# Chapitre 1 : Intro

## Que permet le Javascript ?

* Il permet d’écrire des scripts clients
  + **Script** : Programme destiné à être exécuté
  + **Client** : Tout se déroule sur l’ordinateur de l’internaute

## Les caractéristiques de JS

|  |  |
| --- | --- |
| 1. Un langage de programmation | 1. Impératif |
| 1. Orienté objet | 1. Evènementiel |
| 1. Faiblement / non typé   ***Dynamiquement Typé*** | 1. Où les fonctions sont des objets de premier ordre |
| 1. Qui évolue (ES6) | 1. Sensible à la casse |

*Cf : p17-22 du cours*

## Les commandes de bases

* confirm(txt)
  + Demande une confirmation et renvoi True ou False dépendant de si l’utilisateur a utilisé Enter ou Cancel
* prompt(txt [, valDefaut])
  + Demande une entré (en STRING) à l’utilisateur.
  + Peut éventuellement renvoyer une valeur par défaut
* document.write(txt)
  + Ecrit directement dans le document HTML
* console.log(txt)
  + Ecrit dans la console du navigateur
* alert(txt)
  + Ecrit dans une fenêtre pop-up

## Les manières d’insérer du script

### 1.4.1 : Code interne

* Dans des balises script
  + Il peut y avoir plusieurs balises
  + Elles peuvent se trouver n’importe où
  + <script>...</script>

### 1.4.2 : Code inline

* <button onclick= "codeJS">Cliquez moi !</button>
* <a href= "javascript :*codeJS*">text</a>

### 1.4.3 : code externe

* <script src= "nom\_fichier\_js"></script>
* Exécution synchrone (immédiate)

# Chapitre 2 : Variables et valeurs

## Les types primitifs

|  |  |
| --- | --- |
| Number | String |
| Boolean | Function |
| Object | Undefined |

### 2.1.1 : Les nombres

* Valeurs acceptées :

|  |  |  |
| --- | --- | --- |
| Entiers | 42  -17 | |
| Réels | 342.17 | |
| Autres bases | 037 (octal)  0x3BFF (hexa)  0b011101 (binaire) | |
| Valeurs spéciales | Infinity | 5 / 0 |
| -Infinity |  |
| Nan | 100 / "pas un chiffre" |

* Opérations sur les nombres :

|  |  |  |
| --- | --- | --- |
| Opérateurs usuels | + - \* / % \*\*(exposant) | |
| Opérateurs unaires | * ++ -- | |
| Opérations de comparaison | Plus petit [ou égal] | < <= |
| Plus grand [ou égal] | > >= |
| Est égal | == |
| Est strictement égal  (vérifie le type, ne fait aucune conversion implicite) | === |
| Opérateurs bit à bit | & | ~ ^ << >> >>> | |

### 2.1.2 : Les booléens

* Littéraux pour les booléens :
  + true / false
* Opérations sur les booléens
  + ! && ||
  + Ternaire 🡪 ? :

### 2.1.3 : Les strings

* Encadré par des guillemets ou des apostrophes
  + "pomme", 'Clément'
* Caractères échappés :
  + \n \t \' \" \\
* Les Gabarits / template literals:
  + Encadré par des backtips (`...`)
  + Peut être réparti sur plusieurs lignes
  + Permet d'inclure des expressions régulière
    - ${*nom\_variables*}
* Operations standards

|  |  |
| --- | --- |
| Concaténation  + | "Bonjour" + *nom*   * "Bonjour Clément" |
| Longueur | Nom.length |
| Extraction | Nom[0]  Nom.charAt(0) |

* Quelques principes :
  + Les chaines de caractères sont immuables
    - On ne peut changer juste une valeur du tableau
    - On ne peut pas non plus changer sa valeur *length*
* Opérations prédéfinies

|  |  |  |
| --- | --- | --- |
| Test | s.startsWith(début) | Renvoie vrai ou faux |
| s.endsWith(fin) |
| s.includes(partie) |
| Extraction | s.substr(début, longueur) | Renvoie une nouvelle chaine de caractère |
| s.substring(début, fin) |
| Recherche | s.indexOf(partie) | Renvoie l'index correspondant. -1 si introuvé |
| s.lastIndexOf(partie) |
| Décomposition | s.split(separateur) | Renvoie un tableau décomposant les éléments |

* Javascript est typé dynamiquement
  + Le contenu d’une variable peut changer de type au cours de l’éxécution
  + Même si à chaque moment, une variable à un type précis
  + Ce qui engendre :
    - Aucun type à la déclaration
    - De nombreuses conversions implicites sont effectuées
    - Certaines erreurs logique ne sont pas facilement détectable
* Donner le type d’une variable à un moment précis :
  + typeof ...
    - typeof 12 🡪 "number"
* Les réponses que peuvent donner typeof :
  + number
    - 12
    - parseint("21")
    - NaN
    - -78
    - 342.17
    - 037 (octal) 0x3BFF(hexa) 0110(binaire)
    - Infinity (5/0)
    - -Infinity
  + string
    - "coucou"
  + object
    - new Date()
    - null
  + function
    - function test(){}

## Les variables

### 2.2.1 : Le scope

* Par définition, un scope représente une portion de code (un bloc) qui est comprise entre 2 accolades.
* Exemples :
  + une fonction
  + Une condition
  + une boucle
  + ...
* Une variable n'est pas accessible en dehors de son scope

### 2.2.2 : Les 4 déclarations de variables

1. Pas de déclaration
   1. msg = "hello";
   2. Déclaré au niveau global
2. let
   1. let msg = "hello";
   2. Déclaré localement suivant les règles du scope
3. const
   1. const msg = "hello";
   2. Déclaré localement suivant les règles du scope
   3. Doit obligatoirement être initialisé
   4. Ne peut être modifié par la suite
4. var
   1. var msg = "hello"
   2. Déclaré au niveau de la fonction englobante du scope
   3. Peut être redéclaré.

## Le hoisting

* Dans chaque scope fonctionnel :
  + les délcarations var et function sont hissées vers le début.
  + Pour let et const :
    - Hissage vers le début du bloc
    - Mais on ne peut utiliser la variable avant sa déclaration
      * On parle alors de TDE (Temporal Dead Zone)
  + /!\ Cependant, seules les déclarations sont hissés /!\

## Les conversions explicites

* Tableau des conversions :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A transformer  En ça | Nombre | Booléen | String | Undefined | Object |
| Nombre |  | true / false  1 / 0 | "42" 🡪 42  Vide ou blancs 🡪 0  "mot" 🡪 NaN | NaN | null 🡪 0 |
| Booléen | 0, NaN / autre  False / true |  | Vide "" / "autre…"  false / true | false | null / autre  false / true |
| String | NaN 🡪 "NaN"  Infinity 🡪 "Infinity"  Autre 🡪 "chiffre" | "true", "false" |  | "undefined" | "null" ou via toString() |

* A RETENIR CONCERNANT LE TABLEAU :
  + Les valeurs falsy :
    - false, 0, null, undefined, "", NaN
  + 2 autres méthodes pour convertir en nombre :
    - parseFloat(s)
    - parseInt(s, base)
    - Dans les 2 cas, on s'arrête au premier caractère illégal

## Les conversions implicites

|  |  |
| --- | --- |
| Expression | Priorité |
| + | S'il y a un string : concaténation  Sinon : addition |
| == | Si valeurs de même types :  Comparaison simple. Sauf NaN qui n'est pas égale à lui-même  S'il y a *null* ou *undefined* : alors  null == undefined 🡪 true  sinon 🡪 false  Sinon, conversion en nombre |
| === | Aucune conversion. Donne automatiquement faux si les types sont différents ! |
| < | Si 2 strings, alors c'est l'ordre lexicographique  Sinon, conversion en nombre et comparaison |

## Les opérateurs

### 2.6.1 : typeof()

* Indique le type de son argument (*'undefinded'* si référence inexistante)
  + if (typeof x === 'undefined')

### 2.6.1 : void

* Evalue son argument mais ne renvoie pas de valeur.
  + void(document.form.submit())

### 2.6.3 : Operateur A || B

* Si A converti en booléen donne *true* , alors le résultat est A.
  + Sinon, le résultat est B.
  + On jouera donc ici avec les valeurs falsy
  + Permet l'utilisation de "valeur par défaut"
    - Car peut considérer l'intégrité d'une valeur.
    - /!\ ATTENTION tout de même car dans certains cas, les valeurs falsy sont des valeurs possibles et traitable dans notre algorithme.

### 2.6.4 : Operateur A && B

* Si A converti en bollén donne faux, alors le résultat est A.
  + Sinon, le résultat est B.
  + Permet d'écrire des conditions
    - if ( codeExiste && codeAExecuter )
    - Ici, si le code n'existe pas, rien ne sera exécuter.

# Chapitre 3 : Fonctions, tableaux, objets

## Les méthodes de définition des fonctions

### 3.1.1 - Méthode 1 : Déclaration Standard

function affiche(x) {

console.log("La valeur est : " + x);

}

* Syntaxe similaire à celle du C.
* Déclaration hoisté !

### 3.1.2 – Méthode 2 : expression fonctionnelle

let affiche = function (x) {

console.log("La valeur est : " + x);

};

* Il s'agit d'une déclaration de variable.
* Avec var, la déclaration est hoistée, mais pas l'initialisation !
* Avec let/const, il y a le principe de TDZ.
* Le nom de la fonction est facultatif

### 3.1.3 – Méthode 3 : via le constructeur Function

let affiche = new Function ("x",

'console.log("La valeur est : " + x);');

* Une fonction est ici un objet !
* Argument du constructeur :
  + Les paramètres formels,
  + En dernier, le code de la fonction
* Avec var, la déclaration est hoistée
* Avec let/const, il y a le principe de TDZ.
* A EVITER

## Fonctions prédéfinies

* eval()
  + Retourne la valeur d'un code Javascript
* isNaN, isFinite

## Les tableaux

### 3.3.1 : Caractéristiques

* Hétérogènes
* Dynamiques
* Peuvent contenir des trous
* Taille modifiable (suppression de cases ou ajout de cases vides implicites);

### 3.3.2 : Créer un tableau

Méthode 1 : Utiliser un littéral :

let tab = [1,2,3];

let jours = ["Lundi","Mardi","Mercredi","Jeudi","Vendredi"];

let tab2D = [[1,2],[3,4,5]];

let tabFunction = [1, true, function () {return 42; }];

Méthode 2 : Utiliser un constructeur :

let tab = new Array(1,2,3);

### 3.3.3 : Accéder aux éléments

* Via les indices
* /!\ ATTENTION /!\
  + Les accès hors bornes :
  + Renvoie *undefined* en lecture
  + Augmente la taille du tableau en lecture
* Le tableau doit être déclaré avant d'y écrire quelque chose

### 3.3.4 : Parcours d'un tableau

* Boucle usuelle
  + for, while
  + Avec un indice qui parcours le tableau
* Boucle for\_in
  + for (let indice in tab) { console.log("tab[indice]") };
  + indice prends comme valeur les indices des cases du tableau
  + Les trous sont sautés !
* Boucle for\_of
  + for (let valeur of tab) { console.log(val) };
  + valeur prend le contenu des cases du tableau
  + Les trous ne sont pas sautés mais sont *undefined* !

## Object

### 3.4.1 – Tableau associatif

* Un tableau associatif, (dictionnaire, map) est une structure qui associe des valeurs a certaines clefs.
  + Année["HTML"] = 1;
  + Personne["prénom"] = "Clément";
* Le concept du tableau associatif est une notion logique.

### 3.4.2 – Object en Javascript

* Object = Tableau associatif
  + Clé = nom de la propriété
  + Pour les valeurs :
    - Valeur = valeur de l'attribut
    - Méthode = fonction décrivant la méthode
  + Dynamiques
    - Les attributs/méthodes peuvent être joutés / supprimés / modifiés
* Différences d'approche entre
  + OO par classe :
    - Classe = moule
    - Objet = instance d'une classe
  + OO prototypale de Javascript
    - L'objet est une entité à part entière
    - N'a pas besoin de moule

### 3.4.3 – Créer un objet

* Méthode 1 : Créer un objet vide et lui ajouter des propriétés
  + let obj = {};
    - obj["prenom"] = "Clément" // syntaxe tableau associatif
    - obj.prenom = "Clément" // syntaxe orienté object
  + Pas limité par les règles sur les identificateurs (synatxe, mots réservés…)
    - Mêmes les chiffres sont acceptés (Transformé en chaine de catactère)
    - Pas de syntaxe orienté objet si on utilise des nom qui ne suivent pas les règles des identifiacteurs !
      * personne.4;
      * personne.date de naissance;
      * Ne sont pas acceptés !
      * personne["4"]; // OK
      * personne["date de naissance"]; // OK
* Méthode 2 : Via un littéral pour objet

const h = {

prenom = "homer",

"nom" = "Simpson",

"parle" = function (){ console.log("DOh!"); },

parle () { console.log("DOh!"); }, // Syntaxe courte

}

* + On peut omettre les guillemet sur les noms de propriétés ssi celui-ci ne comporte qu'un mot
  + DÉFINITION DE PROPRIÉTÉ RACCOURCIE EN UTILISANT UNE VARIABLE :

function sommeEtProduit (x, y) {

let somme = x + y;

let produit = x \* y;

return { somme, produit};

} // renvoie l'objet { somme : x+y, produit : x\*y}

* + On peut calculer les noms de propriétés
    - return { [x + y] : somme};

## Utiliser des objets

### 3.5.1 – Accéder aux propriétés

* Lecture :
  + alert(personne.nom);
* Ecriture :
  + personne.nom = "Clément";
* Appel :
  + personne.parle();

### 3.5.2 – Modifier les propriétés

* Du fait que les objets en DYNAMIQUE, on peut :
  + Ajouter des propriétés
  + Modifier des propriétés existantes
  + Supprimer des propriétés
* Un objet déclaré en *const* peut voir ses valeurs modifiés, mais pas ses références !
* Une propriété peut-être déclaré immuable via *readonly*

### 3.5.3 – Test d'égalité

* Lors d'un test d'égalité, on va comparer les références des objet. Donc ce test ne donnera jamais vrai bien que 2 objets aient la même définition et les mêmes attributs !

### 3.5.4 – Opérateur IN

* Indique si l'objet possède ou non la propriété du nom indiqué.

### 3.5.5 – Les boucles

* For\_in
  + Passe en revue les propriétés (les indices sont ici les nom des propriétés)
* For\_of
  + Passe en revue la valeur des propriétés

### 3.5.6 – Nullish coalescing operator

* expr1 ?? expr2
* Renvoie à priori *expr1* sauf si celui-ci vaut null ou undefined

### 3.5.7 – Optionnal chaining operator

* obj.prop?.sprop
* Renvoie :
  + sprop si prop existe
  + undefined si prop vaut null ou undefined

let bart = {

nom : "Bart",

chien : { nom : "PetitPapaNoël" }

};

bart.chien.nom → "PetitPapaNoël"

bart.chat.nom → TypeError Exception

bart.chat → undefined

bart.chat?.nom → undefined

* Pour les tableau associatif

bart.chat["n"+"om"] → TypeError Exception  
bart.chat?.["n"+"om"] → undefined

* Pour les méthodes en vérifiant leurs existences :

bart.aboie() → TypeError Exception

bart.aboie?.() → undefined

# Chapitre 4 : Prototypes et constructeurs

## Les prototypes

### 4.0.1 – Pourquoi un prototype ?

* Lorsque plusieurs objets doivent contenir une même méthode, le mieux et de la définir une seule fois dans un objet à part (le prototype) et de l'"hériter" aux objets concerné. Ainsi, on ne l'enregistre qu'une seule fois.

### 4.0.2 – Comment créer un prototype

* A proprement parlé, un *prototype* est un objet comme un autre. C'est son utilisation qui diffèrent. Pour le créer il faut donc suive les règles de création d'un objet 🡪 3.4.
* Pour associer un prototype, voici la démarche à suivre :
  + let monObjet = Object.creat(prototype);
  + OU ALORS
  + Object.setPrototypeOf(monObjet, prototype);
* Après la création d'un objet via *Object.creat(..)*, on veut pouvoir définir notre objet fraichement créé. Pour ce faire on à la méthode suivante :
  + Object.assign( objet, {nom : "Clément", age : "21", …});
  + Permet d'ajouter plusieurs propriétés à un objets existant en écrasant les anciennes en cas de conflit.

### 4.0.3 : La mécanique des prototypes dans Javascript

* Javascript gèere les prototypes dans la propriété *\_\_proto\_\_* (dunder\_proto) qui est donc une référence vers le prototype.
* Pour bien faire, on ne devrait pas la manipuler normalement.
* C'est grâce à ça que va naitre la chaine des prototypes :
  + Lorsqu'on tente de lire une propriété dans un objet et que JS ne le trouve pas, il va la rechercher dans le prototype, puis dans le prototype du prototype, …
* En revanche, lorsqu'on tente d'y accéder en écriture, c'est une modification direct de l'objet en question.
* Les propriétés d'un objet-prototype sont donc partagés par tous ses descendants
* 2 conséquences en découlent :
  + On peut y placer tous les éléments communs
    - Méthodes
    - Attributs partagés ( attributs static en Java)
  + Si on modifie le prototype, on affecte tous ses héritiers
    - Modification d'une propriété commune
    - Ajout/Suppression d'une propriété commune

### 4.0.4 : Du code pour les prototypes

|  |  |
| --- | --- |
| let nouv = Object.creat(proto) | Pour créer un objet hérité d'un prototype |
| Object.setPrototypeOf(nouv, proto) | Permet de modifier un prototype. (Autant pour mettre un nouveau que pour le changer en cours de route !) |
| Object.getPrototypeOf(nouv) | Pour d'obtenir le prototype actuel d'un objet |
| proto.isPrototypeOf(nouv) | Indique si oui ou non proto est un prototype (DIRECT OU LOINTAIN) d'un objet |
| Object.assign(obj1, obj2) | Pour mettre à jour les propriétés d'un objet |

## Les constructeurs

### 4.1.1 - Descriptions

* Une fonction qui :
  + Crée des objets à partir de ses arguments
  + Va agir sur un objet nouvellement créé avec le mot-clé *this*
  + Utilise le mot-clé *new* afin de créer des objets
  + Commence par une majuscule
  + Attribut automatiquement un prototype aux objets créés
* Exemple :

function Animal (nom, cri) {

this.nom = nom;

this.cri = cri;

}

let chien = new Animal ("Petit Papa Noël", "Wouf");

// \_\_proto\_\_ = Animal.prototype

* Lors de l'appel à *new Animal* :
  + Javascript créé un nouvel objet vide
  + Avec comme prototype l'objet Animal.prototype
  + Puis exécute la fonction avec *this = ce nouvel objet*
* Si on modifie e contenu du Animal.prototype associé à Animal, cela affecte rétroactivement les objet créés par Animal
* Cela existe également pour les constructeurs prédéfinis :
  + String
  + Number
  + Function
  + Array
  + …

### 4.1.2 – instanceof

* obj instanceof Cons
* Indique si l'objet(ou un prototype dans la chaine des prototypes !) a bien été créé par le constructeur.
* Exemple :

Une image contenant texte

Description générée automatiquement

* *Object* est le constructeur le plus général et *Object.prototype*, le prototype ancêtre de tous les objets.

## Faire de l'orienté objet en Javascript

### 4.2.1 – Objets à la volée

* Lorsqu'on a pas besoin d'une architecture complexe

### 4.2.2 – Héritage prototypale

* Plusieurs fonctions renvoyant des objets similaires
* On veut leur associer des fonctions communes 🡪 prototypes
* Object.create();

### 4.2.3 – Fonction constructrices

* Meilleure lisibilité
  + new + nom du constructeur
* Utile si les arguments du constructeur doivent être manipulé
  + Plutôt que juste stocké dans des attributs.
* On utilise le prototype associé automatiquement à la fonction

# Chapitre 5 : DOM et programmation événementielle

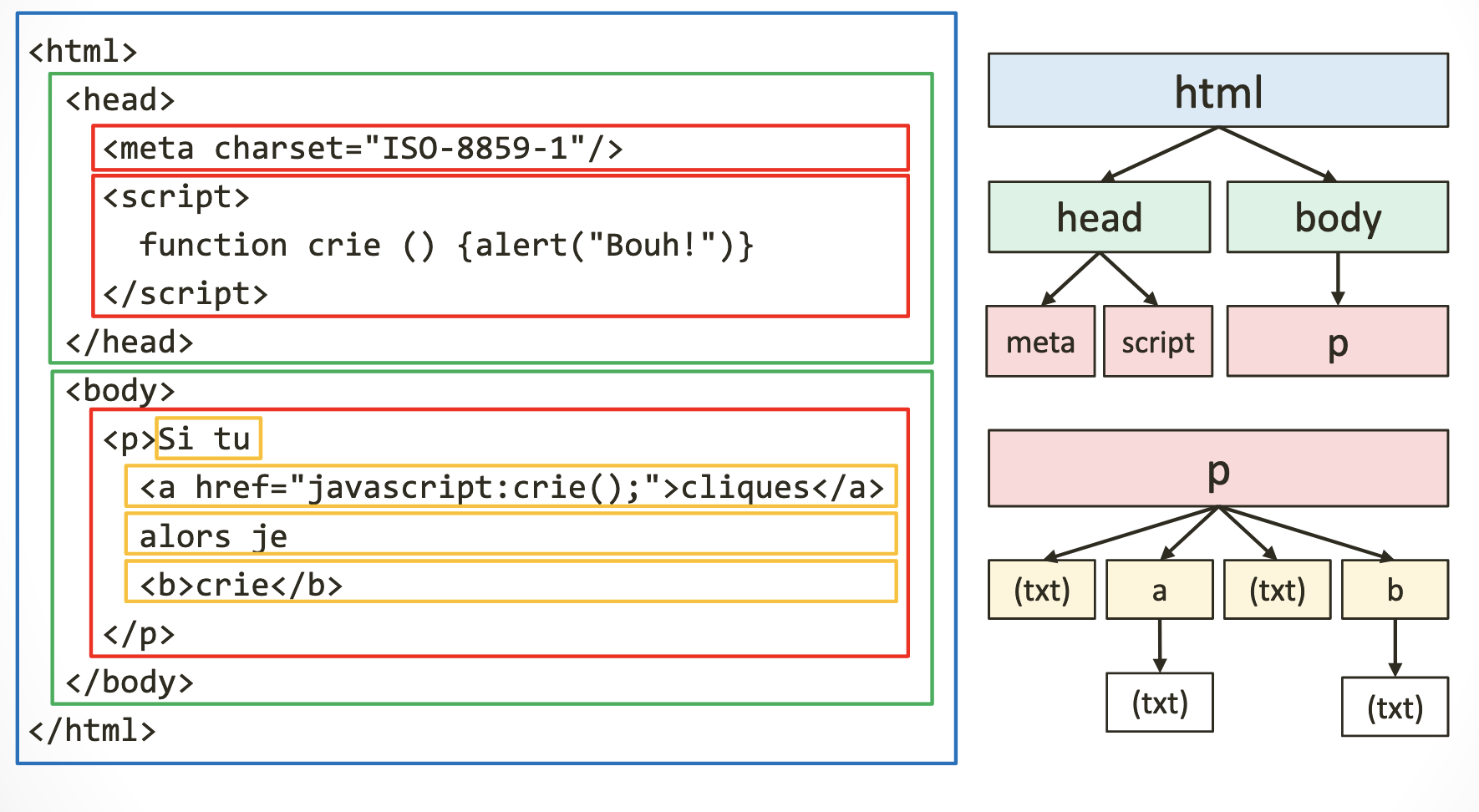
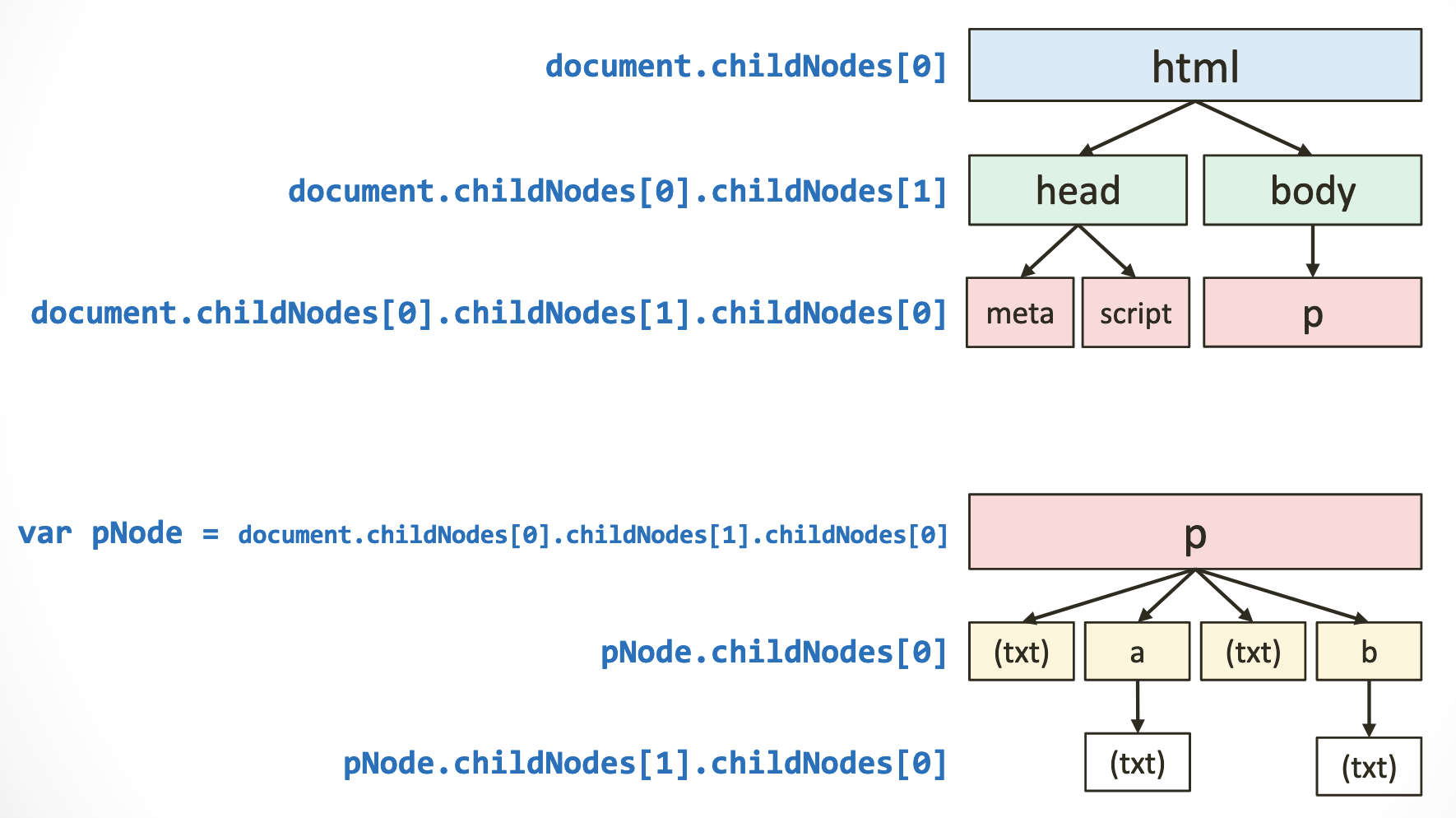
## Le DOM - intro

### 5.0.1 : Description

* DOM = Description Object Model
* Le but est de consulter et de modifier le contenu du document
* Standardisé depuis W3C
* Il indique comment chaque élément de l'arborescence est représenté en mémoire. Et comment utiliser ces objets pour rendre le HTML dynamique :
  + Recueillir des informations sur les éléments HTML
  + Modifier ces informations (Contenu, Attribut, CSS, …)
  + Manipuler l'arborescence HTML (Ajouter, déplacer, supprimer)
* Le DOM couvre également d'autres informations
  + Fenêtre d'affichage
  + Navigateur
  + …

### 5.0.2 – Object Document

* Représente le document HTML
* Structure en arbre
  + Chaque élément correspond à un nœud et est décrit par un objet
* Un nœud peut être :
  + en élément (nœud.nodeName = nom balise HTML)
  + donnée textuelle (nœud.nodeName = "#text")
    - Son contenu se trouve dans *nœud.textContent*
* Exemple d'arborescence :

## Utilisation du DOM

### 5.1.1 – Cibler un élément

#### Méthode 1 : Accès Direct

|  |  |
| --- | --- |
| **Accès direct** | |
| document.head | L'élément head |
| document.body | L'élément body |
| **Collections** | |
| document.anchors | Toutes les ancres <a name = "…"> |
| document.forms | Tous les formulaires <form> |
| document.images | toutes les images <img/> |
| document.link | Tous les liens <a href="…"> |

#### Méthode 2 : Accès indirect (Se déplacer autour d'un élément)

|  |  |  |
| --- | --- | --- |
| **Vers le bas** | | |
| **Collection** | | |
| elem | .childNodes | Fils de elem   * Soit des éléments eux-mêmes * Soit des données textuelles |
| .children | Fils de elem   * Qui sont des éléments * Ignore les éléments textuelles |
| Élément unique | | |
| elem | .firstChild | Premier fils  (dont texte) |
| .firstElementChild | Premier fils étant un élément |
| .lastChild | Dernier fils  (dont text) |
| .lastElementChild | Dernier fils étant un élément |
| **Vers le haut** | | |
| elem.parentNode | | Renvoie le parent du nœud spécifié |
| **Horizontalement** | | |
| elem | .nextSibling | Elément suivant  (dont texte) |
| .nextElementSibling | Elément suivant |
| .previousSibling | Element précédent  (dont texte) |
| .previousElementSibling | Elément précédent |

#### Méthode 3 : Recherche spécifique

|  |  |  |
| --- | --- | --- |
| **Réponse unique** | | |
| document | .getElementById(..) | Élément identifié |
| **Collection** | | |
| elem | .getElementsByTagName(balise) | En fonction de la balise |
| .getElementsByClassName(..) | En fonction du nom de la classe |
| **Via sélecteur CSS** | | |
| elem | .querySelector(..) | Renvoie le premier élément correspondant |
| .querySelectorAll(..). | Tous les éléments |

* Les collections sont itérables via for\_of

### 5.1.2 – Modification du DOM

|  |  |  |
| --- | --- | --- |
| **Modifier un élément** | | |
| Contenu HTML | | |
| elem | .innerHTML | Modifie le contenu HTML, les éléments |
| .textContent | Juste le text contenu dans l'élément |
| Valeur des attributs | | |
| elem | .href | Utilise la syntaxe OO pour cibler une propriété |
| .src |
| .setAttribute() | lien.setAttribute("href", "henallux.be"); |
| .getAttribute() | Donne la valeur de l'attribut |
| Le style d'affichage | | |
| elem.style | .backgroundColor | Transformer les noms des propriétés en camelCase |
| .color |
| Les classes | | |
| elem.classList | .add(..) | Ajouter une classe |
| .remove(..) | Supprimer une classe |
| .toggle(..) | Ajoute ou supprime s'il existe ou non |
| .contains(..) | Indique si une classe est présente ou pas |
| **Modifier l'arborescence** | | |
| Supprimer un nœud | | |
| parent | .removeChild(fils) | Supprimer le fils précisé |
| elem.innerHTML = ""; | | Pour supprimer tous les fils |
| Ajouter/déplacer un nœud | | |
| parent | .appendChild(nœud) | Ajoute elem comme dernier fils |
| .insertBefore(elem, nœud) | Ajout *elem* avant le *nœud* |
| .replaceChild(elem, nœud) | Remplace *nœud* par *elem* |
| Ajouter un nouveau nœud | | |
| Créer un nœud | let elem = document.createElement(balise) | |
| Remplir le nœud | elem.innerHTML = "lien"; | |
| Placer le nœud | utiliser une méthode pour ***Ajouter/déplacer un nœud***  Exemple : parent.appendChild(elem); | |

## Programmation événementiel

### 5.2.1 – Concept

* Associer :
  + Script
  + événement déclencheur
* Le DOM permet d'associer des scripts à une série d'évènements prédéfinis pour chacun des éléments HTML

### 5.2.2 – Lier une action à un événement

#### Méthode 1 : Dans le code HTML

* <button onclick="action();"> Click! </button>

#### Méthode 2 : Via Javascript

* but.addEventListener("click", action);
* but.removeEventListener("click", action);
* Arguments :
  + 1 : type d'évènement
  + 2 : fonction à exécuter

## window : La racine du DOM

### 5.3.1 - Description

* L'objet window représente la fenêtre où le document HTML est affiché
* C'est le plus gros objet du DOM
  + contient par exemple : document
  + On peut omettre le préfix 'window'
* C'est le contexte global

### 5.3.2 – Quelques propriétés

|  |  |  |
| --- | --- | --- |
| window | .document | Le document affiché |
| .location | adresse de la page |
| .history | historique de navigation de cette page |
| .frames | tableau des frames |
| .navigator | objet décrivant le navigateur utilisé |
| .screen | écran où l'affichage se produit |
| .opener | Fenêtre responsable de son ouverture |
| .status | Texte d'état |
| .screenX, .screenY | Position sur l'écran |
| .onload | Action a exécuté dès la fin de chargement de la page |
| .close() | Fermer la page |

# Chapitre 6 : Expressions régulières

## Introduction

### 6.1 - Définition

* Une expression régulière est une séquence de caractère qui représente un motif textuel
* Les expressions régulières se composent de symboles

### 6.2 – Utilisation

* Rechercher un motif dans un texte
* Valider une entrée
* Analyser des données textuelles
* Transformer des données textuelles

## Récap

|  |  |  |
| --- | --- | --- |
| **Caractère simple** | | |
| SAUF | a | Bateau |
| pom | pomme |
| . | n'importe quel caractère |
| \ ^ $ . \* + ? ( ) [ ] { } | | |
| **Motifs composé - Classes** | | |
|  | [0-9] | 0,1,2,3,…9 |
| [a,e,i,o,u,y]  [^a,e,i,o,u,y] | voyelle  Tout sauf une voyelle |
| [a,t,9$] | a – t – 9 - $ |
| **Classes prédéfinies** | | |
|  | \d  \D | un chiffre  Tout sauf un chiffre |
| \w  \W | caractère alphanumérique (dont chiffre)  Tout sauf alphanumérique |
| \s | Un blanc (espace, tab,…) |
| **Décorateurs** | | |
|  | x? | Eventuellement un x |
| \* | Séquence de 0 à x |
| + | Séquence de 1 à x |
| {x}  {x,y} | Répéter X fois  Répéter de X à Y fois |
| Tic|Tac | Tic OU Tac |
| ^mot | "mot" se trouvant en début de ligne |
| zoo$ | zoo se situant en fin de ligne |
| \bpoly | "poly" se trouvant en début de **MOT** |
| (.)v\1 | eve, ava, 3v3, … |
| (\d)(\d)\2\1 | Palindrome à 4 chiffres |

## Regex en Javascript

### 6.2.1 – Les flags

|  |  |
| --- | --- |
| i / ignoreCase | Distinction entre Majuscule minuscule |
| g / global | Recherche juste la première occurrence |
| m / multiline | Texte considéré sur une seule ligne |

### 6.2.2 – 2 méthodes pour créer un objet RegEx

#### Méthode 1 : Le littéral

/motif/flags

* + let regex = /^l[ae]/i;

#### Méthode 2 : La fonction constructrice

RegEx(motif,flags)

* + let regex = new RegExp("^l[ae]", "i");

### 6.2.3 – Utilisation des Regex

|  |  |  |
| --- | --- | --- |
| regex | .test(chaine) | Indique s'il existe une correspondance |
| .exec(chaine) | Décomposition en groupe |
| chaine | .search(regex) | Indice d'une correspondance |
| .match(regex) | Recherche de correspondance |
| .replace(regex,remplac) | Remplacement |
| .split(regex) | Découpe selon des séparateurs |

# Chapitre 7 : Compléments sur les fonctions

## Paramètres optionnelles :

### 7.0.1 – Paramètres optionnelles simples

function créePara (texte, classe) {

let code = "<p";

if (classe !== undefined)

code += " class='" + classe + "'";

code += ">" + texte + "</p>";

return code;

}

Ici, si on ne passe pas assez d'argument, classe n'aura pas de valeur. On vérifie alors s'il contient quelque chose avant de le traiter. On pourrait, via les conversion implicites, juste faire : !classe mais on exclue qu'une valeur "*falsy"* soit valide. (comme 0 par exemple). On préfère alors vérifier via '==='.

### 7.0.2 – Valeurs par défaut (1 / 2)

|  |  |
| --- | --- |
| function salue (nom) {  if (nom) {  alert("Hi, " + nom + "!");  } else {  alert("Hi, anonyme!");  }  } | function salue (nom) {  nom = nom || "anonyme";  alert("Hi, " + nom + "!");  } |

Utiliser l'opérateur "||" pour donner la valeur à *"nom"* en fonction de si la variable est remplie ou pas. On fait appel sans le savoir aux conversions implicites qui change (dans l'opérations OU) la valeur de *"nom"* par un booléen. Donc si *"nom"* est *undefined*, on l'initialise à "anonyme". Il faut encore faire attention aux cas où des valeurs falsy sont acceptables !!

### 7.0.3 – Valeurs par défaut (2 / 2)

function salue (nom = "anonyme") {

alert("Hi, " + nom + "!");

}

/!\ Seul undefined déclenche l'utilisation de la valeur par défaut /!\

On peut également utiliser une variables précédente pour l'initialisation par défaut

function créeLien (url, texte = url) {

return `<a href="${url}">${texte}</a>`;

}

### 7.0.4 – Surcharge

* N'existe pas, on ne peut créer deux fonctions qui possède le mêmes noms et un nombre/type de variables différents.
* Il faut tout traiter dans une seule fonction,
  + Quitte à vérifier le type des valeurs envoyés !

### 7.0.5 – rest…parameter

* Permet d'envoyer un nombre indéfinis de paramètres
* Il ne peut y en avoir qu'un, et doit obligatoirement se trouver à la dernières position !
* Lorsqu'il est dans une fonction appelée :
  + Rassemble toutes les valeurs en un tableau.
* Lorsqu'il est dans un appel :
  + Converti le tableau en liste d'argument
  + C'est alors le *Spread Operator*

function moyenne (...valeurs) {

let somme = 0;

for (let valeur of valeurs)

somme += valeur;

return somme / valeurs.length;

}

### 7.0.6 – Déstructuration

* Permet d'accéder directement aux composants d'une valeur structurée
* Valeur structurée :
  + Objets et tableaux
* Accès direct :
  + Les associer à un nom de variable
* Exemple avec les tableaux :

afficheHeure([13, 37, 30]);

function afficheHeure (tHeure) {

console.log `${tHeure[0]}:${tHeure[1]}:${tHeure[2]}`;

}

function afficheHeure ([h, m, s]) {

console.log `${h}:${m}:${s}`;

}

* Exemples avec les objets :

afficheHeure({heure: 13, min: 37, sec: 30});

function afficheHeure (oHeure) {

console.log `${oHeure.heure}:${oHeure.min}:${oHeure.sec}`;

}

function afficheHeure ({heure, min, sec}) {

console.log `${heure}:${min}:${sec}`;

}

## Le mot-clé *this*

En Javascript, une fonction peut être :

* Une fonction simple
  + afficherHeure();
* Méthode
  + obj.afficherHeure()
* Constructeur
  + new Heure()

Les trois cas possèdent la même syntaxe, c'est leur utilisation qui diffère.

C'est donc la manière dont on les utilise qui va déterminer ce que *this* signifie.