Programmation Web – Avancé

JavaScript & Node.js

Introduction au langage JS côté client

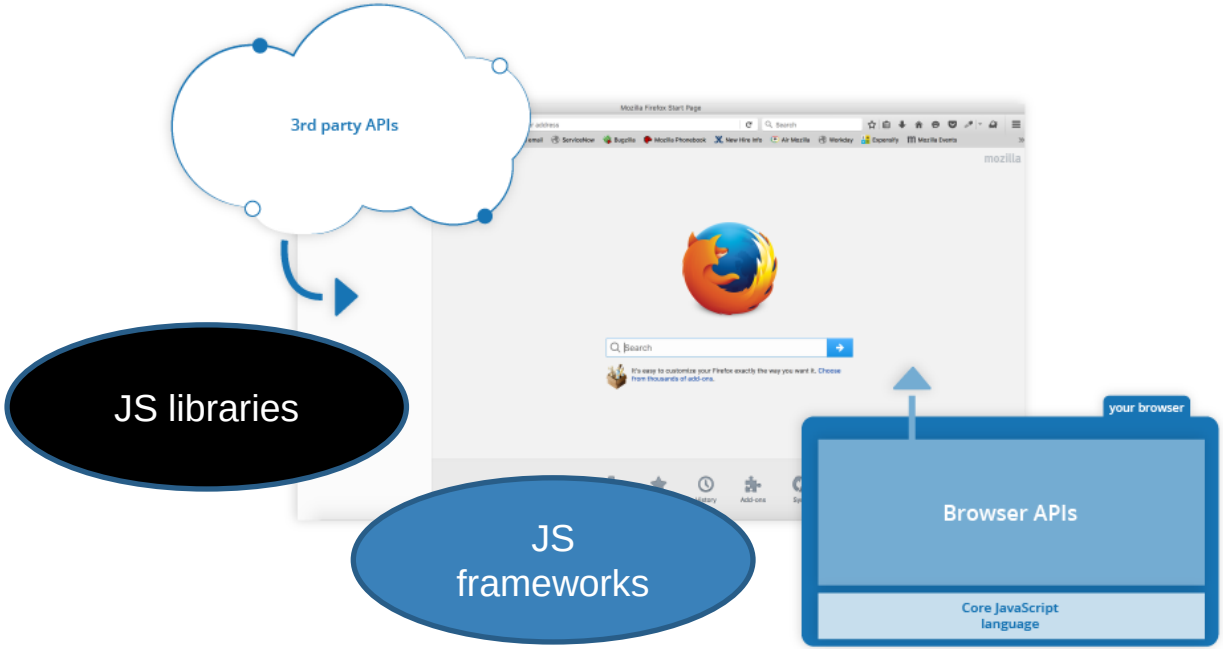
Introduction à l’utilisation de JS côté client

Découvrons le JS côté client…

But de JS côté-client ?

- Création en 1995 par Brendan Eich pour rendre les pages web interactives
- Interactions dynamiques avec l'HTML et le CSS d'un browser
- Exécution de JS après la génération d'une page web comprenant du HTML et potentiellement du CSS

Introduction au JS côté client



Relation entre APIs, le browser et le JS [15.]

JS côté client : quel Software utiliser pour développer ?

- Browser : **Chrome**
- Pour écrire vos scripts : **VS Code**
- Pour exécuter vos scripts sur un serveur local :
  - **Live Server** (extension de VS Code)
  - **Node.js / Express** (distribution de fichiers statiques)
  - **Webpack**

JS côté client : quel Software utiliser pour développer ?

- Gestion de différentes versions de votre code et différentes machines : **Git, Gitlab / Github** et **VS Code**
- Extensions de VS Code (optionnel) :
  - Pour des snippets JS
  - Pour des snippets Bootstrap
  - Pour collaborer sur du code : **Visual Studio Live Share**

Exécution et déploiement d'applications web sur le cloud : quels services ?

- Pour développer & exécuter vos scripts sur le cloud (optionnel) : **CodeSandbox**
- Pour déployer vos applications : **Netlify** ou **Heroku**

JS côté client : où mettre votre code ?

A. Directement dans un browser

B. Dans une page HTML
   1. Dans un fichier externe
   2. Dans un fichier HTML, dans une fonction ou événement associé à un élément HTML
   3. Dans un fichier HTML, à l'aide de la balise **<script>**

JS côté client : où mettre votre code ?

A. Directement dans un browser

- Accès à la console : **F12** ou **CTRL + SHIFT + i** (Chrome)
- **PRESENTATION : opérations mathématiques de base dans Chrome**

JS côté client : où mettre votre code ?

B. Dans une page HTML

1. Dans un fichier externe avec **<script src='file_name.js'>** dans un fichier HTML à la fin de la balise <body>
   - **DEMO : JS dans un script externe**

JS côté client : où mettre votre code ?

- Fichiers séparés pour chaque catégorie de code : **.html**, **.js**, **.css**

JS côté client : où mettre votre code ?

B. Dans une page HTML

2. Dans une fonction ou événement associé à un élément HTML
   - **DEMO : JS interne dans une fonction**
   - Notion : **<body onload='function_Name()'>**

JS côté client : où mettre votre code ?

B. Dans une page HTML

3. Dans la section **<body>** ou **<head>** d'un fichier HTML, à l'aide de la balise **<script>**
   - **DEMO : JS dans la balise script**

Instructions JS

- Composition d'une instruction JS (« statement » normalement séparé par « **;** ») à exécuter par le browser :
  - Expressions
  - Valeurs
  - Opérateurs (**+**, **-**…)
  - Mots clés (**for**, **break**…)
  - Commentaires.

Instructions JS

- Séparation de chaque instruction par un « ; »

```
let x = 1 ;
console.log("x = ", x);
```

Les commentaires

- Ajout de commentaires : via // ou /* et */

```
/**
 * JSDoc as comments
 * @param {message} message to be displayed in console
 */
function raiseAlert(message) {
  // Single line comment
  console.log(message);
  /* Regular comment
    on multiple lines
    */
  console.log("An alert has been raised.");
}
```

Déclaration, initialisation et mise à jour de variables

- Variables sensibles à la casse

```
// two different variables
let monBrowser ;
let monbrowser ;
```

- Pas de déclaration du type de variable (« langage dynamiquement typé »)
  - Type d'une variable déterminé à l'exécution

Déclaration, initialisation et mise à jour de variables

- Pour une portée associée à un bloc : **let**

```
if (true) {
  let blockScope = "Hello";
  console.log(blockScope); // Hello
}

console.log(blockScope); // Uncaught ReferenceError: blockScope is not
                            defined
```

Déclaration, initialisation et mise à jour de variables

- Pour une portée associée à un bloc : **let**
  - variable pas accessible en dehors du block, processée à l'exécution seulement
  - pas de redéclaration possible dans le même bloc
  - Pas de création de propriété sur l'objet global (window.newProperty)
  - Détails [16.]

Déclaration, initialisation et mise à jour de variables

- Pour une portée associée à un bloc mais immuable : **const**

```
if (true) {
  const constVar = "Hello";
  console.log(constVar); // Hello
  const SITE_URL = "http://MyCMS.org";
  console.log(SITE_URL); // http://MyCMS.org
  constVar = "Hi";
  console.log(constVar); // Uncaught TypeError: Assignment to constant
                            variable.
}
```

Déclaration, initialisation et mise à jour de variables

- Pour une portée globale si déclaration en dehors de toute fonction : **var**

```
if (true) {
  var globalVar = "Hello";
  console.log(globalVar); // Hello
}
console.log(globalVar); // Hello
```

Déclaration, initialisation et mise à jour de variables

- Pour une portée globale si déclaration en dehors de toute fonction : **var**
  - processée avant l'execution du code ("hoisting") ;
  - variable "globale" accessible au travers de tout le programme ;
  - redéclarable dans n'importe quel bloc.

Déclaration, initialisation et mise à jour de variables

- Pour une portée associée à une fonction : **let**, **var**, **const**

```
function checkScopeVarInFunction() {
  var varInFunction = "Hello";
  console.log(varInFunction); // Hello
}
checkScopeVarInFunction();
console.log(varInFunction); // Uncaught ReferenceError: varInFunction is not
                                defined
```

Déclaration, initialisation et mise à jour de variables

- Pour l’assignation d’une valeur à une variable non déclarée :
  - Création implicite d’une variable globale

```
function checkScopeVarInFunction() {
  varInFunction = "Hello";
  console.log(varInFunction); // Hello
}
checkScopeVarInFunction();
console.log(varInFunction); // Hello
```

Déclaration, initialisation et mise à jour de variables

- Les dangers des variables globales : **var** ou variable non déclarée

```
var index = 1;
for (index; index <= 3; index++) {
  console.log(index); // 1 2 3
}
print();
function print() {
  for (index; index <= 5; index++) {
    console.log("Print " + index); // Print 4 Print 5
  }
}
```

Déclaration, initialisation et mise à jour de variables

- Déclaration de ses variables
- Portée de bloc via **let** ou **const**

Types de variables

- **Number** (Nombre) : un seul type pour les entiers, réels, doubles…
- **String** (Chaîne) : comprise entre guillemets simples ou doubles.
- **Bool** (Booléen)
- **Array** (Tableau)
- **Object** (Objet)

Types de variables

- Renvoi sous forme d’une chaîne le type d’une expression : opérateur **typeof**

```
console.log(typeof 12); // number
console.log(typeof "I love JS"); // string
console.log(typeof true); // boolean
console.log(typeof undeclaredVariable); // undefined
```

Opérateur d’égalité ou de non égalité stricte

- Stricte sans conversion de type : `===` ou `!==`
- Avec conversion de type : `==` ou `!=`

```
1 === 1; // true
"1" === 1; // false
1 == 1; // true
"1" == 1; // true
0 == false; // true
0 == null; // false
var object1 = { key: "value" },
  object2 = { key: "value" };
object1 == object2; //false
```

Les opérateurs de comparaisons

- Utilisation de l’égalité stricte sauf si volonté de conversion du type

Les opérateurs

- Les opérateurs logiques : **&&**, **||** , **!**
- …
- Détails [17.]

Les conditions

- Instructions conditionnelles : **if … else**
- Détails [18.]

```
let isAuthenticated = false;
if (isAuthenticated) {
  console.log("Render the HomePage.");
  console.log("You are authenticated.");
} else {
  console.log("Render the Login Page."); // Render the Login Page.
  console.log("You are not authenticated."); // You are not authenticated.
}
```

Instructions conditionnelles : switch

```
let foo = 0;
switch (foo) {
  case -1:
    console.log("negative 1");
    break;
  case 0: // foo is 0 so criteria met here so this block will run
    console.log(0);
   case 1: // no break statement in 'case 0:' so this case will run as well
    console.log(1);
    break; // it encounters this break so will not continue into 'case 2:'
  case 2:
    console.log(2);
    break;
  default:
    console.log("default");
}
```

Les fonctions personnalisées et anonymes

- Fonctions personnalisées

```
function welcomeMessage(message) {
  return "Message : " + message;
}
let message = welcomeMessage("Welcome to everyone!");
console.log(message); // Message : Welcome to everyone!
```

Les fonctions personnalisées et anonymes

- Assigner une fonction comme valeur de variable

```
function welcomeMessage(message) {
  return "Message : " + message;
}
let x = welcomeMessage;
message = x("Hi");
console.log(message); // Message : Hi
```

Les fonctions personnalisées et anonymes

- Fonction anonyme (ou sans nom)

```
const welcome = function (message) {
  return "Message : " + message;
};
message = welcome("Hello world ; )");
console.log(message); // Message : Hello world ; )
```

Les fonctions personnalisées et anonymes

- Arrow function

```
const welcome2 = (message) => {
  return "Message : " + message;
};
message = welcome2("Hello world...");
console.log(message); // Message : Hello world...

// OTHER EXAMPLE
const higher = n => n + 1;
console.log(higher(1)); // 2
```

Paramètres (ou arguments) de fonctions

- Optionnels : valeur **undefined** pour un paramètre manquant
- Portée locale au sein de la fonction
- Passage d’argument par valeur, sauf pour les objets
- Passage d’un objet par référence

Paramètres : passage par valeur

```
let myMessage="Hello";
print(myMessage);
function print(myMessage) {
    console.log(myMessage); // Hello
    myMessage = "Good bye";
}
console.log(myMessage); // Hello
```

Paramètres : passage d'un objet par référence

```
let myMessage = { content: "Hello" };
consolePrint(myMessage);
function consolePrint(myMessage) {
  console.log(myMessage.content); // Hello
  myMessage.content = "Good bye";
}
console.log(myMessage.content); // Good bye
```

Les fonctions

- Paramètres optionnels avec valeur par défaut : **=**
- Retour de la valeur d'une fonction : **return**
- Détails [19.]

```
let welcome3 = function (message = "HI DEAR HUMAN!") {
  return "Message : " + message;
};
message = welcome3();
console.log(message); // HI DEAR HUMAN
```

Les boucles

- Les boucles : **for**, **for/in**, **for/of**, **while**, **do/while**
- Détails [20.]

```
for (let index = 0; index < 5; index++) {
  console.log(index); // 0 1 2 3 4
}
```

Les tableaux

- Type **Array** (objet natif) : création d'un ensemble ordonné de valeurs auxquelles ont fait référence avec un nom et un indice.

Création d’un tableau rempli

```
const LIBRARIES = ["Anime.js", "Three.js", "Phaser.io"];
const emptyArray = [];
const LIBRARIES_NOT_RECOMMENDED = new Array("Anime.js", "Three.js",
"Phaser.io");
const emptyArrayNotRecommended = new Array();
const arr = new Array(101) ; // What is the result ?
```

- Création d’un Array à l’aide de [] et non pas via l’appel du constructeur

Parcourir un tableau

- Boucle classique : **for** & **.length**
- Méthode **forEach()** d'un **Array**

```
for (let index = 0; index < LIBRARIES.length; index++) {
  console.log(LIBRARIES[index]); // Anime.js Three.js Phaser.io
}

LIBRARIES.forEach((item, index) => console.log("[" + index + "]: " + item));
// [0]: Anime.js [1]: Three.js [2]: Phaser.io

LIBRARIES.forEach(function (item) {
  return console.log(item); // Anime.js Three.js Phaser.io
});
```

Tableau multi-dimensionnel

- Un élément d’un **Array** est un **Array**

```
const numberOfRows = 2,
const numberOfColumns = 2;
const myTab = [];
for (let x = 0; x < numberOfRows; x++) {
  myTab[x] = [];
  for (let y = 0; y < numberOfColumns; y++) {
    myTab[x].push("[" + x + "][" + y + "]");
    // myTab[x][y] = "[" + x + "][" + y + "]"; not recommended
    console.log(myTab[x][y]);
  }
}
```

Les tableaux

- Autres méthodes associées à un **Array** [21.] : **pop()**, **map()**, **indexOf()** …

</> Template literals : ``

- Inclusion d’expressions dans des Strings via ${}
- String sur plusieurs lignes

```
for (let i = 0; i < LIBRARIES.length; i++) {
  htmlText += `<li class='list-group-item'>
                 ${LIBRARIES[i]}
               </li>`;
}
```

Les objets en JS

- Les chaînes, tableaux, APIs du browser, objets personnalisés…
- Plus de détails dans la partie « L’orienté objet en JS »

Création d’un objet

- Via un « object literal » = liste de paires **key / value** (**nom de propriété / valeur**)

```
let raphael = {
  firstname: "Raphael",
  lastname: "Baroni",
  sayHello: () => "Hi everyone !",
};
```

Création d’un objet

- Via {} ou **new**

```
let sandra = {};
sandra.firstname = "Sandra";
sandra.lastname = "Parisi";
```

Accéder aux propriétés d’un objet

- Soit via un point

```
console.log(raphael.firstname, " :" , raphael.sayHello());
// Raphael  : Hi everyone !
```

- Soit via des ["object_key"]

```
console.log(sandra["firstname"], ",", sandra["lastname"]);
// Sandra , Parisi
```

Les exceptions : lancer une exception

- **throw expression**
- Les exceptions de manière générale [18.]
- Construction de vos propres erreurs (throw String, Number, Boolean ou Object)

```
function divideXByY(x, y) {
  if (y === 0)
    throw "Division by 0 ! ";
  return x / y;
}
divideXByY(5, 0); // Uncaught Division par 0 !
```

Les exceptions : lancer une exception

- Utilisation de l'objet **Error** ou d'un autre type d'erreur (**RangeError**, **SyntaxError**…)

```
function RegularDivideXByY(x, y) {
  if (y === 0) throw new RangeError("Division by 0 ! ");
  return x / y;
}
try {
  RegularDivideXByY(5, 0);
} catch (err) {
  console.log("RegularDivideXbyY():", err.name, ":", err.message);
  // RegularDivideXbyY() : RangeError: Division by 0 !
}
```

Les exceptions : lancer une exception

- Deux propriétés intéressantes de « JS built-in error object » : **name & message**
- Détails sur les types d'erreurs [22.] (différents objets)
- Construction de vos propres classes d'erreur (héritage de la classe **Error**)

Les exceptions : intercepter une exception

- **try … catch**
  - **try{…}**: partie de code monitorée
  - **catch(err){…}** : instructions en réponses à une exception
  - Code à exécuter après **try… catch** : **finally{}**

Les exceptions : intercepter une exception

```
let result;
try {
  result = RegularDivideXByY(5, 0);
} catch (err) {
  console.log("RegularDivideXbyY():", err.name, ":", err.message); // RegularD
ivideXbyY() : RangeError: Division by 0 !
} finally {
  console.log("RegularDivideXbyY() results:", result, "JS Divion's result",
    5 / 0);
   // RegularDivideXbyY() results: undefined JS Division's result Infinity
}
```