Web Academy

Manipulation de page *web* en *JS*

Alexis EL MRINI (@alexis-elmrini)
Tristan LE GODAIS (@PolariTOON)

Sommaire

I. Présentation du *JS*

1. Vocation du *JS*

Le langage *JS* (*JavaScript*) est un langage interprété, orienté objet à prototype et fonctionnel dont le typage est dynamique.

Il est apparu en 1996 dans le navigateur *Netscape Navigator* dans le but de manipuler les pages *web* et s'est rapidement établi comme le langage de script du *web*. Depuis 1997, il est standardisé et évolue sous le nom d'*ECMAScript*. Avec l'émergence de projets comme *GNOME Shell*, *Node.js*, *Moddable*, *Deno* ou *QuickJS*, l'environnement d'exécution du langage ne se limite plus seulement aux navigateurs.

2. Intégration dans un document

La manière d'intégrer un script *ECMAScript* à un document *HTML* dépend à la fois de la variante du langage utilisée et du moment de son évaluation. On distingue ainsi script classique (non strict ou strict) ou modulaire, script interne ou externe et script synchrone (bloquant), différé (récupéré en parallèle de l'analyse du document mais évalué à sa fin) ou asynchrone (récupéré en parallèle de l'analyse et évalué dès que possible, bloquant ainsi la suite de l'analyse).

Certaines combinaisons sont incompatibles. On comptabilise ainsi huit manières d'intégrer un script à un document.

Syntaxe	Localisation	Exécution
`<script>…</script>`	Interne	Synchrone
`<script src="…"></script>`	Externe	Synchrone
`<script defer="" src="…"></script>`		Différée
`<script async="" src="…"></script>`		Asynchrone

Formes d'intégration d'un script classique à un document

Syntaxe	Localisation	Exécution
`<script type="module">…</script>`	Interne	Différée
`<script type="module" async="">…</script>`		Asynchrone
`<script type="module" src="…"></script>`	Externe	Différée
`<script type="module" async="" src="…"></script>`		Asynchrone

Formes d'intégration d'un script modulaire à un document

3. Bonnes pratiques

À l'instar des *CSS*, il est préférable de séparer le code *ECMAScript* du code *HTML* en utilisant un script externe.

Il est également généralement recommandé de différer l'évaluation d'un script pour ne pas bloquer le chargement du reste d'une page *web* en utilisant l'attribut `defer="…"` dans le cas d'un script classique.

L'utilisation d'un script modulaire a l'avantage de permettre d'utiliser certaines des dernières fonctionnalités du langage (notamment l'import et l'export statique de modules). Si toutefois, vous préférez utiliser un script classique, alors le mode strict est fortement recommandé. Pour l'activer, il suffit de débuter le script par `"use strict";`.

II. Types d'expressions

Il existe actuellement huit types natifs en *JS*, dont sept types de valeurs primitives : *Undefined*, *Null*, *Boolean*, *BigInt*, *Number*, *String* et *Symbol*. Le reste des valeurs constitue le type objet : *Object*.

Parmi les objets, on trouve différentes sous-classes natives, notamment : *Function*, *Date*, *RegExp*, *Array*, *Map*, *Set*...

Certains types n'ont pas de notation litérale. Ci-après sont recensées des exemples de notations litérales pour les types qui en possède une.

Type	Notation
Null	`null`
Boolean	`false`, `true`

Aperçu des mots-clés désignant des valeurs primitives

Type	Notation
BigInt	`1n`, `2305843009213693951n`...
Number	`1`, `3.14`...
String	`""`, `'abc'`, `` `def` ``...

Aperçu de la notation litérale des autres valeurs primitives

Type	Objet vide	Notation courte	Notation longue
RegExp	`/(?:)/u`	`/[abc]/u`	`/(?:a|b|c)/u`
Array	`[]`	`[a, b]`	`[a, b,]`
Object	`{}`	`{a, b}`	`{["a"]: a, ["b"]: b,}`
Function	`() => {}`	`a => a`	`(a,) => {return a;}`

Aperçu de la notation litérale des objets

III. Instructions

1. Déclarations de variables

Il existe divers moyens de déclarer des identifiants de variables en *JS*, chacune possèdant ses propres particularités : via les instructions `let …;`, `const …;`, `import …;`, `export …;`, `var …;`, `class … {…}`, `function …(…) …`, `function* …(…) …`, `async function …(…) …` ou `async function* …(…) …`, ou en tant que paramètre formel d'une fonction.

En pratique, les instructions `let …;`, `const …;`, `import …;`, `export …;` et les paramètres formels de fonctions suffisent à mimer le comportement des autres formes de déclarations (sans le calquer pour autant).

`let …;`

- Cette instruction permet de déclarer une variable locale au bloc d'instructions englobant :

```
let a = 0, b;
console.log(a); // 0
console.log(b); // undefined
```

Déclaration `let …;`

La variable est déclarée et initialisée au niveau de l'instruction (à `undefined` par défaut) et peut être réaffectée. L'utiliser avant ou la redéclarer produit cependant une erreur.

`const …;`

- Cette instruction permet de déclarer une constante locale au bloc d'instructions englobant :

```
const a = 0, b = "";
console.log(a); // 0
console.log(b); // ""
```

Déclaration `const`

Comme pour l'instruction `let …;`, la constante est déclarée et initialisée au niveau de l'instruction, mais cette fois-ci, une valeur est requise et la constante ne peut pas être réaffectée. L'utiliser avant ou la redéclarer produit également une erreur.

`var …;`

- Cette instruction permet de déclarer une variable locale au corps de la fonction englobante :

```
console.log(a); // undefined
var a = 0, b;
console.log(a); // 0
```

Déclaration `var …;`

La variable est déclarée au début du corps de la fonction englobante et initialisée à `undefined`. Si une valeur est donnée, la variable est affectée au niveau de l'instruction. L'utiliser avant ou la redéclarer avec `var …;` est possible.

`function …(…) …`

- C'est la manière la plus courante de déclarer des fonctions locales au corps de la fonction englobante en *JS* :

```
console.log(a); // function a()
function a() {…}
console.log(a); // function a()
```

Déclaration `function …(…) …`

La fonction est déclarée et initialisée au début du corps de la fonction englobante. Il est donc possible de l'utiliser avant même l'instruction. La redéclarer écrase juste la définition précédente, rendant celle-ci inutile.

L'instruction `var …;` est présente depuis le début du langage *ECMAScript*, tandis que les instructions `let …;` et `const …;` sont des ajouts plus récents (2015) visant à donner une alternative à `var …;` sans ses défauts de conception.

Il est donc recommandé d'utiliser plutôt les instructions `let …;` et `const …;`. De plus, on privilégie généralement `const …;` si `let …;` n'est pas nécessaire.

Les fonctions et les classes constituent des objets de première classe en *JS*. Ils peuvent donc être déclarés sans utiliser les instructions spécifiques, voire même être utilisés directement en tant qu'expressions.

2. Blocs d'instructions

En *ECMAScript*, les instructions sont ponctuées par un point-virgule `;` terminal (optionnel dans certains cas), sauf dans le cas d'un bloc d'instructions. Délimité par une accolade ouvrante `{` au début et une accolade fermante `}` à la fin, il permet de regrouper plusieurs instructions :

```
{
    …;
    …;
    …;
}
```

Bloc d'instructions

3. Structures de contrôle classiques

Les syntaxes des structures de contrôle du langage *C* (dont *ECMAScript* s'inspire) sont supportées :

```
if (…) …
if (…) … else …
switch (…) {…}
while (…) …
do … while (…);
for (…; …; …) …
```

Structures reprises du langage C

4. Autres syntaxes de boucles

`for (… in …) …`

- Il s'agit d'une boucle `for (…) …` particulière qui permet d'itérer sur les clés (qui ne sont pas des symboles) des propriétés (énumérables) d'un objet dans un ordre arbitraire :

```
const o = {a: 42, b: 13};
for (const k in o) {
    console.log(k, o[k]); // "a" 42 puis "b" 13
}
```

Boucle `for (… in …) …`

`for (… of …) …`

- Il s'agit d'une boucle `for (…) …` particulière qui permet de parcourir n'importe quel objet itérable selon l'ordre qu'il a défini :

```
const o = [42, 13];
for (const v of o) {
    console.log(v); // 42 puis 13
}
```

Boucle *`for (… of …) …`*

5. Étiquettes

Toute instruction peut être étiquettée de la même manière qu'en *C*.

L'instruction `break …;` peut être utilisée avec une telle étiquette n'importe où.

Au sein d'un boucle, les instructions `break …;` et `continue …;` peuvent être utilisées avec une étiquette optionnelle.

Au sein d'une structure `switch (…) {…}`, les étiquettes `case …` (avec n'importe quelle expression) et `default` peuvent être utilisées, ainsi que l'instruction `break …;` avec une étiquette optionnelle.

6. Gestion des erreurs

Le concept d'erreurs en *JS* est similaire à celui d'exceptions en *Java*, et par conséquent leur signalement et leur traitement également :

```
try {
    throw new Error("Custom error");
} catch (error) {
    console.error(error.message);
} finally {
    console.log("Always executed");
}
```

Instruction `throw …;` *et structure* `try {…} catch (…) {…} finally {…}`

IV. Opérations

1. Affectations

L'opérateur `... = ...` affecte le résultat du calcul du membre droit au membre de gauche et retourne ce résultat.

Opérateur	Opération	Type de retour
`... = ...`	Affectation simple	N'importe lequel

Affectations

2. Concaténations

L'opérateur `… + …` tente de convertir préalablement ses opérandes en chaînes de caractères (de type *String*) puis les concatène.

Opérateur	Opération	Type de retour
`… + …`, `… += …`	Concaténation	*String*

Concaténations

3. Opérations arithmétiques

Les opérateurs `… + …`, `… - …`, `… * …`, `… / …`, `… % …`, `… ** …`, `+…`, `-…`, `++…`, `--…`, `…++` et `…--` tentent de convertir préalablement leurs opérandes en nombres (de type *Number*), s'ils ne sont pas de type *BigInt*, puis effectuent un calcul arithmétique.

Opérateur	Opération	Type de retour
`… + …`, `… += …`, `… - …`, `… -= …`	Addition, soustraction	*BigInt* ou *Number*
`… * …`, `… *= …`, `… / …`, `… /= …`, `… % …`, `… %= …`	Multiplication, division, reste de la division euclidienne	
`… ** …`, `… **= …`	Exponentiation	

Opérations arithmétiques binaires

Opérateur	Opération	Type de retour
`+...`	Conversion en nombre	*Number*
`-...`	Opposé	*BigInt* ou *Number*

Opérations arithmétiques unaires

Opérateur	Opération	Type de retour
`++…`, `--…`	Pré-incrémentation, pré-décrémentation	*BigInt* ou *Number*
`…++`, `…--`	Post-incrémentation, post-décrémentation	

Opérations arithmétiques unaires à effets de bord

4. Opérations bit à bit

Les opérateurs `… | …`, `… ^ …`, `… & …`, `… << …`, `… >> …`, `… >>> …` et `~…` tentent de convertir préalablement leurs opérandes en entiers (de type *Number*), s'ils ne sont pas de type *BigInt*, puis effectuent un calcul bit à bit.

Opérateur	Opération	Type de retour
`… | …`, `… |= …`, `… ^ …`, `… ^= …`, `… & …`, `… &= …`	Disjonction bit à bit, disjonction exclusive bit à bit, conjonction bit à bit	*BigInt* ou *Number* (entier)
`… << …`, `… <<= …`, `… >> …`, `… >>= …`	Décalage à gauche, décalage à droite	
`… >>> …`, `… >>>= …`	Décalage à droite non signé	*Number* (entier)

Opérations bit à bit binaires

Opérateur	Opération	Type de retour
`~...`	Négation bit à bit	*BigInt* ou *Number* (entier)

Opérations bit à bit unaires

5. Opérations logiques

Les opérateurs `… ? … : …`, `… || …`, `… && …` et `!…` tentent de convertir préalablement certains de leurs opérandes en booléens (de type *Boolean*).

Opérateur	Opération	Type de retour
`… ? … : …`	Calcul conditionnel	N'importe lequel
`… || …`, `… && …`	Disjonction logique, conjonction logique	
`!…`	Négation logique	*Boolean*
`!…`	Négation logique	*Boolean*

Opérations logiques

6. Comparaisons

Les opérateurs `… === …` et `… !== …` n'effectuent aucune conversion préalable de leurs opérandes, au contraire des opérateurs `… == …` et `… != …`.

Les opérateurs `… <= …`, `… < …`, `… >= …` et `… > …` tentent de convertir préalablement leurs opérandes en valeurs primitives (de type *BigInt*, *Number* ou *String*).

Opérateur	Opération	Type de retour
`… == …`, `… === …`, `… != …`, `… !== …`	Égalité, égalité stricte, inégalité, inégalité stricte	*Boolean*
`… <= …`, `… < …`, `… >= …`, `… > …`	Infériorité, infériorité stricte, supériorité, supériorité stricte	

Opérations logiques

7. Fusion nulle

L'opérateur `… ?? …` est similaire à l'opérateur de disjonction et n'évalue et retourne son opérande droit que si l'opérande gauche est de type *Undefined* ou *Null*.

Opérateur	Opération	Type de retour
`… ?? …`	Fusion nulle	N'importe lequel

Fusion nulle

V. Bibliothèque standard

1. Nombres

Les nombres (`Number`) peuvent être manipulés via un ensemble de fonctions directement intégré au langage, et notamment via l'espace de nom `Math`.

Constantes spéciales

- Les accès `Number.NaN` (`NaN`), `Number.NEGATIVE_INFINITY` (`-Infinity`) et `Number.NEGATIVE_INFINITY` (`Infinity`) représentent respectivement une valeur *not a number*, l'infini négatif et l'infini positif ; d'autres constantes caractéristiques de la représentation flottante sont également disponibles.

Catégories de nombres

- Les appels `Number.isNaN()`, `Number.isFinite()` et `Number.isInteger()` permettent de tester si un nombre est respectivement une valeur *not a number*, un infini négatif ou positif et en entier.

Constantes mathématiques

- Les accès `Math.E`, `Math.PI` représentent les nombres e et π ; d'autres constantes récurrentes en mathématiques sont également disponibles.

Extrema

- Les appels `Math.min(…)` et `Math.max(…)` permettent de rechercher au sein d'une séquence de nombres respectivement le minimum et le maximum.

Conversions en entiers

- Les appels `Math.trunc(v)`, `Math.floor(t)` et `Math.ceil(u)`, `Math.round(w)` correspondent au calculs respectivement de la troncature, de la partie entière inférieure, de la fpartie entière supérieure et de l'arrondi.

Module et argument

- Les appels `Math.hypot(…)` et `Math.atan2(x, y)` correspondent au calculs respectivement de la norme euclidienne d'une séquence de nombres et de l'arc tangente du quotient de deux nombres.

Fonctions trigonométriques

- Les appels `Math.cos(a)`, `Math.sin(b)`, `Math.tan(c)`, `Math.acos(x)`, `Math.asin(y)` et `Math.atan(z)` correspondent aux appels des fonctions mathématiques du même nom ; leurs équivalents hyperboliques sont également disponibles.

Exponentielle et logarithme

- Les appels `Math.exp(a)` et `Math.log(x)` correspondent aux appels respectivement de la fonction exponentielle et de la fonction logarithme naturel ; des variantes sont égalements disponibles.

Fonction pseudo-aléatoire

- L'appel `Math.random()` permet de tirer aléatoirement un nombre entre *0* inclus et *1* exclus selon une distribution supposée uniforme.

2. Chaînes de caractères

Les chaînes de caractères (`String`) peuvent être manipulées via un ensemble de fonctions directement intégré au langage. Elle ont également la particularité d'être itérables (par exemple avec une boucle).

Contenu et longueur

- L'accès `"…"[k]` permet d'accéder au `k`e caractère d'une chaîne tandis que sa longueur s'obtient avec l'accès `"…".length` ; les chaînes de caractères étant immuables, ces propriétés sont disponibles en lecture seule.

Concaténation, extraction et répétition

- Les appels `"…".concat(…)`, `"…".slice(i, j)` et `"…".repeat(l)` permettent respectivement de concaténer une séquence de chaînes à une chaîne, d'extraire la sous-chaîne débutant à la position `i` et de longueur maximale `j` d'une chaîne et de concaténer `l` copies d'un chaîne.

Remplissage

- Les appels `"…".padStart(m, "…")` et `"…".padEnd(n, "…")` permettent de compléter une chaîne avec une sous-chaîne respectivement au début jusqu'à la longueur `m` et à la fin jusqu'à la longueur `n`.

Rognage

- Les appels `"…".trimStart()`, `"…".trimEnd()` et `"…".trim()` permettent de retirer d'une chaîne les blancs respectivement au début, à la fin et aux deux extrémités.

Recherche de sous-chaîne

- Les appels `"…".indexOf("…", i)` et `"…".lastIndexOf("…", j)` permettent de rechercher au sein d'une chaîne la position d'une sous-chaîne respectivement à partir de la position `i` et jusqu'à la position `j`.

Présence de sous-chaîne

- Les appels `"…".startsWith("…", i)` et `"…".endsWith("…", j)` et `"…".includes("…")` permettent de tester au sein d'une chaîne la présence d'une sous-chaîne respectivement à partir de la position `i`, jusqu'à la position `j` et n'importe où.

3. Tableaux

Les tableaux (`Array`) peuvent être manipulées à la fois de manière fonctionnelle et comme des objets via un ensemble de fonctions directement intégré au langage. Ils sont également itérables.

Contenu et longueur

- L'accès `[…][k]` permet d'accéder au `k`e élément d'un tableau tandis que sa longueur s'obtient avec l'accès `[…].length` ; les tableaux étant mutables et flexibles, ces propriétés sont disponibles en lecture et écriture, ce qui permet notamment de les vider.

Concaténation, extraction et aplatissement

- Les appels `[…].concat(…)`, `[…].slice(i, j)` et `[…].flat(l)` permettent respectivement de concaténer une séquence de tableaux à un tablau, d'extraire le sous-tableau débutant à la position `i` et de longueur maximale `j` d'un tableau et d'aplatir `l` niveaux d'imbrication de tableaux.

Remplissage (empilage, enfilage)

- Les appels `[…].unshift(…)` et `[…].push(…)` permettent de compléter un tableau avec une séquence d'éléments respectivement au début et à la fin.

Rognage (désempilage, désenfilage)

- Les appels `[…].shift()` et `[…].pop()` permettent de retirer d'un tableau respectivement le premier élément et le dernier élément.

Recherche d'élément

- Les appels `[…].indexOf(…, i)` et `[…].lastIndexOf(…, j)` permettent de rechercher au sein d'un tableau la position d'un élément respectivement à partir de la position `i` et jusqu'à la position `j`.

Présence d'élément

- L'appel `[…].includes(…)` permet de tester au sein d'un tableau la présence d'un élément.

Renversement et tri

- Les appels `[…].reverse()` et `[…].sort((…) => {…})` permettent respectivement de retourner un tableau en place et d'ordonner un tableau en place de manière stable selon une relation d'ordre.

Quantification

- Les appels `[…].some((…) => {…})` et `[…].every((…) => {…})` permettent de tester au sein d'un tableau si une condition est vérifiée par respectivement au moins un élément et chaque élément.

Association et sélection

- Les appels `[…].map((…) => {…})` et `[…].filter((…) => {…})` permettent de créer à partir d'un tableau un nouveau tableau respectivement en associant à chaque élément un nouvel élément et en sélectionnant les éléments vérifiant une condition ; de nombreuses variantes, combinaisons et généralisations de ces fonctions sont disponibles.

4. *JavaScript Object Notation*

Le format *JSON* est utilisé pour stocker des données structurées non cycliques (valeur nulle, booléens, nombres, chaînes de caractères, listes et dictionnaires) et est notamment employé dans le cadre d'APIs de type *Rest*.

```
{
    "nom": "Villers-lès-Nancy",
    "population": 14455,
    "code": "54578"
}
```

Exemple de fichier JSON *retourné par* `geo.api.gouv.fr`

La syntaxe du format de données *JSON* reprend celle des objets en *JS* et en forme ainsi un sous-ensemble. Ce format à la syntaxe immuable est manipulable grâce à deux fonctions disponibles via l'espace de nom du même nom.

Analyse

- Une chaîne de caractères au format *JSON* peut être analysée et convertie en objet à l'aide de l'appel `JSON.parse("…")`.

Reconstitution

- Un objet peut être converti en chaîne de caractères au format *JSON* à l'aide de l'appel `JSON.stringify(…)`.

VI. Bibliothèques récurrentes

1. Console

L'espace de nom `console` est disponible à la fois dans dans les navgateurs et dans *Node.js*, bien que ne faisant pas partie du langage *ECMAScript* à proprement parler. Il permet d'afficher des données (des objets *ECMAScript*) éventuellement formatées dans la console du navigateur ou dans le terminal sans faire directement appel au débogueur via l'instruction `debugger;`.

Les données sont généralement affichées avec un code couleur :

```
console.debug(…);
console.error(…);
console.info(…);
console.log(…);
console.warn(…);
```

Fonctions basiques de la console

Pour un débogage plus fin, on peut utiliser les fonctions suivantes qui permettent respectivement de vérifier la réalisation d'une condition et d'afficher la trace de la pile d'appel :

```
console.assert(…, …);
console.trace(…);
```

Fonctions avancées de la console

D'autres fonctions permettent de structurer plus aisément l'affichage des données :

```
console.clear();
console.table(…, …);
console.dir(…, …);
console.dirxml(…);
console.group(…);
console.groupCollapsed(…);
console.groupEnd();
```

Fonctions d'affichage structuré de la console

Pour finir, il est également possible de comptabiliser des occurences ou de mesurer des durées avec les fonctions suivantes :

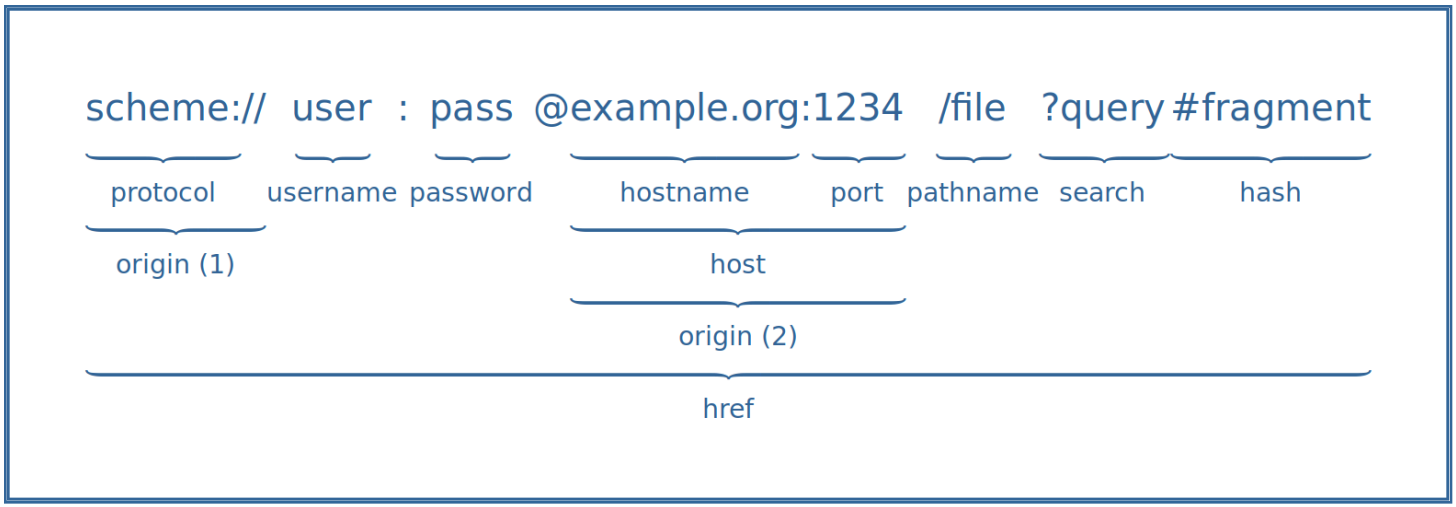
```
console.count("…");
console.countReset("…");
console.time("…");
console.timeLog("…", …);
console.timeEnd("…");
```

Fonctions de mesure de la console

2. *Uniform Resource Locator*

Les adresses de type *URL* servent à identifier et retrouver des ressources. La syntaxe varie selon le schéma employé (`https`, `file`, `data`, `blob`, `view-source`, `mailto`, `tel`...) qui correspond généralement à un nom de protocole.

La classe `URL`, présente à la fois dans les navigateurs et dans *Node.js*, permet d'analyser et de décomposer en un objet une adresse donnée sous forme de chaînes de caractères. Les différentes propriétés correspondent alors à la décomposition détaillée précédemment. Elles sont disponibles à la fois en lecture et en écriture.



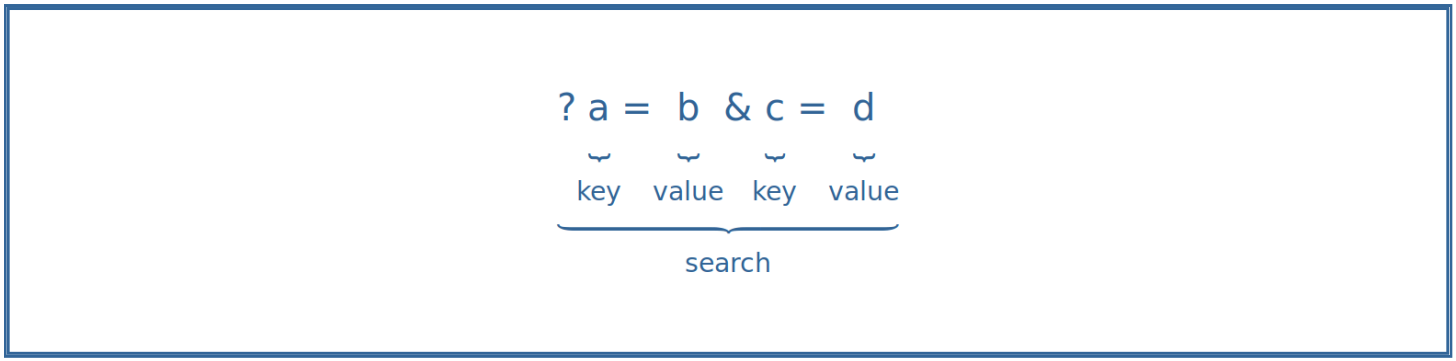
Anatomie d'une adresse URL

Cette classe résout automatiquement les adresses relatives (commençant généralement par `//`, `/`, `../`, `./`, `?` ou `#`).

```
new URL("//example.com", "https://example.org/dir/").href;
// "https://example.com/"
new URL("../", "https://example.org/dir/subdir/").href;
// "https://example.org/dir/"
```

Exemples de résolutions d'adresses relatives

Une propriété supplémentaire, `….searchParams`, instance de la classe `URLSearchParams`, permet de manipuler directement les paramètres d'*URL*, via des fonctions de lecture (`….get("…")`), d'écriture (`….set("…", "…")`), d'ajout (`….append("…", "…")`), de retrait (`….delete("…")`), de test de présence (`….has("…")`), de recherche (`….getAll("…")`), de tri (`….sort()`) ou d'itération (`….forEach((…) => {…})`).



Anatomie des paramètres d'une adresse URL

VII. APIs *web*

1. API *Fetch*

L'API *Fetch* n'est disponible nativement que dans les navigateurs mais des implémentations comme `node-fetch` existent pour *Node.js*. Il s'agit d'une alernative moderne à l'API *XMLHttpRequest* qui elle remonte au temps où l'on parlait encore de techniques *Ajax*.

Cette API consiste principalement en une fonction `fetch(…, {…})` qui permet d'envoyer des requêtes, de réceptionner des réponses et de les traiter, tout ceci de manière asynchrone, c'est-à-dire non bloquante.

On donne généralement à la fonction `fetch(…, {…})` une adresse de type *URL* ainsi qu'un dictionnaire d'options spécifiant par exemple l'en-tête de la requête (`headers`), son corps (`body`), sa méthode (`method`)… L'objet retourné expose la réponse sous forme de propriétés (`….headers`, `….body`, `….status`). Celle-ci peut être décodée comme un tampon, un texte, un format *JSON*, un formulaire ou un fichier grâce à des fonctions spécialisées.

```
const response = await fetch("https://api.github.com/repositories/
221418229");
const json = await response.json();
const stars = json.stargazers_count;
console.log(stars);
```

Exemple d'utilisation de l'API Fetch

2. *Document Object Model*

Le *DOM* est certainement l'une des plus anciennes APIs du *web*, mais aussi la plus importante ! Disponible nativement dans les navigateurs, elle peut aussi être utilisée dans *Node.js* grâce à l'implémentation `jsdom`.

Le *DOM* est essentiellement une représentation arborescente d'un document *XML* (en particulier *HTML*) manipulable via l'objet `document`. Chaque partie du document correspond ainsi à un objet de classe `Node`.

Le document en devient totalement manipulable et navigable via *ECMAScript*. C'est grâce au *DOM* qu'une page peut devenir interactive ! À l'aide d'un système d'évènements, on peut en effet réagir à une action de l'utilisateur (comme un clic souris) ou du navigateur en exécutant alors du code *ECMAScript*.

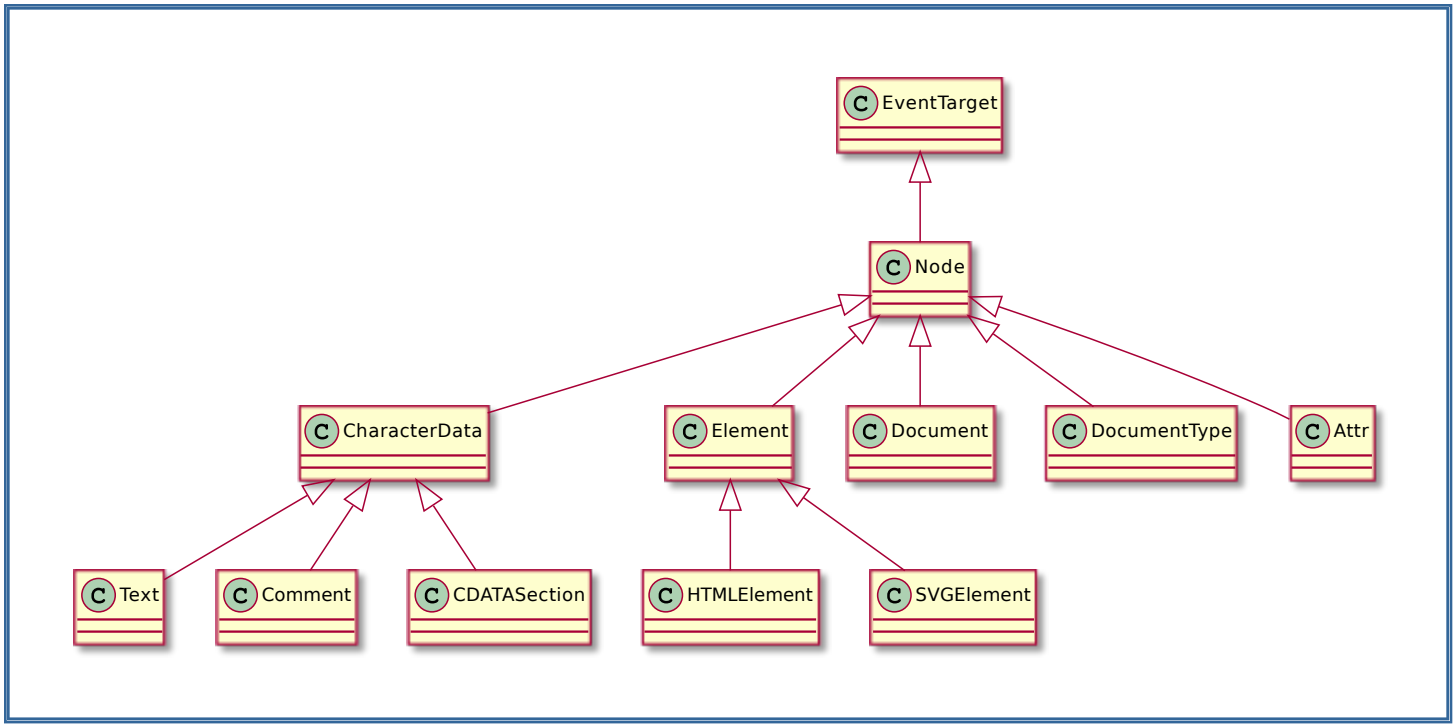


Diagramme des classes simplifié d'un document XML

Pour créer un élément dynamiquement, on utilise la fonction `document.createElement("…")` à laquelle on donne un nom de balise. L'élément ainsi créé n'est pas immédiatement inséré dans le document.

Avant d'ajouter l'élément au document via la fonction `….append(…)` par exemple, on peut le manipuler (lui ajouter des attributs, du contenu…) :

```
const element = document.createElement("button");
element.id = "toggle-button";
element.textContent = "Click me!";
document.body.append(element);
```

Création et insertion d'un élément dans le document

Pour rechercher un certain élément dans le document, plusieurs méthodes existent, mais la plus puissante est d'utiliser la fonction `….querySelector("…")` à laquelle on donne un sélecteur *CSS*.

Un moyen de réagir à un clic souris de l'utilisateur sur cet élément peut alors être d'utiliser la fonction `….addEventListener("…", (…) => {…})` :

```
const element = document.querySelector("#toggle-button");
element.addEventListener("click", (event) => {
    element.classList.toggle("active");
});
```

Recherche d'un élément dans le document et réaction au clic souris

Le *DOM* est une API très riche qui ne se limite pas à ces quelques fonctions. Chaque type d'élément possède ses propres spécificités et le *DOM* sert souvent de pont vers d'autres API comme les APIs *Canvas* ou *Web Audio* par exemple.