Cours Javascript & PHP

L3 Informatique - UE Développement Web

David Lesaint

Université d'Angers

Septembre 2024

Plan

1. UE Développement Web
   - Programme
2. Javascript
   - Introduction
   - Bases
   - L'API DOM
   - Manipuler le CSS
   - Gestion d'évènements
   - Formulaires
   - Programmation asynchrone et promesses
   - APIs Fetch et XHR
   - Web workers
   - Les API
   - Sécurité : CORS et CSP

CM UE : Programme

Programme

Partie 1 : Javascript

- 4 CM de 1h20
- 4 TP de 2h40
- CC1 de 1h30 sur machine en semaine 41 coefficient 4/10

Partie 2 : PHP

- 6 CM de 1h20
- 6 TP de 2h40
- CC2 de 1h30 sur machine en semaine 48 coefficient 6/10

Groupes CM et TP

1 groupe CM

3 groupes de TP

- Groupe 1 : B. Nini en H001
- Groupe 2 : D. Lesaint en H002
- Groupe 3 : C. Vasconcellos en H003

Supports

Espace Moodle : n° 6924, clé 8se43u

- Groupe, calendrier, planning détaillé
- CM : livret (diaporama) + annexes
- TD et TP : livret des énoncés + annexes + corrections
- CC : énoncés + annexes + dépôts

CM JS : Introduction

ECMAScript

ECMAScript

- Ensemble de normes pour les langages de script (JavaScript, C++) standardisées par ECMA International.
- Spécification ECMA-262.
- Dernière version : Edition 14 (ES14 - juin 2023).
- Quelques moteurs JS :
  - Spidermonkey (C/C++) : Firefox (Mozilla), navigateurs fondés sur KHMTL (Konqueror - KDE)
  - V8 : Chrome (Google)
  - Javascriptcore : Safari (Apple)

Utilisation de JavaScript

Pour le navigateur

- Introduction grâce à la balise `script`.
- Utilisé dans la partie `<head>` ou `<body>`.

Utilisation de JavaScript pour le navigateur

js.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Titre de la page</title>
5  <script>
6    var texte='Hello world !';
7  </script>
8  <script src="js-coucou.js"></script>
9  </head>
10 <body>
11   <script>
12   document.write(texte);
13   writecoucou();
14   </script>
15 </body>
16 </html>
```

CM JS : Bases

Spécificités

Spécificités liées à la sécurité

- Ne peut pas lire ou écrire dans le système de fichier de la machine.
- Ne peut exécuter de programme externe.
- Pas de connexion autre qu'avec le serveur web.

Littéraux

Valeurs primitives

- `number` : `111`, `3.14`.
- `string` : `'Jean Dupont'`, `"Jade Durant"`.
- `boolean` : `true` et `false`.
- `undefined` : valeur d'une variable sans valeur.
- `null` : valeur d'une variable qui n'existe pas.

js-undefined-null.js

```
1 typeof undefined   // undefined
2 typeof null        // object
3 null === undefined  // false
4 null == undefined // true
```

Littéraux

Valeurs non primitives

- `object` : `var person = {name:"John", age:50};`
- `function` : `function f2c(t) { return (5/9)*(t-32); }`

Toutes les valeurs, sauf les valeurs primitives, sont des objets

- les fonctions sont un type d'objet particulier
- les tableaux sont de type `object`
  - tableau : `var cars = ["Renault", "Citroen"];`

js-complextypes.js

```
1 typeof [1,2,3,4]          // object
2 typeof {name:'John', age:34}   // object
3 typeof function myFunc(){}     // function
```

Variables

Variables

- Identifiants : peuvent contenir lettres, chiffres, $ et _ mais ne débutent pas par un chiffre
- Leur typage est dynamique
- Elles peuvent être déclarées à tout moment
- On peut utiliser les mots-clés `var` ou `let` pour les déclarer
- On peut les redéclarer : leur valeur est préservée par défaut

Eviter le $ en début de nom de variable (utilisé par de nombreuses librairies JS)

Déterminer le type d'une variable avec typeof

js-typeof.js

```
1  var b=true;    console.log(typeof b); // boolean
2  var s='coucou'; console.log(typeof s); // string
3  var i=10;    console.log(typeof i); // number
4  var k;       console.log(typeof k); // undefined
5  var tab=[1,2];  console.log(typeof tab); // object
6  var p=null;    console.log(typeof p); // object
7  var reg=/\w+/;  console.log(typeof reg); // object
8  function f() {} console.log(typeof f); // function
9          console.log(typeof none); // undefined
10 console.log(b); // true
11 console.log(s); // coucou
12 console.log(i); // 10
13 console.log(k); // undefined
14 console.log(tab); // Array[1,2]
15 console.log(p); // null
16 console.log(reg); // /\w+/
17 console.log(f); // function f() {}
```

Portée de variables

Portée d'une variable

L'espace du script dans laquelle elle va être accessible :

- Espace "global" : l'entièreté du script sauf l'intérieur des fonctions
- Espace "local" : l'intérieur d'une fonction

La portée d'une variable dépend

- D'où elle est déclarée : en dehors ou dans une fonction
- Du type de déclaration : avec `let`, `var` ou sans mot-clé

Règles

- Portée locale pour les variables déclarées dans les fonctions avec `let` ou `var`
- Globale pour les autres

Portée de variables

js-variable-scope-1.js

```
1  let xlet = "let x";
2  var xvar = "var x";
3  xglobal = "global x";
4
5  function externe() {
6    console.log(xlet); // let x
7    console.log(xvar); // var x
8    console.log(xglobal); // global x
9    xvar = "xvar modifiée";
10   xglobal = "xglobal modifiée";
11   let flet = "function : let";
12   var fvar = "function : var";
13   fglobal = "function : globale !!";
14 }
15 externe();
16 console.log(xvar); // xvar modifiée
17 console.log(xglobal); // xglobal modifiée
18 //console.log(flet); // ReferenceError : flet is not defined
19 //console.log(fvar); // ReferenceError : fvar is not defined
20 console.log(fglobal); // function : globale !!
```

Portée de variables : fonctions internes

js-variable-scope-2.js

```
1  var xvar = "var x";
2
3  function externe() {
4      let elet = "externe : let";
5      var evar = "externe : var";
6
7      function interne() {
8          console.log(elet); // externe : let
9          console.log(evar); // externe : var
10         console.log(xvar); // externe : var x
11         let ilet = "interne : let";
12         var ivar = "interne : var";
13     }
14     interne();
15     //console.log(ilet); // ReferenceError : ilet is not defined
16     // console.log(ivar); // ReferenceError : ivar is not defined
17 }
18
19 externe();
```

Opérateurs

Opérateurs

Affectation	`=`	`+=`	`-=`	`*=`	`/=`	`%=`			
Arithmétiques	`+`	`-`	`*`	`/`	`%`	`++`	`-`		
Chaînes	`+`	`+=`							
Comparaison	`==`	`===`	`!=`	`!==`	`>`	`<`	`>=`	`<=`	`?:`
Logiques	`&&`	`||`	`!`						
Binaires	`&`	`|`	`~`	`^`	`<<`	`>>`	`>>>`		
Types	`typeof`		`instanceof`						

Structures de contrôle

Structures de contrôle

JS possède les structures de contrôle du langage C :

- `if/else`
- `switch/case`
- `for`
- `for/of` pour les tableaux ou objets itérables (conteneurs)
- `for/in` pour les objets
- `do/while`
- `while`, `break/continue`
- `throw/try/catch/finally`

Fonctions

Fonctions

- Ce sont des objets "fonction" de type `function`
- Elles peuvent être déclarées à tout moment avec le mot-clé `function`
- Les arguments peuvent ne pas être déclarés
- Type des arguments et valeur retour non déclarés
- Invoquées explicitement, en fonction d'évènements utilisateur, ou auto-exécutées (IIFE)
- Peuvent déclarer et retourner d'autres fonctions

- Variables passées par valeur, objets par référence

Fonctions

js-function.html

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"/></head>
<body>
<p id="demo"></p>
<script>
function toCelsius(f) { return (5/9) * (f-32); }
document.getElementById("demo").innerHTML =
  "This JS function<blockquote>"
  + String(toCelsius)
  + "</blockquote>yields "
  + String(toCelsius(50))
  +"&#8451 for 64&#8457";
</script>
</body>
</html>
```

Déclaration optionnelle des arguments

js-function-arguments.js

```
1 function f() {
2   console.log(arguments.length);
3 }
4 f(); // 0
5 f(1); // 1
6 f('riri','fifi','loulou'); // 3
```

Propriétés et fonctions globales

Les propriétés globales de JS

- `Infinity` : une valeur de type `number` représentant $+/-\infty$
- `NaN` : valeur qui “n’est pas un nombre"
- `undefined` : valeur du type `undefined`

Propriétés et fonctions globales

Les fonctions globales

- `eval(string)` : évalue la chaîne comme du code JS
- `Number(var)` : convertit en nombre
- `String(var)` : convertit en chaîne de caractères
- `int parseInt(string[,radix])` : convertit en entier
- `float parseFloat(string)` : convertit en flottant
- `isFinite(var)` : teste si `var` est un nombre légal et fini
- `isNaN(var)` : teste si `var` est un nombre illégal
- `encodeURI(uri)` : encode un URI en UTF-8 (sauf caractères réservés, alphabet latin, chiffres, ...)
- `decodeURI(uri)` : décode un URI

Utilisation

js-function-global.js

```
1  console.log(parseInt('ff',16)); // 255
2  var str=' 256';
3  var x = 1 + str;
4  console.log(x); // '1 256'
5  var x =1 + Number(str);
6  console.log(x); // 257
7  console.log(eval('2 + 2 * 8 - 3')); // 15
8  // JSON
9  var person=eval("({ name: 'Ada', age : 30 })");
10 console.log(person); // [object Object]
11 console.log(person.name+' '+person.age); // Ada 30
12 var uri=encodeURI('http://www.site.fr/x="10-2<9"');
13 console.log(uri); // http://www.site.fr/x=%2210-2%3C9%22
```

Couche Objet en Javascript

La couche objet en Javascript

La couche objet

JS est un langage à base de *prototypes*

- Tout est objet : il n'y pas de concept de classe
- L'héritage s'effectue par chaînage de prototypes d'objets

Tout objet JS possède

- des membres : propriétés et/ou méthodes (propriétés de type `function`)
- une propriété spéciale de type objet appelée prototype

Les prototypes étant des objets, tout objet a une chaîne de prototypes se terminant avec `null` (le prototype de `Object`)

Tout objet possède

- des propriétés qui lui sont propres (own properties)
- des propriétés héritées de sa chaîne de prototypes

Construction d'objets

Plusieurs possibilités pour construire des objets

1. affectation d'un littéral avec accolades (format proche du format JSON)
2. invocation d'une fonction (alias constructeur) avec le mot-clé `new`
   - `new f(...)`
3. appel de `Object.create(o)` où *o* est un objet qui servira de prototype à l'objet créé

Prototypes prédéfinis

JS a des prototypes prédéfinis : `Object`, `String`, `Date` ...

Propriétés d'objets

Manipulation de propriétés

- Accès : `obj.prop`
- Itération : `for(prop in obj) {...obj[prop]...}`
- Ajout à l'objet : `obj.prop = ...;`
- Destruction : `delete obj.prop;`

- Un objet peut être manipulé comme un tableau associatif de propriétés `nom-valeur` avec la syntaxe `obj["nom"]`
  - que pour les objets : tous les tableaux JS sont indexés numériquement
- `delete` ne s'applique ni aux variables, ni aux fonctions

Méthodes d'objets

Manipulation de méthodes

- Accès : `obj.method`
  - retourne l'objet `function` correspondant
- Invocation : `obj.method()`
- Ajout à l'objet : `obj.method = ...;`

Le mot-clé `this`

- `this` dans le corps d'une fonction fait référence
  - à l'objet que l'on crée dans le contexte d'un constructeur
  - à l'objet appelant dans le contexte d'un appel normal

Construction d'objet littéral : avec propriétés seules

js-object-litteral-property.js

```
1  var joe = {
2      first_name: "Joe",
3      last_name: "Dalton",
4      brothers: [{
5          name: "Jack",
6          age: 30
7      }, {
8          name: "Averell",
9          age: 33
10     }, {
11         name: "William",
12         age: 31
13     }]
14 };
15 for (let p in joe) {
16     if (joe[p] instanceof Array) // <=> joe[p].isArray()
17         for (let i = 0; i < joe[p].length; ++i)
18             console.log(joe[p][i].name);
19     else
20         console.log('nom=' + p + ' valeur=' + joe[p]);
21 }
```

Construction d'objet littéral : avec propriétés et méthodes

js-object-litteral-method.js

```
1  var joe = {
2      name: "Joe",
3      display: function() {
4          console.log(this.name);
5      },
6      brothers: [{
7          name: "Jack",
8          age: 30
9      }, {
10         name: "Averell",
11         age: 33
12     }, {
13         name: "William",
14         age: 31
15     }]
16 };
17 joe.display();
```

Construction avec constructeur

js-object-prototype1.js

```
1  function Person(first) {
2    this.first_name=first;
3    this.last_name="Dalton";
4    this.display=function() {
5      console.log(this.first_name+' '+this.last_name);
6    };
7  }
8  let joe=new Person('Joe');
9  let jack=new Person('Jack');
10 joe.display(); // Joe Dalton
11 jack.display(); // Jack Dalton
12 jack.last_name = "Notlad";
13 joe.display(); // Joe Dalton
14 jack.display(); // Jack Notlad
```

- Toute propriété p initialisée dans le constructeur avec `this.p` est une propriété propre de l'objet créé

Construction avec Object.create

js-object-prototype2.js

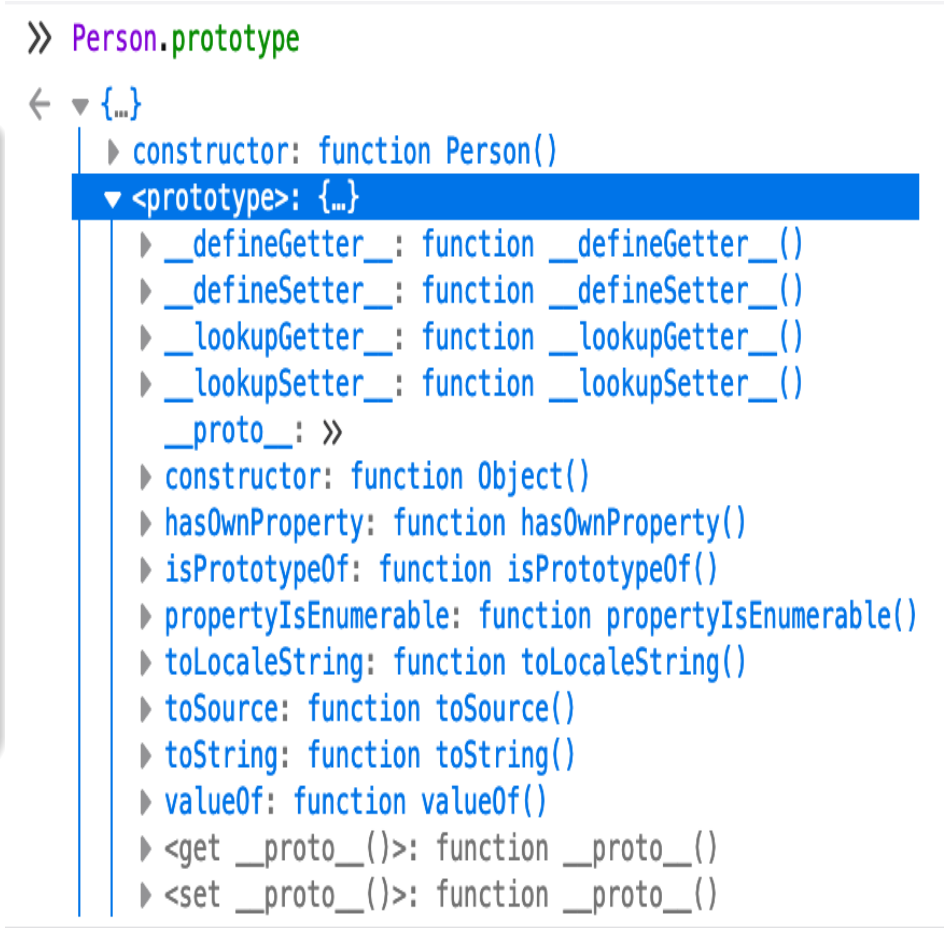
```
1  function Person(first) {
2    this.first_name = first;
3    this.last_name = "Dalton";
4    this.display = function() {
5      console.log(this.first_name + ' ' + this.last_name);
6    };
7  }
8  let joe = new Person('Joe');
9  let jack = Object.create(joe);
10 jack.display(); // Joe Dalton
11 jack.first_name = "Jack";
12 jack.display(); // Jack Dalton
```

Utile pour cloner un objet dont on ne connait pas le constructeur

Prototype et héritage prototypique

L'objet prototype d'un constructeur

- Propriété de type objet que JS ajoute à tout constructeur
- Contient initialement 2 propriétés : `constructor` et `__proto__`
- Accessible via la propriété `prototype` du constructeur



Prototype et héritage prototypique

Le prototype d'un objet

- C'est une propriété interne (généralement dénotée `__proto__` dans la console)
- On y accède via `Object.getPrototypeOf(o)` (ou `o.__proto__`)
- JS lui affecte le prototype objet du constructeur de l'objet

Héritage prototypique

Tout objet hérite des propriétés de son prototype

- Propriétés partagées par tous les objets ayant ce même prototype
- Propriétés distinctes des propriétés propres de l'objet

Chaîne de prototypes

- Un prototype étant lui-même un objet, tout objet hérite des propriétés de sa chaîne de prototypes par "transitivité"
- Lorsqu'on tente d'accéder à une propriété sur un objet, JS la recherche parmi les propriétés propres de l'objet, puis parmi celles de son prototype, puis celles du prototype de son prototype, . . .jusqu'à la trouver ou pas
- Toute chaîne se termine avec le prototype de valeur `null` de `Object`

Héritage prototypique

```
>> Person.prototype
<- {…}
    constructor: function Person()
    <prototype>: {…}
        __defineGetter__: function __defineGetter__()
        __defineSetter__: function __defineSetter__()
        __lookupGetter__: function __lookupGetter__()
        __lookupSetter__: function __lookupSetter__()
        __proto__: {…}
            __defineGetter__: function __defineGetter__()
            __defineSetter__: function __defineSetter__()
            __lookupGetter__: function __lookupGetter__()
            __lookupSetter__: function __lookupSetter__()
            __proto__: null
            constructor: function Object()
            hasOwnProperty: function hasOwnProperty()
            isPrototypeOf: function isPrototypeOf()
            propertyIsEnumerable: function propertyIsEnumerable()
            toLocaleString: function toLocaleString()
            toSource: function toSource()
            toString: function toString()
            valueOf: function valueOf()
            <get __proto__()>: function __proto__()
            <set __proto__()>: function __proto__()
        constructor: function Object()
        hasOwnProperty: function hasOwnProperty()
        isPrototypeOf: function isPrototypeOf()
        propertyIsEnumerable: function propertyIsEnumerable()
        toLocaleString: function toLocaleString()
        toSource: function toSource()
        toString: function toString()
        valueOf: function valueOf()
        <get __proto__()>: function __proto__()
```

Héritage prototypique

js-object-addprototypeproperty.js

```
 1 function Person(first) {
 2   this.first = first;
 3   this.last = "Dalton";
 4 }
 5 var joe = new Person('Joe');
 6 Person.prototype.display = function() {
 7   console.log(this.first + ' ' + this.last + ' -
' + this.job);
 8 };
 9 Person.prototype.job = "outlaw";
10 joe.display(); // Joe Dalton - outlaw
11 var jack = new Person('Jack');
12 jack.last = "Notlad";
13 jack.display(); // Jack Notlad - outlaw
```

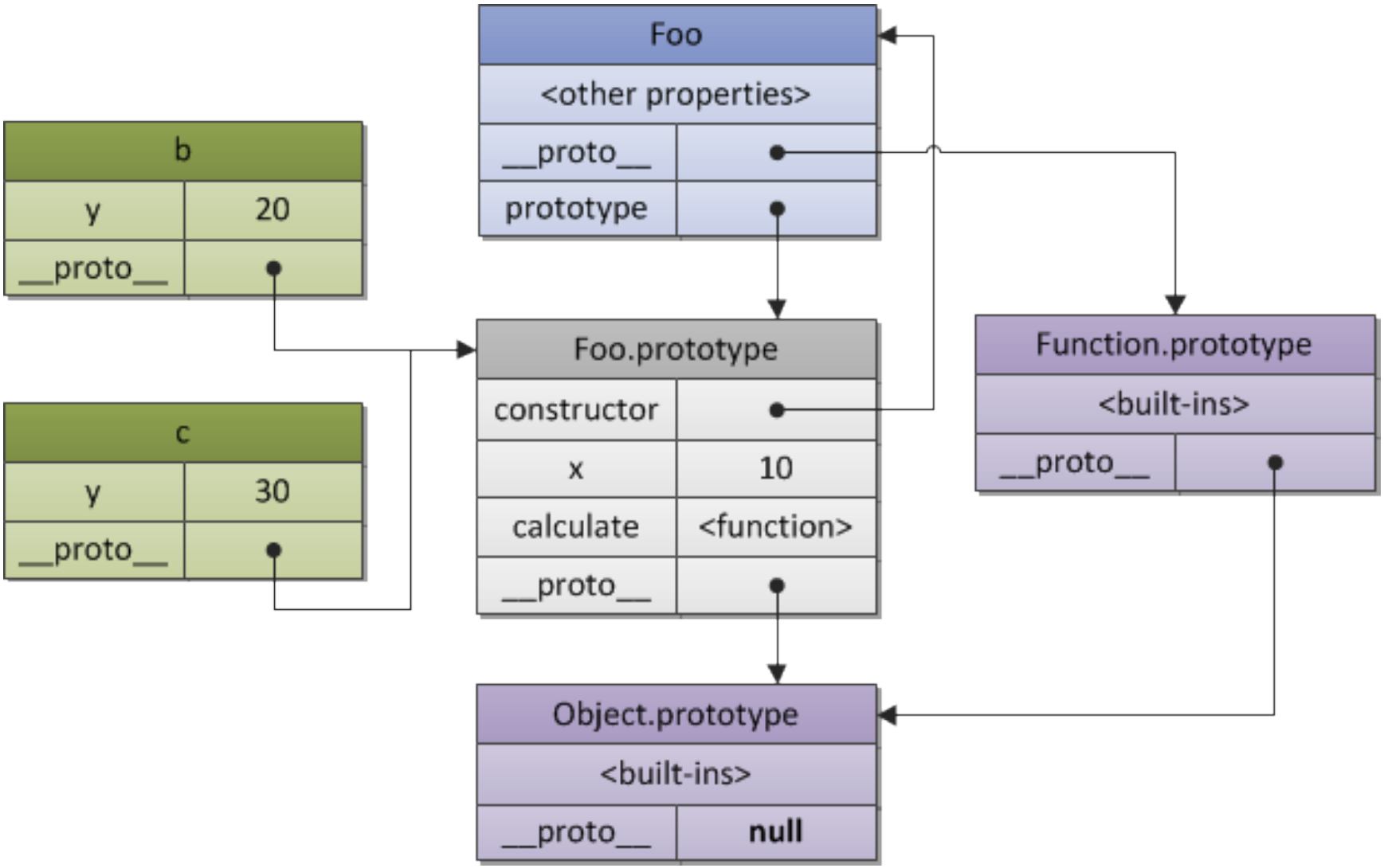
```
» joe
← ▸ Object { first: "Joe", last: "Dalton" }
» joe.__proto__
← ▾ {…}
    ▸ constructor: function Person()
    ▸ display: function display()
      job: "outlaw"
    ▸ <prototype>: Object { … }
» jack
← ▸ Object { first: "Jack", last: "Notlad" }
» jack.__proto__
← ▾ {…}
    ▸ constructor: function Person()
    ▸ display: function display()
      job: "outlaw"
    ▸ <prototype>: Object { … }
```

Héritage par prototype : illustration

objets-proto-vs-prototype.js

```
1  // constructeur
2  function Foo(y) {
3    // propriété en propre pour chaque objet construit avec Foo
4    this.y = y;
5  }
6  // propriété héritée par tout objet construit avec Foo
7  Foo.prototype.x = 10;
8  // méthode héritée par tout objet construit avec Foo
9  Foo.prototype.calculate = function (z) { return this.x + this.y + z; };
10
11 var b = new Foo(20);
12 var c = new Foo(30);
13 b.calculate(30); // 60
14 c.calculate(40); // 80
15 console.log(
16   b.__proto__ === Foo.prototype, // true
17   c.__proto__ === Foo.prototype, // true
18   b.constructor === Foo, // true
19   c.constructor === Foo, // true
20   Foo.prototype.constructor === Foo, // true
21   b.calculate === b.__proto__.calculate, // true
22   b.__proto__.calculate === Foo.prototype.calculate // true
23 );
```

Héritage par prototype : illustration



Mutabilité des objets

- Les objets sont passés par référence aux fonctions
- Deux objets construits avec le même constructeur et ayant les mêmes propriétés ne sont pas égaux

js-object-equality.js

```
1  function Person(first,last) {
2    this.firstname=first;
3    this.lastname=last;
4  }
5  var p1=new Person('Joe','Dalton');
6  console.log(typeof p1); // object
7  var p2=new Person('Joe','Dalton');
8  console.log(p2==p1); // false !!
9  var p3=p1; console.log(p3==p1); // true
10 p3.firstname="Jack";
11 console.log(p3==p1); // true
```

Les fonctions auto-exécutées (IIFE)

Immediately Invoked Function Expression (IIFE)

On peut déclarer et auto-exécuter une fonction en une instruction avec la syntaxe `(function(){...})();`

js-function-selfinvoking.js

```
1  // fonction auto-exécutée
2  (function () {
3    console.log("Good "); return "morning";
4  })(); // affiche 'Good'
5
6  //fonction auto-exécutée
7  var f = (function () {
8    console.log("Good "); return "morning";
9  })();  // affiche 'Good'
10 console.log(f); // affiche 'morning'
```

Les fonctions imbriquées

Fonctions imbriquées

- Une fonction peut en contenir d'autres (c'est un objet)
- On peut ainsi définir des fonctions récursives ainsi que des fermetures

js-function-nested.js

```
1  // fonction externe auto-exécutée
2  f = (function() {
3    console.log("Good ");
4    // fonction interne
5    g = function() {
6      console.log("morning ...");
7    };
8    return g;
9  })(); // affiche 'Good'
10 f(); // affiche 'morning ...'
11 f(); // affiche 'morning ...'
```

Les fermetures (closures)

Fermetures

- Une fermeture est une fonction interne retournée par une fonction anonyme auto-exécutée
- La fonction interne a l'accès exclusif aux propriétés et méthodes de la fonction externe !
- Ces propriétés et méthodes sont "privées"

js-function-closure1.js

```
1 var add = (function () {
2   var counter = 0;
3   return function () {return counter += 1;}
4 })();
5 add();
6 console.log(add()); // 2
7 console.log(add.counter); // undefined
```

Les fermetures

js-function-closure11.js

```
1 var add = (function () {
2   return function () {return this.counter += 1;}
3 })();
4 c1 = add.bind({'counter':0});
5 c2 = add.bind({'counter':20});
6 console.log(c1()); // 1
7 console.log(c1()); // 2
8 console.log(c2()); // 21
9 console.log(c2()); // 22
10 console.log(add()); // NaN
```

Les fermetures

js-function-closure2.js

```
1  var f = function (val) {
2    var p = val;
3    return {
4      setT: function(newval) {p = "GG"+newval;},
5      setP: function(newval) {p = newval;},
6      getP: function () {return p;}
7    }
8  }
9  var some = f("one");
10 console.log(some.getP()); // one
11 some.setP("where");
12 console.log(some.getP()); // where
```

Constructeurs natifs

Constructeurs natifs

Chacun de ces types d'objets possède des méthodes propres :

- `Object`
- `String`
- `Number`
- `Boolean`
- `Array`
- `RegExp` (expression régulière)
- `Function`
- `Date`

- `Math` est un objet global

Recommandations

Privilégier les littéraux : meilleures performances et lisibilité

- `Object` : `var x1 = {};`
- `String` : `var x2 = "";`
- `Number` : `var x3 = 0;`
- `Boolean` : `var x4 = false;`
- `Array` : `var x5 = [];`
- `RegExp` : `var x6 = /()/;`
- `Function` : `var x7 = function(){};`

On peut toujours utiliser sur une valeur primitive les propriétés et méthodes du constructeur correspondant :

```
var x = "une chaîne";
console.log(x.length)
console.log(x.toUpperCase())
```

L'objet Boolean

Boolean

Permet de représenter des valeurs booléennes

1. attributs :
   - `TRUE`
   - `FALSE`
2. méthodes :
   - `valueOf()`
   - `toString()`

Utiliser Boolean

js-boolean.js

```
1  var vrai=Boolean(true);
2  var faux=Boolean();
3  var p=Boolean.TRUE;
4
5  console.log(vrai); // true
6  console.log(faux); // false
7  console.log(typeof p); // undefined
8
9  console.log(vrai.toString()); // true
10
11 p=vrai.valueOf();
12 console.log('p='+p); // p=true
13 p=vrai.toSource();
14 console.log('p='+p); // p=(new Boolean(true))
```

L'objet Number

Number

Permet de représenter les nombres entiers et les réels

1. attributs :
   - `MAX_VALUE`
   - `MIN_VALUE`
   - `NaN`
   - `NEGATIVE_INFINITY`
   - `POSITIVE_INFINITY`
2. méthodes :
   - `valueOf()`
   - `toString(radix)`
   - `toFixed(precision)` notation avec chiffres après la virgule

Utiliser Number

js-number.js

```
1  var n1=new Number(255); // ou n1=255;
2  var n2=new Number(1.5); // ou n2=1.5;
3
4  console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
5  console.log(n1.toString());    // 255
6  console.log(n1.toString(2));   // 11111111
7  console.log(n1.toString(8));   // 377
8  console.log(n1.toString(16));  // ff
9
10 n1=n1*n2;
11 console.log(n1);  // 382.5
12
13 console.log(n2.toString(16)); // 1.8
14 console.log(n2.toFixed(2));   // 1.50
```

L'objet String

String

Permet de représenter les chaînes de caractères

1. attributs :
   - `length` longueur de la chaîne
2. méthodes :
   - `integer charAt(index)`
   - `String concat(s1,s2,...)`
   - `integer indexOf(search[,fromIndex])`
   - `integer lastIndexOf(search[,fromIndex])`
   - `String slice(begin,end)` extrait la sous-chaîne
   - `Array split(separator[,limit])`
   - `String substr(start,length)` extrait la sous-chaîne
   - `String toLowerCase()`
   - `String toUpperCase()`

Utiliser String

js-string.js

```
1 console.log('lassent'.length); // 7
2 var text='les aléas de la semaine lassent';
3 console.log(text.indexOf('la')); // 13
4 console.log(text.lastIndexOf('la')); // 24
5 console.log(text.search('de')); // 10
6 console.log(text.split(' ')); // les,aléas,de,la,semaine,lassent
7 console.log(text.slice(16,-8)); // semaine
8 console.log(text.replace('semaine','journée')); // les aléas de la journée
lassent
9 console.log(text.match(/l\w+/g)); // les,la,lassent
```

L'objet `String` et les expressions régulières

Méthodes en rapport avec les expressions régulières

Permettent de fouiller et modifier des chaînes de caractères à l'aide de regex

- `array match(regex)`
- `boolean search(regex)`
- `boolean replace(regex,newstr or function)`

L'objet RegExp

Regexp

Permet de représenter les expressions régulières
Méthodes :

- `test(string)` retourne `true` si il y a correspondance
- `exec(string)` retourne un tableau qui contient les motifs correspondant à l'expression

Rappel sur les expressions régulières

Une regex suit la syntaxe`/pattern/modificateur`
L'expression du `pattern` peut utiliser des

- crochets et parenthèses
- méta-caractères
- quantificateurs

Rappel sur les expressions régulières

Modificateurs

`q` dans une regex `/pattern/q` prenant la/les valeur(s) :

- `g` : recherche toutes les correspondances (arrêt à la première par défaut)
- `i` : recherche insensible à la casse
- `m` : recherche multi-lignes

Crochets et parenthèses

Correspondance (*match*) avec un ensemble de caractères :

- `[abc]` : n'importe quel caractère parmi `{a,b,c}`
- `[0-9]` : n'importe quel chiffre
- `x|y` : l'expression `x` ou l'expression `y`

Rappel sur les expressions régulières

Méta-caractères

Exemple de méta-charactères :

- `\d` : un chiffre
- `\s` : un espace
- `\w` : un mot
- `\b` : début ou fin de mot

Quantificateurs

Exemples de quantificateurs :

- `x+` : correspondance avec au moins une occurrence de `x`
- `x?` : correspondance avec 0 ou 1 occurrence de `x`
- `x{m,n}` : correspondance avec y$\in$[m,n] répétitions de `x`

Utiliser Regexp

js-regexp.js

```
1  var regex1=/\d+/;
2  var regex2=new RegExp("(\\d+)","g");
3  var regex3=/(\d+)\/(\d+)\/(\d+)/;
4  var dob='30/09/1970';
5  console.log(dob.match(regex1)); // 30
6  console.log(dob.match(regex2)); // 30,09,1970
7  console.log(regex2); // /(\d+)/g
8  console.log(regex3); // /(\d+)\/(\d+)\/(\d+)/
9  console.log(regex3.test(dob)); // true
10 console.log(regex3.exec(dob)); // 30/09/1970,30,09,1970
11 console.log(dob.replace(regex3,'$3-$2-$1')); // 1970-09-30
```

L'objet Math global

Math - les propriétés

Permet de réaliser des calculs complexes

- `E` : constante d'Euler (2,718)
- `LN2` : logarithme népérien de 2 (0,693)
- `LN10` : logarithme népérien de 10 (2,302)
- `LOG2E` : logarithme à base 2 de E (1,442)
- `LOG10E` : logarithme à base 10 de E (0,434)
- `PI` : valeur du nombre PI (3,14159)
- `SQRT1_2` : racine carrée de 1/2 (0,707)
- `SQRT2` : racine carrée de 2 (1,414)

L'objet Math

Math - les méthodes

- `abs`
- `acos`, `asin` , `atan`, , `atan2`
- `ceil`, `floor`
- `cos`, `sin`, `tan`
- `exp`, `log`, `pow`
- `min`, `max`
- `random` : retourne une valeur entre 0 et 1
- `round`
- `sqrt`

Utiliser Math

js-math.js

```
1  console.log(Math.cos(Math.PI)); // -1
2  console.log(Math.sin(Math.PI)); //1.2246467991473532e-16
3
4  console.log(Math.sqrt(2));  //1.4142135623730951
5  console.log(Math.pow(2,10)); // 1024
6
7  console.log(Math.round(Math.sqrt(2))); // 1
8  console.log(Math.random()); //0.3946604376730234
9
10 console.log(Math.ceil(1.4));
11 console.log(Math.ceil(1.8));
12 console.log(Math.floor(1.4));
13 console.log(Math.floor(1.8));
14 console.log(Math.round(1.4));
15 console.log(Math.round(1.8));
```

L'objet Date

Date - les méthodes

Permet de représenter les dates

- `Date()`
- `Date(milliseconds)`
- `Date(y,m,d)`
- `getDay()`, `getMonth()`, `getYear()`, `getFullYear()`
- `getHours()`, `getMinutes()`, `getSeconds()`
- `toGMTString`
- `toLocaleString`
- `toUTCString`

Utiliser Date

js-date.js

```
1 var today=new Date();
2 console.log(today); //Fri Feb 13 2009 16:27:30 GMT+0100 (GMT+01:00)
3 console.log(today.toLocaleDateString()); //fevrier 13, 2009
4 console.log(today.toUTCString()); // Fri, 13 Feb 2009 15:38:52 GMT
5 var origin = new Date(0);
6 console.log(origin.toGMTString()); // Thu Jan 01 1970 01:00:01 GMT+0100
(GMT+01:00)
7 console.log(new Date(2009,0,15)); // Thu Jan 15 2009 00:00:00 GMT+0100
(GMT+01:00)
8
9 console.log(Date.parse(today.toString())); // 1234539588000
```

L'objet Array

Array - propriétés et méthodes

Permet de représenter et traiter les tableaux

- `length`
- `Array concat(array1,array2)`
- `string join(string)`
- `push` : ajoute un nouvel élément à la fin
- `unshift` : ajoute un nouvel élément au début
- `pop` : supprime et retourne le dernier élément
- `shift` : supprime et retourne le premier élément
- `reverse` : inverse l'ordre des éléments

L'objet Array

Array - autres méthodes

- `Array(n)` : construit un tableau de longueur n
- `Array(n1,...,nk)` : construit un tableau dont les éléments sont $n1, \ldots, n_k$
- `slice(begin,end)` : extrait une partie du tableau
- `splice(index,howMany,e1, ..., en)` : ajoute et supprime de nouveaux éléments
- `sort()` : tri des éléments
- `sort(f)` : tri suivant la fonction de comparaison binaire `f` retournant -1, 0 ou 1

Utiliser Array

js-array.js

```
1  var a=[3,8,2,7];
2  console.log(a.length); // 4
3  console.log(a.reverse()); // 7,2,8,3
4  a.sort(); console.log(a); // 2,3,7,8
5  var a=new Array(1,7,3,7,8,2,1);
6  for (i in a) {
7    console.log(i+' '+a[i]); // 0 1, 1 7, ....
8  }
9  console.log(a.join(';')); //1;7;3;7;8;2;1
10 var b=a.slice(2,4); console.log(b); // 3,7
11 a.splice(1,2,20);
12 console.log(a); // 1,20,7,8,2,1
13 function compare(a,b) {
14   return a-b;
15 }
16 a.sort(compare); console.log(a); // 1,1,2,7,8,20
```

Divers - Débogage

Débogage

- Logging avec `console.log(string);`
- Point d'arrêt pour débogage avec `debugger;`

Divers - Mode strict

Mode strict

La déclaration `"use strict";` en début de script/fonction lancera une exception dans les cas suivants :

- usage de variable/objet non déclaré
- destruction de fonction avec `delete`
- duplication de noms d'arguments de fonction
- écriture sur une propriété *read-only* ou *get-only*
- destruction d'une propriété indestructible
- usage de `eval` et `arguments` comme noms de variables
- création de variables via `eval` dans la portée de l'appel
- utilisation de mots-clés réservés (pour plus tard) : `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, `yield`

Recommandations

Recommandations

- Eviter les variables globales
- Déclarer les variables locales (sinon, portée globale !)
- Placer les déclarations en début de script/fonction
- Initialiser les variables
- Préférer les types primitifs à `Object`, `Number`, `String`, `Boolean`
- Attention au transtypage automatique
- Préférer les opérateurs de comparaison `===` et `!==`
- Affecter des valeurs par défaut aux arguments de fonctions
- Terminer les `switch` avec `default:`
- Eviter `eval()`

CM JS : L'API DOM

L'API DOM

Les parties importantes d'un navigateur

Trois types d'objets JS fondamentaux (alias interfaces)

`navigator`

- L'état et l'identité du navigateur (alias user-agent).
- Pour récupérer des données de géolocalisation, le langage préféré de l'utilisateur, le flux média de la webcam, ...

`window`

- La fenêtre (ou onglet) dans lequel la page est chargée.
- Pour redimensionner la fenêtre, lui associer des gestionnaires d'évènements, stocker des données spécifiques à la page ...

`document`

- Le document chargé.
- Pour obtenir une référence d'élément HTML, en changer le contenu texte, en modifier le style, pour créer/ajouter/supprimer des éléments.

L'objet window

Objet global représentant la fenêtre du navigateur

Il n'est pas nécessaire de le spécifier, sauf lorsqu'on manipule des (i)frames.

dom-window.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>L'objet window</title>
6  </head>
7  <body>
8    <script>
9          // 2 instructions équivalentes
10         alert("Hello world!");
11         window.alert("Hello world!");
12     </script>
13 </body>
14 </html>
```

L'objet window

Les fonctions globales ne sont pas des méthodes de window

`isNaN()`, `parseInt()`, `parseFloat()` ...

Une variable déclarée sans var est une propriété de window.

dom-window-1.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>L'objet window</title>
4  </head>
5  <body>
6    <script>
7      let texte = 'globale';
8      (function() { // IIFE (Immediately-Invoked Function Expression)
9        let texte = 'locale';
10       glob = "autre globale"; // déconseillé !
11       alert(window.texte); // 'globale'
12       alert(texte); // 'locale'
13     })();
14     alert(texte); // 'globale'
15     alert(glob); // 'autre globale'
16   </script>
17 </body></html>
```

Le DOM

API pour manipuler documents XML et HTML

- Offre une représentation structurée et orientée objet des éléments, de leurs attributs et de leur contenu.
- Permet l'ajout et la suppression d'objets.
- Permet la modification des propriétés de ces objets à l'aide de méthodes.
- Permet de répondre aux évènements utilisateur.

Mise en oeuvre

- API standardisée autour de JS dans tous les navigateurs.
- API dépendante du langage de scripts côté serveur pour la manipulation de documents XML
  - eg. les classes PHP `DOMDocument`, `DOMElement` etc.

Les normes du DOM

Recommandations du W3C

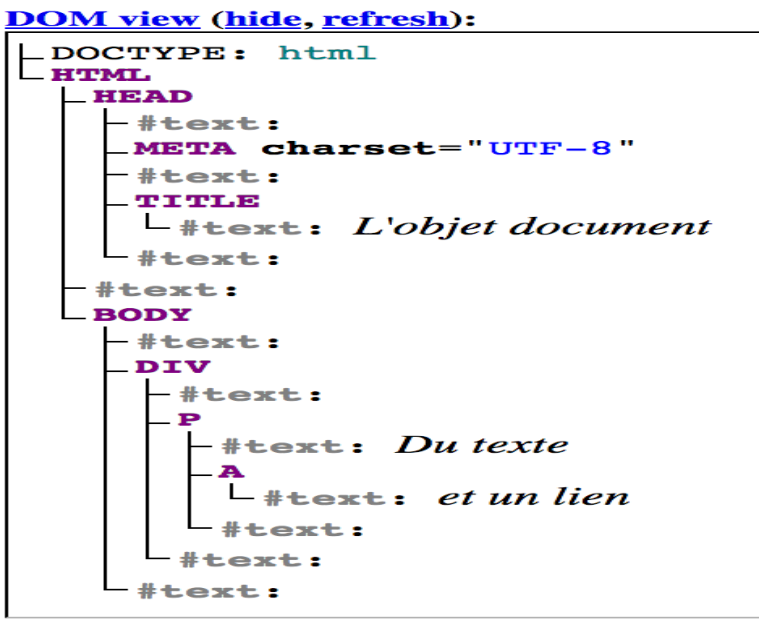
- 1998 : DOM Level 1 (Core + HTML).
- 2000 : DOM Level 2 (`getElementById`, modèle d'évènements, espaces de noms XML, CSS).
- 2004 : DOM Level 3 (support XPath, sérialisation en XML).
- 2015 : DOM Level 4 du WHATWG.
- 2015-. . . : DOM *Living Standard* du WHATWG.

Des bibliothèques permettent d'assurer la compatibilité du code JS pour différents navigateurs (eg. jQuery)

L'arbre DOM d'un document HTML

Arborescence de noeuds de différents types représentant

- Les éléments HTML.
- Leur contenu : noeud(s) texte et/ou noeud(s) élément.



dom-document.html

```
1  <!DOCTYPE html>
2  <html><head>
3  <meta charset="utf-8" />
4  <title>L'objet document</title>
5  </head>
6  <body>
7    <div>
8      <p>
9        Du texte
10       <a>et un lien</a>
11     </p>
12   </div>
13 </body></html>
```

Les attributs d'éléments sont des noeuds non-connectés.

Types de noeuds DOM

La propriété `nodeType` renvoie le type du noeud.

Constante	Valeur	Description
`Node.ELEMENT_NODE`	`1`	Un noeud `Element` tel que `<p>` ou `<div>`.
`Node.TEXT_NODE`	`3`	Le `Text` actuel de l'`Element` ou `Attr`.
`Node.PROCESSING_INSTRUCTION_NODE`	`7`	Une `ProcessingInstruction` d'un document XML tel que la déclaration `<?xml-stylesheet ... ?>`.
`Node.COMMENT_NODE`	`8`	Un noeud `Comment`.
`Node.DOCUMENT_NODE`	`9`	Un noeud `Document`.
`Node.DOCUMENT_TYPE_NODE`	`10`	Un noeud `DocumentType` c'est-à-dire `<!DOCTYPE html>` pour des documents HTML5.
`Node.DOCUMENT_FRAGMENT_NODE`	`11`	Un noeud `DocumentFragment`.

Depuis le DOM-4, suppression des constantes

2 (`ATTRIBUTE`), 4 (`CDATA_SECTION`), 5 (`ENTITY_REFERENCE`), 6 (`ENTITY`) et 12 (`NOTATION`).

Noms de noeuds DOM

La propriété `nodeName` renvoie le nom du noeud.

Interface	nodeName
Attr	identique à `Attr.name`
CDATASection	"#cdata-section"
Comment	"#comment"
Document	"#document"
DocumentFragment	"#document-fragment"
DocumentType	identique à `DocumentType.name`
Element	identique à `Element.tagName`
Entity	nom de l'entité
EntityReference	nom de la référence d'entité
Notation	nom de la notation
ProcessingInstruction	identique à `ProcessingInstruction.target`
Text	"#text"

Types et noms de noeuds DOM

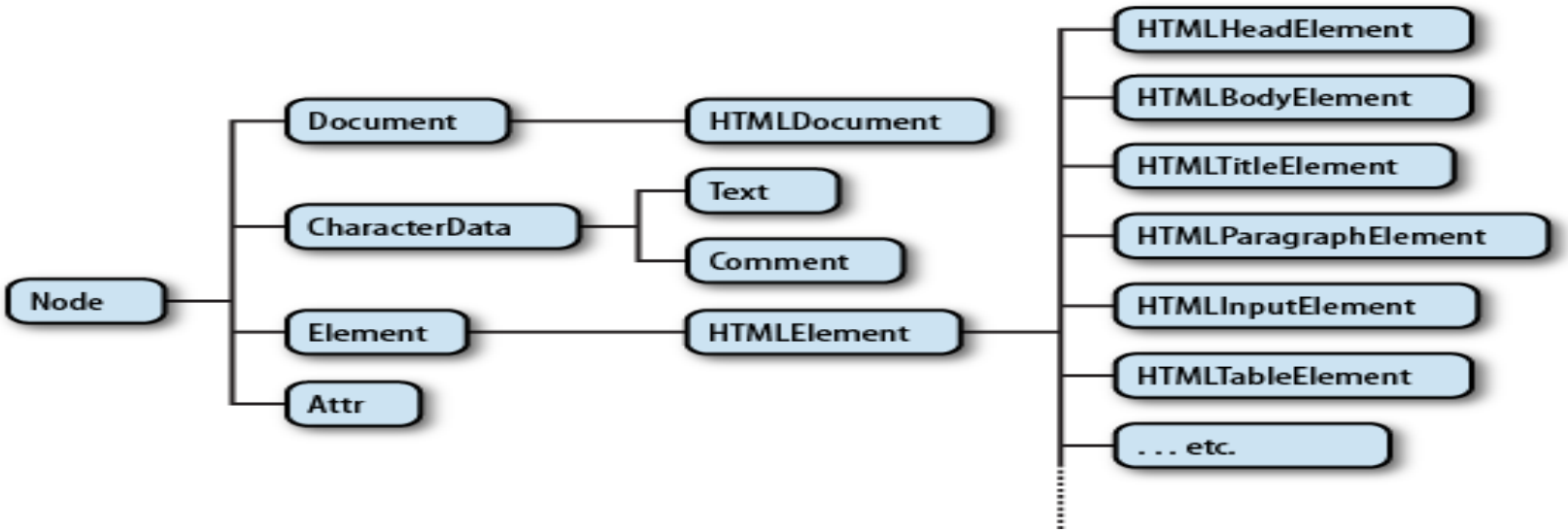
dom-nodetype.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Les propriétés nodeType/nodeName</title>
</head>
<body>
  <div id="myDiv">Du texte.</div>
  <script>
        let myDiv = document.getElementById("myDiv");
        alert(myDiv.nodeType); // 1
        alert(myDiv.nodeName); // DIV
        let myDivId = myDiv.getAttributeNode("id");
        alert(myDivId.nodeType); // 2
        alert(myDivId.nodeName); // ID
        let myDivText = myDiv.firstChild;
        alert(myDivText.nodeType); // 3
        alert(myDivText.nodeName); // #text
    </script>
</body>
</html>
```

Interfaces DOM

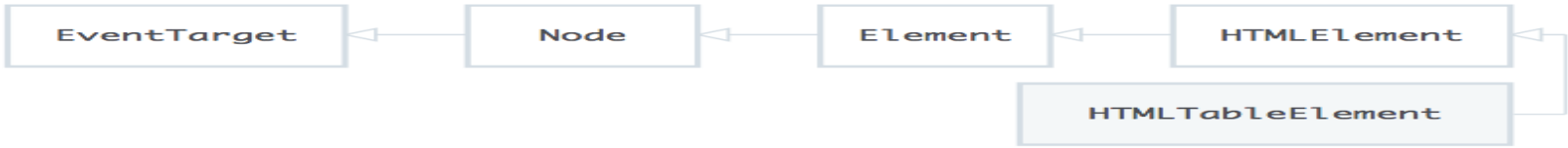
JS fournit une hiérarchie d'interfaces

Propriétés et méthodes auxquelles se conforment les différents noeuds d'un DOM selon leur type.



Chaque objet modélisant un noeud du DOM implémente les propriétés et méthodes de sa chaîne d'interfaces.

Héritage d'interfaces : exemple



dom-interfaces.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8"/>
3  <title>Interfaces DOM</title>
4  </head>
5  <body>
6    <table id="myTable">
7      <tr><td>A1</td><td>A2</td></tr>
8      <tr><td>B1</td><td>B2</td></tr>
9    </table>
10   <script>
11   let table = document.querySelector('#myTable');
12   // HTMLTableElement interface : attribut rows
13   table.rows[0].style.backgroundColor = "red";
14   // HTMLElement interface : attribut title
15   table.title = "Une aide";
16   // Element interface : attribut class
17   table.className = "mesTables";
18   // Node interface : attribut baseURI
19   alert('Base URL de la table :' + table.baseURI);
20   </script>
21 </body></html>
```

Les types d'objets importants

`document` (interface `Document`)

- L'objet correspondant au noeud racine du DOM.

`element` (interface `Element`).

- Un objet correspondant à un noeud du DOM.

`nodeList` (interface `NodeList`).

- Une collection d'`elements` statique ou dynamique.
- Traversable avec `for`, `for...of` et méthode `forEach`.
- Convertible en tableau avec `Array.from()`.

`attribute` (interface `Attr`).

- Un objet correspondant à un attribut d'`element`.

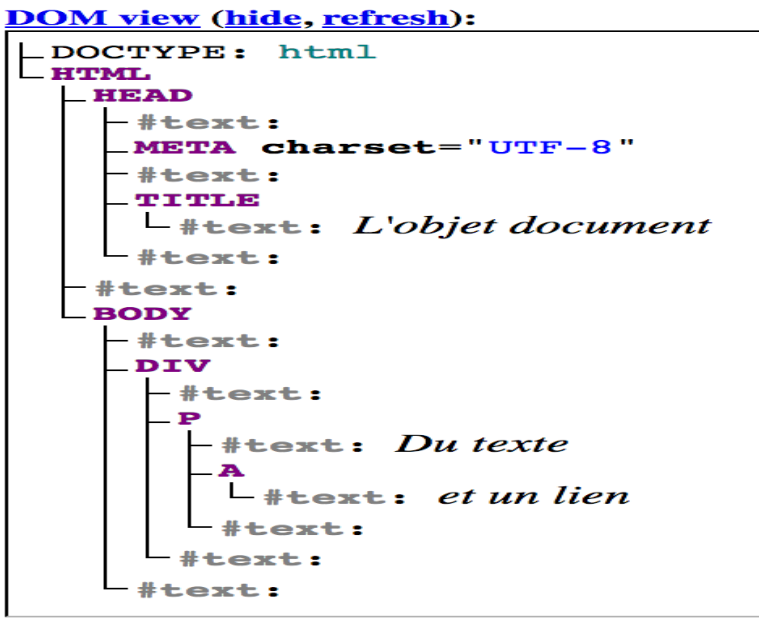
`namedNodeMap` (interface `NamedNodeMap`).

- Une collection dynamique d'`attributes`.
- Traversable avec `for`.
- Convertible en tableau avec `Array.from()`.

L'objet document

Est un objet propriété de `window`

- Représente le noeud racine du DOM.
- Parent de l'élément `HTML`.
- Consulter l'outil `DOM` de Firefox ou Live DOM Viewer.



dom-document.html

```
1  <!DOCTYPE html>
2  <html><head>
3  <meta charset="utf-8" />
4  <title>L'objet document</title>
5  </head>
6  <body>
7    <div>
8      <p>
9        Du texte
10       <a>et un lien</a>
11     </p>
12   </div>
13 </body></html>
```

Accès aux éléments HTML

Avec les méthodes d'objets `Document` ou `Element` :

`getElementById(identifiant)`

- Renvoie l'élément d'attribut `id` égal à `identifiant`.

`getElementsByTagName(balise)`

- Renvoie le tableau (`HTMLCollection`) des éléments de balise `balise`.

`getElementsByName(nom)`

- Renvoie le tableau (`HTMLCollection`) des éléments de formulaires (HTML 5) d'attribut `name` égal à `nom`.

`querySelector(sélecteur)`

- Renvoie le 1er élément correspondant au `sélecteur` CSS.

`querySelectorAll(sélecteur)`

- Renvoie le tableau d'éléments (`HTMLCollection`) correspondant au `sélecteur` CSS.

Accès aux éléments HTML

dom-getelements.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Accès aux éléments HTML</title>
</head>
<body>
  <div id="myDiv">
    <p>
      Du texte <a>et un lien</a>
    </p>
  </div>
  <div>
    <p>Que du texte
  </div>
  <script>
  let div = document.getElementById("myDiv");
  console.debug(div);
  let divs = document.getElementsByTagName("div");
  let i=1;
  Array.from(divs).forEach(function(div) {
    alert("Element n\u{00B0}"+(i++)+" : "+div.innerHTML);
  });
  for(let div of divs) {
    console.debug("Element n\u{00B0}"+" : "+div.textContent);
  }
  </script>
</body>
</html>
```

Accès aux éléments HTML avec sélecteurs CSS3

Séquence	Signification
*	tout élément
E	tout élément de type E
E[foo]	tout élément E portant l'attribut "foo"
E[foo="bar"]	tout élément E portant l'attribut" foo" et dont la valeur de cet attribut est exactement "bar"
E[foo~="bar"]	tout élément E dont l'attribut "foo" contient une liste de valeurs séparées par des espaces, l'une de ces valeurs étant exactement égale à "bar"
E[foo^="bar"]	tout élément E dont la valeur de l'attribut "foo" commence exactement par la chaîne "bar"
E[foo$="bar"]	tout élément E dont la valeur de l'attribut "foo" finit exactement par la chaîne "bar"
E[foo*="bar"]	tout élément E dont la valeur de l'attribut "foo" contient la sous-chaîne "bar"
E[lang|="en"]	tout élément E dont l'attribut 'lang" est une liste de valeurs séparées par des tirets et commençant (à gauche) par "en"
E:root	un élément E, racine du document
E:nth-child(n)	un élément E qui est le n-ième enfant de son parent
E:nth-last-child(n)	un élément E qui est le n-ième enfant de son parent en comptant depuis le dernier enfant
E:nth-of-type(n)	un élément E qui est le n-ième enfant de son parent et de ce type
E:nth-last-of-type(n)	un élément E qui est le n-ième enfant de son parent et de ce type en comptant depuis le dernier enfant
E:first-child	un élément E, premier enfant de son parent
E:last-child	un élément E, dernier enfant de son parent
E:first-of-type	un élément E, premier enfant de son type
E:last-of-type	un élément E, dernier enfant de son type
E:only-child	un élément E, seul enfant de son parent
E:only-of-type	un élément E, seul enfant de son type

Accès aux éléments HTML avec sélecteurs CSS3

`E:empty`	un élément E qui n'a aucun enfant (y compris noeuds textuels purs)
`E:link` `E:visited`	un élément E qui est la source d'un hyperlien dont la cible n'a pas encore été visitée (:link) ou a déjà été visitée (:visited)

`E:active`
`E:hover`
`E:focus`
	un élément E pendant certaines actions de l'usager

`E:target`	un élément E qui est la cible de l'URL d'origine contenant lui-même un fragment identifiant.
`E:lang(c)`	un élément E dont le langage (humain) est c (le langage du document spécifie comment le langage humain est déterminé)
`E:enabled` `E:disabled`	un élément d'interface utilisateur E qui est actif ou inactif.
`E:checked` `E:indeterminate`	un élément d'interface utilisateur E qui est coché ou dont l'état est indéterminé (par exemple un bouton-radio ou une case à cocher)
`E:contains("foo")`	un élément E dont le contenu textuel concaténé contient la sous-chaîne "foo"
`E::first-line`	la première ligne formatée d'un élément E
`E::first-letter`	le premier caractère formaté d'un élément E
`E::selection`	la partie d'un élément E qui est actuellement sélectionnée/mise en exergue par l'usager
`E::before`	le contenu généré avant un élément E
`E::after`	le contenu généré après un élément E
`E.warning`	*Uniquement en HTML*. Identique à E[class~="warning"].
`E#myid`	un élément E dont l'ID est égal à "myid".
`E:not(s)`	un élément E qui n'est pas représenté par le sélecteur simple s
`E F`	un élément F qui est le descendant d'un élément E
`E > F`	un élément F qui est le fils d'un élément E
`E + F`	un élément F immédiatement précédé par un élément E
`E ~ F`	un élément F précédé par un élément E

Accès aux éléments HTML avec sélecteur CSS

dom-selecteur.html

```
<!DOCTYPE html>
<html><head><meta charset="utf-8" />
<title>Accès aux éléments HTML avec sélecteurs CSS</title>
</head>
<body>
  <div id="menu">
    <div class="item">
      <span>Elément 1</span>
      <span>Elément 2</span>
    </div>
    <div class="publicité">
      <span>Elément 3</span>
      <span>Elément 4</span>
    </div>
  </div>
  <script>
    let query = document.querySelector("#menu .item span");
    let queryAll = document.querySelectorAll("#menu .item span");
    alert(query.innerHTML); // Elément 1
    alert(queryAll.length); // 2
    // Elément 1 - Elément 2
    alert(queryAll[0].innerHTML + '-' + queryAll[1].innerHTML);
  </script>
</body></html>
```

Accès aux attributs d'éléments HTML

Avec les méthodes d'objets `Element` :

`getAttribute(att)`

- Renvoie la valeur (`string`) de l'attribut de nom `att`.

`setAttribute(att,val)`

- Fixe à `val` la valeur de l'attribut de nom `att`.

dom-attributs-acces.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Accès aux attributs d'éléments HTML</title>
4  </head>
5  <body>
6    <a id="myLink" href="http://www.anywhere.com">
7      Un lien modifié dynamiquement.
8    </a>
9    <script>
10     let myLink = document.querySelector("#myLink");
11     let href = myLink.getAttribute("href");
12     alert(href);
13     myLink.setAttribute("href","http://es6-features.org");
14   </script>
15 </body></html>
```

Accès direct aux attributs d'éléments HTML

Par une propriété d'objets `Element` de même nom que l'attribut

dom-attributs-acces-direct-1.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>Accès direct aux attributs d'éléments HTML</title>
6  </head>
7  <body>
8    <a id="myLink" href="">un lien</a>
9    <script>
10         let myLink = document.querySelector("#myLink");
11         let href = myLink.href;
12         alert(href);
13         myLink.href = "http://es6-features.org";
14     </script>
15 </body>
16 </html>
```

Accès direct aux attributs d'éléments HTML

CamelCase utilisée pour les propriétés correspondant aux

- Attributs à nom composé (eg. `readonly`) ou comportant un tiret (eg. `background-color`).
- Attributs dont le nom est réservé en JS : `class` (`className` et `classList`), `for` (`htmlFor`).

dom-attributs-acces-direct-2.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Accès direct aux attributs d'éléments HTML</title>
4  <style>
5  .bleu { background:lightblue;} .rouge { color:red;} .noir { color:black;}
6  </style>
7  </head>
8  <body>
9    <a id="myLink" class="rouge" href="">Un lien</a>
10   <script>
11     let myLink = document.querySelector("#myLink");
12     alert("Prêt ?");
13     myLink.style.textTransform = "uppercase";
14     myLink.classList.add("bleu");
15     myLink.classList.remove("rouge");
16     myLink.classList.toggle("noir");
17   </script>
18 </body></html>
```

Récupérer le code HTML

Sous forme de chaîne avec propriété `Element.innerHTML`

dom-innerhtml.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>Accès au code HTML</title>
6  </head>
7  <body>
8    <div id="myDiv">
9      <p>
10       Du texte <a href="#myDiv">et un lien</a>
11     </p>
12   </div>
13    <p>Autre texte</p>
14   <script>
15         let div = document.querySelector("#myDiv");
16         alert(div.innerHTML);
17     </script>
18 </body>
19 </html>
```

Ajouter ou éditer du code HTML

Avec `Element.innerHTML`

dom-innerhtml-1.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>Edition/ajout de code HTML</title>
6  </head>
7  <body>
8    <div id="myDiv">
9      <p>
10       Du texte <a>et un lien</a>
11     </p>
12   </div>
13   <script>
14         let div = document.querySelector("#myDiv");
15         div.innerHTML = "<h1>Un titre écrase le paragraphe</h1>";
16         div.innerHTML += "<h1>Et un second titre en ajout</h1>";
17     </script>
18 </body>
19 </html>
```

Ajouter ou éditer le contenu textuel d'éléments

Avec `Element.textContent`

dom-textcontent.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Edition/ajout de code HTML</title>
</head>
<body>
  <div id="myDiv">
    <p>
      Du texte <a>et un lien</a>
    </p>
  </div>
  <script>
        let div = document.querySelector("#myDiv");
        div.textContent = "ABC";
        div.innerHTML += "<h1>Et un second titre en ajout</h1>";
    </script>
</body>
</html>
```

Naviguer dans le DOM : parent

La propriété `Node.parentNode` permet d'accéder au parent d'un noeud

dom-parent.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="utf-8" />
5  <title>La propriété parentNode</title>
6  </head>
7  <body>
8    <blockquote>
9      <p id="myP">Un paragraphe.</p>
10   </blockquote>
11   <script>
12         let bq = document.querySelector("#myP").parentNode;
13         alert(bq); // object HTMLQuoteElement
14     </script>
15 </body>
16 </html>
```

Naviguer dans le DOM : enfants

`Element.firstChild` et `lastChild` donnent accès aux premier et dernier enfants d'un noeud

dom-firstchild.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8"/>
3  <title>Création d'éléments</title>
4  <link rel="stylesheet" href="dom.css"></head>
5  <body>
6    <div>
7      Voici <p id="myP">du texte <a>et un lien</a></p>
8    </div>
9    <script>
10   let div = document.querySelector("div");
11   alert(div.firstChild.nodeName); // #text
12   alert(div.firstElementChild.nodeName); // P !!
13   </script>
14 </body></html>
```

`firstElementChild` et `lastElementChild` donnent accès aux premier et dernier enfants qui sont des éléments HTML

Naviguer dans le DOM : contenu texte

`Node.nodeValue` et `CharacterData.data` donnent le contenu texte d'un noeud textuel

dom-nodevalue.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Les propriétés nodeValue/data</title>
4  </head>
5  <body>
6    <div>
7      <p id="myP">Du texte <a>un lien</a> et <strong>en emphase</strong></p>
8    </div>
9    <script>
10   let myP = document.querySelector("#myP");
11   alert(myP.firstChild.nodeValue); // Du texte
12   alert(myP.lastChild.firstChild.data); // en emphase
13   alert(myP.nodeValue); // null
14   </script>
15 </body></html>
```

Naviguer dans le DOM : tableau des enfants

`Node.childNodes` renvoie le tableau des enfants d'un noeud
`Node.children` renvoie le tableau des éléments enfants d'un noeud

dom-childnodes.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <title>Les propriétés childNodes et children</title>
6  </head>
7  <body>
8      <div>
9          <p id="myP">Du texte <a>un lien</a></p>
10     </div>
11     <script>
12         let myP = document.querySelector("#myP");
13         for (child of myP.childNodes) {
14             if (child.nodeType === Node.ELEMENT_NODE) {
15                 alert("Noeud element : " + child.firstChild.data); // un lien
16             } else {
17                 alert("Noeud texte : " + child.data); // du texte
18             }
19         }
20         console.log(myP.children); // HTMLCollection { 0: a, length: 1 }
21     </script>
22 </body>
23 </html>
```

Naviguer dans le DOM : adelphes

Node.nextSibling et previousSibling donnent accès au noeud suivant et précédent

dom-nextsibling.html

```
<!DOCTYPE html>
<html><head><meta charset="utf-8" />
<title>Les propriétés nextSibling/previousSibling</title>
</head>
<body>
  <div>
    <p id="myP">Du texte <a>un lien</a></p>
  </div>
  <script>
    let child = document.querySelector("#myP").lastChild;
    while (child) {
      if (child.nodeType === 1) {
        alert(child.firstChild.data); // élément HTML
      } else {
        alert(child.data); // noeud texte
      }
      child = child.previousSibling;
    }
  </script>
</body></html>
```

nextElementSibling et previousElementSibling donnent accès aux éléments HTML suivant et précédent

Naviguer dans le DOM

Attention aux noeuds texte “vides” (espaces, CR . . .) !

dom-noeuds-texte-vides.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Les noeuds texte vides : retours à la ligne, espaces ...</title>
4  </head>
5  <body>
6    <div>
7      <p id="myP1"><a>Une ancre</a></p>
8      <p id="myP2">
9        <a>Une ancre</a>
10     </p>
11   </div>
12   <script>
13     let child1 = document.querySelectorAll("p")[0].firstChild;
14     let child2 = document.querySelectorAll("p")[1].firstChild;
15     if (child1.nodeType !== child2.nodeType) {
16       alert(child2.nodeType); // 3 (Node.TEXT_NODE)
17     }
18   </script>
19 </body></html>
```

Créer et insérer des éléments

En 3 temps

1. Création d'un élément avec la méthode `Document.createElement(tag)`.
2. Affectation des attributs avec la méthode `Element.setAttribute(att,val)`.
3. Insertion de l'élément dans le document avec l'une des méthodes :
   - `Node.appendChild(tag)`
   - `Node.insertBefore(node,referenceNode)`

Création d'un noeud texte avec la méthode `Document.createTextNode(contenuTexte)`.

Créer et insérer des éléments

dom-createelement.html

```
<!DOCTYPE html>
<html><head><meta charset="utf-8" />
<title>Création et ajout d'éléments</title>
</head>
<body>
  <div>
    <p id="myP">Du texte </p>
  </div>
  <script>
    let myP = document.querySelector("#myP");
    let newLink = document.createElement("a");
    alert(newLink.isConnected); //false
    newLink.id = "myLink";
    newLink.href = "http://www.lemonde.fr";
    newLink.title = "Le Monde";
    newLinkText = document.createTextNode("Le site du Monde");
    newLink.appendChild(newLinkText);
    myP.appendChild(newLink);
    alert(newLink.isConnected); //true
  </script>
</body></html>
```

Cloner des éléments

Avec `Node.cloneNode(flag)`

- Si `flag=true`, clone aussi enfants et attributs du noeud.
- Si `flag=false`, ne clone ni enfants ni attributs.

Les gestionnaires d'évènement enregistrés ne sont pas clonés.

dom-clonenode.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Clonage de noeuds</title>
4  </head>
5  <body>
6    <div>
7      <p id="myP">Du texte </p>
8    </div>
9    <script>
10   let myP = document.querySelector("#myP");
11   let p1 = myP.cloneNode(true);
12   let p2 = myP.cloneNode(false);
13   document.querySelectorAll("body")[0].appendChild(p1);
14   document.querySelectorAll("body")[0].appendChild(p2);
15   </script>
16 </body></html>
```

Remplacer des éléments

Avec `Node.replaceChild(newChild,oldChild)`

Remplace l'enfant `oldChild` par `newChild`.

dom-replacechild.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Remplacement de noeud enfant</title>
4  </head>
5  <body>
6    <div><p>Du texte</p></div>
7    <script>
8    let myDiv = document.querySelector("div");
9    let myS = document.createElement("span");
10   myS.innerHTML = "Un &lt;span&gt; en remplacement d'un &lt;p&gt;"
11   myDiv.replaceChild(myS,myDiv.firstChild);
12   </script>
13 </body></html>
```

Supprimer des éléments

Avec `Node.removeChild()` sur noeud parent

dom-removechild.html

```
 1 <!DOCTYPE html>
 2 <html><head><meta charset="utf-8" />
 3 <title>Suppression de noeud enfant</title>
 4 </head>
 5 <body>
 6   <div><p>Du texte</p></div>
 7   <script>
 8     let myP = document.querySelector("div > p");
 9     myP.parentNode.removeChild(myP);
10     /* alternative
11   let myDiv = document.querySelector("div");
12   myDiv.removeChild(myP);
13   */
14  </script>
15 </body></html>
```

Insérer des éléments

Avec `Node.insertBefore()` sur noeud adelphe

dom-insertbefore.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Insertion d'éléments</title>
4  </head>
5  <body>
6    <div><p>Du texte</p></div>
7    <script>
8      let myDiv = document.querySelector("div");
9      let newP = document.createElement("p");
10     newPText = document.createTextNode("Voici ");
11     newP.appendChild(newPText);
12     if(myDiv.hasChildNodes)
13       myDiv.insertBefore(newP, myDiv.lastChild);
14   </script>
15 </body></html>
```

CM JS : Manipuler le CSS

Manipuler le CSS

Cascading Style Sheets (CSS)

Langage de feuilles de style

Pour décrire la présentation d'un document écrit dans un langage de balisage (HTML, XHTML, SVG, XML).

Objectifs

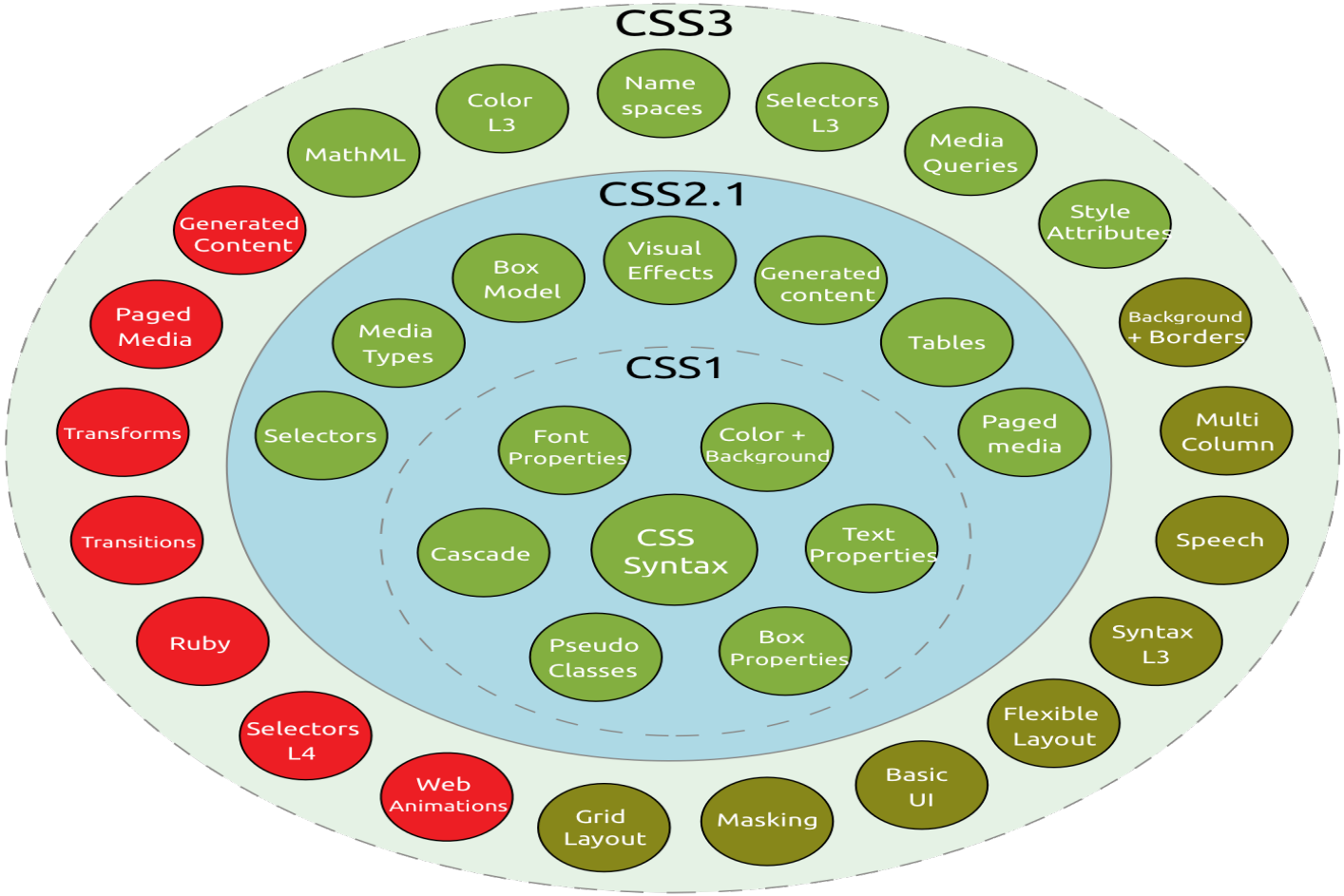
- Séparer la forme du contenu : mise en page, couleurs, polices ...
- Apporter plus de flexibilité et de contrôle sur la présentation.
- Permettre la réutilisation de fichiers CSS de mise en forme dans différentes pages HTML.
- Réduire la complexité des pages HTML (eg. répétition).
- Adapter la présentation aux capacités du terminal et à différents rendus (écran, impression, ...).

Historique

Les versions CSS

- CSS 1 : 1996
  - Polices, couleurs, espacements, alignements, marges ...
  - Statut : Obsolète.
- CSS 2 : 1998-2011
  - Positionnement, types média ...
  - Statut : Recommandation.
- CSS 3 : 1999-...
  - Décomposition en ~50 modules.
  - Statut : Normalisation en cours de chaque module.

De CC 1 à CSS 3

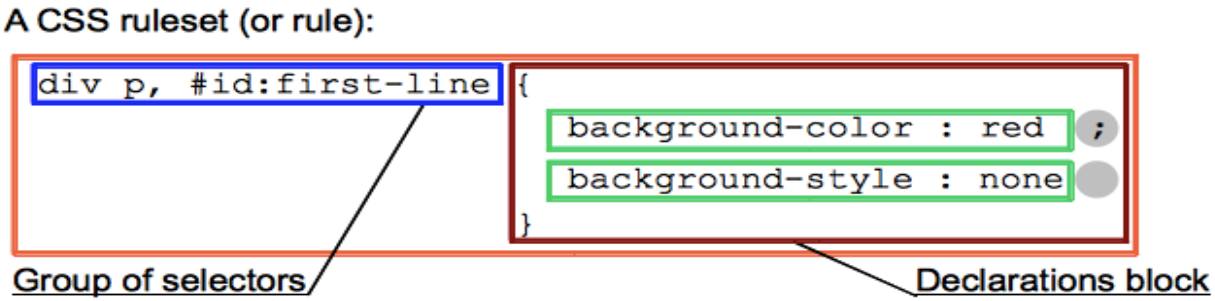


By Krauss - Own work, CC BY-SA 4.0, https ://commons.wikimedia.org/w/index.php ?curid=44954967

Déclarations, blocs et règles CSS

Déclaration = paire `propriété:valeur`

- Propriétés et valeurs sont insensibles à la casse.
- Toute déclaration invalide est ignorée (eg. `font-size:red`).



Bloc = liste entre {} de déclarations séparées par des ;

Règle = sélecteur(s) précédant un bloc de déclarations

- Le moteur de rendu CSS appliquera les déclarations du bloc aux éléments satisfaisant le groupe de sélecteurs.
- Toute règle comportant un sélecteur invalide est ignorée.

Instruction CSS

Une règle ou une @-règle (@-rule)

Différents types de @-règles :

- Méta-données relatives au fichier, eg. `@charset`, `@import`.
- Règles conditionnelles, eg. `@media`, `@document`.
- Règles descriptives, eg. `@font-face`.

Chaque @-règle a sa propre syntaxe et sémantique.

Editer le CSS d'éléments

Avec

- L'attribut `style` d'un élément HTML donné.
- Des règles placées dans un élément `style` de l'en-tête du document HTML.
- Des règles placées dans un fichier CSS importé dans le document HTML.

Editer le CSS d'éléments

css-feuille.css

```
1 @charset "UTF-8";
2
3 div {
4   margin: auto;
5   border: solid 1px green;
6 }
```

css-application.html

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5     <meta charset="utf-8" />
6     <title>Editer le CSS d'éléments HTML</title>
7     <link rel="stylesheet" href="css-feuille.css">
8 </head>
9 <style>
10     div {
11         background-color: orange;
12     }
13 </style>
14 </head>
15
16 <body>
17     <div style="text-align: center;">Je suis un DIV.</div>
18 </body>
19
20 </html>
```

Lecture de propriété CSS en JS

Avec `window.getComputedStyle(element)` qui renvoie en lecture seule un objet contenant toutes les propriétés CSS de l'élément après application des règles (importées ou non)

css-getcomputedstyle.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Accès à propriété CSS avec getComputedStyle</title>
7      <link rel="stylesheet" href="css-feuille.css">
8      <style>
9          div {
10             color: orange;
11         }
12     </style>
13 </head>
14
15 <body>
16     <div style="font-style:italic">green border, orange, italic</div>
17     <script>
18         let div = document.querySelector('div');
19         alert(getComputedStyle(div).border); //0.55px solid rgb(0, 128, 0)
20         alert(getComputedStyle(div).color); //rgb(255, 165, 0)
21         alert(getComputedStyle(div).fontStyle); //italic
22     </script>
23 </body>
24
```

Ecriture de propriété CSS en JS

Avec `element.style.propriété`

- CamelCase pour propriété CSS comportant un tiret.
- Ne pas utiliser pour lire la propriété CSS calculée car n'intègre pas les règles importées !

Ecriture de propriété CSS en JS

css-style-vs-feuille.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Accès à propriété CSS via attribut style d'éléments HTML</title>
7      <link rel="stylesheet" href="css-feuille.css">
8      <style>
9          div {
10             color: orange;
11         }
12     </style>
13 </head>
14
15 <body>
16     <div style="font-style:italic">green border, orange, italic</div>
17     <script>
18         let div = document.querySelector('div');
19         alert(div.style.border); // rien !
20         alert(div.style.color); // rien !
21         alert(div.style.fontStyle); //italic
22         div.style.color = "blue";
23         alert(div.style.color); //blue
24     </script>
25 </body>
26
27 </html>
```

Le chemin critique du rendu

Le chemin critique du rendu

Les différentes étapes pour afficher la page web à partir du document HTML renvoyé et des différentes ressources nécessaires (CSS, scripts JS, images, ...)

1. La construction de l'arborescence du DOM.
2. La construction de l'arborescence du CSSOM.
3. L'exécution du code JS.
4. La construction de l'arbre de rendu.
5. La génération de la mise en page.
6. La conversion du contenu visible final de la page en pixels.

Le contenu peut apparaître avant chargement complet du document HTML

Arbre de rendu, mise en page, fenêtre active

L'arbre de rendu

- Représente ce qui va être affiché sur la page.
- Combine DOM et CSSOM.

La mise en page (layout)

- Détermine la taille de la fenêtre active (viewport).
- Pour pouvoir appliquer les styles CSS utilisant des unités en % ou en viewport.

Une fois la mise en page générée, l'arbre de rendu est converti en pixels pour affichage

Fenêtre active (viewport)

Le viewport

Représente la partie visible d'une page web

- Sa taille dépend de l'appareil et de la taille de l'écran de l'utilisateur.
- Les smartphones utilisent une taille $>$ écran pour éviter le dézoomage.

On peut le recadrer (taille et/ou échelle) avec la balise `meta` :

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0, user-scalable=no">
```

Ralentissement du rendu

Points de blocage

Le CSS bloque le rendu et les scripts.

- L'héritage en cascade impose de l'analyser complètement.
- Les scripts JS doivent attendre la construction du CSSOM.

Les scripts JS bloquent l'analyseur HTML.

- Doit attendre le chargement et l'exécution de chaque `script`

Utiliser les attributs `async` et `defer` des éléments `<script>`

- `async` : charge et exécute le fichier de manière asynchrone.
- `defer` : exécute le fichier une fois le code HTML analysé. Permet de séquencer l'exécution de fichiers JS selon l'ordre d'inclusion.

CM JS : Gestion d'évènements

Gestion d'évènements

Les évènements

Sont

- Déclenchés par l'utilisateur (clic souris, frappe au clavier, etc.) ou générés par une API représentant la progession d'une tâche asynchrone (minuteur, réponse HTTP, fin d'une vidéo, etc).
- Communiqués au code source lorsqu'ils surviennent.

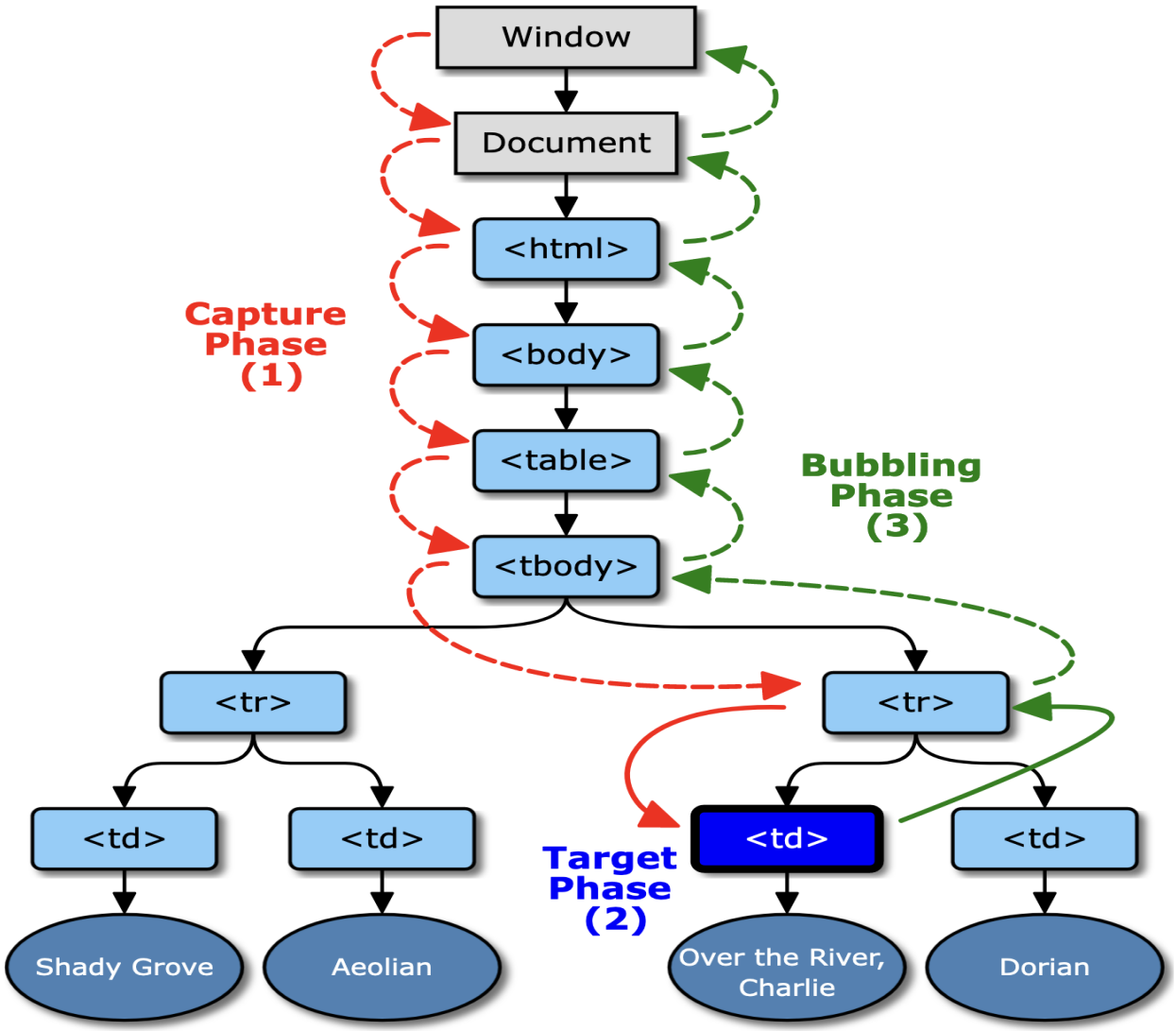
Plusieurs types d'évènements

- Standard (normalisés dans le DOM), non-standard, ou spécifique à un type de navigateur.
- Animation, batterie, appel, CSS, base de données, document, glisser-déposer, élément, focus, formulaire, frame, entrée (clavier, pad, souris), média (audio, vidéo), menu, réseau, notification, popup, impression, ressource, script, capteurs (compas, lampe, gyroscope, etc), session, carte, SMS, SVG, table, texte, touché, mise à jour, champ, vue, websocket, fenêtre, ...

Principes de la gestion d'évènements sur le DOM

- Un évènement a pour cible initiale un élément HTML.
- Un évènement se propage à tous les éléments situés sur la branche du DOM reliant la racine `window` à la cible en trois phases :
  1. Capture (*capture*) : de la racine à la cible.
  2. Cible (*target*) : sur l'élément cible.
  3. Bouillonnement (*bubbling*) : de la cible à la racine.
- Un écouteur (*listener*) intercepte un type d'évènements sur un élément dans une phase choisie en exécutant une fonction de rappel.
- L'enregistrement et le désenregistrement d'un écouteur se programme : objet cible, évènements écoutés, phase d'interception, fonction de rappel.

Phases de propagation



Les interfaces d'évènements

Tout évènement est un objet implémentant l'interface `Event` ou une sous-interface (p. ex. `MouseEvent` hérite de `UIEvent` qui hérite de `Event`)

`Event` fournit une information contextuelle aux écouteurs :

- `type` : son type (p. ex. `click`).
- `target` : son objet cible initial.
- `currentTarget` : l'objet courant sur lequel il est propagé.
- `bubbles` : s'il est en phase de bouillonnement ou de capture.
- `timestamp` : son horodatage relatif à l'epoch, etc.

Les sous-interfaces fournissent des informations supplémentaires

Par ex., `MouseEvent` a des propriétés relatives à la position de la souris, pression et relâchement du bouton, la molette, etc.

Les principaux évènements du DOM

Nom de l'événement	Action pour le déclencher
`click`	Cliquer (appuyer puis relâcher) sur l'élément
`dblclick`	Double-cliquer sur l'élément
`mouseover`	Faire entrer le curseur sur l'élément
`mouseout`	Faire sortir le curseur de l'élément
`mousedown`	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
`mouseup`	Relâcher le bouton gauche de la souris sur l'élément
`mousemove`	Faire déplacer le curseur sur l'élément
`keydown`	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
`keyup`	Relâcher une touche de clavier sur l'élément
`keypress`	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément
`focus`	« Cibler » l'élément
`blur`	Annuler le « ciblage » de l'élément
`change`	Changer la valeur d'un élément spécifique aux formulaires (`input` , `checkbox` , etc.)
`input`	Taper un caractère dans un champ de texte (son support n'est pas complet sur tous les navigateurs)
`select`	Sélectionner le contenu d'un champ de texte (`input` , `textarea` , etc.)

`submit`	Envoyer le formulaire
`reset`	Réinitialiser le formulaire

Enregistrement d'écouteurs (DOM-2)

En invoquant `addEventListener(type,listener[,...])` sur tout objet implémentant `EventTarget`

- `type` : le type d'évènements à écouter (`string`).
- `listener` : la fonction à exécuter (ou objet implémentant `EventListener`) dont l'unique argument est l'objet évènement.

2 variantes pour l'argument optionnel :

- un booléen - faux par défaut - qui donne la priorité d'écoute à l'objet cible courant (= `this`) sur ses descendants dans le DOM.
- un objet à propriétés booléennes - fausses par défaut - :
  - `capture` : priorité d'écoute à l'objet cible courant (= `this`) sur ses descendants dans le DOM.
  - `once` : unique notification de l'écouteur suivie de sa suppression automatique.
  - `passive` : interdiction pour l'écouteur de bloquer l'action par défaut de l'évènement.

Enregistrement d'écouteurs

evenement-addeventlistener.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Enregistrement d'écouteurs</title>
7  </head>
8
9  <body>
10     <span id="clickme">Cliquez-moi !</span>
11     <script>
12         var element = document.querySelector('#clickme');
13         element.addEventListener('click', function() {
14             alert("Vous m'avez cliqué !");
15         }, false);
16         element.addEventListener('click', function() {
17             alert("Je confirme : vous m'avez bien cliqué !");
18         }, false);
19     </script>
20 </body>
21
22 </html>
```

Accès à l'objet cible initial

evenement-addeventlistener-2.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Accès à cible initiale</title>
7  </head>
8
9  <body>
10     <span id="clickme">Cliquez-moi !</span>
11     <script>
12         var span = document.querySelector('#clickme');
13         span.addEventListener('click', function(e) {
14             e.target.innerHTML = "Vous avez cliqué (1) ! ";
15             this.innerHTML += "Vous avez cliqué (2) ! ";
16             console.log(e.target === this); // true
17         }, false);
18     </script>
19 </body>
20
21 </html>
```

Accès à l'objet cible courant

Ne pas confondre cible initiale et cible courante

evenement-addeventlistener-3.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Accès à cibles initiale et courante</title>
7  </head>
8
9  <body>
10     <span id="clickme">Cliquez-moi !</span>
11     <script>
12         var body = document.querySelector('body');
13         body.addEventListener('click', function(e) {
14             console.log("clic sur " + e.target.nodeName); // sur SPAN (1)
15             console.log("propagation sur " + e.currentTarget.nodeName); // sur BODY (2)
16             console.log(e.currentTarget === e.target); // false
17             console.log(e.currentTarget === this); // true
18         }, false); // interception en phase de bouillonnement
19     </script>
20 </body>
21
22 </html>
```

Capture vs. bouillonnement

evenement-addeventlistener-4.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Capture et bouillonnement</title>
7  </head>
8
9  <body>
10     <span id="clickme">Cliquez-moi !</span>
11     <script>
12         var body = document.querySelector('body');
13         var span = document.querySelector('#clickme');
14         body.addEventListener('click', function(e) {
15             console.log("L'évènement traverse  " + e.currentTarget.nodeName);
16         }, true); // interception en phase de capture
17         span.addEventListener('click', function(e) {
18             console.log("L'évènement traverse le " + e.currentTarget.nodeName);
19         }, true); // valeur indifférente ici : interception en phase cible
20     </script>
21 </body>
22
23 </html>
```

Gestionnaires d'évènements souris (1)

evenement-souris.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Evènements souris : mousemove</title>
</head>

<body>
    <div id="position"></div>
    <script>
        var position = document.getElementById('position');
        document.addEventListener('mousemove', function(e) {
            position.innerHTML = ' X = ' + e.clientX + 'px Y = ' + e.clientY + 'px';
        }, false);
    </script>
</body>

</html>
```

Gestionnaires d'évènements souris (2)

evenement-souris-1.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Evènements souris : mouseover, mouseout</title>
</head>

<body>
    <div id="D" style="height:100px;width:100px;background-color:lightcoral;"></div>
    <div id="R"></div>
    <script>
        var D = document.getElementById('D'),
            R = document.getElementById('R');
        D.addEventListener("mouseover", function(e) {
            R.innerHTML += "Le curseur vient d'entrer dans " + e.currentTarget.id + "<br />";
        }, false);
        D.addEventListener("mouseout", function(e) {
            R.innerHTML += "Le curseur vient de sortir de " + e.currentTarget.id + "<br />";
        }, false);
    </script>
</body>

</html>
```

Gestionnaires d'évènements clavier

evenement-clavier.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Evènements clavier : keypress, keyup, keydown</title>
</head>

<body>
    <p><input id="field" type="text" /></p>
    <table style="margin-top: 20px;border: 1px solid black;">
        <tr>
            <td>keydown</td>
            <td></td>
        </tr>
        <tr>
            <td>keypress</td>
            <td></td>
        </tr>
        <tr>
            <td>keyup</td>
            <td></td>
        </tr>
    </table>
    <script>
        var cells = document.querySelectorAll("td:last-child");
        document.addEventListener('keydown', e => cells[0].innerHTML = e.key, false);
        document.addEventListener('keypress', e => cells[1].innerHTML = e.key, false);
        document.addEventListener('keyup', e => cells[2].innerHTML = e.key, false);
```

Désenregistrement d'écouteurs (DOM-2)

Avec `removeEventListener(type,listener[,opti])` en repassant exactement les paramètres passés à `addEventListener()`

Suppose de passer une fonction nommée ou variable fonctionnelle comme listener à `addEventListener()`

evenement-removeeventlistener.html

```
1  <!DOCTYPE html>
2  <html><head><meta charset="utf-8" />
3  <title>Désrengistrer un gestionnaire d'évènements</title>
4  </head>
5  <body>
6    <span id="clickme">Cliquez-moi !</span>
7    <script>
8    var element = document.querySelector('#clickme');
9    var fa1 = function() {alert("Vous m'avez cliqué !");}
10   var fa2 = function() {alert("Je confirme : Vous m'avez bien cliqué !");}
11   element.addEventListener('click', fa1, false);
12   element.addEventListener('click', fa2, false);
13   element.removeEventListener('click', fa1, false);
14   </script>
15 </body></html>
```

Déactivation du comportement par défaut

Avec `Event.preventDefault()` qui empêche l'action par défaut associée à l'objet `Event`

evenement-preventdefault.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Inhiber le comportement par défaut</title>
7  </head>
8
9  <body>
10     <p>Please click on the checkbox control.</p>
11     <form>Checkbox: <input id="c" type="checkbox" /></form>
12     <div id="d"></div>
13     <script>
14         document.querySelector("#c").addEventListener("click", function(event) {
15             document.querySelector("#d").innerHTML += "no visible check!<br>";
16             event.preventDefault();
17         }, false);
18     </script>
19 </body>
20
21 </html>
```

Déactivation du comportement par défaut

evenement-preventdefault-formulaire.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Inhiber le comportement par défaut</title>
7  </head>
8
9  <body>
10     <form method="get" action="script.php">
11         <input name="i1" type="text" /><input name="i2" type="text" /><button
type="submit">Go</button>
12     </form>
13     <script>
14         document.querySelector('form').onsubmit = function(e) {
15             if (Array.from(this.querySelectorAll('input')).some(i => i.value == '')) {
16                 e.preventDefault();
17                 alert('You need to fill in both names!');
18             }
19         }
20     </script>
21 </body>
22
23 </html>
```

Délégation d'évènements basée sur le bouillonnement

Créer un écouteur sur un parent plutôt que sur ses enfants

evenement-delegation.html

```
<!DOCTYPE html>
<html><head><meta charset="utf-8">
<title>Délégation d'évènements</title>
</head>
<body>
  <ul id="parent-list">
    <li id="post-1">Item 1</li>
    <li id="post-2">Item 2</li>
    <li id="post-3">Item 3</li>
  </ul>
  <script>
    document.querySelector("#parent-list").addEventListener(
        "click",
        function(e) {
          // !! balise en majuscules
          if (e.target && e.target.nodeName == "LI") {
            // e.target est l'item cliqué de la liste
            alert("Item " + e.target.id.replace("post-", "") +
                " a été cliqué!");
          }
        });
  </script>
</body></html>
```

CM JS : Formulaires

Gestion des formulaires

La propriété value des champs

formulaire-value.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Formulaires : value</title>
</head>

<body>
    <input class="fields" type="text" size="60" value="Vous n'avez pas le focus !" />
    <textarea class="fields" rows="2" cols="55">Vous n'avez pas le focus !</textarea>
    <script>
        document.querySelectorAll('.fields').forEach((f) => {
            f.addEventListener('focus', (e) => e.target.value = "Vous avez le focus !", false);
            f.addEventListener('blur', (e) => e.target.value = "Vous n'avez pas le focus !", false);
        });
    </script>
</body>

</html>
```

Les propriétés booléennes des champs

disabled et readOnly

formulaire-disabled-readonly.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>Formulaires : disabled, readOnly</title>
7  </head>
8
9  <body>
10     <input type="text" size="60" value="disabled => texte non sélectionnable" />
11     <br />
12     <input type="text" size="60" value="readOnly => texte sélectionnable" />
13     <script>
14         document.querySelectorAll("input[type='text']")[0].disabled = true;
15         document.querySelectorAll("input[type='text']")[1].readOnly = true;
16     </script>
17 </body>
18
19 </html>
```

La propriété checked des cases et boutons radio

formulaire-checked.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Formulaires : checked</title>
</head>

<body>
    <label><input type="checkbox" name="cc[]" value="A" /> Case A</label><br />
    <label><input type="checkbox" name="cc[]" value="B" /> Case B</label><br />
    <label><input type="checkbox" name="cc[]" value="C" /> Case C</label><br />
    <label><input type="radio" name="r" value="1" /> Bouton 1</label><br />
    <label><input type="radio" name="r" value="2" /> Bouton 2</label><br />
    <input type="button" value="Afficher les boutons cochés et les cases décochées" />
    <script>
        let C = Array.from(document.querySelectorAll("input[type='checkbox']"));
        let R = Array.from(document.querySelectorAll("input[type='radio']"));

        function check() {
            C.filter(c => !c.checked).forEach(c => alert("Case décochée : " + c.value));
            R.filter(r => r.checked).forEach(r => alert("Bouton coché : " + r.value));
        }
        document.querySelector("input[type='button']").addEventListener('click', check);
    </script>
</body>

</html>
```

Les propriétés selectedIndex et options des listes déroulantes

formulaire-selectedindex-options.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Formulaires : selectedIndex, options</title>
</head>

<body>
    <select id="list">
        <option value="aucune">Sélectionnez une couleur</option>
        <option value="bleu">Bleu</option>
        <option value="vert">Vert</option>
    </select>
    <script>
        document.querySelector('#list').addEventListener('change', () =>
            alert(list.options[list.selectedIndex].value), false);
    </script>
</body>

</html>
```

Les méthodes submit() et reset() de l'élément form

formulaire-form.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Formulaires : submit(), reset()</title>
</head>

<body>
    <form method="get" action="script.php">
        <input type="text" name="t" value="Tapez" /> <br />
        <button type="button">OK</button> <br />
        <input type="submit" value="OK !" />
        <input type="reset" value="RAZ !" />
    </form>
    <script>
        let form = document.querySelector("form");
        form.addEventListener('submit', e => {
            if (!confirm('Soumettre ?')) {
                e.preventDefault(); // empêche la soumission
            }
        }, false);
        form.addEventListener('reset', e =>
            alert('Vous avez réinitialisé le formulaire !'), false);
        document.querySelector("button").addEventListener('click', () => form.submit(), false);
    </script>
</body>

</html>
```

Les méthodes de gestion du focus et de la sélection

`focus()`, `blur()` et `select()`

formulaire-focus.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Formulaires : focus(), blur()</title>
</head>

<body>
    <form>
        <input id="text" type="text" value="Entrez un texte" /> <br />
        <button type="button">Donner le focus</button><br />
        <button type="button">Retirer le focus</button><br />
        <button type="button">Sélectionner le texte saisi</button>
    </form>
    <script>
        let text = document.querySelector('#text');
        document.querySelectorAll("button")[0].addEventListener('click', e =>
            text.focus(), false);
        document.querySelectorAll("button")[1].addEventListener('click', e =>
            text.blur(), false);
        document.querySelectorAll("button")[2].addEventListener('click', e =>
            text.select(), false);
    </script>
</body>

</html>
```

CM JS : Programmation asynchrone et promesses

Environnement d'exécution JS

Programmation asynchrone

Fil d'exécution (alias thread, tâche)

Exécution d'un ensemble d'instructions du langage machine d'un processeur au sein d'un processus.

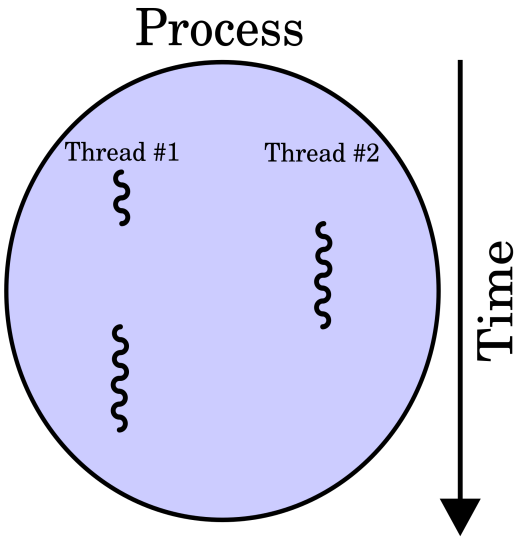
- Possède sa propre pile d'exécution.
- Partage la mémoire virtuelle du processus avec d'autres fils.

Exemples :

- Interaction avec l'utilisateur gérée par un fil.
- Calculs lourds gérés par plusieurs fils.

Avantages :

- Plus de blocage durant les phases de traitement intense, les requêtes sortantes, etc.
- Parallélisation possible sur les architectures multi-processeurs.



©Cburnett

Programmation asynchrone en JS

Quels besoins ?

De nombreuses API doivent exécuter du code de manière asynchrone :

- Récupération de données sur un serveur (images, polices, scripts, texte).
- Requêter une base de données.
- Accéder au flux vidéo d'une webcam.
- Relayer l'affichage à un casque de réalité virtuelle (VR), etc.

JS est mono-tâche

Un seul fil (main thread) et de possibles situations de blocage :

- Téléchargements.
- Invocations de services distants.
- Longues boucles de calcul, etc.

Blocage : exemple

async-simple-sync.js

```
1  document.querySelector('button').addEventListener('click', () => {
2      let myDate;
3      for (const k of Array(10000000).keys()) {
4          myDate = new Date();
5      }
6      console.log(myDate);
7      let pElem = document.createElement('p');
8      pElem.textContent = 'This is a newly-added paragraph.';
9      document.body.appendChild(pElem);
10 });
```

Environnement d'exécution JS

Environnement d'exécution

La pile d'appels (*stack*)

- On y empile les cadres (*frames*).
- Chaque cadre stocke les arguments et variables locales d'un appel de fonction.
- On dépile le cadre du dessus lorsque la fonction associée a terminé.

Le tas (*heap*)

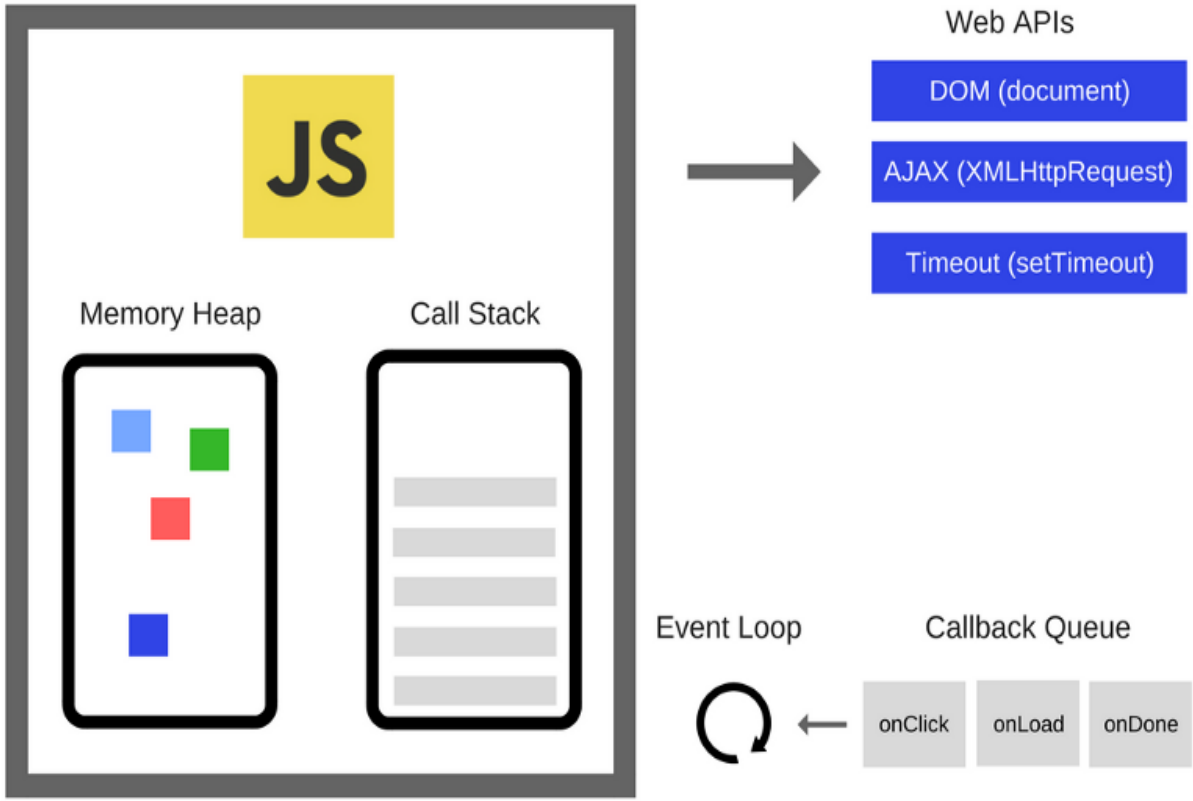
- On y alloue/désalloue de la mémoire pour les objets.

La file de messages (*event queue*)

- Les nouveaux messages y sont ajoutés à la fin dans l'ordre d'arrivée.
- Le premier message sera traité intégralement puis supprimé avant de boucler.

La boucle d'évènements JS (*event queue*)

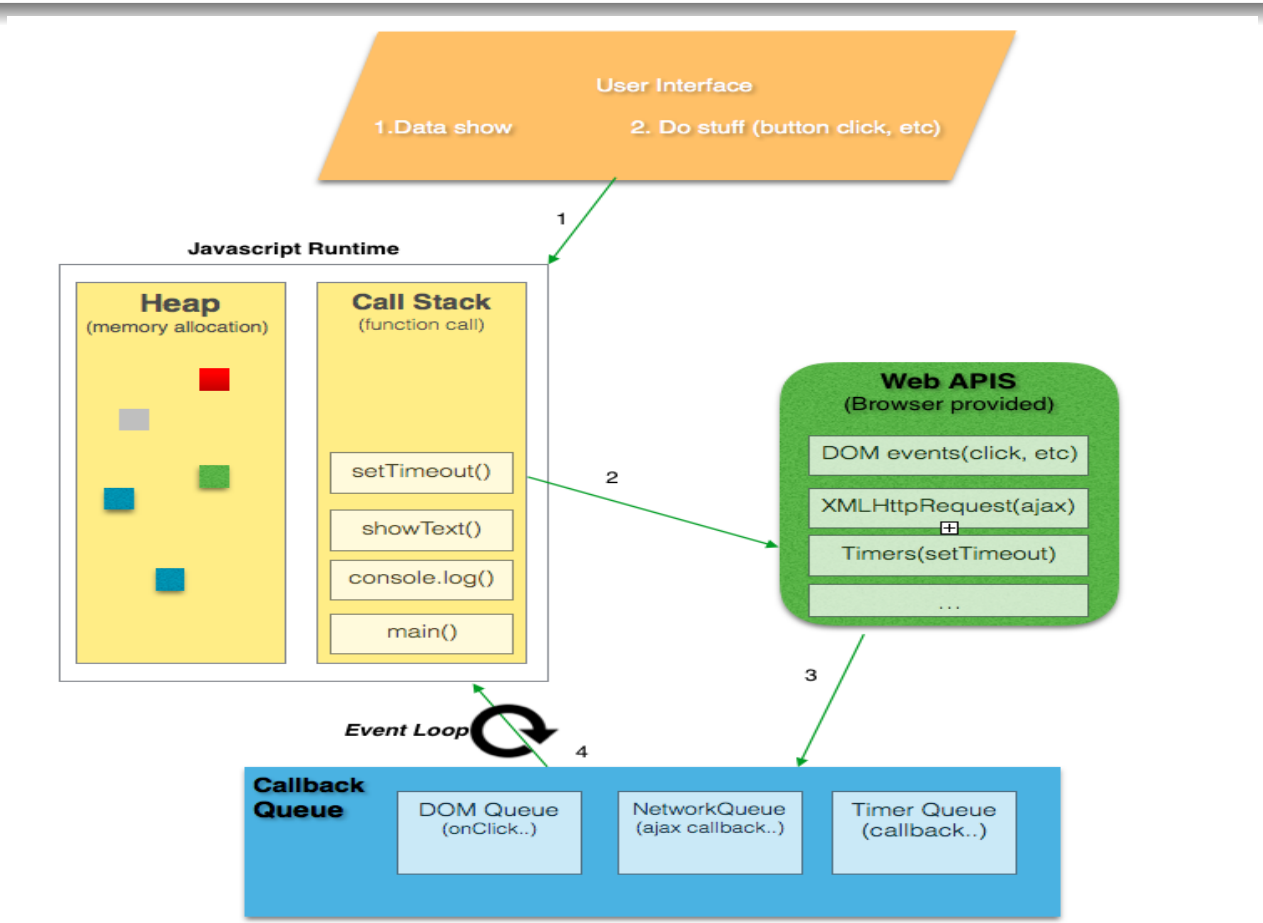
Pile, tas, file et Web APIs



©MDN

La boucle d'évènements JS

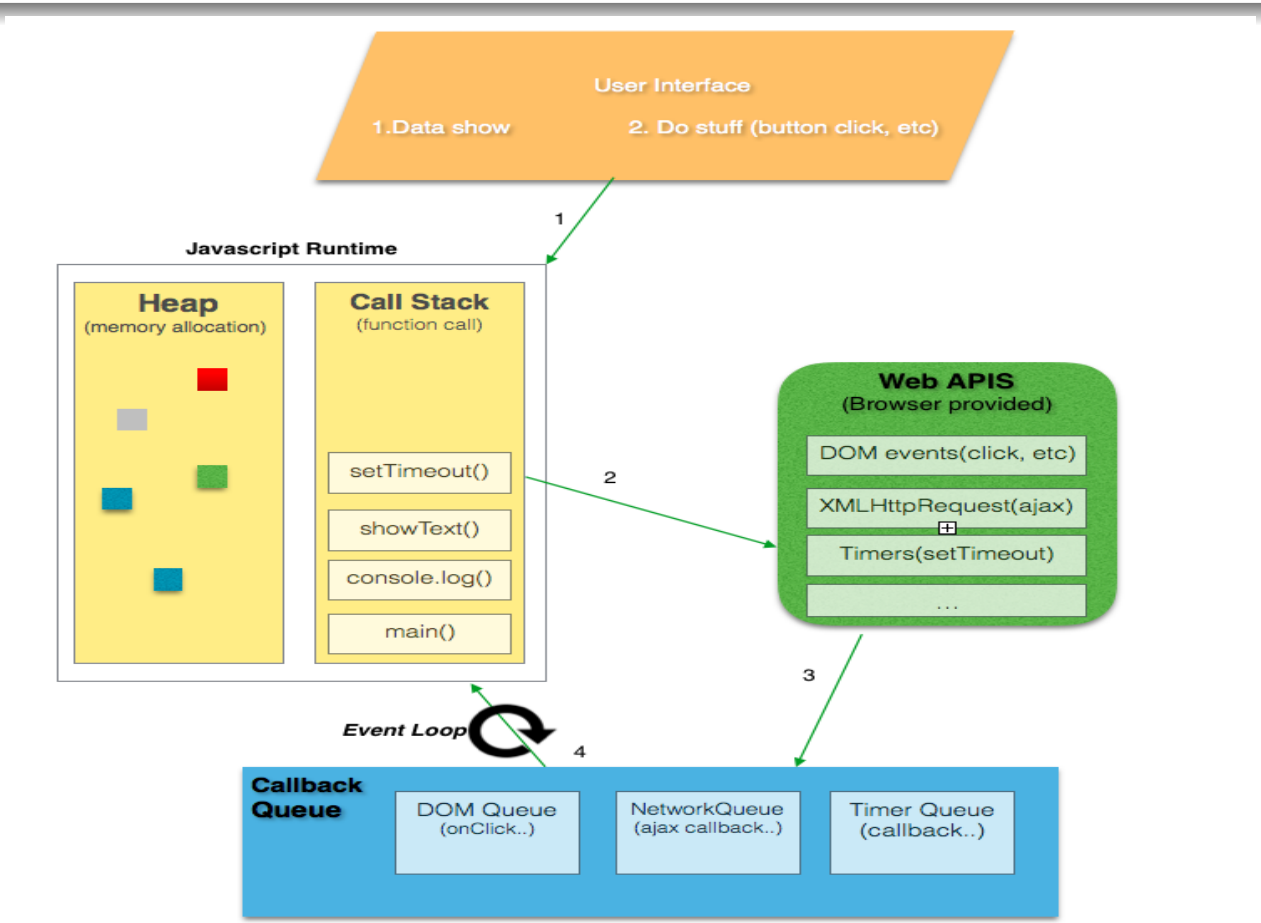
Un message est ajouté à la file lorsqu'un évènement survient et qu'un gestionnaire y a été associé.



©G. Pandvia

La boucle d'évènements JS

La premier message n'est traité et l'est alors intégralement que lorsque la pile est vide.



©G. Pandvia

Exemple avec le minuteur `setTimeout()`

`WindowOrWorkerGlobalScope.setTimeout()` appelée avec 2 arguments

1. Le message = la callback à exécuter quand il sera traité.
2. Le délai minimum à attendre avant que le message ne soit placé dans la file.

async-settimeout.js

```
1  const s = new Date().getSeconds();
2  setTimeout(function() {
3      console.log("Exécuté après " + (new Date().getSeconds() - s) + " secondes.");
4  }, 5000);
5  while (true) {
6      if (new Date().getSeconds() - s >= 2) {
7          console.log("Ouf, on a bouclé pendant 2 secondes");
8          break;
9      }
10 }
11 // Console avec horodatage :
12 // 10:27:20,999 Ouf, on a bouclé pendant 2 secondes
13 // 10:27:24,423 Exécuté après 5 secondes.
```

Les promesses

Fonctions de rappels et promesses

Le support de l'asynchronisme en JS

- Historique : les fonctions de rappels (*callbacks*) asynchrones
- Depuis ES6 (ECMAScript 2015) : les promesses.

De nombreuses API utilisent les promesses

Par exemple, `fetch` (vs. `XMLHttpRequest`).

Les callbacks ne s'exécutent pas toutes de manière asynchrone

Dépend du contexte d'exécution : par exemple, synchrone dans `forEach`.

Fonction de rappel asynchrone

Une fonction `f` passée en paramètre à une fonction `g` laquelle

- S'exécutera en tâche de fond sans bloquer le fil principal.
- Rappellera `f` une fois terminée (la placera dans la boucle).

Exemples : gestionnaires d'évènements DOM avec callback.

async-callback.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>MDN : simple callback</title>
6  </head>
7  <body>
8  <button id="btn">Click!</button>
9  <script>
10 document.querySelector("#btn").addEventListener('click', () => {
11     alert('You clicked me!');
12     let pElem = document.createElement('p');
13     pElem.textContent = 'This is a newly-added paragraph.';
14     document.body.appendChild(pElem);
15   })
16 </script>
17 </body>
18 </html>
```

Fonctions de rappel asynchrones

La fonction asynchrone doit rappeler la fonction de rappel

- Peut lui passer différentes informations (réponse, état, erreur . . .).

async-callback-xhr.js

```
1  function loadAsset(url, type, callback) {
2      let xhr = new XMLHttpRequest();
3      xhr.open('GET', url);
4      xhr.responseType = type;
5      xhr.onload = function() {
6          if (xhr.readyState === xhr.DONE && xhr.status === 200) {
7              callback(xhr.responseText);
8          }
9      };
10     xhr.send();
11 }
12
13 function displayText(responseText) {
14     document.querySelector("#gettext").textContent = responseText;
15 }
16
17 loadAsset('description.txt', 'text', displayText);
```

Les promesses

Nouveau style de programmation asynchrone en JS

Une promesse est un objet JS qui représente le succès ou l'échec d'une opération asynchrone.

Une promesse

- Est un objet `Promise` renvoyé immédiatement mais qui se résoud plus tard de manière asynchrone.
- Sa résolution est soit positive, soit négative.

Une promesse est

- Initialement en attente d'être résolue (pending).
- Puis résolue : soit tenue (fulfilled), soit rompue (rejected).

Les promesses

Initialisation avec `new Promise((resolve,reject)=>{...})`

- Exécute la fonction anonyme en asynchrone.
- Crée 2 propriétés internes désignant l'état et le résultat de la promesse.

La fonction asynchrone résolvera la promesse

- Positivement en appelant `resolve`.
- Négativement en appelant `reject`.

La valeur communiquée à `resolve`/`reject` devient le résultat de la promesse

Les promesses

Le résultat `r` d'une promesse résolue peut être communiqué à des fonctions de rappel anonymes (executor)

En invoquant la méthode `then(succes => {...},échec => {...})` qui appellera la première callback si la promesse est tenue (avec `succes == r`), la seconde sinon (avec `échec == r`).

`then(s=>{...})` avec une seule callback ne gérera que le succès

La méthode `catch(r => {...})` peut alors être chainée pour gérer le cas où la promesse est rompue.

Chaque bloc `then(...)` ou `catch(...)` peut renvoyer une nouvelle promesse

On peut donc enchaîner les promesses avec une suite mêlant blocs `then(...)` et `catch(...)`

Exemple de promesse avec then

async-promise.js

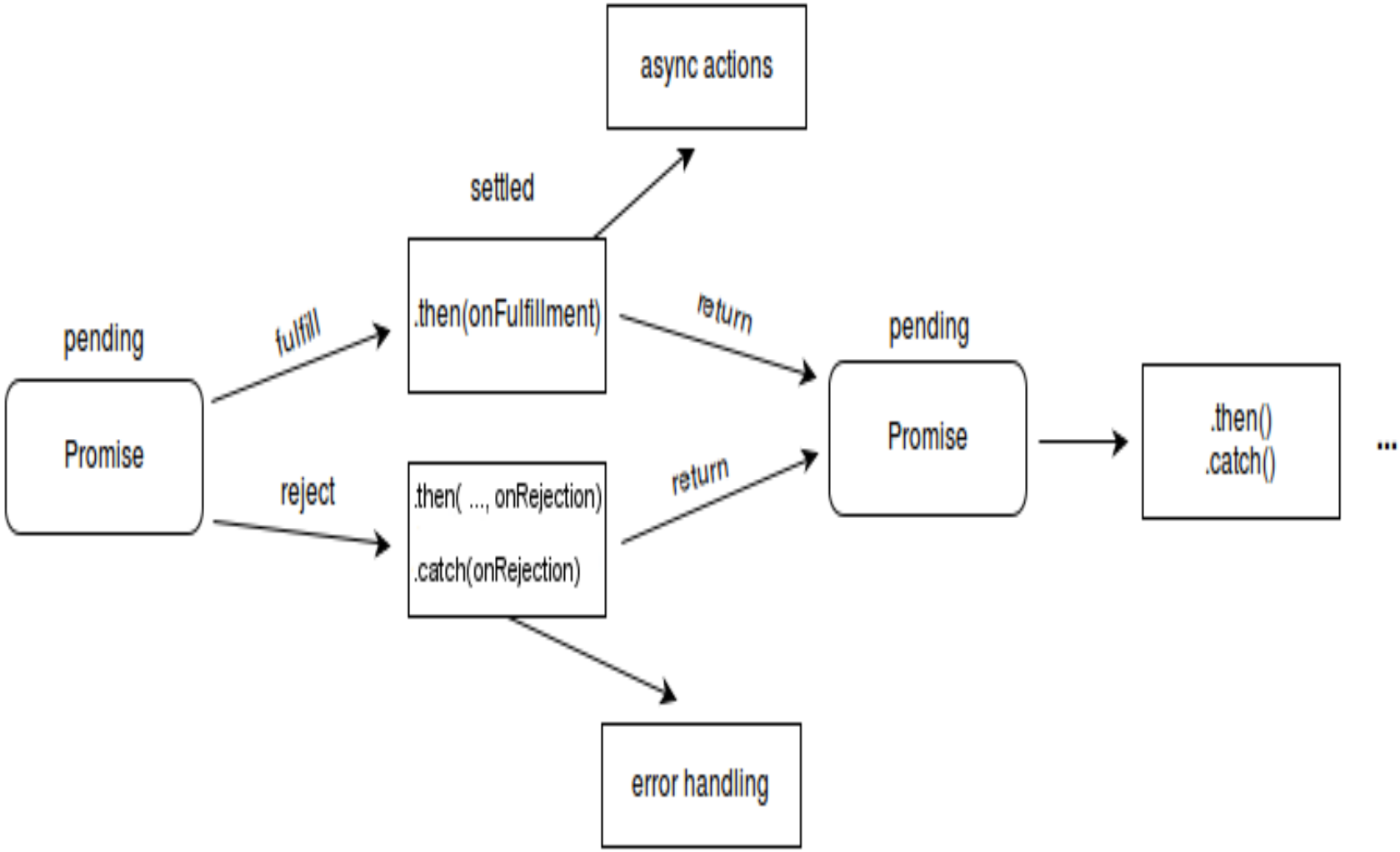
```
let myFirstPromise = new Promise((resolve, reject) => {
    // We call resolve(...) when what we were doing asynchronously was
    // successful, and reject(...) when it failed.
    // In this example, we use setTimeout(...) to simulate async code.
    // In reality, you will probably be using something like XHR or an HTML5 API.
    setTimeout(function() {
        resolve("Success!") // Yay! Everything went well!
    }, 5000)
})

myFirstPromise.then((successMessage) => {
    // successMessage is whatever we passed in the resolve(...) function above.
    // It doesn't have to be a string, but if it is only a succeed message, it probably will be.
    console.log("Yay! " + successMessage)
});
```

Exemple de promesse avec then/catch

async-promise-then-catch.js

```
1  const oddTime = (date) => {
2      return new Promise((resolve, reject) => {
3          parseInt(date.getTime() / 1000) % 2 // (1)
4              ?
5              resolve('le nombre de secondes est impair :-)') :
6              reject('le nombre de secondes n\'est pas impair :-(');
7      });
8  }
9  const now = new Date();
10 oddTime(now) // (2)
11     .then(msg => console.log(msg), msg => console.error(msg));
12
13 oddTime(new Date(now.getTime() + 1000)) // (3)
14     .then(msg => console.log(msg)) // (4)
15     .catch(msg => console.error(msg)) // (5)
```

Enchaînement de promesses



©MDN

Exemple de chaînage de promesses (1)

async-fetch-basic.js

```
1  // Call the fetch() method to fetch the image, and store it in a variable
2  let promise = fetch('coffee.jpg');
3  // Use a then() block to respond to the promise's successful completion
4  // by taking the returned response and running blob() on it to transform it into a blob
5  let promise2 = promise.then(response => response.blob());
6
7  // blob() also returns a promise; when it successfully completes it returns
8  // the blob object in the callback
9  let promise3 = promise2.then(myBlob => {
10   // Create an object URL that points to the blob object
11   let objectURL = URL.createObjectURL(myBlob);
12   // Create an <img> element to display the blob (it's an image)
13   let image = document.createElement('img');
14   // Set the src of the <img> to the object URL so the image displays it
15   image.src = objectURL;
16   // Append the <img> element to the DOM
17   document.body.appendChild(image);
18 })
19 // If there is a problem, log a useful error message to the console
20 let errorCase = promise3.catch(e => {
21   console.log('Problem with your fetch operation: ' + e.message);
22 });
```

Exemple de chaînage de promesses (2)

Ecriture directe

La valeur renvoyée par une promesse tenue est le paramètre passé à la fonction (executor) du bloc `then()` suivant.

async-fetch-1.js

```
1  fetch('coffee.jpg')
2  .then(response => response.blob())
3  .then(myBlob => {
4    let objectURL = URL.createObjectURL(myBlob);
5    let image = document.createElement('img');
6    image.src = objectURL;
7    document.body.appendChild(image);
8  })
9  .catch(e => {
10   console.log('Problem with your fetch operation: ' + e.message);
11 });
```

Promesses vs. callbacks traditionnelles

Différences

- Une promesse ne peut réussir ou échouer qu'une fois.
- On peut ajouter une callback à une promesse même après qu'elle ait réussi ou échoué.
- On peut exécuter plusieurs opérations asynchrones en séquence et dans un ordre précis grâce au chaînage des promesses.

async-promise-1.js

```
1  let myFirstPromise = new Promise((resolve, reject) => {
2     resolve("Success!");
3  })
4  for(let i = 0; i < 10000000; i++) {
5      let date = new Date();
6      myDate = date
7    }
8  myFirstPromise.then((successMessage) => {
9    console.log("Yay! " + successMessage)
10 });
```

Tenir plusieurs promesses avec `Promise.all` (1)

`Promise.all()` prend un tableau de promesses et renvoie une promesse

- Qui sera tenue si toutes les promesses le seront.
- Qui sera rompue dès que l'une d'elles le sera.

async-promise-all-part1.js

```
1  // Define function to fetch a file and return it in a usable form
2  function fetchAndDecode(url, type) {
3    // Returning the top level promise, so the result of the entire chain is returned out of the function
4    return fetch(url).then(response => {
5      // Depending on what type of file is being fetched, use the relevant function to decode its contents
6      if(type === 'blob') {
7          return response.blob();
8      } else if(type === 'text') {
9          return response.text();
10     }
11   })
12   .catch(e => {
13     console.log('Problem fetching resource "${url}": ' + e.message);
14   });
15 }
16 // Call the fetchAndDecode() method to fetch the images and the text, and store their promises in variables
17 let coffee = fetchAndDecode('coffee.jpg', 'blob');
18 let tea = fetchAndDecode('tea.jpg', 'blob');
19 let description = fetchAndDecode('description.txt', 'text');
```

Tenir plusieurs promesses avec `Promise.all` (2)

async-promise-all-part2.js

```
1  // Use Promise.all() to run code only when all three function calls have resolved
2  Promise.all([coffee, tea, description]).then(values => {
3    console.log(values);
4    // Store each value returned from the promises in separate variables; create object URLs from the blobs
5    let objectURL1 = URL.createObjectURL(values[0]);
6    let objectURL2 = URL.createObjectURL(values[1]);
7    let descText = values[2];
8
9    // Display the images in <img> elements
10   let image1 = document.createElement('img');
11   let image2 = document.createElement('img');
12   image1.src = objectURL1;
13   image2.src = objectURL2;
14   document.body.appendChild(image1);
15   document.body.appendChild(image2);
16
17   // Display the text in a paragraph
18   let para = document.createElement('p');
19   para.textContent = descText;
20   document.body.appendChild(para);
21 });
```

async et await (ES7)

async et await

Syntaxe légère pour les promesses ajoutée dans ES7 (ECMAScript 2017)

async

- Se place avant une déclaration de fonction.
- Rend la fonction asynchrone.
- La fonction renvoie alors une promesse à consommer.
- On peut lui ajouter un bloc `.then(){...}` pour la résoudre.

async-async.js

```
1 async function hello1() { return "Hello" };
2 hello1();
3 let hello2 = async function() { return "Hello" };
4 hello2();
5 let hello3 = async () => { return "Hello" }; hello3();
6 hello1().then((value) => console.log(value));
7 hello2().then(console.log);
```

async et await

await

- S'utilise au sein d'une promesse ou fonction asynchrone (précédée de async).
- Se place avant un appel à une fonction asynchrone avec promesse.
- Bloque l'exécution du code qui suit la promesse jusqu'à sa résolution.

async-await.js

```
1 async function hello() {
2   return greeting = await Promise.resolve("Hello");
3 };
4
5 hello().then(alert);
```

Promesses vs. async/await

async-await-async-promise.js

```
1 fetch('coffee.jpg')
2 .then(response => response.blob())
3 .then(myBlob => {
4   let objectURL = URL.createObjectURL(myBlob);
5   let image = document.createElement('img');
6   image.src = objectURL;
7   document.body.appendChild(image);
8 })
9 .catch(e => {
10   console.log('Problem with your fetch operation: ' + e.message);
11 });
```

async-await-async.js

```
1 async function myFetch() {
2   let response = await fetch('coffee.jpg');
3   let myBlob = await response.blob();
4   let objectURL = URL.createObjectURL(myBlob);
5   let image = document.createElement('img');
6   image.src = objectURL;
7   document.body.appendChild(image);
8 }
9 myFetch().catch(e => {
10   console.log('Problem with your fetch operation: ' + e.message);
11 });
```

CM JS : APIs Fetch et XHR

L'API Fetch

L'API `Fetch`

Interface d'accès en JS au protocole HTTP

- Permet de récupérer des données par requêtes asynchrones.
- Utilisable depuis une page web ou dans un service worker.
- Alternative plus puissante que que l'API `XMLHttpRequest`
  - Basée sur les promesses.
  - Fournit une interface aux requêtes, réponses, en-tête et corps des messages HTTP.
- Méthode globale `fetch` à disposition.

fetch()

- Tente de récupérer une ressource web (p. ex. par son URL).
- Permet de paramétrer la requête (en-têtes, corps, etc.).
- Renvoie une promesse qui n'est rompue qu'en cas d'erreur réseau ou absence de réponse, pas en cas d'erreur HTTP (p. ex. 404).
- Et qui se résoud par un objet `Response` modélisant la réponse HTTP
  - `Response.status` fournit le code de la réponse HTTP : Informatif [100–199], Succès [200–299], Redirection [300–399], Erreur client [400–499], Erreur serveur [500–599].
  - `Response.ok` indique si la réponse est fructueuse `[200-299]`.
  - Fournit différentes méthodes renvoyant chacune une promesse permettant d'extraire le corps de la réponse selon son format :
    - `arrayBuffer()` : tableau d'octets.
    - `blob()` : blob (p. ex. images).
    - `formData()` : encodage clé-valeur au format `multipart/form-data`.
    - `json()` : un objet JS résultant du parsing de données JSON.
    - `text()` : une chaîne de caractères.

Arguments de `fetch`

async-fetch-api.js

```
let promise = fetch(url, {
    // * indique l'option par défaut
    method: "GET", // *GET, POST, PUT, DELETE, etc.
    headers: {
        "Content-Type": "text/plain;charset=UTF-8" //pour un corps de type chaine
    },
    body: undefined, // string, FormData, Blob, BufferSource, ou URLSearchParams
    // doit correspondre à Content-Type pour requête POST
    referrer: "about:client",
    // "" (pas de réferent) ou une url de l'origine
    referrerPolicy: "no-referrer-when-downgrade",
    // no-referrer, *no-referrer-when-downgrade, origin,
    // origin-when-cross-origin, same-origin, strict-origin,
    // strict-origin-when-cross-origin, unsafe-url
    mode: "cors", // no-cors, *cors, same-origin
    credentials: "same-origin", // include, *same-origin, omit
    cache: "default", // *default, no-cache, reload, force-cache, only-if-cached
    redirect: "follow", // manual, *follow, error
    integrity: "", // ou hash tel "sha256-abcdef1234567890"
    keepalive: false, // ou true pour que la requête survive à la page
    signal: undefined // ou AbortController pour annuler la requête
});
```

fetch : Récupérer du texte brut par méthode GET

async-xhr-fetch-get-text.php

```php
header("Content-Type: text/plain; charset=UTF-8");
$txt = "Le Lorem Ipsum est simplement du faux texte employé ...";
echo $txt;
```

async-fetch-get-text.js

```javascript
let ft = function() {
    let elt = document.querySelector('#gettext');
    let myRequest = new Request('async-xhr-fetch-get-text.php');
    fetch(myRequest)
    .then(function(response) {
        if (!response.ok) {
            throw new Error("HTTP error, status = " + response.status);
        }
        return response.text();
    })
    .then(function(myText) {
        elt.textContent = myText;
    })
    .catch(function(error) {
        let p = document.createElement('p');
        p.appendChild(document.createTextNode('Error: ' + error.message));
        document.body.insertBefore(p, elt);
    });
}();
```

fetch : Récupérer du JSON par méthode GET (1)

async-xhr-fetch-get-json.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'nom' => 'Clémenceau',
    'surnoms' => [
        'Le tigre',
        'Le père La Victoire',
        'Le tombeur de gouvernements',
        'Le premier flic de France'
    ]
];
echo json_encode($tab);
```

fetch : Récupérer du JSON par méthode GET (2)

async-fetch-get-json.js

```
let fj = function() {
    let elt = document.querySelector('#getjson');
    let myRequest = new Request('async-xhr-fetch-get-json.php');
    fetch(myRequest)
    .then(function(response) {
        if (!response.ok) {
            throw new Error("HTTP error, status = " + response.status);
        }
        return response.json();
    })
    .then(function(myJson) {
        elt.textContent = myJson.nom + " - " + myJson.surnoms[1];
    })
    .catch(function(error) {
        let p = document.createElement('p');
        p.appendChild(document.createTextNode('Error: ' + error.message));
        document.body.insertBefore(p, elt);
    });
}();
```

fetch : Récupérer du XML par méthode GET

async-xhr-fetch-get-xml.php

```php
header("Content-Type: application/xml; charset=UTF-8");
echo file_get_contents("bibliotheque.xml");
```

async-fetch-get-xml.js

```javascript
let fx = function() {
    let elt = document.querySelector('#getxml');
    let myRequest = new Request('async-xhr-fetch-get-xml.php');
    fetch(myRequest)
    .then(function(response) {
        if (!response.ok) {
            throw new Error("HTTP error, status = " + response.status);
        }
        return response.text();
    })
    .then(function(myXml) {
        xml = (new window.DOMParser()).parseFromString(myXml, "text/xml");
        let eltxml = xml.querySelector("titre");
        elt.textContent = eltxml.textContent;
    })
    .catch(function(error) {
        let p = document.createElement('p');
        p.appendChild(document.createTextNode('Error: ' + error.message));
        document.body.insertBefore(p, elt);
    });
}();
```

fetch : Récupérer des données binaires par méthode GET

async-xhr-fetch-get-blob-1.php

```php
// ouvre un fichier en mode binaire
$name = 'tea.jpg';
$fp = fopen($name, 'rb');
// envoie les en-têtes
header("Content-Type: image/jpg");
header("Content-Length: " . filesize($name));
// envoie le contenu du fichier, puis stoppe le script
fpassthru($fp);
exit;
```

async-fetch-get-blob.js

```javascript
let fb = function() {
    let elt = document.querySelector('#getimg > img');
    let myRequest = new Request('flowers.jpg');
    fetch(myRequest)
    .then(function(response) {
        if (!response.ok) {
            throw new Error("HTTP error, status = " + response.status);
        }
        return response.blob();
    })
    .then(function(myBlob) {
        let objectURL = URL.createObjectURL(myBlob);
        elt.src = objectURL;
    })
    .catch(function(error) {
        let p = document.createElement('p');
        p.appendChild(document.createTextNode('Error: ' + error.message));
        document.body.insertBefore(p, elt);
    });
```

fetch : Envoyer des données par méthode GET

async-xhr-fetch-get.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'surname' => $_GET["surname"],
    'name' => $_GET["name"],
    'age' => 10
];
echo json_encode($tab);
```

async-fetch-get.js

```javascript
let fg = function() {
    let elt = document.querySelector("#xget");
    let url = new URL(window.location.href.replace(/[^\/]*$/,
    "async-xhr-fetch-get.php"));
    const params = new URLSearchParams();
    params.set('surname', 'foo');
    params.set('name', 'bar');
    url.search = params.toString();
    fetch(url, {
        method: 'GET'
    })
    .then((response) => response.json())
    .then((myJson) => {
        console.log('Success:', myJson);
        elt.textContent = myJson.surname + " - " + myJson.name + " - " + myJson.age;
    })
    .catch((error) => {
        console.error('Error:', error);
    });
}();
```

fetch : Envoyer des données par méthode POST

async-xhr-fetch-post.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'surname' => $_POST["surname"],
    'name' => $_POST["name"],
    'age' => 10
];
echo json_encode($tab);
```

async-fetch-post.js

```javascript
let fp = function() {
    let elt = document.querySelector("#xpost");
    fetch('async-xhr-fetch-post.php', {
        method: 'POST', // or 'PUT'
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body: new URLSearchParams("surname=foo&name=bar"),
    })
    .then((response) => response.json())
    .then((myJson) => {
        console.log('Success:', myJson);
        elt.textContent = myJson.surname + " - " + myJson.name + " - " + myJson.age;
    })
    .catch((error) => {
        console.error('Error:', error);
    });
}();
```

L'API XMLHttpRequest

Ajax et l'objet XMLHttpRequest

Ajax (Asynchronous JavaScript and XML)

Introduit par Jesse James Garrett en 2005
Réunion de trois technologies afin de créer des interfaces utilisateur riches (RUI)

- Javascript pour l'interaction côté client.
- `XMLHttpRequest` pour les requêtes asynchrones.
- XML, JSON ou texte libre pour l'échange de données client-serveur.

Repose également sur DOM et CSS.

L'objet `XMLHttpRequest` (XHR)

- Recommandation du W3C en 2006.
- Permet d'envoyer une requête asynchrone au serveur.
- Une fois les données reçues, on met à jour le DOM.

Propriétés de XHR

Propriétés

`status` : statut de la réponse HTTP au format numérique

- 200 (OK), 404 (Not Found) ...

`statusText` : statut de la réponse HTTP au format texte. `readyState` : état de l'objet

- 0=*uninitialized*, 1=*open*, 2=*sent*, 3=*receiving*, 4=*loaded*

`responseText` : réponse du serveur au format HTML/JSON.
`responseXML` : réponse du serveur au format XML.

Evènements observables

`onreadystatechange`, `abort`, `error`, `load`, `loadend`, `loadstart`, `progress`, `timeout`

Méthodes de XHR

Méthodes

`open(method,url[,async[,user[,password]]])`

- Ouverture d'une connexion pour envoi ou réception de données (GET, POST) de manière asynchrone par défaut.

`send([content])`

- Envoi de la requête avec ou sans données.

`abort()`

- Mettre fin à la requête.

`setRequestHeader()` / `getReponseHeader()`

- Crée / récupère une en-tête pour la requête HTTP / de la réponse HTTP.

Lorsque l'on a reçu les données du serveur et qu'elles sont prêtes à être traitées : `readyState==4` et `status ==200`

Ajax : bonnes pratiques côté client et serveur

Côté client

Placer le code Ajax dans des IIFE.

Côté serveur (avec PHP)

Spécifier le type MIME dans l'en-tête de la réponse HTTP :

- Appeler `header()` avant tout autre affichage.

Respecter la syntaxe JSON en cas de réponse JSON :

- Appeler `json_encode()` sur "l'objet" à encoder.
- Sinon, veiller aux virgules traînantes, apostrophes, ...

Pour le téléchargement d'un fichier :

- Appeler `filesize()` pour indiquer sa taille dans l'en-tête de la réponse HTTP.
- Appeller `fpassthru()` pour le transmettre s'il est binaire.

Récupérer du texte brut par méthode GET

async-xhr-fetch-get-text.php

```php
header("Content-Type: text/plain; charset=UTF-8");
$txt = "Le Lorem Ipsum est simplement du faux texte employé ...";
echo $txt;
```

async-xhr-get-text.js

```javascript
let xhrt = function () {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', 'async-xhr-fetch-get-text.php', true);
    xhr.responseType = 'text';
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            let elt = document.querySelector("#gettext");
            elt.textContent = xhr.responseText;
        }
    };
    xhr.send();
}();
```

Récupérer du JSON par méthode GET

async-xhr-fetch-get-json.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'nom' => 'Clémenceau',
    'surnoms' => [
        'Le tigre',
        'Le père La Victoire',
        'Le tombeur de gouvernements',
        'Le premier flic de France'
    ]
];
echo json_encode($tab);
```

async-xhr-get-json.js

```javascript
let xhrj = function () {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', 'async-xhr-fetch-get-json.php', true);
    xhr.responseType = 'json'; // JSON parsed into JS object and stored in 'response'
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            let elt = document.querySelector("#getjson");
            elt.textContent = xhr.response.surnoms[0];
        }
    };
    xhr.send();
}();
```

Récupérer du XML par méthode GET

async-xhr-fetch-get-xml.php

```php
header("Content-Type: application/xml; charset=UTF-8");
echo file_get_contents("bibliotheque.xml");
```

async-xhr-get-xml.js

```javascript
let xhrx = function () {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', 'async-xhr-fetch-get-xml.php', true);
    xhr.responseType = 'document'; // type for HTML/XML docs
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            let elthtml = document.querySelector("#getxml");
            let eltxml = xhr.responseXML.querySelector("titre");
            elthtml.textContent = eltxml.textContent;
        }
    };
    xhr.send();
}();
```

Récupérer des données binaires par méthode GET (1)

async-xhr-get-blob.js

```
1  let xhri = function () {
2      let xhr = new XMLHttpRequest();
3      xhr.open('GET', 'coffee.jpg', true);
4      xhr.responseType = 'blob';
5      xhr.onload = function () {
6          if (xhr.readyState === xhr.DONE && xhr.status === 200) {
7              let objectURL = URL.createObjectURL(xhr.response);
8              document.querySelector("#getimg > img").src = objectURL;
9          }
10     };
11     xhr.send();
12 }();
```

Récupérer des données binaires par méthode GET (2)

async-xhr-fetch-get-blob-1.php

```php
// ouvre un fichier en mode binaire
$name = 'tea.jpg';
$fp = fopen($name, 'rb');
// envoie les en-têtes
header("Content-Type: image/jpg");
header("Content-Length: " . filesize($name));
// envoie le contenu du fichier, puis stoppe le script
fpassthru($fp);
exit;
```

async-xhr-get-blob-1.js

```javascript
let xhrb = function () {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', 'async-xhr-fetch-get-blob-1.php', true);
    xhr.responseType = 'blob';
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            let objectURL = URL.createObjectURL(xhr.response);
            document.querySelector("#getblob > img").src = objectURL;
        }
    };
    xhr.send();
}();
```

Envoyer des données par méthode GET

async-xhr-fetch-get.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'surname' => $_GET["surname"],
    'name' => $_GET["name"],
    'age' => 10
];
echo json_encode($tab);
```

async-xhr-get.js

```javascript
let xhrgj = function () {
    let elt = document.querySelector("#xget");
    let xhr = new XMLHttpRequest();
    let url = new URL(window.location.href.replace(/[^\/]*$/,
            "async-xhr-fetch-get.php"));
    const params = new URLSearchParams();
    params.set('surname', 'foo');
    params.set('name', 'bar');
    url.search = params.toString();
    xhr.open('GET', url, true);
    xhr.responseType = 'json'; // JSON parsed into JS object and stored in 'response'
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            elt.textContent = xhr.response["surname"] + " " + xhr.response["age"];
        }
    };
    xhr.send();
}();
```

Envoyer des données par méthode POST

async-xhr-fetch-post.php

```php
header("Content-Type: application/json; charset=UTF-8");
$tab = [
    'surname' => $_POST["surname"],
    'name' => $_POST["name"],
    'age' => 10
];
echo json_encode($tab);
```

async-xhr-post.js

```javascript
let xhrpj = function () {
    let xhr = new XMLHttpRequest();
    xhr.open('POST', 'async-xhr-fetch-post.php', true);
    xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    xhr.responseType = 'json'; // JSON parsed into JS object and stored in 'response'
    xhr.onload = function () {
        if (xhr.readyState === xhr.DONE && xhr.status === 200) {
            let elt = document.querySelector("#xpost");
            elt.textContent = xhr.response["surname"] + " " + xhr.response["age"];
        }
    };
    xhr.send("surname=foo&name=baz");
}();
```

CM JS : Web workers

Web Workers

Web Workers et iframes d'origine multiple

Un Web Worker se construit avec `Worker(f)`

- Exécute le fichier JS `f` passé en argument.
- S'exécute sur un fil différent du fil principal.
- A sa propre pile, tas, et file de messages.
- Ne peut pas accéder au DOM et à l'objet `window`.
- Communique par messages avec le fil principal ou autres workers avec `postMessage()`.
- Peut utiliser XHR et websockets (canal bidirectionnel avec un serveur).

Autres types de workers

- SharedWorker : worker partagés entre scripts s'exécutant dans différentes fenêtres ou iframes.
- ServiceWorker : "proxy" entre navigateur, applis web et réseau (cache, notifications push ...).

Exemple de Web Worker

Récupération des données d'un message via gestionnaire onmessage (propriété de worker) et event.data

worker-simple-sync.js

```
1  const btn = document.querySelector('button');
2  const worker = new Worker('worker-date.js');
3  btn.addEventListener('click', () => {
4      worker.postMessage('Go!');
5      let pElem = document.createElement('p');
6      pElem.textContent = 'This is a newly-added paragraph.';
7      document.body.appendChild(pElem);
8  });
9  worker.onmessage = function(e) {
10     console.log(e.data);
11 }
```

worker-date.js

```
1 onmessage = function(e) {
2     console.log(e.data); // Go !
3     let myDate;
4     Array.from(Array(10000000).keys()).forEach(() => myDate = new Date());
5     postMessage(myDate);
6 }
```

Exemple de Web Worker : suite de Fibonacci (1)

Interception d'erreur via gestionnaire onerror (propriété de worker)

worker-fibonacci.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="UTF-8" />
6      <title>MDN : Test threads fibonacci</title>
7  </head>
8
9  <body>
10     <div id="result"></div>
11     <script>
12         var worker = new Worker('worker-fibonacci.js');
13         worker.onmessage = function(event) {
14             document.getElementById('result').textContent = event.data;
15             console.log('Got: ' + event.data + '\n');
16         };
17         worker.onerror = function(error) {
18             console.error('Worker error: ' + error.message + '\n');
19             throw error;
20         };
21         worker.postMessage('5');
22     </script>
23 </body>
24
```

Exemple de Web Worker : suite de Fibonacci (2)

worker-fibonacci.js

```
1  let results = [];
2
3  function resultReceiver(event) {
4      results.push(parseInt(event.data));
5      if (results.length == 2) {
6          postMessage(results[0] + results[1]);
7      }
8  }
9
10 function errorReceiver(event) {
11     throw event.data;
12 }
13
14 onmessage = function(event) {
15     let n = parseInt(event.data);
16     if (n == 0 || n == 1) {
17         postMessage(n);
18         return;
19     }
20     for (let i = 1; i <= 2; i++) {
21         let worker = new Worker('worker-fibonacci.js');
22         worker.onmessage = resultReceiver;
23         worker.onerror = errorReceiver;
24         worker.postMessage(n - i);
25     }
26 };
```

CM JS : Les API

Les APIs en JS

Les APIs du Web

API = Interface de Programmation Applicative

Ensemble normalisé de classes, méthodes ou fonctions qui sert de façade par laquelle un logiciel abstrait et offre des services à d'autres logiciels.

Web APIs disponibles en JS côté client

API de navigateur

- Intégrées au navigateur Web, eg. API de géolocalisation.

API de parties tierces

- Accessibles depuis un site Web, eg. API Twitter.

Relations entre JS, APIs et autres artefacts JS

JavaScript

- Langage haut niveau intégré aux navigateurs.
- Egalement disponible dans d'autres environnements (par ex. Node.js).

API du navigateur

- Intégrées dans le navigateur au dessus de JS (par ex. API DOM).

API tierces

- Intégrées à des plateformes tierces pour utiliser leurs fonctionnalités dans vos propres pages Web (par ex. Twitter).

Bibliothèques JS

- Fichier(s) JS contenant des fonctions utiles à l'écriture de fonctionnalités courantes (par ex. React).

Modèles/cadriciels JS

- Paquets HTML/CSS/JS/... à installer pour écrire une application Web entière (par ex. Angular). Contrairement aux bibliothèques, les modèles appellent le code développé (Inversion de Contrôle).

APIs de navigateur les plus couramment utilisées

Manipulation HTML et CSS

- DOM, CSSOM (pendant du DOM pour le CSS).

Récupération de données du serveur (AJAX)

- XMLHttpRequest, Fetch.

Dessin et manipulation de graphiques

- Canvas, WebGL.

Audio et vidéo

- HTMLMediaElement, Web Audio, Web RTC.

Périphériques

- Géolocalisation, Notifications, Vibrations, ...

Stockage de données côté client

- Web Storage (paires clé-valeur), IndexedDB (données structurées et indexées dont fichiers/blobs).

Autres

- Intl (internationalisation), WebWorkers (calcul parallèle)

Fonctionnement des APIs

Caractéristiques communes

- Fondées sur des objets.
- Points d'entrée identifiables.
- Utilisent des évènements pour réagir aux changements d'état.
- Peuvent imposer des mécanismes de sécurité supplémentaires.

Les APIs sont fondées sur des objets

Interagissent avec le code en utilisant un/des objets(s) JS qui servent de conteneurs pour :

- Les données utilisées par l'API : contenues dans des propriétés d'objet.
- Les fonctionnalités mises à disposition : contenues dans des méthodes d'objets.

Exemple : objets de l'API Géolocalisation

- `Geolocation` : contient 3 méthodes de récupération des données géographiques.
- `Position` : contient un objet `Coordinates` représentant la position de l'appareil horodatée.
- `Coordinates` : contient latitude, longitude, vitesse, direction du mouvement, ...

Points d'entrée des APIs

Cas simples

- Propriété `navigator.geolocation` qui renvoie l'objet `Geolocation` pour l'API de géolocalisation.
- Objet `Document` pour l'API DOM.

Cas complexes

- Appel de `getContext()` sur une référence à l'élément `<canvas>` (`HTMLCanvasElement`) sur lequel on veut dessiner.

Les APIs utilisent des évènements

Exemple avec `XMLHttpRequest`

Objet représentant une requête HTTP au serveur pour récupérer une ressource (texte, JSON, XML, ...).

- Offre différents gestionnaires d'évènements : eg. `onload` en cas de réponse récupérée avec succès.

apis-xhr.js

```
1 var request = new XMLHttpRequest();
2 request.addEventListener("load", function() {
3   console.log(JSON.stringify(request.response));
4 });
5 request.open('GET', "apis-xhr.php");
6 request.responseType = 'json';
7 request.send();
```

apis-xhr.php

```
1  class A
2  {
3      public $name;
4      function __construct($n)
5      {
6          $this->name = $n;
7      }
8  }
9  $a = new A("Cesar");
10 echo json_encode($a);
```

APIs et règles de sécurité

Les API sont sujettes aux mêmes règles que JS

Exemple : la règle same-origin policy.

Les API peuvent imposer des règles supplémentaires

- Usage imposé du protocole HTTPS.
- Autorisation d'activation : géolocalisation, notifications, etc.
- Etc.

CM JS : Sécurité : CORS et CSP

Sécurité : CORS et CSP

Origine de page web

L'origine d'une page web (origin)

- Définie par le schéma (protocol), l'hôte (domain) et le port de l'URL utilisée.
- Certaines opérations requièrent une même origine (same-origin policy).
- Relaxable avec CORS.

Origine identique (same origin)

`http://example.com/app1/index.html` et `http://example.com`

`http://example.com/app1/index.html` et `http://example.com:80`

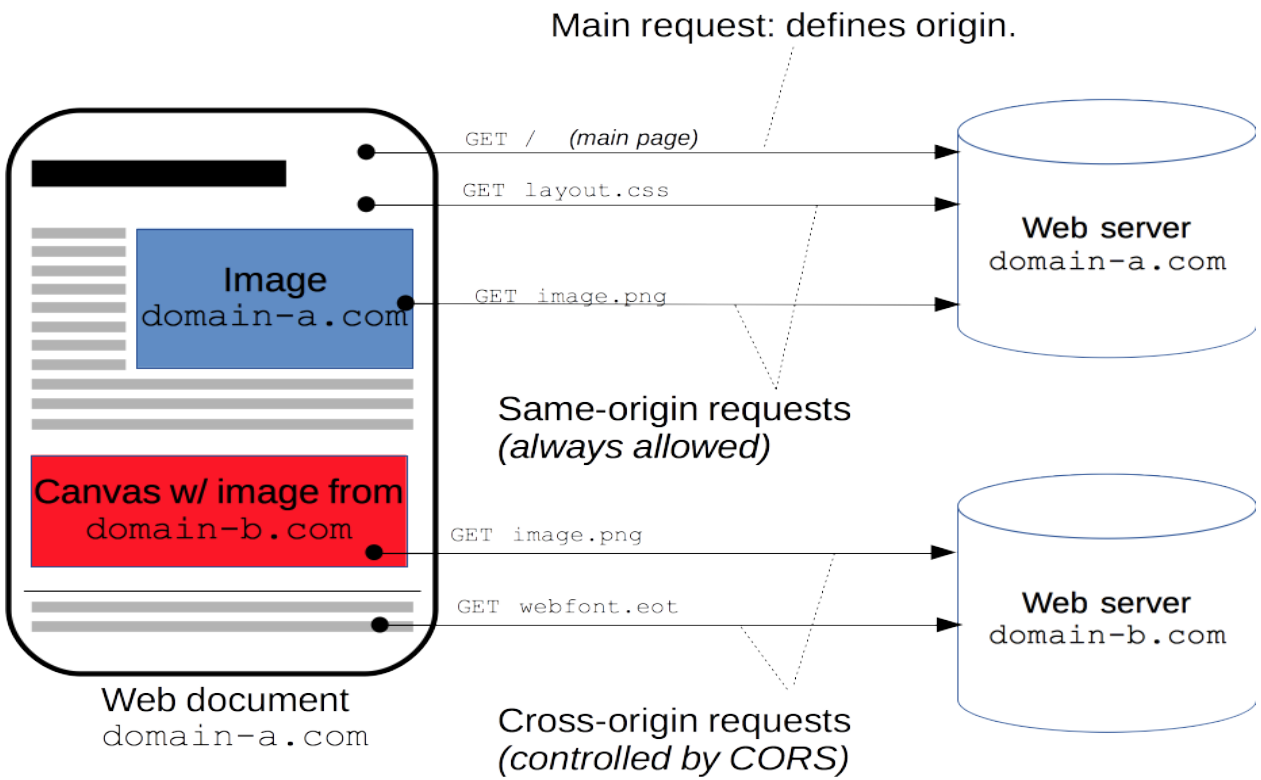
Origines différentes

`http://example.com/app1` et `https://example.com/app2`

`http://www.example.com` et `http://example.com:8080`

Cross-Origin Resource Sharing (CORS)

Mécanisme HTTP permettant à une page web d'accéder à des ressources d'origine différente



©MDN

Fonctionnement de CORS

Ajout d'en-têtes HTTP

- Permettant au serveur d'indiquer les origines autorisées.
- Imposant au navigateur de négocier avec le serveur (preflight) en cas de requêtes à effet de bord (GET, POST).

Exemples de requêtes cross-origin

- Invocations de `XHR` ou `fetch`.
- Téléchargement de polices en CSS : `@font-face { src: url(...);}`
- Téléchargement de textures avec WebGL.
- Téléchargement d'images/vidéos dans un canevas.

- CORS est intégré à `XHR/Fetch` : pas d'en-têtes à programmer.
- Les erreurs CORS ne sont pas relayées à JS.

CORS : exemple

Code JS servi par `https://foo.example`

security-cors-simple-request.js

```
1 const xhr = new XMLHttpRequest();
2 const url = 'https://bar.other/resources/public-data/';
3 xhr.open('GET', url);
4 xhr.onreadystatechange = someHandler;
5 xhr.send();
```

Ressource accessible depuis n'importe quel domaine

`https://bar.other` renvoie `Access-Control-Allow-Origin: *`

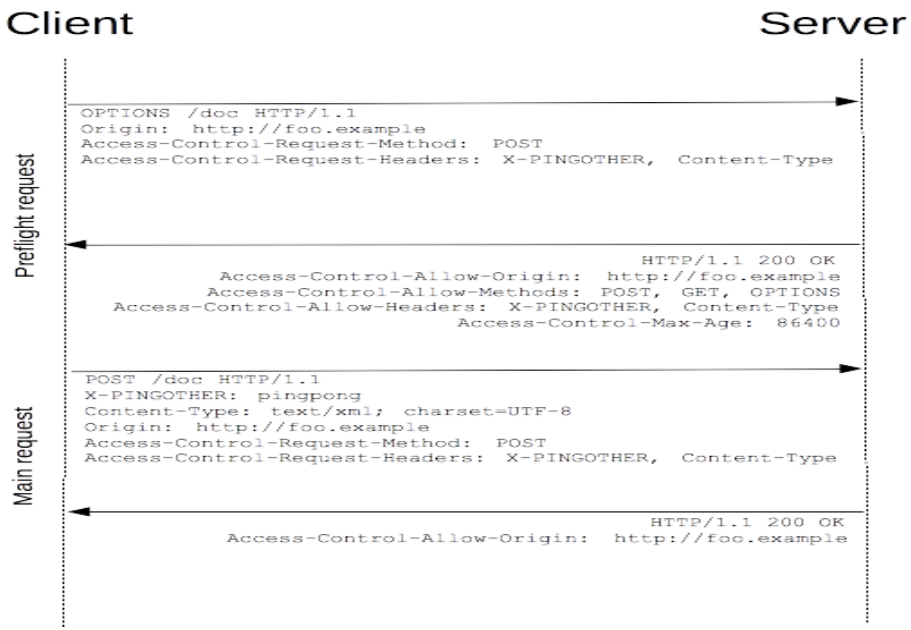
Ressource accessible uniquement depuis `https://foo.example`

`https://bar.other` renvoie `Access-Control-Allow-Origin: https://foo.example`

CORS : preflight request

security-cors-preflight-request.js

```
1 const xhr = new XMLHttpRequest();
2 xhr.open('POST', 'https://bar.other/resources/post-here/');
3 xhr.setRequestHeader('X-PINGOTHER','pingpong');
4 xhr.setRequestHeader('Content-Type','application/xml');
5 xhr.onreadystatechange = handler;
6 xhr.send('<person>Arun</person>');
```



©MDN

`Referrer`

- Adresse de la page web visitée précédente sur laquelle un lien a été suivi.
- Utilisé par les serveurs pour la connexion, le cache, l'analyse de navigation, etc

Content-Security Policy (CSP)

Mécanisme HTTP pour se protéger d'attaques XSS ou d'injection de données

- Un navigateur implémentant CSP peut fonctionner avec un serveur sans support CSP et inversement.
- Les navigateurs utilisent par défaut la *same-origin policy*.

Principes

- Le serveur communique au navigateur les domaines de confiance (whitelisting) selon le type de ressources : scripts, polices, images, ...
- Le navigateur n'exécute/ne charge que les scripts/fichiers des domaines autorisés.

Par défaut, CSP n'autorise pas scripts et styles en ligne (`<script>`, `<style>`, `onclick` ...) et la fonction JS `eval()`.

Politiques CSP

2 alternatives pour “activer” CSP dans un serveur

Le serveur communique sa politique *policy*

- Par ajout aux réponses HTTP de l'en-tête

  `Content-Security-Policy: policy`

- Par insertion dans les pages HTML servies de

  `<meta http-equiv="Content-Security-Policy" content="policy">`

Politique CSP : une suite de directives restreignant chacune l'accès à un certain type de ressources

```
default-src 'self'

default-src 'self' *.trusted.com

default-src 'self'; img-src *; media-src media1.com media2.com; script-src
userscripts.example.com

default-src https://onlinebanking.jumbobank.com

default-src 'self' *.mailsite.com; img-src *
```

CSP : exemple (1)

security-csp.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="UTF-8">
6      <title>Tests CSP (fichier importé en PHP)</title>
7      <script defer src="security-csp.js"></script>
8  </head>
9
10 <body>
11     <button id="btn" onclick="alert('clic - inline HTML')">Click!</button>
12     <script>
13         document.querySelector("#btn").addEventListener('click', () => {
14             alert('Clic - inline JS!');
15             let pElem = document.createElement('p');
16             pElem.textContent = 'This is a newly-added paragraph.';
17             document.body.appendChild(pElem);
18         })
19     </script>
20     <img src="coffee.jpg" alt="no coffee" />
21     <img
src="https://upload.wikimedia.org/wikipedia/commons/4/41/Ch%C3%A2teau_de_Montsoreau%2C_depuis_la_rive_droite_de
height="200px" alt="logo UA" />
22 </body>
23
24 </html>
```

CSP : exemple (2)

security-csp.php

```php
// pas de blocage : renvoie les erreurs à security-csp-parser.php
header("Content-Security-Policy-Report-Only: default-src 'self'; report-uri security-csp-parser.php;");
$htmlfile = file_get_contents("security-csp.html");
$html = <<<HTML
$htmlfile
HTML;
echo $html;
```

security-csp-strict.php

```php
$imgdom = "https://upload.wikimedia.org";
// ne bloquera pas l'exécution de security-csp.js
// ni de le chargement de l'image de upload.wikimedia.org
header("Content-Security-Policy: default-src 'self'; img-src $imgdom");
$htmlfile = file_get_contents("security-csp.html");
$html = <<<HTML
$htmlfile
HTML;
echo $html;
```

OAuth2

Délégation d'autorisation

Pour accorder un accès limité sur une ressource à une application tierce.
Exemples : utiliser l'API d'un site pour le compte d'un utilisateur (Google APIs ...).

Côté serveur

Quelques outils de développement et de tests

- PHP : Guzzle (HTTP Client), Curl
- Clients lourds : SoapUI, JMeter, Postman ...
- Shell : curl + jq