

JavaScript

ES6 (ES2015)

Christophe Lecoutre
lecoutre@cril.fr

IUT de Lens - CRIL CNRS UMR 8188
Université d'Artois
France

Septembre 2017

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery



- ▶ Ethan Brown
Learning JavaScript, 3rd Edition, O'reilly, 2016.



- ▶ Mozilla Developer Network (MDN)
<https://developer.mozilla.org/fr/docs/Web/JavaScript>



JavaScript est un langage de programmation de plus en plus utilisé pour le développement Web.

- ▶ Il est formé principalement de :
 - ▶ ECMAScript, qui fournit les fonctionnalités centrales
- mais aussi de :
 - ▶ DOM (Document Object Model) qui fournit les fonctionnalités pour interagir avec une page web
 - ▶ BOM (Browser Object Model) qui fournit les fonctionnalités pour interagir avec le navigateur

Versions de ECMAScript :

- ▶ ES5, ECMAScript 5.1, publié en 2011
- ▶ ES6, ECMAScript 6, également appelé Harmony, publié en 2015
- ▶ ES7 (2016) et ES8 (2017), releases 'mineures'

JavaScript couplé à HTML

L'élément HTML `<script>` permet d'utiliser du code JavaScript avec une page HTML. Deux attributs courants pour cet élément sont :

- ▶ `src` : indique le fichier dans lequel se situe le code
- ▶ `type` : indique le type de contenu (appelé aussi type MIME).

Remark

La valeur par défaut de `type` étant `"text/javascript"`, on peut éviter de l'écrire.

Inline code. Il suffit d'utiliser l'élément `<script>` directement dans la page HTML, sans attributs.



Example

```
<script>
  function sayHi() {
    console.log('Hi!');
  }
</script>
```

Attention, une erreur se produit si `<` apparaît dans le code.



Example

```
<script>
  function test(a,b) {
    if (a < b) console.log('smaller'); // Erreur
  }
</script>
```

Deux solutions :

1. remplacer `<` par `<`;
2. utiliser CDATA



Example

```
<script>
// <![CDATA
  function test(a,b) {
    if (a < b) console.log('smaller');
  }
// ]]>
</script>
```

External Files

Il suffit donc d'utiliser l'élément `<script>` avec l'attribut `src`.

Pour exécuter le code après le chargement de la page, on utilisera typiquement :



```
<body>
  <!-- HTML content here -->

  <script src='example1.js'> </script>
  <script src='example2.js'> </script>
</body>
```

Remark

Pour garantir que le document HTML est totalement chargé avant d'exécuter du code JS, on pourra utiliser par exemple l'instruction `$ (document) .ready ()` de jQuery.

Inline Code or External Files ?

Il est préférable d'utiliser des fichiers externes pour des raisons de :

- ▶ maintenabilité : le code JavaScript peut être rassemblé dans un même répertoire (ou arborescence)
- ▶ caching : un fichier JS partagé par deux pages ne sera téléchargé qu'une seule fois
- ▶ lisibilité : pas besoin de `<![CDATA`

Remark

L'élément `<noscript>` permet d'afficher un contenu lorsque le navigateur ne supporte pas JS.



Example

```
<body>
  <noscript>
    <p> Cette page nécessite JavaScript </p>
  </noscript>
```

La compatibilité pour ES6 des moteurs JS est visible sur :

<https://kangax.github.io/compat-table/es6/>

Il est recommandé de transformer (compiler) le code ES6 en code ES5 à l'aide d'un transpiler tel que :

- ▶ Traceur
- ▶ Babel

Remark

Il est préférable de procéder à la compilation en amont plutôt que de la faire à la volée au chargement de page (voir diapo suivante).

Pour convertir du code ES6 en code ES5 à la volée (au chargement de pages), on peut utiliser par exemple Traceur comme suit :



```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title> My Wonderful Page </title>
  </head>
  <body>
    ...
    <script src="traceur.js"></script>
    <script src="BrowserSystem.js"></script>
    <script src="bootstrap.js"></script>
    <script type="module">
      // code ES6 ici
    </script>
  </body>
</html>
```

Remark

A noter la valeur "module" pour l'attribut type.

JavaScript with Node



Node.js is an open source server framework. It can be used to interpret directly JavaScript code.

Installing nvm (node version manager):


```
curl -o- https://raw.githubusercontent.com/creationix/  
nvm/v0.33.5/install.sh | bash
```

See <https://github.com/creationix/nvm>.

Installing node:

```
nvm install node
```

Le code suivant, dans un fichier *src/test.js* :



```
let x = 3;  
let y = (v => v*v) (x);  
console.log(`La valeur de y est : ${y}`);
```

peut être exécuté comme suit :

```
node src/test.js
```

sauf si la version de node ne le permet pas.


Après transpilation avec babel (une fois installé) :

```
./node_modules/.bin/babel src --presets es2015 --out-dir dst
```

on peut exécuter :

```
node dst/test.js
```

On peut gérer la transpilation “automatiquement” avec npm, en utilisant un fichier package.json :



```
{
  "name": "test",
  "version": "1.0.0",
  "scripts": {
    "compile": "babel src --presets es2015 --out-dir dst",
    "start": "node dst/test.js",
  },
  "author": "Toto",
  "dependencies": {},
  "devDependencies": {
    "babel-cli": "^6.11.4",
    "babel-preset-es2015": "^6.9.0",
  }
}
```

On exécute alors :

```
npm install
npm run compile
npm start
```

Et si on doit gérer un proxy :

```
npm config set proxy http://cache-adm.univ-artois.fr:8080
npm config delete proxy
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators


Expressions régulières

DOM and Events

jQuery

Commentaires et Identificateurs

Les commentaires :



```
// single line comment

/*
 * multi-line
 * comment
 */
```

Les identificateurs :

- ▶ sont sensibles à la casse : `test` n'est pas `Test`
- ▶ utilisent par convention le style *camel case*, et non le *snake case* : on écrit `sommeNotes` et non `somme_notes`

Remark

Les identificateurs peuvent aussi commencer par 1 ou 2 underscores ('_'). C'est une convention pour des variables spéciales ou 'internes'.

Instructions

Les instructions :

- ▶ se terminent par un point-virgule
- ▶ nécessitent des accolades lorsqu'elles forment un bloc (de plus d'une instruction)



Example

```
let a = 10, b = 5;  
...  
if (a > b) {  
  let tmp = a;  
  a = b;  
  b = tmp;  
}
```

Variables et Constantes

Les mots-clés à utiliser sont :

- ▶ `let` pour une variable
- ▶ `const` pour une constante



Example

```
let sum; // équivalent à 'let sum = undefined;'
let a, b, c = -1; // multiple déclaration
let v = 100;
...
v = 'coucou'; // autorisé mais fortement déconseillé
...
const LIMIT = 10;
const defaultName = 'toto';
```

Warning

On évitera dorénavant l'utilisation de `var` pour déclarer une variable.

Les instructions *let* et *const* définissent des variables et constantes qui sont locales au bloc où elles apparaissent.



Exemple

```
{  
  const x = 3;  
  console.log(x);  
}  
console.log(x); // error
```

Les paramètres de fonction sont aussi locaux à la fonction.



Exemple

```
function f(x) {  
  return x + 3;  
}  
f(5); // 8  
console.log(x); // erreur
```

Une variable définie dans un bloc sera accessible dans tout autre bloc interne (sauf si elle est masquée).



Example

```
let x = 3; // variable de portée globale
let y = 5;
if (true) {
  console.log(x); // erreur car x local pas encore défini
  console.log(y); // 5
  const x = 8; // x global est masqué
  console.log(x); // 8
  if (true) {
    console.log(x); // 8
    console.log(y); // 5
  }
  console.log(x); // 8
}
console.log(x); // 3
```

Types primitifs

Il y a principalement trois types primitifs de données :

- ▶ *boolean*
- ▶ *number*
- ▶ *string*

Il y également trois types primitifs spéciaux :

- ▶ *undefined*
- ▶ *null*
- ▶ *symbol*

Excepté pour *undefined* et *null*, pour chaque valeur primitive il existe un objet équivalent (wrapper) qui peut la contenir: Boolean pour *boolean*, Number pour *number*, String pour *string* et Symbol pour *symbol*.

Type *boolean*

Le type *boolean* comporte deux valeurs : *false* et *true*

Dans un contexte approprié, toute valeur peut être évaluée comme un booléen. Une valeur est considérée comme "falsy" (*false*) si elle est :

- ▶ undefined ou null
- ▶ false
- ▶ 0 ou NaN
- ▶ "" (chaîne de caractères vide)



Example

```
let found = false;  
  
let x; // undefined  
if (x) // condition évaluée à false  
    ...
```

Warning

Tout objet (non null), tout tableau, toute chaîne de longueur supérieure à 0 (comme '' ou 'false') est considérée comme *true*.

Conversion booléenne

Pour convertir n'importe quelle valeur en une valeur booléenne, on peut utiliser :

- ▶ soit !! (double négation)
- ▶ soit la fonction Boolean (sans new)



Example

```
let n = 0; // valeur "falsy"  
let b1 = !!n; // false  
let b2 = Boolean(n); // false (même résultat)
```

Type *number*

Le type *number* sert à représenter aussi bien des valeurs entières que des valeurs réelles.



Example

```
let x = 55;
let y = 070; // octal pour 56 (en base 10)
let z = 0b00001111; // valeur binaire pour 15
const = 0xA, // hexadecimal pour 10
let f1 = 10.2;
let f2 = 3.12e7; // représente 31 200 000
const f3 = 3e-10; // représente 0,0000000003
```

Remark

Les valeurs particulières de l'objet Number sont :

- ▶ Number.MIN_VALUE, Number.MAX_VALUE
- ▶ Number.MIN_SAFE_INTEGER, Number.MAX_SAFE_INTEGER
- ▶ Number.NEGATIVE_INFINITY, Number.POSITIVE_INFINITY
- ▶ Number.EPSILON, Number.NaN (Not a Number)

Conversion numérique

Relatifs aux valeurs et conversions numériques, on trouve ces fonctions :

- ▶ `Number` effectue une conversion
- ▶ `parseInt` effectue une conversion en valeur entière
- ▶ `parseFloat` effectue une conversion en valeur réelle



Example

```
let num1 = Number("hello world"); // NaN
let num2 = Number("00001"); // 1
let num3 = Number(true); // 1
let num3 = parseInt(""); // NaN
let num4 = parseInt(22.5); // 22
let num5 = parseInt("70", 10); // 70 - base 10 spécifiée
let num6 = parseFloat("22.5"); // 22.5
```

Remark

La fonction `Number.isInteger()` permet de tester si une valeur donnée est un entier. Voir aussi `Number.isSafeInteger()`.

Méthode Number.isNaN()

Revoie *true* seulement si le paramètre est un nombre qui vaut NaN.



Example

```
Number.isNaN(NaN); // true
Number.isNaN(Number.NaN); // true
Number.isNaN(0 / 0); // true

// tout le reste renverra : false
Number.isNaN(undefined); // false
Number.isNaN({}); // false

Number.isNaN(true); // false
Number.isNaN(null); // false
Number.isNaN(37); // false
```

Warning

1. Les deux opérateurs d'égalité, `==` et `===`, renvoient *false* quand on teste que NaN est NaN.
2. La fonction globale `isNaN()` a un comportement différent. Voir [MDN isNaN\(\)](#).

Opérations numériques avec Math

Il s'agit d'un objet définissant de nombreuses constantes et fonctions mathématiques.



Example

```
Math.E; // la valeur de e
Math.PI; // la valeur de pi
Math.min(5,12); // 5
Math.max(23,5,7,130,12); // 130
Math.ceil(25.3); // 26
Math.floor(25.8); //25
Math.random(); // valeur aléatoire entre 0 et 1
let n = Math.floor(Math.random()*nb + min);
n; // valeur aléatoire entre min et min+nb (exclus)
```

D'autres fonctions :

Math.abs(x)

Math.exp(x)

Math.log(x)

Math.pow(x,y)

Math.sqrt(x)

Math.sin(x)

Math.cos(x)

Math.tan(x)



Example

```
const x = 19.51;
x.toFixed(3); // '19.150'
x.toFixed(1); // '19.1'

const y = 3800.5;
y.toExponential(3); // '3.801e+4'
y.toExponential(1); // '3.8e+4'

const z = 1000;
z.toPrecision(5); // '1000.0'
z.toPrecision(2); // '1.0e+3'

const w = 12;
w.toString(); // '12' (base 10)
w.toString(16); // 'c' (base 16)
w.toString(8); // '14' (base 8)
w.toString(2); // '1100' (base 2)
```

Type *string*

On utilise les quotes simples (apostrophes) ou doubles (guillemets) pour définir des valeurs chaînes de caractères. L'attribut `length` permet de déterminer la longueur d'une string.



Example

```
let nom = "Wilde", prenom = 'Oscar';  
console.log(prenom.length); // 6  
let message1 = "toto a dit \"je suis malade\".";   
let message2 = "toto a dit 'je suis malade'.";
```

Afficher dans la console la concaténation d'une string avec une valeur numérique :



Example

```
let x = 10;  
console.log("x = ", x);  
  
let y = 20;  
console.log("y = " + y);
```

String templates

En utilisant les backticks (``), depuis ES6, à la place des quotes (simples ou doubles), il est possible d'injecter la valeur d'une variable ou expression numérique dans une string.



Example

```
let monday = 19.5;
console.log("Temperature on Monday is " + monday
           + "\u00b0C");

let friday = 22;
console.log(`Temperature on Friday is ${friday}\u00b0C`);
```

On obtient :

```
Temperature on Monday is 19.5°C
Temperature on Friday is 22°C
```

Méthodes sur les *string*

Il existe de nombreuses méthodes sur les string.



Example

```
let s = "hello world";  
s.length; // 11  
s.charAt(1); // "e"  
s.charCodeAt(1); // 101  
s.slice(3); // "lo world"  
s.slice(-3); // "rld"  
s.substring(3,7); // "lo w"  
s.indexOf("o"); // 4  
s.lastIndexOf("o"); // 7  
s.toUpperCase(); // HELLO WORLD  
s + " !"; // hellow world !  
"a".repeat(6); // "aaaaaa"  
s.includes("wor"); // true
```

Remark

Techniquement, la primitive string est temporairement convertie en un objet String, le temps de l'exécution de la méthode.

Types *undefined* et *null*

Ces deux types ne contiennent qu'une seule valeur (éponyme).

- ▶ la valeur *undefined* fait référence à une variable non déclarée ou déclarée mais pas initialisée.
- ▶ la valeur *null* fait référence à une variable censée référencer un objet mais non encore disponible.



Example

```
let message = "coucou";  
message; // "coucou"  
let m1;  
m1; // undefined  
m2; // erreur  
console.log(typeof m1); // "undefined"  
console.log(typeof m2); // undefined  
...  
let car = null;  
console.log(typeof car); // "object"
```


Types *symbol*

Le type *symbol* est un nouveau type (ES6) permettant de représenter des tokens uniques.

- ▶ Les valeurs sont créées avec la fonction `Symbol()` (pas de `new`)
- ▶ Il est possible de fournir une description en paramètre



Example

```
const RED = Symbol();  
const ORANGE = Symbol("The color of a sunset!");  
RED === ORANGE; // false: every symbol is unique
```

Le type *Object*

Les 6 types primitifs permettent de manipuler des valeurs immuables. Le type *Object* permet de définir des données complexes modifiables.

Un objet est une structure complexe composée de couples nom-valeurs.

Il existe de nombreux types dérivés (héritant) de *Object* :

- ▶ *Boolean*, *Number*, *String* et *Symbol*
- ▶ *Array*, *Map* et *Set*
- ▶ *Function*
- ▶ *RegExp*
- ▶ *Date*

L'opérateur `typeof`

Il est possible de déterminer le type (courant) d'une variable avec l'opérateur `typeof` qui retourne l'une des valeurs suivantes:

- ▶ `'undefined'` si la valeur est indéfinie (variable déclarée mais pas initialisée ou variable non déclarée)
- ▶ `'boolean'`
- ▶ `'number'`
- ▶ `'string'`
- ▶ `'object'` si la valeur est un objet, `null`, ou un tableau (!)
- ▶ `'function'` si la valeur est une fonction

Remark

Les fonctions sont considérées comme des objets qui ont des propriétés spéciales.

Méthodes `valueOf()` et `toString()`

- ▶ La méthode `valueOf()` renvoie la valeur primitive de l'objet sur laquelle elle est appelée
- ▶ La méthode `toString()` renvoie une chaîne de caractères représentant l'objet



Example

```
const d = new Date();  
let x = d.valueOf(); // nombre de millisecondes depuis le  
    1er Janvier 1970  
  
const n = 33.5;  
const s = n.toString();  
s; // "33.5" - une string
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Structures de contrôle

Elles sont très proches de celles de langages tels que C, C++ et Java. Pour rappel, les structures de contrôles sont de trois types :

- ▶ **Séquence** : exécution séquentielle d'une suite d'instructions séparées par un point-virgule
- ▶ **Alternative** : structure permettant un choix entre divers blocs d'instructions suivant le résultat d'un test logique
- ▶ **Boucle** : structure itérative permettant de répéter plusieurs fois la même bloc d'instructions tant qu'une condition de sortie n'est pas avérée

Remark

Toute condition utilisée pour une alternative ou boucle sera toujours placée entre parenthèses.

Warning

Il ne faut pas confondre $x == y$ (test d'égalité) avec $x = y$ (affectation).

L'instruction **if** sans partie else :

```
if (condition) instruction;           if (condition) {  
                                     instruction1;  
if (condition) {                       instruction2;  
    instruction;                       ...  
}                                     }
```



Example

```
if (x >= 0) console.log("valeur positive ou nulle");  
...  
if (note > 12 && note <= 14) {  
    console.log("bravo");  
    mention="bien";  
}
```

Alternative

L'instruction **if...else** :

```
if (condition) instruction1;  
else instruction2;
```

```
if (condition) {  
    instructions1;  
} else {  
    instructions2;  
}
```

```
if (condition1) {  
    instructions1;  
} else if (condition2) {  
    instructions2;  
} else {  
    instructions3;  
}
```



Example

```
if (rank == 1)  
    medaille="or";  
else if (rank == 2)  
    medaille="argent";  
else if (rank == 3)  
    medaille="bronze";
```


L'instruction **switch** :

```
switch (expression) {  
    case valeur1 :  
        instructions1;  
        break;  
    case valeur2 :  
        instructions2;  
        break;  
    ...  
    case valeurN :  
        instructionsN;  
        break;  
    default:  
        instructionsDefault;  
}
```

Remark

Le branchement par défaut n'est pas obligatoire.

Alternative

L'opérateur ternaire ?: permet de remplacer une instruction if...else simple. Sa syntaxe (lorsqu'utilisée pour donner une valeur à une variable) est :

```
variable = condition ? expressionIf : expressionElse;
```

Elle est équivalente à :

```
if (condition) variable = expressionIf;  
else variable = expressionElse;
```



Example

```
let civilite = (sexe == "F") ? "Madame" : "Monsieur";  
let medaille = rank == 1 ? "or" : rank == 2 ? "argent" : "  
    bronze";
```

Remark

Cet opérateur est utile pour les expressions courtes.

Boucle **while**

L'instruction **while** est la structure de boucle universelle :

```
while (condition) instruction;
```

```
while (condition) {  
    instruction1;  
    instruction2;  
    ...  
}
```



Example

```
let num = 1;  
while (num <= 5) {  
    console.log(num);  
    num++;  
}
```

Boucle **for**

L'instruction **for** délimite (en temps normal) le nombre de tours de boucle :

```
for (instructionInit; condition; instructionIter)
    instruction;
```

```
for (instructionInit; condition; instructionIter) {
    instruction1;
    instruction2;
    ...
}
```



Example

```
for (let num = 1; num <= 5; num++)
    console.log(num);
```

Boucle **do-while**

L'instruction **do...while** garantit au moins un tour de boucle :

```
do {  
  instruction1;  
  instruction2;  
  ...  
} while (condition);
```



Example

```
let i = 0;  
do {  
  i += 1;  
  console.log(i);  
} while (i < 5);
```

Boucle **for-in**

L'instruction **for-in** s'utilise avec les objets :

```
for (let key in object)
  console.log(key);
```



Example

```
const player = { name: 'John', team: 'warrior', age: 25 };
for (let key in player) {
  if (!player.hasOwnProperty(key))
    continue;
  console.log(key + ": " + player[key]);
}
```

Il est préférable de passer par un tableau via `Object.keys()`.



Example

```
Object
  .keys(player)
  .forEach(key => console.log(key + ": " + player[key]));
```

Boucle **for-of**

L'instruction **for-of** s'utilise avec les tableaux (et tout objet itérable) :

```
for (let v of array)
  console.log(v);
```



Example

```
const t = { 10, 7, 13, 4 };
for (let v of t)
  console.log(v);

// autre manière de réaliser la boucle
for (let i=0; i < t.length; i++)
  console.log(t[i]);
```

Instructions **break** et **continue**

Certaines instructions permettent un contrôle supplémentaire sur les boucles :

- ▶ **break** permet de quitter la boucle courante
- ▶ **continue** permet de terminer l'itération en cours de la boucle courante



Example

```
for (let i=0; i<5; i++) {  
  for (let j=0; j<5; j++) {  
    if ((i+j)%3 == 0)  
      continue;  
    for (let k=0; k<5; k++) {  
      if ((i+j+k)%3 == 0)  
        break;  
      console.log(i + " " + j + " " + k);  
    }  
  }  
}
```


Exercice

Écrire le code Javascript qui détermine si un nombre entier x est parfait.

Definition

Un nombre est parfait ssi il est égal à la somme de ses diviseurs stricts.

6 est parfait car $6 = 1 + 2 + 3$.



Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Fonctions

La syntaxe classique pour définir une fonction est :

```
function name(arg0, arg1, ..., argN) {  
    statements  
}
```



Example

```
function hello() {  
    return 'Hello world';  
}  
  
hello(); // 'Hello world'
```



Example

```
function sum(num1, num2) {  
    return num1 + num2;  
}  
  
sum(5,10); // 15
```

Remark

Une fonction retourne toujours un résultat, même si rien ne l'indique au niveau de la signature (en-tête). Ce résultat est :

- ▶ soit une valeur retournée par l'instruction `return`,
- ▶ soit *undefined*.

Pour toute fonction f définie :

- ▶ $f()$ représente l'appel de la fonction
- ▶ f représente la référence (adresse) de la fonction



Example

```
const f = hello;
f(); // 'hello world'

const o = {};
o.g = hello;
o.g(); // 'hello world'

const t = [1, 2, 3];
t[1] = hello;
t[1](); // 'hello world'
```

Paramètres

- ▶ un paramètre ne recevant pas de valeur à l'appel est *undefined*
- ▶ pour gérer un nombre variable de paramètres, on peut utiliser l'opérateur 'spread' ...



Example

```
function f(x) { console.log('x = ' + x); }  
f(); // x = undefined  
  
function addPrefix(prefix, ...words) {  
  return words.map(w => prefix + w);  
}  
addPrefix('con', 'cept', 'te'); // ['concept', 'conte']
```

Remark

La portée des variables représentant les paramètres est celle de la fonction.

Arguments (ES5)

Il existe toujours un tableau `arguments` implicite lors de l'appel à une fonction.



Example

```
function f() {  
  console.log(arguments.length);  
  for (let a of arguments)  
    console.log(a);  
}  
  
f("Nicolas"); // affiche 1 Nicolas  
f("Nicolas", 25); // affiche 2 Nicolas 25
```

Remark

Dorénavant, on utilisera plutôt l'opérateur 'spread' ... (d'autant plus que `arguments` ne peut être utilisé avec la notation fléchée).

Paramètres par défaut

Il suffit de préciser la valeur à la définition.



Example

```
function f(a, b = 'default', c = 3) {  
  return `${a} - ${b} - ${c}`;  
}  
  
f(5, 6, 7); // '5 - 6 - 7'  
f(5, 6);    // '5 - 6 - 3'  
f(5);       // '5 - default - 3'  
f();        // 'undefined - default - 3'  
f(5, undefined, 7); // '5 - default - 7'
```

Remark

A noter qu'on peut utiliser la valeur *undefined* à l'appel.

Eclater les valeurs

Parfois, il est nécessaire d'éclater les valeurs d'un tableau, notamment lors d'un appel à une fonction.



Example

```
function f(a, b, c) {  
  return a + b + c;  
}  
  
let t = [10, 3, 7];  
let result = f(...t);  
console.log(result); // 20
```

Remark

En ES5, il aurait fallu écrire `let result = f.apply(null,t);`

Définir une fonction

Il y a deux manières (sensiblement équivalentes) de définir une fonction. La deuxième passe par la création d'une fonction anonyme et la récupération de son adresse.



Example

```
function sum(n1, n2) {  
  return n1+n2;  
}
```

```
sum(5,10); //15
```



Example

```
let sum = function (n1, n2) {  
  return n1+n2;  
};
```

```
sum(5,10); //15
```

Remark

Il est possible de référencer une fonction par deux variables différentes.



Example

```
function sum(n1, n2) { return n1+n2; }  
let g = sum;  
g(5,10); //15
```

Fonctions "fléchées"

Il s'agit d'une notation plus moderne et compacte pour les fonctions anonymes.



Example

```
let f1 = function() { return "hello"; }  
let g1 = () => "hello";  
  
let f2 = function(name) { return "hello" + name; }  
let g2 = name => "hello" + name;  
  
let f3 = function(a, b) { return a + b; }  
let g3 = (a,b) => a + b;
```

Remark

S'il y a plus d'une instruction dans le corps de la fonction, il faut construire un corps de manière classique (avec accolades et instructions return si approprié).

Fonctions et *this*

Il est important de noter qu'au sein d'une méthode, *this* est lié :

- ▶ à la manière dont une fonction est appelée : *dynamic this*,
- ▶ et non à l'endroit où la fonction est définie : *lexical this*



Example

```
let o = {
  name: 'Jo',
  speak: function() {
    return "My name is " + this.name;
  }
}
o.speak(); // "My name is Jo"
let speak = o.speak;
speak === o.speak; // true
speak(); // "My name is undefined"
```

Remark

Lors du second appel (*speak()* sans préfixe), *this* désigne l'objet *global* avec node (et l'objet *window* avec un navigateur).

There are two other ways of defining the method *speak*.



Example

```
let o = {  
  name: 'Jo',  
  speak() { return "My name is " + this.name; }  
}
```



Example

```
let o = {  
  name: 'Jo',  
  speak: () => "My name is " + this.name  
}
```

Warning

- ▶ the first one corresponds to a *method definition* ; *this* remains dynamic.
- ▶ the second one corresponds to an arrow function ; this becomes lexical (*this* will not be bound anymore to the variable *o*). **To be avoided!**

Si on utilise *this* dans une fonction auxiliaire, cela ne convient pas.



Example

```
let o = {
  name: 'Jo',
  speakReverse() {
    function reverse() {
      return this.name.split('').reverse().join('');
    }
    return reverse() + " si eman ym";
  }
}
o.speakReverse(); // TypeError: Cannot read property
                  'split' of undefined
```

Remark

A noter comment on inverse une string avec les méthodes *split*, *reverse*, et *join*.

Tout comprendre sur les fonctions :

<http://2ality.com/2012/04/arrow-functions.html>

Une solution classique : sauver *this* dans *that* :



Example

```
let o = {  
  name: 'Jo',  
  speakReverse() {  
    const that = this;  
    function reverse() {  
      return that.name.split('').reverse().join('');  
    }  
    return reverse() + " si eman ym";  
  }  
}  
o.speakReverse(); // "oJ si eman ym"
```

Remark

Utiliser *that* est une convention. Parfois, on rencontre *self*.

Autre solution : utiliser une fonction fléchée. Le problème ne se pose pas car *this* est défini lexicalement (*this* provient du code englobant).



Example

```
let o = {
  name: 'Jo',
  speakReverse() {
    let reverse =
      () => this.name.split('').reverse().join('');
    return reverse() + " si eman ym";
  }
}
o.speakReverse(); // "oJ si eman ym"
```

Warning

Nouvel avertissement: il ne faut pas utiliser la notation fléchée pour les méthodes d'un objet car *this* est alors l'objet englobant.

Spécifier *this*

Lorsque vous appelez la méthode *call* sur une fonction *f* :

- ▶ le premier argument représente la valeur de *this* pendant l'exécution de *f*
- ▶ les autres arguments représentent les paramètres de la fonction *f*



Example

```
let jo = { name: 'Jo' };  
let alice = { name: 'alice' };  
  
function greet() { return "Hello " + this.name; }  
  
greet();           // Hello  
greet.call(jo);    // Hello Jo  
greet.call(alice); // Hello alice
```




Example

```
function update(weight, job) {  
  this.weight = weight;  
  this.job = job;  
}  
  
update.call(jo, 84, 'singer');  
jo; // { name: 'jo', weight: 84, job: 'singer' }  
update.call(alice, 47, 'actress');  
alice; // { name: 'alice', weight: 47, job: 'actress' }
```

Remark

La fonction *apply* est similaire à *call* mais place tous les arguments, hormis le premier, dans un tableau.



Example

```
let info = [77, 'pilot'];  
update.apply(jo, info);  
jo; // { name: 'jo', weight: 77, job: 'pilot' }
```

Il est possible d'utiliser l'opérateur spread ...



Example

```
update.call(jo, ...info); // same update as previously

let t = [20, 5, 8, 3, 120];
Math.min.apply(null, t); // 3
Math.min.call(null, ...t); // 3
Math.min(...t); // 3
```

La méthode *bind* permet de lier une valeur à *this* de manière permanente.



Example

```
let updateJo = update.bind(jo);
updateJo(73, 'doctor');
jo; // { name: 'jo', weight: 73, job: 'doctor' }
updateJo.call(alice, 55, 'speaker');
jo; // { name: 'jo', weight: 55, job: 'speaker' }
alice; // { name: 'jo', weight: 77, job: 'pilot' }
```

Remark

Il est possible de lier définitivement d'autres paramètres avec *bind*.

En résumé : les fonctions en JavaScript

Une fonction peut donc être définie de trois manières en JavaScript :

- ▶ comme fonction nommée
- ▶ comme fonction anonyme
- ▶ comme fonction anonyme, notation fléchée (ES6)

Mais également :

- ▶ comme définition de méthode (ES6)



Example

```
function contientZero(tab) {  
  for (let v of tab)  
    if (v == 0)  
      return true;  
  return false;  
}
```

Une fonction anonyme doit se placer à un endroit qui s'y prête. Le plus souvent, elle représente une fonction de rappel (callback).



Example

```
// anonymous function
function (tab) {
  for (let v of tab)
    if (v == 0)
      return true;
  return false;
}

// anonymous function (ES6)
(tab) => {
  for (let v of tab)
    if (v == 0)
      return true;
  return false;
}

// anonymous function (ES6) -- methods on arrays
(tab) => tab.some(v => v == 0);
```

Exemple : tri d'un tableau



Exemple

```
let t = [10, 3, 0, 12];  
t.sort();  
console.log("Après tri : " + t);
```

On peut utiliser une fonction (nommée) pour réaliser le tri.



Exemple

```
function comparaison(v1,v2) {  
  if (v1 < v2)  
    return -1;  
  if (v1 > v2)  
    return 1;  
  return 0;  
}  
  
let t = [10, 3, 0, 12];  
t.sort(comparaison);  
console.log("Après tri : " + t);
```

Exemple : tri d'un tableau



Exemple

```
let t = [10, 3, 0, 12];  
t.sort();  
console.log("Après tri : " + t);
```

On peut utiliser une fonction (nommée) pour réaliser le tri.



Exemple

```
function comparaison(v1,v2) {  
  if (v1 < v2)  
    return -1;  
  if (v1 > v2)  
    return 1;  
  return 0;  
}  
  
let t = [10, 3, 0, 12];  
t.sort(comparaison);  
console.log("Après tri : " + t);
```

On peut aussi utiliser une fonction anonyme pour réaliser le tri.



Example

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    if (v1 < v2)
        return -1;
    if (v1 > v2)
        return 1;
    return 0;
});
console.log("Après tri : " + t);
```

On peut aussi simplifier le code.



Example

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    return v1 - v2;
});
console.log("Après tri : " + t);
```

On peut aussi utiliser une fonction anonyme pour réaliser le tri.



Example

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    if (v1 < v2)
        return -1;
    if (v1 > v2)
        return 1;
    return 0;
});
console.log("Après tri : " + t);
```

On peut aussi simplifier le code.



Example

```
let t = [10, 3, 0, 12];
t.sort(function (v1,v2) {
    return v1 - v2;
});
console.log("Après tri : " + t);
```


Encore mieux, en utilisant la nouvelle syntaxe ES6.



Example

```
let t = [10, 3, 0, 12];  
t.sort((v1,v2) => v1 - v2);  
console.log("Après tri : " + t);
```

Exercice: trier avec les valeurs paires en priorité.



Example

```
let t = [10, 3, 0, 12, 5, 90, 11];  
t.sort((v1,v2) => v1%2 == 0 ?  
  (v2%2 == 0 ? v1 - v2 : -1) :  
  (v2%2 == 0 ? 1 : v1 - v2));  
console.log("Après tri : " + t);
```

Encore mieux, en utilisant la nouvelle syntaxe ES6.



Example

```
let t = [10, 3, 0, 12];  
t.sort((v1,v2) => v1 - v2);  
console.log("Après tri : " + t);
```

Exercice: trier avec les valeurs paires en priorité.



Example

```
let t = [10, 3, 0, 12, 5, 90, 11];  
t.sort((v1,v2) => v1%2 == 0 ?  
  (v2%2 == 0 ? v1 - v2 : -1) :  
  (v2%2 == 0 ? 1 : v1 - v2));  
console.log("Après tri : " + t);
```

Encore mieux, en utilisant la nouvelle syntaxe ES6.



Example

```
let t = [10, 3, 0, 12];  
t.sort((v1,v2) => v1 - v2);  
console.log("Après tri : " + t);
```

Exercice: trier avec les valeurs paires en priorité.



Example

```
let t = [10, 3, 0, 12, 5, 90, 11];  
t.sort((v1,v2) => v1%2 == 0 ?  
  (v2%2 == 0 ? v1 - v2 : -1) :  
  (v2%2 == 0 ? 1 : v1 - v2));  
console.log("Après tri : " + t);
```

Fonctions dans un tableau

Possibilité par exemple d'appliquer systématiquement et en séquence des fonctions.



Example

```
const zoom = 2, offset = [1, -3];
const pipeline = [
  function scale(p) {
    return { x: p.x*zoom, y: p.y*zoom };
  },
  function translate(p) {
    return { x: p.x + offset[0], y: p.y + offset[1] };
  }
];

const p = { x: 1, y: 1 };
let p2 = p;
for (let f of pipeline)
  p2 = f(p2);
p2; // { x: 3, y: -1 }
```

Passer une fonction en argument d'une autre fonction

En JavaScript, on utilise cela en permanence (callbacks). Voici un exemple plus académique.



Example

```
function sum(array, f) {  
  if (typeof f !== 'function')  
    f = x => x;  
  return array.reduce((a,v) => a + f(v), 0);  
}  
  
let t = [1, 2, 3];  
sum(t); // 6  
sum(t, x => x*x); // 14  
sum(t, x => Math.pow(x,3)); // 36
```

Retourner une fonction d'une autre fonction

Cela permet par exemple de spécialiser une fonction en spécifiant un ou plusieurs paramètres. Cela est connu sous le terme de *currying*.



Example

```
function newSummer(f) {  
  return array => sum(array, f);  
}  
  
const sumOfSquares = newSummer(x => x*x);  
const sumOfCubes = newSummer(x => Math.pow(x,3));  
  
let t = [1, 2, 3];  
sum(t); // 6  
sumOfSquares(t); // 14  
sumOfCubes(t); // 36
```

Une fonction définie dans un certain contexte a accès aux variables de ce contexte. Si elle est appelée plus tard dans un tout autre contexte, elle gardera un accès aux variables de son contexte de définition. Il s'agit d'une closure.



Example

```
let f;  
{  
  let x = 2;  
  f = function () {  
    return x;  
  }  
}  
f(); // 2
```

Il est possible de définir une fonction et de l'appeler immédiatement. Cela s'appelle une IIFE (Immediately Invoked Function Expression).

L'intérêt principal est qu'une IIFE possède son propre scope qui est protégé car inaccessible de l'extérieur.



Example

```
(function() {  
    // this is the IIFE body  
})();
```



Example

```
let message = (function() {  
    const code = 'coucou';  
    return 'length of the code : ' + code.length;  
})();  
message; // 6
```


On peut simuler une variable statique (comme en C++) avec une IIFE



Example

```
let incrementer = (function() {  
  let i = 0;  
  
  return () => i++;  
})();  
  
incrementer(); // 0  
incrementer(); // 1  
incrementer(); // 2
```

Remark

La variable *i* n'est pas globale et n'est pas accessible en dehors de la fonction.

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Les tableaux peuvent contenir des données de nature différente.



Example

```
let t = []; // tableau vide
let colors = ['red', 'blue', 'green'];
let t1 = [1, 2, 3];
let t2 = ['one', 2, 'three'];
let t3 = [[1, 2, 3], ['one', 2, 'three']];
let t4 = [
  { name: 'jo', age: 33 },
  1,
  () => 'hello',
  'three'
]
t1[0]; // 1
t2[2]; // three
t3[1]; // ['one', 2, 'three']
t3[1][0]; // one
t4[2](); // hello
t1.length; // 3
t4.length; // 4
t4[1].length; // 3
```

Il est possible de connaître et même de modifier la longueur d'un tableau (champ *length*).



Example

```
colors[colors.length]="black"; // nouvelle couleur
colors[99]="pink";
colors.length; // 100
colors[50]; // undefined
colors.length=10; // plus que 10 cases
```

De nombreuses méthodes existent sur les tableaux.



Example

```
let colors = ['red', 'blue', 'green'];
colors; // red, blue, green
colors.join(";"); // red;blue;green
colors.push("black");
colors; // red, blue, green, black
let item = colors.pop();
console.log(item + " " + colors); // black red,blue,green
let item2= colors.shift();
console.log(item2 + " " + colors); // red blue,green
```

Nombreuses méthodes

- ▶ Ajouter/supprimer un élément :
 - ▶ en queue de tableau : *push* et *pop*
 - ▶ en tête de tableau : *unshift* et *shift*
- ▶ Ajouter/supprimer des éléments à n'importe quelle position : *splice*
- ▶ Ajouter des éléments en queue de tableau : *concat*
- ▶ Extraire un sous-tableau : *slice*
- ▶ Remplir un tableau : *fill*
- ▶ Inverser et trier un tableau : *reverse* et *sort*
- ▶ copy and replace au sein d'un tableau : *copyWithin*
- ▶ trouver le premier ou dernier indice d'une valeur : *indexOf* et *lastIndexOf*



Example

```
let t = [0, 1, 2, 3, 4];  
t.reverse(); // 4, 3, 2, 1, 0  
t.indexOf(3); // 1  
t.slice(2, 4); // 2, 1  
t.concat(4, 30); // 4, 3, 3, 1, 0, 4, 30  
t.shift(); // 3, 2, 1, 0  
t.fill(0); // 0, 0, 0, 0, 0
```

Fonctions avec callbacks sur les tableaux

Il existe de nombreuses fonctions sur les tableaux utilisant des callbacks. Certains de ces fonctions retournent une valeur simple.



Example

```
let b1 = t.some(function (valeur) {  
    return valeur == 22;  
});  
console.log("Variable b1 = ", b1);  
  
let j = t.findIndex(function (valeur) {  
    return valeur == 22;  
});  
console.log("Variable j = ", j);  
  
let b2 = t.every(function (valeur) {  
    return valeur > 2;  
});  
console.log("Variable b2 = " + b2);
```

D'autres fonctions avec callbacks sur les tableaux retournent un (nouveau) tableau.



Example

```
let t2 = t.map(function (valeur) {  
    return valeur * 2;  
});  
console.log("Tableau t2 = " + t2);  
  
let t3 = t.filter(function (valeur) {  
    return valeur < 8;  
});  
console.log("Tableau t3 = " + t3);  
  
let t4 = ["cc", "BB", "aa", "DD"];  
t4.sort(function (x, y) {  
    return x.localeCompare(y);  
});  
console.log("Tableau t4 = " + t4);
```

Les mêmes exemples en utilisant la notation fléchée (ES6).



Example

```
let b1 = t.some(v => v == 22);
console.log("Variable b1 = ", b1);

let j = t.findIndex(v => v == 22);
console.log("Variable j = ", j);

let b2 = t.every(v => v > 2);
console.log("Variable b2 = " + b2);

let t2 = t.map(v => v * 2);
console.log("Tableau t2 = " + t2);

let t3 = t.filter(v => v < 8);
console.log("Tableau t3 = " + t3);

let t4 = ["cc", "BB", "aa", "DD"];
t4.sort((x, y) => x.localeCompare(y));
console.log("Tableau t4 = " + t4);
```


Il est possible de parcourir un tableau avec la méthode *forEach*.



Example

```
let t = [3, 4, 2, 8, 5];  
t.forEach(v => console.log(' carre : ' + v*v));
```

Remark

Pour nombre de méthodes, il est possible d'indiquer un second paramètre lors des appels : il s'agit de la valeur à lier à *this*.

Remark

Pour nombre de méthodes, il est possible d'utiliser une fonction callback à deux paramètres (et même à trois), dont le deuxième est l'indice courant.



Example

```
let t = [3, 4, 12, 8, 22, 49, 5];  
let carre = (v) => Number.isInteger(Math.sqrt(v));  
t.find((v,i) => i>2 && carre(v)); // 49
```

Map Reduce

Un exemple avec *map* où les parenthèses sont nécessaires autour du corps de la fonction fléchée à cause des accolades de création d'objets.



Exemple

```
let jeux = ['go', 'de'];  
let prix = [99, 15];  
let t = jeux.map((j,i) => ({nom: j, prix: prix[i]}));  
// [{ nom: 'go', prix: 99 }, { nom: 'de', prix: 15 }]
```

La fonction *reduce* permet de combiner toutes les valeurs d'un tableau. Le premier argument est l'accumulateur, et les trois arguments suivants sont la valeur courante, l'indice courant et le tableau lui-même.



Exemple

```
let t = [0, 1, 2, 3];  
let somme = t.reduce((a, b) => a + b);  
console.log("Somme=" + somme);
```

Remark

Si le premier élément du tableau ne peut pas servir d'initialisation de l'accumulateur, alors il faut le glisser en deuxième argument de *reduce*.

Exercice : écrire le code utilisant la fonction *reduce* permettant de calculer la valeur max présente dans un tableau t.



Example

```
// (version a)
let max = t.reduce((a, b) => {
  if (b > a) return b;
  else return a;
});
console.log("max (a) = " + max);

// (version b)
max = t.reduce((a, b) => b > a ? b : a);
console.log("max (b) = " + max);

// (version c)
max = t.reduce((a, b) => Math.max(a, b));
console.log("max (c) " + max);
```

Remark

Si le premier élément du tableau ne peut pas servir d'initialisation de l'accumulateur, alors il faut le glisser en deuxième argument de *reduce*.

Exercice : écrire le code utilisant la fonction *reduce* permettant de calculer la valeur max présente dans un tableau t.



Example

```
// (version a)
let max = t.reduce((a, b) => {
  if (b > a) return b;
  else return a;
});
console.log("max (a) = " + max);

// (version b)
max = t.reduce((a, b) => b > a ? b : a);
console.log("max (b) = " + max);

// (version c)
max = t.reduce((a, b) => Math.max(a, b));
console.log("max (c) " + max);
```

Exercice : compter le nombre de lettres cumulé sur le nombre de mots d'un tableau donné.



Example

```
let pets = ['cat', 'dog', 'fish'];  
let nb = pets.map(s => s.length).reduce((a, b) => a + b);  
console.log('nb = ' + nb);
```

Exercice : écrire le code permettant de construire un nouveau tableau obtenu en filtrant toutes les valeurs paires d'un tableau, et en transformant chaque valeur restante en son carré.



Example

```
let tab = t.filter(v => v % 2 !== 0).map(v => v * v);  
console.log('Tableau tab : ' + tab);
```

Exercice : compter le nombre de lettres cumulé sur le nombre de mots d'un tableau donné.



Example

```
let pets = ['cat', 'dog', 'fish'];  
let nb = pets.map(s => s.length).reduce((a, b) => a + b);  
console.log('nb = ' + nb);
```

Exercice : écrire le code permettant de construire un nouveau tableau obtenu en filtrant toutes les valeurs paires d'un tableau, et en transformant chaque valeur restante en son carré.



Example

```
let tab = t.filter(v => v % 2 !== 0).map(v => v * v);  
console.log('Tableau tab : ' + tab);
```

Exercice : compter le nombre de lettres cumulé sur le nombre de mots d'un tableau donné.



Example

```
let pets = ['cat', 'dog', 'fish'];  
let nb = pets.map(s => s.length).reduce((a, b) => a + b);  
console.log('nb = ' + nb);
```

Exercice : écrire le code permettant de construire un nouveau tableau obtenu en filtrant toutes les valeurs paires d'un tableau, et en transformant chaque valeur restante en son carré.



Example

```
let tab = t.filter(v => v % 2 == 0).map(v => v * v);  
console.log('Tableau tab : ' + tab);
```

Exercice : compter le nombre de lettres cumulé sur le nombre de mots d'un tableau donné.



Example

```
let pets = ['cat', 'dog', 'fish'];  
let nb = pets.map(s => s.length).reduce((a, b) => a + b);  
console.log('nb = ' + nb);
```

Exercice : écrire le code permettant de construire un nouveau tableau obtenu en filtrant toutes les valeurs paires d'un tableau, et en transformant chaque valeur restante en son carré.



Example

```
let tab = t.filter(v => v % 2 == 0).map(v => v * v);  
console.log('Tableau tab : ' + tab);
```


Functional Programming

C'est la forme de programmation caractérisée par l'enchaînement des appels de fonctions (method chaining).



Example

```
let t = [2, 3, 4].map(v => v * 2).reduce((a,v) => a+v, 0);  
console.log("The total is", t); // 18
```

Nombre de langages intègrent aujourd'hui des mécanismes de programmation fonctionnelle. En Java 8, c'est par le biais de Stream.



Example

```
List<String> list =  
    Arrays.asList("a1", "a2", "b1", "c2", "c1");  
list  
    .stream()  
    .filter(s -> s.startsWith("c"))  
    .map(s -> s.toUpperCase())  
    .sorted()  
    .forEach(s -> System.out.print(s)); // c1c2
```

Maps

Pour construire des maps, depuis ES6 il ne faut plus utiliser les objets mais utiliser l'objet *Map*.



Example

```
let map = new Map();
map.set('toto', 100);
map.set('titi', 50).set('tata', 20);
map.get('titi'); // 50
map.set('titi', 10);
map.get('titi'); // 10
map.has('tutu'); // false
map.has('toto'); // true
map.size; // 3
map.delete('toto');
map.size; // 2
map.clear();
map.size; // 0
```

Remark

Il est possible de passer un tableau de tableaux (avec deux cellules, l'une pour la clé, l'autre pour la valeur) à la construction de la map.

Il est possible d'itérer sur les clés, les valeurs ou les entrées. L'ordre d'insertion est préservé.



Example

```
for (let k of map.keys())  
  console.log(k);  
for (let v of map.values())  
  console.log(v);  
for (let e of map.entries())  
  console.log(e[0] + ' : ' + e[1]);  
for (let [k, v] of map.entries())  
  console.log(k + ' : ' + v);  
for (let [k, v] of map)  
  console.log(k + ' : ' + v);  
  
map.forEach(k => console.log(k));  
map.forEach((k, v) => console.log(k + ' : ' + v));
```

Remark

L'objet *WeakMap* permet de créer des maps avec des clés qui sont nécessairement des objets, et l'impossibilité d'itérer ou de réinitialiser.

Pour construire des sets, depuis ES6 on peut utiliser l'objet *Set*. Lors d'une itération, l'ordre d'insertion est préservé.



Example

```
let set = new Set();
set.add('user');
set.add('admin').add('tech');
set.size; // 3
set.add('admin');
set.size; // 3
set.delete('user');
set; // ['admin', 'tech']

set.forEach(k => console.log(k));
```

Remark

Il existe aussi un objet *WeakSet* avec des valeurs qui sont nécessairement des objets, et l'impossibilité d'itérer ou de réinitialiser.

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Opérateurs classiques

Typiquement, ceux de C, C++ et java:

- ▶ incrémentation/décrémentation (++ , --)
- ▶ arithmétiques (+, -, *, /, %)
- ▶ relationnels (>, <, >=, <=, ==, !=) et (===, !==)
- ▶ logique (!, &&, ||)
- ▶ affectation (=, +=, -=, *=, /=, %=)
- ▶ bit à bit (&, |, <<, >>, ...)



Example

```
let age = 10;  
age++;  
age; // 11  
age > 10 && age < 20; // true  
26 % 5; // 1  
age*=2;  
age; // 22
```

Opérateurs d'égalité

Deux types d'égalité :

- ▶ égalité stricte : deux valeurs sont égales strictement si elles représentent le même objet ou sont de même type primitif et de même valeur. On utilise `===`, et `!==` pour la négation.
- ▶ égalité "souple" : deux valeurs sont égales "souplement" si elles représentent le même objet ou représentent la même valeur après possible conversion de type. On utilise `==`, et `!=`.



Exemple

```
const n = 5, s = "5";  
n === s; // false -- different types  
n !== s; // true  
n === Number(s); // true -- "5" converted to 5  
n !== Number(s); // false  
n == s; // true  
n != s; // false
```

Conseil : toujours comparer des valeurs de même type (et utiliser systématiquement `===` et `!==` pour plus de sécurité).

Comparer des nombres

Entre `Number.MIN_SAFE_INTEGER` et `Number.MAX_SAFE_INTEGER`, tout va bien. Mais observez le comportement des deux codes suivants :



Example

```
let n = 0;
while (true) {
  n+=0.1;
  if (n === 0.3) break;
}
```



Example

```
let n = 0;
while (true) {
  n+=0.1;
  if (Math.abs(n - 0.3) < Number.EPSILON) break;
}
```

Remark

`Number.EPSILON` est une valeur très petite (2.22e-16)

Comportement coupe-circuit

Le comportement coupe-circuit s'observe pour :

- ▶ `b1 || b2` : si `b1` est évalué à vrai, `b2` n'est pas évalué
- ▶ `b1 && b2` : si `b1` est évalué à faux, `b2` n'est pas évalué

Ce comportement est souvent mis à contribution, notamment en testant qu'une valeur est null ou undefined (puisque ce sont des valeurs "falthy").

Par exemple, si `userOptions` est un objet qui peut ne pas avoir été défini préalablement, on peut écrire :



Example

```
let options = {};  
if (userOptions)  
  options = userOptions;
```

ou :



Example

```
let options = userOptions || {};
```

Le type d'un objet

L'opérateur `typeof` est critiqué pour deux choses :

- ▶ `typeof null` retourne `object`
- ▶ `typeof []` retourne `object`

Penser à utiliser :

- ▶ `Array.isArray()`

Destructuration d'affectation

Pour les tableaux, on peut copier les valeurs dans des variables individuelles nommées en une seule instruction.



Example

```
let t = [1, 2, 3];
let [x, y] = t;
x; // 1
y; // 2
z; // error

let t = [1, 2, 3, 4, 5];
let [x, y, ...rest] = t;
x; // 1
y; // 2
rest; // [3, 4, 5]

let x = 5, y = 10;
[x, y] = [y, x];
x; // 10
y; // 5
```

Il est possible d'ignorer des valeurs, et de fournir des valeurs par défaut.



Example

```
let [x, , y] = [1, 2, 3];  
x; // 1  
y; // 3  
  
let [x, , , ...y] = [1, 2, 3, 4, 5, 6];  
x; // 1  
y; // [4, 5, 6]  
  
let [x, y, z = 3] = [1, 2];  
z; // 3  
  
function f([a, b, c = 4] = [1, 2, 3]) {  
  console.log(a, b, c);  
}  
f(); // 1 2 3  
f(undefined); // 1 2 3  
f([10, 20]); // 10 20 4
```

Il y a de nombreux autres usages de l'opérateur ... sur les tableaux.



Example

```
let t1 = [2, 3, 4];
let t2 = [1, ...t1, 5, 6];
console.log(t2); // [1, 2, 3, 4, 5, 6]

let t3 = [1];
t3.push(...t1);
console.log(t3); // [1, 2, 3, 4];

let a1 = [1], a2 = [2];
let a3 = [...a1, ...a2, ...[3, 4]], a4 = [5];

function f(a, b, c, d, e) { return a + b + c + d + e; }
console.log(f(...a3, ...a4)); // 15
```

Destructuration d'objets

Pour les objets, il faut respecter le nom des champs (peu importe l'ordre).



Example

```
let o = { b: 2, c: 3, d: 4 };  
let {a, b, c} = o;  
a; // undefined  
b; // 2  
c; // 3  
d; // erreur
```

Lorsque la destructuration n'est pas effectuée au moment de la déclaration, il faut encadrer avec des parenthèses.



Example

```
let o = { b: 2, c: 3, d: 4 };  
let d, b, c;  
{d, b, c} = o; // erreur  
({d, b, c} = o); // ok  
console.log(d, b, c); // 4 2 3
```

Il est possible d'utiliser des valeurs par défaut et d'autres noms que ceux des propriétés.



Example

```
let {a, b, c = 3} = {a: 1, b: 2};  
console.log(a, b, c); // 1 2 3
```

```
let o = {name: 'toto', age: 20 };  
let {name: x, age: y} = o;  
x; // 'toto'  
y; // 20
```

```
let o = {name: 'toto', other: {age: 20}};  
let {name, other:{age}} = o;  
name; // 'toto'  
age; // 20
```

Destructuration d'arguments

Pour les tableaux, comme d'habitude, il faut prêter attention à l'ordre des arguments.



Example

```
function sentence([subject, verb, object]) {  
  return subject + ' ' + verb + ' ' + object;  
}  
  
let t = ['I', 'love', 'JavaScript'];  
sentence(t); // 'I love JavaScript'
```

En fait, on aurait pu écrire :



Example

```
function sentence(subject, verb, object) {  
  return subject + ' ' + verb + ' ' + object;  
}  
  
let t = ['I', 'love', 'JavaScript'];  
sentence(...t); // 'I love JavaScript'
```


Pour les objets, le nom des champs permet le matching.



Example

```
function sentence({subject, verb, object}) {  
  return subject + ' ' + verb + ' ' + object;  
}  
  
let o = {  
  verb: 'love',  
  object: 'JavaScript',  
  subject: 'I'  
};  
sentence(o);  
  
function f({name = 'toto', age = 20, profession = 'pilot'}  
  = {}) {  
  console.log(name, age, profession);  
}  
f({name: 'titi', age: 30}); // 'titi 30 pilot'  
f(); // 'toto 20 pilot'  
f(undefined); // 'toto 20 pilot'
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Création directe d'un objet

Notation littérale à privilégier :

- ▶ symboles '{' et '}' encadrant la définition de l'objet
- ▶ introduction de champs (propriétés) et fonctions (méthodes), constitués d'un nom suivi de ':' et de la valeur
- ▶ symbole ',' comme séparateur entre les différents champs/fonctions



Example

```
let person = {  
  name: 'Alice',  
  age: 20,  
  job: 'singer',  
  
  introduction() {  
    return this.name + ' ' + this.age + ' ' + this.job;  
  }  
};
```

Remark

Pour les méthodes, on privilégiera la notation concise ('method definition' comme ci-dessus).

Accès aux membres d'un objet

Pour accéder à tout membre (champ ou fonction) d'une variable de type *object*, on utilise généralement la notation pointée :

nom de la variable suivi de '.' suivi du nom du membre

Toutefois, il est également possible d'utiliser les crochets pour accéder aux membres.



Example

```
person.name;      // Alice
person['name'];   // Alice
let field='name';
person[field];    // Alice

person.introduction(); // Alice 20 singer
person['introduction'](); // Alice 20 singer
```

Il est important de noter qu'on utilise :

- ▶ *this* si on se trouve dans le code d'une fonction de l'objet
- ▶ le nom de la variable (objet) si on se trouve dans du code par ailleurs



Example

```
let game = {
  finished: false,
  ...
  isFinished() {
    return this.finished;
  },
  getPieceAt(row, col) {
    ...
  }
};

if (game.isFinished()) {
  ...
}
```

Notation concise des méthodes

Cela revient à supprimer : `function`. L'un des intérêts est de pouvoir utiliser *super*.



Example

```
let player = {  
  _name: 'toto',  
  get name() { return this._name; },  
  set name(s) { this._name=s; },  
  start() { console.log('start'); },  
  stop() { console.log('stop') }  
}  
  
player.name; // 'toto'  
player.start(); // 'start'  
player.name='titi';  
player.name; // 'titi'  
player.stop(); // 'stop'
```

Remark

Les mots-clés *get* et *set* placés devant des noms de méthodes permettent de définir des fonctions simulant et contrôlant l'accès à un champ.

Parcourir un objet

On peut utiliser **for in**, mais il faut généralement prêter attention aux propriétés héritées.



Example

```
let o = { apple: 1, orange: 2, apricot: 3, banana: 4 };
for (let p in o)
  if (o.hasOwnProperty(p))
    console.log(p + ': ' + o[p]);
```

On utilise plutôt `Object.keys()` :



Example

```
Object.keys(o)
  .filter(p => p.match(/^a/))
  .forEach(p => console.log(p + ': ' + o[p]));
// apple : 1
// apricot : 3
```

Objets dynamiques

Il est possible de :

- ▶ modifier dynamiquement la structure d'un objet en ajoutant ou retirant (delete) des propriétés (champs ou méthodes).
- ▶ modifier la valeur d'un champ, et aussi d'une méthode (dangereux)



Example

```
let obj = {}; // objet vide
obj.size=3;
obj.color='yellow';
obj; // { size: 3, color: 'yellow' },
obj.hello = function() { return 'hello'; }
obj.hello(); // hello
obj.hello = function() { return 'goodbye'; }
obj.hello(); // goodbye
delete obj.hello;
obj.hello(); // erreur
obj.address = { street: 'rue de la paix', city: 'Paris' };
obj.address.city; // Paris
obj['address']['city']; // Paris
```


Créer une classe

Dans une classe, on peut placer un constructeur (un seul !) et des méthodes (on n'utilise pas le mot-clé *function*).



Exemple

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  atNorthEast() {  
    return this.x >= 0 && this.y >= 0;  
  }  
}  
  
let a = new Point(3, -1);  
let b = new Point(2, 8);  
a.atNorthEast(); // false  
b.atNorthEast(); // true  
  
a instanceof Car;    // true  
b instanceof Array;  // false
```

Méthodes statiques

Ce sont des méthodes qui ne concernent pas des instance spécifiques mais la classe.



Example

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  static distance(a, b) {  
    const dx = a.x - b.x;  
    const dy = a.y - b.y;  
    return Math.hypot(dx, dy);  
  }  
}  
  
let p1 = new Point(5, 5);  
let p2 = new Point(10, 10);  
Point.distance(p1, p2);
```

Il faut définir une propriété au niveau de la classe.



Example

```
class Car {  
  constructor(make) {  
    this.make=make;  
    this.id=Car.id++;  
  }  
}  
Car.id = 0;  
let c1 = new Car('Peugeot');  
let c2 = new Car('Renault');  
  
c1.id; // 0  
c2.id; // 1
```

Prototype

Toutes les classes ont un prototype. Par exemple `Point.prototype`.
Une méthode appartient à un prototype. Par exemple
`Point.prototype.atNorthEast`.

- ▶ Lorsqu'un objet est créé avec *new*, le prototype de la classe est copié dans le champ `__proto__` de l'objet.
- ▶ Lorsqu'une méthode est appelée, une recherche est effectuée au niveau de l'objet, puis du prototype de l'objet, puis du prototype du prototype de l'objet, etc.



Example

```
p1.atNorthEast === Point.prototype.atNorthEast; // true
p1.atNorthEast === p2.p1.atNorthEast; // true

p1.atNorthEast = () => true;
p1.atNorthEast === Point.prototype.atNorthEast; // false
p1.atNorthEast === p2.p1.atNorthEast; // false
```

Il faut utiliser le mot-clé *extends* et appeler le super-constructeur si on insère un constructeur dans la sous-classe.



Example

```
class Vehicle {
  constructor() {
    this.passengers = [];
  }

  addPassenger(p) {
    this.passengers.push(p);
  }
}

class Car extends vehicle {
  constructor() {
    super();
  }

  deployAirbags() { console.log('bwoosh'); }
}
```



Example

```
let v = new vehicle();  
v.addPassenger('Jo');  
v.addPassenger('Alice');  
v.passengers; // ['Jo', 'Alice']  
let c = new Car();  
c.addPassenger('John');  
c.addPassenger('Lucie');  
c.passengers; // ['John', 'Lucie']  
v.deployAirbags(); // erreur  
c.deployAirbags(); // 'bwoosh'
```



Example

```
class Sup {  
  constructor() {  
    this.name = 'sup';  
    this.isSup = true;  
  }  
  
  f() { return true; }  
}  
  
Sup.prototype.sneaky = 'not recommended';  
  
class Sub extends Sup {  
  constructor() {  
    super();  
    this.name = 'sub';  
    this.isSub = true;  
  }  
}
```



Example

```
let o = new Sub();
for (let p in o)
  console.log(p + ' : ' + o[p]
    + (o.hasOwnProperty(p) ? '' : ' (inherited)'));

// name : sub
// isSup : true
// isSub : true
// sneaky : not recommended (inherited)
```

Remark

A noter que les champs déclarés dans la super-classe le sont aussi dans la sous-classe.

Héritage multiple

Il faut créer une fonction mixin et ensuite l'appeler sur le prototype d'une classe.



Example

```
class InsurancePolicy { ... }

function makeInsurable(o) {
  o.addPolicy = function(p) { this.policy = p; }
  p.getPolicy = function() { return this.policy; }
  o.isInsured = function(p) { return !!this.policy; }
}

makeInsurable(Car.prototype);
let car = new Car();
car.addPolicy(new Policy());
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Fonction *setTimeout*

La fonction *setTimeout* permet d'exécuter une fonction donnée après un laps de temps donné. Il faut bien comprendre l'aspect asynchrone : l'exécution du thread principal ne se place pas en attente.



Example

```
console.log('avant');  
setTimeout(function() {  
    console.log('après 1 mn');  
}, 60*1000);  
console.log('après');
```

Le résultat est :

```
avant  
après  
après 1 mn
```

Fonction *setInterval* et *clearInterval*

- ▶ la fonction *setInterval* exécute indéfiniment la méthode donnée à un intervalle de temps donné
- ▶ la fonction *clearInterval* permet de stopper la fonction *setInterval*



Example

```
const before = new Date();
const id = setInterval(function() {
  let now = new Date();
  if (now.getMinutes() !== before.getMinutes()) {
    console.log('stopped');
    return clearInterval(id);
  }
  console.log('again');
}, 5*1000); // 5 secondes
```

Portée des variables

Il faut être très vigilant quant à la portée des variables en présence d'instructions asynchrones.

Tester les 4 versions suivantes :



Example

```
let i;
for (i=5; i>=0; i--)
  setTimeout(() => console.log(i), (5-i)*1000);

for (let i=5; i>=0; i--)
  setTimeout(() => console.log(i), (5-i)*1000);

for (var i=5; i>=0; i--)
  setTimeout(() => console.log(i), (5-i)*1000);

function aux(i) {
  setTimeout(() => console.log(i), (5-i)*1000);
}
for (var i=5; i>=0; i--)
  aux(i);
```

Error-first Callbacks

Mécanisme pour communiquer un échec. Principe: le premier argument du callback représente un objet erreur.



Example

```
let fs = require('fs');

let f = 'toto.txt';
fs.readFile(f, function(err, data) {
  if (err)
    return console.error('Error : ' + err.message);
  // ...
});
```

Remark

La valeur *undefined* est retournée par *console.error*.

Remark

Error-first callbacks sont encore très utilisés (par exemple, en développement Node.js).

Capturer une exception ?

Le code suivant n'est pas correct car:

- ▶ le *try catch* est placé dans *readSketchyFile*
- ▶ alors que l'exception est lancée dans le corps de la fonction callback



Example

```
let fs = require('fs');
function readSketchyFile() {
  try {
    fs.readFile('toto.txt', function(err, data) {
      if (err) throw err;
    });
  } catch (err) {
    console.log('pb with the file');
  }
}
readSketchyFile();
```

Warning

Il n'est pas possible d'attraper des exceptions lancées par une fonction callback.

Promises

Les promesses règlent nombre de problèmes des callbacks. Lorsqu'on réalise un traitement de façon asynchrone à l'aide d'une promesse, on récupère un objet *Promise* qui indique :

- ▶ soit un succès (*fulfilled*)
- ▶ soit un échec (*rejected*)



Example

```
const p = new Promise((resolve, reject) => {  
  // réaliser une tâche asynchrone et appeler :  
  // soit resolve(valeur); // si promesse tenue  
  // soit reject(raison); // si promesse rompue  
});
```

Warning

Une fois qu'une instruction *resolve* ou *reject* est exécutée, la promesse est établie (*settled*), i.e., ne pourra plus être remise en cause, même si le code asynchrone continue éventuellement de s'exécuter.

On attache des callbacks à une promesse comme suit :



Example

```
// une seule fonction callback (succès)
p.then(
  function(valeur) {
    // promesse tenue
  }
);

// une seule fonction callback (échec)
p.catch(
  function(raison) {
    // promesse rompue
  }
);

// deux fonctions callback
p.then(
  function(valeur) {
    // promesse tenue
  },
  function(raison) {
    // promesse rompue
  }
);
```

Un exemple où le résultat semble conforme à ce qui est attendu :



Example

```
function countdown(seconds) {  
  return new Promise(function(resolve, reject) {  
    for (let i=seconds; i>=0; i--) {  
      setTimeout(function() {  
        if (i == 13) reject('Bad luck');  
        else if (i == 0) resolve('YES');  
        else console.log('Step : ' + i);  
      }, (seconds-i)*1000);  
    }  
  });  
}  
  
countdown(5).then(  
  function(messageSuccess) {  
    console.log(messageSuccess);  
  },  
  function(messageFailure) {  
    console.log(messageFailure);  
  }  
);
```

Un exemple où le résultat ne semble pas conforme à ce qui est attendu.
La promesse est rompue mais le code continue de s'exécuter.



Example

```
const p = countdown(15);  
p.then(  
  function(messageSuccess) {  
    console.log(messageSuccess);  
  },  
  function(messageFailure) {  
    console.log(messageFailure);  
  }  
);
```

Le résultat est :

```
Step 15  
Step 14  
Bad luck  
Step 12  
Step 11  
...
```

Callback Hell

L'enchaînement d'appels de fonctions callback devient très vite illisible :

```
faireTruc1(function(result1) {  
  faireTruc2(result1, function(result2) {  
    faireTruc3(result2, function(result3) {  
      console.log('Résultat final :' + result3);  
    }, failureCallback);  
  }, failureCallback);  
}, failureCallback);
```



Example

```
fs.readFile('a.txt', function(err, dataA) {  
  if (err) ...  
  fs.readFile('b.txt', function(err, dataB) {  
    if (err) ...  
    fs.readFile('c.txt', function(err, dataC) {  
      if (err) ...  
      ...  
    }  
  }  
});
```

Avec une chaîne de promesses, le code est beaucoup plus lisible.

```
faireTruc1()  
.then(result1 => faireTruc2(result1))  
.then(result2 => faireTruc3(result2))  
.then(result3 => console.log('Res. final : ' + result3))  
.catch(failureCallback);
```

Tester le code suivant :



Example

```
new Promise((resolve, reject) => {  
  console.log('Initial');  
  resolve();  
})  
.then(() => {  
  throw new Error('Something failed');  
  // line above, similar to reject('Something failed');  
  console.log('Do this');  
})  
.catch(() => console.log('Do that'))  
.then(() => console.log('Do this whatever happened'));
```

Illustration avec EventEmitter

Node.js fournit une classe *EventEmitter*.



Example

```
'use strict';  
  
class Countdown extends require('events').EventEmitter {  
  constructor(seconds) {  
    super();  
    this.seconds=seconds;  
  }  
  
  go() {  
    const that = this;  
    return new Promise(function(resolve,reject) {  
      for (let i=that.seconds; i >= 0; i--) {  
        setTimeout(function() {  
          if (i === 13) reject('Bad luck');  
          else if (i === 0) resolve('Go');  
          else that.emit('tick', i);  
        }, (that.seconds-i)*1000);  
      }  
    });  
  }  
}
```

Lorsqu'on exécute :



Example

```
const c = new Countdown(3)
  .on('tick', function(i) {
    if (i > 0) console.log('Step : ' + i);
  });

c.go()
  .then(function(message) {
    console.log(message);
  })
  .catch(function(raison) {
    console.log('Erreur : ' + raison);
  });
```

On obtient :

```
Step 3
Step 2
Step 1
Go
```

Si on exécute le même code avec `CountDown(15)` à la place de `CountDown(3)`, on obtient :

```
Step 15
Step 14
Erreur : Bad Luck
Step 12
...
Step 1
```

Sur cet exemple, pour stopper l'exécution proprement, il faut garder les ids des fonctions en timeout.



Example

```
const timeoutIds = [];
...
timeoutIds.push(setTimeout(function() {
  ...
  if (i === 13) {
    timeoutIds.forEach(clearTimeout);
    reject('Bad luck');
  }
}
```




Example

```
function ajax() {  
  return new Promise(function(resolve, reject) {  
    let r = new XMLHttpRequest();  
    r.open("GET", "https://mdn.github.io/learning-area/  
      javascript/oops/json/superheroes.json");  
    r.addEventListener("load", function() {  
      if (r.status === 200)  
        resolve(r.responseText);  
      else  
        reject("Server Error : " + r.status);  
    }, false);  
    r.addEventListener("error", function() {  
      reject("Cannot make AJAX Request");  
    }, false);  
    r.send();  
  });  
}
```



Example

```
ajax().then(function(value) {  
    return JSON.parse(value);  
}).then(function(value) {  
    console.log(value.squadName);  
    return value;  
}).catch(function(reason) {  
    console.log(reason);  
});
```

On peut construire une promesse tenue comme ceci :



Example

```
Promise.resolve({name: 'toto'}).then(function(value) {  
  console.log(value.name);  
});
```

On peut construire une promesse rompue comme ceci :



Example

```
Promise.reject({name: 'toto'}).catch(function(value) {  
  console.log(value.name);  
});
```

Methodes de *Promise*

On peut construire une promesse aboutie lorsque les promesses d'une collection sont toutes abouties, ou l'une d'entre elles est rompue.



Example

```
let p1 = new Promise(function(resolve, reject) {
  setTimeout(function() { resolve('Me'); }, 1000);
});
let p2 = new Promise(function(resolve, reject) {
  setTimeout(function() { resolve('You'); }, 2000);
});
Promise.all([p1, p2]).then(function(value) {
  console.log(value); // ['Me', 'You'] after 2 seconds
});
```

Si dans *p1*, on remplace `resolve()` par `reject()` :



Example

```
Promise.all([p1, p2]).catch(function(reason) {
  console.log(reason); // 'Me' after 1 second
});
```

On peut construire une promesse aboutie dès qu'une promesse parmi une collection est aboutie.



Example

```
let p1 = new Promise(function(resolve, reject) {
  setTimeout(function() { resolve('Me'); }, 1000);
});

let p2 = new Promise(function(resolve, reject) {
  setTimeout(function() { resolve('You'); }, 2000);
});

Promise.race([p1, p2]).then(function(value) {
  console.log(value); // 'Me' after 1 second
}, function(reason) {
  console.log(reason);
});
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Un itérateur est un objet permettant d'accéder aux éléments d'une collection un par un (à l'aide d'un curseur interne). Un itérateur définit une méthode *next()* qui retourne un objet représentant l'élément suivant dans la séquence. Cet objet de la forme `{ done: x, value: y }` contient donc :

- ▶ un champ booléen *done* de valeur *true* quand on est arrivé à la fin de l'itération,
- ▶ un champ *value* de valeur *undefined* quand on est arrivé à la fin de l'itération.

Remark

Une fois créé, un itérateur peut être utilisé :

- ▶ explicitement en appelant sa méthode *next()*
- ▶ ou implicitement en utilisant une boucle *for of*

Itérateurs sur les tableaux

Sur un tableau, la méthode *values()* retourne un itérateur. Mais pour des raisons techniques, elle n'est pas reconnue par Chrome et Firefox.

Pour construire un itérateur (sur un tableau), il faut retourner un objet équipé de la méthode *next()* :



Example

```
function iteratorOn(array) {  
  let cursor = 0;  
  return {  
    next: function() {  
      if (cursor >= array.length)  
        return { done: true };  
      return { value: array[cursor++], done: false };  
    }  
  }  
}
```




Example

```
let t = ['t', 'i', 't', 'o', 'u'];

let it = iteratorOn(t); // idéalement t.values()
it.next(); // { value: 't', done: false }
it.next().value; // i
it.next().value; // t
it.next().value; // o
it.next().value; // u
it.next().done; // true

for (let lettre of t) // itérateur implicite
  console.log(lettre);

let it2 = iteratorOn(t);
let curr = it2.next();
while (!curr.done) {
  console.log(curr.value);
  curr = it2.next();
}
```

Itérateur sur les classes

Pour rendre un objet itérable, il faut intégrer une fonction de nom *Symbol.iterator* retournant un itérateur, c'est-à-dire, un objet doté de la méthode *next()*.



Example

```
class Log {  
  constructor() {  
    this.messages = [];  
  }  
  
  add(message) {  
    this.messages.push({message, timestamp: Date.now()});  
  }  
  
  [Symbol.iterator]() {  
    return iteratorOn(this.messages);  
  }  
}
```

Remark

A noter le raccourci *message* pour *message: message*



Example

```
let log = new Log();
log.add('breakfast');
log.add('lunch');
log.add('diner');

for (let {message, timestamp} of log)
  console.log(new Date(timestamp) + ' : ' + message);
```

Il est possible de construire une classe permettant une itération infinie.



Example

```
class FibonacciSequence {
  [Symbol.iterator]() {
    let [prev, curr] = [0, 1];
    return {
      next() {
        [prev, curr] = [curr, prev + curr];
        return { value: prev, done: false };
      }
    };
  }
}
```



Example

```
let fib = new FibonacciSequence();  
let i = 0;  
for (let v of fib) {  
    console.log(n);  
    if (++i > 6) break;  
}
```

On obtient :

```
1  
2  
3  
5  
8  
13  
21
```

Un générateur est comme une fonction ordinaire sauf que :

- ▶ un astérisque suit le nom de la fonction
- ▶ le générateur retourne implicitement un itérateur à l'appel. Il est alors possible de solliciter le générateur/itérateur avec la méthode *next()*
- ▶ le générateur peut rendre la main, *yield*, à n'importe quel moment (pour attendre le prochain appel *next*)



Example

```
function* rainbow() {  
  yield 'red';  
  yield 'orange';  
  yield 'yellow';  
  yield 'green';  
  yield 'blue';  
  yield 'indigo';  
  yield 'violet';  
}
```



Example

```
let r = rainbow();
r.next(); // { value: 'red', done: false }
r.next().value; // 'orange'
r.next().value; // 'yellow'
r.next().value; // 'green'

for (let color of rainbow())
  console.log(color);
```



Example

```
function* fibonacci() {
  let [prev, curr] = [0, 1];
  while (true) {
    [prev, curr] = [curr, prev + curr];
    yield curr;
  }
}

for (let v of fibonacci()) {
  if (v > 100) break;
  console.log(v);
}
```

Communiquer avec le générateur

L'instruction *yield* retourne une expression qui représente le paramètre passé au moment de l'appel à *next()*, si il est présent.



Example

```
function* interrogate() {  
  let name = yield 'your name?';  
  let color = yield 'your favorite color?';  
  return `favorite color of ${name}: ${color}`;  
}  
  
let i = interrogate();  
i.next(); // { value: 'your name?', done: false };  
i.next('Alice').value; // 'your favorite color?'  
i.next('orange'); // { value: 'favorite color of Alice:  
  orange', done: true }
```

Terminaison de générateurs

Si on exécute *return* à un moment donné dans le générateur, alors :

- ▶ le champ *done* sera *true*
- ▶ le champ *value* sera la valeur retournée (peu recommandé)



Example

```
function* test() {  
  yield 'a';  
  yield 'b';  
  return;  
  yield 'c';  
}  
  
for (let t of test())  
  console.log(t); // 'a', puis 'b'
```


Générateurs 'récuratifs'

Il est possible de solliciter un objet itérable (par exemple, un générateur secondaire) avec *yield**.



Example

```
function* gaux() {  
  yield 2;  
  yield 3;  
}  
  
function* g() {  
  yield 1;  
  yield* gaux();  
  yield* [4, 5];  
}  
  
let g = g();  
g.next().value; // 1  
g.next().value; // 2  
g.next().value; // 3  
g.next().value; // 4  
g.next().value; // 5  
g.next().done;  // true
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Construction

Il y a deux façons de construire une expression régulière (regex) :

- ▶ en utilisant une description littérale entourée du caractère '/'
- ▶ en utilisant le constructeur de *RegExp*

Par exemple, si on souhaite rechercher une occurrence de 'toto', on pourra écrire indifféremment :



Example

```
const r1 = /toto/;  
const r2 = new RegExp('toto');
```

Deux indicateurs sont très souvent employés :

- ▶ g : recherche globale ; retrouve toutes les correspondances plutôt que la première.
- ▶ i : la casse est ignorée.



Example

```
const r1 = /toto/ig;  
const r2 = new RegExp('toto', 'ig');
```

Utiliser les regex

On utilisera les fonctions *match*, *test* et *replace* le plus souvent.



Example

```
const input = 'As I was going to Saint Ives';
const r = /\w{3,}/ig;

input.match(r); // ["was", "going", "Saint", "Ives"]
input.match(/\w{3,}/ig); // écriture équivalente

r.test(input); // true
/\w{3,}/ig.test(input); // écriture équivalente

input.replace(/\w{4,}/ig, '****'); // "As I was **** to
**** ****"

input.search(r); // 5 (première occurrence démarre à 5)
```

Remark

La fonction *exec* permet de passer en revue les correspondances.

Méta-caractères de répétition

- ▶ x^* : x répété 0 ou plusieurs fois
- ▶ x^+ : x répété 1 ou plusieurs fois
- ▶ $x^?$: x répété 0 ou 1 fois



Example

```
'Un booléen'.match(/bo*/g); // ["boo"]  
'Un bateau bleu'.match(/bo*/g); // ["b", "b"]  
'Ce matin'.match(/bo*/g); // null  
  
'maison'.match(/o+/g); // ["o"]  
'boom'.match(/o+/g); // ["oo"]  
  
'gel'.match(/e?le?/g); // ["el"]  
'angle'.match(/e?le?/g); // ["le"]  
'oslo'.match(/e?le?/g); // ["l"]
```

Warning

Le comportement par défaut, “gourmand”, fait correspondre le plus de caractères possible.

- ▶ $x\{n\}$: x répété n fois
- ▶ $x\{n, \}$: x répété n fois ou plus
- ▶ $x\{n, m\}$: x répété au moins n fois et au plus m fois
- ▶ $x|y$: correspond à x ou y
- ▶ $[xyz]$: n'importe quel élément présent entre les crochets. Possibilité d'utiliser '-' pour un intervalle. A l'intérieur des crochets, les caractères '.' et '^' n'ont pas besoin d'être échappés.
- ▶ $[\^xyz]$: n'importe quel élément non présent entre les crochets. Possibilité d'utiliser '-' pour un intervalle. Les caractères '.' et '^' n'ont pas besoin d'être échappés.



Example

```
'Mozillaaa'.match(/a{2}/g); // ["aa"]  
'aaaa'.match(/a{2,}/g); // ["aaaa"]  
'feu vert'.match('vert|rouge'); // ["vert"]  
'10 carottes et 5 navets'.match(/[0-9]+/g); // ["10", "5"]
```

Ensembles de caractères

.	n'importe quel caractère (excepté caractère de saut de ligne)
\d	un chiffre (digit). Equivalent à [0-9].
\D	tout caractère qui n'est pas un chiffre. Equivalent à [^0-9].
\s	tout caractère blanc (espace, tabulation, saut de ligne, ...)
\S	tout caractère non blanc.
\w	tout caractère alphanumérique (word character). Equivalent à [A-Za-z0-9_].
\W	tout caractère non alphanumérique
\b	correspond à une limite de mot (word boundary).
\B	correspond à une "non-limite" de mot.
^	correspond au début de séquence
\$	correspond à la fin de séquence

Remark

Une limite de mot correspond à la position où un caractère de mot n'est pas suivi ou précédé d'un autre caractère de mot.



Example

```
const messyPhone = '(505) 555-1515';  
const neatPhone = messyPhone.replace(/\D/g, '');  
  
const valid = /\S/.test(x); // tester si au moins un  
    caractère non blanc  
  
const text = 'it was nice and it was beautiful';  
text.match(/^\w+/g); // ["it"]  
text.match(/\w+$/g); // ["beautiful"]
```

Avec l'indicateur m, on peut travailler sur plusieurs lignes.



Example

```
const text = 'One line\nTwo lines\nThree lines';  
text.match(/^\w+/mg); // ["One", "Two", "Three"]  
text.match(/\w+$/mg); // ["line", "lines", "lines"]
```

Remark

Pour matcher tout caractère (y compris les sauts de ligne), on pourra utiliser `[\s\S]`.

Groupes

Il y a deux formes de groupes :

- ▶ (x) correspond à x et garde la correspondance en mémoire.
- ▶ (?x) correspond à x et ne garde pas la correspondance en mémoire.



Example

```
// se termine par .com ou .org ou .edu ?  
text.match(/[a-z]+\.(?:com|org|edu)/i);  
  
// commence par http://, https:// ou //  
text.match(/(?:https?)?\:\/\/[a-z][a-z0-9-]+[a-z0-9]+/ig);
```

Les groupes 'capturants' peuvent être réexploités en utilisant :

- ▶ \1, \2, ... dans l'expression régulière,
- ▶ %1, %2, ... dans une expression de remplacement.



Example

```
let text = 'lucie et anna';  
text.match(/([a-z])([a-z])\2\1/ig); // ['anna']
```

Greedy versus Lazy

Par défaut, le comportement est gourmand (greedy).



Example

```
let text = '<i>greedy</i> or <i>lazy</i>';  
text = text.replace(/<i>(.)</i>/ig, '<b>$1</b>');  
text; // '<b>greedy</b> or <i>lazy</b>'
```

Pour passer en mode paresseux (lazy), on peut faire suivre '*' de '?'.



Example

```
let text = '<i>greedy</i> or <i>lazy</i>';  
text = text.replace(/<i>(.*?)</i>/ig, '<b>$1</b>');  
text; // '<b>greedy</b> or <b>lazy</b>'
```

Remark

Tous les méta-caractères de répétition *, +, ? {*n*}, {*n*,} et {*n*, *m*} peuvent être suivis de ? pour les rendre paresseux.



Example

```
let text = "<a id='toto' href='http://toto'> Toto </a>";  
text = text.replace(/<a .*?(href='.*?').*?>/, '<a $1>');  
text; // '<a href='http://toto'> Toto </a>'
```

En plus de \$1, \$2, ... on peut utiliser :

- ▶ '\$' pour représenter tout ce qui précède la correspondance
- ▶ '\$&' pour représenter la correspondance
- ▶ '\$' pour représenter tout ce qui succède à la correspondance



Example

```
let text = 'One two three';  
text.replace(/two/, '($')'); // 'One (One ) three'  
text.replace(/\w+/g, '($&)'); // '(One) (two) (three)'  
text.replace(/two/, "($')"); // 'One ( three) three'
```



Example

```
let text = "<a id='toto' href='http://toto'> Toto </a>";
text = text.replace(/<a .*?(href='.*?') .*?>/,
  function(match, g1, offset) {
    console.log(match); // <a id='toto'
                        href='http://toto'>
    console.log(g1);    // href='http://toto'
    console.log(offset); // 0
    return '<a ${g1}>';
  }
);
text; // '<a href='http://toto'> Toto </a>'
```

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

L'objet `window` représente la fenêtre du navigateur.



Example

```
console.log(this === window); // true in the browser

// below, three equivalent statements
console.log(navigator.userAgent);
console.log(this.navigator.userAgent);
console.log(window.navigator.userAgent);

let w = window.open("http://www.wrox.com", "wrox");
if (w == null)
    console.log("fenetre bloquee");
else {
    ...
    w.close();
}
```

Warning

Le navigateur peut être configuré de manière à empêcher de modifier son emplacement, de modifier sa taille ou encore d'afficher une fenêtre

Des boîtes de dialogues peuvent être ouvertes en utilisant les méthodes `alert`, `confirm` et `prompt`.



Example

```
if (confirm("Are you sure?"))  
  alert("I am glad");  
else  
  alert("I am sorry");  
  
let name = prompt("What is your name?", "Toto");  
if (name != null)  
  alert("Welcome " + name);
```



En plus de `window`, il est possible d'utiliser les objets du BOM (Browser Object Model) tels que `location`, `navigator`, `screen` et `history`.



Example

```
location.href="http://www.wrox.com";
location.reload();
location.port=8080;

console.log(navigator.appName); // nom du navigateur
console.log(navigator.javaEnabled);
console.log(screen.colorDepth);
console.log(screen.width);
window.resizeTo(screen.availWidth,screen.availHeight);

history.go(2); // go forward 2 pages
history.back(); // go back one page
if (history.length === 0)
    console.log("this is the first page");
```


DOM (Document Object Model) est une API pour manipuler les documents HTML et XML.

L'objet `document` représente la page HTML chargée par le navigateur.

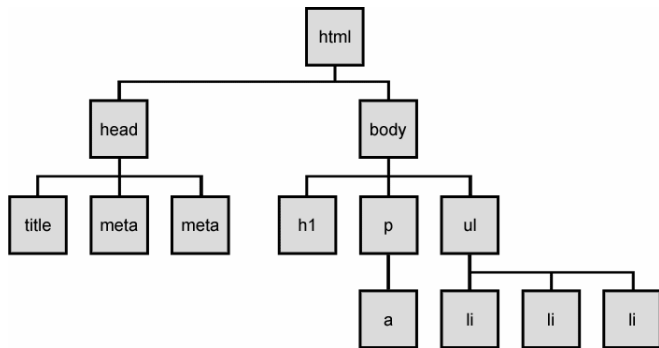


Example

```
document.childNodes[0]; // <!DOCTYPE html>
document.childNodes[1]; // <html ...>
document.childNodes[1].childNodes[0] === document.head;
document.childNodes[1].childNodes[1] === document.body;

document.title = "nouveau titre";
let url = document.URL;
let domain = document.domain;
console.log(url + " " + domain);
```

DOM Tree



Plusieurs méthodes permettent de récupérer des éléments HTML :

- ▶ *getElementById*
- ▶ *getElementsByClassName*
- ▶ *getElementsByTagName*



Example

```
<body>
  <p id="p1"> Hello ! </p>
  <p id="p2"> Bye ! </p>

  <script>
    let p = document.getElementById("p1");
    p.style.color="blue";
    document.getElementById("p2").style.color="red";
  </script>
</body>
```

Remark

Les méthodes DOM retournant une collection le font avec un objet `HTMLCollection`. Pour le convertir en tableau, on peut écrire par exemple `[...document.getElementsByTagName('p')]`



Example

```
<body>
  <ol>
    <li>HTML</li>
    <li>CSS</li>
  </ol>

  <script>
    let list = document.getElementsByTagName("li");
    for (let i=0; i<list.length; i++)
      list[i].style.color="green";

    for (let elt of document.getElementsByTagName("li"))
      elt.style.color="red";

    [...document.getElementsByTagName("li")].forEach(
      elt => elt.style.color="yellow");
  </script>
</body>
```

Query Selectors

Deux méthodes permettent d'utiliser les sélecteurs CSS :

- ▶ *querySelector* : returns the first element that matches a specified CSS selector(s) in the document
- ▶ *querySelectorAll* : returns all elements in the document that match a specified CSS selector(s)



Example

```
document.querySelector("h2, h3").style.color = "red";  
  
for (let e of document.querySelectorAll("div.example p"))  
    e.style.backgroundColor = "red";
```



En utilisant cette propriété, il est possible de modifier le contenu d'un élément.



Example

```
let div1 = document.getElementById("d1");  
let div2 = document.getElementById("d2");  
let div3 = document.getElementById("d3");  
  
div2.innerHTML=div1.innerHTML;  
div3.innerHTML="je suis <strong> content </strong>";
```

Remark

La propriété `innerText` (`textContent` pour Firefox) est similaire à `innerHTML`, mais ne traite que du texte simple.

Warning

Éviter d'utiliser `outerText` et `outerHTML`.



Il existe de nombreuses méthodes DOM pour créer dynamiquement des éléments.



Example

```
let table = document.createElement("table");
for (let i=1; i<=10; i++) {
  let row = document.createElement("tr");
  for (let j=1; j<=10; j++) {
    let cell = document.createElement("td");
    cell.appendChild(document.createTextNode(i*j));
    row.appendChild(cell);
  }
  table.appendChild(row);
}
document.getElementById("d1").appendChild(table);
```

Modifier le style dynamiquement

Tout élément HTML dispose d'un attribut `style` en JS. Les noms des propriétés CSS doivent être convertis en camel case. Par exemple :

Propriété CSS	Propriété JavaScript
background-image	style.backgroundImage
color	style.color
font-family	style.fontFamily



Example

```
let div = document.createElement("myDiv");  
myDiv.style.backgroundColor="red";  
myDiv.style.width="100px";  
myDiv.style.border="1px solid border";
```

Warning

La propriété `float` correspond à un mot réservé de Javascript. Il faut alors utiliser `cssFloat` (ou `styleFloat` pour IE).

Un événement est provoqué par une action de l'utilisateur ou du navigateur lui-même. Les événements ont des noms tels que `click`, `load` et `mouseover`. Une fonction appelée en réponse à un événement se nomme un écouteur (event handler ou event listener). Souvent, leur nom commence par `on` comme par exemple `onclick` ou `onload`.

Associer des écouteurs aux événements possibles peut se faire de trois manières différentes:

- ▶ HTML
- ▶ DOM Level 0
- ▶ DOM Level 2

HTML Event Handlers

On utilise des attributs HTML pour déclarer les écouteurs. La valeur de ces attributs est le code JavaScript à exécuter lorsque l'événement est produit.



Example

```
<body>
  <input type="button" onclick="console.log('clicked') ">
  <input type="button" onclick="showMessage() ">

  <script>
    function showMessage() {
      console.log("hello world");
    }
  </script>
</body>
```

Remark

Il est préférable d'éviter cette approche car elle mélange code HTML et code JS.

On utilise les propriétés des éléments pour leur associer des écouteurs.



Example

```
<body>
  <input type="button" id="but1" value="click">
  <input type="button" id="but2" value="click">

  <script>
    function but1Listener() {
      console.log("but1 cliqued");
    }
    let but1 = document.getElementById("but1");
    but1.onclick=but1Listener;

    document.getElementById("but2").onclick = function() {
      console.log("but2 cliqued");
    }
  </script>
</body>
```

Quand un événement se produit, toutes les informations le concernant sont enregistrées dans un objet. Il est possible de récupérer cet objet comme paramètre d'une fonction écouteur.



Example

```
function listener(evt) {  
  switch(evt.type) {  
    case "click" :  
      console.log(evt.clientX + " " + evt.clientY);  
      break;  
    case "mouseover" :  
      this.style.backgroundColor="red";  
      break;  
    case "mouseout" :  
      this.style.backgroundColor="";  
  }  
}  
  
let but3 = document.getElementById("but3");  
but3.onclick=listener;  
but3.onmouseover=listener;  
but3.onmouseout=listener;
```

Dom Level 2 Event Handlers

On utilise la méthode *addEventListener* qui permet d'associer des écouteurs à un élément cible. La cible peut être un nœud dans un document, le document lui-même, un élément window ou un objet XMLHttpRequest.

La méthode prend deux ou trois paramètres :

- ▶ le type de l'événement, tel que 'click', 'mousedown', 'mouseup', ...
- ▶ l'écouteur, un objet implémentant EventListener, ou une fonction,
- ▶ un booléen optionnel indiquant si l'écouteur doit être exécuté pendant la phase de capture ou de bullage (false, valeur par défaut).

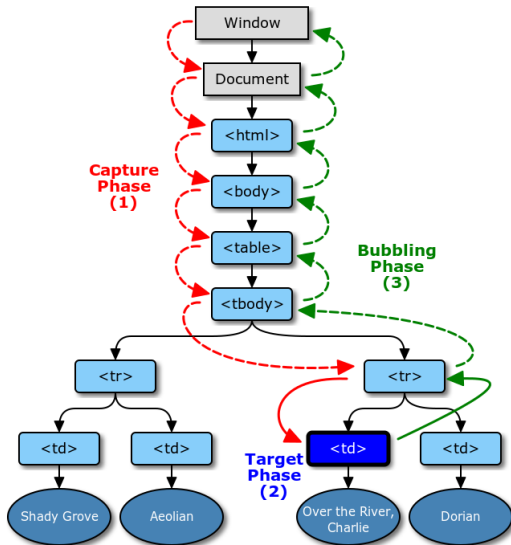


Example

```
<input type="button" value="Lancer">

<script>
  let src = document.querySelector('input');
  src.addEventListener("click", function() {
    console.log('clicked');
  });
</script>
```

Flux événementiel du DOM





Example

```
<input type="button" value="Lancer">
<p> Here ! </p>

<script>
  let custom = new CustomEvent("custom", {
    detail: {message: "Hello World!", time: new Date()},
    bubbles: true,
    cancelable: true
  });

  let src = document.querySelector('input');
  src.addEventListener("click", function() {
    dst.dispatchEvent(custom);
  });

  let dst = document.querySelector('p');
  dst.addEventListener("custom", function(evt) {
    dst.innerText= evt.detail.message;
  });
</script>
```

Parmi les instructions permettant de contrôler les événements :

- ▶ `evt.preventDefault()`
- ▶ `evt.stopPropagation()`
- ▶ `evt.stopImmediatePropagation()`

Il y a de nombreuses catégories d'événement prédéfinis.

<https://developer.mozilla.org/en-US/docs/Web/Events#Categories>

Événements souris

On trouve :

click	dblclick
mousedown	mouseup
mouseover	mouseout
mousemove	

Les propriétés utiles et accessibles à partir de l'objet `event` sont :

clientX	clientY
screenX	screenY
shiftKey	ctrlKey
altKey	metaKey

Événements clavier

On trouve :

keydown	keyup
keypress	

Les propriétés utiles et accessibles à partir de l'objet `event` sont :

shiftKey	ctrlKey
altKey	metaKey
keyCode	

Exemples de valeurs pour `keyCode`:

- ▶ 40 pour Down Arrow
- ▶ 65 pour A
- ▶ 112 pour F1

On trouve :

load	unload
abort	error
select	change
submit	reset
resize	scroll
focus	blur

Remark

La plupart de ces évènements sont liés aux formulaires ou l'objet `window`.

Événements load et unload

Pour l'objet `window`, l'événement `load` se produit lorsque la page complète est chargée, incluant les ressources externes telles que les images, les fichiers JavaScript et CSS.



Example

```
<script>
  window.addEventListener("load", function(event) {
    console.log("All resources finished loading!");
  });
</script>
```

Remark

Il est possible d'associer cet événement à des éléments `img`.

Remark

L'évènement `unload` se produit typiquement lorsqu'on change de page. Cela permet par exemple de libérer proprement certaines ressources.

Événements submit et reset

Ils sont associés à un formulaire.



Example

```
<form action="#">
  <input name='name'>
  <input type='reset'>
  <input type='submit'>
</form>
<script>
  let form = document.querySelector('form');
  form.addEventListener("submit", function(evt) {
    let name = form.name.value;
    if (name === 'toto') {
      evt.preventDefault();
      alert('You cannot send the data');
    }
  });
  form.addEventListener("reset", function(evt) {
    if (!confirm("Are you sure?"))
      evt.preventDefault();
  });
</script>
```



Example

```
<input>
<script>
  let e = document.querySelector('input');
  e.addEventListener("focus", function(evt) {
    if (e.style.backgroundColor !== "red")
      e.style.backgroundColor="yellow";
  });
  e.addEventListener("blur", function(evt) {
    if (e.value.match(/\d/g))
      e.style.backgroundColor = "red";
    else
      e.style.backgroundColor = "";
  });
</script>
```

Ajax (Asynchronous JavaScript and XML) permet une communication asynchrone avec un serveur. Des éléments de la page peuvent être mis à jour avec les données du serveur (sans recharger toute la page).

Il faut tout d'abord un serveur (à lancer avec node.js) :



Example

```
'use strict';
const http = require('http');
const server = http.createServer(function(req, res) {
  res.setHeader('Content-Type', 'application/json');
  res.setHeader('Access-Control-Allow-origin', '*');
  res.end(JSON.stringify({
    platform: process.platform,
    nodeVersion: process.version,
    uptime: Math.round(process.uptime())
  }));
});
const port= 7070;
server.listen(port, function() {
  console.log('Ajax server started on port ' + port);
});
```

Coté client :



Example

```
<script>
  function refresh() {
    const req = new XMLHttpRequest();
    req.addEventListener('load', function(evt) {
      const data = JSON.parse(this.responseText);
      console.log(data);
    });
    req.open('GET', 'http://localhost:7070', true);
    req.send();
  }
  setInterval(refresh, 1000);
</script>
```

Remark

En pratique, il faudrait utiliser la variable *data* pour mettre à jour la page. A faire en exercice.

Plan

Introduction

Variables et Types

Structures de contrôle

Fonctions

Arrays, Maps and Sets

Expressions et opérateurs

Objects and Classes

Comportement asynchrone

Iterators and generators

Expressions régulières

DOM and Events

jQuery

Librairie Javascript développée depuis 2006

Avantages :

- ▶ interface simple et puissante pour écrire du code
- ▶ aplanit les différences entre navigateurs
- ▶ beaucoup de ressources disponibles

Utiliser jQuery :

- ▶ download à www.jquery.com
- ▶ ou utiliser un CDN (Content Delivery Network). Par exemple :



```
<script src="http://ajax.googleapis.com/ajax/libs/jquery  
/1.11.1/jquery.min.js">  
</script>
```

Sélectionner avec jQuery

L'instruction `$(selecteur)`, ou `jquery(selecteur)`, retourne le ou les éléments sélectionnés.



Example

```
<h1 id="test"></h1>

$("#test").text("premier essai");
```

L'instruction `$(document).ready()` permet d'exécuter du code (en argument de `ready`) lorsque le document DOM est totalement chargé



Example

```
$(document).ready(function() {
    $("#test").text("premier essai");
});
```

Warning

Utiliser une fonction anonyme dans `ready()`



Example

```
<body>
  <ul id="menu">
    <li class="item"><p>This is a paragraph</p></li>
    <li class="item">No paragraph here</li>
    <li class="item">No paragraph here</li>
  </ul>
  <p>Email: <input type="email" id="email"/></p>
  <p>Plain Text: <input type="text" id="text"/></p>

  <script src="//ajax.googleapis.com/.../jquery.min.js">
  </script>
  <script src="my.js"></script>
</body>

$(document).ready(function() {
  $(".item").css("color", "blue");
  $("#menu .item p").css("color", "red");
  $("input[type=email]").css("border", "10px solid blue");
});
```

Sélectionner des éléments avec jQuery

Sélecteurs spécifiques à jQuery (voir <http://api.jquery.com/category/selectors/jquery-selector-extensions/>) :

Sélecteur	Sens	Exemple
:eq()	élément à la position donnée	<code>\$("li:eq(2)")</code>
:gt()	éléments aux positions supérieures	<code>\$("li:gt(2)")</code>
:lt()	éléments aux positions inférieures	<code>\$("li:lt(2)")</code>
:first	premier élément matché	<code>\$("tr:first")</code>
:last	dernier élément matché	<code>\$("tr:last")</code>
:even	éléments aux positions paires	<code>\$("tr:even")</code>
:odd	éléments aux positions impaires	<code>\$("tr:odd")</code>
:animated	éléments en cours d'animation	<code>\$("p:animated")</code>
:selected	éléments sélectionnés	<code>\$("option:selected")</code>
:visible	éléments visibles	<code>\$("p:visible")</code>
:hidden	éléments cachés	<code>\$("p:hidden")</code>
:has()	éléments contenant	<code>\$("div:has(p)")</code>
:parent	éléments parents	<code>\$("p:parent")</code>

Créer des éléments avec jQuery

L'instruction `$(element)` crée un nouvel élément.



Example

```
<div id="container"></div>

$(document).ready(function() {
  let time = new Date().getHours();
  let elem = $("

# ").attr("id", "greeting").hide(); if (time < 12) elem.text("Good Morning"); else elem.text("Good Afternoon"); $("#container").append(elem); $("#greeting").show("slow"); });


```

Remark

Il est possible d'écrire :



```
if (time < 12)
  elem = $("

# 


```

Quelques méthodes

Les méthodes suivantes sont couramment utilisées :

- ▶ `text()` : get/set le contenu textuel de l'élément
- ▶ `html()` : get/set le contenu html d' l'élément
- ▶ `css()` : get/set les propriétés CSS
- ▶ `attr()` : get/set les attributs de l'élément
- ▶ `hide()`, `show()` et `toggle()` : cache ou rend visible l'élément
- ▶ `fadeIn()`, `fadeOut()` et `fadeToggle()` : pour jouer avec l'opacité
- ▶ `slideUp()`, `slideDown()` et `slideToggle()` : pour un effet glissant
- ▶ `addClass()`, `removeClass()`, `hasClass()` et `toggleClass()` permettent de modifier dynamiquement la valeur de l'attribut `class`

Remark

La plupart permettent d'effectuer du `method chaining`

Exemple avec attr()



Example

```
<ul>
  <li id="contact"><a>Toto</a></li>
</ul>

$(document).ready(function() {
  $("#contact a").attr({
    "href" : "http://www.toto.com/",
    "title" : "Visit Toto website",
    "id" : "atoto"
  });
  let li = $("<li>").text($("#contact a").attr("title"));
  $("ul").append(li);
});
```


Exemple avec css()



Example

```
<h1>Beginning HTML and CSS</h1>
<p>
  <strong>The border property is:</strong>
  <span id="result"></span>
</p>

$(document).ready(function() {
  $("h1").css({
    "font-size" : "200%",
    "color" : "#ffffff",
    "height" : "100px",
    "width" : "500px",
    "background-color" : "#61b7ff",
    "border" : "10px solid #003366"
  });
  $("#result").text($("h1").css("border"));
});
```

Exemple avec html()



Example

```
$(document).ready(function() {  
    let methods = ["attr()", "css()", "html()", "addClass()",  
        , "removeClass()", "hasClass()", "toggleClass()"];  
    let list="";  
    for (let i=0, len=methods.length; i < len; i++)  
        list += "<li>" + methods[ i ] + "</li>";  
    $("#methods").html(list);  
});
```

Exemple avec toggleClass()



Example

```
.selected {  
  background : #666;  
  color : #fff;  
}  
  
<ul>  
  <li id="home" class="item">Home</li>  
  <li id="about" class="item">About</li>  
  <li id="contact" class="item">Contact</li>  
</ul>  
  
$(document).ready(function() {  
  $("#home").on("click", function() {  
    $("#home").toggleClass("selected");  
  });  
});
```

Exemple avec show() et toggle()



Example

```
$(document).ready(function() {  
    $("#slow").show("slow");  
    $("#fast").show("fast");  
    $("#ms").show(1500);  
    $("#toggle").on("click", function() {  
        $("#toggled").toggle();  
    });  
});
```

Exemple avec slideToggle()



Example

```
<dl>
  <dt>Term </dt>
  <dd>
    <ul>
      <li>item 1</li>
      <li>item 2</li>
      <li>item 3</li>
    </ul>
  </dd>
  ...
</dl>

$(document).ready(function() {
  $("dt").on("click", function() {
    $(this).next().slideToggle();
  })
});
```

Gestion des événements

Méthodes importantes :

- ▶ `on()` pour enregistrer un écouteur
- ▶ `off()` pour supprimerun écouteur
- ▶ `trigger()` pour déclencher un événement

en remplacement depuis jQuery 1.7 de `click()`, `submit()`, `live()`, `die()`, `delegate()`, `undelegate()`, `bind()` and `unbind()`



Example

```
$(document).ready(function() {  
    function toggler() {  
        $(this).next().slideToggle();  
    }  
    $("dt").on("click", toggler);  
    $("button").on("click", function() {  
        $("dt").trigger("click").off("click", toggler);  
    });  
});
```

Méthode on() à trois arguments

Le second paramètre désigne le type d'éléments qui est concerné par l'écouteur (même si un élément de ce type est créé plus tard).

“Listen to every click on the whole document, and if it happens on an `<a>` element, fire this event.” :



Example

```
$(document).on( "click", "a", function() {  
    //code goes here  
});
```

jQuery UI

jQuery UI:

- ▶ collection de GUI widgets
- ▶ download à www.jqueryui.com ou utiliser un CDN

Squelette :



```
<head>
  <meta charset="utf-8">
  <title>jQuery UI Draggable and Droppable</title>
  <link rel="stylesheet" href="http://ajax.googleapis.com/
    ajax/libs/jqueryui/1.11.1/themes/smoothness/jquery-
    ui.css"/>
</head>
<body>
  ...
  <script src="http://ajax.googleapis.com/ajax/libs/jquery
    /1.11.1/jquery.min.js">
  </script>
  <script src="http://ajax.googleapis.com/ajax/libs/
    jqueryui/1.11.1/jquery-ui.min.js">
  </script>
</body>
```




Example

```
<div id="container">
  <div id="draggable">
    <p>Drag here</p>
  </div>
  <div id="droppable">
    <p>Drop here</p>
  </div>
</div>

$(document).ready(function() {
  $("#draggable").draggable();
  $("#droppable").droppable( {
    drop: function(event, ui) {
      $(this).css("border", "4px solid").html("<p>Dropped
        !</p>");
    }
  });
});
```



Example

```
<div id="resizable"> <p>Resizable Element</p> </div>
```

```
$(document).ready( function() {  
    $("#resizable").resizable();  
});
```

```
<ol id="selectable">  
    <li>These items are selectable </li>  
    <li>These items are selectable </li>  
    <li>These items are selectable </li>  
</ol>
```

```
$(document).ready( function() {  
    $("#selectable").selectable();  
});
```



Example

```
<ol id="sortable">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
  <li>Five</li>
</ol>

$(document).ready(function() {
  $("#sortable").sortable().disableSelection();
});
```

Et tous les autres widgets sur <http://jqueryui.com/>. En particulier :

- ▶ accordion
- ▶ tabs