PROGRAMAÇÃO WEB CLIENTE

UFCD(s) 5414

GUIA DE LABORATÓRIO 3.1
DECISÕES, CICLOS E FUNÇÕES (ßeta)

OBJECTIVOS

- Conhecer os mecanismos principais da linguagem JavaScript para controlar o fluxo da execução.
- Aprender as decisões *`if`* e *`switch`*, os ciclos *`while`*, *`for`*, *`for-of`* e *`for-in`*.
- Aprender a definir funções simples

INSTRUÇÕES

PARTE I – DECISÕES IF E SWITCH

1. Aceda ao REPL do navegador ou do Node.js.

***NOTAS:** 1) Para vários dos exemplos que se seguem não é exibida a linha de comandos da consola. Pode introduzir esses exemplos linha-a-linha na consola, mas é mais conveniente escrever o código num ficheiro `.js` (um script) e depois executá-lo via Node ou através de uma página HTML com o elemento* **`script`** *apropriado. 2) Todos os exemplos que utilizam as funções `prompt` e `alert` devem ser executados no navegador.*

Os computadores seleccionam, guardam, acedem, percorrem e, de um modo geral, manipulam os dados que armazenam. Esta manipulação envolve tomadas de decisões e tarefas repetitivas que possuem um início e um fim. Um programa de software, escrito numa qualquer linguagem de programação, especifica essas actividades de manipulação e a ordem pela qual acontecem. Nesse sentido, um programa procura controlar a execução dessas operações. Controlar significa determinar que instruções devem ser executadas em cada instante, e qual a ordem pela qual devem ser executadas.

2. Pretendemos indicar se um utilizador é menor ou maior de idade em função da sua idade:

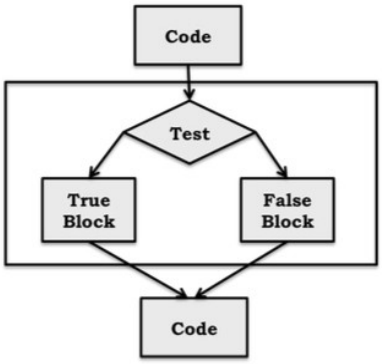
```
const idade = parseInt(prompt("Idade? "), 10);
if (idade >= 18) {
    alert("Maior de idade");
}
else {
    alert("Menor de idade");
}
```

Este pedaço de código deve ser executado no ***navegador*** *(na consola ou embebido num página html), uma vez que só aí possui acesso às funções `prompt` e `alert`.*

Tipo	Membros / Exemplos
Decisão	`if-else  switch  operador ternário`
Ciclo	`while  do-while  for  for-of  for-in`
Transferência de Controlo	`return  break  continue try-catch-finally  throw`
Invocação de funções/métodos	`parseInt("2") console.log("Ok") Math.pow(2, 3)  etc.`
Iteradores e Geradores	`[Symbol.iterator] gerador.next()/.send() function* yield yield*`

Tabela não-exaustiva com mecanismos (síncronos) para controlo da execução em JavaScript

Os exemplos que vimos até agora foram bastante lineares na sua execução. Cada instrução é executada pela ordem em que é introduzida. A instrução ***if*** *permite "ramificar" o fluxo de execução em duas partes mediante o resultado de uma* ***expressão booleana*** *que, no contexto do* ***if****, designamos por* ***condição****.*
Neste exemplo, se a condição for verdadeira, isto é se idade for superior ou igual a 18, a mensagem `Maior de idade` *é exibida. Senão (****else****), é exibida a mensagem* `Menor de idade`*.*
Já agora, a instrução ***if*** *é uma instrução* ***composta*** *pois necessita de mais do que uma (sub)instrução (os dois prints) para ficar completamente definida.*



Fluxograma a ilustrar o fluxo de execução da decisão IF-ELSE

O formato geral do ***if*** *é o seguinte:*

```
if (expressão_booleana) {
    bloco_código_expressao_verdadeira
}
else {
    bloco_código_expressão_falsa
}
```

A indentação, apesar de opcional, é importante de um ponto de vista da leitura do programa. Alinhámos cada um dos blocos para indicar que são executados se a cláusula if ou else forem seleccionadas.
Sintaticamente, JavaScript é uma linguagem que descende da linguagem C. Em linguagens como C, blocos de código são delimitados com um determinado símbolo (eg, neste caso são as chavetas { e }) ou palavra-reservada (eg, em Pascal ou Ruby, podem ser as palavras ***begin*** *e* ***end****).*

3. Vamos solicitar ao utilizador um valor que é suposto ser preço sem IVA de um produto. Se o tipo de produto for "Alimentação" o IVA a aplicar deverá ser 6%, se for "Higiene" deverá ser 13%, caso contrário deverá ser 23%. Depois calculamos e exibimos o montante de IVA e o preço final . Escreva o seguinte código:

```
const preco = parseInt(prompt("Preço do produto? "), 10);
const tipoProd = prompt("Tipo do produto? ");
let IVA;
```

```
if (tipoProd.toUpperCase() === 'A') {
    IVA = 6;
}
else if (tipoProd.toUpperCase() === 'H') {
    IVA = 13;
}
else {
    IVA = 23;
}
console.log(
    `Montante de IVA: ${(preco*(IVA/100)).toFixed(2)} €  (IVA: ${IVA.toFixed(2)}%)
    Preço Final: ${(preco*(1 + IVA/100)).toFixed(2)} €\n`
)
```

*Como vimos no laboratório anterior, a instrução **if** é uma instrução condicional designada por decisão. Em geral, se a condição de um **if** for verdadeira, ie, se a expressão lógica que sucede a palavra-reservada "**if**" produzir o valor booleano **true**, então a instrução ou bloco de instruções que sucede o **if** é executado. Se a condição for falsa, então duas coisas podem acontecer: se a instrução **if** possuir uma parte (designada por "cláusula") que se inicie com a palavra reservada **else**, então o bloco de instruções desse **else** é executado em vez do bloco de instruções do **if**. No fundo estamos a dizer algo como "se isto for verdade então faz aquilo, senão faz outra coisa qualquer". Se não utilizarmos a cláusula **else** (é opcional) então, o programa prossegue na instrução que sucede a toda instrução composta **if**.*
*Suponhamos que alguém projecta as seguintes alternativas para um dia de férias: se estiver sol vai à praia, se estiver encoberto vai para o campo e se estiver a chover vai ao mercado. Temos aqui um conjunto de três alternativas que dependem de três condições. Em JavaScript, uma alternativa condicional indica-se com **else if**. Não existe limite para o número de alternativas condicionais. Note-se que a cláusula **else** é uma alternativa incondicional dado que é sempre executada caso a condição do **if** seja falsa.*
*Resumindo, podemos utilizar o **if** para tomar mais do que duas decisões. Neste caso utilizamos o formato **if** `(cond1){ ... }` **else if** `(cond2) ...` **else if** `(condN) {...}` **else** `{...}`. Note-se que o bloco **else** não é obrigatório. Cada uma das cláusulas **else ifs** é designada por alternativa condicional, ao passo que a cláusula **else** é designada por alternativa incondicional.*

4. Poderíamos também ter utilizado um ***switch-case*** para determinar o montante de IVA:

```
let IVA;
switch(tipo) {
   case "A"
      IVA = 6;
      break;
   case "H":
      IVA = 13;
      break;
   default:
      IVA = 23;
      break;
 }
```

*A instrução **switch-case** é uma estrutura de **decisão** baseada no valor de uma expressão. Está implícita uma comparação com **===** e, como tal, não necessitamos de escrever explicitamente as condições de comparação de igualdade.*
*O **switch** é uma estrutura de decisão múltipla que testa se uma **expressão** produz um valor **===** a um dos valores presentes na lista de **case**s. Se for o caso, o bloco de instruções associado a esse **valor constante** é executado.*

O formato geral de um **`switch`** *é:*

```
switch (expressão) {
  case valor1: conjunto_de_instruções1
  ...
  case valorN: conjunto_de_instruçõesN
  [default: conjunto_de_instruções_default]
}
```

Tipicamente, os valores `valor1, ..., valorN` *tendem a ser* ***`number`*** *ou* ***`string`****. Se um destes valores for* `===` *ao valor de* `expressão` *então o* `conjunto _de_instruções` *correspondente é executado. O* `conjunto_de_instruções` *correspondente à cláusula* ***`default`*** *é executado se nenhum dos valores dos vários cases for igual ao valor da* `expressão`*. A cláusula* ***`default`*** *é opcional. No exemplo anterior fomos obrigados a terminar cada* **`case`** *com um* ***`break`****, caso contrário o código continuaria para o* **`case`** *de baixo.*
Consultar: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch

5. Passemos para outro mecanismo de selecção. Introduza agora o seguinte código no REPL:

```
>>> let x = 19
```

6. Agora pretendemos inicializar a variável `y` em função do valor de `x`, de acordo com a seguinte lógica: se `x` for superior a 10, `y` deverá ter `'sup'`, caso contrário deverá ter o valor `'inf'`:

```
>>> let y;
>>> if (x > 10) {
...   y = 'sup';
... } else {
...   y = 'inf';
... }
'sup'
>>> y
'sup'
```

7. Alternativamente pode utilizar o operador ternário para introduzir uma expressão condicional:

```
>>> y = x > 10 ? 'sup' : 'inf';
'sup'
```

8. Vamos agora inicializar o valor da variável `z`, que deverá ser `'sup'` se `x > 15`, `'interm'` se `10 <= x <= 15` e `'inf'` caso contrário:

O ***operador ternário*** *ou* ***expressão condicional*** *é um tipo de* ***`if`*** *para o caso, muito comum, em que queremos atribuir ou devolver um de dois valores consoante o resultado de uma condição. Por exemplo:*

```
let x = ... um valor numérico ...;
// ...
let msg;
if (x % 2 === 0) {
  msg = "X é par";
} else {
  msg = "X é ímpar.";
}
```

O pedaço de código `msg =` *encontra-se repetido nos dois ramos do* ***`if`****. O operador ternário permite simplificar a inicialização da string* `msg`*:*

```
int x;
// ...
let msg = (x % 2 === 0 ? "X é par" : "X é ímpar.");
```

O operador ternário possui três partes correspondentes aos seus três operandos:

```
CONDIÇÃO ? CONSEQUÊNCIA : ALTERNATIVA
```

`CONDIÇÃO` *é uma expressão booleana, ao passo que* `CONSEQUÊNCIA` *e* `ALTERNATIVA` *são expressões que é habitual (mas não é obrigatório) devolverem valores do mesmo tipo de dados.*

```
>>> let z = x > 15 ? 'sup' : x >= 10 ? 'interm' : 'inf';
```

9. Apesar não ser necessário, há quem prefira utilizar parênteses para "destacar" a ordem de avaliação:

```
>>> let z = (x > 15 ? 'sup' : (x >= 10 ? 'interm' : 'inf'));
```

PARTE II – CICLOS E CICLO WHILE

10. Pretendemos exibir todos os números pares de 0 a 10. Introduza no REPL:

```
>>> const log = console.log
>>> log(0)
0
>>> log(2)
2
>>> log(4)
4
>>> log(6)
6
>>> log(8)
8
>>> log(10)
10
```

O objectivo de programar passa por "dar ordens" ao computador para ele fazer o trabalho repetitivo por nós. Ou seja, passa por poupar trabalho. Exibir os números pares de 0 a 10 um-a- um, é repetitivo mas "faz-se". Fazer o mesmo para um conjunto mais alargado de números, como, por exemplo, 1000, é impraticável.
Uma das vantagens dos computadores é a sua capacidade de processar grandes volumes de dados em pouco tempo. Se o tempo que se poupa no processamento for utilizado na escrita do código, então não ganhámos eficiência. Por outro lado, por vezes queremos processar 10 números (como aqui), mas, noutras ocasiões, queremos que o mesmo programa processe 100 números, e noutras ocasiões ainda, podemos estar interessados em apenas 5 números. Ou seja, é conveniente que o mesmo "pedaço" de programa se adapte ao "volume" da informação a tratar. É para isso que servem os ciclos....=

11. Agora faça o mesmo mas para todos os números pares de `50` a `1750`.

NOTA: Não, não faça isto!! Era a brincar.

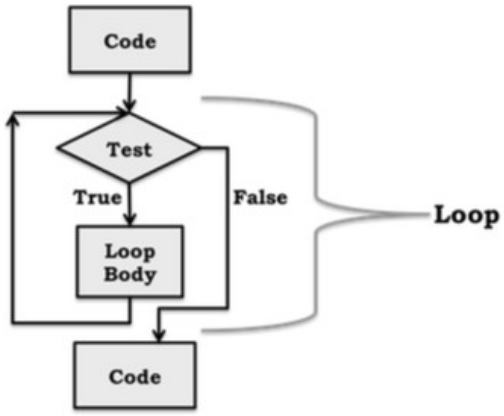
12. Escreva o seguinte bloco de código no REPL:

```
>>> let numero = 0;
>>> while (numero <= 10) {
...   log(numero);
...   numero = numero  + 2;
... }
```

Uma instrução que comece com a palavra reservada **`while`** *define um ciclo. Um ciclo é uma instrução que quebra a habitual "sequencialidade" do programa. Pode levar que múltiplas instruções sejam repetidas várias vezes. A palavra* **`while`** *é seguida de uma expressão lógica, designada por condição, envolvida em parênteses. Esta condição vai determinar se o ciclo continua, isto é, se as instruções entre chavetas são executadas, ou não (...)*

(...) Enquanto a condição for verdadeira, o bloco de instruções do **`while`** *(as tais instruções entre chavetas) é executado. Caso contrário, isto é, quando o resultado da condição for* **`false`**, *o programa "prossegue" a seguir à chaveta que fecha o bloco de instruções. Neste exemplo, utilizamos a variável* **`numero`** *para manter o número par actual que estamos a exibir. Assim que exibimos um número par, passamos para o seguinte incrementando-o em duas unidades. Entretanto o fluxo de execução "bate" na chaveta e a condição entre parênteses volta a ser testada. Em sendo verdade, isto é, enquanto o número for inferior ou igual a 10, as instruções* `log(numero)` *e* `numero = numero + 2` *são executadas.*

13. Volte a executar o mesmo código mas dando o valor inicial `50` à variável numero e substituindo `10` por `1750`.



1a forma de repetição (loop): O ciclo `while`

O ciclo **`while`** *é semelhante a um* **`if`**, *com a diferença que o bloco de instruções do ciclo é executado enquanto a condição for verdadeira.*
Este tipo de problemas resolve-se melhor com o ciclo **`for`**, *conforme veremos a seguir:*

14. Normalmente, o código em cima escreve-se (por questões estilísticas) da seguinte forma:

```
let i = 0;
while (i <= 10) {
   log(i);
   i += 2;
}
```

*É tradição dar o nome "i" a variáveis que controlam ciclos que percorrem gamas de números. Neste caso, a variável i substitui a variável numero. Por outro lado, os operadores +=, *= e restantes operadores de atribuição simplificam a escrita do código. Por exemplo, escrever* `valor +=10` *é uma forma abreviada de escrever,* `valor = valor + 10`.

15. Agora vamos fazer um programa que solicita um número ao utilizador e indica o resultado da soma de todos os números até esse número (na verdade, existe uma fórmula que permite obter directamente o resultado da soma). Introduza:

```
let num = parseInt(prompt("Indique um numero: "));
let soma = 0;
let i = 1
while (i <= num) {
       soma +=  i;
       i += 1;
}
log("Resultado:", soma);
```

NOTA*: Nos laboratórios seguintes veremos outros modos (alguns melhores) de resolver este tipo de problemas.*

16. Introduza este outro exemplo no editor:

```
let nome = "";
```

```
while (nome !== "Alberto") {
   nome = prompt("Qual o seu nome?");
}
alert("Fim!");
```

17. Caso não seja necessário reter o valor do nome numa variável, então podemos escrever:

```
while (prompt("Qual o seu nome?") !== "Alberto");
alert("Fim!");
```

PARTE III – CICLOS FOR E FOR-OF

18. Para escrever ciclos de contagem, ie, ciclos onde uma variável percorre uma gama de números seguindo uma ordem pré-definida, é mais apropriado utilizar o ciclo ***for***.

```
for (let i = 0; i <= 10; i += 2) {
   console.log(i);
}
```

O ciclo ***for*** *é um ciclo com quatro partes: inicialização, condição, actualização e bloco de instruções. O seu formato geral é:*

```
for  (INICIALIZAÇÃO; CONDIÇÃO; ACTUALIZAÇÃO) {
     BLOCO_DE_INSTRUÇÕES
}
```

A ***inicialização*** *é executada apenas uma vez antes do ciclo se iniciar. Depois, à semelhança do que sucede com o* **while**, *o* ***bloco de instruções*** *é executado enquanto que a condição for verdadeira. Após o bloco de instruções ter sido executado, a* ***actualização*** *é também executada e a condição volta a ser testada. Se for falsa, o fluxo de execução continua a seguir ao* **for**, *senão o bloco de instruções é repetido.*

19. Vamos desenvolver código em JavaScript para obter o nome de um utilizador e exibir esse nome na vertical. Como não sabemos a dimensão do nome, vamos ter que utilizar um ciclo.

```
let nome = prompt("Como se chama? ");

let i = 0;
while (i < nome.length) {
    console.log(nome[i]);
   i += 1;
}
```

Teste o código com o nome `Alberto`.

Relembrando, uma string é uma sequência de caracteres numerada a partir de 0. A variável i é aqui utilizada para armazenar a posição (ie, o número de ordem) dos caracteres. Se o utilizador inserir `Armando`*, então:*

```
i === 0 => nome[i] === 'A'
i === 1 => nome[i] === 'R'
...
```

O método **console.log** *acrescenta uma nova linha e daí o efeito vertical.*

20. Vamos agora utilizar o ciclo ***for-of***. Escreva o seguinte código:

```
let nome = prompt("Como se chama? ")
for (let car of nome) {
       log(car)
}
```

*Em JavaScript, o ciclo **for-of** permite aceder a todos os elementos de uma colecção (na verdade, a partir qualquer a partir de qualquer objecto ou função a patir do qual se possa obter um iterador, tópico a abordar num futuro laboratório). Noutras linguagens, este ciclo é designado de foreach ("para cada"), nome que é mais apropriado para a semântica da instrução. O formato geral do **for-of** é:*

```
for (let VAR of COLECÇÃO) {
  BLOCO_DE_INSTRUÇÕES
}
```

*Enquanto que o ciclo **for** é um ciclo geral, semelhante ao **while**, e mais utilizado para percorrer gamas de valores em progressão, o **for-of** percorre os itens da colecção pela ordem pela qual eles estão dispostos nessa colecção, não sendo necessário definir indíces para aceder aos elementos inviduais.*

21. Queremos agora desenvolver um programa para calcular a média de números passados introduzidos através de um campo de texto num formulário de uma página HTML. Os números devem ser separados por um ou mais espaços. Vamos ignorar a validação dos números (ou seja, vamos assumir que o utilizador introduz realmente números válidos). Obtenha o ficheiro `media.zip` que contém a página já feita em HTML e CSS, e um script JavaScript com parte do código que vamos desenvolver.

22. Vamos terminar a função `showAverage` utilizando os ciclos ***for***, primeiro, e depois ***for-of***. Comece por acrescentar o seguinte código à definição de `showAverage`:

```
function showAverage() {
    const numbersString = document.getElementById('numbers').value.trim();
    const values = numbersString.split(/\s+/);

    // COMPLETAR A PARTIR DAQUI

    let soma = 0;
    for (let i = 0; i < values.length; i += 1) {
        soma += parseFloat(values[i]);
    }
    alert(`
       Soma  : ${soma}
       Média : ${(soma/values.length).toFixed(2)}`
    );
}
```

23. Vamos agora desenvolver a versão para o ciclo ***for-of***. Substitua o ciclo ***for*** pelo seguinte:

```
    let soma = 0;
    for (let value of values) {
        soma += parseFloat(value);
    }
```

E volte a testar.

*Podemos e, na maioria dos casos, devemos utilizar ciclos **for-of** para percorrer a maioria das estruturas de dados em JavaScript moderno. Código com ciclos **for-of** está sujeito a menos erros e é, em geral, mais legível e elegante.*

24. Podemos utilizar ciclos ***for-of*** para percorrer a maioria das estruturas de dados em JavaScript moderno. Isto permite escrever código mais legível, e menos sujeito aos erros clássicos que ocorrem com maior frequência quando utilizamos um ciclo ***while*** ou ***for*** tradicional (erros lógicos na definição das condições de fim de ciclo, erros na actualização de indíces, etc.). Vejamos alguns (pequenos) algoritmos com ***for-of***:

Arrays

```
let nomes = ['Alberto', 'Armando', 'Arnaldo'];
for (let nome of nomes) {
    console.log(nome);
}

for (let nome of nomes.slice().reverse()) {
    console.log(nome);
}

for (let [i, nome] of nomes.entries()) {
    console.log(`${i} => ${nome}`);
}

let pontos3D = [
    {x: 20, y: 10, z: 30},
    {x: -1, y: 5.2, z: 10},
    {x: -1.7, y: -2, z: -3},
];
for (let {x, y, z} of pontos3D) {
    console.log(`X: ${x}   Y: ${y}   Z: ${z} `);
}
```

Strings

```
let txt = 'Alberto';
for (let ch of txt) {
    console.log(ch);
}

for (let [i, ch] of [...txt].entries()) {
    console.log(`${i} => ${ch}`);
}
```

```
for (let ch of [...txt].reverse().join('')) {
    console.log(ch);
}
```

Objectos

```
let pessoa = {nome: 'Alberto', apelido: 'Antunes', idade: 27};
for (let [key, value] of Object.entries(pessoa)) {
    console.log(`${key} => ${value}`);
}

for (let key of Object.keys(pessoa)) {
    console.log(`${key} => ${pessoa[key]}`);
}

for (let value of Object.values(pessoa)) {
    console.log(`valor de propriedade => ${value}`);
}
```

Mapas

```
let pessoas = new Map([
    [112301, {nome: 'Alberto', apelido: 'Antunes', idade: 27}],
    [892714, {nome: 'Armando', apelido: 'Almeida', idade: 41}],
    [473331, {nome: 'António', apelido: 'Aveleda', idade: 22}],
]);
for (let [id, pessoa] of pessoas) {
    console.log(`${id} => ${pessoa.nome}`);
}

for (let id of pessoas.keys()) {
    console.log(`ID: ${id}`);
}

for (let pessoa of pessoas.values()) {
    console.log(`${pessoa.apelido}, ${pessoa.nome}`);
}
```

Gamas de valores

```
for (let i of [...Array(5).keys()]) {
    console.log(i, 5 - i - 1);
}

for (let i of _.range(5)) {      // _.range definida na biblioteca lodash
    console.log(i, 5 - i - 1);
}

for (let i of _.range(5, 11)) {
    console.log(i);
```

```
}

for (let i of _.range(5, 11, 2)) {
    console.log(i);
}

for (let i of _.range(10, 4, -1)) {
    console.log(i);
}

for (let i of _.range(10, 4, -1)) {
    console.log(i);
}
```

Gamas de valores + sequências

```
let codigo = 'A8B2C0';
for (let i of _.range(0, codigo.length, 2)) {
    console.log(codigo[i]);
}

let mensagem = "emrahc met tpircsavaj";
for (let i of _.range(mensagem.length - 1, -1, -1)) {
    console.log('.'.repeat(i), mensagem[i]);
}
```

Contagens

```
let frutas = ['abacate', 'nêspera', 'marmelo', 'diospiro',
              'marmelo', 'diospiro', 'abacate', 'diospiro'] ;
let contadores = new Map();
for (let fruta of frutas) {
    let n = contadores.get(fruta) || 0;
    contadores.set(fruta, n + 1);
}
console.log(contadores);
```

Agrupar

```
let frutas = ['abacate', 'nêspera', 'marmelo', 'diospiro', 'pera', 'anona'];
let grupos = new Map();
for (let fruta of frutas) {
    let grupo = grupos.get(fruta.length) || [];
    grupo.push(fruta);
    grupos.set(fruta.length, grupo);
}
console.log(grupos);

let nomes = ['Alberto', 'Armando', 'Arnaldo'];
let idades = [19, 12, 30, 24];
```

```
for (let [nome, idade] of _.zip(nomes, idades)) {
    console.log(`${nome} -> ${idade}`);
}
```

PARTE IV – BREAK E CONTINUE

25. O ciclo seguinte termina sem que a condição fique falsa:

```
for (let i = 1; i <= 10; i++) {
    if (i % 3 === 0) {
        break;
    }
    console.log(i);
}
```

26. E, agora, acrescente o seguinte ciclo e observe que números são exibidos:

```
for (let i = 1; i <= 10; i++) {
    if (i % 2 === 0) {
        continue;
    }
    console.log(i);
}
```

No âmbito dos ciclos, as palavras reservadas ***break*** *e* ***continue*** *permitem transferências de controlo directas para determinados "locais". Assim, a instrução* ***break*** *termina o ciclo que envolve directamente esta instrução. A próxima instrução a ser executada após um* ***break*** *é a instrução que surge após este ciclo interrompido.*
O ***continue*** *passa o controlo para a próxima iteração de um ciclo, saltando por cima do código que resta executar nesta iteração.*
O ***break*** *e o* ***continue*** *podem ser utilizados com todos as instruções de ciclo do JavaScript.*
Quando utilizamos o ***continue*** *com um ciclo* ***for*** *tradicional a instrução seguinte ao* ***continue*** *corresponde à actualização do ciclo* ***for****. Neste exemplo, o* `i++` *é executado após o* ***continue****.*

27. Queremos que um programa indique se um determinado número introduzido pelo utilizador é um número primo. Introduza o seguinte código:

```
let n = parseInt(prompt("número> "));
let semFactores = true;
let x;
for (x of _.range(2, n)) {
    if (n % x === 0) {
        semFactores = false;
        break;
    }
}
console.log(n, semFactores ? 'é primo' : `é igual a ${x} * ${n/x}`)
```

PARTE IV – FUNÇÕES: BREVE INTRODUÇÃO

28. Temos utilizado várias funções e métodos (`parseInt`, `parseFloat`, `isFinite`, `String.split`, `Array.join`, ...), mas agora queremos saber como definir uma nova função. Por exemplo, vamos

definir uma função para calcular e devolver um montante com IVA. Para tal, a função recebe o montante sem IVA bem como a taxa de IVA a aplicar:

```
function comIVA(valor, taxaIVA) {
    return valor * (1 + taxaIVA/100);
}

let montante = parseFloat(prompt("Indique um montante: "));
console.log(comIVA(montante, 23).toFixed(2));
```

*Uma **função** é um bloco de código com um nome. Este blogo de código pode depois ser reutilizado várias vezes através da invocação desse nome. Para definir uma função utilizamos a palavra-reservada **`function`**:*

```
function fun(param1, ... , paramN) {
    inst1;
    ...
    instN;
}
```

*`fun` é o **nome** ou **identificador** da função. `param1`, ..., `paramN`, entre parêntesis, constitui a **lista de parâmetros** da função. **Parâmetros** são variáveis **locais** à função e servem para receber informação que vem de fora da função. Uma função pode não definir parâmetros. Para devolver valores para fora da função, a função utiliza a palavra-reservada **`return`**. A instrução **`return`** termina imediatamente a função e devolve o resultado das expressões à direita. Uma função é invocada fazendo*

```
fun(arg1, ..., argN)
```

***Argumentos** são os valores dos parâmetros, e devem ser indicados entre parênteses. No exemplo anterior, os argumentos de `comIVA` são a o valor da variável `valor` e o literal 23. `prompt` e `parseFloat` são também funções. No exemplo anterior, são **invocadas** recebendo como argumentos o texto `"Indique um montante"` e o texto efectivamente introduzido pelo utilizador, respectivamente.*
*Um **método** é uma função que é invocada "num contexto" especial, o contexto de um objecto. Por seu turno, um **objecto** é um "valor" que agrega propriedades. Algumas dessas propriedades podem ser funções. Acontece que, quando agregadas a um objecto, estas funções tendem a ser designadas por métodos. Ou seja, um método é uma função definida dentro de um objecto, sendo este objecto o seu argumento principal.*

Por esta altura, é natural que algumas destas noções relacionadas com métodos e classes pareçam demasiado abstractas e até confusas. O tempo e a prática vão ajudar a clarificá-las.
De um ponto de vista prático, podemos olhar para um método como uma função que se invoca da seguinte forma:

```
arg1.met(arg2, ..., argN)
```

`arg1` é o objecto e `met` é um método que é também uma propriedade desse objecto. Quando dermos objectos e classes, tudo isto vai fazer mais sentido. Até lá, necessitamos apenas de saber que determinadas funções são invocadas (ie, utilizadas) indicando em primeiro lugar o nome da função e depois a lista de argumentos. Existe uma outra categoria de funções, os métodos, que devem ser invocados escrevendo em primeiro lugar um objecto (o argumento principal da função), seguido do operador de acesso . (ponto), seguido do nome do método, seguido da lista de argumentos com os restantes argumentos.

29. Uma função pode devolver dois valores. Por exemplo, a função anterior poderia devolver o montante final bem como o próprio montante de IVA:

```
function comIVA(valor, taxaIVA) {
    let montanteIVA = valor * (taxaIVA/100);
    return [valor + montanteIVA, montanteIVA];
}

const taxaNormal = 23;     // em %
let preco = parseFloat(prompt("Indique um preço: "));

let [precoFinal, montanteIVA] = comIVA(preco, taxaNormal);

console.log(`Preço Final: ${precoFinal.toFixed(2)} ` +
            `IVA: ${montanteIVA.toFixed(2)} (${taxaNormal.toFixed(2)} %)`);
```

30. Em cima vimos um exemplo de uma função que calcula o resultado de uma fórmula (um montante com IVA). Vamos agora ver alguns exemplos de funções que permitem identificar se os argumentos possuem determinadas propriedades. Estas funções irão devolver ***true***, se o objecto verificar a característica a determinar, caso contrário devolvem ***false***.

```
// é um número par?
function isEven(num) {
    return num % 2 === 0;
}


// é um número ímpar?
function isOdd(num) {
    return num % 2 === 1;
}

// é um número primo?
function isPrime(num) {
    if (!Number.isInteger(num) || num < 2) {
        return false;
    }
    for (let x of _.range(2, num)) {
        if (num % x === 0) {
           return false;
        }
    }
    return true;
}
```

```
// é uma letra (ocidental)?
function isLetter(ch) {
   return ch.length >= 1
       && (ch[0] >= 'A' && ch <= 'Z'
           || ch[0] >= 'a' && ch <= 'z');
}
// é um caractere de espaçamento?
function isWhiteSpace(ch) {
    return ch.length >= 1
        && ch[0].trim().length === 0;
}
// é uma vogal?
function isVowel(ch) {
    return ch.length >= 1
        && "aeiou".includes(
                ch[0].toLowerCase());
}
```

```
// é uma password válida? isto é, tem pelo menos 6 caractares, uma minúscula,
// uma maiúscula, um dígito e um símbolo?
```

```
function isValidPassword(pwd) {
    const letras = String.fromCharCode(..._.range('a'.charCodeAt(), 'z'.charCodeAt() + 1));
    const LETRAS = letras.toUpperCase();
    return pwd.length >= 6
        && _.intersection([...pwd], [...letras]).length !== 0
        && _.intersection([...pwd], [...LETRAS]).length !== 0
        && _.intersection([...pwd], [...'0123456789']).length !== 0
        && _.intersection([...pwd], [...'#$!%']).length !== 0;
}
```

31. Teste agora:

```
>>> [isEven(12), isOdd(13)]
[ true, true ]
>>> [isOdd(12), isEven(13)]
[ false, false ]
>>> [isPrime(1), isPrime(9)]
[ false, false ]
>>> [isPrime(11), isPrime(10163)]
[ true, true ]
```

```
>>> [isLetter('a'), isLetter('B')]
[ true, false ]
>>> [isLetter('#'), isLetter('9'), isVowel('a')]
[ false, false, true ]
>>> [isWhiteSpace(' '), isWhiteSpace('\n')]
[ true, true ]
>>> isValidPassword('Xcv12$')
true
```

Funções booleanas como estas são também designadas de predicados. São muito úteis para acedermos a uma parte muito importante da linguagem JavaScript: a programação funcional.Teremos um laboratório sobre funções e outro sobre objectos onde também abordaremos funções.
Estas funções, além de úteis para validar informação, permitem um tipo de programação que dispensa ciclos, uma potencial fonte de bugs (*), para percorrer, filtrar, seleccionar ou resumir (entre outras funcionalidades) colecções de elementos. Por exemplo, teste (resultados omitidos):

```
>>> let nums = [10, 5, 8, 20, 3, 110, 10163]
>>> nums.find(isOdd)
>>> nums.find(isEven))
>>> nums.filter(isPrime)
>>> nums.every(isOdd)
>>> nums.some(isOdd)
```

EXERCÍCIOS DE REVISÃO

1. Com que valores ficam as variáveis nas seguintes atribuições:

1.1 `c = [..."ABCDEFGHIJKLMNOPQRSTUVWXYZ"].join('.')[10]`

1.2 `x = [{a: 1, b: 2}, {a: 10, b: 20}].slice(0, -1)[0]['b']`

1.3 `g = 'g'.charCodeAt() - 'f'.charCodeAt()`

1.4 `r = 4 * 3 ** 2`

1.5 `s = _.intersection([..."armanda"], [...['igreja', 'cidade'][{a: 0, b: 1}['b']]])`

1.6 `b1 = 0 !== 1  ? 72 : 99`

1.7 `b2 = (0 === 1 ? 44 : ('a' < 'A' ? 33 : 22))`

1.8 `nome = "ALBERTO".toLowerCase().slice(2, -2).split('').join('/')`

2. Escreva um ciclo *for-of* para exibir os números pares de 0 a 100 excepto os múltiplos de 21. Pode utilizar mecanismos de lodash.

3. O que é exibido pelas seguinte instruções? A não ser quando realmente necessário, utilize o REPL apenas para confirmar os resultados que obteve:

`let codigo = {A: 19, B: 20}` `log(Object.entries(codigo).join(' + '))`	
`let codigo = new Map([['A', 19], ['B', 20]])` `log(codigo.get('B'), codigo.get('C'), codigo.get('C') || 21)`	

`let pessoa = {nome: 'alberto', idade: 23, altura: 190}`
`log(pessoa.nome[0].toUpperCase() +`
`  pessoa.nome.slice(1))`
`log(pessoa[pessoa.idade] || pessoa.altura)`	

`let x = 'ABC-DEF-GHI--JKL'.split('-')` `log(x.join('.'))`	

```
let formulas = [[1, -1, 2], [3, 2, 2]]
let m = formulas.slice()
m[0] = [9, 9, 9]
log(m, '--', formulas[0])
m = Array.from(formulas)
m[0][0] = 9
log(m, '--', formulas[0])
```

```
let t = "1,2,3"
log(t.slice(-1))
let vals = [10, 20, 30]
vals.splice(-2, 0, t)
log(vals)
```

```
let txt = '';
let nums = [10, 11];
if (nums.length !== 0) {
  log("Um");
  if (!txt) {
    log("Dois");
    txt = 'abc';
    nums = [];
  }
  else {
    log("Três");
    txt = 'xey';
    nums.splice(-1, 0, ...[12, 13]);
  }
}
txt = txt.replace('a', '').replace('e', '');
log(nums.length !== 0 ? "Quatro" : "Cinco");
log(nums.length === 0 ? txt : nums);
```

```
let itens = [['a', 13], ['d', 11], ['b', 15], ['c', 10]]
let outrosItens = []
for (let [i1, i2] of _.zip(itens, itens.slice().reverse())) {
    outrosItens.push([i1[0], i2[1]])
}
log(outrosItens)
```

```
let i = 0
let val = 1
while (val > 0 && (i <= 2 || !_.includes([3, 7], val))) {
    val = parseInt(prompt('Valor? '))
    log(val + 1)
    i += 1
}
log('Fim', val, i)
```

NOTA:
1. Primeiro assuma que o utilizador pretende introduzir: 10, -5, 1, 4, 7
2. Depois assuma que o utilizador pretende introduzir 3, 7, 8, 3, 6, -1

4. Converta os seguintes ciclos *for*/*for-of* em ciclos *for-of*/*for*. Se achar conveniente, pode utilizar as facilidades defindas em lodash.

```
let nome = "ALBERTO"
for (let i = 0; i < nome.length; i+=1) {
    log(nome[i])
}
```

```
for (let i = 1; i > -7; i-=1) {
    log(i)
}
```

```
let apelido = "ANTUNES"
for (let ch of Array.from(apelido).reverse()) {
    log(ch);
}
```

```
let nums = [10, 2, -2]
for (let [i, n] of nums.entries()) {
    log(i, n)
}
```

```
let nome = 'LARA'
for (let [i, ch] of [...nome].reverse().entries()) {
    log(nome.length - i, ch)
}
```

EXERCÍCIOS DE PROGRAMAÇÃO

***Instruções**: Para cada um dos problemas seguintes, desenvolva uma pequena página HTML com os elementos necessários para que o utilizador introduza a informação necessária e visualize os resultados pretendidos. Ignore preocupações estilísticas e, em particular, não se preocupe com cores, layouts e tipos de letra. Concentre-se na implementação correcta do código JavaScript.*

5. Desenvolva uma página para exibir efeitos especiais em texto. Para tal o utilizador introduz algum texto e depois a sua página exibe:

5.1 Texto na diagonal

5.2 Texto em V

5.3 Texto em cruz

6. Defina as seguintes funções:

6.1 Indicar se uma string possui mais do que uma palavra.

6.2 Devolver todas as letras de uma string

6.3 Indicar se um array possui pelo menos um elemento positivo.

6.4 Devolver todos os elementos positivos de um array.

6.5 Devolver todas as palavras de uma string. Por palavra entenda-se todas as sequências de caracteres consecutivos que não são caracteres de espaçamento.

6.6 Desenvolva uma função para calcular a fórmula resolvente. A função deve chamar-se resolvente, deve possuir três parâmetros de entrada, a, b e c, e deve devolver um array com os dois resultados.

***NOTAS**: 1) Desenvolva versões puramente iterativas destas funções. Não utilize `filter, map, every, etc.`*

7. Faça uma página para validar um NIF (Número de Identificação Fiscal).

De acordo com a Wikipedia (procurar por 'Número de identificação fiscal') as regras são as seguintes:

"*O NIF tem 9 dígitos, sendo o último o digito de controlo. Para ser calculado o digito de controlo:*

1. *Multiplique o 8.º dígito por 2, o 7.º dígito por 3, o 6.º dígito por 4, o 5.º dígito por 5, o 4.º dígito por 6,o 3.º dígito por 7, o 2.º dígito por 8, e o 1.º digito por 9*
2. *Adicione os resultados*
3. *Calcule o Módulo 11 do resultado, isto é, o resto da divisão do número por 11.*
4. *Se o resto for 0 ou 1, o dígito de controle será 0*
5. *Se for outro algarismo x, o dígito de controle será o resultado de 11 - x"*

8. Faça uma página para traduzir as coordenadas "simbólicas" do Excel para coordenadas lineares. Por exemplo, em Excel, internamente, a célula A1 corresponde à célula na linha 0 e coluna 0. Exemplos :

```
Coordenadas: Z 2
Linha: 1  Coluna: 25

Coordenadas: AA 3
Linha: 2  Coluna: 26

Coordenadas: AB 17
Linha: 16  Coluna: 27

Coordenadas: CD 17
Linha: 16  Coluna: 81
```

9. Desenvolva uma página através da qual introduz texto que depois é formatado a X colunas e alinhado à esquerda. Idealmente, palavras não são cortadas, excepto se a dimensão de uma palavra for superior a X.

Texto original	**Texto formatado a 20 colunas**
10/10/10: Fui ao mercado comprar peixe para o jantar. Encontrei o Alberto e o Armando. Convidei-os para jantar. Conversámos sobre o António, que eles encontraram no casamento do Arnaldo.	10/10/10: Fui ao mercado comprar peixe para o jantar. Encontrei o Alberto e o Armando. Convidei-os para jantar. Conversámos sobre o António, que eles encontraram no casamento do Arnaldo.

REFERÊNCIAS:

[1]: *Marijn Haverbeke*, *"Eloquent JavaScript, 3rd Ed."*, *2018, No Starch Press*, https://eloquentjavascript.net/index.html
[2]: JavaScript: MDN (Mozilla Developer Network): https://developer.mozilla.org/en-US/docs/Web/JavaScript
[3]: Gramar and Types @ MDN, https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Grammar_and_Types