



GUIA DE LABORATÓRIO 2.2 - EXTRA

ELEMENTOS BÁSICOS DE JS

(Beta)

OBJECTIVOS

- Introduzir tipos de dados utilitários: `Date` para manipular datas e tempo e `Map` para definir colecções genéricas de elementos
- Breve introdução aos objectos

INSTRUÇÕES

PARTE I – DATAS E TEMPO: OBJECTO GLOBAL DATE E DATE-FNS

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. Também pode utilizar o Node.js, executando o comando `node` na *shell* do sistema operativo.
2. O objecto `Date` representa uma data/hora. Internamente contém o número de milissegundos desde 1 de Janeiro 1970, UTC/GMT. Introduza:

```
>>> let dt1 = new Date()
>>> dt1
2018-11-16T13:43:00.487Z
>>> let dt2 = Date();
>>> dt2
'Sat Mar 16 2019 13:51:37 GMT+0000 (Western European Standard Time)'
>>> [typeof dt1, typeof dt2]
[ 'object', 'string' ]
>>> let [dt3, dt4] = [
  new Date(2020, 2, 2),
  new Date(2020, 2)]
>>> [dt3.getDate(), dt4.getDate()]
[2, 1]
```

*O construtor de uma data deve ser invocado com o operador **new** (este operador será abordado no laboratório sobre objectos; para já, basta saber que ele é utilizado para obter objectos de determinados tipos de dados).*

*Se não utilizarmos **new**, a função `Date` devolve uma string. Esta função construtora aceita diferentes listas de argumentos. Quando não passamos um argumento, ele devolve um objecto que representa a data/hora actual. A outra forma mais habitual consiste em indicar, por esta ordem, ano, "mês", dia, horas minutos, segundos e microsegundos. Apenas ano e "mês" são obrigatórios. "mês" representa o mês de 0 a 11, onde 0 corresponde a Janeiro e 11 a Dezembro.*

Consulte: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

Trabalhar com datas e tempo é uma das necessidades mais comuns à maioria dos programas, em especial em aplicações empresariais e financeiras. À primeira vista lidar com tempo parece simples e directo. Porém, ao fim de pouco tempo, apercebemo-nos de várias e subtis complexidades. Por exemplo, o resultado de somar um dia a uma data depende do dia do mês em questão, do mês em concreto e até do ano (por exemplo, se se trata de um ano bissexto ou não). Se introduzirmos outros aspectos com fusos horários e mudanças de hora, então o problema é ainda mais complicado.

*JavaScript possui um suporte limitado para datas e tempo através do objecto **Date**. Este objecto representa o número de milissegundos desde 1 de Janeiro de 1970, UTC/GMT (Coordinated Universal Time/Greenwich Mean Time, o relógio mundial de referência). Um dia corresponde a 86.400.000 milissegundos. Este objecto tem um suporte limitado para zonas (ie, fusos horários; timezones ou só zones em inglês). Na maioria dos casos utilizamos este objecto para representar uma data/hora sem fusos horários, isto é, pela perspectiva de um observador, como por, exemplo, um calendário ou relógio de parede.*

*É complicado obter informação sobre o dia do ano, semana do ano e dia da semana a partir deste objecto, por isso, mais à já de seguida introduzimos uma alternativa - o módulo **date-fns**. Para contagem do tempo utiliza o calendário ISO-8601, calendário que se baseia na calendário Proléptico Gregoriano (calendário introduzido pelo Papa Gregório XIII em 1582, e estendido posteriormente para lidar com datas anteriores a 1582).*

```
>>> [dt3.getMonth(), dt3.toString()]
[ 2, 'Mon Mar 02 2020 ...resto...' ]

>>> [dt3.getYear(), dt3.getFullYear()]
[ 120, 2020 ]

>>> dt3.getTime()
1583107200000

>>> let dt5 = new Date(2020, 2), dt6 = new Date(2030, 2)

>>> [`${dt4.getFullYear()}/${dt4.getMonth()+1}/${dt4.getDate()}`,
    `${dt5.getFullYear()}/${dt5.getMonth()+1}/${dt5.getDate()}`]
[ '2020/3/1', '2020/3/1' ]

>>> [dt4 === dt4, dt4 === dt5, dt4.getTime() === dt5.getTime()]
[ true, false, true ]

>>> [dt4 < dt5, dt4 <= dt5, dt4 < dt6]
[ false, true, true ]

>>> dt4 >= dt5 && dt4 <= dt5
true
```

3. Datas podem ser convertidas em strings de variadas formas. Destacamos:

```
>>> dt5.toString() // formato ECMA232
'Sun Mar 01 2020 00:00:00 GMT+0000 (Western European Standard Time)'

>>> dt5.toDateString() // formato en_US
'Sun Mar 01 2020'

>>> [dt5.toLocaleString(), dt5.toUTCString(), dt5.toISOString()]
[ '01/03/2020, 00:00:00',
  'Sun, 01 Mar 2020 00:00:00 GMT',
  '2020-03-01T00:00:00.000Z' ]
```

4. Ao contrário da tendência actual na maioria das linguagens, objectos `Date` são mutáveis:

```
>>> let dt7 = dt6

>>> dt6.setFullYear(2021)
1614556800000

>>> [dt6.toString(), dt7.toString()]
["Mon Mar 01 2021", "Mon Mar 01 2021"]
```

5. Observe como são tratadas datas "inválidas":

```
>>> [new Date(2020, 5, 32).toString(), new Date(2020, 12, 20).toString()]
[ 'Thu Jul 02 2020', 'Wed Jan 20 2021' ]
```

6. Objectos `Date` "de base" são algo limitados na funcionalidade (eg, fornecem suporte muito limitado para aritmética de datas), além de alguns métodos são enganadores, inconsistentes e cheios de "surpresas". Vamos ver uma alternativa: a biblioteca `date-fns` (<https://date-fns.org/>). Esta biblioteca não define um tipo de dados para data/horas, antes consiste num conjunto de funções que trabalham sobre o tipo nativo `Date`.

Obtenha `date-fns` a partir do sítio indicado e instale-o no seu ambiente. A partir deste passo assumimos que vai iniciar uma nova sessão do REPL.

NOTAS: Consulte a página para saber como instalar <https://date-fns.org/docs/Getting-Started#installation> . Resumidamente, as alternativas principais são:

- 1) Instalar em Node.js com `npm install [-g | --save]` e depois fazer `require`. Se instalar globalmente, então certifique-se que a variável de ambiente `NODE_PATH` indica o caminho para a directoria global com os módulos.
- 2) Obter o link da CDN da biblioteca, carregá-lo para memória com um ficheiro html com elemento `script` e aceder à consola do navegador
- 3) Obter os links da CDN da biblioteca, descarregar o ficheiro `.js` pretendido com a biblioteca (normal ou "minificado") e proceder como em 2)
- 4) Proceder como em 2) ou 3) para obter a biblioteca, aceder à consola e introduzir:
`document.head.innerHTML += '<script src="CAMINHO_FICHEIRO_OU_LINK_CDN"> </script>';`
- 5) Aceder ao site da biblioteca e abrir aí a consola do navegador

No caso da variante Node, vamos assumir que fez algo similar a `const dateFns = require('date-fns')`. Nas restantes variantes, as variantes "navegador", aceda às funções através do objecto `dateFns`.

7. Introduza agora:

```
>>> dateFns.isToday(new Date())
true

>>> let dt1 = new Date(2020, 1, 28, 15), dt2 = new Date(2020, 1, 29, 10)

>>> [dateFns.isBefore(dt1, dt2), dateFns.isAfter(dt2, dt1)]
[ true, true ]

>>> let dt3 = new Date(2020, 1, 29, 10)
```

```
>>> [dateFns.isEqual(dt1, dt2), dateFns.isEqual(dt2, dt3)]
[ false, true ]

>>> dateFns.max([dt1, dt2]).toDateString()
'Sat Feb 29 2020'

>>> dateFns.formatDistance(dt1, dt2)
'about 19 hours'

>>> dateFns.formatDistanceToNow(dt1)
'about 1 year'

>>> dateFns.formatDistanceToNow(dt1, {addSuffix: true})
'about 1 year ago'

>>> dateFns.formatDistanceToNow(new Date(2018, 0, 5), {addSuffix: true})
'over 3 years ago'

>>> [dt1.toDateString(), dateFns.isFriday(dt1)]
[ 'Fri Feb 28 2020', true ]

>>> [dateFns.lastDayOfMonth(dt1).toDateString(), dateFns.isLastDayOfMonth(dt1)]
[ 'Sat Feb 29 2020', false ]

>>> [dateFns.isLeapYear(dt1), dateFns.getDaysInYear(dt1)]
[ true, 366 ]

>>> [dateFns.isSameDay(dt1, dt2), dateFns.isSameMonth(dt1, dt2),
    dateFns.isSameYear(dt1, dt2)]
[ false, true, true ]
```

8. E, claro, também temos aritmética de datas algo difícil de obter apenas com os métodos "nativos" que integram os objectos `Date`.

```
>>> dateFns.getDayOfYear(dt1)
59

>>> [dateFns.differenceInDays(dt1, dt2), dateFns.differenceInDays(dt2, dt1)]
[ -0, 0 ]

>>> dateFns.differenceInCalendarDays(dt2, dt1)
1

>>> dateFns.addDays(dt1, 1).toDateString()
'Sat Feb 29 2020'

>>> dt1.toDateString()
'Fri Feb 28 2020'

>>> dateFns.addDays(dt1, 2).toDateString()
'Sun Mar 01 2020'

>>> dateFns.addYears(dateFns.addDays(dt1, 1), 1).toDateString()
'Sun Feb 28 2021'

>>> dateFns.addYears(new Date(2020, 1, 28), 1).toDateString()
'Sun Feb 28 2021'
```

```
>>> dateFns.addMonths(new Date(2020, 11, 5), 1).toDateStrinɡ()
'Tue Jan 05 2021'

>>> dateFns.addWeeks(new Date(2020, 11, 1), 1).toDateStrinɡ()
'Tue Dec 08 2020'
```

9. A função `format` permite muitas opções de formatação, um pouco ao estilo do comando `date` dos sistemas operativos Unix:

```
>>> [dateFns.format(dt1, 'dd/MM/yyyy'), dateFns.format(dt1, 'yy -> MM -> dd')]
[ '28/02/2020', '20 -> 02 -> 28' ]

>>> [dateFns.format(dt1, 'dd MMM yyyy'), dateFns.format(dt1, 'dd MMMM yyyy')]
[ '28 Feb 2020', '28 February 2020' ]
```

10. Em Node.js podemos *localizar* a formatação.

NOTA: À data actual - Março de 2019 - apenas instalações com módulos *CommonJS*, como é o caso do *Node.js*, é que suportam localização. Os navegadores não suportam nem vão suportar *CommonJS*.

```
>>> const ptLoc = require('date-fns/locale/pt')

>>> dateFns.format(dt1, 'dd MMM yyyy', {locale: ptLoc})
'28 fev 2020'

>>> dateFns.format(dt1, 'dd MMMM yyyy', {locale: ptLoc})
'28 fevereiro 2020'

>>> dateFns.formatDistance(dt1, new Date(2018, 0, 5), {addSuffix: true, locale: ptLoc})
'daqui a aproximadamente 2 anos'
```

PARTE II – OBJECTOS

11. Objectos serão abordados em laboratório próprio. Aqui fica uma breve introdução. Vamos definir alguns literais de objecto:

```
>>> let pessoa = {nome: "Alberto", idade: 23, cidade: "Lisboa"}

>>> typeof pessoa
'object'

>>> console.log(pessoa.nome, pessoa['cidade'])
Alberto Lisboa

>>> pessoa.apelido = 'Antunes'
'Antunes'

>>> pessoa['nacionalidade'] = 'portuguesa'
'portuguesa'

>>> pessoa
{ nome: 'Alberto', idade: 23, cidade: 'Lisboa', apelido: 'Antunes', nacionalidade: 'portuguesa' }
```

Como referido, *objectos* são uma **associação dinâmica de propriedades**, ou seja, de pares **chave** → **valor**. As *chaves* têm que ser *strings* e devem ser **únicas** (isto é, não é possível introduzir duas *chaves* com o mesmo nome; o valor da segunda *chave* sobrepõe-se ao valor da primeira). Os valores podem pertencer a qualquer tipo de dados.

Objectos são muito úteis em JavaScript para:

- definir uma **agregação** "ligeira" de campos relacionados entre si, um pouco como noutras linguagens utilizamos um registo (record) ou uma estrutura (struct); é neste sentido que vamos utilizar *objectos* nestes primeiros laboratórios
- **coleções genéricas** de elementos onde, ao contrário do que sucede com sequências como arrays, não há uma noção de posição; no entanto, para esta utilização podemos preferir utilizar mapas.
- definir um conceito, no sentido de **programação orientada por objectos**, onde um *objecto* **encapsula** atributos de dados e métodos (ie, funções), ou seja, **estado** e **comportamento**; esta utilização será explorada noutro laboratório.

JavaScript fornece uma notação muito conveniente para definir **literais de objecto** (object literals):

```
let obj = { chave1: valor1, chave2: valor2, .... , chaveN: valorN }
console.log(obj.chave1)
```

As **chaves** têm que ser *strings* ou `Symbol`, porém o JavaScript não assinala um erro se utilizarmos como um *chave* um valor de outro tipo de dados, uma vez que é feita uma conversão automática desse valor para *string*. Também é aceite, como *chave*, um identificador válido da linguagem. Este é depois convertido numa *string*. Ou seja, se a *chave* for um nome válido para uma variável, função, etc., então não precisamos de colocar o nome entre plicas ou aspas. Cada propriedade é acedida através do operador de acesso `.` (ponto) escrevendo `objecto.chave`.

Ao contrário dos valores primitivos, que são passados **por valor**, *objectos* são passados **por referência**. Quer isto dizer que quando associamos um *objecto* a um identificador (eg, a uma variável local, a um parâmetro de uma função - falaremos sobre funções nos próximos laboratórios) estamos a passar o endereço de memória do *objecto*. Isto é, se `var1 = objecto1` e depois `var2 = var1`, então `var1` e `var2` são o mesmo *objecto*. Ao passo que se `var1 = primitivo1` e `var2 = var1`, cada uma das variáveis possui a sua **cópia** do valor primitivo. Por outro lado, isto também significa que dois *objectos* são iguais com `===` se forem exactamente o mesmo *objecto*, e não meramente cópias um do outro. Ou seja, quando os operandos são *objectos*, o operador `===` compara referências.

À semelhança do que sucede com arrays, também existe uma notação especial para **destructuring assignment** com *objectos*. Por exemplo:

```
let obj = {chave1: 5, chave2: 6, chave3: 7}
let {chave1, chave3} = obj // definidas variáveis chave1 e chave2 com valores 5 e 7, respec.
let {chave2} = obj        // definida variável chave2 com valor 6
```

```
>>> let {nome, idade} = pessoa

>>> console.log(`NOME: ${nome} IDADE: ${idade}`)
NOME: Alberto IDADE: 23

>>> let [idade2, pessoa2] = [idade, pessoa]

>>> [idade2, pessoa2.idade] = [24, 24]
[ 24, 24 ]

>>> [idade, pessoa.idade]
[ 23, 24 ]

>>> let [obj1, obj2] = [{a: 10, b: 20}, {a: 10, b: 20}]

>>> let obj3 = obj1
```

```
>>> obj1 === obj2
false

>>> let x = 3, y = -4

>>> let ponto2D = {x, y}, ponto3D = {x, y, z: -109}

>>> console.log(ponto2D, ponto3D)
{ x: 3, y: -4 } { x: 3, y: -4, z: -109 }
```

```
>>> obj1 === obj3
true
```

PARTE III – MAP E SET

12. Mapas permitem associar qualquer tipo de dados a qualquer tipo de dados, e não apenas inteiros (com arrays) ou strings (com objectos) a outros valores.

```
>>> let portos = new Map([ ['ftp', 21], ['ssh', 22], ['smtp', 25], ['http', 80] ])

>>> portos
Map { 'ftp' => 21, 'ssh' => 22, 'smtp' => 25, 'http' => 80 }

>>> portos.size
4

>>> [typeof portos, portos.constructor]
[ 'object', [Function: Map] ]

>>> portos.get('ftp')
21

>>> portos.get('sssh')
undefined

>>> portos.has('ftp')
true

>>> portos.has('sssh')
false

>>> portos.delete('ssh')
true

>>> portos.delete('ssh')
false

>>> portos.set('ftp', 19)
... mapa com ftp => 19 ...

>>> portos.set('pop3', 110)
... mapa com novo par pop3 => 110 ...
```

Consultar estas e outras operações em:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
<https://medium.com/front-end-weekly/es6-map-vs-object-what-and-when-b80621932373>

Um **Map** pertence a uma categoria de tipos abstractos de dados designados de **mapas**. Mapas são colecções de elementos que, à semelhança de objectos, e como o seu nome indica, mapeiam valores, que designamos por **chaves (keys)**, noutros valores, que designamos por ... **valores**. Tal como arrays, e ao contrário de strings, **Mapas** são tipos de dados **mutáveis**. As chaves são **únicas** e são mantida pela ordem de inserção. Todos os tipos de dados podem ser utilizados como chaves, mas idealmente devemos utilizar tipos de dados **imutáveis**.

Criamos objectos a partir do construtor **Map**. Este aceita um iterável de pares chave → valor (veremos depois o que é um iterável; por agora, basta saber que pode ser um array):

```
>>> meuMapa = new Map([[chave1, valor1],
...
[chaveN, valorN]])
```

Um dicionário vazio pode ser obtido com **new Map()**.

13. Podemos aceder separadamente às chaves e aos valores:

```
>>> portos.keys()
[Map Iterator] { 'ftp', 'smtp', 'http', 'pop3' }
```

```
>>> Array.from(portos.keys())
['ftp', 'smtp', 'http', 'pop3' ]
>>> Array.from(portos.values())
[19, 25, 80, 110]
```

14. Podemos converter facilmente mapas em arrays e, tal como com arrays, podemos clonar e concatenar (*merge*) mapas:

```
>>> let portos2 = new Map(portos)
>>> portos2
Map { 'ftp' => 19, 'smtp' => 25, 'http' => 80, 'pop3' => 110 }
>>> portos === portos2
false
>>> let portos3 = new Map([[ 'https', 443 ], [ 'ftp', 21 ]])
>>> let portos4 = new Map([ ...portos2, ...portos3 ])
>>> portos4
Map { 'ftp' => 21, 'smtp' => 25, 'http' => 80, 'pop3' => 110, 'https' => 443 }
```

15. Em teoria, as chaves de um mapa podem ser de qualquer tipo de dados. Na prática, acabamos por utilizar chaves únicas baseadas em strings ou Numbers. Porquê? Porque os mapas utilizam referências para armazenar os valores. Vejamos alguns exemplos:

```
>>> let jogador1 = {id: 'xhp123', nome: 'Alberto'}
>>> let jogador1Copia = {id: jogador1.id, nome: jogador1.nome}
>>> let jogador2 = {id: 'try345', nome: 'Armando'}
>>> let resultados = new Map()
>>> resultados.set(jogador1, 25)
Map { { id: 'xhp123', nome: 'Alberto' } => 25 }
>>> resultados.set(jogador2, 40)
Map { { id: 'xhp123', nome: 'Alberto' } => 25,
      { id: 'try345', nome: 'Armando' } => 40 }
>>> resultados.set(jogador1Copia, 27)
Map { { id: 'xhp123', nome: 'Alberto' } => 25,
      { id: 'try345', nome: 'Armando' } => 40,
      { id: 'xhp123', nome: 'Alberto' } => 27 }
```

A última invocação de `Map.set` deveria ter actualizado o resultado do 'Alberto' ao invés de ter introduzido uma nova associação no mapa.

16. A solução passa por utilizar os ids, que são strings primitivas, como chaves. Indicamos duas implementações possíveis:


```
>>> resultados = new Map()
>>> resultados.set(jogador1.id, 25)
Map { 'xhp123' => 25 }
```

```
>>> resultados.set(jogador2.id, 40)
Map { 'xhp123' => 25, 'try345' => 40 }
```

```
>>> resultados.set(jogador1Copia.id, 27)
Map { 'xhp123' => 27, 'try345' => 40 }
```

```
>>> resultados = new Map()
>>> resultados.set(jogador1.id,
                    [jogador1, 25])
Map { 'xhp123' =>
  [ { id: 'xhp123', nome: 'Alberto' }, 25 ] }
>>> resultados.set(jogador2.id,
                    [jogador2, 40])
Map { 'xhp123' =>
  [ { id: 'xhp123', nome: 'Alberto' }, 25 ],
  'try345' =>
  [ { id: 'try345', nome: 'Armando' }, 40 ] }
>>> resultados.set(jogador1Copia.id,
                    [jogador1Copia, 27])
Map { 'xhp123' =>
  [ { id: 'xhp123', nome: 'Alberto' }, 27 ],
  'try345' =>
  [ { id: 'try345', nome: 'Armando' }, 40 ] }
```

17. Estude o *standard built-in object Set* na MDN (ou noutro local com informação relevante).

PARTE IV – VERDADEIRO OU FALSO...

18. Em JavaScript um valor *falsy* é um valor que é avaliado a *false* em contexto booleano. Um valor é *truthy* se não for *falsy* (princípio Lili Caneças).

```
>>> let bool = Boolean
>>> [bool(null), bool(undefined), bool(0), bool(NaN), bool('')]
[ false, false, false, false, false ]
>>> [!null, !undefined, !0, !NaN, !'']
[ true, true, true, true, true ]
>>> [!!null, !!undefined, !!0, !!NaN, !!'']
[ false, false, false, false, false ]
>>> [bool({}), bool([]), bool(37), bool(-2), bool("alberto"), bool(Infinity),
    bool(new Date())]
[ true, true, true, true, true, true, true ]
```

Um "contexto booleano" é um contexto onde se espera um valor lógico. É o que sucede com instruções de decisão, como a instrução `if`, ou instruções de repetição, como o `while` ou o `for`, instruções que abordaremos mais à frente.

EXERCÍCIOS DE REVISÃO

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

1.1 `b = (opcao !== 't' && opcao !== 'T')`

Primeiro assumo que o valor de opcao é 'T' e que depois é 'F'.

1.2 `[x, y, z] = [3, 'alberto', [1, 2, 3]]`

`[b, c, d] = [!(x !== 3), !(y === 'armando'), !!z]`

1.3 `x = 3, y = undefined`

`z = x || 14`

`w = y || 15`

`p = (x % 2 === 1) && 16 || 17`

1.4 `c = 'X++P++T++0'.split('+').join('.')`

1.5 `p = [1, 2, 3, 4].slice(1).indexOf(4) + [1, 2, 3, 4].slice(1,3)[1]`

NOTA: neste e em alguns exercícios omitimos `let/const` e `;` por brevidade

2. Indique como remover elementos duplicados de um array e de uma string com `Set`?

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

<pre>let obj = {valor1: 12, valor2: 14, codigo: 'valor1'} obj[obj.codigo] = 74 obj[obj[obj.codigo]] = 'armando' log(obj)</pre>	
<pre>let [x, cliente] = [12, {nome: 'Ana', idade: 23}] let [y, cli] = [x, cliente] y += 10 ++cli.idade console.log(x, cliente)</pre>	
<pre>let [nome1, nome2] = ['Alberto', new String('Alberto')] log(nome1 === nome2, typeof nome2) log(nome1.toUpperCase() === nome2.toUpperCase(), typeof nome2.toUpperCase())</pre>	
<pre>let vals = [1, 2, 3, 4, 5] let [x, y, ...z] = vals let nums = [...z, x, y] log(nums.slice(-2))</pre>	

<pre>let procs1 = new Map([['ls', 192], ['grep', 321], ['init', 1]]) procs1.set('ls', 193) procs1.set('mkfs', 14) console.log(Array.from(procs1).join('+'))</pre>	
<pre>const date = new Date(2019, 0, 1) const date2 = dateFns.addDays(date, 3) log(dateFns.eachDay(date, date2))</pre>	
<pre>let matriz = [[10, 1, 8], [0, 12, 4]] c = matriz.slice() c[c.length-1] = [1, 1, 1] log(matriz[matriz.length-1]) c = matriz.slice() c[c.length-1][2] = 14 log(matriz[matriz.length-1])</pre>	

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.

- Utilizando `date-fns`, faça uma calculadora de dias entre duas datas introduzidas num formulário. A sua página deve indicar o número de dias decorridos entre as duas datas. Se a segunda data ficar vazia, assuma que toma o valor da data actual.
- Faça um tradutor de meses localizado. O utilizador introduz o mês e um código de língua (eg, 'en', 'pt', 'fr', etc) e a página indica o nome do mês na língua introduzida. Não é necessário utilizar `Date` ou `date-fns`. Recorra objectos e/ou mapas e/ou arrays.

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]: Grammar and Types @ MDN, https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Grammar_and_Types