>Ao final da construção do switch ele é executado e retorna à instrução do "case" que foi igual ao conteúdo da variável "opção". No caso, "abrir tela de consulta" foi a mensagem impressa.</p> <p>Obs.: ao comparar <i>strings</i>, é preciso ter cuidado com o caractere espaço, que também é comparado, mas pode não ficar visível ao leitor do código.</p>
---	---

A comparação usando o "switch" pode ser feita com *strings* ou com números, e a instrução "case" pode conter várias instruções a serem executadas. Para que a instrução "case" seja finalizada e o programa possa continuar para a instrução após o "switch", um comando "break" deve ser utilizado, como mostrado no quadro anterior.

3.2 CONTROLES DE REPETIÇÃO

Uma forma de trazer produtividade para o desenvolvimento é a possibilidade de o próprio computador repetir alguns trechos do algoritmo. Isso elimina digitação e deixa o código menor, facilitando a leitura.

Certas instruções podem ser utilizadas para realizar essa repetição e, quando utilizadas em conjunto com variáveis e controles de decisão, podem ser adaptadas para muitas situações.

No próximo quadro, temos a primeira instrução de repetição: *"while"*. Essa instrução vai repetir uma ou mais instruções enquanto a condição ser verdadeira.

Quadro 7 – Exemplo de uso do *"while"* para repetição de código


 <pre>>> var contador = 1; ← undefined >> while (contador < 10) { console.log("Texto de teste nr.", contador); contador++; }</pre> <p>Texto de teste nr. 1</p> <p>Texto de teste nr. 2</p> <p>Texto de teste nr. 3</p> <p>Texto de teste nr. 4</p> <p>Texto de teste nr. 5</p> <p>Texto de teste nr. 6</p> <p>Texto de teste nr. 7</p> <p>Texto de teste nr. 8</p> <p>Texto de teste nr. 9</p>	<p>No exemplo ao lado, para controlar a condição sendo testada pelo <i>"while"</i> foi usada a variável contador. Essa variável é comparada com o valor 10 e, enquanto a comparação for verdadeira (ou seja o valor for menor que 10), as instruções dentro das {} serão repetidas.</p> <p>Uma das instruções que é repetida neste fluxo é um incremento no valor de contador. Desta forma, o valor inicial é incrementado de um em um a cada repetição.</p> <p>A cada ciclo, o teste é executado e, se ainda for verdadeiro, a repetição acontece.</p>
--	---

Nesse exemplo, a mensagem impressa vai até 9, pois o incremento da variável contador é a última instrução a ser executada dentro da repetição. Numa das repetições, o valor de contador chega a 10 e, por ser a última instrução, nada é impresso e volta para o teste do *"while"*. No teste, 10 não é menor que 10, resultando em falso e interrompendo a repetição e não imprimindo a mensagem com o valor 10.

Ao analisar o código que é usado para a repetição da impressão até 9, o leitor pode perceber que praticamente o mesmo algoritmo pode ser usado para imprimir 5, 20, 500 ou **1.000.000** de frases na tela com pouca modificação. Bastaria, nesse caso, alterar o valor 10 na condição. Apesar de ser mais didático, esse código mostra como uma repetição na programação pode ser produtiva.

Outra forma de realizar uma repetição é com a instrução *"for"*, que basicamente é uma reorganização da estrutura do *"while"*. Veja um exemplo a seguir.

Quadro 8 – Repetição usando a instrução *"for"*

	<p>No exemplo ao lado, foi utilizada novamente a variável contador e a mesma frase será impressa a cada repetição.</p> <p>Desta vez, todas as informações necessárias estão na linha da instrução <i>"for"</i>: a inicialização da variável contador, o teste de condição (<code>contador < 10</code>) e o incremento de contador.</p> <p>A cada ciclo de repetição, apenas o teste e o incremento da linha do <i>"for"</i> são repetidos.</p>
--	--

No exemplo com o uso da instrução *"while"*, o controle da condição muitas vezes fica fora da visão do desenvolvedor, o que pode levar a erro na execução da repetição. Já no outro quadro, o *"for"* facilita a organização da estrutura de repetição mantendo todos os itens usados para controlar o ciclo na mesma linha. Independentemente da forma de estruturar a repetição, as duas instruções podem ser usadas de forma semelhante.

TEMA 4 – LISTAS E FUNÇÕES

Ao trabalhar com volumes de dados repetitivos ou trechos de códigos que podem ser aproveitados em outras partes do programa ou mesmo de outros sistemas, a linguagem de programação tem estruturas para armazenamento como as **listas**, bem como formas de agrupamento de código chamadas de **funções**.

4.1 LISTAS (ARRAYS)

Ao analisar algumas situações que precisam ser automatizadas, deparamos com um grupo de dados que são semelhantes, ou podem ser classificados num mesmo grupo. Se tivéssemos de criar variáveis para armazenar esses dados, elas poderiam ficar com nomes semelhantes e trazer um pouco de confusão ao serem acessadas mais adiante no código. Para esses casos e também para permitir uma maior automatização do código, foram criadas as listas (*arrays*), que nada mais são do que variáveis com um indexador. Veja, na Figura 8, um exemplo de *array* com a impressão de um dos elementos da lista.

Figura 8 – Uso básico de “array”



```
>> var lista_de_carnes = [ "picanha", "contra filé", "costela", "alcatra"];  
← undefined  
>> console.log("O churrasco hoje vai ter ", lista_de_carnes[2])  
O churrasco hoje vai ter  costela
```

No exemplo da Figura 8, temos uma lista de *strings* contendo tipos de carnes e cada item da lista tem uma posição controlada pelo *array*. A posição em um *array* inicia sempre em **zero**. É por esse motivo que, ao solicitar a impressão do item de *lista_de_carnes* da posição 2, foi impressa a *string* “costela”.

Com a criação de *arrays*, temos uma facilidade ainda maior ao criar algoritmos que podem ser utilizados para um volume variável de informações. Este é um dos grandes desafios do desenvolvedor: **criar um algoritmo que pode ser utilizado em várias situações, sem que seja necessário fazer alterações no código.**

Veja o exemplo a seguir. Nele, foi usado o “*for*” para apresentar o menu de opções de carnes.

Quadro 9 – Uso do array em laço de repetição usando “for”

<pre>>> console.log("Neste açougue temos as seguintes opções:"); for (x=0; x < 4; x++) { console.log(x, " ", lista_de_carnes[x]) }</pre> <p>Neste açougue temos as seguintes opções:</p> <p>0) picanha 1) contra filé 2) costela 3) alcatra</p>	<p>No código ao lado, temos um controle de repetição que fará a variável <i>x</i> iniciar em 0 e vai ser incrementada até 3.</p> <p>Essa mesma variável está sendo reaproveitada para imprimir cada um dos itens do <i>array</i> <i>lista_de_carnes</i>: <i>lista_de_carnes</i>[0], <i>lista_de_carnes</i>[1], e assim por diante, enquanto durar a repetição.</p>
--	---

Ao analisar o código do quadro, podemos perceber que, com poucas modificações, ele serviria para imprimir uma lista muito maior de opções: bastaria acrescentar mais itens no *array* “*lista_de_carnes*” e trocar o valor 4 que indica a quantidade de itens do *array* (e de repetição).

Pesquisando um pouco sobre *arrays* e tentando deixar o código mais genérico (para evitar alterações a cada mudança no *array*), você vai encontrar uma forma de o próprio *array* informar a quantidade de itens: *array.length*. O resultado do código mais genérico pode ser visto a seguir.

Quadro 10 – Código com verificação automática de itens do array

<pre>>> console.log("Neste açougue temos as seguintes opções:"); for (x=0; x < lista_de_carnes.length; x++) { console.log(x, " ", lista_de_carnes[x]) }</pre> <p>Neste açougue temos as seguintes opções:</p> <pre>0) picanha 1) contra filé 2) costela 3) alcatra</pre>	<p>Ao utilizar o termo "<i>length</i>" junto ao <i>array</i> <i>lista_de_carnes</i>, o desenvolvedor não precisa mais contar os elementos para criar a condição do laço de repetição.</p> <p>Também não precisa alterar o código caso sejam inseridos ou eliminados itens do <i>array</i> <i>lista_de_carnes</i>.</p>
---	---

Outra forma de deixar mais genérica a exploração de itens de um *array* pode ser por meio da utilização de estruturas de repetição específicas para trabalhar com listas como o "*for..in*". Mas o mais importante é o desenvolvedor conseguir solucionar a situação inicial. Cada desenvolvedor tem uma forma de lidar com a situação e, à medida que vai praticando e experimentando novas situações no desenvolvimento de código, vai considerando qual a melhor abordagem. As linguagens vão evoluindo e incorporando novas instruções, permitindo deixar o código mais genérico, mais rápido ou mais legível.

4.2 FUNÇÕES

As funções foram criadas para que o desenvolvedor pudesse tornar trechos do seu código mais reutilizáveis, tanto dentro de seu programa quanto em outros programas que precisassem executar uma determinada tarefa de forma semelhante.

Muitas vezes, em grandes sistemas, eram desenvolvidas funções tão práticas e úteis que acabavam sendo utilizadas em diferentes sistemas e por diversos programadores.

Mas nem sempre a criação de uma função é óbvia desde o início. Muitas vezes, o desenvolvedor percebe que aquele trecho que já está funcionando poderia ser aproveitado em outras partes ou em outros sistemas e, então, transformar aquele trecho em uma função.

Para entender como esse processo pode acontecer e também ver como funciona uma função, vamos iniciar com um código simples e didático e depois transformá-lo em uma função mais genérica. A seguir, temos o código inicial.

Quadro 11 – Código de tabuada do número 5

<pre>>> for(num=1; num < 11; num++) { console.log("5 x ", num, " = ", 5 * num); }</pre> <p>5 x 1 = 5</p> <p>5 x 2 = 10</p> <p>5 x 3 = 15</p> <p>5 x 4 = 20</p> <p>5 x 5 = 25</p> <p>5 x 6 = 30</p> <p>5 x 7 = 35</p> <p>5 x 8 = 40</p> <p>5 x 9 = 45</p> <p>5 x 10 = 50</p>	<p>O "for" ao lado realiza uma repetição e mostra como resultado a tabuada do número 5 com um intervalo de 1 a 10.</p> <p>Esse trecho de código tem muitos itens fixos que podem ser convertidos para opções mais configuráveis. Por exemplo: o número que vai controlar a repetição pode ser uma variável assim como o número que será usado para executar a tabuada pode também ser uma variável.</p> <p>Assim, se o desenvolvedor precisasse realizar novamente uma tabuada de outro número, bastaria alterar a variável.</p>
--	--

Mesmo que o desenvolvedor acrescentasse variáveis na repetição, como indicado no quadro, ainda assim teria que copiar e colar o trecho de código para outra parte do programa e fazer as adaptações necessárias.

Mas, e se fosse possível deixar o código no mesmo local e apenas pedir para que ele fosse executado em outra parte do programa?

Essa é a lógica que está por trás das funções: o desenvolvedor cria uma identificação para aquele trecho de código e pede a sua execução em outras partes do programa.

No próximo quadro, em (a), podemos verificar a criação de uma função em seu formato mais simples: identificando um trecho de código com um nome para ser chamado posteriormente. O termo "*function*" antes do nome da função indica que o que vem a seguir é a definição de uma função e o código que for inserido entre { } será executado quando a função for chamada.

Quadro 12 – Criação da função "tabuada()"

a)	b)
----	----

<pre>>> function tabuada() { for(num=1; num < 11; num++) { console.log("5 x ", num, " = ", 5 * num); } }</pre>	
<p>Em (a), temos a definição da função chamada <code>tabuada()</code> e, dentro dela <code>{ }</code>, temos o <code>"for"</code> usado anteriormente.</p> <p>A partir da definição da função, sempre que o programador precisar imprimir a tabuada do 5, bastaria chamar a função <code>"tabuada()"</code>, como pode ser visto em (b).</p>	

Este já é um passo interessante, mas podemos deixar a função ainda mais genérica com a utilização de parâmetros. Os parâmetros são informações passadas para a função quando ela for chamada, e podem ser informações diferentes que auxiliam na adaptação da função para aquele ponto do código que ela está sendo necessária. Internamente na função, esses parâmetros podem ser usados como variáveis dentro do código, como pode ser visto na Figura 9.

Figura 9 – Função `"tabuada()"` com parâmetros

```
>> function tabuada(tab, limite) {
  for(num=1; num < limite; num++) {
    console.log(tab, " x ", num, " = ", tab * num);
  }
}
```

A seguir, temos três exemplos de chamadas da função `"tabuada()"` com diferentes parâmetros. O código é executado sem a necessidade de reconfigurar o controle de repetição ou a frase que contém o cálculo da tabuada.

Quadro 13 – Exemplos de execução da função `"tabuada()"` com parâmetros

			» <code>tabuada(83, 6)</code>		
			83 x 1 = 83		
			83 x 2 = 166		
			83 x 3 = 249		
			83 x 4 = 332		
			83 x 5 = 415		
» <code>tabuada(9, 5)</code>					
9 x 1 = 9					
9 x 2 = 18					
9 x 3 = 27					
9 x 4 = 36					
» <code>tabuada(6, 4)</code>					
6 x 1 = 6					
6 x 2 = 12					
6 x 3 = 18					

O desenvolvedor pode executar a função com diferentes limites, e o que vai impactar para aquele ponto de execução é o espaço da tela que o resultado vai ocupar. Nos casos em que a função vai executar diversas operações e a informação não vai ser finalizada dentro da função, e sim utilizada em diversas outras funções, o resultado pode ser retornado usando o termo *"return"*, e assim ser usado em expressões ou armazenado em variável.

Muitos dos utilitários da linguagem de programação são feitos com funções que recebem parâmetros. Estas são as funções padrão da linguagem e ficam disponíveis para o desenvolvedor, bastando para isso chamar a função e informar os parâmetros corretos e na ordem que devem ser usados.

No Javascript, também temos as funções padrão, que normalmente tratam de questões que foram enfrentadas rotineiramente pelos programadores que desenvolveram a linguagem ou que trabalharam no início da implementação. Mesmo depois, com as novas versões da linguagem, outras funções são criadas e algumas são eliminadas ou alteradas.

Mas, além das funções padrão da linguagem, empresas, desenvolvedores ou comunidades criam conjuntos de funções e disponibilizam muitas vezes reunidas numa "biblioteca" de funções ou *frameworks*, como visto em aulas anteriores.

4.2.1 ESCOPO

Outra consideração que o desenvolvedor precisa compreender é o impacto que as funções têm na visibilidade do conteúdo de variáveis: o **escopo**. Esse conceito define se o conteúdo de uma variável permanece disponível em trechos específicos de código e estão associados a conflitos que podem acontecer em códigos maiores, nos quais variáveis com a mesma identificação podem ser criadas, mas com usos totalmente diferentes.

Essa situação pode acontecer com o mesmo programador que desenvolve códigos grandes e acaba criando nomes de variáveis iguais sem intenção, ou quando são reaproveitadas funções de

outros sistemas ou de bibliotecas e que também têm variáveis que podem criar conflitos com o código atual. Essas questões de conflitos e escopo de variáveis serão testadas com mais detalhes nas aulas práticas.

Quando é declarada uma variável (usando ou não o termo "*var*") **fora de uma função**, o seu conteúdo fica visível para todo o arquivo (escopo global), ou seja, podemos usar o conteúdo dentro de outras funções ou fora delas. No exemplo da Figura 10, podemos verificar a situação de uso da variável "*cliente*" que foi declarada no escopo global (fora de funções) e depois sendo utilizada dentro das funções *calcula_pendencia()* e *lista_ultimas_compras()*.

Figura 10 – Variável com escopo global

```
>> var cliente = "EMPRESA ACME SA";
← undefined

>> function calcula_pendencias() {
  console.log("Pagamentos da empresa: ", cliente);
}
← undefined

>> function lista_ultimas_compras(){
  console.log("Últimas compras da empresa: ", cliente);
}
← undefined

>> calcula_pendencias();
Pagamentos da empresa:  EMPRESA ACME SA
← undefined

>> lista_ultimas_compras();
Últimas compras da empresa:  EMPRESA ACME SA
```

Ao declarar uma variável (usando o termo "*var*") dentro de uma função, o conteúdo passado a essa variável fica visível **apenas** dentro da função (escopo local). No exemplo da Figura 11, temos a variável "*cliente*" sendo declarada dentro da função *nome_cliente()* que é executada e mostra a informação da variável. Mas, quando é utilizada dentro da função *calcula_pendencias()*, como no exemplo anterior, o navegador emite um erro de execução "*ReferenceError: cliente is not defined*" indicando que a variável "*cliente*" não está definida.

Figura 11 – Variável com escopo local

```

>> function nome_cliente() {
    var cliente = "IND. ELETRÔNICA LTDA";
    console.log("Cliente:", cliente);
}
< undefined

>> function calcula_pendencias() {
    console.log("Pagamentos da empresa: ", cliente);
}
< undefined

>> nome_cliente();
    Cliente: IND. ELETRÔNICA LTDA
< undefined

>> calcula_pendencias();
❗ ▶ ReferenceError: cliente is not defined [Learn More]
>> |

```

Ao começar a utilizar as variáveis em seus códigos, o desenvolvedor vai se deparar com algumas situações em que o escopo global não é interessante, pois não quer deixar o conteúdo visível para outras funções e também não quer criar uma função só para limitar a visibilidade. Numa das revisões da linguagem Javascript, foram criados os termos: *let* e *const*.

Com o "*let*", ficou possível a criação de escopo de bloco `{ }`. Assim, uma variável criada dentro de uma repetição com "*for*", por exemplo, não seria visível fora do bloco da repetição, como podemos ver no quadro a seguir.

Quadro 14 – Escopo de bloco usando "*let*"

<pre> >> var tab = 5; < undefined >> for (let num = 1; num < 10; num++) { console.log(tab, " x ", num, " = ", tab * num); } 5 x 1 = 5 5 x 2 = 10 5 x 3 = 15 5 x 4 = 20 5 x 5 = 25 5 x 6 = 30 5 x 7 = 35 5 x 8 = 40 5 x 9 = 45 < undefined >> console.log(tab); 5 < undefined >> console.log(num); ❗ ▶ ReferenceError: num is not defined [Learn More] </pre>	<p>A variável <i>num</i> criada dentro do "<i>for</i>" foi utilizada na repetição e impressa. Depois de finalizado o bloco do "<i>for</i>", a variável deixa de existir.</p> <p>Na impressão das variáveis na sequência, a variável <i>tab</i> permaneceu visível, mas, ao imprimir a variável <i>num</i>, ela não existia mais, gerando o erro "ReferenceError: num is not defined".</p>
---	--

No caso de *"const"*, ele tem o mesmo escopo de bloco que o *"let"*, mas quando recebe um conteúdo, não pode ser alterado, ou seja, se for declarado *"const tab = 5"*, a variável *tab* **não** poderá receber outro valor.

4.2.2 RETORNO DE RESULTADOS

Os exemplos sobre escopo foram construídos principalmente para exemplificar, de forma didática, a visibilidade do conteúdo de uma variável declarada dentro ou fora de funções. Normalmente, para deixar a situação mais genérica, as funções poderiam ser criadas com o nome do cliente sendo passado como parâmetro, e as funções que primeiro identificam o cliente (seja vindo de uma base de dados ou digitação do usuário) podem retornar o resultado. Veja, no exemplo da Figura 13, uma forma mais genérica de criar a mesma situação. Desta vez, a função *nome_cliente()* identifica o cliente e guarda em variável local (que pode ser utilizada dentro da função para diversas outras situações) e depois de apresentar no console, a função **retorna** a informação para quem a executou.

Figura 12 – Funções mais genéricas com retorno e parâmetros

```
>> function nome_cliente() {  
  var cliente = "IND. ELETRÔNICA LTDA";  
  console.log("Cliente:", cliente);  
  return cliente;  
}  
← undefined  
  
>> function calcula_pendencias(NOMECLI) {  
  console.log("Pagamentos da empresa: ", NOMECLI);  
}  
← undefined  
  
>> var BD_CLIENTE = nome_cliente();  
      Cliente: IND. ELETRÔNICA LTDA  
← undefined  
  
>> calcula_pendencias( BD_CLIENTE)  
      Pagamentos da empresa:  IND. ELETRÔNICA LTDA  
← undefined  
  
>> calcula_pendencias( "EMPRESA NOVA SA")  
      Pagamentos da empresa:  EMPRESA NOVA SA
```

Depois de um tempo experimentando e testando alternativas, o desenvolvedor vai criando novas formas de resolver os problemas: com mais variáveis, menos funções, diferentes comandos ou com o auxílio de bibliotecas externas, cada desenvolvedor pode criar seu algoritmo e chegar ao mesmo resultado de diferentes maneiras. A linguagem Javascript permite utilizar diferentes estratégias a cada

situação: por exemplo, se o desenvolvedor não precisasse da informação do cliente em outras partes desse programa, poderia usar o retorno diretamente na chamada da função, pois o dado de retorno (*string*) é o mesmo, conforme o próximo quadro.

Quadro 15 – Retorno de função repassado como parâmetro

<pre>>> calcula_pendencias(nome_cliente()); Cliente: IND. ELETRÔNICA LTDA Pagamentos da empresa: IND. ELETRÔNICA LTDA</pre>	<p>No código ao lado, o resultado da função <i>nome_cliente()</i> é repassado como parâmetro para a função <i>calcula_pendencias()</i>.</p> <p>O resultado mostra a impressão feita na função <i>nome_cliente()</i> e depois a impressão da função <i>calcula_pendencias()</i>.</p>
---	--

O retorno de uma função pode ser feito com diferentes tipos de dados: números, *strings*, *arrays* ou até mesmo uma outra função. Algumas situações, como o retorno de objetos, serão vistas em outra aula.

TEMA 5 – TEMPOS E DATAS

Dentro do Javascript, temos como trabalhar com períodos de tempo e com a formatação de datas usando objetos existentes na própria linguagem. Vamos verificar como funcionam os objetos no Javascript com mais detalhes em aulas posteriores, mas utilizaremos o objeto relacionado à data e hora neste tema para exemplificar como podemos manipular dias, horas e meses dentro de nossos códigos.

5.1 DATAS E HORAS

Para consultar a data do dispositivo em que o Javascript está sendo executado, podemos invocar o "*new Date()*" e vamos ter como retorno uma *string* com todas as informações da data e hora daquele momento. Se for acionado mais de uma vez, como mostrado na figura (a) do próximo quadro, teremos os valores de segundos alterados a cada impressão, ou seja, a data do dispositivo será consultada a cada execução do "*new Date()*". Se essa informação for armazenada em uma variável, como mostrado na figura (b) do próximo quadro, ela terá a versão da data quando o "*new Date()*" foi executado. Mesmo que entre o armazenamento na variável e a impressão do conteúdo

tenham se passado horas, ao ser impressa a variável, o conteúdo da data será o da inicialização da variável, como mostrado no exemplo (b) do quadro a seguir (a última impressão da variável "dia" mostrou o minuto e segundo original).

Quadro 16 – Execução de Date() no console

a)



```
>> new Date()
← ▶ Date Wed Dec 30 2020 16:55:32 GMT-0300 (Horário Padrão de Brasília)

>> new Date()
← ▶ Date Wed Dec 30 2020 16:55:41 GMT-0300 (Horário Padrão de Brasília)

>> new Date()
← ▶ Date Wed Dec 30 2020 16:55:47 GMT-0300 (Horário Padrão de Brasília)
```

b)



```
>> dia = new Date();
← ▶ Date Wed Dec 30 2020 17:03:02 GMT-0300 (Horário Padrão de Brasília)

>> console.log(dia)
  ▶ Date Wed Dec 30 2020 17:03:02 GMT-0300 (Horário Padrão de Brasília)
← undefined

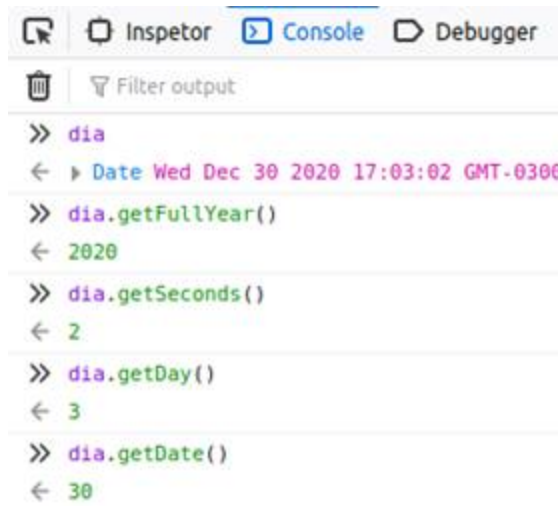
>> new Date()
← ▶ Date Wed Dec 30 2020 17:04:03 GMT-0300 (Horário Padrão de Brasília)

>> console.log(dia)
  ▶ Date Wed Dec 30 2020 17:03:02 GMT-0300 (Horário Padrão de Brasília)
```

O resultado de "new Date" pode ser acessado separando os componentes da data e hora, permitindo que possam ser feitos testes e comparações com o dia, ano ou minutos. Um exemplo pode ser visto em (a), no próximo quadro: a partir da variável "dia", foram executados os métodos "getFullYear()" para separar o ano (quatro dígitos), o "getSeconds()" para mostrar os segundos, o "getDay()" para mostrar o dia da semana (iniciando em zero) e o "getDate()" para mostrar o dia do mês.

Quadro17 – Inicialização de data e separação de componentes

a)



```
<img alt="Screenshot of a web development console showing date-related code and output." data-bbox="341 32 692 257"/>
>> dia
< > Date Wed Dec 30 2020 17:03:02 GMT-0300

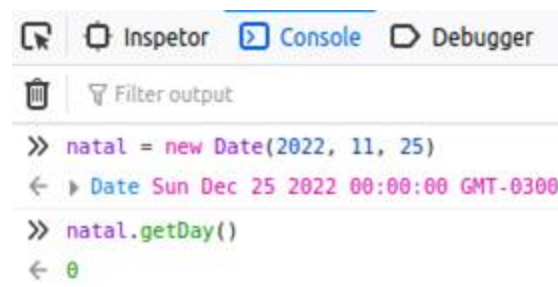
>> dia.getFullYear()
< 2020

>> dia.getSeconds()
< 2

>> dia.getDay()
< 3

>> dia.getDate()
< 30
```

b)



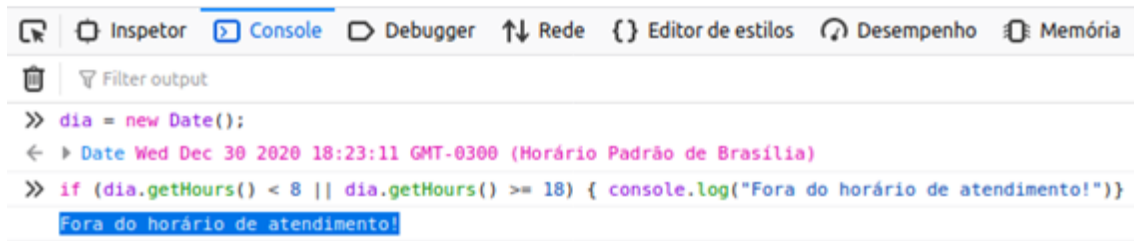
```
<img alt="Screenshot of a web development console showing date configuration and day extraction." data-bbox="341 321 692 449"/>
>> natal = new Date(2022, 11, 25)
< > Date Sun Dec 25 2022 00:00:00 GMT-0300

>> natal.getDay()
< 0
```

No mesmo quadro, em (b), vemos uma opção de configurar uma data específica **futura** para depois extrair o dia da semana (0=Domingo). Neste caso, a data está sendo fornecida pelo desenvolvedor e não está sendo coletada do dispositivo. Esse tipo de sequência é comum nos aplicativos quando precisam verificar se uma data de pagamento vai cair no fim de semana ou se um processamento deve executar somente durante a semana e, para isso, precisam testar uma data passada ou futura para extrair depois os seus componentes ou realizar cálculos entre essas datas.

Um exemplo de como essas informações sobre data e hora podem ser usadas dentro dos algoritmos está na Figura 13.

Figura 13 – Condição usando componentes de data



```
<img alt="Developer console toolbar with tabs: Inspetor, Console, Debugger, Rede, Editor de estilos, Desempenho, Memória" data-bbox="160 46 874 70"/>
>> dia = new Date();
< > Date Wed Dec 30 2020 18:23:11 GMT-0300 (Horário Padrão de Brasília)
>> if (dia.getHours() < 8 || dia.getHours() >= 18) { console.log("Fora do horário de atendimento!");
Fora do horário de atendimento!
```

No código da Figura 13, temos um *"if"* que vai mostrar uma mensagem caso o horário coletado do dispositivo naquele trecho do programa esteja abaixo de 8 ou acima de 18 horas. Esse trecho pode estar antes de uma transferência de dinheiro, no momento de login em um sistema ou na tela de atendimento de suporte. Conhecendo as informações de data e hora que podem ser acessadas e sabendo construir as estruturas de decisão, o leitor pode criar diversos outros testes e tarefas que devem ser associados com a verificação de tempo.

5.2 INTERVALOS

Quando o código Javascript começa a ser executado, ele inicia pela primeira linha e vai até a última (de cima para baixo). A execução é linha a linha. Se tem um bloco de repetição, ele executa todo o ciclo e só depois passa para a linha de código após a repetição. As funções são executadas independentemente do tamanho e tempo que vão demorar, e só quando são completadas a execução segue para a próxima instrução. Esse tipo de método é também chamado de *síncrono*.

Em alguns casos, quando se sabe que a tarefa precisa esperar um evento que não se pode controlar quando ocorrerá, temos a necessidade de uma alternativa em que o código possa continuar sua execução e, quando a situação (ou evento) finalmente ocorrer, o trecho de código que vai tratar daquela situação possa ser chamado e executado. Essa forma de tratamento é considerada assíncrona e funciona como se os códigos pudessem ser executados em paralelo de forma integrada, mas independentes.

O Javascript permite a execução de funções de forma assíncrona para que o código possa ser executado a partir de um evento imprevisível ou demorado e que não interrompa a execução das demais tarefas que o sistema precisa executar enquanto o evento não aconteça. Pode ser algo que o usuário precisa informar ou uma informação que será produzida pelo hardware, como a chegada de um dado via rede.

Em alguns casos, é possível deixar agendada uma função para executar uma tarefa com base em um intervalo de tempo, sem que o resto do código seja interrompido ou fique suspenso esperando a execução. Para esse tipo de solução, temos o `setTimeout()` e o `setInterval()`.

Na Figura 14, há um exemplo de função que foi criada na declaração da variável `agenda`, que recebe um `"setInterval()"`. Depois da declaração, são executados dois `"console.log('teste')"`, indicando que outros comandos podem ser executados e, depois (após completar os primeiros 5 segundos), iniciou a impressão das linhas agendadas.

Figura 14 – Execução de função a cada intervalo de tempo

```
>> new Date();
< ▶ Date Sun Jan 03 2021 09:32:22 GMT-0300 (Horário Padrão de Brasília)
>> agenda = setInterval( function linha() { console.log("Nova linha as ", new Date()); } , 5000);
< 84468
Nova linha as ▶ Date Sun Jan 03 2021 09:34:45 GMT-0300 (Horário Padrão de Brasília)
>> console.log("teste")
teste
< undefined
>> console.log("teste")
teste
< undefined
Nova linha as ▶ Date Sun Jan 03 2021 09:34:50 GMT-0300 (Horário Padrão de Brasília)
Nova linha as ▶ Date Sun Jan 03 2021 09:34:55 GMT-0300 (Horário Padrão de Brasília)
Nova linha as ▶ Date Sun Jan 03 2021 09:35:00 GMT-0300 (Horário Padrão de Brasília)
Nova linha as ▶ Date Sun Jan 03 2021 09:35:05 GMT-0300 (Horário Padrão de Brasília)
Nova linha as ▶ Date Sun Jan 03 2021 09:35:10 GMT-0300 (Horário Padrão de Brasília)
>> clearInterval(agenda)
```

A função `"setTimeout()"` é semelhante na sua declaração, mas executa a função **apenas uma vez**, **após** o intervalo de milissegundos.

FINALIZANDO

Nesta aula foi apresentada a linguagem Javascript, identificando e explanando os principais conceitos que a linguagem utiliza, sua importância na automação de tarefas nos navegadores e a crescente utilização da linguagem em outros domínios fora do navegador, tendo inclusive como exemplo o próprio editor de código a ser utilizado em exemplos mais sofisticados nesta disciplina.

Após lidar com a parte mais teórica, os comandos da linguagem foram apresentados, sempre associados a exemplos práticos que podem ser reproduzidos diretamente no console dos principais

navegadores.

Nesta aula, foram selecionados os principais pontos fundamentais para entender outros conceitos da linguagem que serão explorados nas outras aulas e no desenvolvimento de aplicação web. Grande parte do que a linguagem proporciona encontra-se também em outras linguagens de programação: estruturas de decisão, variáveis, estruturas de repetição, funções, *arrays*, enfim, todos esses tópicos detalhados na presente aula são pontos importantes em outras linguagens, mas foram explicados e exemplificados no contexto da linguagem de *script* no ambiente de um navegador.

Nas próximas aulas, integraremos as tecnologias apresentadas até o momento (HTML, CSS e Javascript) para, assim, tornar possível o desenvolvimento de aplicativos SPA.

REFERÊNCIAS

BROWN, E. **Learning Javascript**: Add Sparkle and Life to Your Web Pages. O'Reilly Media, 2016.

DEITEL, P. J.; DEITEL, H. M. **Ajax, rich internet applications e desenvolvimento Web para programadores**. Pearson Prentice Hall, 2009.

SOBRE Javascript. **Developer Mozilla**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/Javascript/About_Javascript#Recursos_para_Javascript>. Acesso em: 22 mar. 2021.

SELAKOVIC, M.; PRADEL, M. Performance issues and optimizations in Javascript: an empirical study. In: Proceedings of the 38th International Conference on Software Engineering, 2016, Austin. **Anais...** Austin, Texas, 2016, p. 61-72.

SPINELLIS, D. Java makes scripting languages irrelevant?. **IEEE software**, v. 22, n. 3, p. 70-71, 2005.

[1] Não ser tipada está relacionado ao conceito de definição estática de tipo de dado que as variáveis suportam.

[2] ECMA, abreviação de "European Computer Manufacturers Association", fundada em 1961, atualmente é uma associação internacional responsável por padronização de sistemas de informação (<<https://www.ecma-international.org>>).

