Web Academy

Manipulation de page *web* en *ECMAScript*

Sommaire

I. Présentation d'*ECMAScript*

1. Vocation d'*ECMAScript*

Le langage *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 *Node.js* ou *GNOME Shell* en 2009, 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 septs types natifs en *ECMAScript*, dont six types de valeurs primitives : *Undefined*, *Null*, *Boolean*, *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`
Number	`1`, `3.14`...
String	`""`, `'abc'`, `` `def` ``...

Aperçu de la notation litérale des 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 *ECMAScript*, 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 *ECMAScript* :

```
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 *ECMAScript*. 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.

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*) puis effectuent un calcul arithmétique.

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

Opérateurs arithmétiques binaires

Opérateur	Opération	Type de retour
`+…`, `-…`	Conversion en nombre, opposé	*Number*
`++…`, `--…`	Pré-incrémentation, pré-décrémentation	
`…++`, `…--`	Post-incrémentation, post-décrémentation	

Opérateurs arithmétiques unaires

4. Opérations bit à bit

Les opérateurs `… | …`, `… ^ …`, `… & …`, `… << …`, `… >> …`, `… >>> …` et `~…` tentent de convertir préalablement leurs opérandes en entiers (de type *Number*) 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
		Number (entier)

`… << …`, `… <<= …`, `… >> …`, `… >>= …`, `… >>> …`, `… >>>= …`	Décalage à gauche, décalage à droite, décalage à droite non signé	
`~…`	Négation bit à bit	

Opérations bit à bit

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*

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 *Number* ou *String*).

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

Opérations logiques