The image shows the JavaScript logo, which consists of the letters 'JS' in a bold, black, sans-serif font. The logo is centered within a solid yellow square. This square is positioned on the left side of the slide, with the text 'Valeurs et références' to its right.

JS

Valeurs et références

Valeurs et références

Lorsqu'un programme est exécuté le système d'exploitation prépare deux zones mémoires :

- ◆ la stack ("pile")
- ◆ le heap ("tas")

Ces zones mémoire sont utilisées pour le stockage des variables.

- ◆ La stack est initialisée lors du chargement du programme et le système d'exploitation lui associe une **taille fixe** ;
- ◆ Le heap est une zone dite "dynamique" dont la taille, fixée initialement par le système d'exploitation, **peut évoluer au runtime**. On l'utilisera donc pour stocker nos variables lorsque l'on aura une quantité conséquente de données.

→ Attention à ne pas confondre la stack mémoire et la call stack qui sont deux entités différentes. La stack mémoire permet de stocker nos variables, la call stack est un mécanisme interne aux interpréteurs JavaScript permettant de stocker et de gérer l'ordre d'exécution de nos instructions.

Dans certains langages (comme en C ou en C++) on doit se soucier de la zone mémoire que l'on utilise.

Par défaut toutes les variables seront stockées sur la pile et on procédera à de l'**allocation dynamique** pour stocker une variable sur le tas.

Qui dit allocation dynamique, dit gestion de mémoire : il faut supprimer l'espace occupé par une variable lorsque l'on en a plus besoin. Certains langages mettent en place un **garbage collector** ("ramasse miettes") pour affranchir le développeur de cette tâche.

En JavaScript les moteurs d'interprétation suivent ces 3 règles très simples :

- Toutes les variables de types primitifs (number, string, boolean, undefined, null) sont stockées sur la stack ;
- Les objets et fonctions sont stockées sur le heap et sont référencées par des variables stockées sur la stack ;
- Il existe un garbage collector qui procède automatiquement aux désallocations lorsqu'elles sont nécessaires (ie. lorsque les variables du heap ne sont plus référencées par des variables de la stack).

Valeurs et références

- Lorsque l'on manipule des variables de types primitifs, on dit que l'on manipule des variables par valeur. Cela signifie que lorsque l'on change de scope, on travaille sur des "copies" des variables initiales (cas typique d'un appel de fonction).
- Lorsque l'on travaille avec des objets on dit que l'on travaille avec des références. Lorsque l'on change de scope, les références ne changent pas et on manipule in fine les mêmes structures de données.

Illustrons le passage de paramètres d'une fonction par valeur :

```
function incrementerEntier(entier) {  
  ++entier; //1  
}  
  
var unEntier = 0;  
incrementerEntier(unEntier);  
console.log(unEntier); //0
```

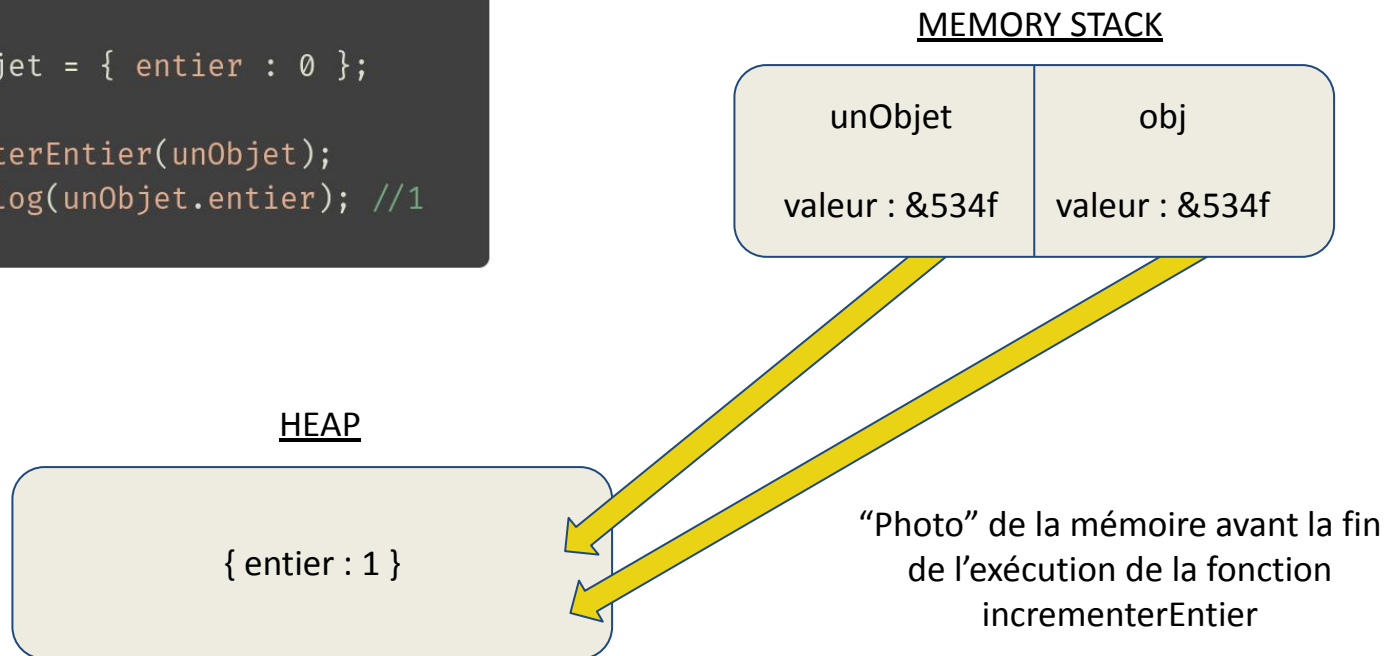
unEntier	entier
valeur : 0	valeur : 1

"Photo" de la stack mémoire avant
la fin de l'exécution de la fonction
incrementerEntier

Valeurs et références

Illustrons le passage de paramètres d'une fonction par référence :

```
function incrementerEntier(obj) {  
  ++obj.entier;  
}  
  
var unObjet = { entier : 0 };  
  
incrementerEntier(unObjet);  
console.log(unObjet.entier); //1
```



Valeurs et références

Il faut faire attention aux conséquences de la gestion mémoire des objets en JavaScript :

- ◆ La recherche d'un objet dans un tableau se fait naturellement par référence (utilisation de `indexOf` ou `includes` plutôt pour des types primitifs, `find` pour des objets) ;
- ◆ La copie d'objets n'est pas aisée car bien souvent ce sont les références qui sont copiées. On parle de copie de surface vs copie de profondeur.

```
let tabEtudiants = [
  { prenom : "Jean" },
  { prenom : "Marie" },
  { prenom : "Paul" }
];

console.log(tabEtudiants.includes({prenom : "Marie"})); //false

let camille = { prenom : "Camille" };

tabEtudiants.push(camille);

console.log(tabEtudiants.includes(camille)); //true
```

Illustration de la problématique de recherche avec des références

Valeurs et références



```
let camille = { prenom : "Camille" };

let camille2 = camille;

camille.dateNaissance = new Date(2020, 02, 04);

//Affiche bien la date de naissance
console.log(camille2.dateNaissance);
```

Les variables `camille` et `camille2` sont déclarées sur la stack et ont pour valeur une adresse d'un objet stocké sur le heap.



```
let camille = { prenom : "Camille" };

let camille2 = Object.assign({}, camille);

camille.dateNaissance = new Date(2020, 02, 04);

//Affiche undefined
console.log(camille2.dateNaissance);
```

Pour copier un objet et non sa référence, on peut utiliser `Object.assign`

```
let camille = {
  prenom : "Camille",
  amis : ["Pierre", "Paul", "Jacques"]
};

let camille2 = Object.assign({}, camille);


camille2.amis.push("Laura");

/* Camille et camille2 ont la même
   référence vers le tableau d'amis */
console.log(camille.amis);
```

Problème de la copie avec `Object.assign` : il ne s'agit pas d'une réelle copie en profondeur.

Si une propriété de l'objet copié fait référence à une variable sur le heap, c'est la référence qui est copiée.

C'est une des difficultés en JavaScript : le langage ne propose aucune méthode native permettant de réaliser une copie de profondeur.



```
let camille = {
  prenom : "Camille",
  amis : ["Pierre", "Paul", "Jacques"]
};

let camille2 = JSON.parse(JSON.stringify(camille));

camille2.amis.push("Laura");

//Il existe désormais 2 tableaux d'amis
console.log(camille.amis);
```

Une astuce souvent utilisée, dans le cas d'objets sérialisables, est de sérialiser puis de désérialiser un objet pour le cloner.



```
const _ = require("lodash");

let camille = {
  prenom : "Camille",
  amis : ["Pierre", "Paul", "Jacques"]
};

let camille2 = _.cloneDeep(camille);

camille2.amis.push("Laura");

//Pas de modification de camille
console.log(camille.amis);
```

On peut utiliser la librairie lodash pour réaliser de véritables copies en profondeur.

Pour installer lodash :
npm install lodash --save-dev