Algorithmique et programmation : JavaScript

ENSEEIHT FULLSTACK – JavaScript

Xavier Crégut <prenom.nom@enseeiht.fr>

Sommaire

Motivation

- Apprendre les aspects Algorithmique & Programmation de JavaScript
  - Les aspects algorithmique étant connus, on se concentre sur la syntaxe
  - Et on détaillera les nouveaux concepts
- Principaux concepts :
  - Algorithmique
  - Collections
  - Programmation fonctionnelle
  - DOM (Document Object Model)
  - Programmation asynchrone
  - Programmation objet

Carte d'identité

Origines

- 1994 par Netscape
- Au départ pour rendre vivantes les pages Web
- Maintenant aussi sur le back-end
- Normalisé par ECMA International : ECMA-262, 860 pages

Parenté : Java

- mais de très loin !
- le nom Java était à la mode
- les structures de contrôle viennent de Java qui les avait empruntées à C
- certaines fonctions de la biblothèque (parseInt...)
- mais un langage très différent !

Principales caractéristiques

- Langage de script embarqué dans un hôte (navigateur, Node.js, etc.)
- Langage à typage dynamique
- Programmation impérative, fonctionnelle et objet par prototype
- Programmation asynchrone
- Compatibilité avec les principaux navigateurs avec les normes
- Limiter les erreurs dans le navigateur, donc comportements par défaut, erreurs silencieuses...

Utilisation dans un navigateur : élément `script`

Directement dans un fichier HTML

```
<script type="text/javascript">
    // <![CDATA[    plus sûr, surtout si <, > ou & utilisés
    ... le code JS ...
    // ]]>
</script>
```

Dans un fichier externe

```
<script src="un_fichier.js" type="text/javascript"></script>
```

- Le fichier `un_fichier.js` contient le code JS.
- Attention : ne pas utiliser la forme contractée `<script />`

Exemple (on ne fait plus de page comme ça !)

fichier navigateur-es.html

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="utf-8" />
    <title>Un titre</title>
  </head>
  <body>
    <h1>Le contenu qui suit est engendré par script</h1>
    <script type="text/javascript"> // <![CDATA[
      document.writeln("<p>un paragraphe</p>");
      document.writeln("<h1>Salutations</h1>");
    // ]]> </script>
    <script src="navigateur-es.js" type="text/javascript"></script>
    <b>Fin.</b>
  </body>
</html>
```

- C'est lui que l'on ouvre dans un navigateur
- Il utilise l'élément script en précisant le type du script
  - soit du script écrit directement dans la page
  - soit dans un fichier externe
- Le chargement d'un module se mettrait dans l'entête (head)

Fichier navigateur-es.js

```
"use strict";   // inutile si module car implicite....
let encore;
do {
    const nom = prompt("Votre nom ? ");
    alert("Bonjour " + nom + " !");
    document.writeln(`<p>Bonjour ${nom}</p>`);
    encore = confirm("On continue ?");
} while (encore);
document.write(`<p>Bonjour à tous</p>\n`);
```

- Les alert, prompt et confirm sont désagréables pour l'utilisateur.
- Ils sont en plus bloquants : rien d'autre ne peut être fait.

Exercice

1. Lire et comprendre le code précédent
2. Charger le fichier `navigateur-es.html` dans le navigateur
3. Utiliser F12 pour afficher la console développeur puis onglet console
4. Supprimer le `let` ou le `const` (ou les deux) et recharger la page
5. En mode « strict » les variables doivent être déclarées (`let` ou `const`) !

Utilisation via Node.js

Installer Node.js (installation locale via nvm)

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
$ source ~/.bashrc    # prendre en compte les modifications (ou nouveau terminal)
$ nvm install 22      # Installe la version 22 / ou nvm install node
$ nvm use 22          # Utilise la version 22
$ which node
/home/cregut/.nvm/versions/node/v22.11.0/bin/node
$ node                # Lancer Node.js
> 2 + 3               // une expression JS
5
> .exit              # pour quitter.  On peut aussi faire : Ctrl-d
```

Installer Node.js (sur le système)

On peut aussi l'installer en suivant les instructions sur le site de Node.js.

Sommaire

Les commentaires

Commentaire jusqu'à la fin de la ligne

```
// Ce commentaire se termine avec la ligne
```

- À utiliser pour les commentaires issus des raffinages
- À utiliser en fin de ligne pour expliquer, justifier l'instruction

Commentaire sur plusieurs lignes

```
/* Ce commentaire peut se continuer
   sur plusieurs lignes. On le termine avec : */
```

- Attention, ces commentaires ne peuvent pas être imbriqués.

Commmnentaires de documentation

```
/** Retourne le cube d'un nombre.
* @param {number} x - le nombre.
* @return {number} Le cube de x.
*/
function cube(x) {
    return x ** 3;
}
```

- Utilisés par exemple par JSDoc
- Notez les deux * au début du commentaire
- Une syntaxe particulière pour :
  - décrire les paramètres :
    `@param {type} nom - description`
  - la valeur de retour
  - ...
- `jsdoc -d=doc fichier.js ...`

Point-virgule, retour à la ligne et indendation

Point-virgule en fin d'instruction

Le **point-virgule « ; » est la norme** en fin d'instruction
Souvent on peut l'omettre :

- JS définit des **règles d'insertion** d'un point-virgule
- Ces règles sont **clairement définies**
- Donc on peut savoir quand il est nécessaire de les mettre ou pas
- IMHO, ne pas les mettre c'est :
  - jouer au Geek
  - prendre des risques si on ne connait pas bien les règles
- voir ici et là

Mon conseil : Mettre les points-virgules !

Indentation

- Les retours à la ligne et les blancs sont libres en JS
- On s'en sert pour rendre le code plus lisible
- Par exemple, en utilisant la même indentation pour les instructions du même bloc

Types élémentaires

boolean (Booléen)

- 2 valeurs possibles : `false` et `true`
- opérateurs : `&&` (et) `||` (ou) `!` (non)
- `typeof true === 'boolean'`
- `typeof objet` : obtenir le type de objet

number (nombres)

- Pas de distinction entre entier et nombre à virgule flottante
- `typeof 5 === 'number' && typeof 2.5 === 'number'`
- opérateurs arithmétiques usuels + - * /
- % reste de la division : `5 % 3 === 2 && 4.5 % 2 === .5`
- ** : exponentiention (`2.0 ** 3 === 8`)
- `Math.floor(x)` : la partie entière (`Math.floor(4.99) === 4` et `Math.ceil(4.99) === 5`)
- Conversions en nombre : `Number`, `parseInt` et `parseFloat` :
  - `Number('3.5') === 3.5`
  - `Number(true) === 1` et `Number(false) === 0`
  - `isNaN(Number('3.5x'))`
  - `parseInt('3.5x') === 3`
  - `parseFloat('3.5x') === 3.5`
  - `~~3.5 === 3`, `~~"3.5" === 3` et `~~"3.5x" === 0`
  - `"25" * 1 === 25` (opérateurs arithmétiques sauf +, concaténation)

string (Chaînes de caractères)

- Pas de distinction entre caractère et chaîne de caractères
- Possibilité d'utiliser des apostrophes ou des guillemets pour délimiter la chaîne
- `typeof 'nom' === 'string' && typeof "nom" === 'string'`
- `'\n'` (retour à la ligne), `'\t'` (tabulation), `'\\'` (back-slash), etc.

Opérateurs

- + pour la concaténation :
  - `'java' + 'script' === 'javascript'`
  - `'a' + 2 === 'a2'`
- De nombreuses méthodes dans l'objet String
  - `' a b  c  '.trim() === 'a b  c'`
  - `'Mot'.toLowerCase() == 'mot'`
  - `'mot'.small() === '<small>mot</small>'`
  - ...

Chaîne de gabarits

- Chaîne avec expression JS à l'intérieur : délimiteur ` (accent grave, back-quote)
- Exemple : `` `2 + 3 donne ${2 + 3}` === '2 + 3 donne 5' ``
- Le contenu des `${...}` est remplacé par son évaluation en JS
- Ces chaînes peuvent s'étendre sur plusieurs lignes

égalité (==) vs égalité strict (===)

Égalité historique : == et !=

- Réalise des conversions implicites sur les opérandes pour décider si vrai ou faux
- Conduit à des résultats contre-intuitifs (voir fin de l'exemple ci-dessous)
- $\Longrightarrow$ Création de === et !==

Égalité strict : === et !==

- Comme == et != sans les conversions implicites
- e1 === e2 vrai si e1 et e2 du même type et e1 == e2

Exemples : voir Which equals operator

```
2 == '2'    // true
2 === '2'   // false

'' == 0     // true
0 == '0'    // true
'' == '0'   // false !   Non transitif
```

Conseil

- Toujours utiliser === et !==

Tableaux

Principe

Représenter une collection d'objets avec une position (un indice) pour chaque objet.

Création

```
let t1 = new Array();   // un tableau vide : [],  t1.length == 0
let t2 = new Array(10); // [ <10 empty items> ],  t1.length == 10
let t3 = new Array(1, 2, 'trois');  // [ 1, 2, 'trois' ]
let t4 = Array(1, 2, 'trois');      // ou sans `new`
let t5 = [1, 2, 'trois'];   // [ 1, 2, 'trois' ]       !!! forme à préférer !!!
```

Indices

```
let premier = t4[0];    // premier === 1 // premier élément à l'indice 0
let x = t4[156];        // x === undefined : un indice invalide

t4[1] = 4;              // [ 1, 4, 'trois' ]
t4[100] = 0;            // [ 1, 4, 'trois', <97 empty items>, 0 ], length === 101

delete t4[100];         // [ 1, 4, 'trois', <98 empty items> ], length === 101;

t4.length = 2;          // [ 1, 4] : plus que 2 éléments !
```

Remplacer une partie d'un tableau : splice

splice(début, nombre, ...nouveaux)

- remplace les `nombre` éléments commençant à l'indice `début` les les autres paramètres
- retourne le tableau des éléments remplacés.

Exemples

```
// insérer : avec `nombre` === 0
t3.splice(1, 0, -1, -2);          // [1, -1, -2, 2, 'trois']

// remplacer : avec `nombre` > 0
t3.splice(2, 2, 'a', 'b', 'c');  // [1, -1, 'a', 'b', 'c', 'trois']

// début < 0 : on compte à partir de la fin (tab.length + debut)
t3.splice(-2, 2, 'C', 'D');       // [1, -1, 'a', 'b', 'C', 'D']

// supprimer : avec `nombre` > 0 et pas d'éléments donnés
t3.splice(2, 4);                  // [1, -1]

// debut trop grand => debut = tab.length
t3.splice(10, 2, 4, 5, 6);        // [1, -1, 4, 5, 6]
```

Quelques méthodes (d'autres à suivre. . .)

```
let taille = t4.push(7);    // t4 : [ 1, 4, 7 ], taille : 3
x = t4.pop();               // t4 : [ 1, 4 ], x : 7
    // ==> On retrouve les opérations d'une pile (sommet à droite, grands indices)

premier = t4.shift();       // t4 : [ 4 ], premier : 1
taille = t4.unshift(9);     // t4 : [ 9, 4 ], taille : 2

t4.push('sept');            // t4 : [ 9, 4, 'sept' ]
let s = t4.join(" -> ");    // s === '9 -> 4 -> sept'

t4 = [1, 7, 3, 18];
t4.reverse();               // t4 : [ 18, 3, 7, 1 ]

t4.sort();                  // t4 : [ 1, 18, 3, 7 ]    // logique ?

t4.sort(function (g, d) { return g - d }) // t4 : [ 1, 3, 7, 18 ]
    // Le paramètre est la fonction qui compare deux éléments g et d.
    //     résultat = 0 alors g === d
    //     résultat < 0 alors g < d
    //     résultat > 0 alors g > d

let ist4tab = Array.isArray(t4);    // ist4tab: true
let is5tab = Array.isArray(5);      // is5tab: false
```

Objet (première approche)

Principe

Un objet est un tableau associatif (un dictionnaire).

Exemple

```
let r1 = { x: 5, y: 10, direction: "est", 'z': 0, 10 : 'ok?' } ;
// r1 : { '10': 'ok?', x: 5, y: 10, direction: 'est', z: 0 }    // z et non 'z'
let x = r1['x'];     // x === 5
x = r1['abc'];       // x === undefined
x = r1.x;            // x === 5 notation objet (plus légère)

r1['x'] = 6;         // r1.x === 6
r1.x = 7;            // r1.x === 7

let v = r1[10];      // v === 'ok?'
// v = r1.10;        // erreur de syntaxe

delete r1.x;         // r1 : { '10': 'ok?', y: 10, direction: 'est', z: 0 }
```

Et les méthodes ?

Un attribut dont la valeur est une fonction... (Voir la partie programmation objet)

Exprimer des assertions

Avec JS : Assert

- Pour documenter le code (non évalué) :

```
Assert: 3 + 2 === 5;
```

console.assert

- Si la condition est fausse, elle affiche dans la console (error)
- Pas d'interruption du programme

```
console.assert(2 + 3 === 5);     // vrai => pas d'effet
console.assert(2 * 3 === 5);     // faux => console.error(...)
```

Avec le module assert de Node.js

- Interruption de l'exécution si la condition est fausse
- De nombreuses variantes suivant les tests à faire

```
import { strict as assert } from 'assert';  // syntaxe ES6

assert(3 + 2 === 5);        // ou assert.ok(3 + 2 === 5);
assert.equals(3 + 2, 5);    // ok
assert.equals(3 * 2, 5);    // lève une exception
```

- Installer le module esm et faire `require('esm')` pour utiliser la syntaxe ES6 des modules.

Module assert : affichage quand une erreur est détectée

Exécution de l'exemple précédent

```
> node exemple-assert.mjs
node:internal/modules/run_main:122
    triggerUncaughtException(
    ^

AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

6 !== 5

    at file:///home/cregut/projects/ens/js/cours/exemples/cours/exemple-assert.mjs:5:8
    at ModuleJob.run (node:internal/modules/esm/module_job:268:25)
    at async onImport.tracePromise.__proto__ (node:internal/modules/esm/loader:543:26)
    at async asyncRunEntryPointWithESMLoader (node:internal/modules/run_main:116:5) {
  generatedMessage: true,
  code: 'ERR_ASSERTION',
  actual: 6,
  expected: 5,
  operator: 'strictEqual'
}

Node.js v22.11.0
```

Bloc

Définition

Un bloc est délimité par les accolades

```
{ //---------------------------------- début du bloc (après {)
    console.log("Instruction 1");
    console.log("Instruction 2");
    console.log("Instruction 3");
} //---------------------------------- fin du bloc (avant })
```

Convention

Les instructions d'un même bloc ont la même indentation

Intérêt

- Regrouper les instructions (voir structures de contrôle)
- Limite la portée des variables (voir déclaration de variables)

Variables et constantes

Déclarer une variable : trois mots-clés

- `var` : historique mais problématique $\Longrightarrow$ préférer `let`
- `let` : déclarer une variable dont la portée est le bloc
- `const` : déclarer une variable toujours associée au même objet (une constante !)

```
const TVA = 0.20;    // une constante
let prix = 99.90;        // une variable
let prix_ht = prix / (1 + TVA);
prix = 199.90;           // on change sa valeur
prix_ht = prix / (1 + TVA);
TVA = .196;      // Interdit : TVA est const ==> TypeError

let x = 5, y = 7;    // plusieurs variables sur une même ligne
let z;           // variable indéfinie : z === undefined
let t, u, v = 5;     // t et u undefined et v === 5
let x = 3;       // interdit : x déjà déclaré ==> SyntaxError
```

Valeurs prédéfinies

- `undefined` : une variable qui n'a pas été initialisée
  - mais on peut explicitement affecter `undefined` à une variable
- `null` : une valeur particulière qui signifie "pas d'objet associé"

Portée des variables

let et const

- La portée de la variable est son bloc de déclaration
- Interdit de déclarer deux variables avec le même nom dans le même bloc

```
{ //------------------------------------ début du bloc 1
    assert.throws(() => x, ReferenceError); // x n'existe pas encore
    let x = 5, y = 7;
    { //------------------------------------ début du bloc 2
        let x = 'X';          // masquage de x du bloc 1
        assert.equal(x, 'X');
        assert.equal(y, 7);  // accès à y du bloc 1 (englobant)
        const n = 1;          // locale à ce bloc
        assert.equal(n, 1);
    } //------------------------------------  fin de la portée de x ('X') et n
    assert.equal(x, 5);       // le x de ce bloc (x bloc 2 n'existe plus)
    assert.throws(() => n, ReferenceError);     // n non définie (hors bloc 2)
} //------------------------------------  fin de la portée de x (5) et y
```

const

```
    const MAX = 10;
    // MAX = 5; // TypeError: Assignment to constant variable.
    assert.throws(() => MAX = 5, TypeError);
```

var

- La portée de la variable est la fonction (ou le script).
- On peut définir plusieurs fois la même variable.
- Peut conduire à des erreurs difficiles à identifier (voir fermeture)
- $\Longrightarrow$ préférer `let` et `const`

```
{ //------------------------------------ début du bloc 1
    assert.equal(x, undefined); // on peut utiliser x avant son var (hoisting)
    assert.equal(n, undefined); // on peut utiliser n avant son var (hoisting)
    var x = 5;
    { //-------------------------------------- début du bloc 2
        var x = 'X';        // masquage ? Non
        assert.equal(x, 'X');
        var n = 1;          // locale à ce bloc ? Non
        assert.equal(n, 1);
    } //-------------------------------------  fin de bloc 2
    assert.equal(x, 'X');       // on voit la modif faite dans bloc 2
    assert.equal(n, 1);         // n existe toujours
    var x = 7;                  // pas de problème, toujours même x
    assert.equal(x, 7);
} //-----------------------------------  fin de bloc 1
```

Opérateur (plus prioritaire en haut), plus complet

P	A	Operator type	Individual operators
20	G	member	`. []`
20	G	call / create instance	`() new`
18	G	negation/increment	`+ ++`
17	G	negation/increment	`! ~ - + typeof void delete`
16	D	exponentiation	`**`
15	G	multiply/divide	`* / %`
14	G	addition/subtraction	`+ -`
13	G	bitwise shift	`<< >> >>>`
12	G	relational	`< <= > >= in instanceof`
11	G	equality	`== != === !==`
10	G	bitwise-and	`&`
9	G	bitwise-xor	`^`
8	G	bitwise-or	`|`
7	G	logical-and	`&&`
6	G	logical-or	`||`
4	D	conditional	`?:`
3	D	assignment	`= += -= **= *= /= %= <<= >>= >>>= &= ^= |= &&= ||=`
1	G	comma	`,`

- P : Priorité (1 faible), A Associativité : G pour Gauche et D pour droite
- Gauche : 2 - 3 - 4 = (2 - 3) - 4 et Droite : 3 ** 1 ** 2 = 3 ** (1 ** 2)

Entrées-sorties

Elles dépendent du contexte d'exécution du script : navigateur, Node.js, etc.

L'objet console (navigateur et Node.js)

- Dans un navigateur, la console s'ouvre via F12
- console.log(...) : afficher sur la console
- console.info(...), warn(...), error(...) : différents niveaux de journalisation
- console.table(...) : afficher un tableau (ou un objet)
- console.assert(bool, str) : affiche str ssi bool faux

Dans un navigateur

- alert(), confirm(), prompt() : ne sont plus utilisés dans une vraie application
  - `let nom = prompt("Nom ? ");`
  - `alert("Bonjour" + nom + " !");`
  - `encore = confirm("On continue ?");`
- la console : log, etc.
- page HTML dont formulaires (voir DOM et événements) : document

Affectation

Affectation

```
let x;
x = 5;
x = 'X';
```

- **But :** Changer la valeur associée à une variable.
- La valeur d'une affectation est la valeur affectée (l'affectation est une expression).
- Typage dynamique : le type de la variable est le type de sa valeur

Chaîner les affectations

```
let x, y, z = 5;
x = y = z;  // donner la valeur de z à x et y.
// x = (y = z)
```

Formes contractées

```
let x = 1;
x += 4;     // x === 5   car x = x + 4;
x *= 2;     // x === 10  car x = x * 2;
```

Pré/post incrémentation/décrémentation

```
let a = 5, b = 7, r, x, y;
r = ++a + b--;                // r === 13 && a === 6 && b === 6
a = 5; b = 7;
r = (x = ++a) + (y = b--); // r === 13 && a === 6 && b === 6 && x === 6 && y === 7
```

Conditionnelle `if`

```
if (x > max) {
    max = x;
}
```

ou (déconseillé) :

```
if (x > max)
    max = x;
```

```
if (n1 === n2) {
    resultat = "égaux";
} else {
    resultat = "différents";
}
```

```
if (n === 0) {
    return 'nul';
} else if (n < 0) {
    return 'négatif';
} else {
    return 'positif';
}
```

- La condition est forcément entre parenthèses
- On peut ne pas mettre les accolades s'il n'y qu'une seule instruction (**déconseillé**)
- **Conseil :** Toujours mettre les accolades sauf pour SinonSi (`else if`)
- **Remarque :** `jslint` signale que les `else` sont inutiles après un `return`

Si arithmétique

```
individu = (age >= 18 ? "majeur" : "mineur");
```

- Syntaxe : `condition ? valeur_vrai : valeur_faux`
- Intérêt : C'est une expression, pas une instruction

Conditionnelle `switch`

```
let r;
switch (c) {
case 'o':
case 'O':
    r = "affirmatif";
    break;
case 'n':
case 'N':
    r = "négatif";
    break;
default:
    r = "!?!?!?!?";
}
```

- L'expression dans les parenthèses du `switch` est évaluée
- Sa valeur est comparée successivement aux différents `case`
- L'exécution continue sur le premier `case` qui correspond
- Si aucun `case` ne correspond, l'exécution continue dans `default`
- Un `break` fait sortir de `switch`
- En l'absence de `break`, l'exécution continue sur les cas suivants
- Que donne l'exécution si c vaut `'o'`, `'N'` ou `'x'` ?

Répétition `while`

```
const taux = 0.03;
const objectif = 10000;
let capital = 5000;
let nbAnnees = 0;
while (capital < objectif) {
    nbAnnees++;
    capital = capital * (1 + taux);
}
```

- Un **TantQue** classique !

```
while (condition) {
    instructions;
}
```

Répétition do ... while

```
// Tirer des nombres aléatoires
let nb;
do {
    nb = Math.floor(Math.random() * 100);
    assert.ok(0 <= nb && nb < 100);
    console.log("Nouveau nombre : " + nb);
} while (nb <= 90);
```

```
do {
   instructions;
while (condition);
...
```

- Un **Répéter ... JusquÀ ...**.
- Ou plutôt un **Répéter ... TantQue ...**
- Les instructions seront exécutées une fois au moins (condition après les instructions)
- Quand l'évaluation de la condition est fausse, sortie de la boucle

Répétition `for` (forme historique)

Une généralisation du `while` qui intègre :

1. l'initialisation des variables de boucle,
2. la condition de continuation (on reste dans la boucle) et
3. le passage au cas suivant (incrémentation)

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

Forme générale :

```
for (initialisation; continuation; incrémentation) {
    instructions;
}
```

Équivalente à :

```
{   //! limiter la portée des variables déclarées dans initialisation
    initialisation;
    while (continuation) {
        instructions;
        incrémentation;
    }
}
```

Répétition `for` (trois formes)

Objectif : Calculer la somme des éléments d'un tableau `tab`.

Classique

```
let somme = 0;
for (let indice = 0; indice < tab.length; indice++) {
    somme += tab[indice];
}
```

Sur les indices d'un tableau : `for in`

```
let somme = 0;
for (const indice in tab) {
    somme += tab[indice];
}
```

Sur les éléments d'un tableau : `for of`

```
let somme = 0;
for (const element of tab) {
    somme += element;
}
```

break et continue

- `break` : sortir de la répétition courante
- `continue` : passer au cas suivant (i.e. aller à l'évaluation de la composition)
- Ne pas en abuser !

```
première: for (let i = 0; i < 7; i++) { // for étiquetté « première »
    console.log("i = " + i);
    for (let j = 0; j < 4; j++) {
        console.log("j = " + j);
        if (j == i) {
            continue première; // ==> la boucle « première »
        }
    }
}
console.log("Fin !");
```

Sommaire

- Définition
- Égalité sur les objets
- Valeur par défaut
- Nombre variable de paramètres
- Destructuration
- Expression fonction
- Fonctions fléchées
- Truthy et Falsy
- Bilan passage de paramètres

Fonctions

```
/** le suivant de nombre à la distance donnée.
 * Exemple le suivant de 1 à la distance 5 est 6.
 * @param {number} nombre - le nombre dont on cherche le suivant
 * @param {number} distance - la distance entre nombre et son suivant
 * @return {number} le suivant de nombre à la distance demandée
 */
function suivant(nombre, distance) {
    return nombre + distance;
}

let s1 = suivant(1, 5); // s1 === 6
```

Très classique. . .

- documentation : un commentaire structuré devant la fonction
- signature avec les paramètres formels : nombre, distance
- appel avec les paramètres effectifs : 1, 5
- en JS, on ne peut pas nommer les paramètres, seulement positionnels

Exercices

Conversion entre pouces et centimètres

Étant donnée une longueur exprimée en pouce (p), centimètre (c) ou mètre (m), l'afficher en pouce et en centimètre (voir les exemples suivants). L'unité pourra être donnée en minuscule ou majuscule.

```
> pouceCentimetre(1, 'p');
1 p = 2.54 cm
> pouceCentimetre(25.4, 'c');
10 p = 25.4 cm
```

```
> pouceCentimetre(2, 'm');
78.7402 p = 200 cm
> pouceCentimetre(1, 'k');
0 p = 0 cm
```

Égalité sur les objets

Égalité de deux tableaux de nombres

Écrire une fonction qui renvoie vrai si deux tableaux de nombres sont égaux.

1. Peut-on utiliser == ou === ? La réponse doit être justifiée.
2. Écrire cette fonction et la tester.
   - Avec un `while`
   - Avec un `do ... while`
   - Avec chacune des trois formes du `for`
3. Quelle version préférer ?

(devoir) Même exercice (questions 2 et 3) pour calculer la fréquence d'un entier dans un tableau.

Comportement de == et ===

Principe

Les opérateurs d'égalité compare la valeur de la variable, pas l'objet associé

Conséquence

- Sur les types élémentaires (boolean, number, string) compare bien la valeur : égalité logique
- Pour les tableaux et les objets, compare les adresses : égalité physique

Exemples

```
assert.ok("bonjour" === "bonjour");
const salut = "Bonjour";
const petitSalut = salut.toLowerCase();
assert.ok(petitSalut == "bon" + "jour");
assert.ok(petitSalut === "bon" + "jour");
assert.ok([1, 2, 3] !== [1, 2, 3])
const o1 = { x: 5, y: 5 };
const o2 = { x: 5, y: 5 };
assert.ok(o1 != o2);
assert.ok(o1 !== o2);
assert.deepEqual(o1, o2);   // compare bien le « contenu » des deux objets
```

Valeur par défaut

```
function suivant(nombre, distance = 1) {
    return nombre + distance;
}

const s1 = suivant(6);       // s1 === 7
const s2 = suivant(1, 5);    // s2 === 6
```

- Classique : si le paramètre effectif n'est pas donné à l'appel, la valeur par défaut est utilisée.

Nombre variable de paramètres

```
function join(separator) {
    let resultat = '';
    for (let index = 1; index < arguments.length; index++) {
        if (index > 1) {
            resultat += separator;
        }
        resultat += arguments[index];
    }
    return resultat;
}

const s1 = join(" -> ", 1, 2, "trois");
console.log("s1 = " + s1);
// console: s1 = 1 -> 2 -> trois
```

- arguments : le tableau des paramètres effectifs
- arguments.length : le nombre **total** de paramètres effectifs donc
- si on utilise des paramètres explicites, il faut en tenir compte (ici index commence à 1)
- mais on peut faire mieux maintenant. . .

Destructuration et ... (spread/rest operator) — Array

```
const a = [1, 2, 3, 4];

const [x, y] = a;   // initialiser x et y à partir des premiers éléments de a
assert.equal(x, 1);
assert.equal(y, 2);

let [u, v, ...w] = a;   // ...w
assert.equal(u, 1);
assert.equal(v, 2);
assert.deepEqual(w, [3, 4]);

[u, , v] = [2, 5, 3];   // changer u et v et ignorer une valeur du tableau
assert.equal(u, 2);
assert.equal(v, 3);

assert.equal(Math.max(...w), 4);    // ~ Math.max(3, 4)

[u = 1, v = 2] = [5];   // avec valeurs par défaut
assert.equal(u, 5);
assert.equal(v, 2);

[u, v] = [v, u];        // permuter les valeurs des variables
assert.equal(u, 2);
assert.equal(v, 5);
```

Nombre variable de paramètres

```
function join(separator, ...objets) {
    let resultat = '';
    for (const objet of objets) {
        resultat += (resultat === '' ? '' : separator) + objet
    }
    return resultat;
}
const s1 = join(" -> ", 1, 2, "trois"); // s1: s1 = "1 -> 2 -> trois"
```

- ... ne peut apparaître qu'une fois, toujours devant le dernier paramètre.

Concaténer des tableaux (voir aussi concat)

```
const t1 = ['a', 'b', 'c'], t2 = ['e', 'f'];
const t3 = [...t1, ...t2];  // t3: ['a', 'b', 'c', 'e', 'f'], t1 et t2 inchangés
console.log('t3 =', t3);    // t3 = ['a', 'b', 'c', 'e', 'f']
console.log('t3 =', ...t3); // t3 = a b c e f
const t4 = [...t1, ...t2, 4, ...t2];
```

Destructuration et ... (spread/rest operator) — objet

```
const o = {a: 1, b: 2, c:3, d: 4};

assert.throws( () => a, ReferenceError);  // la variable `a` n'existe pas

const {a, b} = o;   // définition de a à partir des valeur de o (même clé)
assert.equal(a, 1);
assert.equal(b, 2);

let {x = 7, c = 9, d, ...y} = o;    // valeur par défaut et spread operator
assert.equal(x, 7);    // valeur par défaut utilisée
assert.equal(c, 3);    // valeur dans o
assert.equal(d, 4);    // valeur dans o
assert.deepEqual(y, {a: 1, b: 2}); // les autres entrées de o

({c, d} = {c: 0, d: 1});    // parenthèses obligatoires
assert.ok(c === 0 && d === 1);

const {a: u = 9, b: v = 8, z: w = 7} = o;
assert.equal(u, 1);     // u reçoit la valeur de a dans o, 9 ignorée
assert.equal(v, 2);     //
assert.equal(w, 7);     // valeur par défaut car z non dans o
```

Quelques utilisations intéressantes à la fin de Affecter par décomposition

Exercice destructuration (décomposition)

Quelles sont les variables et leurs valeurs après les affectations suivantes ?

```
const [ a = 1, b = 3, c ] = [7, 4, 2, 9, 8, 3];

const [ e = 1, f, ...g ] = [7, 4, 2, 9, 8, 3];

const [ u, , , v = 4 ] = [7, 4, 2, 9, 8, 3];

const [ x, , , y = 4, z ] = [8, 2, 9];



const { a, b: c = 2, d: e = 4, f, ...g } = { a: 5, b: 6, c: 7, e: 8, h: 9};
```

Expression fonction

```
const suivant = function (nombre, distance = 1) {
    return nombre + distance;
}

let s1 = suivant(1, 5); // s1 === 6
let s2 = suivant(6);    // s2 === 7

(function (x, n) {
    const xn = x ** n;
    console.log(`${x}^${n} = ${xn}`);
})(2, 3);
// console: 2^3 = 8
```

- On pourrait donner un nom à la fonction (dans les deux cas)
  - utile si la fonction est récursive !
- Attention dans la fonction utilisée directement, ne pas oublier les parenthèses autour de la définition de la fonction..

Fonctions Fat arrow (fonctions fléchées)

- Notation plus concise (depuis ES6)
- On utilise => (d'où le nom fat arrow) et plus de mot-clé `function`

```
const suivant = (nombre, distance = 1) => {
    return nombre + distance;
}

let s1 = suivant(1, 5); // s1 === 6
let s2 = suivant(6);    // s2 === 7
```

- S'il n'y qu'une seule instruction on peut omettre les accolades et le `return`

```
const suivant = (nombre, distance = 1) => nombre + distance;

let s1 = suivant(1, 5); // s1 === 6
let s2 = suivant(6);    // s2 === 7
```

- Et s'il n'y a qu'un paramètre, on peut omettre les parenthèses

```
const cube = x => x ** 3;

let c2 = cube(2);   // c2 === 8
let c3 = cube(3);   // c3 === 27
```

Exercice

Appel de fonction

On considère la fonction suivante :

```
function f(a, b) {
    a = a || 3;
    b = b || "abc";
    console.log(`a = ${a} et b = ${b}`);
    return a + b;
}
```

Indiquer si les appels suivants sont corrects et expliquer les résultats obtenus.

```
let a, b, r;
r = f(1, 2);
r = f(b = 2, a = 1);
r = f(5);
r = f();
r = f('', 'xyz');
r = f(1, 2, 3);
r = f(0, true);
r = f(6, []);
```

Truthy et falsy

Principe

- Tout valeur en JS peut être considérée comme un booléen.
- « valeurs vraies » (**truthy**) si elles sont évaluées comme vraies dans un contexte booléen
- « valeurs fausses » (**falsy**) si elles sont évaluées comme fausses dans un contexte booléen

Valeurs fausses (falsy)

```
false
0
-0
0n
""
null
undefined
NaN
```

```
function truthyOrFalsy(c) {
    return c ? "truthy" : "falsy";
}

console.log(truthyOrFalsy(""));  // falsy
console.log(truthyOrFalsy(NaN)); // falsy
console.log(truthyOrFalsy([]));  // truthy
console.log(truthyOrFalsy({}));  // truthy
```

Exemple d'utilisation

- On peut le combiner avec l'évaluation en court-circuit des opérateurs logiques

```
let MAX = valeur || 10;
```

Compléments sur le passage de paramètres

Pas de vérification entre paramètres effectifs et paramètres formels

- Tous les appels sont valides
- Pas possible de nommer les paramètres à l'appel :
  - mais l'affectation ayant une valeur on pourrait penser que c'est possible
  - en fait on affecte des variables locales, pas les paramètres formels !
- Si un paramètre formel n'est pas fourni, le paramètre formel est `undefined`
- S'il y a plus de paramètres, `arguments` permet de les exploiter

Exercice

Robots

1. Écrire une fonction `robot` qui retourne un objet robot à partir de son abscisse (x, un nombre), de son ordonnée (y, un nombre) et sa direction (direction, une chaîne)
2. Écrire une fonction `robotToString` qui produit une chaîne de caractère qui correspond à la représentation d'un robot (exemples : `"(2,4)>est>"` ou `"(7,-3)>sud>"`).
3. Créer et afficher un robot.
4. Écrire une fonction `flotte` qui retourne un tableau de N robots à partir d'un premier robot. Deux paramètres deltaX et deltaY permettront de créer les autres robots en ajoutant deltaX à l'abscisse et deltaY à l'ordonnée du robot précédent, la direction est la même. Si deltaX ou deltaY ne sont pas donnés, ils auront respectivement les valeur 4 et 3.
5. Créer une flotte de robots avec la fonction précédente et les afficher.

Sommaire

Principe de la programmation fonctionnelle

Pas d'effet de bord

- Pas d'état (et donc pas de changement d'état)
- Tous les sous-programmes sont des fonctions **pures** qui
  - ne modifient pas leurs paramètres (ni aucune autre donnée) et
  - calculent de nouvelles valeurs (objets)
- Tout est expression (avec une valeur) !

Décomposition fonctionnelle

```
Programme(x1...xM) =
    F (e1, e2, ..., eN)
Avec
    e1 = F1(..., xI, ...)
    e2 = F2(..., xI, ...)
    ...
    eN = FN(..., xI, ...)
```

Intérêt du fonctionnel

- pas d'ordre sur d'évaluation des expressions en paramètre
- parallèlisation naturelle (exécution en // des expressions)
- lisibilité (et donc maintenance) améliorée (pas d'état)
- tests en boîte noire facilités (entrées et résultats explicites)

Structures de contrôle

1. Conditionnelle : Si condition Alors valeurVraie Sinon valeurFaux FinSi
2. Composition fonctionnelle (voir ci-dessus)
3. Récursivité (le pendant des boucles).

Exemple

Exemple

```
/** Distance entre deux points du plan. */
function distance(p1, p2) {

    function carre(x) {
        return x * x;
    }

    const dx2 = carre(p1.x - p2.x);
    const dy2 = carre(p1.y - p2.y);
    return Math.sqrt(dx2 + dy2);
}

function Point(x, y) {
    return {x: x, y: y};
}

assert.equal(distance(Point(3, 2), Point(7, 5)), 5);
```

Récursivité

Définition

Une fonction récursive est une fonction dont l'implantation contient un appel à elle-même.

Important : Il faut donc bien comprendre la spécification de cette fonction !

Exemple : factorielle récursive

```
function fact(n) {
    if (n <= 1) {    // cas terminal
        return 1;
    } else {         // cas général
        return n * fact(n - 1);
    }
}
```

Exécution

```
fact(4) --> 4 * fact(3) car non 4 <= 1
                \--> 3 * fact(2) car non 3 <= 1
                          \--> 2 * fact(1) car non 2 <= 1
                                    \--> 1 car 1 <= 1
                                    <-- 1
                          <-- 2 * 1 = 2
                <-- 3 * 2 = 6
```

Fondement mathématique

Récurrence

Le corps de la fonction `factorielle` précédente correspond à la définition mathématique de la factorielle donnée sous forme de **récurrence** :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

```
function fact(n) {    // récursive
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}
```

Autre formulation mathématique

On pourrait définir la factorielle par un produit conduisant à une version non récursive.

$$n! = \prod_{i=1}^{i=n} i = 1 \times 2 \times 3 \ldots \times (n-1) \times n$$

```
function fact(n) {    // itérative
    let resultat = 1;
    for (let k = 2; k <= n; k++) {
        resultat = resultat * k;
    }
    return resultat;
}
```

Les définitions par récurrence (et donc les SP récursifs) sont souvent plus concises et claires que leurs équivalents itératifs.

Terminaison

Motivation

Il faut être sûr que les appels récursifs s'arrêtent.

Terminaison

- Prévoir un (ou plusieurs) cas de base (terminal) sans appel récursif.
- Dans le cas général, mettre en évidence un entier positif (taille du problème) qui décroit strictement à chaque appel récursif (c'est le **Variant**).

Application à la factorielle :

- On définit la taille du problème de fact(n) comme étant n
- le cas terminal correspond à $n \leq 1$
- dans le cas général ($n > 1$), l'appel récursif est `fact(n - 1)` de taille strictement inférieure ($n - 1 < n$).

Récursivité terminale

Définition : Une fonction est **récursive terminale** quand le résultat de l'appel initial est directement celui du dernier appel récursif.
Règle : Aucune opération n'est réalisée sur le retour d'un appel récursif.

Factorielle en récursivité terminale

```
function fact_t(k, r=1) {
    if (k <= 1) {
        return r;
    } else {
        return fact_t(k - 1, k * r);
    }
}

function fact(n) {
    return fact_t(n, 1);
}
```

Le calcul de factorielle de 4 :

```
fact(4, 1) --> fact(3, 4) --> fact(2, 12) --> fact(1, 24) --> 24
        <-24--          <-24--           <-24--           <-24--
```

Toute fonction récursive terminale peut être réécrite avec une répétition

```
// récursivité terminale
function fact_t(k, r=1) {
    if (k <= 1) {
        return r;
    } else {
        return fact_t(k - 1, k * r);
    }
}

function fact(n) {
    return fact_t(n, 1);
}

f = fact(4)
```

```
// version itérative
function fact(n) {
    let resultat = 1;
    let k = n;
    while (k > 1) {
        resultat = resultat * k;
        k = k - 1;
    }
    return resultat;
}

f = fact(4)
```

Correspondances

- resultat est équivalent à r
- resultat est initialisé à 1, valeur de r lors du premier appel à fact_t
- la condition du `while` correspond à la condition du cas général cas général (k > 1)
- les instructions du `while` correspondent à l'appel récursif
  - le nouveau resultat est k * resultat
  - le nouveau k est k - 1
- le `while` devrait ici être replacé par un `for`

Exercices

Écrire une fonction récursive pour les questions suivantes.

1. Calculer x (nombre) à la puissance n (exposant) sans utiliser l'opérateur **.
2. Calculer la somme des n premiers entiers.

Induction

Principe

L'induction est une généralisation de la récurrence qui consiste à s'appuyer sur la structure de l'information pour déterminer les cas à traiter.

Exemples

- Sur les entiers : on raisonne sur l'entier et son suivant.
  - cas de base $n = 0$
  - cas général : suivant de n (donc $n + 1$).
  - on retrouve le principe de la récurrence
- Sur un tableau
  - cas d'un tableau vide
  - cas général : un premier élément et les éléments suivants
- Sur un arbre binaire (deux fils au plus) :
  - cas d'un arbre vide
  - cas général : un noeud avec sa valeur, ses sous-arbres gauche et droit
- Sur les chiffres d'un nombre
  - on s'intéresse aux chiffres d'un nombre
  - quelle induction ?

Opération de Array qui permettent la programmation fonctionnelle

Méthodes pures de Array

- concat() : concaténer deux tableaux `t1.concat(t2)`

```
const t1 = ['a', 'b', 'c'], t2 = ['e', 'f'];
const t3 = t1.concat(t2);  // t3: ['a', 'b', 'c', 'e', 'f'], t1 et t2 inchangés
const t4 = t1.concat(t2, 4, t2);      // plusieurs paramètres possibles
// t4: [ 'a', 'b', 'c', 'e', 'f', 4, 'e', 'f' ]
```

- slice(debut, fin) : une copie superficielle du tableau entre debut (inclus) et fin (exclu)

```
t3.slice(1, 4);  // [ 'b', 'c', 'e' ]
t3.slice();      // [ 'a', 'b', 'c', 'e', 'f' ] copie superficielle
t3.slice(3);     // [ 'e', 'f' ]
```

- `includes(x, debut)` : est-ce que la tableau contient `x` à partir de l'indice `debut` ?
- `Array.isArray(objet)` : vrai ssi objet est un tableau
- `join(sep)` : produire une chaîne en joignant les éléments du tableau par sep
- `indexOf(x, début)` : l'indice de la première occurrence de `x` à partir de l'indice `début`

```
const i5a = t4.indexOf('e');      // 3
const i5b = t4.indexOf('e', 4);   // 6
const i5c = t4.indexOf('e', 7);   // -1 car non trouvé
```

- `lastIndexOf(x, début)` : comme `indexOf` mais recherche de droite à gauche (indice décroit)

Exercices

Tous les exercices suivants sont à écrire en récursif.

1. Calculer la fréquence de x dans un tableau
   ```
   frequence_tab([1, 2, 3, 1, 2, 1], 2) === 2
   frequence_tab([1, 2, 3, 1, 2, 1], 1) === 3
   frequence_tab([1, 2, 3, 1, 2, 1], 3) === 1
   frequence_tab([1, 2, 3, 1, 2, 1], 0) === 0
   frequence_tab([], 0) === 0
   ```
2. Calculer la fréquence d'un chiffre dans un nombre
   ```
   frequence_nb(121231, 2) === 2
   frequence_nb(121231, 1) === 3
   frequence_nb(1000, 0) === 3
   frequence_nb(0, 0) === 1
   ```
3. Calculer le nombre d'éléments non nuls d'un tableau
4. Déterminer l'égalité logique de deux tableaux de nombres
5. Écrire une fonction de tri par fusion qui retourne un tableau trié. Le principe est le suivant. Si le tableau contient au plus un élément, il est déjà trié. Sinon, on découpe le tableau en 2, la première moitié et la deuxième moitié. On trie les deux puis on fusionne les deux tableaux triés. C'est un algorithme du type « diviser pour régner » dont la complexité en $O(n.log_2(n))$.
   1. Dérouler l'algorithme sur le tableau [7, 1, 3, 7, 4].
   2. Le programmer

Exercices sur tableaux imbriqués (arbres)

1. Applatir un tableau
   `aplati([1, [2, 3]])` donne `[1, 2, 3]`
   `aplati([[0], 1, [2, 3], [4, [5]], []])` donne `[0, 1, 2, 3, 4, 5 ]`
2. Afficher un tableau
   `toStr([1, [2, 3], [4, [5], []])` donne `'[1, [2, 3], [4, [5], []]'`
3. Déterminer l'égalité logique de tableaux de nombres ou de tableaux

Exercices (vers map)

1. Écrire une fonction qui étant donné un tableau de nombres retourne un nouveau tableau avec les mêmes nombres mis au cube.
```
> auCube([1, 2, 3, 4]);
[1, 8, 27, 64]
```
2. Écrire une fonction qui étant donné un tableau de nombres retourne un nouveau tableau qui contient `true` si le nombre à l'indice correspondant est pair, `false` sinon.
```
> versPairImpair([1, 2, 3, 4]);
[false, true, false, true]
```
3. Écrire une fonction qui étant donné un tableau de nombres retourne un nouveau tableau avec la factorielle de ces mêmes nombres.
```
> versFactorielle([1, 2, 3, 4]);
[1, 2, 6, 24]
```
4. Écrire une fonction qui étant donné un tableau de robots retourne un nouveau tableau qui contient l'abscisse de ces robots
```
> const robots = [ robot(1, 2, "est"), robot(6, 2, "est"),
                   robot(2, 3, "sud"), robot(7, 9, "est") ];
> versAbscisses(robots);
[1, 6, 2, 7]
```
5. On constate que les algorithmes ci-dessus sont très proches. Les généraliser sous la forme d'une fonction `map`. Cette fonction `map` pourra alors être utilisée pour écrire les fonctions précédentes.

Exercices (vers filter)

1. Écrire une fonction qui, étant donné un tableau de nombres, retourne un nouveau tableau qui ne contient que les nombres positifs, dans le même ordre.
```
> positifs([1, -4, 6, -2, 3]);
[1, 6, 3]
```
2. Écrire une fonction qui, étant donné un tableau de nombres, retourne un nouveau tableau qui ne contient que les nombres pairs, dans le même ordre.
```
> pairs([1, -4, 6, -2, 3]);
[-4, 6, -2]
```
3. Écrire une fonction qui, étant donné un tableau de robots, retourne un nouveau tableau qui ne contient que les robots direction 'est' avec l'abscisse inférieure à l'ordonnée.
```
> const robots = [ robot(1, 2, "est"), robot(6, 2, "est"),
                   robot(2, 3, "sud"), robot(7, 9, "est") ];
> quelquesRobots(robots)
[ robot(1, 2, "est"), robot(7, 9, "est") ]
```
4. On constate que les algorithmes ci-dessus sont très proches. Les généraliser sous la forme d'une fonction `filter`. Cette fonction `filter` pourra alors être utilisée pour écrire les fonctions précédentes.

Exercices (vers reduce)

1. Écrire une fonction qui retourne la somme des nombres d'un tableau de nombres
```
> somme([2, -4, 6, -2, 3]);
5
```
2. Écrire une fonction qui retourne le produit des nombres d'un tableau de nombres
```
> produit([2, -4, 6, -2, 3]);
288
```
3. Écrire une fonction qui retourne le plus grand des nombres d'un tableau de nombres. Elle renvoie `undefined` si le tableau est vide.
```
> max([2, -4, 6, -2, 3]);
6
```
4. Écrire une fonction qui retourne le plus petit des nombres d'un tableau de nombres. Elle renvoie `undefined` si le tableau est vide.
```
> min([2, -4, 6, -2, 3]);
-4
```
5. On constate que les deux premières fonctions ont la même structure. Les généraliser sous la forme d'une fonction nommée `reduce`. Comment faire pour que cette fonction intègre aussi les deux derniers cas ?

map/filter/reduce

map

Obtenir un **nouveau** tableau de même taille que le tableau **tab** peuplé des éléments obtenus en appliquant une fonction **callback** aux éléments du tableau tab (dans l'ordre des indices croissants).
La fonction callback prend comme paramètre :

- la **valeur** à traiter (une tableau tab)
- l'**indice** de cette valeur dans le tableau
- le **tableau** lui-même (sur lequel map a été appelé valeur === tableau[indice])

```
let nouveau = tab.map(callback)
```

Exemple

- `[1, 2, 3].map(x => x ** 3);`

Remarque : Le terme *callback* (fonction de rappel) est ici impropre (mais utilisé en JS).

filter

Obtenir un **nouveau** tableau qui ne contient que les élémnents du tableau **tab** sur lesquels la fonction **callback** s'évalue à vrai.
La fonction callback prend comme paramètre :

- la **valeur** à traiter (une tableau tableau tab)
- l'**indice** de cette valeur dans le tableau
- le **tableau** lui-même (sur lequel map a été appelé valeur === tableau[indice])

Exemple

```
[1, -3, 2, 3, -1, -6].filter(x => x > 0);              // [ 1, 2, 3 ]
[1, -3, 2, 3, -1, -6].filter((x, i) => i % 2 == 0);  // [ 1, 2, -1 ]
    // Ici, on utilise l'indice pour ne garder que les éléments d'indice pair
[1, -3, 2, 3, -1, -6].filter((x, i, t) => x > t[i-1]); // [ 2, 3 ]
    // i et t pour ne garder que les éléments plus grands que le précédent
```

Exercice : Pourquoi le dernier marche-t-il ?

Exercice

- Comment obtenir le tableau des éléments d'indices pairs d'un tableau `t` ?
- Comment obtenir le tableau des éléments d'indices impairs d'un tableau `t` ?

reduce

Appliquer une fonction **callback** sur tous les éléments d'un tableau avec une éventuelle valeur initiale. Cette fonction prend deux paramètres :

1. un accumulateur (variable qui joue le rôle d'accumulateur) initialisé avec la valeur initiale, à défaut le premier élément du tableau
2. la **valeur**, un élément du tableau (successivement tous les éléments du tableau, indice croissants)
3. l'**indice** de cette valeur dans le tableau
4. le **tableau** lui-même (sur lequel map a été appelé valeur === tableau[indice])

```
[5, 1, 5, 3, 5].reduce( (somme, x) => somme + x);     // 19
[5, 1, 5, 3, 5].reduce( (nb5, x) => nb5 + (x === 5 ? 1 : 0), 0 );  // 3
```

reduceRight

Identique à `reduce` mais les éléments du tableaux sont parcourus en partant de la fin.

Exemples

```
[1, 2, 3].reduce( (tab, x) => {
    tab.push(x);
    return tab;
    }, []);          // [1, 2, 3]
```

```
[1, 2, 3].reduceRight( (inverse, x) => {
    inverse.push(x);
    return inverse;
    }, []);                  // [3, 2, 1]
```

Autres exemples

```
[5, 1, 5, 3, 5].reduce( (croissant, x, i, t) => croissant && t[i-1] <= x);
        // résultat : false.  Fonctionne ?

[1, 2, 3, 5, 7].reduce( (croissant, x, i, t) => croissant && t[i-1] <= x);
        // résultat : true.  Fonctionne ?

[0, 1, 2, 3, 5].reduce( (croissant, x, i, t) => croissant && t[i-1] <= x);
        // Résultat ? Fonctionne ?

[0, 2, 5, 3].reduce( (croissant, x, i, t) => {
        if (i > 0) {
            return croissant && t[i-1] <= x;
        } else {
            return true;
        }
    }, true)

[0, 2, 5, 3].reduce( // avec le si arithmétique
        (croissant, x, i, t) => (i == 0) ? true : croissant && t[i-1] <= x
        , true)
```

Question : Quel est le résultat de `0 && true`, `5 && false`, `1 && true` ?

Exercices

1. On peut reprendre les exercices précédents en utilisant map/reduce/filter/...
2. Si on considère une liste d'objets livres, on veut obtenir les informations suivantes :
   1. Le nombre de livres de plus de 800 pages
   2. Les titres des livres de plus de 800 pages (affichés un par ligne)
   3. Le nombre de livres de la catégorie Internet
   4. Le total des pages des livres de la catégorie Internet
   5. Toutes les catégories utilisées
   6. Tous les titres de livres qui ont JavaScript dans le titre

   On pourra utiliser ce fichier JSON ou celui-ci
   Pour charger un fichier JSON :

```
const fs = require('fs');
// syntaxe ES6 : import fs from 'fs';
const filename = 'mon-fichier.json';
const books = JSON.parse(fs.readFileSync(filename, 'utf8'));
console.log(books);
```

Autres méthodes : some, every, find, findIndex, forEach

- Sur le même modèle que map et filter
- some : vrai si le *callback* est vrai sur un élément au moins du tableau
- every : vrai si le *callback* est vrai pour tous les éléments du tableau
- find : le premier élément sur lequel le *callback* retourne vrai
- findIndex : idem find mais retourne l'index, pas l'élément
- forEach : appliquer une action (le callback) sur chaque élément

```
[1, 2, 3, 4].some( x => x % 2 === 0);   // true
[1, 2, 3, 4].some( x => x < 0);         // false
[1, 2, 3, 4].every( x => x % 2 === 0);  // false
[1, 2, 3, 4].every( x => x > 0);        // true
[1, 2, 3, 4].every( (x, i, t) => (i > 0) ? x >= t[i-1] : true);        // true
[1, 2, 3, 4].find( x => x > 2);         // 3
[1, 2, 3, 4].findIndex( x => x > 2);    // 2 (indice de 3)
[1, 2, 3, 4].forEach(x => console.log(x));
[1, 2, 3, 4].forEach((x, i, t) => console.log(x, i, t));

// l'évaluation se fait en court-circuit. La preuve :
const t1 = [1, 2, 3];
t1.some( (x, i, t) => t[i] = 7);
console.log(t1);                        // [ 7, 2, 3 ]
    // Attention : on ne devrait pas se servir de some pour modifier un tableau
```

Compléments : for et objet

```
const o1 = {x: 1, nom: 'abc'};

console.log("\nAvec for in :");
for (const p in o1) {
    console.log('-', p, ':', o1[p]);
}
console.log("\nAvec for of :");
try {
    for (const v of o1);  // Interdit !
} catch (TypeError) {
    console.log("Erreur : objet pas itérable");
}
console.log("\nAvec Object.keys() :")
for (const p of Object.keys(o1)) {
    console.log('-', p, ':', o1[p]);
}
console.log("\nAvec Object.entries() :")
for (const [k, v] of Object.entries(o1)) {
    console.log('-', k, '->', v);
}
console.log("\nAvec Object.values() :")
for (const p of Object.values(o1)) {
    console.log('-', p);
}
```

```
> node for-objet.js

Avec for in :
- x : 1
- nom : abc

Avec for of :
Erreur : objet pas itérable

Avec Object.keys() :
- x : 1
- nom : abc

Avec Object.entries() :
- x -> 1
- nom -> abc

Avec Object.values() :
- 1
- abc
```

Compléments : forEach à la place de for

forEach et objet

```
const o1 = {x: 1, nom: 'abc'};

console.log("\nObject.keys(...).forEach :")
Object.keys(o1).forEach( cle =>
    console.log('-', cle, '->', o1[cle])
)
console.log("\nObject.entries(...).forEach :")
Object.entries(o1).forEach( ([cle, valeur]) =>
    console.log('-', cle, '->', valeur)
)
```

```
> node foreach-objet.js

Object.keys(...).forEach :
- x -> 1
- nom -> abc

Object.entries(...).forEach :
- x -> 1
- nom -> abc
```

forEach et tableau (équivalent de enumerate de Python)

```
const tab = [6, 5, 9];
Object.entries(tab).forEach( ([i, x]) =>
    console.log(i, '->', x)
)
```

```
> node foreach-tab.js
0 -> 6
1 -> 5
2 -> 9
```

Sommaire

- Manipuler le DOM
- DOM et événements

DOM (Document Object Model)

- normalisé par le W3C
- représentation objet d'un page HTML (ou d'un document XML)
  - un arbre d'objets correspondant aux éléments HTML/XML
  - avec les attributs
- pour la manipuler via un programme, en particulier JavaScript

Vocabulaire HTML

```
<tag attr="val" id="42" class="name1 name2">
    innerHTML
</tag>
```

- `id` : (identifiant) sa valeur est unique pour tout le document
- `class` : classes d'appartenenance d'un élément (classes séparées par des blancs)
- `innerHTML` : le contenu de l'élément (un bout de DOM)

Objets du DOM

- Document : représente une page Web chargée dans un navigateur
- Node : un noeud dans l'arbre, généralise Element, Text, Comment...
- Element : un élément HTML/XML (de la balise ouvrante à la balise fermante)
- Text : Le contenu textuel d'un élément.
- NodeList : une liste de noeuds (ce n'est pas un Array JavaScript)
- Attr : attribut d'un élément.

DOM : Créer des objets et les assembler

Les méthodes sont dans l'objet Document (ce sont des fabriques).

Créer un élément

```
const e1 = document.createElement('h1');     // crée un élément <h1>...</h1>
```

Créer un attribut

voir aussi Element.attributes

```
const a1 = document.createAttribute("msg"); // un attribut (Attr) nommé msg
a1.value = "salut";                         // définir sa valeur (string)
e1.setAttributeNode(a1);                    // l'attacher à l'élément h1
e2.setAttribute("age", 18);                 // fait les 3 étapes précédentes
```

Créer un texte

```
const t1 = document.createTextNode("Bonjour !");  // Créer un texte (Text)
e1.appendChild(t1);
```

Rattacher les objets créés au DOM

- replaceWith : remplacer un élément existante
- appendChild : insérer un élément comme fils d'un autre
- insertAdjacentElement : insérer un nouveau nœud par rapport à l'élément cible.

Exploiter un élément

L'interface générale est donnée par Element qui hérite de Node :

```
e1.innerText        // la string des textes contenus dans e1
e1.innerHTML        // contenu de r1 (une DOMString avec balises HTML)
e1.attributes       // les attributs de e1
e1.className        // l'attribut `class` de l'élément e1
e1.id               // l'attribut `id` de e1
e1.children         // les fils de l'éléments e1
e1.tagName          // le nom de l'élément (balise), par exemple "h1"

// functions
el.appendChild(node) // rajoute un enfant (node) à el, en dernière position
el.insertAdjacentElement(position, el2) // insérer el2 relativement à el
```

Exploiter plusieurs éléments

- NodeList et HTMLCollection regroupent plusieurs éléments
- Ce ne sont pas des tableaux JS
- ES6 fournit `Array.from` qui construit un tableau à partir d'un objet

```
const allDiv = document.getElementsByTagName('div');
Array.from(allDiv).forEach(unDiv => console.log(unDiv));
```

Recherche d'éléments dans le DOM

Obtenir un élément particulier : on connait son identifiant (id)

```
elt = document.getElementById(id);
```

Obtenir plusieurs éléments avec getElementsByTagName et getElementsByClassName

- Avec des méthodes disponibles sur `Document` et `Element`, retourne HTMLCollection

```
getElementsByTagName(n);     // les éléments nommés `n`
getElementsByClassName(n);   // les éléments de la classe `n`
getElementsByName(n);        // les éléments dont l'attribut `name` vaut `n`
```

Avec sélecteur CSS : querySelector et querySelectorAll

```
let first = document.querySelector('.special'); // premier élément de la classe « sp
let all = document.querySelectorAll('.special') // NodeList des éléments de la class
```

Spécificités

Spécificités Js

- Chaque langage / bibliothèque peut définir ses propres extensions
- Accéder à l'élément `body` : `document.body`
- Ajouter des enfants à un nœud : `append`
  - accepte plusieurs paramètres
  - des `Node`, mais aussi des `str` (crée un élément `Text`)
  - ne retourne rien (`appendChild` retourne l'élément ajouté)

```
document.body.append("Un texte", document.createElement("br"), "La suite...");
```

- Accéder à l'élément body : `document.body`
  - inutile donc d'utiliser `getElementsByTagName` ou `querySelector` pour body.
- Attributs DOM prédéfinis :
  - `elt.id` : accéder à / modifier l'attribut `id` de l'élément `elt`
  - `elt.className` : idem pour l'attribut `class` de `elt`.
  - `elt.classList` : pour ajouter (`add`), supprimer (`remove`) ou inverser (`toggle`) une valeur.

```
document.body.className = 'n1';     // <body class='n1'>
document.body.classList.add('n2'); // <body class='n1 n2'>
```

- Accès aux éléments d'un formulaire par leur nom : `myform.aname`

Exercice

books

Conseil : Exécuter régulièrement (recharger la page) et construire les programmes petit à petit.

1. Écrire un script qui, à partir d'un tableau de livres, crée un DOM pour les afficher sous la forme d'une liste (`ol`) faisant apparaître son titre, les autres caractéristiques sont présentées sous la forme d'une liste (`dl`).
   a. Le faire d'abord en utilisant une boucle `for`
   b. Le faire en utilisant `map` et autres.
2. Modifier le script pour que :
   - l'élément `li` qui correspond à un livre ait pour identifiant son numéro ISBN et la classe `title`
   - les `dt` correspondant à une caractéristique auront pour classe le nom de cette caractéritique (isbn, pageCount, status, authors, categories...)
3. (optionnel) Prévoir une feuille de style pour mettre un peu de couleur...

books (suite)

1. Récupérer, lire et comprendre le code founi :
   - books.html
   - books.js
   - books.css
2. Écrire le code de la fonction `domBooksToJSON` qui, à partir des **informations du document HTML**, produit le tableau JS des livres avec les seuls attributs title, pageCount, catagories et authors.
3. Ajouter une table **tout en haut du document** qui présente les informations obtenues par la fonction `domBooksToJSON`.
4. Comment faire pour remplacer tout le corps du document par la table précédente ?

DOM et événements : Principe

La partie interactive d'une page HTML est gérée :

- en définissant des fonctions (callback)
- associées à des événements
- souvent sur des éléments du DOM

Interface EventTarget et addEventListener

```
let e1 = document.getElementById('abc');
e1.addEventListener(typeEv, callback);
```

- typeEv : chaîne de caractères identifiant l'événement
- callback : fonction appelée pour gérer l'événement
- le paramètre de callback est un objet représentant l'événement

Événements (liste complète)

- mouse : click, dblclick, mouseup, mousedown
- keyboard : keypress, keydown, keyup
- focus : focus, focusin, blur, focusout
- forms : submit (quand on clique sur un <input type="submit"/> dans un <form>)
- forms element : input (quand la valeur d'un input, textarea... change)
- window.DOMContentLoaded : DOM utilisable
- window.load : page chargée

Gérer un événement d'un élément

Principe : positionner la fonction qui fera le traitement (gestionnaire/handler).

inline (ancienne façon)

```
<a id="theLink" href="anywhere.html" onclick="doSomething()">
```

modèle traditionnel

```
<script>
let element = document.getElementById('theLink');
element.onclick = doSomething;
</script>
```

Attention : On ne peut donc associer qu'un seul gestionnaire.

JS moderne

```
<script>
element.addEventListener('click',doSomething)
element.addEventListener('click',doAnother)
</script>
```

- permet d'associer plusieurs gestionnaires à un même événement
- et de le supprimer : removeEventListener

Déclencher un événement avec du code

```
const ev = new Event('xxx');
element.dispatchEvent(ev);
```

Exemple : simuler un clic souris

```
const b = document.createElement('button');
b.innerHTML = 'OK';
b.addEventListener('click', e => console.log('It works!'));
document.body.appendChild(b);

// simulate a click
const ev = new MouseEvent('click');
b.dispatchEvent(ev);
```

Exercice (comprendre)

Pour cet exercice, on part du fichier multiplication.html.

1. Faire que les champs du formulaire correspondant aux opérandes soient initialisés à 5 (à gauche) et 6 (à droite). Côté HTML ou JS ?
2. Quand l'utilisateur clique sur le bouton *calculer* l'opération est réalisée et son résultat s'affiche.
3. Dès qu'on change la valeur de l'opérande gauche, le calcul se fait.
4. Dès qu'on tape un caractère sur l'opérande de droite, le calcul se fait.
5. Est-ce que le calcul se refait quand on utilise **delete** ? Peut-être faut-il prendre un autre événement. . .
6. Voyons ce qui se passe si on change la valeur de l'opérande gauche par script.
   a. Forcer la valeur de l'opérande à gauche à 42 quand on clique sur `changer gauche`.
   b. Est-ce le calcul se fait ?
   c. Déclencher explicitement l'événement de changement sur l'opérande gauche.
7. Que se passe-t-il si on fait une boucle sans fin quand on clique sur le bouton calculer ?

Exercice (books)

1. Actuellement la page est complétée par un appel direct à une fonction dans le script. Ceci devrait être fait une fois le document chargé. Modifier en conséquence le script.
2. Ajouter juste avant le titre `h1` la possibilité de masquer ou montrer les descriptions courtes (shortDescription). On pourra définir un élément div qui contient deux éléments span, l'un pour cacher, l'autre pour montrer. On utilisera la classe `cache` pour laquelle le fichier `books.css` définira `display: none;`.
   a. Où définir ces éléments div et span ?
   b. Comment leur attacher une réaction ?
   c. Implanter le comportement demandé.
3. Faire en sorte que quand on clique sur un titre de livre, on cache ou on montre, alternativement, toutes les descriptions de ce livre.
4. En JS, modifier le style des titres des livres pour que le curseur change quand il est dessus. On utilisera la valeur `pointer` pour le curseur.

Sommaire

Définitions

Séquentiel ou concurrent

Séquentiel : Les choses se font les une après les autres.
- ce qu'on a fait jusqu'à présent

Concurrent : Plusieurs choses se font simultanément
- le système Unix avec plusieurs programmes qui s'exécutent simultanément
- les membres d'une équipe projet, d'un orchestre, d'une équipe de foot.

Différent type de concurrence

Vrai parallélisme : Plusieurs processeurs
- les membres d'une équipe projet, d'un orchestre, d'une équipe de foot.
- plusieurs cœurs sur un processeur
- plusieurs ordinateurs (applications réparties)

Parallélisme par entrelacement : le processeur est partagé entre plusieurs activités
- **préemptif :** un ordonnanceur interrompt l'activité en cours et alloue le processeur à une autre
- **coopératif :** une activité s'interrompt pour laisser la possibilité aux autres de s'exécuter

Communication entre activités :
- par mémoire partagée (file d'exécution/*thread*) : attention aux conflits lecture/écriture écriture/écriture
- par envoi de message (processus) : protocole de communication, synchrone/asynchrone

Appel synchrone : l'appelant est bloqué et attend que la fonction appelée termine son exécution
Appel asynchrone : l'appelant n'est pas bloqué et continue son exécution. Il a juste déclenché l'exécution de l'autre sous-programme
Fonction asynchrone : (par extension) une fonction dont tous les appels ne font de manière asynchrone

Exemples

Synchrone : récupérer un paquet à la poste.
- le client donne l'avis de passage
- il attend jusqu'à ce que l'employé revienne avec le paquet

Asynchrone : un enseignant donne un exercice à faire à des étudiants
- les étudiants font l'exercice
- l' enseignant peut faire autre chose pendant ce temps
- éventuellement répondre à des questions d'étudiants
- faire le tour des étudiants pour voir comment ils avancent

Exercice

Les activités suivantes sont-elles synchrones ou asynchrones ?
- Dans un restaurant, un serveur prend une commande à une table
- Le serveur donne la commande à la cuisine

Donner d'autres exemples. . .

Et JavaScript ?

- Un **seul fil d'exécution** pour exécuter le programme JavaScript
- Utilisation de **fonctions asynchrones** pour ne pas bloquer l'exécution pour les activités longues
  - récupérer une ressource via une requête HTTP
  - dialoguer avec le serveur...
  - une telle fonction asynchrone s'appelle **fibre** ou **coroutine**
- Utilisation de **Worker** pour les activités qui demandent du temps de calcul
  - activités qui s'exécutent en tâche de fond
  - communication par message
- Voir Gestion de la concurrence et boucle des événements

Exemples

- Que constate-t-on quand la fonction associée au clic sur un bouton boucle (ou met beaucoup de temps à s'exécuter) ?
- Exemples (développés sur les transparents suivants) :
  1. Écrire une fonction qui affiche les carrés de plusieurs nombres (plusieurs manières)
     - Que se passe-t-il si on met un var au lieu de const devant v ?
  1. Écrire un chrono qui a intervalle régulier afficher le temps jusqu'à une certaine limite.

Afficher les carrés : version synchrone

exemple-carre-afficher-synchrone.mjs

```
function carre(x) { // Afficher le carré de x
    console.log();
    console.log(`carre ${x}: J'ai recu x =`, x);
    const x2 = x * x;
    console.log(`carre ${x}: Je l'affiche :`, x2);
    console.log(`=== Le carré de ${x} est ${x2}`);
}

function afficher_les_carres(...valeurs) {
    for (const v of valeurs) {
        console.log();
        console.log(`main: appel de carre(${v})`);
        carre(v);
        console.log(`main: après appel à carre(${v})`);
    }
}

afficher_les_carres(2, 3);
```

```
> node exemple-carre-afficher-syn

main: appel de carre(2)

carre 2: J'ai recu x = 2
carre 2: Je l'affiche : 4
=== Le carré de 2 est 4
main: après appel à carre(2)

main: appel de carre(3)

carre 3: J'ai recu x = 3
carre 3: Je l'affiche : 9
=== Le carré de 3 est 9
main: après appel à carre(3)
```

Comment exécuter une méthode en asynchrone ?

setTimeout : demander à exécuter une fonction après un délai en millisecondes.

`setTimeout(f, delai, p1, p2...)`

- f : la fonction à exécuter
- delai : dans combien de millisecondes (au plus tôt) l'exécuter
- p1, p2... : les paramètres qui seront transmis à f

Attention : une fonction asynchrone est exécutée quand le délai est atteint et que le thread principal n'a plus rien à faire

Exemple

exemple-asynchrone.mjs

```
function f1()  { console.log("Je suis f1."); }
function f2(x) { console.log(`Je suis f2, x = ${x}.`); }
function main() {
    console.log("main: début");
    setTimeout(f1, 1000);
    setTimeout(f2, 500, 5);
    setTimeout(f2, 0, 7);
    console.log("main: fin");
}
main(); // while (true);
```

```
> node exemple-asynchrone.mjs
main: début
main: fin
Je suis f2, x = 7.
Je suis f2, x = 5.
Je suis f1.
```

1. Est-ce que l'exécution semble cohérente ?
2. Que se passe-t-il si on décommente la dernière ligne.

Afficher les carrés : version asynchrone

exemple-carre-afficher-asynchrone.mjs

```
function carre(x) { // Afficher le carré de x
    console.log();
    console.log(`carre ${x}: J'ai recu x =`, x);
    const x2 = x * x;
    console.log(`carre ${x}: Je l'affiche :`, x2);
    console.log(`=== Le carré de ${x} est ${x2}`);
}

function afficher_les_carres(...valeurs) {
    for (const v of valeurs) {
        console.log();
        console.log(`main: appel de carre(${v})`);
        setTimeout(carre, 0, v);
        console.log(`main: après appel à carre(${v})`);
    }
}

afficher_les_carres(2, 3);
```

```
> node exemple-carre-afficher-asy

main: appel de carre(2)
main: après appel à carre(2)

main: appel de carre(3)
main: après appel à carre(3)

carre 2: J'ai recu x = 2
carre 2: Je l'affiche : 4
=== Le carré de 2 est 4

carre 3: J'ai recu x = 3
carre 3: Je l'affiche : 9
=== Le carré de 3 est 9
```

1. Que se passe-t-il si on remplace `setTimeout(carre, 0, v)` par `setTimeout(() => carre(v), 0)` ?
2. Et si en plus on remplace `const v` par `var v` ?
3. Et si on fait `let x;` avant le `for` et `let x = v;` dans le for et `setTimeout(() => carre(x)...` ?
4. Et si la fonction carre retourne le carré au lieu de l'afficher ?

Afficher les carrés (fonction avec résutat) : version synchrone

exemple-carre-fonction-erreur-synchrone.mjs

```
function carre(x) {
    console.log(`carre ${x}: J'ai recu x =`, x);
    const x2 = x * x;
    console.log(`carre ${x}: Je retourne :`, x2);
    return x2;
}

function afficher_carres(...valeurs) {
    for (const v of valeurs) {
        console.log();
        console.log(`main: appel de carre(${v})`);
        const leCarre = carre(v);
        console.log(`main: après appel à carre(${v})`);
        console.log(`main: valeur reçue :`, leCarre);
    }
}

afficher_carres(2, 3);
```

```
> node exemple-carre-fonction-err

main: appel de carre(2)
carre 2: J'ai recu x = 2
carre 2: Je retourne : 4
main: après appel à carre(2)
main: valeur reçue : 4

main: appel de carre(3)
carre 3: J'ai recu x = 3
carre 3: Je retourne : 9
main: après appel à carre(3)
main: valeur reçue : 9
```

Afficher les carrés (fonction avec résutat) : version asynchrone fausse !

exemple-carre-fonction-erreur-asynchrone.mjs

```
function carre(x) {
    console.log(`carre ${x}: J'ai recu x =`, x);
    const x2 = x * x;
    console.log(`carre ${x}: Je retourne :`, x2);
    return x2;
}

function afficher_carres(...valeurs) {
    for (const v of valeurs) {
        console.log();
        console.log(`main: appel de carre(${v})`);
        const leCarre = setTimeout(carre, 0, v);
        console.log(`main: après appel à carre(${v})`);
        console.log(`main: valeur reçue :`, leCarre);
    }
}

afficher_carres(2, 3);
```

```
> node exemple-carre-fonction-err

main: appel de carre(2)
main: après appel à carre(2)
main: valeur reçue : Timeout {
  _idleTimeout: 1,
  _idlePrev: [TimersList],
  _idleNext: [TimersList],
  _idleStart: 19,
  _onTimeout: [Function: carre],
  _timerArgs: [Array],
  _repeat: null,
  _destroyed: false,
  [Symbol(refed)]: true,
  [Symbol(kHasPrimitive)]: false,
  [Symbol(asyncId)]: 8,
  [Symbol(triggerId)]: 0,
  [Symbol(kAsyncContextFrame)]: u
}

main: appel de carre(3)
main: après appel à carre(3)
main: valeur reçue : Timeout {
  _idleTimeout: 1,
  _idlePrev: [TimersList],
  _idleNext: [Timeout],
  _idleStart: 20,
  _onTimeout: [Function: carre]
```

- On a bien un résultat. . . mais pas celui de carre
- La fonction n'a pas encore été exécutée
- On a juste demandé à ce qu'elle le soit
- L'objet permet de contrôler l'exécution de la fonction
- En particulier l'annuler : clearTimeout
- Comment gérer le résultat de carre ?

Afficher les carrés (fonction avec résutat) : version asynchrone !

exemple-carre-fonction-asynchrone.mjs

```
function carre(x, suite) {  // Retourner le carre de x
    console.log(`carre ${x}: J'ai recu x =`, x);
    const x2 = x * x;
    console.log(`carre ${x}: Je donne à suite :`, x2);
    suite(x2);
}

function afficher_carres(...valeurs) {
    for (const v of valeurs) {
        console.log();
        console.log(`main: appel de carre(${v})`);
        function afficher(leCarre) {
            console.log(`main: valeur reçue :`, leCarre);
        }
        setTimeout(carre, 0, v, afficher);
        console.log(`main: après appel à carre(${v})`);
    }
    console.log(`main: FIN.`);
}

afficher_carres(2, 3);
```

```
> node exemple-carre-fonction-asy

main: appel de carre(2)
main: après appel à carre(2)

main: appel de carre(3)
main: après appel à carre(3)
main: FIN.
carre 2: J'ai recu x = 2
carre 2: Je donne à suite : 4
main: valeur reçue : 4
carre 3: J'ai recu x = 3
carre 3: Je donne à suite : 9
main: valeur reçue : 9
```

- On donne à la fonction appelé en asynchrone, la fonction qu'elle devra appeler après.
- Ainsi, quand son résultat est obtenu, elle appelle cette fonction.

Exemple (à ne pas suivre !) : Chronomètre (version synchrone)

chrono-synchrone.mjs

```
let _nbChrono = 0;
export default function chrono(duree, delai, runEachDelai) {
    _nbChrono++;
    runEachDelai = runEachDelai || 'Chrono ' + _nbChrono;
    if (typeof runEachDelai === 'string') {
        const name = runEachDelai;
        runEachDelai = (x, d) => console.log(name + ' ' + x);
    }
    const start = new Date().getTime();
    let count = 0;  // ellapsed delai count
    let ellapsed;
    do {
        const now = new Date().getTime();
        ellapsed = now - start;
        if (ellapsed / delai >= count) {
            runEachDelai(ellapsed, duree);
            count++;
        }
    }
    while (ellapsed < duree);
}

chrono(1000, 300)
chrono(1000, 200, (x, d) => console.log('Mon chrono : ' + (100 * x / d).toFixed(2)))
chrono(1000, 500, (x, d) => console.log('Secondes : ' + (x / 1000).toFixed(2)))
```

```
> node chrono-synchrone.mjs
Chrono 1 0
Chrono 1 300
Chrono 1 600
Chrono 1 900
Mon chrono : 0.00
Mon chrono : 20.00
Mon chrono : 40.00
Mon chrono : 60.00
Mon chrono : 80.00
Mon chrono : 100.00
Secondes : 0.00
Secondes : 0.50
Secondes : 1.00
```

- Attente active : 100% CPU
- Rien d'autres ne peut s'exécuter
- Chronos en séquence !
- Conclusion : ne pas faire ainsi !

setInterval : appeler une fonction à intervalle régulier

`setInterval(f, delai, p1, p2...)`

- f : la fonction à exécuter
- delai : tous les combien de millisecondes l'exécuter (au plus tôt)
- p1, p2... : les paramètres qui seront transmis à f

Exemple: Chronomètre (version asynchrone)

chrono-asynchrone.mjs

```
let _nbChrono = 0;
export default function chrono(duree, delai, runEachDelai) {
    _nbChrono++;
    runEachDelai = runEachDelai || 'Chrono ' + _nbChrono;
    if (typeof runEachDelai === 'string') {
        const name = runEachDelai;
        runEachDelai = (x, d) => console.log(name + ' ' + x);
    }
    function chaqueDelai() {
        const now = new Date().getTime();
        const ellapsed = now - start;
        runEachDelai(ellapsed, duree);
    }
    const start = new Date().getTime();
    const id = setInterval(chaqueDelai, delai)
    setTimeout(() => clearInterval(id), duree);
}

chrono(1000, 300)
chrono(1000, 200, (x, d) => console.log('Mon chrono : ' + (100 * x / d).toFixed(2)))
chrono(1000, 500, (x, d) => console.log('Secondes : ' + (x / 1000).toFixed(2)))
```

```
> node chrono-asynchrone.mjs
Mon chrono : 20.10
Chrono 1 300
Mon chrono : 40.10
Secondes : 0.50
Chrono 1 601
Mon chrono : 60.20
Mon chrono : 80.20
Chrono 1 901
```

- peu de CPU utilisé : top
- activités mises en attente
- réveillées quand utile
- tuées quand délai atteint

Compteur automatique

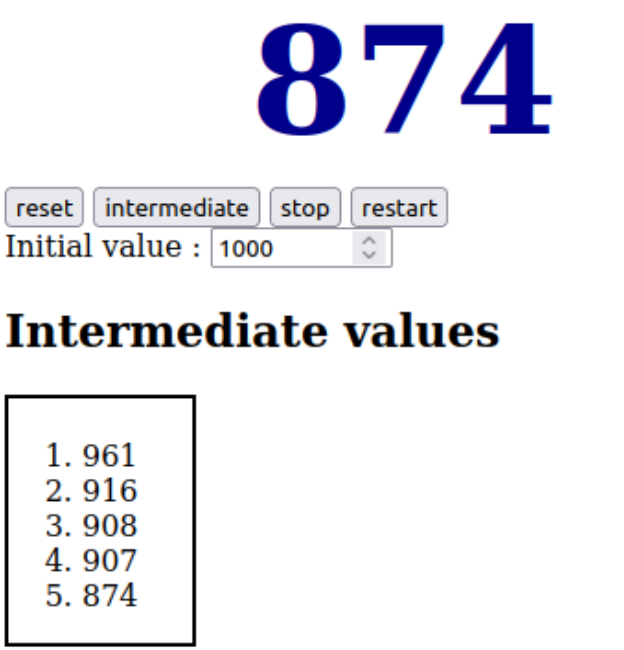
Compléter le fichier `dom-countdown.mjs` pour réaliser un décompte comme le montre la figure ci-après. Le décompte démarre dès le chargement de la page avec comme valeur initiale, la valeur indiquée dans le champ du formulaire.

Le décompte peut être interrompu en utilisant `stop` et (re)démarré en utilisant `restart`.

Un clic sur le bouton `intermediate` mémorise une valeur intermédiaire (si pas déjà enregistrée).

Le bouton `reset` réinitialise le compteur à la valeur initiale et vide les temps intermédiaires. Ce bouton n'a pas d'effet sur le décompte : il reste arrêté ou en fonctionnement suivant le cas.

Il est conseillé de prévoir une fonction par bouton. . .



Défaut de l'approche historique : exemple de realine

Objectif

Écrire une fonction qui demande un nom et une ville, puis dit où il habite.

Version synchrone avec le module readline-sync

use-readline-sync.mjs

```
import prompt_ from "prompt-sync";    // npm install prompt-sync
const prompt = prompt_();

function main() {
    const nom = prompt('Ton nom ? ');
    const ville = prompt('Ta ville ? ');
    console.log(`${nom} habite ${ville}`);
}

main();
```

```
Ton nom ? Xavier
Ta ville ? Toulouse
Xavier habite Toulouse
```

Problème

- Tout est bloqué jusqu'à ce que l'utilisateur valide sa saisie
- On peut le vérifier en démarrant un chrono (avant l'appel `main()`) :
  ```
  import chrono from "./chrono-asynchrone.mjs";
  chrono(10000, 500);
  ```
- Solution : utiliser une version asynchrone : readline

Exemple avec le module readline (asynchrone)

use-readline-exemple-async.mjs

```
import readline from "readline";

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question("Une information ? ", function(reponse) {
    console.log('Le réponse est :', reponse);
    rl.close();
});
```

Exercice

1. Vérifier que ce programme marche bien
2. Démarrer un chronometre. Que constate-t-on ?
3. Écrire l'équivalent du programme précédent avec readline.
4. Démarrer le chronomètre avant de démarrer la saisie. Que constate-t-on ?
5. Que faudrait-il faire si on voulait demander une 3ème information. Et une 4ème ?
6. Quel est le défaut de cette approche ?

Les promesses

Promesse : objet qui représente une valeur en devenir.

- La valeur est peut-être là (la promesse est tenue, *fulfilled*),
- elle le sera plus tard (la promesse est en attente, *pending*)...
- ou jamais (la promesse est rompue, *rejected*).

Syntaxe

```
new Promise( /* exécuteur */ function(resolve, reject) {
    ...
})
```

- L'exécuteur est la fonction qui va essayer de tenir la promesse.
- Cette fonction prend en paramètre deux fonctions :
  - resolve : la promesse est tenue et son paramètre est sa valeur : `resolve(valeur)`
  - reject : la promesse est rompue et son paramètre est la raison : `reject('la raison')`

Intérêt

- fournit des méthodes qui permettent de réagir à :
  - une promesse tenue : `then(function)` qui peuvent être chaînées
  - une promesse rompue : `catch(function)` après le dernier `then`

Application à readline

use-readline-mypromise.mjs

```
import readline  from "readline";

function input(question, limite) {
    return new Promise( (resolve, reject) => {
        const rl = readline.createInterface({
            input: process.stdin,
            output: process.stdout
        });
        const id = setTimeout(() => {
            rl.close();
            reject('Trop long à répondre');
        }, Number(limite) || 4000);
        rl.question(question, function(reponse) {
            clearTimeout(id);
            rl.close();
            resolve(reponse);
        });
    });
}

function main() {
    let nom;
    input("Ton nom ? ")
        .then( reponse => {
            nom = reponse;
            return input("Ta ville ? ");
        })
        .then( ville => {
            console.log(`${nom} habite ${ville}`
        })
        .catch( e => {
            console.log(e);
            console.log('Fin.');
        });
}

main();
```

1. Expliquer le code précédent
2. Comment faire s'il y a une 3ème information à demander ? Et une 4ème ?
3. Quels défauts ?

async et await

await

- `await` s'applique à une promesse
- il provoque l'attente de la résolution de la promesse (tenue ou rompue)
- si la promesse est tenue l'exécution continue sur la ligne suivante
- si la promette est rompue l'exception se propage

async

- une fonction qui contient un `await` doit être déclarée `async`

Exemple

sleep.mjs

```
async function sleep(duree) {
    console.log("sleep: début");
    await new Promise( resolve => setTimeout( () => resolve(), duree))
    console.log("sleep: fin");
}

async function main() {
    console.log("main: début");
    await sleep(1000);
    console.log("main: fin");
}

main();
```

Application à readline

use-readline-await.mjs

```
import readline  from "readline";

function input(question, limite) {
    return new Promise( (resolve, reject) => {
        const rl = readline.createInterface({
            input: process.stdin,
            output: process.stdout
        });

        const id = setTimeout(() => {
            rl.close();
            reject('Trop long à répondre');
        }, Number(limite) || 2000);
        rl.question(question, function(reponse) {
            clearTimeout(id);
            rl.close();
            resolve(reponse);
        }) ;
    })
}

async function main() {
    try {
        const nom = await input("Ton nom ? ");
        const ville = await input("Ta ville ? ");
        console.log(`${nom} habite ${ville}`);
    } catch (e) {
        console.log("Problème : ", e);
    }
}

main();
```

- On retrouve un style proche du style synchrone
- **Attention :** On séquentialise le code et on supprime donc du parallèllisme potentiel
- **Solution :** Promise.all ou Promise.allSettled qui créent à partir d'un itérable de promesses.

Utilisation de yesno.wtf

Le site YesNo peut aider à prendre des décisions de type oui/non en répondant l'un ou l'autre. Il propose une API que nous allons utiliser.

Le code fourni (`yesno.*`) montre comment l'utiliser en affichant les données retournées dans la console du navigateur.

1. Charger le fichier `yesno.html` et consulter les messages de la console.
2. Comprendre le code de `simpleUseThen` et `simpleUseAwait`.
3. Écrire une fonction `oneYesNo()` qui ajoute un élément `section` dans le document HTML qui contient un élément `article` avec l'image (`img`) de la réponse et son texte, `'yes'` ou `'no'` (élément `h2`).
   a. On écrira une version avec les promesses (suffixe Then).
   b. On écrira une version avec await/async (suffixe Await).
4. Écrire une fonction `severalYesNo(count)` qui ajoute une section au document HTML contenant `count` éléments `article` contenant chacun le résultat d'un appel à `yesNo()`.
   a. On écrira une version avec les promesses (suffixe Then).
   b. On écrira une version avec await/async (suffixe Await).
   c. Comment se chargent les images ? Quel est le temps nécessaire pour charger toutes les images ?
5. Écrire une fonction `yesNoCount(times)` qui appelle `times` fois `yesNo()` et « retourne » le nombre de réponses `'yes'` et le nombre de réponses `'no'`.
   a. On écrira une version avec les promesses (suffixe Then).
   b. On écrira une version avec await/async (suffixe Await).
   c. Quel est le temps nécessaire pour obtenir le résultat ?

Synthèse

Exploitons TheCocktailDB

Construire une petite applications qui récupère tous les cocktails proposés par TheCocktailDB et propose de n'afficher que ceux qui utilisent certains ingrédients (sélectionnés dans la liste de tous les ingrédients utilisés).

Sommaire

- Des objets sans classe
- Définir une classe à l'ancienne
- Syntaxe ES6 (à préférer)
- Et les propriétés, les attributs et méthodes de classe...

Définir un objet sans classe

```
const o1 = {     // un objet
    x: 1,        // attribut
    y: 2,

    translater: function (dx, dy) { // méthode
        this.x += dx;
        this.y += dy;
    },

    deplacer: (dx, dy) => { //! Ne fonctionne pas
        this.x += dx;   // Ici, pas de this englobant
        this.y += dy;
    },
};
```

```
console.log(o1);
console.log("== translater");
o1.translater(1, 2);
console.log(o1);
try {
    console.log("== deplacer");
    o1.deplacer(1, 2);
    console.log(o1);
} catch (e) {
    console.log(e.message);
}
```

```
{
  x: 1,
  y: 2,
  translater: [Function: translater],
  deplacer: [Function: deplacer]
}
== translater
{
  x: 2,
  y: 4,
  translater: [Function: translater],
  deplacer: [Function: deplacer]
}
== deplacer
Cannot read properties of undefined (reading 'x')
```

Définir un deuxième objet sans classe

```
const o2 = new Object();
o2.x = 1;
o2.y = 2;
o2.translater = function (dx, dy) {
    this.x += dx;
    this.y += dy;
}
o2.deplacer = (dx, dy) => {
    this.x += dx;
    this.y += dy;
}
```

```
console.log(o2);
o2.translater(1, 2);
console.log(o2);
assert.notDeepEqual(o1, o2);
    // o1 et o2 différents
    // à cause des méthodes

o2.translater = o1.translater;
o2.deplacer = o1.deplacer

assert.deepEqual(o1, o2);
```

```
{
  x: 1,
  y: 2,
  translater: [Function (anonymous)],
  deplacer: [Function (anonymous)]
}
{
  x: 2,
  y: 4,
  translater: [Function (anonymous)],
  deplacer: [Function (anonymous)]
}
```

Conclusion : Pas très pratique. . . mais utile pour un seul objet (forme avec {}).

Constructeur et méthodes

```
function Point(x, y) { // constructeur
    this.x = x;
    this.y = y;
}

Point.prototype.translater = function (dx, dy) {
    this.x += dx;
    this.y += dy;
}
```

Constructeur

- Une fonction qui initialise un objet dont le nom est `this` (prédéfini)
- Son rôle est donc de définir et initialiser les attributs de l'objet
- Par convention son nom commence par une majuscule

Méthodes

- Les méthodes sont définies dans le prototype de la classe
- Elles seront partagées par tous les objets de la classe

Création des objets

```
const p1 = new Point(1, 2);
console.log("p1 =", p1);
p1.translater(1, 2);
console.log("p1 =", p1);
console.log("p1 =", p1.toString());
console.log("Point.prototype =", Point.prototype);
console.log("Point.prototype.constructor =", Point.prototype.constructor);
```

```
p1 = Point { x: 1, y: 2 }
p1 = Point { x: 2, y: 4 }
p1 = [object Object]
Point.prototype = { translater: [Function (anonymous)] }
Point.prototype.constructor = [Function: Point]
```

- Quand on affiche les objets, les méthodes n'apparaissent pas.
- Elles sont dans le prototype.
- Le prototype a un attribut qui correspond au constructeur
- La méthode `toString` existe (pas d'erreur) mais résultat général

Ne pas oublier le `new`

```
try {
    const p2 = Point(1, 2);
} catch (e) {
    console.log(e.message);
}
```

```
Cannot set properties of undefined (setting 'x')
```

La méthode toString

```
Point.prototype.toString = function() {
    return `(${this.x}, ${this.y})`;
}
```

```
console.log("p1 =", p1);
console.log("(toString) p1 =",
    p1.toString());


console.log("Point.prototype =",
    Point.prototype);

console.log("Point.prototype.constructor =",
    Point.prototype.constructor);
```

```
p1 = Point { x: 2, y: 4 }
(toString) p1 = (2, 4)
Point.prototype = {
  translater: [Function (anonymous)],
  toString: [Function (anonymous)]
}
Point.prototype.constructor = [Function: Point]
```

L'héritage

```
function PointNomme(nom, x, y) {
    Point.call(this, x, y); // appelé la fonction Point sur cet objet (this)

    this.nom = nom;
}
```

- `Point.call` : exécuter la fonction `Point` avec les paramètres fournis
- On passe explicitement `this` dans les paramètres
- Appelle donc le constructeur de `Point` pour initialiser dans `PointNomme` les mêmes attributs
- On n'a pas (encore) accès aux méthodes de `Point` !

```
try {
    const pn0 = new PointNomme("A", 1, 2);
    console.log("pn0 =", pn0);
    console.log("(toString) pn0 =",
        pn0.toString());

    pn0.translater(2, 2);
} catch (e) {
    console.log (e.message);
}
console.log("PointNomme.prototype =",
    PointNomme.prototype);
console.log("PointNomme.prototype.constructor =",
    PointNomme.prototype.constructor);
```

```
pn0 = PointNomme { x: 1, y: 2, nom: 'A' }
(toString) pn0 = [object Object]
pn0.translater is not a function
PointNomme.prototype = {}
PointNomme.prototype.constructor = [Function: PointNom
```

Établir le lien d'héritage via le prototype

```
PointNomme.prototype = Object.create(Point.prototype);
PointNomme.prototype.constructor = PointNomme;
```

- On définit le prototype puis on rétablit le constructeur
- `Object.create` : crée un object avec pour prototype l'objet en paramètre

```
const pn0 = new PointNomme("A", 1, 2);
console.log("pn0 =", pn0);
pn0.translater(2, 2);
console.log("(toString) pn0 =",
    pn0.toString());

console.log("PointNomme.prototype =",
    PointNomme.prototype);
console.log("PointNomme.prototype.constructor =",
    PointNomme.prototype.constructor);
```

```
pn0 = PointNomme { x: 1, y: 2, nom: 'A' }
(toString) pn0 = (3, 4)
PointNomme.prototype = Point { constructor: [Function:
PointNomme.prototype.constructor = [Function: PointNom
```

Redéfinition de méthodes

```
PointNomme.prototype.toString = function() {
    return `${this.nom}:${Point.prototype.toString.call(this)}`;
}
```

- La méthode « redéfinie » est définit dans le prototype de la classe.
- Lors de l'appel d'une méthode elle est cherchée :
  - dans l'objet, puis
  - dans son prototype, puis
  - dans le prototype du prototype, etc.
- On le constate avec `toString` :
  - Quand seul le constructeur était défini, la méthode `toString` appelée était celle de Object,
  - puis quand le prototype a été correctement définie, celle de `Point`
  - et maintenant celle de `PointNommé`.

```
console.log("(toString) pn0 =",
    pn0.toString());
console.log("PointNomme.prototype =",
    PointNomme.prototype);
```

```
(toString) pn0 = A:(3, 4)
PointNomme.prototype = Point {
  constructor: [Function: PointNomme],
  toString: [Function (anonymous)]
}
```

Définition d'une classe (syntaxe ES6)

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    translater(dx, dy) {
        this.x += dx;
        this.y += dy;
    }

    distance(autre) {
        const dx2 = (this.x - autre.x) ** 2;
        const dy2 = (this.y - autre.y) ** 2;
        return (dx2 + dy2) ** .5;
    }

    toString() {
        return `(${this.x}, ${this.y})`;
    }
}
```

Définition d'une sous-classe (syntaxe ES6)

```
class PointNomme extends Point {
    constructor(nom, x, y) {
        super(x, y);
        this.nom = nom;
    }

    toString() {
        return this.nom + ':' + super.toString();
    }
}
```

- `super(...)` : appel du constructeur de la superclasse
  - initialise la partie Point du PointNommé
- `super.toString()` : appeler la méthode `toString` telle que définie dans la superclasse.

Utilisation

```
const pp1 = new Point(0, 0);
console.log("pp1 =", pp1);
console.log("pp1 = " + pp1);
pp1.translater(1, 2);
console.log("pp1 = " + pp1);

const pp2 = new Point(4, 6);
assert.equal(pp1.distance(pp2), 5);

const pn1 = new PointNomme("A", 0, 0);
console.log("pn1 =", pn1);
console.log("pn1 = " + pn1);
pn1.translater(1, 2);
console.log("pn1 = " + pn1);
assert.equal(pn1.distance(pp2), 5);
```

```
pp1 = Point { x: 0, y: 0 }
pp1 = (0, 0)
pp1 = (1, 2)

pn1 = PointNomme { x: 0, y: 0, nom: 'A' }
pn1 = A:(0, 0)
pn1 = A:(1, 2)
```

Propriétés, static... sur la classe Duree

```
class Duree {
    #duree = 0  //! déclarer un attribut privé (obligatoire)
    static MINUTES_PAR_HEURE = 60;

    constructor (heures, minutes) {
        this.#duree = heures * Duree.MINUTES_PAR_HEURE + minutes;
    }

    #setDuree(heures, minutes) {
        this.#duree = heures * Duree.MINUTES_PAR_HEURE + minutes;
    }
            //! Écriture sur 1 ligne : gain de place pour la diapositive
    get minutes()   { return this.#duree % Duree.MINUTES_PAR_HEURE; }
    set minutes(m)  { this.#setDuree(this.heures, m); }
    get heures()    { return Math.floor(this.#duree / Duree.MINUTES_PAR_HEURE); }
    set heures(h)   { this.#setDuree(h, this.minutes); }

    toString() {
        return `${this.heures}:${(this.minutes < 10 ? '0' : '') + this.minutes}`
}   }
```

- get : définir une propriété en lecture (accesseur, *getter*)
- set : définir une propriété en modification (mutateur, *setter*)
- `static` : attribut et méthode de classe
- # : attribut ou méthode privée (non utilisable hors de la classe)

Utilisation de la classe Duree

```
console.log("Duree.MINUTES_PAR_HEURE:",
        Duree.MINUTES_PAR_HEURE);
const d1 = new Duree(1, 28);
console.log("d1 = ", d1);
console.log("d1 = " + d1);
console.log("d1.minutes = " + d1.minutes);
console.log("d1.heures = " + d1.heures);
d1.heures = 2;
console.log("d1 = " + d1);
d1.minutes = 59;
console.log("d1 = " + d1);
d1.minutes = 62;
console.log("d1 = " + d1);

// console.log("d1.#duree", d1.#duree); // Syntax error.
// d1.#setDuree(10, 55);                // Syntax error.
```

```
Duree.MINUTES_PAR_HEURE: 60
d1 =  Duree {}
d1 = 1:28
d1.minutes = 28
d1.heures = 1
d1 = 2:28
d1 = 2:59
d1 = 3:02
```

Sommaire

Pourquoi TypeScript

Motivation

- Sur-ensemble de JavaScript développé par Microsoft
- Ajoute le typage statique : **détection au plus tôt des erreurs**
  - d'où le nom du langage !
- A proposé des extensions qui ont fait leur chemin dans JavaScript :
  - les fonctions fléchées (fat arrow)
  - la syntaxe de classes

Intérêt du typage statique

- Lisibilité : les informations de type aident à l' explicitation de l'intention
- Économique :
  - les informations de type devraient être écrites dans les commentaires
  - l'inférence de type fait que certains types peuvent être omis
- Détection des erreurs plutôt (15% des erreurs détectées ainsi)
- Évolutivité :
  - lors de l'évolution de l'application, plus facile d'identifier l'impact des modifications
  - plus facile de maintenir une application de grande taille

Inconvénients

- Écrire les informations de type prend un peu de temps. . . mais c'est rentables
- Les types peuvent être difficiles à exprimer si :
  - une variable est affectée avec des objets de types différents. . . pas une bonne idée !
  - une fonction peut renvoyer des objets de types différents. . . pas une bonne idée

Principe

Principe

- Les programmes TypeScript s'écrivent dans des fichiers d'extension `.ts`
- Ils ne sont pas directement exécutables.
- Ils sont traduits en JavaScript (via un compilateur TS -> JS)

Installation

```
npm install -g typescript
tsc -v                          # affiche numéro de version si installation ok
tsc hello.ts                    # produit hello.js
tsc hello.ts --outfile bonjour.js
```

Exemple

hello.ts

```ts
const hello = (nom: string) =>  {
    return 'Bonjour ' + nom;
};
{
    const monNom = 'Xavier';
    console.log(hello(monNom));
}
```

hello.js

```js
var hello = function (nom) {
    return 'Bonjour ' + nom;
};
{
    var monNom = 'Xavier';
    console.log(hello(monNom));
}
```

Un exemple

Que penser du programme suivant ?

```
const origine = {x: 0, y: 0};
let nombres = [1, 6, 3, 4];
let titre = 'des nombres';
const taille = nombres.lenght;
titre = nombres[1];
const premier = nombres.pop(0);
nombres.unshift('10');
nombres.push(origine.X);
console.log(nombres);
nombres.append(10);
```

Que signale `tsc` sur le programme précédent ?

Résultat fournis par tsc

Message d'erreurs signalés par tsc

```
exemple_erreurs.ts(4,24): error TS2551: Property 'lenght' does not exist on type 'number[]'. Did you mean 'lengt
exemple_erreurs.ts(5,1): error TS2322: Type 'number' is not assignable to type 'string'.
exemple_erreurs.ts(6,29): error TS2554: Expected 0 arguments, but got 1.
exemple_erreurs.ts(7,17): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number
exemple_erreurs.ts(8,22): error TS2551: Property 'X' does not exist on type '{ x: number; y: number; }'. Did you
exemple_erreurs.ts(10,9): error TS2339: Property 'append' does not exist on type 'number[]'.
```

Remarques

- tsc est plus bavard que ça...
- il donne des explications sur le contexte
- il propose des corrections possibles

Déclaration d'un type

```
nom: type   // variable locale, paramètre, attribut
```

Inférence de type

TypeScript pour déduire le type d'une variable du contexte.

```
let titre = 'des nombres';      // let titre: string = 'des nombres';
let nombres = [1, 6, 3, 4];     // let nombres: number[] = [1, 6, 3, 4];
```

Définition des types

Types élémentaires

- `string` : chaînes de caractères
- `number` : nombres
- `bigint` : entiers non bornées
- `boolean` : booléens

Type union |

- Syntaxe : `T1 | T2`
- Soit un objet de type T1, soit un objet de type T2

```
let x : number | string;
x = '10';
x = 21;
```

Type `object`

- Pour n'importe quel objet. On peut faire plus précis...

Type tableau

- ajouter `[]` après le type des éléments du tableau : `number[]`, `string[]`, `(number | string)[]`, etc.

Les n-uplets (tuple)

Les n-uplets (tuple)

- Principe : définir le type de chaque élément d'un tableau

```
let place: number;
let personne: [string, number, boolean] = ['Elise', 1, true];
const prenom: string = personne[0]; // ok
place = personne[1];  // ok
```

Les fonctions

Pas de surprise. . .

```
type Unite = "cm" | "p" | "m";  // alias de type

function longueur_cm(valeur: number = 0, unite: Unite = 'cm'): number {
    switch (unite) {
        case 'm':
            valeur *= 100;
        case 'cm':              // fall through
            return valeur;
        case 'p':
            return valeur * 2.54;
    }
}
```

Remarques

- si le type peut être déduit d'une valeur par défaut, inutile de le préciser (exemple : valeur)
  - ok pour `valeur`
  - ko pour `unite` car le type serait `string` et non `Unite`
- Le type du résultat peut être inféré du code de la fonction
  - déconseillé car ne peut alors pas vérifier le code de la fonction
  - pratique pour du code ancien dont les informations de type sont incomplètes

```
function longueur_cm(valeur = 0, unite: Unite = 'cm') { ...
```

Les paramètres optionnels

- Mettre un ? après un nom de paramètre signifie qu'il peut ne pas être fourni
- Sa valeur est alors `undefined`

```
function f(a: number, b?: number): number {
    if (b !== undefined) {
        return a * b;
    } else {
        return a;
    }
}

console.log("f(2, 3) =", f(2, 3));  // 6
console.log("f(2) =", f(2));        // 2
```

Pas de type de retour : void

```
function log(...objets: any[]): void {
    console.log(...objets);
}
```

Définir un type fonction

```
let autre : (valeur: number, unite: Unite) => number;

autre = (valeur: number, unite: Unite): number => {
    return Number(valeur);
};

type number2boolean = (_: number) => boolean;

const estImpair: number2boolean = (x: number): boolean => {
    return x % 2 === 1;
};

function filter(elements: number[], critere: (x: number) => boolean): number[] {
    const resultat = [];
    for (const x of elements) {
        if (critere(x)) {
            resultat.push(x);
        }
    }
    return resultat;
}

console.log("filter > 2:", filter([1, 3, 2], x => x > 2));
console.log("filter impair:", filter([1, 3, 2], estImpair));
```

Les objets

Les objets

- Principe : préciser le type de chaque attribut
- Vérifie la bonne utilisation et la bonne initialisation de la variable

```
let robot: {    // définition du type de la variable robot
    x: number;
    y: number;
    direction: string;
};

robot = { // définition de la variable
    x: 1,
    y: 2,
    direction: 'est',
};

robot.direction = 2;     // Type 'number' is not assignable to type 'string'.
console.log(robot.z);    // Property 'z' does not exist on type '{ x: number; y: number; direction: string; }'

robot = {
    X: 10,         // Object literal may only specify known properties,
    y: 'loin',     // Type 'string' is not assignable to type 'number'.
    // Type is missing the following properties from type '{ x: number; y: number; direction: string; }': x, dire
};
```

Les interfaces

Principe

- Définir un nom de type objet (convention, nom en majsucule)
- Utiliser ce nom pour typer les variables
- Mêmes vérification qu'avec les objets (diapositive précédente)
- **Intérêt :** factoriser la définition du type

```
interface Robot {
    x: number;
    y: number;
    direction: string;
}

let r1: Robot = {
    x: 10,
    y: 20,
    direction: 'est',
};
```

Et les méthodes ?

Définition des méthodes

- soit la notation classique
- soit la notation fléchée (fat arrow)

```
interface Point {
    x: number;
    y: number;
    translater(dx: number, dy: number): void;
    deplacer: (dx: number, dy: number) => void;
    distance(autre: Point): number;
    distance_bis: (autre: Point) => number;
}
```

Et les classes

C'est la syntaxe reprise par ES6 (déjà vue).

Types spéciaux

Types spéciaux

- `any` : n'importe quelle valeur (désactive le contrôle de type)
- `unknown` : un type inconnu.
  - On peut l'affecter.
  - On ne peut pas s'en servir pour initialiser un varialbe d'un autre type

```
let x1: any = 10;
let y1: string = x1;    // OK

let x2: unknown = 10;
let y2: string = x2;    // error TS2322: Type 'unknown' is not assignable to type 'string'.
let y3: string = x2 as string
```

D'autres concepts

- Une classe peut réaliser une interface : `implements`
- Héritage entre les interfaces : `extends`
- La généricité : `<T>`
- L'opérateur bang (`!`)
- ...

Références

- Références intéressantes :
  - freeCodeCamp: The Ultimate Beginners Guide
  - freeCodeCamp : full tutorial : peut-être. . .