Les boucles sous toutes leurs formes.

Le JavaScript dispose d'une demi-dizaine d'instructions distinctes permettant d'effectuer une boucle sur une variable. Quelles sont les méthodes les plus adaptées aux différents types de valeurs ? Quelles sont les implications de telle ou telle méthode ? Dans quel contexte préférer une instruction plutôt qu'une autre ? Autant de questions auxquelles nous allons apporter une réponse.

Le JavaScript comporte de très nombreuses instructions permettant d'effectuer une boucle sur une valeur. Cependant, si nous considérons seulement celles dont c'est l'objet premier – en omettant celles dont la boucle est inhérente à leur fonctionnement (`.map` , `.filter` …) – nous en dénombrons exactement cinq :

- **For**
- **For of**
- **For in**
- **While**
- **Do while**

À cela s'ajoute une méthode dont disposent de nombreux itérables : `.forEach` . Voyons donc quelles sont les différences fondamentales entre ces différentes instructions.

1. While, le basique.

   Le while est l'instruction de boucle la plus simple, tant que la condition retourne true, la boucle se poursuit.

```
const countUntil = 10;
let i = 0;

while (i <= countUntil) {
    console.log(i++); // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}
```

Étant donné que la condition attendue est une expression, on peut tout à fait y placer une expression qui modifie notre structure de contrôle, comme incrémenter notre variable par exemple.

```
const countUntil = 10;
let i = 0;

while (i++ < countUntil) {
    console.log(i); // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}
```

Vous avez peut-être remarqué que la comparaison est passé d'un inférieur ou égal à un exclusivement inférieur à. En effet, la condition est exécutée avant chaque tour de boucle. C'est d'ailleurs pour cela que le premier exemple compte à partir de zéro tandis que le second commence à un.

De ce fait, la première fois que la condition est évaluée, i vaut 0, il est incrémenté et le corps de la boucle est exécuté. Lors de la dernière exécution, i vaut 9, est incrémenté et la boucle s'exécute. Enfin, la condition est de nouveau exécutée mais i vaut cette fois 10 donc la boucle s'arrête là.

2. For

Elle prend en paramètres trois expressions optionnelles, séparées par des points virgules :

- Initialisation : valeur à initialiser avant le démarrage de la boucle,
- Condition : condition qui détermine si la boucle continue ou s'arrête, elle est évaluée avant chaque tour de boucle,
- Expression finale : expression évaluée après chaque tour de boucle.

```
const test = 5;

for (let i = 0; i < test; i++) {
    console.log(i); // 0, 1, 2, 3, 4
}
```

3. For in, for of pour les objets :

La boucle for in permet de boucler sur les propriétés énumérables et non Symbol d'un objet.

```
const person = {
    firstname: 'Mickey',
    lastname: 'Mouse',
    nickname: 'Mick'
};

for (const propt in person) {
    console.log(propt); // "firstname", "lastname", "nickname"
    console.log(person[propt]); // "Mickey", "Mouse", "Mick"
}
```

la boucle for..of est plus récente. Cette dernière boucle sur les valeurs d'un itérable (objet, tableau, Set, Map, NodeList…).

```
const person = {
    firstname: 'Mickey',
    lastname: 'Mouse',
    nickname: 'Mick'
};

for (const val of person) {
    console.log(val); // "Mickey", "Mouse", "Mick"
}
```

for..in liste les clefs tandis que for..of liste les valeurs,

for..in et for..of n'itèrent pas sur les mêmes choses. Tandis que for..in boucle sur toutes les propriétés énumérables autre que Symbol, for..of parcourt l'ensemble des données contenues dans l'objet itérable.

4. Do … while :

Son fonctionnement est sensiblement le même que la boucle while sauf que son fonctionnement est inversé, la boucle s'exécute puis évalue la condition.

```
14
15  //boucle do while
16  // execute une premiere fois le code puis verifie la condition
17
18  compteur = 1;
19  do{
20      console.log(compteur);
21      compteur++;
22  }while(compteur <= 10);
23  console.log ("fin de boucle while");
24
```

la boucle s'exécute avant d'évaluer la condition. Ainsi, lors du premier tour, i est à 0, la boucle poursuit jusqu'à ce que i soit à 10, alors la condition est de nouveau évaluée, elle est toujours à true car i peut-être inférieur ou égal à 10, donc i est incrémenté, passe à 11 et une nouvelle exécution de boucle est validée. i valant maintenant 11, la prochaine condition stoppe la boucle.

5. Instructions de contrôle :

Il existe deux instructions spécifiques permettant de contrôler le comportement des boucles :

- break pour l'interrompre,

```
for (let i = 0; i < haystack.length; i++) {
    if (needle === haystack[i]) {
        index = i;
        break;
    }
}
```

- continue pour passer directement à l'itération suivante.

```
// on décide de trouver le nombre de fibonacci pour tous les nombres pairs entre x e
t y
// (x et y non inclus)
const x = 10;
const y = 30;

for (let i = x; i < y; i++) {
    // si c'est impair, on passe directement au tour suivant
    if (i % 2 !== 0) continue;
    console.log(fibonacci(i));
}
```

6. Label de bloc :

En plus des deux instructions que nous venons de voir, il est possible d'utiliser l'instruction de bloc avec un label.

break termine par défaut la boucle immédiate, mais si plusieurs boucles sont imbriquées, le break ne met pas fin à la boucle parente. En donnant un label à la boucle parente, c'est possible.

```
let indexParent = 0;
let indexChild = 0;
const needle = 5;
const haystack = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

outer: for (let i = 0; i < haystack.length; i++) {
    for (let k = 0; k < haystack[i].length; k++) {
        if (needle === haystack[i][k]) {
            indexParent = i;
            indexChild = k;
            break outer;
        }
    }
}
```

Sans le label, quand bien même la valeur serait trouvée dans le premier tableau, nous devrions itérer sur toutes les valeurs du parent. Le label nous permet ici d'indiquer à break que nous voulons stopper une boucle de plus haut niveau (en plus de la boucle courante).