JavaScript - Fundamentals – Parte 2

Strict Mode

Strict Mode é um modo mais rigoroso de interpretação da linguagem JavaScript, que proíbe certas práticas que sempre foram permitidas, mas não são recomendadas (como a criação de variáveis globais implícitas).

De outra maneira, Strict Mode é uma nova feature do **ECMAScript 5** que permite fazer o código JavaScript rode em **modo mais rigoroso**. Neste modo, a engine de JavaScript tem seu comportamento modificado, gerando erros que antes eram silenciados e, até mesmo, proibindo o uso de certas partes da linguagem que são tidas como problemáticas, nos forçando assim a escrever um código de melhor qualidade e ajuda a captar bugs mais precocemente.

Existe duas maneiras de se utilizar o "use strict":

- No topo do arquivo, a diretiva aplica o modo estrito para o arquivo todo;
- Como a primeira linha de código de uma função, a diretiva aplica o modo estrito somente dentro da função (incluindo outras funções eventualmente declaradas dentro dela).

O grande benefício de se usar o Strict Mode é reduzir a chance de existirem no código bugs difíceis de localizar (como conflito de nome ao usar uma variável ou uma variável não declarada). Ela também mostra que indicadores como **implements**, **interface, let, package, private, protected, public, static e yield** são palavras reservadas quando utilizados no strict mode.

Functions

Funções são pedaços de códigos que podemos reutilizar diversas vezes. É igual uma variável, porém possui diversos códigos em seu escopo. Vale lembrar que variável segura um dado, enquanto **funções podem ter um ou mais linhas de códigos.**

Exemplo de sintaxe de função simples:

Para iniciar uma função, precisamos primeiro declara-la, depois dar um nome e, em seguida, usar os parâmetros se necessário. Esses parâmetros serão usados para o recebimento de argumentos (dados) e retorno deles. Essa parte da função (parâmetros) pode ser chamada de variáveis.

Exemplo com uso de parâmetros:

```
function escreva(mensagem) {
    console.log(`A mensagem de hoje é: ${mensagem}`);
}

escreva('Tenha um bom dia!');
```

As variáveis que tiverem dentro do parâmetro receberão dados quando a função for chamada. O dado recebido vai ser utilizado dentro do escopo da função e retornará o dado, se o desenvolvedor quiser que a função retorne um dado. Em via de regra, funções podem ou não retornar dados. No exemplo acima, podemos chamar a função diversas vezes com mensagens diferentes.

Funções com Retorno

Podemos retornar um valor em uma função. Quando retornamos um valor, podemos armazena-los em uma variável.

```
JS script.js > ...
1   function soma (x, y) {
2       return x + y;
3   }
4
5   const calcSoma = soma(10, 20);
6   console.log(calcSoma);
```

No exemplo acima, invocamos a função “soma” colocando argumentos para ficarem dentro do parâmetro. Ou seja, dados de entrada. Uma vez que esses dados estejam dentro da função, ela retornará a soma de x + y (10 + 20). De maneira melhor, os dados voltam para o parâmetro. Podemos ver o resultado dessa operação pelo console do JavaScript ou armazena-lo em outra variável.

Outro exemplo de retorno de função:

```
function fruitProcessor(apples, oranges) { //declaração da função
    const juice = `Juice with ${apples} apples and ${oranges} oranges.`; //
    linhas de códigos
    return juice; //o return irá retornar o valor para a constante juice, assim
    que utilizarmos a função fruitProcessor();
}

const appleJuice = fruitProcessor(5, 0); //aqui estamos invocando/chamando/
rodando a função fruitProcessor. Ela irá retornar dado para apples e oranges,
como foi estabelecido na declaração do fruitProcessor. Ou seja, 5 apples e 0
oranges.
console.log(appleJuice);
const orangeJuice = fruitProcessor(6, 8); //mesma coisa com valores diferentes.
console.log(orangeJuice);
```

Esse processo **fruitProcessor()**, que está usando a função, é chamado de “invoking”, “running” ou “calling”. Desse modo, a função irá receber dados para alocar em **apples** e **oranges** (ambas variáveis sem dados declaradas nos parâmetros). O que vai fazer os dados voltarem para dentro dos parâmetros é a função é chamada de **return.**

Functions Declarations vs Expressions

Funções podem ser declaradas ou expressas. A diferença entre ambas é a forma como podemos usa-la. Uma **função declarada** pode ser referenciada antes de ser declarada, ou seja, ela não ficará presa apenas em uma parte do código. Em outras palavras, elas são carregadas antes de qualquer código ser executado.

Exemplo de função declarada:

```
5
6   apresentacao('Felipe', 24);
7
8   function apresentacao(nome, idade) {
9       console.log(`Meu nome é ${nome} e tenho ${idade} anos`);
10      return apresentacao;
11  }
12
13  apresentacao('Felipe', 24);
```

Quando invocamos a função **apresentacao()** ela poderá ser usada antes ou depois da declaração, sem nenhum problema ou erro de execução. Isto acontece porque, por padrão, o parser do JavaScript lê todo o código antes de executá-la. E quando ele faz esse parse, ele move todas as definições de funções e variáveis para o topo de sua análise para só então percorrer o código. Isto é chamado de **hoisting**.

Agora, uma expressão de função, o nome não é mais obrigatório pois, na maioria dos casos, se trata de uma função anônima. O nome dela, na verdade, ficará dentro de uma variável.

Exemplo de expressão de função:

```
//função expressada

const apresentacao2 = function (nome, idade) {
    console.log(`Meu nome é ${nome} e tenho ${idade} anos`);
    return apresentacao2;
}

const aluno_nome = apresentacao2('Paulo', 26);

//------------------------------------------------------
```

Além do fato de que, expressões são obrigatoriamente criadas para gerar um valor, enquanto declarações apenas ditam o que deve ser feito para, futuramente (e se requisitadas), possam gerar um valor.

Arrow Function

Arrow Function são, na verdade, um meio de produzir uma função de forma simplista. Ou seja, com menos linhas de código, mais limpo e fácil de ler.

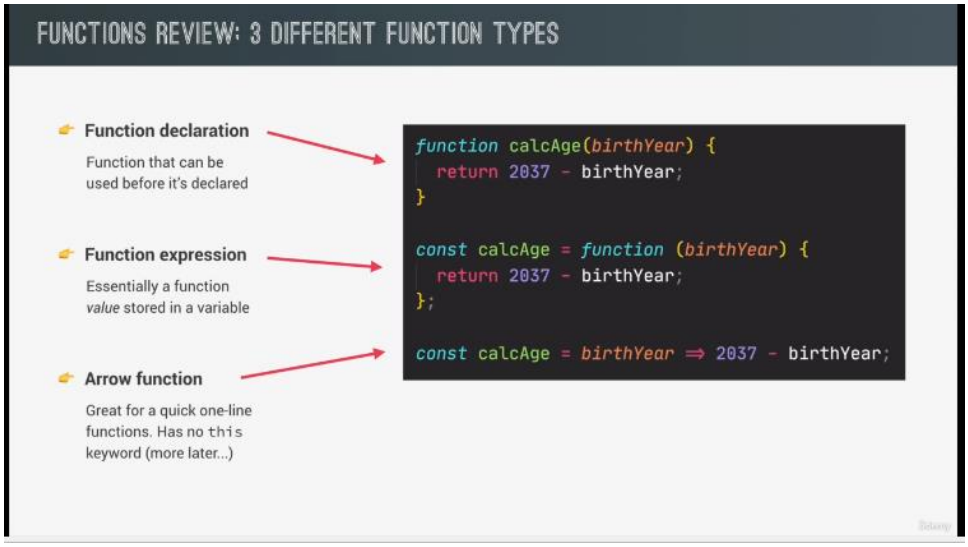
```
const candidato = (nome, idade, pais) => console.log(`Meu nome é ${nome}, tenho${idade} e moro no ${pais}`);

const grupo_cadidatos = candidato('Felipe', 24, 'Brasil');
```

No final, Arrow Functions são apenas mais uma forma de observar as funções que pretendemos aplicar. Se for apenas uma linha, então Arrow é uma boa opção de código.

Functions Calling Other Functions

```
function cutFruitPieces(fruit) {
    return fruit * 4;
}

function fruitProcessor (apples, oranges){
    const applePieces = cutFruitPieces(apples);
    const orangesPieces = cutFruitPieces(oranges);

    const juice = `Juice with ${applePieces} pieces of apple and ${orangesPieces} pieces of orange.`;
    return juice;
}

console.log(fruitProcessor(2, 3));

//Data Flow
/*
1 - Invocamos fruiProcessor;
2 - os argumentos de fruitProcessor vai para o parametro dele;
3 - Depois do parametro, ele vai pra função cutFruitPieces;
4 - Chegando na função cutFruitPieces ele tem que ir para o parametro dele;
5 - o parametro retorna pra dentro do escopo da função e faz o que deve ser feito.
6 - ela volta pra função invocada dentro de fruitProcessor;
7 - ela fica armazenada na variavel applePieces ou orangesPieces;
8 - que assim vai ser armazenado em Juice, e aplicado no Template Literals.
9 - o console.log(fruitProcessor) é mostrado no console.
*/
```

O principal objetivo de chamar outra função é não repetir métodos que poderiam ser encaixados em uma única função. Assim, não precisamos repetir a mesma coisa durante vários processos. Se repetimos, então teremos que, futuramente, alterar cada valor, pedaço por pedaço. Com a função, podemos alterar apenas uma linha de código e alterar toda uma cadeia de resultados.



Arrays

São estruturas de dados usadas para armazenar vários valores em uma única variável. Se você tem uma lista de itens (uma lista de nomes de carros, por exemplo), armazenar os carros em variáveis individuais pode ser assim:

```
let carro_tipo1 = 'Celta'
let carro_tipo2 = 'BMW'
let carro_tipo3 = 'Carro de goiano'
let carro_tipo4 = 'HB20'
```

Quanto mais dados tivermos, mais variáveis temos que criar para atribuí-las. Com arrays, podemos colocar todos os em apenas uma variável, assim podemos acessar os valores consultando o número de índice(i).

Exemplo de Array:

```
// //Com Arrays podemos armazenar todos os dados acima em uma
única variável chamada cores.

// //Exemplo Array:

const cores = ['Azul', 'Amarelo', 'Vermelho', 'Laranja'];
console.log(cores);
```

Arrays também podem ser expressadas, mas dessa vez usando New Array. Em vez de colchetes, usamos parênteses:

```
//Outra forma de fazer a função Array. Em vez de colchetes,
usamos parenteses.

const cores2 = new Array ('Azul', 'Amarelo', 'Vermelho',
'Laranja');
console.log(cores2);
```

Arrays podem armazenar qualquer tipo de dado - números, strings, funções, etc.

Para acessar os dados de uma Array, precisamos usar o índice [n = número da posição dos elementos].

```
18
19  const cores2 = new Array ('Azul', 'Amarelo', 'Vermelho',
    'Laranja');
20  console.log(cores2);
21  //Como no exemplo abaixo, podemos ter acesso ao conteúdo de
    Array
22  console.log(cores2[0]);
```

Como Arrays são baseadas em zero-index, o primeiro elemento começa sempre com zero e adiante, então o seguinte será número 1.

Assim como podemos acessa-la, também podemos conferir o tamanho de uma Array ou em outras palavras, conferir quantos dados ela possui. Basta que usemos a propriedade **length. Essa propriedade realmente vai nos dizer quantos elementos uma Array tem, deixando de lado o zero-index.**

```
9   const cores2 = new Array ('Azul', 'Amarelo', 'Vermelho',
    'Laranja');
0   console.log(cores2);
1   //Como no exemplo abaixo, podemos ter acesso ao conteúdo de uma
    Array
2   console.log(cores2[0]);
3   console.log(cores2.length); //length irá mostrar a quantidade
    de elementos de uma array.
```

Assim como podemos ter acesso de um dado de uma Array pelo índice e saber a quantidade de dados dela com length, podemos **também conferir qual é o último elemento de uma array usando também length:**

```
const cores2 = new Array ('Azul', 'Amarelo', 'Vermelho',
'Laranja');
console.log(cores2);
//Como no exemplo abaixo, podemos ter acesso ao conteúdo de uma
Array
console.log(cores2[0]);
console.log(cores2.length); //length irá mostrar a quantidade
de elementos de uma array.
console.log(cores2[cores2.length - 1]);//podemos usar length
para saber qual é o último elemento de uma array.
```

console.log(cores2[cores2.length - 1]);

Usamos menos 1 para subtrair o dado da Array. Ou seja, “tiramos” ele da Array, no exemplo acima, para mostrar o elemento no console. **Na verdade, como a função dele própria diz, nomeDaArray.length vai calcular essa subtração da direta para a esquerda.**

Podemos reatribuir um dado de uma Array normalmente, usando índice.

```
const cores2 = new Array ('Azul', 'Amarelo', 'Vermelho',
'Laranja');

cores2[1] = 'Preto'; //iremos substituir a cor amarela por
preto.
```

Arrays podem ser usadas com qualquer tipo de valores. Por exemplo, podemos fazer uma operação matemática dentro dela e ela irá funcionar corretamente, já que JavaScript espera receber expressões e Array tem a função de dizer, no final, um resultado de um valor.

Podemos inserir variáveis, cálculos matemáticos, outras arrays e funções.

```
28
29  const primeiroNome = 'Felipe';
30  const felipe = [primeiroNome, 'Miranda', 'Marreiros', 2021 -
    1997, 'Amanda', 'Desempregado', cores];
31
32  console.log(felipe);
33
34
```

Se formos olhar a quantidade de dados dentro da Array felipe, podemos ver que existe 7 elementos. Outra array dentro dela não acrescentam na array principal.

Exemplo agora com funções e arrays, calculando a idade:

```
34  const calcIdade = function (idade){
35      return (2021 - idade);
36  }
37
38  const idades = [1990, 1982, 2002, 2009, 2015];
39
40  const result1 = calcIdade(idades[0]);
41  const result2 = calcIdade(idades[1]);
42  const result3 = calcIdade(idades[2]);
43  const result4 = calcIdade(idades[idades.length - 1]); //pulamos
    para o último com length.
44  console.log(result1, result2, result3, result4);
```

Também podemos guardar resultados de uma array em outra array.

```
34  const calcIdade = function (idade){
35      return (2021 - idade);
36  }
37
38  const idades = [1990, 1982, 2002, 2009, 2015];
39
40  const result1 = calcIdade(idades[0]);
41  const result2 = calcIdade(idades[1]);
42  const result3 = calcIdade(idades[2]);
43  const result4 = calcIdade(idades[idades.length - 1]); //pulamos para o último com length.
44
45  const todosResultado = [calcIdade(idades[0]), calcIdade(idades[1]), calcIdade(idades[2]),
    calcIdade(idades[idades.length - 1])];
46
47  console.log(todosResultado);
```

Operações Básicas com Arrays

Métodos que adicionam novos elementos em uma Array:

- **NomeDaArray.push():** adiciona um elemento no final de uma Array. Ou seja, da direita para esquerda.
- **NomeDaArray.unshift():** adiciona um elemento no início de uma Array. Ou seja, da esquerda para direita.

Métodos que removem um elemento de uma Array:

- **NomeDaArray.pop():** remove um elemento do final de uma Array. Ou seja, da direita para esquerda. Aqui não precisamos usar argumentos para tirar o elemento, simplesmente tira do final.
- **NomeDaArray.shift():** remove o primeiro elemento de uma Array. Ou seja, da esquerda para direita.

Outros métodos:

- **NomeDaArray.indexOf():** mostra em qual posição um elemento de uma array está.

- **NomeDaArray.includes():** fazemos uma comparação strict equality, ou seja, faz uma comparação literal.

Introdução a Objetos

Objetos são como uma espécie de "super variáveis" que armazenam uma "**coleção de valores**" referenciados por nome, e que podem ser recuperados para serem utilizados em diversas outras partes de um programa. Em JavaScript praticamente qualquer tipo de dado é um objeto.

Cada item dessa "coleção de valores" é chamado de propriedade. Cada propriedade é composta por um par de "nome: valor". Quando uma propriedade armazena uma função, ela se torna o que chamamos de **método**.

Exemplo comparando Array com Objeto:

```
1  //exemplo de array
2
3  const carro1 = ['Vermelho', 2018, 'BMW', 1589];
4
5  //exemplo de objeto
6
7  const carro_descricao  = {
8      cor : 'Vermelho',
9      anoFabricao : 2018,
10     tipo : 'BMW',
11     kmRodado : 1589,
12     grupoFab : ['PNI', 'BR', 'KSM']
13 };
14
15 //Objetos podem conter qualquer tipo de dados, inclusive arrays.
16
```

A maneira mais simples (e recomendável) de se criar objetos em JavaScript é usando o que chamamos de notação literal, como no exemplo acima. Um objeto literal é composto por um par de chaves, que envolve uma ou mais propriedades. Cada propriedade segue o formato de "nome : valor" e devem ser separados por **vírgula.**

A diferença entre Array e Objeto é a forma como ambos vão lidar com os dados. **Arrays possuem uma organização de dados mais estruturada**, enquanto **Objetos tratam de forma mais desorganizada** – a posição de um dado não influência seu acesso.

Dot vs Bracket Notation

Há duas formas de acessar uma propriedade em um objeto: usando **dot notation** ou **bracket notation**. Dot nation é usado com mais frequência, mas ambos têm seus objetivos.

Exemplo usando Dot Notation:

```
const descricaoPessoa = {
    nome : 'Felipe',
    sobrenome : 'Miranda Marreiros',
    idade : 24,
    altura : 1.65,
    cor : 'Pardo',
    localizacao : ['Santana', 'Amapá', 'Brasil']
};

//dot notation

console.log(descricaoPessoa.nome); //podemos acessar uma propriedade de um objeto por meio de dot notation, especificando o nome do objeto + o nome da propriedade
```

Para acessar uma propriedade em um objeto, basta dizer o nome do objeto e o nome da propriedade.

Exemplo usando Bracket Notation:

```
const descricaoPessoa = {
    nome : 'Felipe',
    sobrenome : 'Miranda Marreiros',
    idade : 24,
    altura : 1.65,
    cor : 'Pardo',
    localizacao : ['Santana', 'Amapá', 'Brasil']
};

//bracket notation

console.log(descricaoPessoa['sobrenome']); //podemos acessar uma propriedade de um objeto usando bracket notation, ou colchetes, mas diferente de dot notation, precisamos colocar o nome da propriedade em forma de string.
```

Podemos acessar uma propriedade de um objeto usando bracket notation, ou colchetes, mas diferente de dot notation, precisamos colocar o nome da propriedade em forma de string.

Usando variável para acessar propriedades de um objeto:

```
const descricaoPessoa = {
    primeiroNome : 'Felipe',
    sobreNome : 'Miranda Marreiros',
    idade : 24,
    altura : 1.65,
    cor : 'Pardo',
    localizacao : ['Santana', 'Amapá', 'Brasil']
};


const verNomes = 'Nome'; //especificamos uma parte do valor que se repete em forma de string
console.log(descricaoPessoa['primeiro' + verNomes]); //tudo que tem ligação com nome é mostrado no console.
console.log(descricaoPessoa['sobre' + verNomes]);
```

Podemos usar variáveis para acessar uma propriedade de um objeto, desde que essa propriedade tenha particularidade em repetição. Tudo que tem ligação com a variável será acessada por meio de interpolação. Em outras palavras, em um valor que se repete, podemos pegar uma parte dela e usá-lo como referência. **Isso porque, quando nós usamos bracket notation, aqui será permitido usar expressões**. Ou seja, expressões são métodos que geram um valor final. O **contrário não pode ser feito com dot notation, pois o método não aceita expressões.**

Exemplo de quando não sabemos qual propriedade acessar:

```
const interessadoEm = ('O que você quer saber sobre Felipe? Escolha entre primeiroNome, sobreNome, idade, altura, cor, localizacao');

if(descricaoPessoa[interessadoEm]){ //se o usuário digitar as propriedades que existem dentro do objeto descricaoPessoa, JavaScript retornará com a informaçao
    console.log(descricaoPessoa[interessadoEm]);
} else { //caso contrário, se ele digitar outra coisa que não tenho no objeto, ele ficará com a mensagem else.
    console.log('Propriedade inválida. Escolha entre primeiroNome, sobreNome, idade, altura, cor, localizacao');
}
```

Exemplo para adicionar novas propriedades em um objeto:

```
//podemos também adicionar novas propriedades em um objeto com dot ou bracket notations

descricaoPessoa.profissao = 'programador'; //adicionando com dot

descricaoPessoa['situacao'] = 'solteiro'; //adicionando com bracket

console.log(descricaoPessoa);
```

Basicamente colocamos o nome do objeto + o nome da nova propriedade e o valor que ela vai receber.

Objetos e Métodos

Como já observamos, quando usamos objetos podemos inserir propriedades e os mais diversos tipos de estrutura de dados como arrays, variáveis de fora do escopo ou do corpo do objeto, operações matemáticas. Também podemos inserir funções que são chamados aqui de métodos.

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log('Started')
  }
}

car.start()
```

Exemplo de objeto com métodos:

```
28  const friends = ['Amanda', 'Paulo', 'Liana']; //Array
29
30
31  const felipe = { //objeto
32
33      firstName : 'Felipe',
34      lastName : 'Marreiros',
35      birthyear : 1997,
36      occupation : 'Programmer',
37      friends : friends, //array
38      hasDriversLicense : false,
39
40      calcAge : function () {
41          this.age = 2021 - this.birthyear;
42          return this.age;
43      },
44
45      description: function () {
46
47          this.aboutPerson =`${this.firstName + ' ' + this.lastName} is a ${this.calcAge
            ()}-old ${this.occupation}, he ${this.hasDriversLicense === true ? "has a
            driver's license" : "has no driver's license"}.`;
48          return this.aboutPerson;
49      }
50
51  };
```

Podemos usar funções dentro de um objeto da mesma maneira que usaríamos fora dele com todas suas funcionalidades. No exemplo acima, temos um objeto com usando uma **função chamando outra função.**

Funções em Objetos

Quando uma função é incorporada em um objeto, ela passa a se chamar método. No entanto, sua sintaxe muda um pouco. Originalmente podemos declarar uma variável como no exemplo abaixo.

```
function name(params) {

}
```

No entanto, temos que redefinir a sintaxe para encaixar dentro de um objeto, ficando assim

```
    calcPerc : function () {

        this.calcPerc = (this.population / 7900 ) * 100;
        return this.calcPerc;
    },
```

Observação: Arrow Functions não funcionam dentro de um objeto. A razão é... Porque não! Na verdade, ainda está longe da minha compreensão. Aqui é explicado melhor.

Objetos e This – Don't Repeat Yourself

Quando estamos desenvolvendo em JavaScript, temos que encontrar uma forma de não repetir comandos que poderiam ser executados apenas com uma representação do comando. Por exemplo, ao usar o objeto **felipe** para fazer outras operações, no caso, usá-lo em uma função, poderíamos muito bem, ao criar uma nova propriedade na função **description**, usar **felipe.aboutPerson**, no entanto, "teríamos" que repetir o objeto chamado **felipe** por todo código. O objetivo da função é ser chamado em qualquer lugar do código por ser algo flexível.

Se quiséssemos usar a função **description** em outro objeto, teríamos que reescrever o código com novo nome do objeto. Nesse caso, podemos resolver isso apenas usando *this.*propriedadeDoObjeto. "This" representa o objeto como todo, nele podemos consultar todas as propriedades de um objeto que esteja em seu corpo.

Ou seja, **this.aboutPerson** e **felipe.aboutPerson** são a mesma coisa e é aqui que entra o Don't Repeat Yourself, a forma de sincronizar dados sem fazer alterações manualmente.

Outro exemplo usando *this:*

```
describeCountry: function () {

    this.summary = `${this.country} has ${this.population} million people, which is about ${this.
    calcPerc()} of the world.`;
    return this.summary;
}
```

Interação - The For Loop

Iteração é o ato de iterar (repetir) uma função por um determinado período de tempo até que uma condição seja alcançada. Na área de desenvolvimento, usa-se a iteração para alcançar um código mais limpo e um melhor desempenho na aplicação. Quando nos referimos à programação, a iteração está presente na estrutura de dados, como listas, filas ou ainda em qualquer coisa que se repita muitas vezes.

```
//Imagine que precisamos repetir a função log abaixo 10 vezes
console.log('lifting weights repetition 1');
console.log('lifting weights repetition 2');
console.log('lifting weights repetition 3');
console.log('lifting weights repetition 4');
console.log('lifting weights repetition 5');
console.log('lifting weights repetition 6');
console.log('lifting weights repetition 7');
console.log('lifting weights repetition 8');
console.log('lifting weights repetition 9');
console.log('lifting weights repetition 10');

//Poderiamos fazer isso dessa maneira, porém, estariamos quebrando a regra do Don't
Repeat Yourself

//Além disso, por exemplo, se tivessemos um problema de ortografia na mensagem,
teriamos que consertar mensagem por mensagem, o que também quebra a regra do Don't
Repeat Yourself
```

Na imagem acima, temos uma repetição da mensagem com console.log. Podemos repetir o mesmo processo usando apenas duas linhas de códigos, usando o **FOR LOOP.**

Em JavaScript, assim como em outras linguagens de programação, existem diferentes tipos de estruturas de repetição. Trata-se de um recurso muito importante para facilitar a execução de loops, ou seja, quando um ou vários **comandos precisam se repetir por diversas vezes enquanto uma condição for verdadeira**, ou até que um comando de parada seja solicitado.

For determina que uma ação deve ser executada a partir de uma condição inicial até que seja encontrada outra que interrompa o laço. O fluxo de repetição é controlado por uma variável, que é testada a cada repetição até que se encontre a condição de parada.

Exemplo usando For Loop:

```
//Para resolver esse problema, podemos usar o For loop


//O For Loop continua rodando enquanto a condição estabelecida
for verdadeira
for(let rep = 1; rep <= 10; rep++){
    console.log(`lifting weights repetition ${rep}`);
};
```

Sintaxe do For Loop

O For Loop tem três expressões específicas seguidas da declaração do bloco de códigos que serão executados enquanto a condição for verdadeira. As expressões devem ser separadas por ponto e vírgula (semicolon).

- **Expressão Inicial** (**let rep = 1**): corresponde à declaração da variável e atribuição de qual é o valor inicial utilizado. Vale dizer que também pode ser atribuída uma expressão para inicializar a variável. **No exemplo utilizado, o loop começa com 1**. Poderia começar com **0 em caso de Arrays que são zero-based.**
- **Expressão Condicional** (**rep <= 10**): é uma expressão em que é feito um teste booleano para constar se uma condição seja verdadeira, e então, executar o código correspondente ou sair da estrutura de repetição. Quando a condição for falsa, o loop para. **No exemplo utilizado, o loop termina quando chegar em 10.**
- **Atualização da expressão inicial** (**rep++**): é a alteração da variável utilizada na primeira expressão. Sem ela, a repetição não se inicia. Em outras palavras temos **rep = rep + 1.**

Usando Loops com Array

Podemos utilizar arrays com for loop em diversas situações. No exemplo abaixo, podemos usá-lo para mostrar todos os elementos da array no console.

```
const felipeArray = [
    'Felipe',
    'Miranda',
    2021 - 1997,
    'Front-End',
    ['Paulo', 'Jéssica', 'Jeferson']
]

for(let i = 0; i <= felipeArray.length; i++){

    console.log(felipeArray[i]);
}
```

Em vez de usarmos o índice da array manualmente ([0], [1], [2]), podemos simplesmente usar a variável que criamos no início do for loop, no caso i. Ele terá como base a quantidade de elementos que deve operar devido a propriedade length.

Em outras palavras, o loop acima só vai parar quando a quantidade de elementos da array ser alcançado, que no caso será 4. **Vale relembrar que length conta exatamente a quantidade de elementos da array e ele não leva em conta a array ser zero-based.**

Também podemos usar for loop para guardar dados de uma array em outra vazia, como por exemplo:

```
const felipeArray = [
    'Felipe',
    'Miranda',
    2021 - 1997,
    'Front-End',
    ['Paulo', 'Jéssica', 'Jeferson']
]

const guardar = [];

for(let i = 0; i <= felipeArray.length; i++){

    guardar.push(felipeArray[i]);

}

console.log(guardar);
```

Isso tudo pode ser feito graças ao método push() da array. Assim, basta colocar a array que deseja replicar na array vazia.

Outro exemplo usando for com array, desta vez calculando idades:

```
const years = [1997, 2001, 1991, 2010];

const ages = [];

for(let i = 0; i <= years.length; i++){

    ages.push(2021 - years[i]);
}

console.log(ages);
```

Podemos passar operações matemáticas.

For – Continue

Podemos usar **Continue** para sair da iteração do loop e ir para a próxima. Em outras palavras, segundo o W3Schools, Continue é uma forma de "pular por cima" de um loop se uma condição for estabelecida. Por exemplo, se quisermos que um for-loop mostre apenas strings de uma array, podemos fazer isso com Continue, pois ele passará por cima dos outros elementos que não são strings.

```
const felipeArray = [
    'Felipe',
    'Miranda',
    2021 - 1997,
    'Front-End',
    ['Paulo', 'Jéssica', 'Jeferson']
]

const guardar = [];

for(let i = 0; i <= felipeArray.length; i++){

    if(typeof felipeArray[i] !== 'string') continue;

    console.log(felipeArray[i], typeof felipeArray[i]);

};
```

For – Break

Diferente de Continue que passa por cima de uma iteração, Break realmente termina um loop. Em outras palavras, ele encerra a repetição se uma condição for estabelecida. Assim como dito no início, **for** precisa que uma condição seja verdadeira para que permaneça na iteração. Caso a operação chegue ao ponto de ser falsa, ele encerrará naturalmente. Usamos Break para acabar precocemente.

```
const felipeArray = [
    'Felipe',
    'Miranda',
    2021 - 1997,
    'Front-End',
    ['Paulo', 'Jéssica', 'Jeferson']
]

const guardar = [];

for(let i = 0; i <= felipeArray.length; i++){

    if(typeof felipeArray[i] === 'number') break;

    console.log(felipeArray[i], typeof felipeArray[i]);
};
```

No exemplo acima, assim que a condição estabelecida encontrar um número na Array, o for-loop vai ser encerrado imediatamente. Os outros elementos da Array não irão ser executados. Assim, apenas as strings ‘Felipe’ e ‘Miranda’ serão mostradas no console.

Looping Backwards e Loops in Loops

Looping Backwards: Podemos fazer um loop de trás para frente em uma array. Anteriormente, usávamos o índice ([0], [1]) para ter uma direção de como loop deveria se comportar, usando também a propriedade length.

```
const felipe = [
    'Felipe',
    'Miranda',
    2021 - 1997,
    'front-end',
    ['Jéssica', 'Paulo', 'Fernanda']
];

//podemos fazer um loop com uma array de trás pra frente, em vez de seguirmos a ordem do index ([0], [1]).

for(let i = felipe.length - 1; i >= 0; i--){
    console.log(i, felipe[i]);
};
```

Sintaxe: diferente de um loop normal, temos em mente que length diz o tamanho da propriedade, então o índice começará com o tamanho, então devemos especificar a posição do último elemento da array usando **nomeDaArray.length - 1**, e dizer uma condição que, enquanto o índice for maior ou igual a zero **(i >= 0)**, o loop continuará rodando até chegar em 0 (que nesse caso será o primeiro índice da array). Para que o loop comece o seu processo, temos que atualizar a variável i como no exemplo acima (i--). Desta forma, temos **i = i – i**.

```
4 ▶ (3) ['Jéssica', 'Paulo', 'Fernanda']
3 'front-end'
2 24
1 'Miranda'
0 'Felipe'
>
```

Loops in Loops: podemos usar loops dentro de loops.

```
//looping in loops

for(let exercicio = 1; exercicio < 4; exercicio++){ //este loop inicia uma vez a cada 5 carregamentos de pesos (segundo loop);
    console.log(`Começando exercicio ${exercicio}`);

    //a cada inicio de exercicio, este loop será executado (primeiro loop)
    for (let repeticao = 1; repeticao < 6; repeticao++){
        console.log(`Repetição de carregamento de peso ${repeticao}`);
    }

}
```

Imagine você, Felipe do futuro, que precise que uma condição x dependa de uma condição y para iniciar um processo. Bem, esta é uma boa oportunidade para usar loop in loop. Também chamado de nested loop. Então, teremos como resultado:

```
Começando exercicio 1
Repetição de carregamento de peso 1
Repetição de carregamento de peso 2
Repetição de carregamento de peso 3
Repetição de carregamento de peso 4
Repetição de carregamento de peso 5
Começando exercicio 2
Repetição de carregamento de peso 1
Repetição de carregamento de peso 2
Repetição de carregamento de peso 3
Repetição de carregamento de peso 4
Repetição de carregamento de peso 5
Começando exercicio 3
Repetição de carregamento de peso 1
Repetição de carregamento de peso 2
Repetição de carregamento de peso 3
Repetição de carregamento de peso 4
Repetição de carregamento de peso 5
> 
```

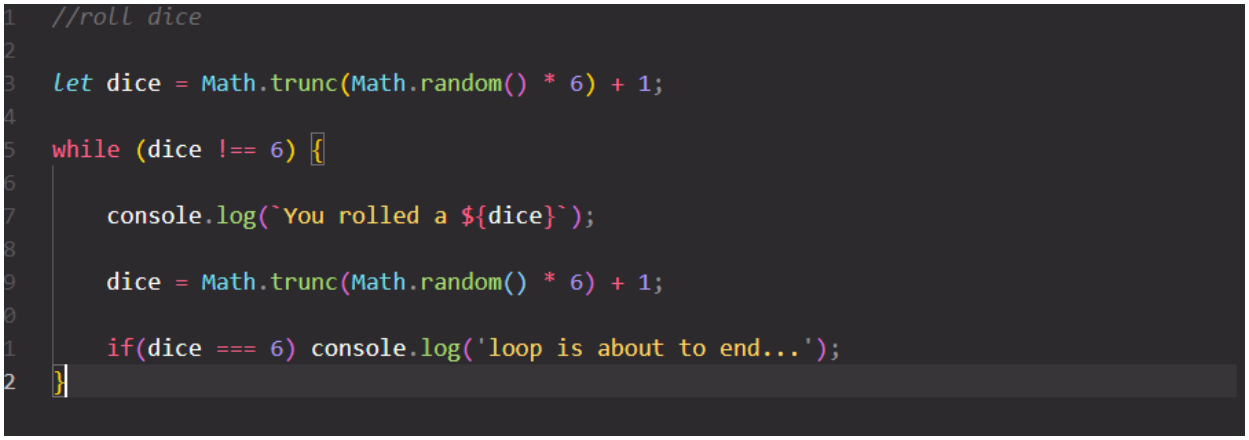
While Loop

While tem uma sintaxe diferente de For, uma vez que separamos as expressões do loop por parte, por exemplo:

```
//for loop

for (let rep = 1; rep <= 10; rep++) {

    console.log(`Lifting weights repetition ${rep}`);
};

//while lopp
let rep = 1;

while(rep <= 10){

    console.log(`Lifting weights repetition ${rep}`);

    rep++;
};
```

Deixamos a variável fora da função e apenas usamos a expressão booleana dentro dos parênteses. A atualização da variável fica no escopo dela.

Diferente de For, While faz um pré-teste antes de executar uma operação. A cada momento que While avalia uma iteração, ele avalia se é verdadeira ou falso. Caso seja verdadeira, o loop prossegue. Caso seja falso, o loop acaba, como no exemplo abaixo:



```
//roll dice

let dice = Math.trunc(Math.random() * 6) + 1;

while (dice !== 6) {

    console.log(`You rolled a ${dice}`);

    dice = Math.trunc(Math.random() * 6) + 1;

    if(dice === 6) console.log('loop is about to end...');
}
```

No exemplo do dado, se o valor for diferente de 6, então a mensagem continuar rodando (verdadeiro). Se por algum momento o dado for 6, então o loop acaba e a mensagem é apresentada. Ou seja, todos os números antes de 6 vão ser testados com base na condição imposta **(dice !== 6)**.