3: Boucles for

Qu'est-ce qu'une boucle ?

Une boucle permet de répeter plusieurs fois une séquence d'instuctions.

Pour afficher les nombres de `1` à `3` dans la console JavaScript, on pourrait utiliser les instructions suivantes:

```
console.log(1);
console.log(2);
console.log(3);
```

Par contre, le code deviendrait très fastidieux à écrire (et à lire) dans le cas où on voudrait afficher les nombres de `1` à `10000` !

Pour ce genre de répétition, le mot-clé `for` permet de définir une seule fois les instructions qui doivent êtres répétées, puis de spécifier combien de fois on souhaite qu'elles soient répétées.

Pour afficher les nombres de `1` à `10000`, il suffit donc d'écrire le code suivant:

```
for ( var i = 1; i <= 10000; i++ ) {
  console.log( i );
}
```

On pourrait traduire ce code de la manière suivante:

> Pour chaque valeur de `i`, croissant de `1` à `10000` (compris), afficher la valeur de `i` dans la console.

À quoi servent les boucles ?

Les boucles sont donc très utiles pour éviter les redondances dans un programme (ex: jouer 5 fois le même son, mettre tous les champs d'un formulaire en majuscules...), mais elles sont surtout indispensables dans de nombreuses applications courantes:

- Les jeux tour-par-tour consistent en une boucle qui se termine lorsqu'un joueur remporte la partie;
- Les jeux d'action utilisent une boucle permettant de mettre à jour l'affichage (frame par frame, pour utiliser la terminologie exacte) en fonction des actions du/des joueur(s);
- Ainsi que les algorithmes de tri et de manipulation de données utilisés dans 99% des logiciels.

Javascript fournit quatre mots-clés pour définir des boucles: `do`, `while`, `until` et `for`. La forme de boucle la plus courante est `for` car c'est la plus générique / adaptable. Nous allons donc seulement travailler avec des boucles `for` dans le cadre de ce cours.

Anatomie d'une boucle `for` en JavaScript

Reprenons l'exemple de boucle que nous avons vu plus haut:

```
for ( var i = 1 ; i <= 10000 ; i++ ) {
  console.log( i );
}
```

Cette boucle est définie par:

- l'usage du mot clé `for`;
- une liste d'instructions (saisie entre accolades `{}`) à répéter tant que la condition est vraie: `console.log( i );` (dans notre exemple, il n'y a qu'une seule instruction, mais on peut en mettre une infinité);
- une condition (expression conditionnelle, comme dans une condition `if`): `i <= 10000`;
- une instruction d'itération qui sera exécutée après chaque itération de la boucle: i++`(qui, ici, incrémente la valeur de` monNombre`, c'est à dire augmente sa valeur de` 1 `);
- et une instruction d'initialisation qui ne sera exécutée qu'une seule fois: `var i = 1` (ici, on créée une variable `i` et on lui affecte la valeur initiale `1`).

On appelle **itération** chaque répétition de la boucle.

Pour synthétiser, voici la syntaxe à utiliser pour définir une boucle `for` en JavaScript:

```
for( /* initialisation */ ; /* condition */ ; /* incrémentation */ ) {
  /* instructions à répeter */
}
```

À noter que, dans la plupart des cas, les boucles sont utilisées pour itérer:

- sur un intervalle (dans notre exemple: nombres entiers entre `1` et `10000`),
- ou sur une énumération de valeurs (ex: un tableau/Array, comme on le verra plus tard).

Traçage de l'exécution d'une boucle `for`

Afin de mieux comprendre le fonctionnement de la boucle `for` et de la manière de saisir ces trois paramètres, nous allons interpréter une boucle comme le fait un navigateur web (ou tout autre interpréteur JavaScript).

Prenons la boucle `for` suivante:

```
console.log('on va boucler');
for ( var i = 0; i < 4; i++ ) {
  console.log('i', i, i < 4);
}
console.log('on a fini de boucler');
```

Voici la manière dont elle va être interprétée et exécutée par la machine:

```
console.log('on va boucler');            // => affiche: on va boucler
```

```
// interprétation de la boucle => on commence par l'initialisation
var i = 0; // initialisation de la boucle, en affectant 0 à la variable i

// --- première itération de la boucle ---
i < 4 ?                         // condition vraie, car i vaut 0
console.log('i', i, i < 4);     // => affiche: i 0 true
i++;                            // incrémentation => i vaut maintenant 1

// --- seconde itération de la boucle ---
i < 4 ?                         // condition vraie, car 1 < 4
console.log('i', i, i < 4);     // => affiche: i 1 true
i++;                            // incrémentation => i vaut maintenant 2

// --- troisième itération de la boucle ---
i < 4 ?                         // condition vraie, car 2 < 4
console.log('i', i, i < 4);     // => affiche: i 2 true
i++;                            // incrémentation => i vaut maintenant 3

// --- quatrième itération de la boucle ---
i < 4 ?                         // condition vraie, car 3 < 4
console.log('i', i, i < 4);     // => affiche: i 3 true
i++;                            // incrémentation => i vaut maintenant 4

// --- cinquième itération de la boucle ---
i < 4 ?                         // condition fausse, car i==4 => fin de boucle

// boucle terminée => on interprète les instructions suivantes.
console.log('on a fini de boucler');     // => affiche: on a fini de boucler
```

Il est très pratique de décomposer une boucle de cette manière lorsqu'elle ne se comporte pas comme voulu. (*débogage*)

Interrompre l'exécution d'une boucle: `break`

Dans certains cas, il est pratique d'interrompre l'exécution d'une boucle, pendant l'exécution d'une de ses itérations (tel que définie entre les accolades de définition de la boucle).

```
for (var i = 0; i < 10; i++) {
  var commande = prompt('entrez une commande');
  if (commande === 'quitter') {
    break; // => on sort de la boucle for, la suite programme continue de
s'exécuter (après ses accolades du for)
  }
}
```

Cependant, l'usage de `break` est non recommandé, car il rend la logique plus complexe à comprendre en lisant le code. Il est généralement possible et plus élégant d'intégrer la condition de sortie dans la condition de la boucle.

```
var commande;
for (var i = 0; i < 10 && commande !== 'quitter'; i++) {
  commande = prompt('entrez une commande');
}
```

Ici, on a intégré la condition à l'aide de l'opérateur logique `&&`, donc la boucle continuera d'itérer tant que `i < 10` *ET* que `commande !== 'quitter'`.

Comment savoir si un nombre est multiple d'un autre ?

Pour savoir si un nombre est multiple de `3` et/ou de `5`, nous allons utiliser deux fonctions fournies ci-dessous:

```
function estMultipleDeTrois(nombre) {
  return nombre % 3 === 0;
}
function estMultipleDeCinq(nombre) {
  return nombre % 5 === 0;
}
```

Après avoir copié-collé la définition de ces deux fonctions dans la console JavaScript, vous pouvez les utiliser de la manière suivante:

```
estMultipleDeTrois(2); // => retourne false, car 2 n'est pas multiple de 3
estMultipleDeTrois(6); // => retourne true, car 6 est un multiple de 3
estMultipleDeCinq(6); // => retourne false, car 6 n'est pas un multiple de 5
estMultipleDeCinq(15); // => retourne true, car 15 est un multiple de 5
```

Vous pouvez alors appeler ces fonctions dans les parenthèses de vos conditions `if`, car, comme une expression de comparaison de valeurs, un appel de fonction retourne une valeur qui est vraie (`true`) ou fausse (`false`).

Exemple:

```
var monNombre = 5; // valeur fournie en guise d'exemple
if (estMultipleDeCinq(monNombre)) {
  console.log('monNombre est multiple de 5');
} else {
  console.log('monNombre n\'est pas multiple de 5');
}
```