1 - Primeiros Passos:

### História do JavaScript:

Em 1989, Tim Berners-Lee inventou a *world wide web*, inicialmente utilizada apenas para a comunidade científica para fazer pesquisas acadêmicas, e criou o primeiro navegador CERN (*Organização Europeia para Pesquisa Nuclear*). Em 1991 o senador americano Al-Gore publica uma lei para incentivo para a computação e, um desses incentivos, era para alavancar o *mosaic*, um dos primeiros navegadores populares da internet, mas ainda não rodava nenhum arquivo de *javascript*!

Em 1993, Marc Andreessen, um dos criadores da *mosaic*, vai para a califórnia e funda a *Netscape* e domina grande parte do mercado de navegadores, porém eles ainda possuíam uma aparência estática e não muito confortável, pois havia a demanda por páginas mais dinâmicas já na época. Foi em 1995 que a *netscape* contratou Brendan Eich para aprimorar uma linguagem chamada *scheme* dentro dos navegadores (que já possuía uma sintaxe parecida com *java*) e assim nasceu o que chamamos hoje de *javascript.*

Inicialmente o javascript se chamaria *Mocha*, com uma influência da linguagem C e C# mas com fundamentos mais avançados para a época. No mesmo ano em que foi criado, o *Mocha* foi rebatizado por *Livescript* e relançado nos betas dos navegadores da netscape 2.0 e, em alguns meses, foi renomeada para *JavaScript*, aproveitando todo o marketing que a linguagem Java já possuía naquele momento.

Na mesma época, a *Microsoft* criou o seu próprio navegador (*internet explorer)* e, através de uma engenharia reversa nos navegadores da netscape aplicou uma cópia renomeada de *JScript* em seus navegadores - este nome obviamente foi utilizado para não gerar nenhum processo contra a marca!

Com o crescimento exponencial do uso da internet, a *netscape* solicitou à ECMA (Associação Europeia de Fabricantes de Computador - que age como uma célula neutra nos assuntos voltados à computação) uma padronização da linguagem JavaScript e, em 1997, veio a primeira padronização do JavaScript conhecido como *ES1*. Várias versões vieram ao longo do tempo, mas a mais famosa e recente foi a *ES6*, que veio em 2015 e rege as regras de formatação que conhecemos atualmente.

### Variaveis:

Existem três tipos diferentes de variáveis: ***let, const e var***. A **const** é uma constante que permanece no **escopo global** de nossos códigos, ou seja, podemos **utilizá-la ao longo de todo o código e não pode ser re-atribuída.** Já as variáveis do tipo **let e var** podem ser utilizadas no **escopo local** e podem ser **alteradas ao longo de nossos códigos.**

Nelas atribuímos alguns tipos de valores, como ***números,***

***‘strings’, [arrays], boolens=true||false, objetos e funções.***

Não podemos começar os nomes de variáveis com números ou

símbolos e nem utilizar espaços (pois o sistema não reconhece). Como convenção, utilizamos o conceito de **camelCase** no javascript e sempre que necessitarmos utilizar um espaço entre as palavras, colocamos uma letra maiúscula nela.

Por sintaxe, não conseguimos utilizar as mesmas palavras no código que não estejam com as letras minúsculas e maiúsculas pois o javascript é ***case sensitive.***

### Tipos primitivos, tipagem dinâmica e operações aritméticas:

Os tipos primitivos são os **de referência** e os **de valor.** Os primitivos de referência são, geralmente, *strings literais*, *números literais, booleanos, undefined ou null* e consistem em valores emitidos primariamente e podem ser redefinidas em respectivas referências ao seus ***var ou let.*** O *JavaScript* possui tipagem dinâmica, ou seja, ele consegue identificar, mesmo sem você dizer explicitamente, o valor que a variável está sendo recebido ou o tipo de variável que está sendo impressa.

O **typeof** é o comando que usamos para identificar qual é a modalidade do tipo de variável e imprime no console se ela é *string, number ou qualquer outro tipo que pode ser armazenado dentro das variáveis.*

Conseguimos fazer diversas operações aritméticas dentro do javascript (e isso se prova bem útil, já que computadores são calculadoras modernas e bem eficazes no processamento de dados!). Podemos calcular **soma (+), subtração (-), multiplicação (\*), divisão (/), exponenciais (\*\* - é o número vezes ele mesmo)** e processar esses dados conforme nossa necessidade. Podemos também incrementar *usando ++* (que acrescenta + 1 ao valor armazenado na variável) e decrementar *usando --* (que subtrai - 1 ao valor armazenado na variável).

### Condições If e Else:

São operadores que criam **condições** em nossos códigos e o **if else** podem ser traduzidos literalmente por **se e senão**. Sua formatação padrão é **if (condição){ else if (condição){} else (condição){} }** eisso quer dizer que *“se” a condição for verdadeira, execute x, senão se for verdadeira execute y e se não for nenhum execute z.*

Criar essas condições dá dinamismo às informações e podemos filtrá-las conforme queremos e dentro das condições que precisamos atender.

Pela primeira vez esbarramos com **entradas e saídas de dados**. As entradas podem ser primitivas e caracterizadas por valores armazenados em variáveis e constantes ou por arrays de bancos de dados. As saídas são os resultados que obtemos dos nossos operadores no geral. **Saídas também podem ser definidas *(em construções web)* como impressões no escopo da página que estamos trabalhando.**

### Operadores lógicos:

Na linguagem *JavaScript* temos três principais operadores lógicos que são  **&&, || e !** e respectivamente se referem a **and, or e not (ou ‘e’, ‘ou’ e ‘não’ em português)** e são de extrema importância tanto para a lógica de programação quanto para a construção de operações dentro dos códigos.

O *&&* é um operador binário, ou seja, precisa de dois elementos para funcionar corretamente e retorna um **booleano** para ambos os elementos dentro da operação. Como no javascript, sua ordem de operação será estabelecida da esquerda para a direita e quando encademos diversos destes operadores, eles se relativizam e seu resultado estará dentro da sequência de resultados que podem ser expressados como a seguir:

console.log(true && true); // true

console.log(true && false); // false

console.log(false && true); // false

console.log(false && false); // false

Já com o *||* possui uma execução também binária e

necessitamos de dois elementos em torno dele para que funcione corretamente. E para que seu retorno seja verdadeiro, um de seus valores deve ser true **ou** ser considerado **truthy (são operadores que já carregam o *true* consigo).**

Abaixo temos uma tabela mostrando como é o resultado de

cada operação, assim como no operador "AND":

console.log(true || true); // true

console.log(true || false); // true

console.log(false || true); // true

console.log(false || false); // false

Já o **not** é unário e precisa somente de um elemento para funcionar e, geralmente, inverte o valor desse elemento tornando oposto. Se tivermos uma variável ou valor considerado true , podemos gerar o resultado oposto simplesmente fazendo !variável . Ou seja, false .

O valor da operação **“ console.log(!(2 + 2) === 4); “** nos retorna ***false*** somente por conta do operador **! ou not**.

Então, sabendo que o resultado original da operação ali em cima é true , quando a gente insere o operador NOT antes da operação, teremos o valor contrário a nossa resposta final, que é false.

### Switch e Case:

Uma outra forma de estabelecermos operações **condicionais** em nossos códigos são os métodos **switch-case.** Elas se estruturam de maneira semelhante aos condicionais *if/else* e semanticamente são estabelecidas como a seguir: **switch (variável pŕe definida){ case $condição: operação desejada; break (que freia o condicional, caso essa alternativa seja a correta); }.**

Caso as condições impressas no código não sejam as que queremos, colocamos dentro do corpo da função a palavra **default** e tem um funcionamento similar ao último **else,** e aponta como saída o que não foi expressado dentro de nossas condições.

2 - Array e loop For:

### Arrays (listas):

No mundo das variáveis podemos criá-las de dois tipos: **as simples e compostas**. As simples são a que estamos acostumados, como *let e var*. As estruturas compostas são baseadas em **arrays** e com elas podemos armazenar diversas informações entre **[x, y, z]**, sempre nessa formatação padrão. São comumente utilizadas para a entrada e saída de múltiplas informações dentro de uma única variável.

Para fins de estruturas de visualização de informação, precisamos frisar que o array possui um **índice** que sempre se inicia do **zero [0]**. Dentro do javascript possuímos diversos meios de localizar, adicionar ou manipular informações dentro dos *arrays*, sem nem mesmo ter que saber quais são as informações contidas nele. Essas maneiras são bem utilizadas, pois estruturas de informações em bancos de dados possuem diversas posições em *arrays* (podem ser milhões) e com esses meios do *javascript* conseguimos construir estruturas padrões para qualquer entrada e saída destes bancos de dados.

Para armazenar mais um atributo de dentro da variável composta do *array* é comumente utilizado o **.push(“”)** e ele vem após o nome da variável (**nomeDaVariavel.push(“”)**). Essa maneira imprime o valor desejado, entre parênteses, *ao final* de nossos arrays. Para imprimir esse valor no *início* de nosso array, utilizamos o comando **unshift().**

Conseguimos retirar o *último* item de nosso array pelo comando **.pop()** e podemos retirar o *primeiro* item da lista do array pelo comando **.shift()**.

Em momentos que precisamos saber o tamanho de nossos arrays utilizamos **.length** após o nome da nossa variável que contém o array (**nomeDaVariavel.lenght**) e esse comando nos mostra o tamanho de nosso array iniciando do zero [0]. Já o método **.sort (nomeDaVariavel.sort()**) nos retorna o nosso array já ordenado (pode ser em ordem alfabética ou por crescente numérico). Podemos consultar os atributos únicos do nosso array utilizando seu número de ordem no formato **nomeDaVariavel[x], onde x é a posição do elemento que queremos visualizar.**

Também conseguimos encontrar o índice de um item específico do array utilizando o comando **.indexOf(“x”)** onde **x** deve ser o valor do item que queremos localizar.

Podemos utilizar laços de repetição **“for”** para percorrer os índices de nossos arrays e manipulá-los de acordo com nossas necessidades e assim, criando algoritmos para nossos sistemas.

#### **Documentação de *arrays:***

<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array>

### For:

O **for** é uma das estruturas de laço de repetição que consegue elaborar um *loop* repetitivo de funções determinadas em nossos códigos sem precisarmos elaborar caso por caso. É extremamente importante ao manipularmos grandes bancos de dados ou estruturas que podemos inserir livremente informações adicionais. Com essas estruturas conseguimos elaborar algoritmos para nossos softwares.

Existem diversas maneiras de utilizarmos o **for**, mas a estrutura inicial e que conseguimos manipular dados de uma maneira mais fácil é **for (var i = 0; i < array.length; i +=1;)**. Dentro dos atributos do **for**, primeiramente identificamos a variável que desejamos percorrer, seu valor que determina a posição que vamos percorrer *a partir dela*, a distância que queremos percorrer (*array.length percorre o tamanho de um array, por exemplo, e se tivéssemos utilizado o símbolo <=, um de seus itens seria* ***undefined*** *pois os índices do array iniciam do zero[0]*) e, por último, determinamos de quanto em quanto devemos percorrer esse dado (*no exemplo, é de 1 em 1, pois* ***+=*** *significa* ***i = i +1*** *dentro de nosso exemplo*).

Dentro do corpo ( *{ }* )da função do **for** damos as coordenadas de como lidar com essas varredura de informações que conseguimos na saída do nosso laço de repetição. Podemos executar outras funções dentro do corpo do loop for e conseguimos criar estruturas ainda mais elaboradas por conta disso!

### For (.. of ..) :

A estrutura **for (.. of ..)** é uma nova funcionalidade que foi aprimorada em 2015, com a chegada do ECMAScript6 (**ES6**) e com ele conseguimos iterar (***fazer ou dizer novamente; repetir, reiterar.***) arrays e objetos. A estrutura continua sendo uma laço de repetição no qual podemos iterar os valores das propriedades das variáveis compostas.

A estrutura for/of é construída em sintaxe por ***for* (i *of* index)** onde **i** é a variável que criamos para armazenar os dados enquanto **index** é percorrido. Dentro do corpo da função também conseguimos criar métodos para manipular os dados e elaborar melhor sua saída de dados, como no exemplo a seguir, que soma + 1 a cada elemento:

let arrOfNumbers = [10, 20, 30];

for (let sum of arrOfNumbers) {

sum += 1;

console.log(sum);

}

// 11

// 21

// 31

3 - Lógica de Programação e Algoritmos

### Lógica de Programação:

A lógica de programação são conjuntos de normas baseadas em lógicas matemáticas que visam facilitar nosso raciocínio no momento em que estamos criando nossos códigos. A lógica de programação é uma perspectiva que adquirimos em *escopo global*, pois não está baseada somente em uma linguagem específica, e vem através da *repetição*, pois está amplamente conectada ao ato de codar. A cada passo que damos, a cada código que escrevemos, ficamos mais perto do aprendizado da lógica e, esse raciocínio, é o que nos faz escrever *algoritmos* melhores.

**Primeiramente, temos que constatar que o computador é burro!** Nossos computadores não possuem sentidos auditivos, visuais, táteis ou quaisquer outro que nos torna humanos. A ferramenta computador não passa de uma **calculadora** avançada, por isso, temos que ser extremamente detalhados e pontuais em nossas informações.

Para começarmos a solucionar nossos problemas, é indicado fragmentá-lo em pedaços menores e detalhar todos esses pedaços para criar uma **narrativa.** Essa narrativa costuma ajudar as pessoas que têm a difícil tarefa de traduzir o mundo real para códigos do mundo virtual.

### Algoritmos:

Os algoritmos são comandos encadeados com certa lógica e que buscam solucionar problemas. Como já dissemos, **computadores são burros** e levam tudo ao *pé da letra* e por isso devemos descrever nosso passo a passo bem detalhado para ele em nossos códigos - **inclusive em nossa sintaxe de escrita**, pois o que passaria despercebido para nós, seres humanos, como um erro de ortografia, tende a travar blocos inteiros de códigos quando uma letra está fora do lugar!

Por sua vez, os *algoritmos* são as linhas de pensamento que utilizamos para solucionar um problema. Quando nos deparamos com esses problemas, devemos **parar para raciocinar e enxergar a solução e possíveis caminhos para a sua resolução**, pois existem inúmeros caminhos até ela e é aí que o algoritmo soluciona o problema. O algoritmo *deve ser visto como o caminho que utilizamos para solucionar o problema em questão* e é formado por todo esse conjunto, desde a descrição do problema, passando pela tomada de decisões, até os códigos finais. Ele é a junção disso tudo em forma de código!

*“Agora, pegue papel e caneta ou abra um bloco de texto e gaste 5 minutos escrevendo um algoritmo sobre como se vestir para uma festa.”*

1. Uma hora antes da festa;
2. Tomo banho;
3. Passo perfume, desodorantes e creminhos paficaxeroso;
4. Escovo meus dentes;
5. Escolho aquelaaa cuequinha top;
6. Experimento o look já pré-definido em minha cabeça;
7. Se ficou bom, coloco o cinto, as meias e sapato;
8. Se não ficou bom, experimento outra calça e blusa até retornar ao passo #6;
9. Faço meu penteado;
10. Guardo minhas chaves, carteira e celular no bolso;
11. #partiufesta!

***Após escrever o algoritmo, responda às seguintes perguntas:***

* ***Eu resolvi o problema?***

Simmm! O problema foi devidamente solucionado.

* ***Havia outras maneiras de resolver o problema?***

Há outras maneiras de se resolver, como se vestir mais rápido, ou não seguir alguns passos da resolução. Alguns meios são melhores, outros piores, mas funcionam igualmente.

* ***A maneira que eu escolhi foi a mais eficiente possível, ou havia como fazer a mesma coisa com menos passos?***

Existem maneiras mais fáceis para concluir o processo sim!

* ***Seria possível inverter ou retirar algum passo?***

É possível alterar a ordem da vestimenta, ou pular alguns passos, como *“se já tomei banho há pouco tempo, preciso tomar outro?”.*

* ***Se eu fosse um computador, conseguiria entender todas as instruções?“***

Não, porque se você tomar um banho sendo computador, iria queimar todo e morrer tentando! Huehue Brincadeiras a partes, possivelmente não entenderia, pois pulei diversos detalhes como: *abrir a porta do banheiro, fechar porta do banheiro, entrar no chuveiro, abrir porta e entrar no quarto, abrir armário para pegar perfume* e mais uma sequência de lógicas pontuais.

O passo a passo lógico que fizemos acima possui o nome de **baby steps** (*passos de bebê*) e, a princípio, podem parecer um pouco bobos, mas são esses passos que nos levam a uma resolução detalhada para formularmos nossos algoritmos. Os baby steps nos ajudam a analisar friamente todas as etapas de problemas que vamos nos deparar e nos ajudar a fragmentá-los em partes menores para que a solução venha, uma por vez, e na sequência que precisamos. Geralmente essa resolução pode ser dada em três etapas: **1 - Interpretação; 2 - Criação do algoritmo; 3 - Codificação do algoritmo;**

Portanto, ao interpretar o problema ou enunciado, utilizamos o *baby steps* para criar a nossa lista para resolver o problema (assim como criamos o passo a passo para nos vestir para uma festa). Logo aṕos, usando os mesmos *baby steps* dividimos nossa interpretação e colocamos nosso conhecimento em prática, traduzindo do *‘mundo real’* para o *‘mundo técnico'* e explicitando palavra como *'adicionar um array’* caso exista a necessidade de uma variável composta, ou *‘elaborar um loop de repetição’* caso nossa necessidade seja encontrar um elemento, ou ainda *‘elaborar uma estrutura condicional’* caso exista um **‘se’, ‘senão’, ‘caso’** ou a necessidade de validação em uma etapa existente para elaborar a resposta ou solução. E, logo na sequência, partimos para o VSCode e codamos essa criação de algoritmo, para transmitir à máquina nossa solução em uma linguagem que ela entenda!

É muito importante nos perguntarmos a cada criação de algoritmo, resolução de problema ou utilização da logística de programação:

* Eu resolvi o problema?
* Havia outras maneiras de resolver o problema?
* A maneira que eu escolhi foi a mais eficiente possível?
* Seria possível inverter ou retirar algum passo?
* Se eu fosse um computador, conseguiria entender todas as instruções?

E lembre-se de que, se algo der errado e ficamos travados diante de um problema, recomece tudo utilizando os **baby steps** e procurando desenvolver uma solução com uma ‘nova cabeça’!

### Complexidade de código:

Códigos muito complexos não são uma boa prática de mercado, pois isso dificulta a testagem e o aprimoramento ao longo do tempo. É normal nos sentirmos impostores ao apresentar códigos complexos e muito detalhados no início de nossas carreiras, mas o mais importante é que **estamos entregando e praticando**! Conforme a prática acontece, conseguimos ver onde usamos demais certo aspecto ou onde poderíamos utilizar ferramentas diferentes para deixar o código ainda mais enxuto. O importante é repeti-los todos os dias, se desafiar a conhecer algo novo e aprimorar 1% por dia!

4 - Objetos e Funções:

### Objetos:

Já vimos que variáveis no javascript nada mais são do que containers para armazenamento de dados. Um **objeto** é um dado independente, com propriedades e tipos especificados entre chaves ( *{ }* ) e que podem armazenar valores. É uma estrutura muito utilizada para modelar problemas e **fazer abstrações** do mundo real. A sintaxe variavel do tipo objeto é dada da seguinte maneira:

**let objeto = { valor : x, valor2: y, valor3 : z; };**

Dentro dos valores de objetos podemos armazenar tanto strings, quanto números, quanto arrays e booleanos ou **novos objetos**. Para acessar esses valores, basta escrever o nome da variável seguida de um ponto **( . )** e o nome do valor (por exemplo: **objeto.valor3** e o resultado é *“z”*). Podemos também utilizar a ***notação de colchetes***, utilizando o nome da variável seguido de ***colchetes*** **( [ ] )** com o valor contido no objeto entre aspas dentro das chaves (por exemplo: **objeto[‘valor3’]**. Caso fosse **objeto[‘valor4’]=w,** esse valor seria armazenado dentro da variável com o valor nomeador de *valor4*).

Já para acessar **valor** que está contido **em objetos dentro de um objeto** utilizamos também o ponto ( . ), ou seja, vamos dizer que nosso objeto é **let obj1 = {valor1 : {sub1 : a, sub2 : b,}, valor2 : c}**. Conseguimos acessar o valor b chamando pela variável **obj1.valor1.sub2.**

**Podemos nomear os valores dentro de objetos com números**, mas só conseguimos chamar esse valor se utilizamos a chamada através da *notação de colchetes ( [ ] ).*

Quando identificamos um *array composto de objetos*, ou um *array dentro de um objeto*, utilizamos as mesmas conotações padrão para nos comunicar com estes *arrays!*

E é importante frisar que, no JavaScript utilizamos somente **camelCase** ( e não o *kebab-case*).

### For (.. in ..):

O laço de repetição **for / in** funciona para percorrer tanto os *arrays* quanto para *objetos.* O atributo ***in*** mostra o conteúdo em nomenclatura do dos objetos e pode ser traduzido literalmente do inglês por **(dentro)**. É uma funcionalidade que percorre as propriedades dos objetos que forem enumerados com base na ordenação de inserção ao objeto, e não nos valores das propriedades, como o  *for / of*. Quando utilizamos a propriedade *for / in* para percorrer arrays o que retorna são os índices de suas posições (**e não os próprios valores)**, pois os *arrays* são variáveis compostas primárias (*e diferente de quando utilizamos for / of para o mesmo propósito).*

### Funções:

Para deixar nossos códigos mais organizados e exercermos a lógica da programação, dividimo-nos em blocos lógicos, mais conhecidos como funções. São as **funções** que permitem automatizar certos comandos em códigos maiores, encapsulando instruções que executam tarefas específicas.

Se tivéssemos que recriar as funcionalidades do carro em nossos códigos, colocaríamos o ato de *ligar e desligar, acelerar, frear, direcionar, dar seta, dar ré, ligar e desligar faróis e etc*, não é mesmo? Para descrevê-las em um código, temos que declarar as funções através da palavra **function**, seguido do **nomeDaFunção** e seus **parâmetros,** *que podem ou não existir, e separando-os por vírgula.* Os parâmetros são convertidos em **variáveis locais** dentro de uma função, o que não impede de criarmos outras dentro do escopo dela!

Funções devem **retornar** alguma informação, depois de executar o bloco que desejamos. As funções **não** são executadas por leitura do código e precisamos **chamá-las.**

Voltando às tarefas do carro, podemos declarar o ato de desligar e ligar o carro através de:

**var statusCarro = ‘desligado’;**

**function ligarDesligar () {**

**If (statusCarro === ‘desligado’){**

**statusCarro = ‘ligado’;**

**} else {**

**statusCarro = ‘desligado’;**

**}**

**return statusCarro;**

**}**

Onde statusCarro esta, em **escopo global**, recebendo a string

‘desligado’. Após chamar a função, a variável passa pelo bloco, obedecendo as ordens direcionadas à ela, que modifica-se conforme acionamos, ou seja, todas as vezes que chamamos a função a string muda de valor, retornando o status oposto ao atual, e funciona **exatamente como se estivessemos virando a chave de um carro!**

Para saber se nossa função precisa ou não de parâmetros, devemos pensar em nossas lógicas fundamentais. Precisamos atribuir uma variável para exercer o código dentro do bloco da função? Devemos praticar os **baby steps** aqui!

Quando um código está muito complexo (**o que é extremamente comum**), devemos quebrá-lo em funções menores - ou **subfunções** - dentro do bloco de código da função. Podemos acionar funções descritas no escopo global ou local dentro de nossas funções! Não é recomendável uma função exercer diversas tarefas ao mesmo tempo e para isso é que conseguimos quebrá-la em tarefas menores, para diminuir sua complexidade.