PROGRAMAÇÃO WEB CLIENTE

UFCD(s) 792 - 793

GUIA DE LABORATÓRIO 2.1
ELEMENTOS BÁSICOS DE JS (ßeta)

OBJECTIVOS

- Conhecer o ambiente de desenvolvimento
- Tomar contacto com os elementos básicos da linguagem: valores, variáveis e tipos de dados

INSTRUÇÕES

PARTE I – CONSOLA, VALORES E TIPOS DE DADOS

1. Aceda à consola do navegador. Por norma, nos navegadores Chrome e Firefox acede-se à consola através de F12. Consulte a documentação do seu navegador ou peça ajuda ao formador.

Como vimos, no contexto de uma página Web, inserimos código JavaScript através do elemento HTML `script`*. Nestes laboratórios vamos recorrer a uma* ***consola*** *de JavaScript. Esta consola é similar à linha de comandos de um sistema operativo só que especializada em instruções da linguagem JavaScript. A grande vantagem reside no facto de ser interactiva, o que permite testar uma ou mais instruções e ter uma resposta imediata. Os navegadores modernos mais populares fornecem uma destas consolas, tipicamente acessíveis a partir do menu "Developer Tools". Fora do navegador podemos sempre utilizar o* ***Node.js*** *que também fornece uma destas consolas. No mundo das linguagens dinâmicas, estas consolas são designadas pelo acrónimo* ***REPL*** *de "Read Eval Print Loop", acrónimo que descreve o modo de interação entre um programador e a consola.*

A linguagem ***JavaScript*** *é uma* ***linguagem padronizada*** *pela European Computer Manufacturers Association ou* ***ECMA*** *[lê-se "é-que-ma"]. As três versões mais relevantes do standard, designado de* ***ECMAScript*** *e abreviado de* ***ES****, são ES3, publicado em 1999, ES5, de 2009, e o* ***ES6****, saído em* ***2015****. De 2015 em diante, é publicado um standard anual e que recebe a designação do ano. À data actual, Março de 2019, o standard actual é o ES2018 dado que o ES2019 ainda não foi aprovado. Comparativamente com o ES6, também designado de* ***ES2015****, os standards ES2016, ES2017 e ES2018 registam muito menos alterações, sendo o ES6 ainda o standard de referência da linguagem.*
Desde o ES5 que a linguagem oferece dois modos de funcionamento, normal e "strict" (rigoroso ou estrito). No ***modo strict*** *alguns mecanismos da linguagem considerados perigosos são inibidos, e mais erros são assinalados do que em modo normal. Algumas das novas funcionalidades do ES6 exigem modo strict, como é o caso dos módulos, que veremos mais à frente.*
As primeiras implementações da linguagem utilizavam interpretadores para processar o código. Hoje em dia, apesar de continuar com "uma aparência" interpretada, a maioria das implementações modernas utiliza um ***compilador JIT*** *(Just In Time), e daí ser mais habitual a utilização do termo "compilador" do que "interpretador". Devido ao JIT, código JavaScript tende a ser mais rápido que código equivalente escrito noutras linguagens dinâmicas, como Python ou Ruby.*

*Código em JavaScript é uma mistura de **definições** e de **comandos**. São estes os dois tipos básicos de instruções. Ambos os tipos de instruções alteram o estado do ambiente, mas as instruções do primeiro tipo servem para acrescentar novos conceitos à linguagem, ao passo que as outras são executadas e produzem imediatamente algo.*
Estas instruções, o script (ie, o programa), podem ser introduzidas uma a uma no REPL, ou carregadas a partir de um ficheiro.
*Um **comando** (**command** ou **statement**) instrui o compilador a fazer algo. Vamos começar por ver alguns exemplos de comandos e depois passamos às definições.*
*Podemos ainda dividir as instruções em **simples** e **compostas**. Uma instrução simples (que é sempre um comando) vale por si só, isto é, não faz parte de um conjunto de instruções. Uma instrução composta "necessita " de outra instruções para ficar completamente definida. As definições são sempre instruções compostas, mas não são as únicas.*

2. Introduza a seguinte expressão na linha de comandos (área 1), pressione ENTER e observe o que sucede na saída (área 2):

```
>>> 3 * (14 + 26)
120
```

*A consola mostra na saída (área 2) o resultado de tudo o que introduzimos na linha de comandos. Neste caso introduzimos uma expressão aritmética. Uma **expressão** é um "valor" já calculado ou por calcular. Quando está por calcular, esse valor resulta da combinação de outros "valores" através de operadores (ou funções).*
No nosso caso os valores `3`, `14` e `26` são combinados através dos operadores `*` e `+`.
O ponto-e-vírgula é opcional mas, por razões que se tornarão aparentes mais à frente, vamos utilizá-lo sempre em scripts.

3. Agora introduza cada uma das seguintes expressões separadas por ENTER:

```
>>> 70 + 357
427
>>> 9.81 / 3
3.27
>>> 150 * (1 + 23/100)
184.5
>>> 20 - (2.5 * 8)
0
>>> +Infinity
Infinity
>>> -Infinity
-Infinity
>>> +Infinity - 10000
+Infinity
>>> +Infinity - +Infinity
NaN
>>> +Infinity - -Infinity
Infinity
```

*Os valores `70`, `357`, `9.81`, etc., são valores numéricos, pertencentes ao tipo de dados **number**. Ao contrário de outras linguagens, como C ou Java, em JavaScript não há separação entre números inteiros e números decimais. JavaScript utiliza o formato em vírgula flutuante de dupla precisão (64 bits) definido pelo standard **IEEE 754**, oficialmente designado de **binary64** mas mais conhecido por **double**. Desses 64 bits, a mantissa é composta por 52 bits que são então utilizados para representar o número em fracção binária, ao passo que os 11 bits remanescentes representam um expoente entre -1023 e +1024, para um coefeciente de base 2. Sobra um bit que serve para indicar o sinal do número, ie, se o número é positivo ou negativo. Este formato permite representar números reais entre $-(2^{53} - 1)$ e $2^{53} - 1$. Existem três valores numéricos simbólicos: `-Infinity`, `+Infinity` e `NaN` (`NaN` significa* "Not a Number" *mas qual é o seu tipo de dados? Sim, é `'number'`. Bem vindos a JavaScript!).*

4. Agora, introduza apenas a seguinte expressão e, novamente, observe o sucedido:

*A expressão está errada: falta o valor à direita do operador `*` . Este tipo de erro, de violação das regras de escrita de código JavaScript, é designado por **erro sintáctico**. Todos os erros detectados pelo ambiente são exibidos também na saída.*

```
>>> 3*
```

***NOTA**: Se a consola não permitir a introdução da expressão inválida, tente introduzir* `eval("3*")`.

5. Introduza o seguinte texto na linha de comandos:

```
>>> "O Alberto é amigo do Armando"
'O Alberto é amigo do Armando'
>>> 'O Alberto é amigo do Armando'
'O Alberto é amigo do Armando'
```

Para saber o que "fazer" com a informação e como armazená-la da melhor maneira, a maioria das linguagens de programação estrutura a informação em tipos de dados. Um ***tipo de dados*** *define um universo de valores e estabelece quais as operações admissíveis para esses valores.*
Até agora vimos apenas um tipo de dados: o tipo numérico. Neste passo introduzimos o tipo ***string****. O tipo de dados string consiste numa sequência de zero ou mais caracteres e é utilizado para representar texto. Os caracteres (ie, o texto) são escritos entre aspas ou entre* plicas. *Ou seja, em cima também poderíamos ter escrito:* `'O Alberto é amigo do Armando'`

6. Introduza agora cada uma das seguintes expressões e registe o que observa:

```
>>> "4 + 70"
'4+70'
>>> "Três vezes sete é 3*7"
'Três vezes sete é 3*7'
>>> "Três vezes sete é " +
3*7
'Três vezes sete é 21'
>>> "Alberto" + " " + "Alves"
'Alberto Alves'
```

Como JavaScript não interpreta o conteúdo de uma string, podemos escrever o que queremos entre aspas ou plicas. Para além disto, podemos utilizar o operador + para concatenar uma string com outra string ou com outros valores (como, por exemplo, números).
Acedemos aos caracteres individuais de uma string através do método `charAt` *ou do operador de indexação* `[]`*. Falaremos de* ***métodos*** *e* ***funções*** *neste e noutros laboratórios. Para já basta saber que são operações com nome, como é o caso de* `charAt`.

```
>>> "Alberto".length
7
>>> "Alberto".toUpperCase()
'ALBERTO'
>>> "Alberto".indexOf("to")
5
>>> "Alberto".includes("rt")
true
```

```
>>> "Alberto".charAt(0)
'A'
>>> "Alberto".charAt(6)
'o'
>>> "Alberto"[0]
'A'
>>> "Alberto"[6]
'o'
```

```
>>> typeof "Alberto"
'string'
>>> typeof "Alberto"[0]
'string'
>>> typeof 23
'number'
>>> typeof true
'boolean'
```

JavaScript é uma ***linguagem dinâmica****. Entre outros aspectos, isto significa que os tipos de dados estão associados aos valores e não a identificadores como variáveis ou funções. Como as variáveis não estão associadas a um tipo de dados em concreto, é possível atribuir valores de diferentes tipos de dados à mesma variável.*

O ES6 define 8 tipos de dados, dos quais 6 são identificados pelo operador **`typeof`** *(este operador é mencionado a seguir) :*
- **`boolean`***: valores lógicos* **`true`** *e* **`false`**.
- **`null`** [1] *: tipo de dados que representa ausência de valor, a noção de vazio, ou algo que não estão definido*
- **`undefined`** [2] *: semelhante a* **`null`**, *mas mais utilizado para representar um nome que ainda não foi associado um valor, ou seja, cujo valor é indefinido.*
- **`number`***: valores numéricos como anteriormente indicado*

*- **string**: sequências de caracteres em Unicode, em que cada caractere é um elemento de 16 bits; de acordo com o standard da linguagem o sistema de codificação pode ser UCS-2 ou UTF-16. Porém, ao dar garantias que cada caractere ocupa 16 bits, e como alguns caracteres em UTF-16 necessitam de 32 bits, na prática os caracteres são expostos como em UCS-2.*
*- **symbol**: tipo introduzido com ES6; representa um valor único; para já, vamos ignorar este tipo de dados*
*- **object**: colecção dinâmica de propriedades; uma **propriedade** é uma associação entre uma string[3], designada de chave (key) e um valor (value); este valor pode ser primitivo, uma função (que neste contexto é designada de método) ou outro objecto.*
*- **bigint**: números inteiros de precisão variável; foi adicionando recentemente à linguagem*
Em termos de complexidade, os tipos de dados de JavaScript estão divididos em duas categorias: primitivos e objectos. À excepção do tipo de dados `object`, todos os tipos de dados da lista anterior são primitivos.
***Tipos primitivos** são tipos de dados cujos valores não são objectos e que possuem algum suporte especial ao nível da implementação, sendo, provavelmente, implementados directamente em linguagem máquina. Os valores destes tipos de dados são imutáveis, ou seja, não conseguimos mudar nenhum valor primitivo; apenas podemos obter cópias com as alterações.*
Objectos são, como referido, colecções de propriedades. Um mapeamento entre chaves e valores é uma outra forma de definir um objecto. Estas propriedades, que também podemos designar de atributos, tanto podem possuir dados ou funções. Ao contrário de valores primitivos, objectos são mutáveis, ainda que seja possível redefinir este comportamento. Podemos aceder a uma propriedade de um objecto através do operador . (ponto) escrevendo `objecto.propriedade` .
*Como veremos, os objectos podem pertencer a "sub-tipos" de dados (o tipo de dados **object** é, na verdade, uma família de tipos de dados). Cada um destes sub-tipos de dados define a sua colecção de propriedades, que, lá está, depois podem ser acedidas através do operador . (ponto).*

*Os valores primitivos não possuem propriedades. Contudo, à excepção de **null** e **undefined**, os tipos primitivos possuem um tipo de dados equivalente no mundo dos objectos. Estes tipos, designados de wrapper types, são **Boolean**, **Number**, **String** e **Symbol**. No fundo, estes wrapper types[4] existem para dar acesso a mecanismos de objectos aos valores de tipos primitivos. Existe ainda o wrapper type **Object** utilizado, na maioria dos casos, para criar objectos vazios ou com um conjunto pré-definido de propriedades.*

*O operador **typeof** devolve uma string com a designação do tipo de dados de um valor. Este operador apenas discrimina entre 6 tipos de dados: todos os da lista em cima, excepto **null**, uma vez que o tipo de dados de **null** é `'object'`.*
*A maioria das linguagens permite que os programadores acrescentem **novos tipos de dados** à linguagem, por vezes designados de User Defined Types (UDT). Por norma isto envolve mecanismos como classes, estruturas, renomeação de tipos, genéricos, interfaces, etc. Tipicamente, estes mecanismos "colam" uma "etiqueta" ou "rótulo" nos objectos com a identificação do tipo de dados[5]. Em JavaScript não existe nenhum mecanismo formal para definir novos tipos de dados e não são coladas "etiquetas" ou "rótulos" de forma explícitos. É possível, no entanto, definir funções (falaremos de funções mais à frente; para já funções são operações às quais podemos dar um nome) para criar objectos, funções essas que são designadas de **construtores**. O tipo de dados dos objectos criados através destes construtores é o nome da função que lhes deu origem. No entanto, o facto de um objecto ter sido criado por este ou por aquele construtor é completamente ignorado pelo operador **typeof**, sendo que para ele todos os objectos são do tipo `object`.*
*Como mencionado, o operador **typeof** apenas conhece 6 tipos de dados, e não distingue os "sub-tipos" de dados pertencentes à "categoria" `object`. Dado um objecto, o operador booleano **instanceof** permite perguntar se esse objecto foi criado com um determinado construtor [6].*
Um aspecto que ficará mais claro ao longo destes laboratórios, é que tudo o que não é um valor de um tipo de dados primitivo é um objecto, e isto abrange também as próprias funções. Ou seja, funções, incluindo os construtores, são objectos.
*Ou seja, tipos de dados são objectos. Por isso é que em JavaScript designamos os tipos de dados como **Number**, **Array**, **String**, etc, também como objectos. Isto pode ser confuso, especialmente para quem vem de linguagens estáticas (...)*

(...) e fortemente tipificadas, como C++ ou Java, mas ao contrário de outras "confusões" da linguagem, esta é uma "boa confusão" e que simplifica a linguagem.

Os últimos parágrafos destas caixas de texto foram dedicados extensivamente a tipos de dados (talvez em demasia...) mas esta noção não é nem de perto nem de longe tão importante em JavaScript como noutras linguagens. A noção mais importante é, na verdade, a noção de ***objecto****: uma coleção de propriedades que podemos utilizar para definir conceitos tão variados como* `String`, `Number`, `Carro`, `ContasBancarias`, `Utilizador`, `CarrinhoCompras`, *etc. E talvez ainda mais importante que a noção de objecto é a noção de* ***função****, e esta será abordada mais à frente.*

1 - Na verdade **`null`**, *deveria ser, mas não é um tipo de dados, mas sim um* ***valor primitivo****. Qual o tipo de dados de* **`null`**? `'object'`. *Porquê? Foi um erro de concepção, impossível de alterar a partir do momento que ficou cristalizado num dos primeiros standards da linguagem.*
2 - **`undefined`** *é um tipo de dados com um valor apenas: o valor primitivo* **`undefined`**.
3 - A partir de ES6, chaves também podem ser do tipo **`symbol`**.
4 - O termo mais correcto é wrapper objects. Em JavaScript, sub-tipos de dados do tipo **`object`** *são objectos.*
5 - Em linguagens estáticas e compiladas por norma essa etiqueta existe apenas durante a compilação.
6 - Por simplicidade omitimos que, na verdade, o operador **`instanceof`** *indica se a propriedade* **`prototype`** *de um construtor faz parte da cadeia de protótipos do objecto, isto é, se o construtor aparece na hierarquia de construtores que deram origem ao objecto. Isto está relacionado com o conceito de herança, tal como ele é aplicado em JavaScript.*

7. Para introduzir caracteres especiais dentro de uma string utilizamos a barra \ . Por exemplo:

```
>>> console.log("Amanhã vai chover.\nOu não...")
Amanhã vai chover.
Ou não...
>>> console.log("Gosto de \"aspas\" ao pequeno almoço.")
Gosto de "aspas" ao pequeno almoço.
>>> console.log("Qual o dia dois dias\n\ndepois de anteontem?")
Qual o dia dois dias

depois de anteontem?
>>> console.log("\tR: É o dia antes de amanhã...")
        R: É o dia antes de amanhã...
>>> console.log("E como introduzir a barra \\ ? Fácil! Basta escrevê-la 2
vezes: \\\\")
E como introduzir a barra \ ? Fácil! Basta escrevê-la 2 vezes: \\
```

`console` é uma referência para o objecto `Console`, que é a consola de depuração (debugging) global do ambiente de execução. A consola é uma janela em modo texto que utilizamos para interagir com o ambiente de execução. Podemos introduzir comandos, através do REPL da consola, mas podemos também enviar texto que permite auxiliar na depuração e compreensão dos nossos scripts. A consola não serve, porém, para interagir com os utilizadores.

PARTE II – OPERADORES E VARIÁVEIS

8. Continuando, desta feita introduza:

```
>>> 3 > 2
true
>>> 3 !== 2
```

```
true
>>> 3 === 2
false
>>> "Alberto" === "alberto"
false
>>> "Alberto" === "Alberto"
true
>>> !(3 === 2)
true
>>> !(3 !== 2)
false
>>> 3 > 2 && 10 !== 8
true

>>> 3 > 22 && 10 !== 8
false
>>> 3 > 22 || 10 !== 8
true
>>> !(3 > 22 || 10 !== 8)
false
>>> typeof 3 === typeof 4
true
>>> true !== false
true
>>> "abc" >= "xyz"
false
>>> "abc" >= "Xyz"
true
```

*Em cima (ao lado?) escrevemos várias expressões - eg: `30 - 5*6` . Uma **expressão** é uma operação, mais ou menos complexa, que produz um valor. Utilizamos operadores para "combinar" outros valores e o resultado de outras expressões, e assim produzir o resultado da expressão. Um **operador** é um simbolo que representa uma operação. Exemplos de operadores: `+` (soma), `/` (divisão), `<<` (deslocamento binário à esquerda), etc. A expressão `3*(14 + 25)` combina duas subexpressões - `3` e `14 + 25` - através do operador `*` para produzir o valor `117`.*
Expressões não necessitam necessariamente de envolver valores numéricos:
. `"Bom dia," + "Alberto"` → expressão que junta duas strings (string é o termo que utilizamos para texto) `"Bom dia, "` e `"Alberto"` e produz o texto `"Bom dia,Alberto"`
. `10 > 20` → expressão que produz o valor booleano **`false`**

*O operador `===` avalia se duas expressões produzem o mesmo valor. O operador **`!==`** avalia o oposto, isto é, se duas expressões produzem valores diferentes. Note-se que ambos os operadores produzem* **`true`** *ou* **`false`**. *Designam-se por **operadores booleanos** ou **lógicos**. Outro operador lógico, o `!`, devolve o valor lógico oposto ao da expressão à direita. Repare que cada valor booleano é representado por uma palavra (que não está entre aspas ou plicas). A linguagem JavaScript reserva determinadas palavras para serem utilizadas pelo interpretador e essas palavras, designadas por **palavras-reservadas** ou **palavras-chave**, não podem ser utilizadas pelos programadores para dar nomes a objectos (algo que, veremos, é muito comum).*

*Os outros dois operadores fundamentais da lógica são os operadores **`&&`** ("E lógico") e `||` ("OU" lógico). O primeiro devolve* **`true`** ***apenas*** *quando as expressões à esquerda e à direita do operador forem **ambas verdadeiras**. O segundo devolve* **`false`** ***apenas*** *quando as expressões à esquerda e à direita do operador forem **ambas falsas**.*
*Um pouco de terminologia: operadores **unários**, como o `!`, são os que avaliam apenas uma expressão, **binários**, os que avaliam duas (`&&`, `+`, etc.), **ternários**, os que avaliam três (veremos mais à frente), etc.*

9. Nem sempre podemos aplicar todos os operadores a todos os tipos de operandos. Por exemplo:

```
>>> 4 * 'bacalhau'
NaN
>>> 10 / '5 0'
NaN
```

10. Números de vírgula flutuante são uma aproximação a números reais e não são mesmo números reais:

```
>>> 5.9 - 2
3.9000000000000004
```

```
>>> 1.1 + 2.2
3.3000000000000003
>>> 0.83 + 0.7
1.5299999999999998
>>> 1.0 % 0.1
0.09999999999999995
// devia dar resto 0 porque 1 é divisível por 0.1
```

> **numbers** *não são reais. São uma representação eficiente mas inexacta. Em situações em que a exactidão é crucial (eg, aplicações financeiras), devemos utilizar outros tipos de dados. Não existe nenhum no standard, mas existem bibliotecas como* **bignumber.js** *que fornecem tipos de dados apropriados.*

11. E é bom termos consciência disso, caso contrário as consequências podem ser graves:

```
>>> (1.1 + 2.2) - 3.3 === 0
False
>>> 0.2 + 1 - 1 === 0
false
```

O melhor que conseguimos é testar se dois valores numéricos com casas decimais são *aprox.* iguais.

12. Introduza o seguinte:

```
>>> let lado = 10
>>> let comp = 20
>>> let area = lado * comp
>>> console.log("Área do rectângulo é:",
area)
Área do rectângulo é:  200
>>> let raio = 3
>>> area = Math.PI * raio ** 2
28.274333882308138
>>> console.log("Área da circunf. é:", area)
Área da circunf. é: 28.274333882308138
>>> console.log("Área da circunf. é:",
                area.toFixed(1))
Área da circunf. é: 28.3
```

13. Uma outra forma de atribuirmos valores passa por utilizar *destructured assignment*. Veremos que isto envolve arrays ou objectos literais, mas para já ilustramos o procedimento:

```
>>> let [a, b] = [20, 30]
>>> console.log(a, b)
20 30
>>> let [c, d] = [2 * b, a + 1]
```

> `lado`, `comp`, `area` *são variáveis. Em JavaScript, uma* ***variável*** *é apenas e só um nome que referencia um objecto. Este nome pode associar-se a diferentes objectos ao longo da "vida" do programa e daí a designação "variável" (varia o objecto ao qual a variável está associada). As variáveis, no fundo, representam a história do programa. Elas servem para guardar informação que pode ser utilizada mais tarde. Indicam em que "***estado***" é que um programa se encontra e permitem-nos tomar decisões em função do que já se passou.*
>
> *Criamos uma variável quando atribuimos um valor/objecto a um identificador com o* ***operador de atribuição*** **=**. *O operador* **=** *indica que o objecto que está à direita está ligado (bounded) à variável que está à esquerda. Note-se que este operador _não_ serve para "perguntar" se o que está à direita é igual ao que está à esquerda (esse operador é o* **===** *ou, em alguns casos, o* **==***).*
>
> *A* ***declaração*** *de uma variável pode (e deve) ser precedida das palavras reservadas* **let**, **var** *ou* **const**. *Se quisermos que a associação entre o identificador e o nome seja permanente, utilizamos a palavra-reservada* **const**. *Com* **const** *definimos uma* ***constante***, *isto é, uma variável que não varia (sim, é uma contradição em termos). À frente explicamos as diferenças entre estas duas palavras-reservadas e porque é que devemos utilizar* **1) const** *sempre que possível,* **2) let** *quando não podemos utilizar* **const**, *e* **3)** *porque é não se deve utilizar* **var** *em JavaScript moderno. Se não utilizarmos nenhuma destas palavras-reservadas, o JavaScript cria uma variável e torna-a* ***global***. *Quando falarmos de funções veremos o que são variáveis globais e porque é que as devemos utilizar com cuidado.*

```
>>> c + d
81
```

14. Cada variável tem uma existência autónoma, e atribuir um valor a uma delas não altera a outra:

```
>>> let z = 40
>>> let [x, y] = [z, z]
>>> z = 14
14
```

```
>>> y = 28
28
>>> [x, y, z]
[ 40, 28, 14 ]
```

15. Vamos trocar os valores das variáveis. Normalmente, precisamos de uma variável temporária para guardar o valor de uma delas. Aqui podemos utilizar *destr. assign.* para fazer a troca numa instrução:

```
>>> let tmp = a
>>> a = b
30
>>> b = tmp
20
>>> [a, b]
[30, 20]
```

```
>>> let [i, k] = [20, 30]
>>> [i, k]
[20, 30]
>>> [i, k] = [k, i]
[30, 20]
>>> console.log("i:", i, "k:", k)
i: 30 k: 20
```

16. Como vimos, para comparar valores e efectuar outros tipos de "testes" temos o tipo `boolean` e os operadores relacionais `<`, `<=`, `>`, `>=`, `!=`, `==`, `===` e `!==` :

```
>>> let x = 5, y = 10
>>> let log = console.log
>>> log
[Function: bound consoleCall]

>>> log("X é maior do que Y?", x > y)
X é maior do que Y? false
>>> log("X é menor do que Y?", x < y)
X é menor do que Y? true

>>> x = y;
10
>>> log("X é menor do que Y?", x < y)
X é menor do que Y? false
>>> log("X é igual a Y?", x === y)
X é igual a Y? true
>>> log("X é diferente de Y?", x !==
y)
X é diferente de Y? false

>>> x = '10'
```

O conjunto de identificadores definidos e acessíveis em qualquer altura é designado de ***espaço de nomes*** *(**namespace**). Ou seja, um espaço de nomes é um contexto onde determinados nomes são válidos. Um **identificador** é simplesmente um termo complicado para "nome". Como na maioria das linguagens, um identificador não pode conter espaços e tem que começar por uma letra, traço inferior (_) ou dólar ($); os restantes caracteres podem também ser dígitos. A linguagem é "case sensitive", logo os identificadores* `a` `b` `c` *e* `ABC` *são distintos. Determinados identificadores, designados por **palavras-reservadas,** estão reservados para a própria linguagem e já têm um significado. Não podemos utilizar esses nomes para nomear objectos.*

*Em ES5, o standard em vigor até 2015, existiam apenas dois espaços de nomes: o global e o local a cada função. Isto é problemático quando temos programas muitro grandes, com muitas definições, onde é possível que ocorram conflitos de nomes. Assim, com ES6 apareceram módulos e pacotes. Um **módulo** corresponde a um conjunto de definições e declarações relacionadas e guardadas num ficheiro com o nome do módulo (e com extensão* `.js` *ou idealmente* `.mjs`*). Estas definições só existem dentro do módulo, apesar de poderem ser exportadas para outros módulos. Falaremos de módulos e de pacotes noutros laboratórios.*

```
'10'
>>> [x == y, x === y]
[ true, false ]
```

*O operador **===** significa "do mesmo tipo de dados e igual", ao passo que **==** significa "valor considerado igual, ainda que os objectos possam ser de tipos de dados diferentes". Ou seja, o operador **==** compara igualdade após tentar fazer converter os objectos para o mesmo tipo de dados. Por exemplo:*

```
33 == "33"   // true
33 === "33"  // false
```

*Os operadores **==** e **!=** devem ser evitados e preteridos em favor **===** e **!==**. Citando "JavaScript: The Good Parts" de Douglas Crockford:*

JavaScript has two sets of equality operators: **===** and **!==**, and their evil twins **==** and **!=**. The good ones work the way you would expect. If the two operands are of the same type and have the same value, then **===** produces **true** and **!==** produces false. The evil twins do the right thing when the operands are of the same type, but if they are of different types, they attempt to coerce the values. the rules by which they do that are complicated and unmemorable. These are some of the interesting cases:

```
'' == '0'          // false
0 == ''            // true
0 == '0'           // true
false == 'false'   // false
false == '0'       // true

false == undefined  // false
false == null       // false
null == undefined   // true
' \t\r\n ' == 0     // true
```

Consultar: https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons

PARTE III – MAIS SOBRE STRINGS

17. Como vimos, strings são sequências de caracteres numeradas a partir de `0`. Podemos utilizar o operador de indexação `[]` para indexar uma string, indicando um número de ordem ou posição entre `[` e `]`.

```
>>> "Alberto".length
7
>>> let nome = "Henrique"
'Henrique'
>>> nome.length
8
>>> nome = "António"
'António'
>>> [ nome[0], nome[1] ]
[ 'A', 'n' ]
>>>
>>> [ nome[6], nome[nome.length - 1] ]
[ 'o', 'o' ]
```

18. Como as strings são imutáveis, é um erro (que passa de forma "silenciosa") tentar modificar um caractere da string:

```
>>> nome[6] = 'a'
```

```
'a'
>>> nome
'António'
```

19. Podemos utilizar o método `slice` para fatiar uma string ou um array (que veremos a seguir):

```
>>> let txt = 'ABCDEF'
>>> txt.slice(0, 2)  // caracts. 0 e 1
'AB'
>>> txt.slice(2, 4)  // caracts. 2 e 3
'CD'
>>> txt.slice(0, txt.length)  // caracts. 0 a 6 (exclusive, ou seja, 0 a 5)
'ABCDEF'
>>> txt.slice(0)   // por omissão 2o valor é txt.length
'ABCDEF'
>>> txt.slice()    // uma cópia de txt de forma sucinta
'ABCDEF'
>>> txt.slice(1)   // todos os caracts. menos o primeiro
'BCDEF'
>>> txt.slice(10)  // fora do índice, devolve '' (e não dá erro)
''
```

***Fatias** (slices) são subsequências da sequência original. Por exemplo, dada a string `xyz` ,*

`xyz.slice(inicio, fim)` → nova string contendo os caracteres de `xyz[inicio]` a `xyz[fim-1]`.

Ou seja todos caracteres cujos índices estejam no intervalo aberto `[inicio, fim)`. Obtemos uma cópia da string fazendo: `xyz.slice(0, xyz.length)`. Por omissão, `inicio` tem o valor `0` e `fim` o valor `.length`. Deste modo, `xyz.slice()` também devolve uma cópia da string.

Para ajudar a visualizar o fatiamento pode ser útil olhar para os índices dos caracteres como estando entre os caracteres:

```
 +---+---+---+---+---+---+
 | A | M | A | Z | O | N |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Note-se que tudo o que aprendemos sobre slicing aplica-se aos outros tipos de sequência que vamos dar (arrays)

20. "Fatiamento" também funciona com índices negativos. A lógica está ilustrada na caixa anterior:

```
>>> txt.slice(-6, -4)  // caracts. -6 e -5 (ou 0 e 1)
'AB'
>>> txt.slice(-4, -2)  // caracts. -4 e -2 (ou 2 e 3)
'CD'
>>> txt.slice(-6)      // caracts. -6 até ao fim
'ABCDEF'
>>> txt.slice(-6, -1)  // caracts. -6 até -1 (exclusive, ou seja -6 a -2 ou 0 a 4)
'ABCDE'
>>> txt.slice(-6, 2)   // caracts. -6 e 1 (isto é, 0 e 1)
'AB'
```

21. As strings suportam outras operações. Resumidamente, aqui ficam algumas das mais comuns

```
>>> let nome = "Arm" + "ando"
>>> nome
'Armando'
>>> nome.toUpperCase()
'ARMANDO'
>>> nome
'Armando'
>>> nome = nome.toLowerCase()
'armando'
>>> nome
'armando'
>>> nome.repeat(2)
'armandoarmando'
>>> nome = nome[0].toUpperCase() +
... nome.slice(1).toLowerCase()
'Armando'

>>> // pesquisas
>>> nome.startsWith("ar")
false
>>> nome.startsWith("ra")
false
>>> nome.startsWith("Ar")
true
>>> nome = 'armando'
'armando'
>>> // indexOf: devolve posição de "ma"
>>> nome.indexOf("ma")
2
>>> nome.indexOf("a")
0
>>> nome.indexOf("z")
-1
>>> // procura a partir do índice 1
>>> nome.indexOf("a", 1)
3
>>> nome.lastIndexOf("a")
3
>>> nome.lastIndexOf("a", 2)
0

>>> // split: divide nome "à volta" de
>>> // uma string (neste caso, de "m")
>>> nome.split("m")
```

Em JavasScript podemos invocar determinadas operações escrevendo:

`operacao(argumentos)`

Este é o caso de, por exemplo, operações como `parseInt` *e* `isNaN`*. No entanto, para outras operações, como as relacionadas com strings, utilizamos a forma*

`umaString.operacao(outrosArgumentos)`

Isto deve-se ao facto de `parseInt` *e "companhia" serem* ***funções globais****. Ou seja, são operações que estão definidas apenas no espaço de nomes global. Os seus* ***operandos****, também designados de* ***argumentos****, devem ser passados entre parênteses. As operações que manipulam strings são específicas de ... strings e foram definidas no espaço de nomes do objecto* **String**. *Este objecto é como que um* ***modelo*** *a partir do qual criamos outros objectos semelhantes, ie, outras strings; nesse modelo indicamos as operações suportadas por todas as strings. Essas operações, que também são funções, são designadas de* ***métodos*** *e acedemos a elas fazendo* `obj.operacao(...)`*.*

Como as operações, isto é, os métodos `startsWith`*,* `endsWith`*,* `toUpperCase`*, etc, estão definidos no objecto* **String***, por vezes vamos nos referir a elas desta forma:* **String**`.startsWith`*,* **String**`.endsWith`*,* **String**`.toUpperCase`*, etc.*

NOTA: *na documentação aparecem como, por exemplo,* **String**`.prototype.startsWith`*; mais à frente falaremos de protótipos.*

Consultar estas e outras operações em:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

```
[ 'ar', 'ando' ]
>>> nome.split("a")
[ '', 'rm', 'ndo' ]
>>> "Alberto Armando Antunes".split(" ")
[ 'Alberto', 'Armando', 'Antunes' ]
>>> "Alberto   Armando  Antunes".split(" ")
[ 'Alberto', '', '', 'Armando', '', 'Antunes' ]
```

22. Finalmente, consulte a seguinte documentação sobre *template literals* (tradução livre: "modelos de string literais") e analise os seguintes exemplos:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
>>> let nome = "Alberto", idade = 23, preco = 25.16 * 1.23
>>> let log = console.log
>>> log(`O ${nome}, que tem ${idade} anos, comprou o livro de programação
por ${preco} euros`)
>>> log(`O ${nome}, que tem ${idade} anos, comprou o livro de programação \
por ${preco.toFixed(2)} euros`)
```

PARTE IV – ARRAYS

23. Vamos agora trabalhar com *arrays*. Comece por introduzir:

```
>>> let nomes = ["Alberto", "António", "Armando", "Arnaldo"]
>>> [nomes, typeof nomes, nomes.constructor]
[ [ 'Alberto', 'António', 'Armando', 'Arnaldo' ], 'object', [Function: Array] ]
>>> [ Array.isArray(nomes), Array.isArray(3), Array.isArray("array") ]
[ true, false, false ]
>>> // arrays podem ser indexados e fatiados
>>> [ nomes[0], nomes[nomes.length - 1] ]
[ 'Alberto', 'Arnaldo' ]
>>> nomes.slice(-1)
[ 'Arnaldo' ]
>>> nomes.slice(1)
[ 'António', 'Armando', 'Arnaldo' ]
>>> // 'António' está em nomes?
>>> 'António' in nomes
false
>>> [0 in nomes, 1 in nomes, 7 in nomes]
[ true, true, false ]
>>> nomes.slice(1, 3)
[ 'António', 'Armando' ]
```

*Um **array** é uma sequência **heterogénea** de elementos, isto é, de elementos de qualquer tipo de dados. Na verdade, um array em JavaScript não é aquilo a que se chama array noutras linguagens, como C ou C++. Nessas linguagens um array é uma coleção homogénea de elementos, todos alinhados consecutiva e contiguamente em memória. Aqui, um array é, na verdade, um objecto onde cada posição é uma propriedade do objecto. E ao contrário de verdadeiros* arrays *(que também existem em JavaScript, mas são mais comuns em C ou C++), os elementos não estão alinhados contiguamente em memória; nem sequer as referências estão alinhadas contiguamente em memória, como sucede, por exemplo, com uma lista em Python ou uma ArrayList em Java.*

24. Podemos modificar o conteúdo do *array*:

```
>>> nomes[0] = 'Alberta'
```

```
>>> nomes
[ 'Alberta', 'António', 'Armando', 'Arnaldo' ]
>>> nomes.splice(1)
[ 'António', 'Armando', 'Arnaldo' ]
>>> nomes
['Alberta']
>>> nomes.push('Antónia')
2
>>> nomes.push('Armanda', 'Arnalda', 'Andreia')
5
>>> nomes
[ 'Alberta', 'Antónia', 'Armanda', 'Arnalda', 'Andreia' ]

>>> // vamos concatenar arrays
>>> nomes2 = nomes.concat('Anabela', 'Arlete')
>>> nomes2.push('Ana')
>>> [nomes, nomes2]
[ ['Alberta', … , 'Andreia'], ['Alberta', … , 'Andreia', 'Anabela', 'Arlete', 'Ana'] ]

>>> // remover parte(s) do array
>>> nomes.splice(1, 2)      // 2 é o número de elementos a remover
[ 'Antónia', 'Armanda' ]
>>> nomes
[ 'Alberta', 'Arnalda', 'Andreia' ]
>>> nomes.splice(1, 1, 'Adelina', 'Arminda')
[ 'Arnalda' ]
>>> nomes
[ 'Alberta', 'Adelina', 'Arminda', 'Andreia' ]
>>> // se pretendermos aceder e/ou remover um elemento podemos usar shift ou pop
>>> nomes.pop()
'Andreia'
>>> nomes
[ 'Alberta', 'Adelina', 'Arminda' ]
>>> nomes.shift()
'Alberta'
>>> nomes
[ 'Adelina', 'Arminda' ]

>>> // e agora a função inversa do String.split
>>> nomes.join(",")
'Adelina,Arminda'
>>> "Alberto Arnaldo Armando".split(" ").join("+")
'Alberto+Arnaldo+Armando'
```

*Ao contrário de strings, os arrays são **mutáveis**. Quer isto dizer que, não só podemos alterar o conteúdo dos elementos da lista, como podemos modificar a sua estrutura (acrescentar ou remover elementos).*

Desta demonstração omitimos métodos muito importantes, como `find`, `filter` ou `every`, entre outros. Estes métodos, muito usados em programação funcional, recebem funções como argumento e, como tal, deixamos o seu estudo para o laboratório de funções.

Consultar estas e outras operações em:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

25. A propósito de *arrays* e de operadores, consulte a documentação sobre o operador ... (*spread operator*), em particular a sua utilização fora de funções, em:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

EXERCÍCIOS DE REVISÃO

1. O que é um "identificador"?

2. Quais os tipos de dados primitivos em JavaScript?

3. Leia a seguinte a informação sobre **`undefined`** vs **`null`**: https://codeburst.io/understanding-null-undefined-and-nan-b603cb74b44c e indique as principais diferenças entre ambos.

4. Indique duas formas de verificar se o resultado de uma operação numérico foi `NaN`?

5. Quais dos seguintes nomes de variáveis estão correctos?

- **5.1** `1nome`
- **5.2** `nome1`
- **5.3** `$nome`
- **5.4** `"nome um"`
- **5.5** `nome-um`
- **5.6** `_nome`
- **5.7** `$_`
- **5.8** `_1`

6. Escreva e teste na consola expressões para calcular:

- **6.1** A divisão do dobro da soma de 19 com 20 por 3.
- **6.2** A área de um quadrado de lado 4.
- **6.3** A área de um quadrado com lado introduzido pelo utilizador.
- **6.4** A área de um triângulo de base 10 e altura 8.
- **6.5** O triplo da soma do triplo da divisão 81 por 27 com 18.

7. Será que a seguinte expressão é verdadeira?

```
((4 >= 6) || ("grass" !== "green")) && !(((12 * 2) === 144) && true)
```

Primeiro tente obter a solução por si e depois confirme na consola de JavaScript.

8. Consulte e anote o que fazem os operadores: `+=`, `++`, `||`, `*=` e `%`.

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

`let p = 10;` `log(p*2);`	

`let b = 60, c = 20;`
`log(b /c + ",");`
`b = b + 3; c = c + 1;`
`log(b /c + "\n" + "F\ni\nm\n!");`	

`let q = 2.3, r = 10;`
`q *= r;`
`log("Valor de q", q, "\n");`	

`let nome1 = "Alberto", nome2 = "Armando";`
`log(nome1 + "," + nome2);`
`let nome = nome1;`
`nome1 = nome2;`
`nome2 = nome;`
`log(nome1 + "," + nome2);`	

`let p = 10;`
`log(p++*2);`
`log(p);`	

`let char1 = '\u8C93', char2 = '\uD83C\uDF4C'` `log(char1, char2);`	

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

10.1 `b = (2 > 3)`

10.2 `x = 3`
`[x, y] = [x + 1, x + 1]`

10.3 `c = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[10]`

10.4 `ci = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".slice(-16, -15).charCodeAt(0)`

10.5 `d = (2.5 * (4.0 ** (11 % 4))) + 'a'.charCodeAt()`

10.6 `ck = 'U42'[+(2 !== 2 + 1)]` `// nota: +"1" => 1,  +false => 0`

10.7 `il = [20, 10, 1, 30, 5].indexOf(1)`

10.8 `p = 'ABC,DEF,GHI,'.split(',')`

10.9 `nome = 'ALBERTO'`
`[x, y, ...resto] = nome`

NOTA: **`let/const`** *e* *`';'`* *omitidos por brevidade*

11. Considerando que inicialmente `vals = [12, 13, 14, 15, 16]` , responda às seguintes questões.

11.1 `vals[2]`

11.2 `vals.slice(2)`

11.3 `vals.slice(-vals.length, 1)`

11.4 `vals.slice(-vals.length + 1, 2)`

11.5 `vals.slice(2, -1)`

11.6 `vals2 = [vals, 17, 18]`

11.7 `vals3 = [...vals, 17, 18]`

11.8
```
nums = vals.slice(1, 4)
nums.splice(1, 2, 4, 5)
nums = ?    vals = ?
```

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.*

12. Um grupo de pessoas participou num jantar em que todos encomendaram o menu turístico e pretende desenvolver um script em JavaScript página para calcular a conta. Para tal, o *script* deve obter o número de pessoas envolvidas no jantar e calcular o valor da conta. O menu custa 15,00 € + IVA por pessoa. Assuma que o IVA é 23% e a gorjeta para o empregado é de 10% sobre o montante total com IVA. O programa deve exibir a despesa total sem IVA e sem gorjeta, o montante de IVA, o valor da gorjeta e a despesa total final. Na primeira versão o utilizador introduz apena os número de pessoas. Na segunda versão pode introduzir preço, IVA e gorjeta, sendo que os valores em cima indicados continuam a ser valores por omissão.

13. Faça um *script* para calcular o preço de venda final de um produto. Para tal recebe o preço do produto, o valor da taxa de IVA a aplicar e (opcionalmente) o valor de um desconto a aplicar ao valor final do produto. O valor do IVA e do desconto deve ser dado em percentagem.

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