



# JavaScript

Programmation objet en JavaScript

# Qu'est ce-que le JavaScript ?

JavaScript a été créé en 1995 par Brendan Eich (Netscape Communications Corporation). Il a été utilisé à l'origine pour ajouter de l'interactivité aux pages Web.

Le JavaScript est un des langages de programmation les plus populaires, car il est maintenant utilisé dans plusieurs domaines.

- Web.
- Serveur: Via l'utilisation de Node.js
- Mobile: Via l'utilisation de Ionic ou de React Native.
- Desktop: Via l'utilisation de Electron.

# Un peu d'Histoire...

Avant JavaScript toutes les pages web étaient statiques. Une fois la page chargée, le HTML était figé. NetScape décide d'ajouter un langage script à son navigateur. Pour ce faire, ils choisissent de partir sur 2 options.

- Collaboration avec Sun Microsystems pour embarquer le langage JAVA dans leurs navigateurs.
- Création d'un langage de script pour son navigateur.

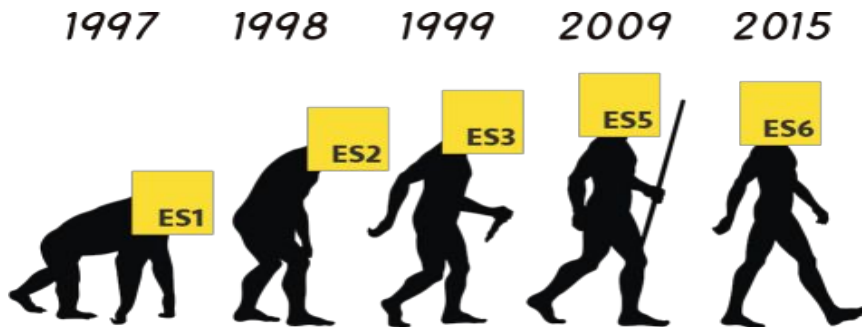
Rapidement la première option est abandonnée. La première version du langage est un langage minimaliste incorporé à NetScape 2.0 Beta. Son nom original était LiveScript avant d'être renommé en Javascript en raison de la prédominance du langage Java lors de son lancement (par soucis de marketing).

Microsoft décide de s'inspirer de NetScape pour leurs navigateurs Internet Explorer et utilise un langage de script similaire au JavaScript appelé JScript.

# Un peu d'Histoire...

Pour éviter d'avoir des langages de scripts différents en fonction des navigateurs, NetScape décide de normaliser le langage avec une spécification standard appelée ECMAScript.

Le langage Javascript a beaucoup évolué depuis sa normalisation, donnant par exemple l'ES1, l'ES2, etc... Pour en venir en 2015 à l'ES6 (ECMAScript 6). A partir de cette version, de nombreuses fonctionnalités supplémentaires ont fait leur apparition. Pour cette raison, il est désormais convenu d'appeler le Javascript post-ES6 le Javascript « moderne ».



# JavaScript

Le JavaScript est un langage de script de haut niveau (high level). Il peut être caractérisé de la façon suivante

- Dynamiquement typé (le type des variables est déterminé à l'exécution).
- Faiblement typé (utilisation de coercition de type).
- Un seul fil d'exécution (single-threaded).
- Utilisation d'un ramasse-miettes (Garbage Collector).
- Interprété ou compilé à la volée (Just-in-time).
- Orienté objet (orienté prototype).
- Multi-paradigmes (objet, impératif, fonctionnel).

# Où s'exécute le JavaScript ?

## Exécution côté client (navigateur Web)

A l'origine le JavaScript ne s'exécutait que dans les navigateurs internet. Chaque navigateur possède un moteur JavaScript qui est chargé d'exécuter le code JavaScript.

- Chrome: V8
- FireFox: SpiderMonkey
- Safari: WebKit



# Où s'exécute le JavaScript ?

## Exécution côté serveur (au sein de Node.js)

En 2009 Ryan Dahl a eu l'idée de prendre le programme Open Source du moteur V8 et de l'intégrer dans un programme C++. Ce programme s'appelle Node et il permet d'exécuter le JavaScript en dehors du navigateur.

Le JavaScript peut donc s'exécuter côté client (dans le navigateur) ou côté serveur (dans Node).



# Mettre en place l'environnement

Pour la poursuite de ce cours, nous allons installer les programmes suivants

- L'éditeur de code (IDE): Visual Studio Code.  
<https://code.visualstudio.com/download>
- Node.js  
<https://nodejs.org/en/download/>
- Le navigateur Google Chrome

Pour vérifier que Node est bien installé sur vos postes vous pouvez taper la commande

`node --version` ou `node -v`

La version de Node devrait s'afficher. Si elle ne s'affiche pas, cela veut dire que Node est mal installé.



# Test d'exécution Node.js

Le but de ce test est de vérifier que tout est convenablement installé et que vous pouvez exécuter des programmes JavaScript avec Node.

Voici les étapes à suivre.

1. Créer un répertoire qui contiendra l'application à exécuter.
2. Créer un fichier index.js qui contiendra le code suivant  
`console.log("Hello World");`
3. Exécuter le programme à l'aide du terminal d'exécution  
`node index.js`

# Test d'exécution côté client

Ici nous allons exécuter le JavaScript côté navigateur. Pour cela vous allez suivre les étapes suivantes

- Créer un fichier index.html qui va référencer le JavaScript précédemment créé.  
Pour cela ajouter le tag suivant avant de fermer la balise <head>.  
`<script defer src="./index.js"></script>`  
`</head>`
- Vous pouvez écrire du texte dans le tag <body>  
`<body><h1>Test JavaScript</h1></body>`
- Vous pouvez Installer l'extension Visual Studio Code "Live Server" et ouvrir votre fichier index.html à l'aide de "Live Server" ou ouvrir le fichier index.html directement avec votre navigateur.
- Une fois la page affichée, inspectez la page pour vérifier que "Hello World" est bien écrit dans la console (Ctrl + Shift + J sur Chrome, Ctrl + Shift + I sur FireFox).

# Les variables (le mot clé var)

Une variable est tout simplement un conteneur qui permet de stocker une valeur.  
Les noms de variable doivent suivre la convention de nommage camelCase.

## Utilisation du mot clé 'var'

Avant l'ES6 la seule façon disponible pour créer une variable était d'utiliser le mot clé 'var'

```
var maVariable = 'Bonjour';
```

La portée d'une variable déclarée avec 'var' est le contexte d'exécution courant, c'est-à-dire : la fonction qui contient la déclaration ou le contexte global si la variable est déclarée en dehors de toute fonction.

## Le hoisting

Les déclarations de variables déclarées par 'var' sont traitées avant que le code soit exécuté, quel que soit leur emplacement dans le code. Ainsi, déclarer une variable n'importe où dans le code équivaut à la déclarer au début de son contexte d'exécution (fonction courante ou fichier courant).

# Les variables (le mot clé var)

Déclarer une variable n'importe où dans le code équivaut à la déclarer au début de son contexte d'exécution (fonction courante ou fichier courant).

```
console.log(maVariable); //undefined  
var maVariable = 'Bonjour';
```

Est l'équivalent à

```
var maVariable;  
console.log(maVariable); //undefined  
maVariable = 'Bonjour';
```

# Les variables (variables globales)

Si on affecte une valeur à une variable qui n'a pas été déclarée (le mot-clé var n'a pas été utilisé), cela devient une variable globale.

Voici un exemple de déclaration de variable globale.

```
function x(){  
    maVariable = 'Bonjour';  
}  
x();  
console.log(maVariable); //Bonjour
```

# Les constantes et les variables (const et let)

Depuis l'ES6 les déclarations de variables peuvent se faire à partir des mots clé 'const' et 'let'.

Le mot clé 'const' est utilisé pour déclarer des constantes.

Le mot clé 'let' est utilisé pour déclarer des variables.

Les variables déclarées à l'aide de 'const' et 'let' ne sont pas hoistées et la portée de ces déclarations est limitée au bloc de déclaration.

```
{  
  const maConstante = 3.14;  
  let maVariable = 0;  
}
```

# Le typage dynamique

JavaScript possède un typage dynamique. C'est-à-dire que l'interpréteur attribue aux variables un type lors de l'exécution.

```
let maVariable = 'Bonjour';  
console.log(typeof maVariable); //string  
maVariable = 42;  
console.log(typeof maVariable); //number
```

# Comparaison de valeur

Lorsqu'on compare 2 valeurs de différents types, JavaScript effectue automatiquement une coercition de type (transformation d'un type à une autre)

```
console.log("13" > 12); //true
```

Pour les opérateurs d'égalité et d'inégalité, le JavaScript possède 2 types d'opérateurs. Les opérateurs standards qui utilisent la coercition de type.

```
console.log("42" == 42); //true  
console.log("1,2" != [1,2]); //false
```

L'égalité et l'inégalité stricte. Ces opérateurs considèrent toujours des opérandes de types différents comme étant différents.

```
console.log("42" === 42); //false  
console.log("1,2" !== [1,2]); //true
```



# Coercition de type

La coercition de type (type coercion en anglais) est la conversion automatique ou implicite de valeurs d'un type de données à un autre.

```
const valeur1 = '5';  
const valeur2 = 9;  
//la 2ème valeur est transformée en string  
let somme = valeur1 + valeur2;  
console.log(somme); //59  
console.log(typeof somme); //string
```

# Coercition de type

## Attention! la coercition de type peut produire des effets inattendus

La coercition peut se produire lors de l'utilisation d'une opérande.

```
const log = console.log;
log(0 == "0"); //true
log(0 == []);  //true
log("0" == []); //false

log(9+"1"); //91
log("91"-"1"); //90
log(9+"1"-"1"); //90

log(true === 1); //false
log(true+true === 2); //true
```

# Coercition de type

Pour ne pas avoir de comportements inconsistants, les coercitions de type doivent être évitées.

Utilisez les égalités strictes (===) et les inégalités strictes (!===).

```
console.log(x === 10);
```

Utilisez des conversions explicites pour comparer 2 valeurs

```
console.log(Number(x) > 10);
```

Utilisez des conversions explicites pour utiliser des opérandes mathématiques

```
console.log(Number(x) + 5);
```

Concaténer les chaînes de caractères avec des template strings.

```
console.log(`Concaténation de x et y: ${x}${y}`);
```

# Les valeurs truthy et falsy

Dans un contexte booléen, toutes les valeurs sont truthy sauf les valeurs suivantes qui sont falsy

- 0 et NaN (pour le type number).
- 0n (pour le type bigint).
- "" (pour le type string).
- False (pour le type boolean).
- null.
- undefined

Toutes les valeurs de type objet sont truthy.

Pour vérifier qu'un objet n'est ni null ni undefined, on peut utiliser la condition suivante

```
if (obj) //obj !== null && obj !== undefined
  console.log("Mon objet n'est ni null ni undefined");
```

# Les valeurs truthy et falsy

Les valeurs truthy et falsy sont utilisées dans des expressions booléennes.

```
console.log("" ? true : false); //false
console.log(!!"0"); //true
console.log(!!{}); //true
console.log(!!0); //false
console.log(!!null); //false
console.log(!!undefined); //false
console.log(!![]); //true
```

# truthy et falsy (avec l'opérateur &&)

L'opérateur && peut être utilisé avec des valeurs truthy ou falsy pour retourner la première valeur falsy ou la dernière valeur si toutes les valeurs sont truthy.

```
console.log(true && "chien"); //chien  
console.log(false && "chien"); //false
```

```
console.log([] && "chien"); //chien  
console.log(null && "chien"); //null
```

# truthy et falsy (avec l'opérateur ||)

L'opérateur || peut être utilisé avec des valeurs truthy ou falsy pour retourner la première valeur truthy ou la dernière valeur si toutes les valeurs sont falsy.

```
console.log(true || "chat"); //true
```

```
console.log(false || "chat"); //chat
```

```
console.log([] || "chat"); //[]
```

```
console.log(null || "chat"); //chat
```

# Les types de données

En Javascript, une valeur est soit un objet, soit une valeur primitive.

Voici les différents types de primitives en JavaScript.

**Number** : Ce sont des nombres à valeur flottante.

**String** : Les Chaînes de Caractères permettent de stocker des séquences de lettres, de mots ou de lignes de texte.

**Boolean** : Ces valeurs sont de deux types : Vrai ou Faux.

**Undefined** : Type donné à une variable ne possédant pas encore de valeur.

**Null** : Cette valeur est affectée pour signifier une absence de valeur.

**Symbol** (ES2015) : Il s'agit d'une valeur dont l'unicité est garantie.

**BigInt** (ES2020) : Concerne les entiers trop grands pour être correctement représentés par les primitives Number.



# Les types de données objet

Tous les types non primitifs en JavaScript sont des objets.

Pour connaître le type d'une variable, on peut utiliser le mot clé 'typeof'.

```
console.log(typeof(5)); //number  
console.log(typeof("5")); //string  
console.log(typeof(true)); //boolean  
console.log(typeof({})); //objet
```

**Attention! Ne pas utiliser la fonction typeof sur une valeur null**

```
console.log(typeof(null)); //object
```

Préférez la syntaxe suivante pour vérifier qu'une variable 'obj' est de type objet

```
console.log(obj instanceof Object); //false
```

# Les types de données objet

Tous les types suivants sont des objets

```
console.log(typeof(new Boolean(true))); //object  
console.log(typeof(new String(5))); //object  
console.log(typeof(new Number(5))); //object
```

En effet le JavaScript possède des objets wrapper pour les types suivants: string, number et boolean. Cela permet d'appeler des méthodes définies au niveau de l'objet.

```
console.log("12345".substring(0,2)); //12  
console.log(15.123.toFixed(1)); //15.1  
console.log(true.toString()); //true
```

# Les types de données objet

Pour savoir si une variable est de type objet, on peut utiliser le mot clé 'instanceof'.

```
console.log(5 instanceof Object); //false
console.log({} instanceof Object); //true
console.log(function(){} instanceof Object); //true
console.log([] instanceof Object); //true
console.log(true instanceof Object); //false
console.log(new String(5) instanceof Object); //true
console.log(null instanceof Object); //false
```

# Les conditions (bloc if else)

Exemple d'utilisation de blocs if else.

Le code suivant affiche "Tiède".

```
const temperature = 25;  
if (temperature > 30)  
  console.log("Chaud");  
else if (temperature >= 20)  
  console.log("Tiède");  
else  
  console.log("Froid");
```

# Les conditions (l'opérateur conditionnel)

L'exemple précédent peut être re-écrit à l'aide de l'opérateur ternaire conditionnel.

```
const temperature = 25;  
const text = (temperature > 30) ?  
  "Chaud"  
  : (temperature >= 20) ?  
    "Tiède"  
    : "Froid";  
console.log(text);
```

# Les conditions (bloc switch case)

Voici un exemple de Switch case.

Affiche le jour de semaine si on est un week-end sinon affiche "Jour de semaine"

```
switch (new Date().getDay()) {  
  case 6:  
    console.log("Samedi");  
    break;  
  case 0:  
    console.log("Dimanche");  
    break;  
  default:  
    console.log("Jour de semaine");  
}
```

# Boucles et itérations (while)

Le code suivant affiche le message "Affiche 5 fois" 5 fois à l'écran

```
let index=5;  
while (index-- > 0)  
{  
  console.log("Affiche 5 fois");  
}
```

# Boucles et itérations (do while)

Ce code effectue la même opération que précédemment

```
let index=5;  
do  
{  
  console.log("Affiche 5 fois");  
}  
while (--index > 0);
```



# Boucles et itérations (for)

Même opération que précédemment

```
for (let i=0; i<5; ++i)
{
  console.log("Affiche 5 fois");
}
```

# Boucles et itérations (for of)

L'instruction for...of crée une boucle qui fonctionne avec les objets itérables.

```
const tableau = [2,3,5,7,11];  
for (let element of tableau)  
{  
  console.log(element);  
}
```

# Boucles et itérations (for in)

L'instruction for...in permet d'itérer sur l'ensemble des propriétés énumérables d'un objet.  
Le programme suivant affiche toutes les propriétés de l'objet 'obj'.

```
const obj = { one: 1, two: 2 };  
for (let element in obj)  
  console.log(`${element}: ${obj[element]}`);  
//one: 1  
//two: 2
```

# Les chaînes de caractères

Déclaration des longues chaînes de caractères

```
//déclaration d'une longue chaîne de caractère sur une seule ligne
```

```
const longTexte = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, \
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam";
console.log(longTexte);
```

```
//déclaration d'une chaîne de caractère sur une plusieurs lignes
```

```
const longTexte2 = `Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam`;
console.log(longTexte2);
```

# Les chaînes de caractères

Les chaînes de caractères sont immuables en JavaScript.

On ne peut pas modifier le contenu d'une variable de type string.

```
let text = 'Bonjour';  
console.log(text[0]); // 'B'  
text[0] = 'Z';  
console.log(text[0]); // 'B'  
console.log(text);    // 'Bonjour'
```

# Les chaînes de caractères

## Concaténation de chaînes de caractères

```
const hello = "Hello";  
const world = 'world';  
console.log(hello + ' ' + world);    //Hello world  
console.log(hello.concat(' ',world)); //Hello world
```

## Interpolation de chaînes de caractères

```
const one = 1;  
const two = 2;  
console.log(`${one} + ${two} = ${one+two}`); //1 + 2 = 3
```

# Les chaînes de caractères

Autres opérations sur les chaînes de caractères

```
const sentence = "The quick brown fox jumps over the lazy dog.";
//vérifie si la chaîne contient la sous-chaîne
console.log(sentence.includes('fox')); //true
//retourne le premier index de 'q'
console.log(sentence.indexOf('q')); //4
//retourne le dernier index de 'd'
console.log(sentence.lastIndexOf('d')); //40
//retourne le 3ème caractère
console.log(sentence[2]); //e
//retourne une sous-chaîne de caractères
console.log(sentence.substring(4,9)); //quick
//retourne la taille de la chaîne de caractère
console.log(sentence.length); //44
```

# Les fonctions (déclaration)

Une fonction sert à éviter la répétition du code. On peut utiliser une fonction pour réaliser toute une série d'instructions au lieu de devoir réécrire plusieurs fois nos instructions.

Voici la déclaration d'une fonction qui ne prend aucun paramètre et qui ne retourne rien.

```
function maFonction(){} 
```

Voici la déclaration d'une fonction qui prend 2 paramètres et qui retourne la somme de ces 2 paramètres

```
function additionner(a,b){ return a+b; }  
console.log(additionner(2,3)); //5  
console.log(additionner('Bon','jour')); //Bonjour
```



# Les fonctions (déclaration)

Le hoisting fonctionne aussi avec les fonctions, ce qui vous permet d'organiser votre code de la manière dont vous le souhaitez.

Le code suivant fonctionne également

```
console.log(additionner(2,3)); //5  
console.log(additionner('Bon','jour')); //Bonjour  
function additionner(a,b){ return a+b; }
```

# Les fonctions (fonctions expressions)

Il existe une syntaxe pour créer une fonction appelée Expression de Fonction.

Cela nous permet de créer une nouvelle fonction au milieu de n'importe quelle expression

```
const additionner = function(a,b){ return a+b; }  
console.log(additionner(2,3)); //5  
console.log(additionner('Bon','jour')); //Bonjour
```

**Attention il n'y a pas de hoisting pour les fonctions expressions.** La fonction doit être déclarée avant d'être utilisée.

# Les fonctions (fonctions fléchées)

Une expression de fonction fléchée (arrow function en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction.

```
const additionner = (a,b) => a+b;  
console.log(additionner(2,3)); //5  
console.log(additionner('Bon','jour')); //Bonjour
```

**Attention il n'y a pas de hoisting pour les fonctions fléchées.** La fonction doit être déclarée avant d'être utilisée.

# Les fonctions (les surcharges)

Il n'est pas possible de surcharger une fonction en JavaScript. Une fonction ne peut avoir qu'une seule signature.

Si plusieurs fonctions sont définies avec le même nom, seule la dernière fonction est prise en considération.

```
function test() {  
  console.log("Fonction sans paramètres");  
}  
function test(a, b) {  
  console.log(`Fonction avec paramètres ${a}, ${b}`);  
}  
test(); //Fonction avec paramètres undefined, undefined
```

# Les fonctions (les arguments)

On peut manipuler les arguments d'une fonction en utilisant le paramètre 'arguments'.

```
function lireArguments()  
{  
  console.log(`le nombre d'arguments est ${arguments.length}`);  
  //retourne un objet contenant les paramètres de la fonction  
  console.log(arguments);  
  if (arguments.length > 0)  
    //retourne la valeur du premier argument  
    console.log(`Le premier argument est ${arguments[0]}`);  
}
```

# Les fonctions (les propriétés)

Toute fonction f possède les propriétés suivantes

**name:** renvoie le nom de la fonction

**length:** renvoie le nombre de paramètres attendus par la fonction.

**prototype:** cette propriété est utilisée quand la fonction est utilisée comme constructeur (avec l'opérateur new). Elle devient la propriété de l'objet instancié.

```
function maFonction(one){}  
console.log(maFonction.name); //maFonction  
console.log(maFonction.length); //1
```

# Les fonctions (fonctions récursives)

Les fonctions peuvent s'appeler elles-mêmes.

```
function factoriel(n)
{
  if (n === 0)
    return 1;
  return n * factoriel(n-1);
}

console.log(factoriel(5)); //120
```

# Les fonctions

Les fonctions peuvent prendre en paramètre d'autres fonctions

```
function executer(num1, num2, operation, afficher)
{
  const resultat = operation(num1, num2);
  afficher(resultat);
}
```

```
executer(3, 7, (a,b) => a+b, console.log); //10
```



# Les fonctions (le currying)

Les fonctions peuvent renvoyer d'autres fonctions

```
function curry(f) {  
  return function(a) {  
    return function(b) {  
      return f(a, b);  
    };  
  };  
}
```

```
let curriedSum = curry((a,b) => a+b);  
console.log(curriedSum(1)(2)); // 3
```

# Les tableaux

Les tableaux servent en programmation à stocker plusieurs valeurs en une seule variable.

Création de tableaux vides

```
const tableau = [];  
const tableau2 = new Array();  
const tableau3 = Array();
```

Création de tableaux avec valeurs initialisées

```
const tableau = [1,2,3];  
const tableau2 = Array(1,2,3);  
//crée un tableau de 5 éléments initialisé avec des 0  
const tableau3 = Array(5).fill(0);  
console.log(tableau);    //[1,2,3]  
console.log(tableau2);   //[1,2,3]  
console.log(tableau3);   //[0,0,0,0,0]
```

# Les tableaux

Pour parcourir le tableau, il faut passer par une notation entre crochets et se servir de l'index de l'élément que l'on cherche. Les tableaux commencent avec un index de valeur 0.

## Obtenir la taille d'un tableau

```
console.log(monTableau.length); //3
```

## Récupérer la valeur d'un élément de tableau

```
const monTableau = ['A', 'B', 'C'];  
console.log(monTableau[1]); //B  
console.log(monTableau.at(1)); //B  
console.log(monTableau.at(-1)); //C  
console.log(monTableau[monTableau.length-1]); //C
```

# Les tableaux

Ajouter des éléments dans un tableau

```
const lettres = ['B', 'C'];
```

Ajouter un élément à la fin d'un tableau

```
console.log(lettres.push('D')); //3  
console.log(lettres); //[ 'B', 'C', 'D' ]
```

Ajouter un élément au début d'un tableau

```
console.log(lettres.unshift('A')); //4  
console.log(lettres); //[ 'A', 'B', 'C', 'D' ]
```

# Les tableaux

Supprimer des éléments d'un tableau

```
const lettres = ['A', 'B', 'C', 'D'];
```

Retirer le dernier élément d'un tableau (index length-1)

```
console.log(lettres.pop()); // 'D'  
console.log(lettres); // [ 'A', 'B', 'C' ]
```

Retirer le premier élément d'un tableau (index 0)

```
console.log(lettres.shift()); // 'A'  
console.log(lettres); // [ 'B', 'C' ]
```

# Les tableaux

Vérifier le contenu de tableaux

```
const pets = ['cat', 'dog', 'bat', 'dog'];  
console.log(pets.includes('cat'));    //true  
console.log(pets.indexOf('dog'));    //1  
console.log(pets.lastIndexOf('dog')); //3
```

Inverser un tableau

```
console.log(pets.reverse()); //['dog', 'bat', 'dog', 'cat' ]
```

Attention le tableau appelant la fonction 'reverse' est modifié

```
console.log(pets);           //[ 'dog', 'bat', 'dog', 'cat' ]
```

# Les tableaux

La méthode `sort` permet d'ordonner un tableau

```
const tableau = ['E', 'C', 'D', 'B', 'A'];  
tableau.sort()  
console.log(tableau); // [ 'A', 'B', 'C', 'D', 'E' ]
```

Par défaut la méthode `sort` ordonne les tableaux par ordre alphabétique

```
const tableau = [10, 7, 18, 2, 65];  
tableau.sort()  
console.log(tableau); // [ 10, 18, 2, 65, 7 ]
```

Pour palier à ce problème, on peut passer une fonction spécifique pour ordonner notre tableau.

# Les tableaux (map filter reduce)

## La fonction map

La fonction 'map' crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau.

```
const tab = [2,3,5];  
const resultat = tab.map(element => 2*element);  
console.log(resultat); // [ 4, 6, 10 ]  
console.log(tab); // [ 2, 3, 5 ]
```



# Les tableaux (map filter reduce)

## Le fonction filter

La méthode 'filter' retourne un nouveau tableau contenant tous les éléments du tableau d'origine qui remplissent une condition déterminée.

```
const tab = [1,2,3,4,5,6];  
const resultat = tab.filter(element => element % 2 === 0);  
console.log(resultat); // [ 2, 4, 6 ]
```

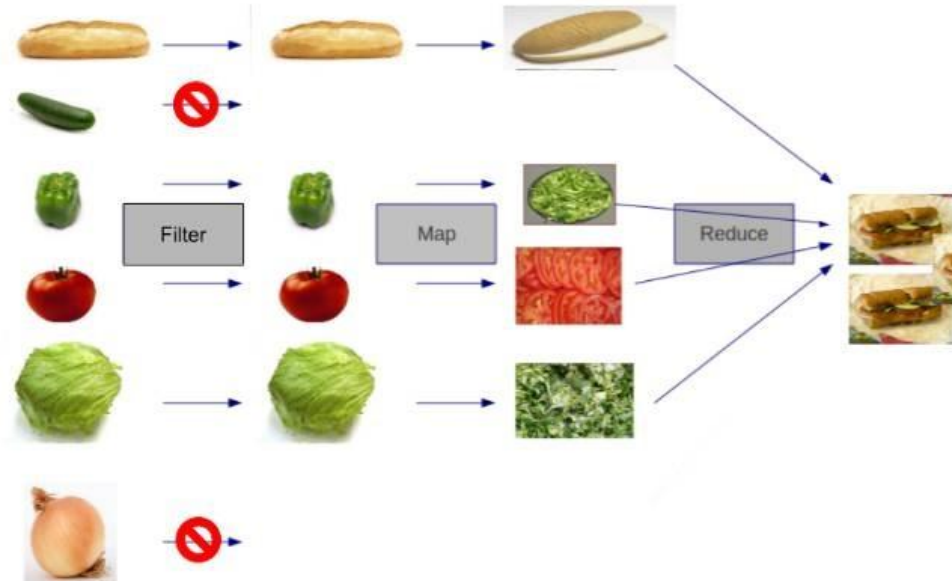
# Les tableaux (map filter reduce)

## La fonction reduce

La fonction 'reduce' applique une fonction qui traite chaque valeur d'une liste afin de la réduire à une seule valeur.

```
const tab = [1,2,3,4,5,6];  
const resultat = tab.reduce((accumulateur, element) => accumulateur + element);  
console.log(resultat); // 21
```

# Les tableaux (map filter reduce)



# Les tableaux (autres fonctions)

Voici une liste de fonction pouvant être utiles lors de la manipulation de tableaux

```
const tab = [6,2,3,1,4,5];
```

La fonction 'some' retourne vrai si au moins un élément satisfait la fonction (le prédicat).

```
console.log(tab.some(element => element > 5)); //true
```

La fonction 'every' retourne vrai si tous les éléments satisfont la fonction (le prédicat).

```
console.log(tab.every(element => element > 5)); //false
```

La fonction 'sort' ordonne le tableau (**Attention la fonction 'sort' modifie le tableau appelant**)

```
console.log(tab.sort((a,b) => a-b)); //[ 1, 2, 3, 4, 5, 6 ]
```

```
tab.forEach(element => console.log(element)); //affiche tous les éléments du tableau
```

# Les tableaux à plusieurs dimensions

Voici comment créer et manipuler des tableaux à plusieurs dimensions

```
const tab2d = [  
  [0,1,2],  
  [1,2,'?'],  
  [2,3,4]  
];  
  
console.log(tab2d[0]); //[ 0, 1, 2 ]  
console.log(tab2d[1][2]); //?  
tab2d[1][2] = 3;  
console.table(tab2d); //affiche le nouveau tableau
```

(index)	0	1	2
0	0	1	2
1	1	2	3
2	2	3	4

# Les objets

Un objet est une entité à part entière qui possède des propriétés et un type.

Voici comment créer un objet vide en JavaScript

```
const obj1 = {};  
const obj2 = new Object();
```

Voici comment créer un objet avec plusieurs éléments

```
const obj = { "one": 1, "two": 2 };
```

Pour accéder à une propriété d'un objet

```
const obj = { "one": 1, "two": 2 };  
console.log(obj.one);    //1  
console.log(obj["two"]); //2
```

# Les objets

Il est possible d'ajouter des propriétés à un objet existant

```
obj.three = 3;  
console.log(obj); //{ one: 1, two: 2, three: 3 }
```

Il est possible d'ajouter une propriété contenant une fonction

```
obj.afficher = function(){console.log(this);}   
obj.afficher(); //{ one: 1, two: 2, three: 3, afficher: [Function (anonymous)] }
```

Il est possible de supprimer une propriété d'un objet

```
delete obj.afficher;  
console.log(obj); //{ one: 1, two: 2, three: 3 }
```

# Les objets

Lorsque l'on compare 2 objets en JavaScripts, on compare leurs références pas leurs valeurs

```
const obj1 = { prix: 20 };  
const obj2 = { prix: 20 };  
const obj3 = obj1;  
console.log(obj1 === obj2); //false  
console.log(obj1 === obj3); //true
```



# Les objets

Il y a 2 façons de définir des fonctions qui retournent des objets

Utilisation d'une fonction standard pour créer un objet

```
function creerObjet(un, deux) {  
  return { "one": un, "two": deux };  
}  
const obj = creerObjet(1,2);  
console.log(obj);    //{ one: 1, two: 2 }
```

Dans ce cas, la fonction 'creerObjet' crée un objet avec 2 propriétés.

# Les objets (les constructeurs)

Utilisation d'une fonction constructrice pour créer un objet

```
function Personne(nom, age) {  
  this.nom = nom;  
  this.age = age;  
}  
const personne = new Personne("Tom", 35);  
console.log(personne); //Personne { nom: 'Tom', age: 35 }
```

**Attention le mot clé 'new' ne doit pas être omis.**

```
const personne = Personne("Tom", 35);  
console.log(personne); //undefined
```

Il faut se mettre en mode strict pour détecter ce genre d'erreur ('use strict' au début du fichier JavaScript).

# Les objets (les constructeurs)

Les fonctions constructrices (constructor functions) permettent de créer un type d'objet particulier.

- Par convention on utilise le PascalCase pour nommer les fonctions constructrices (première lettre en majuscule).
- Pour appeler la fonction constructrice on utilise le mot clé 'new'.
- Le mot clé 'this' fait référence à l'instance de l'objet en cours de création.

# Les objets (les prototypes)

Les prototypes sont un mécanisme au sein de JavaScript qui permettent aux objets d'hériter des propriétés d'autres objets.

Tous les objets JavaScript héritent du prototype d'objet.

Par exemple, l'objet créé précédemment hérite du prototype de Object.

```
function creerObjet(un, deux) { return { "one": un, "two": deux } }  
const obj = creerObjet(1,2);  
console.log(obj);    //{ one: 1, two: 2 }  
console.log(obj instanceof Object); //true
```

Cela permet à mon objet d'utiliser des fonctions définies dans le prototype d'objet

```
console.log(obj.toString()); // [object Object]
```

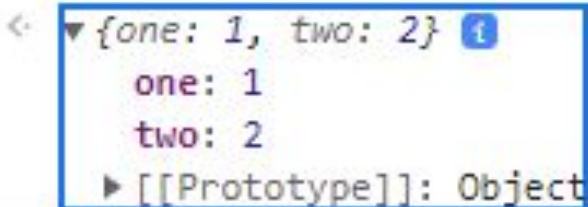
# Les objets (les prototypes)

Il est possible d'explorer les propriétés d'un objet à l'aide des outils de développement de votre navigateur (Ctrl + Shift + J pour Chrome, Ctrl + Shift + I pour FireFox).

Nous pouvons voir que l'objet créé à l'aide de la fonction précédente hérite du prototype de l'objet.

```
const obj = creerObjet(1,2);
```

```
> obj;
```



```
< {one: 1, two: 2} ⓘ  
  one: 1  
  two: 2  
  ► [[Prototype]]: Object
```

# Les objets (les prototypes)

L'objet créé à l'aide de la fonction constructrice est plus complexe car il crée un prototype pour son type.

```
function Personne(nom, age) { this.nom = nom; this.age = age; }  
const personne = new Personne("Tom", 35);
```

Le constructeur défini au niveau du prototype fait référence à la fonction qui a servi à créer l'objet.

```
> personne;  
◀ ▼ Personne {nom: 'Tom', age: 35} ⓘ  
  age: 35  
  nom: "Tom"  
  ▼ [[Prototype]]: Object  
    ► constructor: f Personne(nom, age)  
    ► [[Prototype]]: Object
```

# Les objets (les prototypes)

Lors de la création d'un objet à l'aide d'une fonction constructrice plusieurs opérations sont effectuées

```
const personne = new Personne("Tom", 35);
```

Est l'équivalent au code suivant

```
const personne = {};  
Object.setPrototypeOf(personne, Personne.prototype);  
//personne.constructor("Tom", 35);  
Personne.call(personne, "Tom", 35);
```

Il est intéressant de noter que

```
console.log(Object.getPrototypeOf(personne) === Personne.prototype); //true
```

# Les objets (les fonctions)

Il est intéressant de noter qu'une fonction n'est qu'un type d'objet particulier.

```
const additionner = function(a,b){ return a+b; }
```

Est équivalent à

```
const additionner = new Function('a', 'b', 'return a+b;');
```

Comme toute fonction nous avons les propriétés suivantes

```
console.log(additionner.constructor === Function); //true  
console.log(additionner.prototype.constructor === additionner); //true
```



# Les objets (les fonctions)

Il est important de noter que pour une fonction `f`

`f.prototype` n'est pas le prototype de l'objet `f` mais une propriété qui sert aux fonctions constructrices lors de l'appel de l'opérateur 'new' pour affecter le prototype de l'objet créé.

Pour une fonction `f`

**`f.prototype`  $\neq$  `Object.getPrototypeOf(f)`**

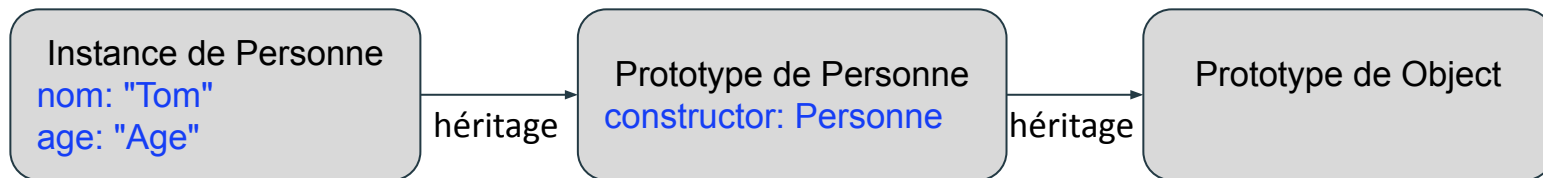
```
function f(){}  
console.log(f.prototype !== Object.getPrototypeOf(f));    //true  
console.log(f.prototype === Object.getPrototypeOf(new f)); //true
```

# Les objets (héritage et prototype)

JavaScript n'utilise qu'une seule structure : les objets.

Chaque objet possède une propriété privée qui contient un lien vers un autre objet appelé le prototype. Ce prototype possède également son prototype et ainsi de suite.

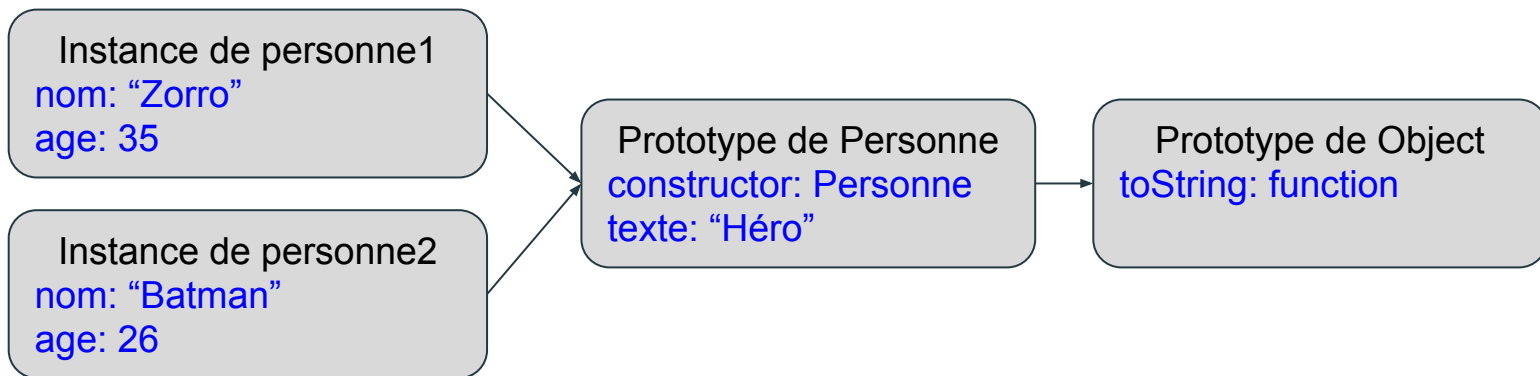
```
function Personne(nom, age) { this.nom = nom; this.age = age; }  
const person1 = new Personne("Tom", 35);
```



# Les objets (héritage et prototype)

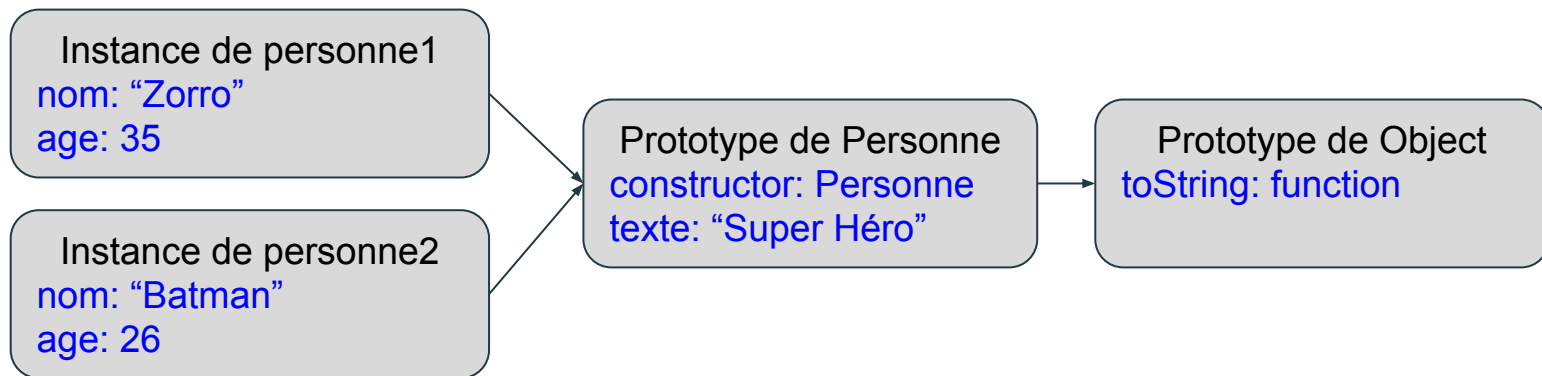
Toutes les instances d'un objet font référence au même prototype

```
function Personne(nom, age) { this.nom = nom; this.age = age; }  
Personne.prototype.texte = "Héro";  
const personne1 = new Personne("Zorro", 35);  
const personne2 = new Personne("Batman", 26);
```



# Les objets (héritage et prototype)

```
console.log(personne1.texte); //Héro
console.log(personne2.texte); //Héro
(Object.getPrototypeOf(personne1)).texte = "Super Héro";
console.log(personne2.texte); //Super Héro
console.log(personne2.toString()); //[object Object]
```



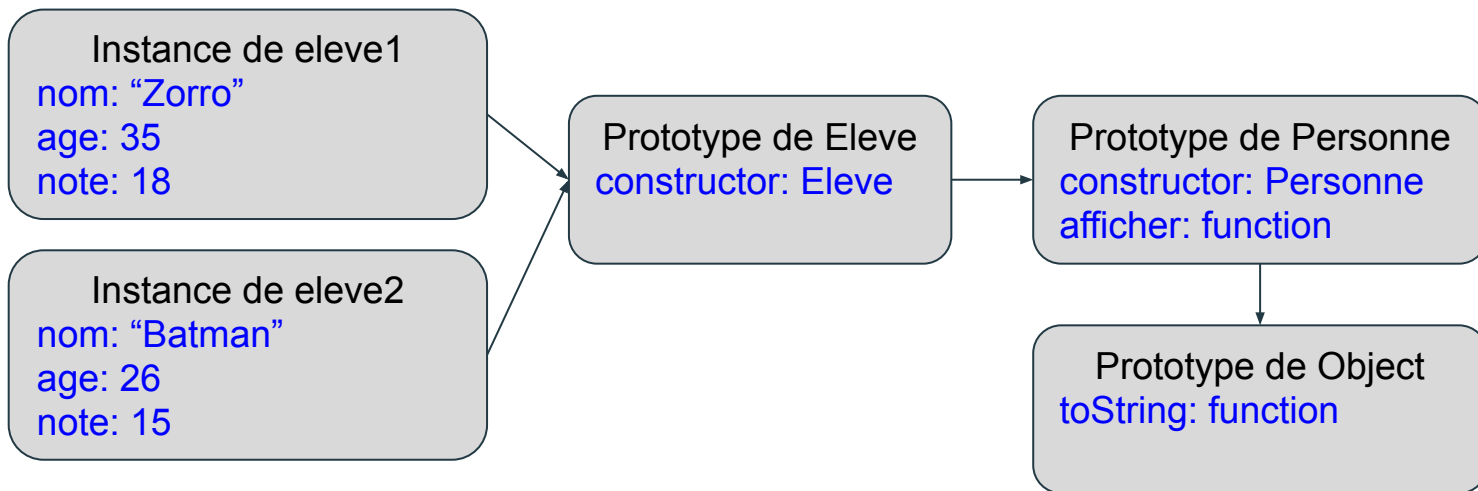
# Les objets (héritage et prototype)

Mise en place de l'héritage de prototype

```
function Personne(nom, age){ this.nom = nom; this.age = age; }  
function Eleve(nom, age, note){  
  this.note = note;  
  Personne.call(this, nom, age); //initialisation du constructeur parent  
}  
//définition de l'héritage  
Eleve.prototype = Object.create(Personne.prototype);  
Eleve.prototype.constructor = Eleve;  
//définition d'une fonction au niveau du prototype de Personne  
Personne.prototype.afficher = function() { console.log(this); }  
//création des instances  
const eleve1 = new Eleve("Zorro", 35, 18);  
const eleve2 = new Eleve("Batman", 26, 15);
```

# Les objets (héritage et prototype)

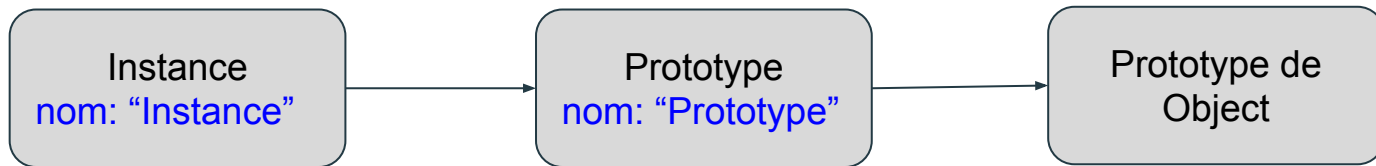
```
eleve1.afficher(); //Eleve { note: 18, nom: 'Zorro', age: 35 }  
eleve2.afficher(); //Eleve { note: 15, nom: 'Batman', age: 26 }
```



# Les objets (shadowing)

Attention une propriété peut en cacher une autre!

```
const obj = Object.create({ nom: "Prototype"});  
console.log(obj.nom); // Prototype;  
obj.nom = "Instance";  
console.log(obj.nom); // Instance;  
console.log(Object.getPrototypeOf(obj).nom); // Prototype;
```



# Les objets (héritage et prototype)

Le différence entre les langages à base de classes (comme JAVA et C#) et l'héritage de prototype en JavaScript.

- Dans les langages à base de classes, les classes représentent le modèle de l'objet à créer. Une classe peut également hériter d'une autre classe.
- En JavaScript, un prototype est une instance d'un objet. L'héritage se fait directement entre objets via le chaînage de prototypes.

Il n'y a pas de classes en JavaScript. Il n'y a que des fonctions.

Le mot clé 'class' introduit dans la version ES6 n'est qu'un sucre syntaxique pour désigner une fonction constructrice (constructor function).



# Les objets (propriétés statiques)

On peut définir des propriétés statiques en définissant ces propriétés directement au niveau de la fonction

```
Eleve.description = "Ceci est un élève";  
Eleve.bonjour = function(){ console.log("Bonjour élève")};
```

On ne peut pas récupérer une propriété statique via une instance

```
console.log(eleve1.description); //undefined  
console.log(Eleve.description); //Ceci est un élève
```

On ne peut pas appeler une fonction statique via une instance

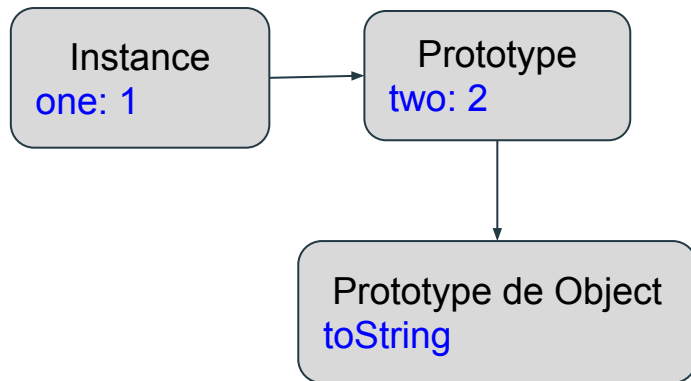
```
eleve1.bonjour(); //error  
Eleve.bonjour(); //Bonjour élève
```

# L'opérateur in

L'opérateur 'in' renvoie true si une propriété donnée appartient à l'objet donné (directement ou via sa chaîne de prototype).

```
const obj = {  
  one: 1  
};  
Object.setPrototypeOf(obj, {two: 2});
```

```
console.log('one' in obj); //true  
console.log('two' in obj); //true  
console.log('three' in obj); //false  
console.log('toString' in obj); //true
```



# Les fonctions toString() et valueOf()

Les fonctions toString() et valueOf() sont définies au niveau du prototype de l'objet.

Elles sont automatiquement appelées dans certaines conditions.

**toString()** renvoie une chaîne de caractères représentant l'objet.

**valueOf()** renvoie la valeur primitive d'un objet donné.

Lorsqu'une interpolation de chaîne de caractères est utilisée, la méthode toString() est appelée.

Lorsque l'opérateur '+' est utilisé, la méthode valueOf() est appelée si elle est définie sinon la méthode toString() est appelée

# Les fonctions toString() et valueOf()

Appel des fonctions toString() et valueOf() implicitement

```
class Test {  
    constructor(nom) { this.nom = nom; }  
    toString(){ return `toString: ${this.nom}`; }  
    valueOf(){ return `valueOf: ${this.nom}`; }  
}  
  
const test = new Test("A");  
console.log(test); //Test { nom: 'A' }  
console.log(`${test}`); //toString: A  
console.log(test+ ""); //valueOf: A  
console.log(test.toString()); //toString: A  
console.log(test.valueOf()); //valueOf: A
```

# Le mot clé this

Dans la plupart des langages le mot clé 'this' fait référence à l'instance courante. En JavaScript le mot clé 'this' se comporte différemment en fonction du contexte.

Pour les fonctions non fléchées, 2 cas de figures

1. Si la fonction est appelée comme constructeur (à l'aide de 'new'), la valeur de 'this' est l'objet en cours de création.
2. Sinon la valeur de 'this' sera déterminée à partir de la façon dont une fonction est appelée.

Pour les fonctions fléchées, la valeur du 'this' est capturée lors de la déclaration.

En dehors des fonctions, 'this' fait référence à l'objet global ou à undefined en mode strict.

# Le mot clé this

Fonction appelée comme un constructeur (à l'aide du mot clé 'new')

```
function Text() {  
  console.log(this);  
  this.texte = "Bonjour";  
}  
const t = new Text;  //Text {}
```

Le mot clé 'this' fait référence à l'objet en cours de création.

# Le mot clé this

Fonction appelée directement

```
function Text() {  
  console.log(this);  
}  
Text(); //undefined
```

Fonction appelée à partir d'un objet

```
const obj = {  
  afficher: function(){ console.log(this) }  
}  
obj.afficher(); //{ afficher: [Function: afficher] }  
const f = obj.afficher;  
f(); //undefined
```

# Le mot clé this

Les fonctions fléchées ne possèdent pas de propre valeur pour 'this' la valeur est donc capturée lors de la déclaration.

```
const obj = {  
  afficher: () => console.log(this)  
}  
obj.afficher(); //affiche l'objet global ou un objet vide (pour Node)
```



# Le mot clé this

Utilisation des fonctions fléchées dans les fonctions constructrices

```
function Vide() {  
  this.afficher = () => console.log(this);  
}  
Vide.prototype.afficherPrototype = () => console.log(this);  
const obj = new Vide();  
obj.afficherPrototype(); //affiche globalThis ou un objet vide (pour Node)  
obj.afficher(); //Vide { afficher: [Function (anonymous)] }  
const f = obj.afficher;  
f(); //Vide { afficher: [Function (anonymous)] }
```

# Les mots clés call apply bind

Dans certains cas, on veut explicitement attribuer une valeur 'this' lors de l'appel d'une fonction.

## La fonction call

La fonction 'call' réalise un appel à une fonction avec une valeur 'this' donnée.

```
const obj = {  
  afficher: function(){ console.log(this) }  
}  
const f = obj.afficher;  
f(); //undefined  
f.call(obj);    //{ afficher: [Function: afficher] }
```

# Les mots clés call apply bind

## La fonction apply

Se comporte comme la fonction call, sauf que les paramètres sont passés dans un tableau

```
const obj = {  
  afficher: function(p1,p2){ console.log(this, p1, p2) }  
}  
const f = obj.afficher;  
f('p1','p2'); //undefined p1 p2  
f.call(obj,'p1','p2'); // { afficher: [Function: afficher] } p1 p2  
f.apply(obj,['p1','p2']); // { afficher: [Function: afficher] } p1 p2
```

# Les mots clés call apply bind

## La fonction bind

La fonction 'bind' crée une nouvelle fonction qui, lorsqu'elle est appelée, a pour contexte 'this' la valeur passée en paramètre.

```
const obj = {  
  afficher: function(p1,p2){ console.log(this, p1, p2) }  
}  
const f = obj.afficher  
f.call(obj, 'p1', 'p2');    //{ afficher: [Function: afficher] } p1 p2  
f.apply(obj, ['p1', 'p2']); //{ afficher: [Function: afficher] } p1 p2  
const fBound = obj.afficher.bind(obj);  
fBound('p1', 'p2');    //{ afficher: [Function: afficher] } p1 p2
```

# Les closures

Une closure est la paire formée d'une fonction et des références à son état environnant (l'environnement lexical). Une closure donne accès à la portée d'une fonction externe à partir d'une fonction interne.

La fonction ci-dessous affiche des nombres de 0 à 9

```
const tab = [];  
for (let i=0; i<10; ++i)  
  tab.push(() => console.log(i));  
for (let element of tab)  
  element();
```

# Les closures

Le code suivant permet de capturer la variable 'idx' qui n'est plus accessible de l'extérieur.

```
function index(idx) {  
  return ({  
    afficher: () => console.log({idx}),  
    inc: () => ++idx,  
    dec: () => --idx  
  });  
}  
  
const p1 = index(1);  
const p2 = index(2);  
p1.afficher(); // { idx: 1 }  
p2.afficher(); // { idx: 2 }  
p1.dec();  
p2.inc();  
p1.afficher(); // { idx: 0 }  
p2.afficher(); // { idx: 3 }
```

# Les closures

Simulation de la fonction 'bind' à l'aide d'une closure

```
const obj = {  
  afficher: function(){ console.log(this) }  
}  
function myBind(f, obj) {  
  const fBound = () => f.call(obj);  
  return fBound;  
}  
const f1 = obj.afficher;  
f1(); //undefined  
const f2 = myBind(f1, obj);  
f2(); //{ afficher: [Function: afficher] }
```

# Les closures

Les fonctions fléchées ne possédant pas leurs propre valeur de 'this', capture celle-ci à l'aide d'une closure.

```
function Capture(text){
  this.text = text;
  //capture de la valeur du this lors de la déclaration
  this.afficherThis = () => console.log(this);
}
const obj = new Capture("Capture du this");
const afficher = obj.afficherThis;
afficher(); //Capture { text: 'Capture du this', afficherThis: ...}
```



# Le mot clé class

Le mot clé 'class' a été introduit avec ECMAScript 2015. Les classes sont un « sucre syntaxique » par rapport à l'héritage prototypal.

En effet, cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript!

Elle fournit uniquement une syntaxe plus simple pour créer des objets et manipuler l'héritage.

```
class Rectangle {  
  constructor(hauteur, largeur) {  
    this.hauteur = hauteur;  
    this.largeur = largeur;  
  }  
}  
  
const rect = new Rectangle(50, 100);
```

# Le mot clé class

Dans l'exemple suivant, toutes les propriétés et fonctions sont définies au niveau de l'instance. Néanmoins le constructeur est défini au niveau du prototype.

```
class Rectangle {  
  constructor(hauteur, largeur) {  
    this.hauteur = hauteur;  
    this.largeur = largeur;  
    this.afficherType = () => console.log(typeof Rectangle);  
  }  
  afficherNomFonction = () => console.log(Rectangle.name);  
}  
const rect = new Rectangle(50, 100);  
rect.afficherType(); //function  
rect.afficherNomFonction(); //Rectangle  
console.log(rect);  
//Rectangle {afficherNomFonction: ..., hauteur: 50, largeur: 100, afficherType: ...}
```

# Le mot clé class

Déclaration des méthodes au niveau du prototype

```
class Rectangle {  
  constructor(hauteur, largeur) { this.hauteur = hauteur; this.largeur = largeur; }  
  afficherType(){ console.log(typeof Rectangle); }  
}  
Rectangle.prototype.afficherNomFonction = () => console.log(Rectangle.name);
```

Les fonctions constructor, afficherType, afficherNomFonction sont définies au niveau du prototype

```
Object.getOwnPropertyNames(Rectangle.prototype).forEach(item => console.log(item));  
// constructor  
// afficherType  
// afficherNomFonction
```

# Le mot clé class

**Attention, les fonctions définies dans les classes ne sont pas énumérables.**

On ne peut pas énumérer les fonctions définies dans la classe précédente à l'aide d'une boucle for/in

```
for(let key in Rectangle.prototype)
  console.log(key);
// afficherNomFonction
```

Ces propriétés ne peuvent pas être copiées en utilisant la fonction Object.assign ou l'opérateur de spread.

```
console.log(Object.assign({}, Rectangle.prototype));
console.log({...Rectangle.prototype});
// { afficherNomFonction: [Function (anonymous)] }
// { afficherNomFonction: [Function (anonymous)] }
```

# Le mot clé class

Le mot clé 'static' permet de définir des fonctions et des propriétés statiques

```
class ClassWithStaticMethod {  
    static staticProperty = 'valeur de staticProperty';  
    static staticMethod() { return 'method statique'; }  
    static {  
        console.log('Initialisation de la classe statique');  
    }  
}  
  
//Initialisation de la classe statique  
console.log(ClassWithStaticMethod.staticProperty); //valeur de staticProperty  
console.log(ClassWithStaticMethod.staticMethod()); //method statique
```

# Le mot clé class (propriétés privées)

Pour déclarer des propriétés ou des méthodes privées, il faut faire précéder le nom par un '#'.

```
class Rectangle {  
  
    //propriétés privées  
    #hauteur;  
    #largeur;  
    constructor(hauteur, largeur) {  
        this.#hauteur = hauteur;  
        this.#largeur = largeur;  
    }  
    //méthode privée  
    #afficherPrive(){ console.log(`hauteur: ${this.#hauteur}, largeur: ${this.#largeur}`); }  
    afficher(){ this.#afficherPrive(); }  
}
```

# Les getters et setters

Les getters sont des fonctions implicitement appelées lors de la lecture d'une propriété d'un objet.

Les setters sont des fonctions implicitement appelées lors de la modification d'une propriété d'un objet.

Pour déclarer un getter, il faut utiliser le mot clé **'get'**.

Pour déclarer un setter, il faut utiliser le mot clé **'set'**.

```
class Simple {  
  get prop() { console.log("getter");}  
  set prop(value) { console.log(`setter: ${value}`);}  
}  
const obj = new Simple;  
obj.prop; //getter  
obj.prop = 'OK'; //setter: OK
```

# Les getters et setters

Voici un exemple plus complexe de l'utilisation des getters et setters

```
class maClasse {  
  #texte;  
  constructor (texte) { texte = texte; }  
  get texte() { console.log("lecture de la variable");  
    return this.#texte;  
  }  
  set texte(value) { console.log("écriture de la variable");  
    this.#texte = value;  
  }  
}  
  
const obj = new maClasse("ES6");  
const t = obj.texte; //lecture de la variable  
obj.texte = "ES7"; //écriture de la variable  
console.log(obj); //maClasse {}
```



# Le mot clé class (héritage)

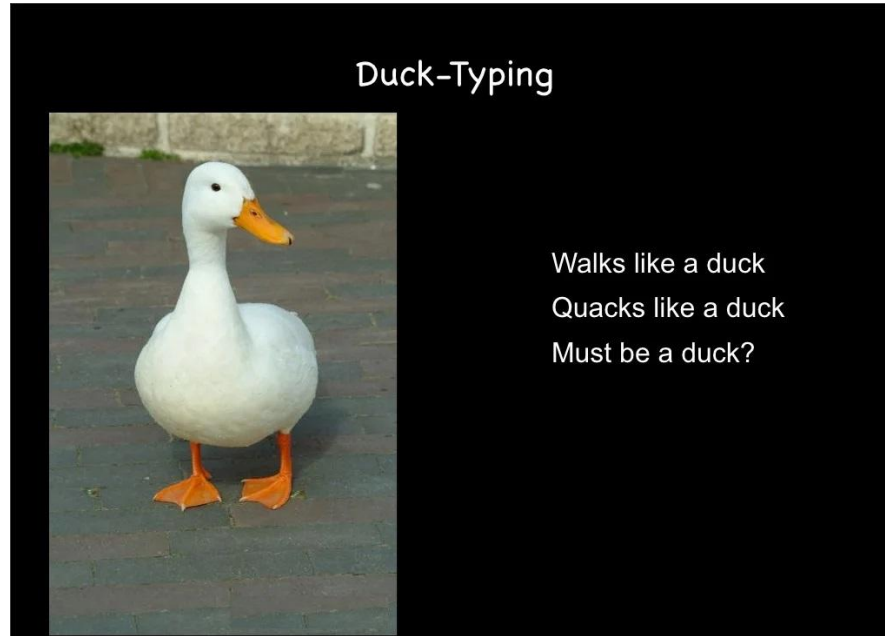
Dans l'exemple suivant la classe Cercle hérite de la classe Forme. Cela veut dire que le prototype de Cercle hérite du prototype de Forme.

Le mot clé **'extends'** permet d'étendre une classe parente.

Le mot clé **'super'** fait référence au constructeur de la classe parente.

```
class Forme {  
    constructor(type) {  
        this.type = type;  
    }  
}  
  
class Cercle extends Forme {  
    constructor(rayon) {  
        super("Cercle");  
        this.rayon = rayon;  
    }  
}
```

# Duck type



# Duck type

Le Duck typing est une technique qui utilise les propriétés et méthodes d'un objet pour déterminer son type.

```
class Duck{
  quack = () => console.log("Le canard cancanne");
}
class Dog{
  bark = () => console.log("Le chien aboie");
}
const animaux = [new Duck(), new Dog(), { quack: () => console.log("Un intrus cancanne")}];
for (const animal of animaux) {
  if (animal.quack)
    animal.quack();
}
//Le canard cancanne
//Un intrus cancanne
```

# Les mixins

Un mixin est une technique qui permet de réutiliser un ensemble de fonctions précédemment définies dans un objet sans passer par l'héritage.

```
const creerDessinateur = () => (  
  { dessiner: (obj) => console.log(`Dessiner ${obj}`) }  
);  
const creerSculpteur = () => (  
  { sculpter: (obj) => console.log(`Sculpter ${obj}`) }  
);  
const creerDessinateurSculpteur = () => Object.assign(creerDessinateur(), creerSculpteur());  
const zebre = creerDessinateurSculpteur();  
zebre.dessiner("cercle"); //Dessiner cercle  
zebre.sculpter("carré"); //Sculpter carré
```

# Destructuring

Le Destructuring (affectation par décomposition) est une expression JavaScript qui permet d'extraire des données d'un tableau ou d'un objet.

Exemple de destructuring de tableau

```
const [a, b] = [2,3,5,7,11];  
console.log(a); //2  
console.log(b); //3
```

Destructuring d'un objet. Les noms des propriétés à gauche de l'opérande d'affectation doivent correspondre aux noms des propriétés de l'objet à droite de l'opérande.

```
const { one, four, three : trois } = { one: 1, two: 2, three: 3, four: 4 };  
console.log(one); //1  
console.log(trois); //3  
console.log(four); //4
```

# Spread

Le spread (syntaxe de décomposition) permet d'étendre un itérable en lieu et place de plusieurs arguments (pour les appels de fonctions) ou de plusieurs éléments (pour les tableaux) ou de paires clés-valeurs (pour les objets).

Spread lors d'un appel de fonction

```
const somme = (a, b, c) => a+b+c;  
const nombres = [1, 2, 3];  
console.log(somme(...nombres)); //6
```

# Spread

L'opérateur de spread avec un tableau

```
const nombres = [1, 2, 3];  
const nombres2 = [...nombres, 4];  
console.log(nombres2); // [1, 2, 3, 4]
```

L'opérateur de spread avec un objet

```
const obj = { one: 1, two: 2 };  
const obj2 = { ...obj, three: 3 };  
console.log(obj2); //{ one: 1, two: 2, three: 3 }
```

# L'opérateur du reste

L'opérateur du reste permet de représenter un nombre indéfini d'arguments sous forme d'un tableau.

Opérateur reste lors de l'appel d'une fonction

```
const afficher = (...elements) => console.log(elements);  
afficher(1, 2, 3); // [1, 2, 3]
```

Opérateur reste lors de l'initialisation d'un tableau

```
const [premier, ...rest] = [2, 3, 5, 7, 11];  
console.log(premier); // 2  
console.log(rest); // [ 3, 5, 7, 11 ]
```



# L'opérateur du reste

Opérateur reste lors de l'initialisation d'un objet

```
const {one, ...rest} = {one: 1, two:2, three: 3};  
console.log(one);    //1  
console.log(rest);   //{ two: 2, three: 3 }
```

# Les itérateurs

Un itérateur est un objet qui définit une séquence.

Un itérateur est un objet qui expose une fonction `next()` qui retourne un objet ayant les 2 propriétés suivantes

- `value`: représente l'index suivant de la séquence.
- `done`: si la séquence est terminée retourne `true` sinon retourne `false`.

Voici un exemple d'itérateur

```
let indexSuivant = 0;
const iter = {
  next: () => ({ value: indexSuivant++, done: indexSuivant > 5 })
};
```

# Les itérateurs

Les fonctions retournant un itérateur permettent d'encapsuler la première valeur d'une séquence

```
function creerIterateur(nb) {  
  let indexSuivant = 0;  
  return {  
    next: () => ({ value: indexSuivant++, done: indexSuivant > nb })  
  }  
}  
  
const iter = creerIterateur(5);  
let {value, done} = iter.next();  
while (!done){  
  console.log(value); //0 1 2 3 4  
  //des parenthèses pour que JS ne considère pas l'objet à gauche de l'opérateur comme un bloc  
  ({value, done} = iter.next());  
}
```

# Les générateurs

Un générateur est un type de fonction spécial qui retourne un itérateur.

Une fonction devient un générateur lorsqu'elle contient une ou plusieurs expressions 'yield' et qu'elle utilise la syntaxe 'function\*'.

Les générateurs permettent de simplifier l'écriture des itérateurs.

```
function* creerIterateur(nb) {  
  for (let indexSuivant = 0; indexSuivant < nb; ++indexSuivant)  
    yield indexSuivant;  
}
```

# Les générateurs

Les générateurs sont utilisés de la même manière que les itérateurs

```
function* creerIterateur(nb) {  
  for (let indexSuivant = 0; indexSuivant < nb; ++indexSuivant)  
    yield indexSuivant;  
}  
const iter = creerIterateur(5);  
let {value, done} = iter.next();  
while (!done){  
  console.log(value); //0 1 2 3 4  
  //des parenthèses pour que JS ne considère pas l'objet à gauche de l'opérande comme un bloc  
  ({value, done} = iter.next());  
}
```

# Les itérables

Un objet est considéré comme itérable s'il implémente la méthode '@@iterator', cela signifie que l'objet doit avoir une propriété avec la clé 'Symbol.iterator'. Cette propriété doit retourner un itérateur.

Les itérables permettent de parcourir la séquence à l'aide de boucles for..of

```
const iterable = {  
  [Symbol.iterator]: function*() {  
    for (let indexSuivant = 0; indexSuivant < 5; ++indexSuivant)  
      yield indexSuivant;  
  }  
}  
  
for(const element of iterable)  
  console.log(element); //0 1 2 3 4
```

# Les Sets

Un objet Set permet de stocker un ensemble de valeurs uniques de n'importe quel type.

Créer un Set

```
const items = new Set();  
const items2 = new Set([1,2,3,2]);
```

L'objet Set ne garde que les valeurs uniques

```
console.log(new Set([1,2,3,2])); // Set(3) { 1, 2, 3 }
```

# Les Sets

Manipulation des objets Set.

```
const mySet1 = new Set();  
console.log(mySet1.add(1)); //Set(1) { 1 }  
console.log(mySet1.add(5)); //Set(2) { 1, 5 }  
console.log(mySet1.has(1)); //true  
console.log(mySet1.has(3)); //false  
console.log(mySet1.delete(5)); //true  
console.log(mySet1.size); //1  
console.log(mySet1); //Set(1) { 1 }
```



# Les Maps

Un objet Map contient des paires de clé-valeur et mémorise l'ordre dans lequel les clés ont été insérées. La clé doit être une valeur unique non nulle et immuable.

Créer un dictionnaire

```
let dico1 = new Map();  
let dico2 = new Map([['key1', 'value1'], ['key2', 'value2']]);
```

Si des clés de mêmes valeurs sont insérées, l'objet Map ne garde que la dernière valeur.

```
console.log(new Map([['key1', 1], ['key1', 2]])); //Map(1) { 'key1' => 2 }
```

# Les Maps

Manipulation des dictionnaires

```
const log = console.log;
const map1 = new Map();
log(map1.set('a', 1)); //Map(1) { 'a' => 1 }
log(map1.set('b', 2)); //Map(2) { 'a' => 1, 'b' => 2 }
log(map1.get('a')); //1
log(map1.set('a', 97)); //Map(2) { 'a' => 97, 'b' => 2 }
log(map1.get('a')); //97
log(map1.delete('b')); //true
log(map1.size); //1
log(map1); //Map(1) { 'a' => 97 }
```

# Les Maps (où sont mes valeurs ?)

**Attention!!! Ne pas utiliser les [] pour accéder ou modifier un élément du dictionnaire.**

```
const log = console.log;
const map1 = new Map();
//!Attention ajoute la propriété 'a' à l'objet map1 pas au dictionnaire!
map1['a'] = 1;
log(map1.get('a')); //undefined
log(map1.a); //1
log(map1.set('z', 26)); //Map(1) { 'z' => 26, a: 1 }
//!Attention retourne la propriété 'z' de l'objet map1
log(map1['z']); //undefined
```

# Les Symbols

Le type Symbol permet de créer des valeurs primitives dont l'unicité est garantie. Les symboles sont souvent utilisés pour ajouter des clés de propriétés uniques à un objet afin que celles-ci ne rentrent pas en conflit avec des clés ajoutées par un autre code.

Création d'une valeur Symbol.

**Attention! Ne pas utiliser de mot clé 'new' pour la création d'un symbole.**

```
let symbole1 = Symbol();  
let symbole2 = Symbol('toto');
```

L'unicité est garantie

```
console.log(Symbol('toto') === Symbol('toto')); //false
```

# Les Symbols

Comme nous l'avons vu précédemment, les symboles permettent d'implémenter certaines méthodes spécifiques. Comme la méthode @@iterator qui permet de définir un itérateur.

```
const iterable = {  
  [Symbol.iterator]: function*() {  
    yield 0;  
    yield 1;  
    yield 2;  
  }  
}  
for(const element of iterable)  
  console.log(element); //0 1 2
```

# Les Symbols

D'autres méthodes peuvent être implémentée comme le `@@toStringTag` qui permet de définir un tag spécifique à l'objet lors de l'appel à la méthode `toString()`

```
console.log((new Map()).toString()); //[[object Map]]
const obj = {};
console.log(obj.toString()); //[[object Object]]
Object.assign(obj, { get [Symbol.toStringTag]() { return "Special" } });
console.log(obj.toString()); //[[object Special]]
```

Les Symbols permettent également d'avoir une sorte d'encapsulation. Par exemple, si je n'avais pas accès à la valeur de mon symbole, je ne pourrais pas appeler ma méthode comme ci-dessous

```
console.log(obj[Symbol.toStringTag]); //Special
```

# Les enumerations

Il n'existe pas de type enum en JavaScripts. A la place vous devez utiliser une variable de type objet.

Voici comment on peut représenter une énumération en JavaScript. La fonction `Object.freeze` empêche d'ajouter ou de modifier une propriété de mon objet.

```
const Cote = Object.freeze({  
  Pile: Symbol("Pile"),  
  Face: Symbol("Face")  
});
```

Utilisation de l'énumération précédemment définie

```
const pieceCote = Cote.Face;  
console.log(pieceCote === Cote.Face); //true  
console.log(pieceCote.description); //Face
```