Javascript

Said Gounane
2020/2021

Sommaire

Introduction

- Le JavaScript est un langage de programmation créé en 1995.
- Le JavaScript est aujourd'hui l'un des langages de programmation les plus populaires et il fait partie des langages web dits « standards » avec le HTML et le CSS.
- Son évolution est gérée par le groupe ECMA International qui se charge de publier les standards de ce langage.

Introduction

- Le JavaScript est un langage dynamique ;
- Le JavaScript est un langage côté client et aussi côté serveur ;
- Le JavaScript est un langage (interprété) JIC (just-In-Time Compilation) ;
- Le JavaScript est un langage orienté objet.

Introduction: API, librairies et framework

- **Une API** (« Application Programming Interface » ou « Interface de Programmation ») est une interface qui nous permet d'utiliser facilement une application. (Google Maps API, Twitter API)
- **Une librairie** ou « bibliothèque » JavaScript est un ensemble de fichiers de code JavaScript homogènes qui se concentrent sur un aspect particulier du langage qu'on va devoir télécharger pour les utiliser.(jQuery).
- **Un framework** ou « cadre de travail » est relativement similaire à une « super librairie ». lorsqu'on télécharge une librairie, on peut l'utiliser comme on le souhaite tandis que pour utiliser un framework il faut respecter ses règles. (Angular.js et React.js)

Introduction: Où écrire le code JavaScript?

1. Directement dans la balise ouvrante d'un élément HTML ;

   <button onclick="alert('bonjour!!')"> cklic me</button>

2. Dans un élément script, au sein d'une page HTML ;

   <script> alert("Bonjour!!") </script>

3. Dans un fichier séparé contenant exclusivement du JavaScript et portant l'extension .js.

   <script src="monScript.js" async></script>

LES VARIABLES

- Une variable est un conteneur servant à stocker des informations de manière temporaire.
- Pour déclarer une variable en JavaScript, nous allons devoir utiliser le mot clé ***var*** ou le mot clé ***let***.
- Le nom d'une variable doit obligatoirement commencer par une lettre ou un underscore (_) et ne doit pas commencer par un chiffre ;
- Le nom d'une variable ne doit contenir que des lettres, des chiffres et des underscores mais pas de caractères spéciaux ;
- Le nom d'une variable ne doit pas contenir d'espace.
- En JavaScript le nom des variables est sensible à la casse.

  let nom;

LES VARIABLES

- Initialiser une variable

  Let a=5

  Var nom="Adnane"

  nom="Yasser"

  a="bonjour!!"

LES VARIABLES: let vs var

- Avec ***var*** on peut effectuer des manipulations en haut du code et la déclarer en fin de code car le JavaScript va traiter les déclarations de variables effectuées avec var avant le reste du code JavaScript. (remontée)

  nom="yasser";

  Var nom;

- les variables utilisant la syntaxe **let** doivent obligatoirement être déclarées avant de pouvoir être utilisées.

  let nom;

  nom="yasser";

LES VARIABLES: let vs var

- Avec **var**, on avait le droit de déclarer plusieurs fois une même variable en utilisant à chaque fois ***var***.

  Var nom="Adnane";

  Var nom="Yasser";

- La nouvelle syntaxe avec ***let*** n'autorise pas cela. Pour modifier la valeur stockée dans une variable avec la nouvelle syntaxe, il suffit d'utiliser le nom de la variable et de lui affecter une autre valeur.

LES VARIABLES: let vs var

- La portée des variables:

  Les variables déclarées avec ***var*** et celles avec ***let*** au sein d'une fonction n'ont pas la même portée, c'est-à-dire qu'on ne va pas pouvoir les utiliser aux mêmes endroits.

LES VARIABLES: let vs var

- La syntaxe de déclaration des variables avec ***let*** correspond à la nouvelle syntaxe. La syntaxe avec ***var*** est l'ancienne syntaxe qui est vouée à disparaître.
- Vous devriez toujours utiliser le mot clé ***let*** pour déclarer vos variables.

Les types de données en JavaScript

- En JavaScript, il existe 7 types de données:
  - ***String;***
  - ***Number;***
  - ***Boolean;***
  - ***Null;***
  - ***Undefined;***
  - ***Symbol;***
  - ***Object;***
- En JavaScript on n'a pas besoin de préciser à priori le type de valeur qu'une variable va pouvoir stocker.
- on peut utiliser la fonction ***typeof(variable)***pour vérifier le type d'une variable

Les types de données en JavaScript

- Les types de valeurs ***Null*** et ***Undefined*** sont des types un peu particuliers car ils ne contiennent qu'une valeur chacun : les valeurs ***null*** et ***undefined***.
- La valeur ***null*** correspond à l'absence de valeur connue. Pour qu'une variable contient ***null***, il va falloir stocker cette valeur qui représente donc l'absence de valeur de manière explicite.
- La valeur ***undefined*** correspond à une variable « non définie », c'est-à-dire une variable à laquelle on n'a pas affecté de valeur.
- Si on déclare une variable sans lui attribuer de valeur, alors son type sera ***Undefined***. Si on déclare une variable et qu'on lui passe ***null***, alors son type sera ***Object***.

Les opérateurs arithmétiques

Opérateur	Nom de l'opération associée
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste d'une division euclidienne)
**	Exponentielle (élévation à la puissance d'un nombre par un autre)

Les opérateurs d'affectation

Opérateur	Nom de l'opération associée
+=	Additionne puis affecte le résultat
-=	Soustrait puis affecte le résultat
*=	Multiplie puis affecte le résultat
/=	Divise puis affecte le résultat
%=	Modulo (reste d'une division euclidienne)

Opérateurs de comparaison

Opérateur	Définition
==	Permet de tester l'égalité sur les valeurs
===	Permet de tester l'égalité en termes de valeurs et de types
!=	Permet de tester la différence en valeurs
<>	Permet également de tester la différence en valeurs
!==	Permet de tester la différence en valeurs
<	Permet de tester si une valeur est strictement inférieure à une autre
>	Permet de tester si une valeur est strictement supérieure à une autre
<=	Permet de tester si une valeur est inférieure ou égale à une autre
>=	Permet de tester si une valeur est supérieure ou égale à une autre

Opérateurs logique

Nom	Symbole	description
AND	&&	Lorsqu'il est utilisé avec des valeurs booléennes, renvoie ***true*** si ***toutes*** les comparaisons sont évaluées à ***true*** ou ***false*** sinon
OR	||	Lorsqu'il est utilisé avec des valeurs booléennes, renvoie ***true*** si au ***moins*** l'une des comparaisons est évaluée à ***true*** ou ***false*** sinon
NOT	!	Renvoie ***false*** si une comparaison est évaluée à ***true*** ou renvoie ***true*** dans le cas contraire

La concaténation

- La concaténation est un mot généralement utilisé pour désigner le fait de rassembler deux chaînes de caractères pour en former une nouvelle.
- En JavaScript, l'opérateur de concaténation est le signe +.
  - Lorsque le signe + est utilisé avec deux ***nombres***, il sert à les ***additionner***.
  - Lorsqu'il est utilisé avec autre chose que deux nombres, il sert d'opérateur de concaténation.
  - Si on utilise l'opérateur + pour concaténer une ***chaîne*** de caractères puis un ***nombre***, alors le JavaScript va considérer le nombre comme une ***chaîne*** de caractères.

```
let x=4+'3';

let y="salut"+'tous';
```


Les littéraux de gabarits

- En javascript les chaînes sont toujours entourés avec des apostrophes ou des guillemets droits
- On peut aussi utiliser les accents graves **`**.
- Toute expression placée entre les accents graves va être interprétée en JavaScript. **Mais** il va falloir placer les expressions entre ***${*** et ***}***.

```
let age =20;

console.log(`j'ai ${age} ans`);
```

- l'utilisation des littéraux de gabarits conserve les retours à la ligne et les décalages dans le résultat final.

Les constantes en JavaScript

- Une constante est similaire à une variable. Cependant, à la différence des variables, on ne va pas pouvoir modifier la valeur d'une constante.
- Pour créer ou déclarer une constante en JavaScript, nous allons utiliser le mot clef ***const***.

  const pi = 3.14;

  const planete = 'Mars';

Structures de contrôle

- On appelle « structure de contrôle » un ensemble d'instructions qui permet de contrôler l'exécution du code.
- Il existe deux grands types de structure de contrôle de base qu'on retrouve dans la plupart des langages informatiques et notamment en JavaScript :
  - ***Les structures de contrôle conditionnelles*** (ou plus simplement les « conditions »)
  - ***Les structures de contrôle de boucles*** (ou plus simplement les « boucles »).

Structures de contrôle conditionnelles

- Les structures de contrôle conditionnelles (ou plus simplement conditions) vont nous permettre d'exécuter une série d'instructions si une condition donnée est vérifiée ou (éventuellement) une autre série d'instructions si elle ne l'est pas.
- Nous avons accès aux structures conditionnelles suivantes en JavaScript :
  - La condition ***if(test)*** ;
  - La condition ***if(test)… else*** ;
  - La condition ***if(test1)… elseif(test2)… else***.

Structures de contrôle conditionnelles

- La condition ***if(test)*** va juste nous permettre d'exécuter un bloc de code si et seulement si le résultat d'un ***test*** vaut ***true***.
- Toute valeur évaluée par le JavaScript dans un contexte booléen va être évaluée à true à l'exception des valeurs suivantes:
  - Le booléen ***false*** ;
  - La valeur ***0*** ;
  - Une ***chaîne de caractères vide*** ;
  - La valeur ***null*** ;
  - La valeur ***undefined*** ;
  - La valeur ***NaN*** (« Not a Number » = « n'est pas un nombre »).

Structures de contrôle conditionnelles: ternaires

- Les structures conditionnelles ***ternaires*** correspondent à une autre façon d' écrire nos conditions en utilisant une syntaxe basée sur l'opérateur ternaire ***? :***.
- Les structures ternaires vont se présenter sous la forme suivante :

```
test ? code à exécuter si true : code à exécuter si false;

x>10 ? console.log('x>10') : console.log('x<=10');
```


Structures de contrôle conditionnelles: switch

```
let x=2;

switch(x){
    case val1 :  instructions1; break;
    case val2 :  instructions2; break;
    case val3 :  instructions3; break;
    Default :  instructions4;
}
```


Structures de contrôle boucle

- Les boucles vont nous permettre d'exécuter plusieurs fois un bloc de code, tant qu'une condition donnée est vérifiée.
- Nous disposons de six boucles différentes en JavaScript :
  - La boucle ***while*** (« tant que ») ;
  - La boucle ***do… while*** (« faire… tant que ») ;
  - La boucle ***for*** (« pour ») ;
  - La boucle ***for… in*** (« pour… dans») ;
  - La boucle ***for… of*** (« pour… parmi ») ;
- Les boucles se composent de trois choses :
  - Une valeur de départ pour initialiser la boucle et nous servir de compteur ;
  - Un test de sortie qui précise le critère de sortie de la boucle ;
  - Un itérateur qui va modifier la valeur de départ de la boucle à chaque nouveau passage jusqu'au moment où la condition de sortie est vérifiée.

Les fonctions

- Une fonction correspond à un bloc de code nommé et réutilisable et dont le but est d'effectuer une tâche précise.
- Le langage JavaScript dispose de nombreuses fonctions que nous pouvons utiliser pour effectuer différentes tâches
- le code d'une fonction est réutilisable : cela veut dire qu'on va pouvoir appeler une même fonction autant de fois qu'on le souhaite afin qu'elle accomplisse plusieurs fois la même opération.
- Pour exécuter le code d'une fonction, il suffit de l'appeler. Pour faire cela, on n'a qu'à écrire le nom de la fonction suivi d'un couple de parenthèses et éventuellement préciser des arguments entre les parenthèses.

Les fonctions

- Pour définir une fonction, on va utiliser le mot clé ***function*** suivi du nom que l'on souhaite donner à notre fonction puis d'un couple de parenthèses dans lesquelles on peut définir des paramètres et d'un couple d'accolades dans lesquelles on va placer le code de notre fonction.

```
function maFonction(arg1, arg2, ...){
    Instruction1;
    Instruction2;
    return maVar;
}
```


Portée des variables (scop)

- La « portée » d'une variable désigne l'espace du script dans laquelle elle va être accessible.
- En JavaScript, il n'existe que deux espaces de portée différents :
  - ***l'espace global*** : l'entièreté d'un script à l'exception de l'intérieur de nos fonctions.
  - ***L'espace local*** : l'espace dans une fonction.
- ***l'espace global*** d'un script va être accessible à travers ***tout le script***, même depuis une fonction.
- Une ***variable définie dans une fonction*** n'est accessible que dans ***cette même fonction*** et ne peut pas être manipulée depuis l'espace global du script.

Portée des variables (scop): var vs let

- Lorsqu'on utilise ***let*** pour définir une variable à l'intérieur d'une fonction en JavaScript, la variable va avoir une portée dite « ***de bloc*** » : la variable sera accessible dans le ***bloc*** dans lequel elle a été définie et dans les blocs que le bloc contient.
- Une variable avec le mot clé ***var*** dans une fonction aura une portée élargie puisque cette variable sera alors accessible dans tous les blocs de la fonction.

Fonctions anonymes

- Les fonctions anonymes sont des fonctions qui ne possèdent pas de nom.
- On utilise les fonctions anonymes lorsqu'on n'a pas besoin d'appeler notre fonction par son nom c'est-à-dire lorsque le code de notre fonction n'est appelé qu'à un endroit dans notre script.
- On crée une fonction anonyme de la même façon qu'une fonction classique, en utilisant le mot clé ***function*** mais en ***omettant le nom*** de la fonction après.

```
function(arg1, arg2, ...){
    ....
}
```

Fonctions anonymes

Pour exécuter une fonction anonyme :

1. Enfermer le code de notre fonction dans une variable et utiliser la variable comme une fonction ;

```
let f=function(){...};   // definition
f();  // execution
```

2. Auto-invoquer la fonction anonyme ;

```
(function(){...})();  //definition et execution
```

3. Utiliser un ***événement*** pour déclencher l'exécution de notre fonction.

Les fonctions récursives

- Une fonction récursive est une fonction qui va s'appeler elle-même au sein de son code.
- Les fonctions récursives vont nous permettre d'exécuter une action en boucle et jusqu'à ce qu'une certaine condition de sortie soit vérifiée.

```
Function decompt(n){
    if(n>0){
        console.log(n);
        Return decompt(n-1)
    }
}
```


POO en javascript

- JavaScript est un langage qui intègre l'orienté objet dans sa définition, ce qui fait que tous les éléments du JavaScript vont soit être des objets, soit pouvoir être convertis et traités comme des objets.
- Un objet en JavaScript est un conteneur qui va pouvoir stocker plusieurs variables appelées propriétés.
- Lorsqu'une propriété contient une fonction en valeur, on appelle alors la propriété une méthode.
- Un objet est donc un conteneur qui va posséder un ensemble de ***propriétés*** et de ***méthodes***.

POO en javascript: objet littéral

```
let personne={
    nom: "yahyaoui",
    age: 20,
    direBonjour: function(){ console.log("bonjour!!");}
};
personne.direBonjour();// ou bien personne["direBonjour"]();
personne.age = 23; //ou bien personne["age"] = 23;
pesonne.prenom = "Yasser";
```

POO en javascript: constructeur d'objets

- En javascript on peut créer d'objet à l'aide de constructeur d'objets qui n'est autre qu'une fonction constructeur.
- Pour construire des objets à partir d'une fonction constructeur, nous allons devoir suivre deux étapes :
  - Définir la fonction constructeur
  - Appeler ce constructeur à l'aide du mot clé ***new***.
- Dans une fonction constructeur, on définit un ensemble de propriétés et de méthodes.
- Les objets créés à partir de ce constructeur possèdent automatiquement les propriétés et méthodes définies dans le constructeur.

POO en javascript: constructeur d'objets

```
function Utilisateur(n, a, m){
  this.nom = n;
  this.age = a;
  this.mail = m;
  this.bonjour = function(){console.log('Bonjour, je suis ' + this.nom + ', j\'ai ' + this.age + ' ans'); }
}
Let user1=new Utilisateur("Adnane",24, "adnane@gmail.com");
Let user2=new Utilisateur("yasser",22, "yasser@gmail.com");
user1.tail = 178;  // on peut attribuer d'autre propriétés à l'objet
```


POO en javascript: prototype

- Nous avons pu créer plusieurs objets semblables en appelant plusieurs fois une fonction constructeur personnalisée Utilisateur() et en utilisant le mot clef ***new***.
- On crée alors deux variables qui vont stocker deux objets créés à partir de ce constructeur.
- En utilisant le constructeur plusieurs fois, on va ***copier*** à chaque fois la méthode ***bonjour()*** qui est ***identique*** pour chaque objet.
- l'idéal serait de ne définir cette méthode qu'une seule fois et que chaque objet puisse l'utiliser lorsqu'il le souhaite.

  ⇒ Il faut utiliser les ***prototypes***.

POO en javascript: prototype

- Il existe deux grands types de langages orientés objet :
  - ceux basés sur les **classes**,
  - et ceux basés sur les **prototypes**.
- Dans les langages orientés objet ***basés sur les classes***, tous les objets sont ***créés à partir de classes*** et vont hériter des propriétés et des méthodes définies dans la classe.
- Dans les langages orientés objet ***utilisant des prototypes*** comme le JavaScript, ***tout est objet*** et il ***n'existe pas de classes*** et l'héritage va se faire au moyen de prototypes.

POO en javascript: prototype

- Les ***fonctions*** en JavaScript ***sont des objets***.
- Lorsqu'on crée une fonction, JavaScript va ***automatiquement*** lui ajouter une ***propriété prototype*** qui n'est utile que lorsque la fonction est utilisée comme constructeur.
- Cette propriété ***prototype*** possède une valeur qui est elle-même un objet.
- Par défaut, la propriété ***prototype*** d'un constructeur ne contient que deux propriétés :
  - une propriété ***constructor*** qui renvoie vers le constructeur contenant le prototype
  - une propriété ***__proto__*** qui contient elle-même de nombreuses propriétés et méthodes.
- Lorsqu'on crée un objet à partir d'un constructeur, JavaScript va ajouter automatiquement une propriété ***__proto__*** à l'objet créé égale à la propriété ***__proto__*** du constructeur.

POO en javascript: prototype

- Le contenu de la propriété **prototype** d'un constructeur va être partagé par tous les objets créés à partir de ce constructeur.
- Comme cette propriété est un objet, on va pouvoir lui ajouter des propriétés et des méthodes que tous les objets créés à partir du constructeur vont partager.
- Les objets créés à partir du constructeur ***ne possèdent pas*** vraiment les ***propriétés*** et les ***méthodes*** définies dans la propriété ***prototype*** du constructeur mais vont ***pouvoir y accéder*** et se « partager » ces membres définis dans l'objet **prototype** du constructeur.

POO en javascript: prototype

- Définir des propriétés et des méthodes dans le prototype d'un constructeur permet de les rendre accessibles à tous les objets créés à partir de ce constructeur sans que ces objets aient à les redéfinir.
- Généralement on définit les ***propriétés*** des objets au sein du ***constructeur*** et les ***méthodes*** dans le ***prototype*** du constructeur.
- Les différents objets se « partagent » les mêmes propriétés et les mêmes méthodes définies dans le constructeur.

POO en javascript: La chaîne des prototypes

- Lorsqu'on essaie d'accéder à un ***membre d'un objet***, le navigateur va chercher ce membre au sein de l'objet. S'il n'est pas trouvé,
- Il cherche au sein de la propriété ***__proto__*** de l'objet dont le contenu est égal à celui de la propriété ***prototype*** du constructeur qui a servi à créer l'objet.
- Si le membre est trouvé dans la propriété __proto__ de l'objet (c'est-à-dire s'il a été défini dans la propriété prototype du constructeur), alors il est utilisé.
- Sinon on va aller chercher dans la propriété ***__proto__*** dont dispose également le constructeur et qui va être égale au prototype du constructeur du constructeur.
- Tous les objets en JavaScript descendent par défaut d'un objet de base qui s'appelle ***Object***.
- Object permet de créer des objets génériques vides grâce à la syntaxe ***new Object()***.

POO en javascript: Les classes

- JavaScript a également dans ses dernières versions introduit le mot clé ***class***
- JavaScript est toujours un langage orienté objet à prototypes et, en tâche de fond, il va ***convertir*** nos « ***classes*** » selon son modèle ***prototypes***.
- Les classes JavaScript ne sont donc qu'une nouvelle syntaxe qui nous est proposée par le JavaScript pour les gens plus habitués à travailler avec des langages orientés objet basés sur les classes.

POO en javascript: Les classes

```
//pas besoin de préciser "function" devant constructor et autres méthodes classe
class Ligne{
    constructor(nom, longueur){
        this.nom = nom;
        this.longueur = longueur;
    }
    taille(){
        console.log( 'Longueur de  ' + this.nom + ' : ' + this.longueur )
    };
}
let geo1 = new Ligne('geo1', 10);
let geo2 = new Ligne('geo2', 5);
geo1.taille();
geo2.taille();
```


POO en javascript: Classes étendues et héritage

```
class Rectangle extends Ligne{
    constructor(nom, longueur, largeur){
        super(nom, longueur); //Appelle le constructeur parent
        this.largeur = largeur;
    }
    aire(){
        console.log('Aire de ' + this.nom + ' : ' + this.longueur * this.largeur)
    };
}
let geo3 = new Rectangle('geo3', 7, 5);
geo3.aire();
geo3.taille();
```


Valeurs primitives et objets prédéfinis

- Le JavaScript possède deux grandes catégories de types de données : les valeurs ***primitives*** et les ***objets***.
- On appelle valeur primitive en JavaScript une valeur qui n'est pas un objet et qui ne peut pas être modifiée.
- les valeurs primitives sont passées et comparées par valeur tandis que les objets sont passés et comparés par référence.
- Si deux valeurs primitives ont la même valeur, elles vont être considérées égales.
- Si deux objets définissent les mêmes propriétés et méthodes avec les mêmes valeurs, ils ne vont pas être égaux. Pour que deux objets soient égaux, il faut que les deux fassent référence aux mêmes membres.

Valeurs primitives et objets prédéfinis

- Chaque type de valeur primitive, à l'exception de ***null*** et de ***undfenied***, possède un équivalent objet prédéfini en JavaScript.
- JavaScript possède quatre objets natifs ***String***, ***Number***, ***Boolean*** et ***Symbol*** qui contiennent des propriétés et des méthodes.

```
let s1="Bonjour";
let s2= new String("Bonjour");
console.log(typeof s1); // string
console.log(typeof s2); // object
```


L'objet String

- Propriété
  - length
  - prototype
- Methodes
  - includes()
  - startsWith(), endsWith()
  - substring(), slice()
  - indexOf(), lastIndexOf()
  - replace()
  - trim()
  - toLowerCase(), toApperCase()
  - match, matchAll() et search()

L'objet Number

- Propriété
  - MIN_VALUE, MAX_VALUE
  - MIN_SAFE_INTEGER, MAX_SAFE_INTEGER
  - NEGATIVE_INFINITY, POSITIVE_INFINITY
  - NaN
- Methodes
  - isFinite(), isInteger(), isNaN()
  - isSafeInteger()
  - parseInt(), parseFloat()
  - toFixed(), toString()

L'objet Math

- Propriété
  - Math.E, Math.pi, Math.SQRT2
- Methodes
  - floor(), ceil(), round() et trunc()
  - random()
  - min(), max()
  - abs()
  - cos(), sin(), tan(), acos(), asin() et atan()
  - exp() et log()

Les tableaux: Array

```
let prenoms = ['yasser', 'adnane', 'anas', 'hiba'];
let ages = [29, 27, 29, 30];
let produits = ['Livre', 20, 'Ordinateur', 5, ['Magnets', 100]];
console.log(prenom[0])
for(let valeur of prenoms){
    console.log(valeur);
}
```


Les tableaux: Array

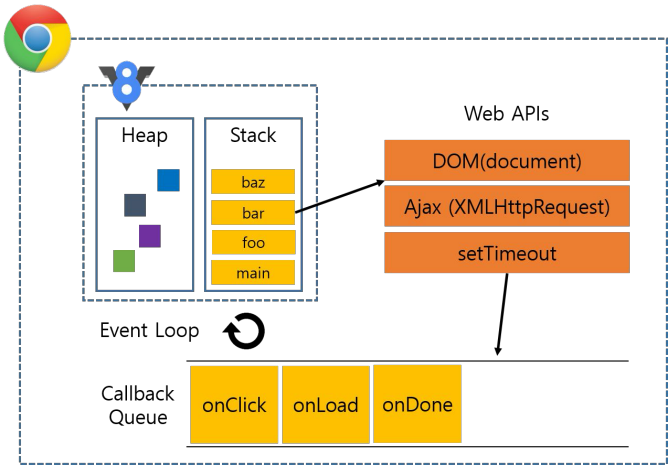
```
let personne = {
  'prenom' : 'yasser',
  'age' : 29,
  'sport' : 'trail',
  'cours' : ['HTML', ' CSS', ' JavaScript']
};
for(let p in personne){
    console.log(personne[p]);
}
```


Les tableaux: Array

- Propriété
    - Length
    - prototype
- Methodes
    - Push(), pop()
    - unshift() et shift()
    - splice()
    - slice(), join()
    - concat()
    - includes()

L’objet Date

- Plusieur facon pour créer un objet Date:
    - let date1 = Date();
    - let date2 = Date.now();
    - let date3 = new Date();
    - let date4 = new Date('March 23, 2019 20:00:00');
    - let date5 = new Date(1553466000000);
    - let date6 = new Date(2019, 0, 25, 12, 30);
- L’objet date possede plusieur méthodes:
    - getDay() , getDate(), getMonth(), getFullYear(), getHours(), getMinutes() getSeconds() getMilliseconds()



Javascript

BOM: Browser Object Model

Introduction

En javascript on trouve deux type d'API

- API intégrées aux navigateurs web. Ces API sont rassemblées dans ce qu'on appelle le BOM (Browser Object Model):
  - l'API **DOM** (Document Object Model): Manipuler le HTML et le CSS d'une page,
  - l'API **Geolocation** : Définir des données de géolocalisation
  - l'API **Canvas:** Dessiner et manipuler des graphiques dans une page ...
- Les API externes, proposées par certains logiciels ou sites:
  - d'API Google Map: Intégrer avec des cartes dans nos pages web
  - l'API Twitter: Afficher une liste de tweets sur un site
  - l'API YouTube qui permet d'intégrer des vidéos sur un site...

BOM

- Le BOM est une super API composée de plusieurs API dont certaines sont composées de plusieurs API et etc.
- A la base du BOM, on trouve l'interface Window qui représente une fenêtre de navigateur contenant un document.
- les objets globaux, variables globales et fonctions globales appartiennent automatiquement à cet objet.
- Window est un objet dit « implicite » : généralement on n'a pas besoin de le mentionner pour utiliser ses méthodes et ses propriétés.

BOM

Les objets suivants appartiennent au BOM et sont tous des enfants de **Window** :

- **Navigator** : représente l'état et l'identité du navigateur ;
- **History** : permet de manipuler l'historique de navigation du navigateur
- **Location:** fournit des informations relatives à l'URL de la page courante ;
- **Screen:** permet d'examiner les propriétés de l'écran qui affiche la fenêtre courante ;
- **Document:** le **DOM** dans son ensemble que nous étudierons en détail dans la suite.

Window: propriétés

L'objet Window possède de nombreuses propriétés:

- ***document***, ***navigator***, ***location*** qui retournent des références aux objets du même nom.
- ***innerHeight***, **innerWidth**: retournent la hauteur et la largeur de la partie visible de la fenêtre de navigation (la partie dans laquelle le code est rendu).
- ***outerHeight*** et ***outerWidth***: retourner la hauteur et la largeur de la fenêtre du navigateur en comptant les options du navigateur.
- ….

Window: Méthodes

L'objet Window possède de nombreuses méthodes permettant :

- afficher des boîtes de dialogue
- ouvrir une fenêtre
- Fermer une fenêtre
- redimensionner une fenêtre
- déplacer une fenêtre

Window: Méthodes

- Var f = **open**("url", "id","width=120, height=200", ...)
- Var rep=**prompt**("message")
- **alert**("message")
- Var rep=**confirm**("message")

Navigator

- L'objet Navigator donne des informations sur le « user agent » (navigateur et préférences enregistrées (langue, etc.))
- Ces informations proviennent de l'utilisateur et ne sont donc totalement fiables.
- Il faut demander une autorisation à l'utilisateur avant de récupérer certaines de ces informations.
- L'objet Navigator est accessible en utilisant la propriété navigator de Window: var nav=Window.navigator

Navigator: propriétés

Les propriétés les plus intéressantes sont :

- **language** : la langue définie dans le navigateur ;
- **geolocation** : un objet Geolocation qui peut être utilisé pour définir la localisation de l'utilisateur ;
- **cookieEnabled** : les cookies sont autorisés ou non;
- **platform** : retourne la plateforme utilisée par le navigateur.

Navigator: Méthodes

Les propriétés les plus intéressantes sont :

- **language** : retourne la langue définie dans le navigateur ;
- **geolocation** : retourne un objet Geolocation qui peut être utilisé pour définir la localisation de l'utilisateur ;
- **cookieEnabled** : retourne si les cookies sont autorisés ou non;
- **platform** : retourne la plateforme utilisée par le navigateur.
- **appName** :retourne le nom du navigateur ;
- **appVersion** :retourne la version du navigateur utilisée ;
- **userAgent** : retourne l'en-tête du fichier user-agent envoyé par le navigateur.

Geolocation

Geolocation nous permet de géolocaliser d'un appareil. Cet objet est une propriété de l'objet Navigator. Il met trois méthodes à notre disposition qui ne sont disponibles que dans des contextes sécurisés (HTTPS) pour des raisons de sécurité :

- getCurrentPosition()
- watchPosition()
- clearWatch()

Geolocation

- **getCurrentPosition()** retourne un objet Position qui permet d'obtenir la position actuelle de l'appareil en retournant un objet Position ;
- **watchPosition()** permet de définir une fonction de retour qui sera appelée automatiquement dès que la position de l'appareil change.
- **clearWatch()** est utilisée pour supprimer la fonction de retour passée à watchPosition().

History

- L'objet History permet de manipuler l'historique du navigateur des visiteurs pour la session courante (ex: charger page précédente).
- La propriété history de Window permet d'obtenir une référence à l'objet History.
- L'interface History implémente plusieurs propriétés et méthodes:
  - Length
  - go(n)
  - back() ⇔ go(-1)
  - Forward ⇔ go(1)

Location

- L'interface Location fournit des informations relatives à l'URL d'une page.
- On peut accéder à Location à partir des interfaces **Window** ou **Document**, en utilisant leur propriété location.
- L'objet Location donne accès à plusieurs propriétés et 4 méthodes.
  - Search; pathname; href; hostname; port; protocole …
  - assign("url")
  - replace("url")
  - reload()

Screen

- L'objet Screen donne accès à des informations concernant l'écran du, comme la taille et la résolution de l'écran.
- Ces informations sont utilisées pour proposer différents affichages à différents visiteurs par exemple.
- On peut récupérer un objet **Screen** en utilisant la propriété screen de **Window**

Var s=window.screen

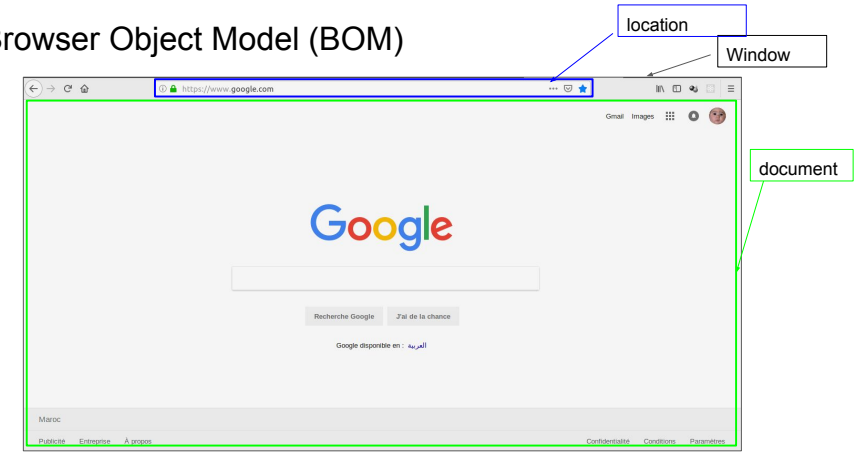
Screen:propriétés

- ***width*** : retourne la largeur totale de l'écran ;
- ***availWidth*** : retourne la largeur de l'écran moins celle de la barre de tâches ;
- ***height*** : retourne la hauteur totale de l'écran ;
- ***availHeight*** : retourne la hauteur de l'écran moins celle de la barre de tâches ;
- ***colorDepth*** : retourne la profondeur de la palette de couleur de l'écran en bits ;
- ***pixelDepth*** : retourne la résolution de l'écran en bits par pixel.

Javascript

DOM: Document Object Model

Browser Object Model (BOM)



Browser Object Model (BOM)

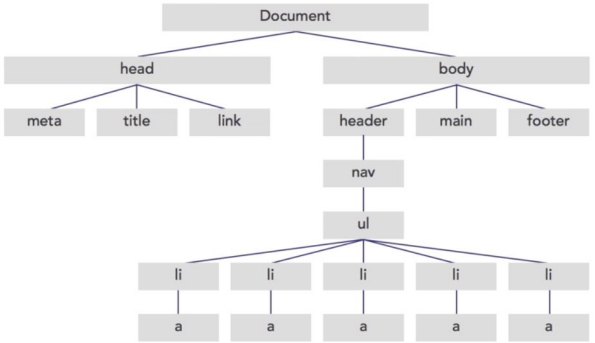
Window est l'objet du plus haut niveau dans le BOM et possède un ensemble de propriétés et méthode pour interagir avec le Browser

- window.innerWidth
- window.open()
- window.location
- Window.**document**
- ....

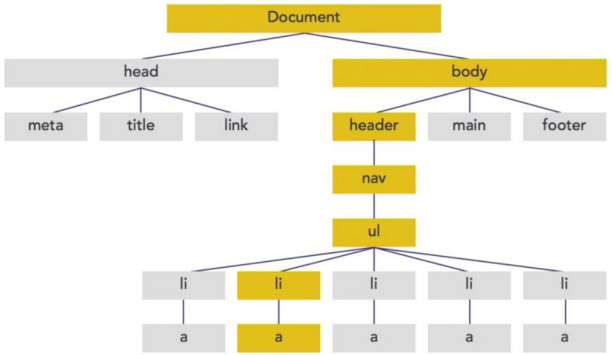
Le Document Object Model (DOM)

- Le DOM est une API qui s'utilise avec les documents HTML, et qui va nous permettre, via le JavaScript, d'accéder au code HTML d'un document.
- C'est grâce au DOM que nous allons pouvoir:
    - ***modifier des éléments HTML (afficher ou masquer un par exemple),***
    - ***ajouter des éléments HTML***
    - ***déplacer des éléments HTML***
    - ***Supprimer des éléments HTML***

Le Document Object Model (DOM)



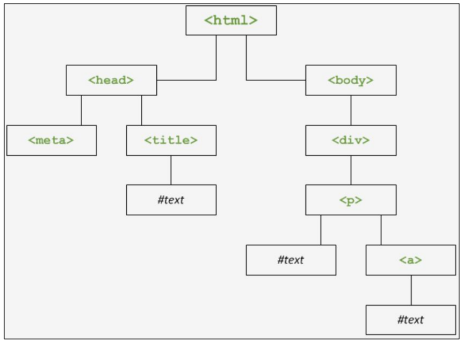
Le Document Object Model (DOM)



Le Document Object Model (DOM)

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
</head>

<body>
    <div>
        <p>Un peu de texte <a>et un lien</a></p>
    </div>
</body>
</html>
```



Accéder aux éléments

- document.body
- Document.title
- document.URL
- document.getElementById(“#id”),
- document.getElementsByTagName(“tagName”)
- document.getElementsByClassName(“.class”)
- document.getElementsByName(“name”)
- ...

Accéder aux éléments

document.querySelector(“css selector”)

=> Retourne Le premier élément qui vérifie le sélecteur css spécifier

document.querySelectorAll(“css selector”)

=> Retourne Tous les éléments vérifiant le sélecteur CSS

Ces deux méthodes sont les plus utilisées

Accéder aux éléments

1. querySelector()
2. querySelectorAll()

```
<div id="menu">

    <div class="item">
        <span>Élément 1</span>
        <span>Élément 2</span>
    </div>

    <div class="publicite">
        <span>Élément 3</span>
        <span>Élément 4</span>
    </div>

</div>

<div id="contenu">
    <span>Introduction au contenu de la page...</span>
</div>
```

```
var query = document.querySelector('#menu .item span'),
    queryAll = document.querySelectorAll('#menu .item span');

alert(query.innerHTML); // Affiche : "Élément 1"

alert(queryAll.length); // Affiche : "2"
alert(queryAll[0].innerHTML + ' - ' + queryAll[1].innerHTML); // Affiche : "Élément 1
- Élément 2"
```

Accéder aux attributs d’un éléments

Les attributs des élément HTML peuvent êtres accède en lecture/écriture ou en lecture seule

1. element.atributes
2. element.innerHTML
3. element.outerHTML
4. element.clientHight
5. element.className
6. element.classList
7. element.id
8. ...

Accéder aux attributs d’un éléments

Pour modifier les attributs en lecture seul on utilise de methodes:

1. element.classList.add(“une nouvelle classe”)
2. element.classList.remove(“une classe existante”)
3. element.classList.contains(“une classe”)
4. element.haseAttribute(attribut)
5. element.getAttribute(attribut)
6. element.setAttribute(attibut, value) // modifier ou ajouter un attribut
7. element.removeAttribute(attribut)

Exp: ***document.querySellector(“a”).setAttribut(“target”,”_blank”)***

Accéder aux éléments

getAttribute() et setAttribute()

```
<body>
    <a id="myLink" href="http://www.un_lien_quelconque.com">Un lien modifié
dynamiquement</a>

    <script>
        var link = document.getElementById('myLink');
        var href = link.getAttribute('href'); // On récupère l'attribut « href »

        alert(href);

        link.setAttribute('href', 'http://www.siteduzero.com'); // On édite l'attribut
« href »
    </script>
</body>
```

Accéder aux éléments (Elenment.className)

```
<head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
    <style>
        .blue {
            background: blue;
            color: white;
        }
    </style>
</head>
<body>
    <div id="myColoredDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>
    <script>
        document.getElementById('myColoredDiv').className = 'blue';
    </script>
</body>
```

Accéder aux éléments (Elenment.classList)

```
var div = document.querySelector('div');
// Ajoute une nouvelle classe
div.classList.add('new-class');
// Retire une classe
div.classList.remove('new-class');
// Retire une classe si elle est présente ou bien l'ajoute si elle est absente
div.classList.toggle('toggled-class');
// Indique si une classe est présente ou non
if (div.classList.contains('old-class')) {
    alert('La classe .old-class est présente !');
}
// Parcourt et affiche les classes CSS
var result = '';
for (var i = 0; i < div.classList.length; i++) {
    result += '.' + div.classList[i] + '\n';
}
alert(result);
```

Ajout d’un élément au DOM

1. Creer l’element <= document.createElement()
2. Créer le noeud texte de cet élément <= document.createTextNode()
3. Ajouter le noeud texte à l'élément <= document.appendChild()
4. Ajouter l'élément au DOM <= document.appendChild()

Exercice:

Ajouter un élément <***caption>....</caption>*** a l'élément <***figure ..> …</figure>***

```
9       <figure class="mafig">
10          <img class="monImg" src="img.jpeg" alt="un texte
            ">
11      </figure>
```

Ajout d’un élément au DOM

```
1  const fig=document.querySelector(".mafig");
2  const img=fig.querySelector(".monImg");
3  var altTxt=img.getAttribute("alt");
4  var capElmt=document.createElement("figcaption");
5  var capTxt=document.createTextNode(altTxt);
6  capElmt.appendChild(capTxt);
7  fig.appendChild(capElmt);
8  console.log(fig);
```

Une nouvelle methode : .append()

```
1  const fig=document.querySelector(".mafig");
2  const img=fig.querySelector(".monImg");
3  var altTxt=img.getAttribute("alt");
4  var capElmt=document.createElement("figcaption");
5  capElmt.append(altTxt);
6  fig.append(capElmt);
```

Style CSS Inline

Avec l’attribut style on peut ajouter n’import quel propriété CSS à n'importe quel élément.

- ***Element.style;*** => **uniquement** le **Inline** CSS de l'élément {attribut:“value”, …. } mais pas les autre style définie dans des fichier css au dans le head.
- ***Element.style.color=”blue”;***
- ***Element.style.backgroundColor=”yellow”;*** //**backgroundColor** pas **background-color**
- ***Element.style.cssText=”color: blue; backgound-color: yellow; …”***
- ***Element.style.setAttribute(“style”, ”color: blue; backgound-color: yellow; …”);***

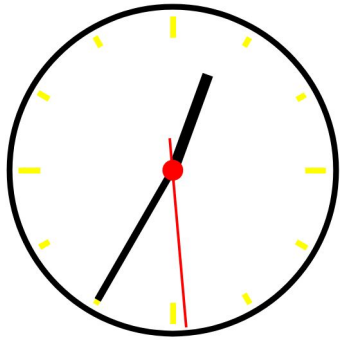
Style CSS Inline

Les style CSS inline remplace les style definie dans les feuille de style.

Dans la plupart des cas il vaut mieu définir des règles CSS et gérer les class avec javascripte

Exercice

1. Créer une horloge en utilisant une image SVG à l'aide d'un éditeur de graphiques vectoriel.
2. Créer un fichier css pour modifier son apparence.
3. Écrire un code javascript pour animer cette horloge.

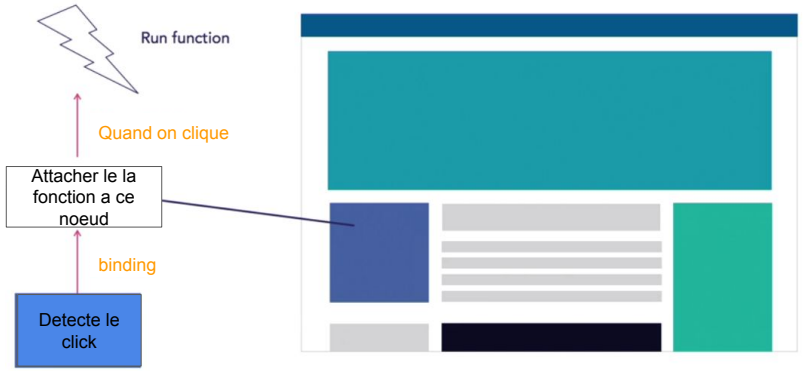


Javascript

Les événements

Les événements

Les événements



Les événements

load	Click	Keydown	Change
error	Dblclick	Keyup	Input
resize	Mouseover	Keypress	Select
online	Mouseout	Focus	Reset
Offline	Mousedown	Blur	Submet
Scrol	Mouseup	...	...
...	Mousemove		

https://developer.mozilla.org/en-US/docs/Web/Events

Les événements

```js
function eventclbk(e){
    e.preventDefault();// annule le comportement par defaut
    // autres traitement si l evennement e surgit
}

//element.onevent= eventclbk;
element.onclick = eventclbk;
```

Les événements

event.js

```js
const elmt=document.querySelector(".normal");
function eventclbk(e){
    console.log("cliked");
    elmt.classList.toggle("normal");
}
elmt.onclick = eventclbk;
```

event.html

```html
    <style type="text/css">
        .normal{
            background-color: yellow;
        }
        div{
            background-color: blue;
            text-align: center;
            font-size: 20px;
            height: 40px;
        }
    </style>
</head>
<body>
    <div class="normal">click me</div>
</body>
```

Les événements

```js
const elmt=document.querySelector(".normal");
function eventclbk1(e){
    elmt.classList.toggle("normal");
}
function eventclbk2(e){
    console.log("Div clicked");
}
elmt.onclick = eventclbk1;
elmt.onclick = eventclbk2;
```

Problème: uniquement la fonction eventclbk2 sera exécutée

Les événements

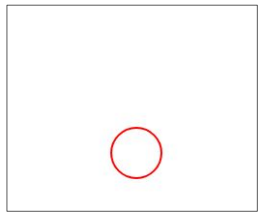
element.addEventListener("event", function_clbck, [true/false]);

```
const elmt=document.querySelector(".normal");
function eventclbk1(e){
    elmt.classList.toggle("normal");
}
function eventclbk2(e){
    console.log("Div clicked");
}
elmt.addEventListener("click",eventclbk1,false);
elmt.addEventListener("click",eventclbk2,false);
```

Les événements

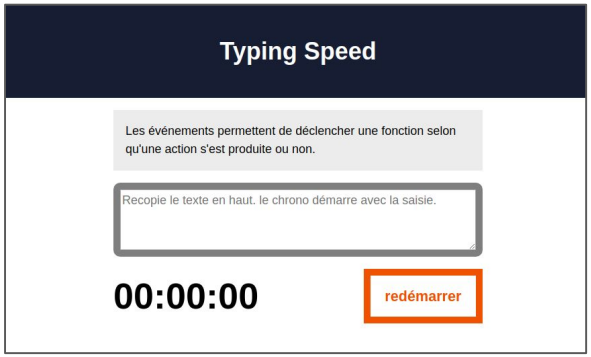
Exercice:

1. Le cercle se déplace pour garder son centre en symétrie avec le curseur de la souris.
2. Le cercle change de couleur lorsqu'il touche curseur.
3. Utiliser ***.clientX*** et ***.clientY*** de l'objet ***event***



Les événements

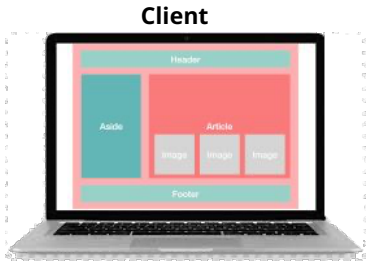
Exercice:



Javascript

Ajax

Principe

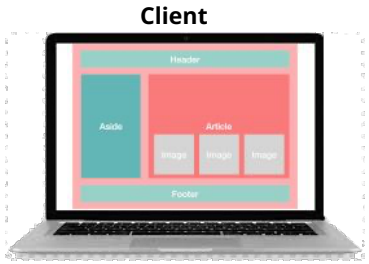


Javascript

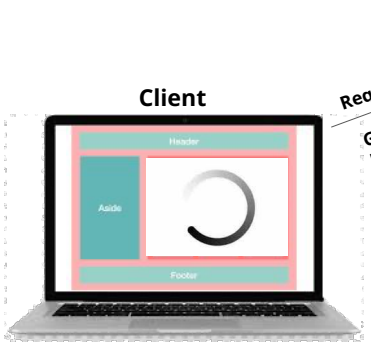
CSS

HTML

Principe



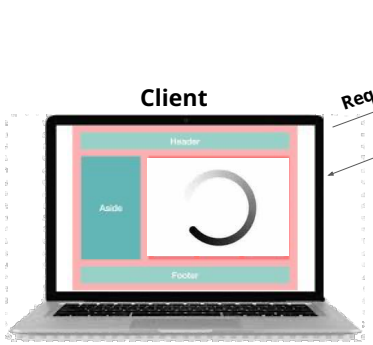
Principe



Requête

GET
POST
UPDATE
PATCH
DELETE
....

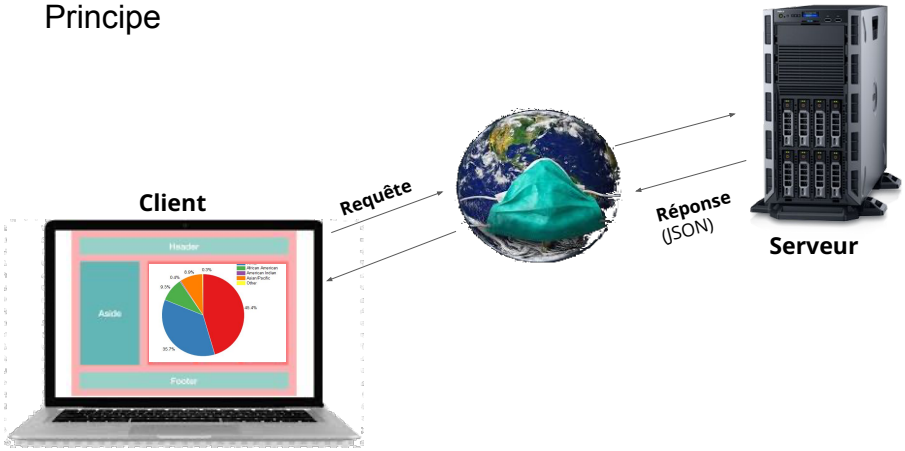
Principe



Requête

Réponse
(JSON)

Principe



XMLHttpRequest

Méthodes

- **new XMLHttpRequest()** : Créer un objet XMLHttpRequest
- **open(method,url,async,user,psw)** : Définir les paramètres de la requête
- **send()** : Envoyer la requête
- **abort()** : Annuler la requête

Propriétés

- **readyState :** Etat actuel de la requête (0, 1, 2, 3, 4)
- **Onreadystatechange :** La fonction à invoquer au changement de **readyState**
- **Status :** L'état du retour du serveur (200, 403, 404 …)
- **Response:** Les données renvoyées par le serveur

XMLHttpRequest (GET)

```
let req=new XMLHttpRequest();
req.open("GET","https://jsonplaceholder.typicode.com/todos");
req.onreadystatechange=()=>{
    let resp
    console.log(req.readyState);
    if(req.readyState==4 && req.status==200){
        resp=JSON.parse(req.responseText)
    }
    console.log(resp);
}
req.send()
```

XMLHttpRequest (POST)

```
let httpReq = new XMLHttpRequest();
let url = "localhost/api/user";
let params = JSON.stringify({ name: "Gounane", age:21});
httpReq.open("POST", url, true);
httpReq.setRequestHeader("Content-type", "application/json; charset=utf-8");
httpReq.setRequestHeader("Content-length", params.length);
httpReq.setRequestHeader("Connection", "close");
httpReq.onreadystatechange = function() {
    if(http.readyState == 4 && httpReq.status == 200) {
        alert(httpReq.responseText);
    }
}
httpReq.send(params);
```

Javascript

ES6: ECMA Script 6

ES6

Javascript introduit par Netscape

Puis ECMA International pour la standardisation

ES6 == EcmaScript 6 (2015) ES7 (2016) ES8(2017)

⇒ Le code javascript devient plus simple

Mais pas supporter par la plupart des navigateur

==>Babel

ES6

Javascript

ES6

```
var a=function(x,y){
    return x+y;
}
```

```
const a=(x,y)=>x+y;
```

ES6: destructuring

Javascript

ES6

```
const etudiant={
    nom: "Baddi",
    prenom: "Ahmed",
    age: 24
}
const nom=etudiant.nom;
const prenom=etudiant.prenom;
const age=etudiant.age;
```

```
const etudiant={
    nom: "Baddi",
    prenom: "Ahmed",
    age: 24
}

const {nom, prenom, age}=etudiant;
```

ES6: propriétés des objets

```
const a="nom";
const b="pre";
const etudiant={
    [a]: "Baddi",
    [b+a]: "Ahmed",
    age: 24
}
console.log(etudiant.prenom);
```

ES6: propriétés des objets

```
const nom="Baddi";
const prenom="Ahmed";
const age=24;
const etudiant={
    nom: nom,
    prenom: prenom,
    age: age
}
//ES6
const etudiant={
    nom,
    prenom,
    age
}
```

ES6: template Strings

```
const nom="Baddi";
const prenom="Ahmed";
const age=24;
let s= "Mon nom est"+nom+"j'ai "+age+"
    ans";
//ES6
let s=`Mon nom est: ${nom} j'ai ${age}
    ans`;
```

ES6: Arguments par defaut

```
function cal(a=0,b=1){
    return a/b;
}

calc(); // 0
calc(4);// 4
calc(4,8);// 0.5
```

ES6: Arrow function

```
function somme(a,b){
    return a+b;
}
//ES6
const somme=(a,b)=> {
    return a+b
}

const somme=(a,b)=> a+b;
```

ES6: Closures

```
function f1(){
    var a="Bonjour";
    function f2(){
        console.log(a);
    }
    return f2;
}
var f=f1();
f();
```

ES6: Closures

```
function f1(){
    var a="Bonjour";
    function f2(){
        console.log(a);
    }
    return f2;
}
var f=f1();
f();
```

```
//ES6
const f1=()=>{
    const a="Bonjour";
    const f2=()=>{
        console.log(a);
    }
}
var f=f1();
f();
```

ES6: Currying

Transformer une fonction a plusieurs variable en plusieur fonctions a une seule variable.

```
const prod1=(a,b)=> a*b;
prod1(3,4); // 12

const prod2=(a)=>(b)=>a*b;
prod2(3);// ?
prod2(3)(4);// ?
```

ES6: Currying

Transformer une fonction a plusieurs variable en plusieur fonctions a une seule variable.

```
const prod1=(a,b)=> a*b;
prod1(3,4); // 12

const prod2=(a)=>(b)=>a*b;
prod2(3);// ?
prod2(3)(4);// ?
```

ES6: Compose

```
const comp = (f,g) => (a) => f(g(a));
let somme = (n) => n+1;
comp(somme,somme)(3);// ?
```

Array

```
const a=[1,4,3,8];
let b=a.forEach((n)=>{
    n*2;
});
console.log(b);// undefined
let b=a.map((n)=> n*2);
console.log(b);// [2,8,6,16]

let c=a.filter((n)=> n%2 === 0);
console.log(c);// [4,8]

let d=a.reduce((s,n)=> s+n,2);
console.log(d);// 18
```

Objets : references

```
let a={val: 3};
let b=a;
let c={val: 3};
console.log(a===b);//true
console.log(a===c);//false
a.val=2;
console.log(a.val);//2
console.log(b.val);//2
console.log(c.val);//3
```

Objets: contexte

```
const obj={
    a:function(){
        console.log(this);
    }
}
obj.a();//obj
console.log(this);// window
function a(){
    console.log(this);
}
a()// window
```

Objets: instanciation

```
class Homme{
    constructor(nam, age){
        this.nom=nom;
        this.age=age;
    }
    hi(){
        console.log(`Je suis ${this.nom}`);
    }
}
let h=new Homme("yakoubi",22);
```

Objets: instanciation

```
class Etudiant extends Homme{
    constructor(nam, age, note){
        super(nom,age);
        this.note=note;
    }
    getNote(){
        console.log(`Note: ${this.note}`);
    }
}
let h=new Homme("yakoubi",22,17);
```