

JavaScript

JavaScript

A yellow square containing the letters 'JS' in a bold, black, sans-serif font.

- Historiquement permet de programmer des interactions au sein des navigateurs
 - **Interagir** : savoir qu'un bouton a été cliqué
 - **Afficher** : manipuler la page web pour rendre visible des nouvelles parties
 - **Communiquer** : envoyer ou recevoir des requêtes
- **C'est le seul langage disponible côté navigateur**
- Mais est aussi disponible côté serveur via NodeJS

Exemple

```
<!doctype html>
<html lang="fr">
  <head>
    <title>Exemple</title>
    <script>
      var msg = "hello";
      alert(msg);
    </script>
  </head>
  <body>
  </body>
</html>
```



Best practice

```
<!doctype html>  
<html lang="fr">  
  <head>  
    <title>Example</title>  
    <script src="script.js"></script>  
  </head>  
  <body>  
  </body>  
</html>
```



Messages de log

L'instruction qui permet d'afficher des messages de log en JavaScript est :

script.js

```
console.log('Hello, world!');
```

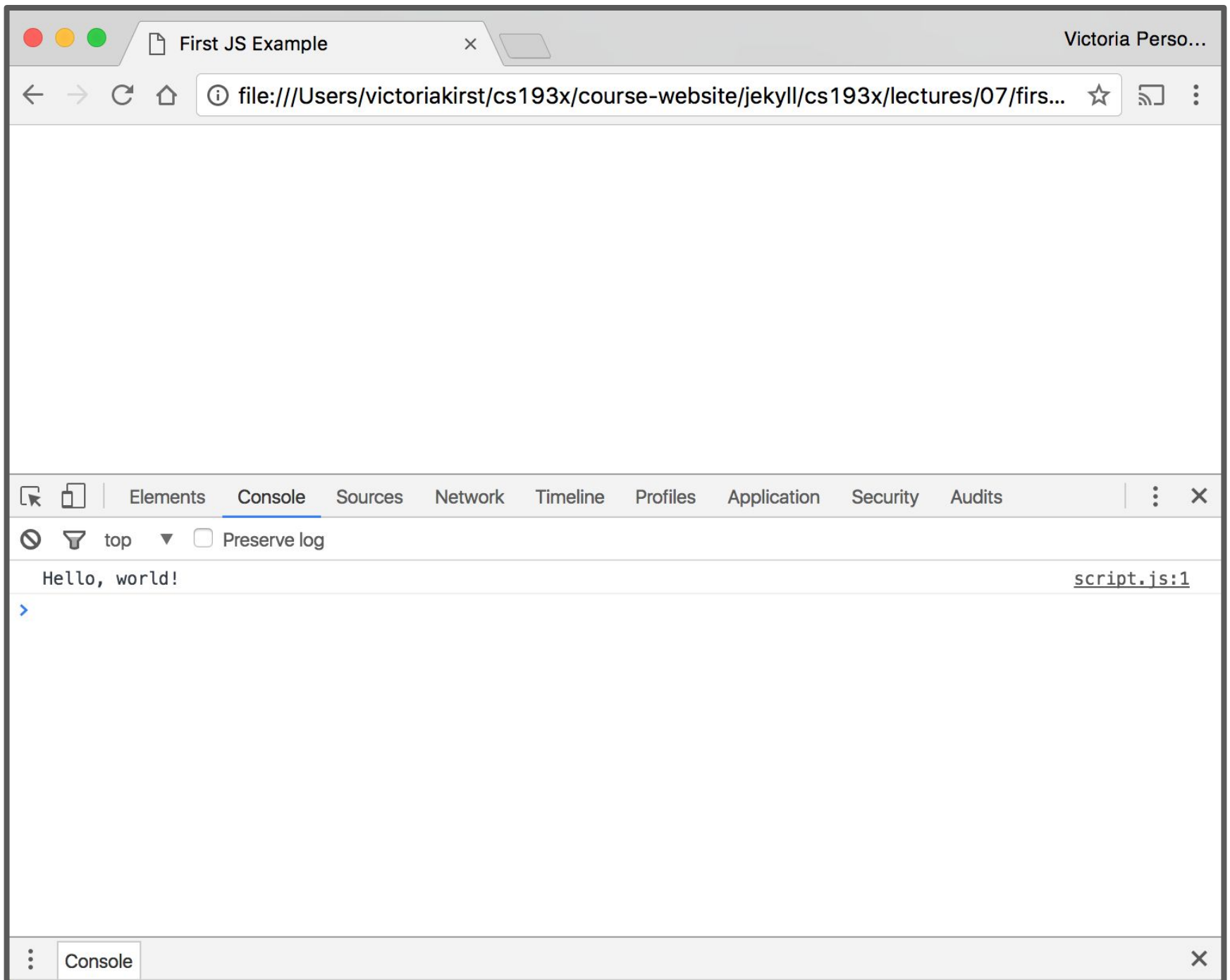
Exécution de JavaScript

Il n'y a pas de "**main method**"

- Le script est exécuté de haut en bas

Il n'y a pas de **compilation** par le développeur

- JavaScript est compilé et exécuté à la volée par le navigateur



Similarités avec Java, C, ...

for-loops:

```
for (let i = 0; i < 5; i++) { ... }
```

while-loops:

```
while (notFinished) { ... }
```

comments:

```
// comment or /* comment */
```

conditionals (if statements):

```
if (...) {  
    ...  
} else {  
    ...  
}
```


Fonctions

La syntaxe suivante est une des manières de définir une fonction en JavaScript

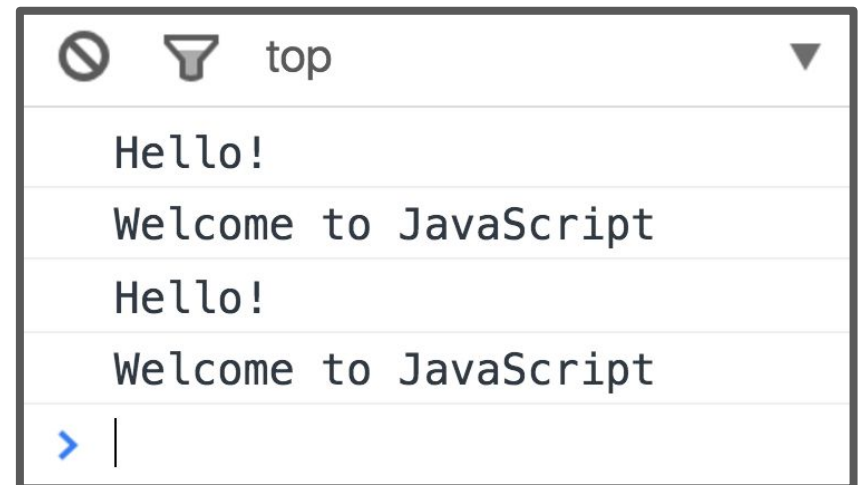
```
function name() {  
    statement;  
    statement;  
    ...  
}
```

Exemple

```
hello();  
hello();  
  
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

Cela fonctionne car les déclarations de fonctions sont ***hoisted*** : déplacées au sommet du scope dans lesquelles elles sont définies

il faut éviter de se reposer sur ce mécanisme



Variables

Trois façons de déclarer des variables en JS

```
// Function scope variable  
var x = 15;  
// Block scope variable  
let fruit = 'banana';  
// Block scope constant; cannot be reassigned  
const isHungry = true;
```

Le langage est dynamiquement typé

Paramètres de fonctions

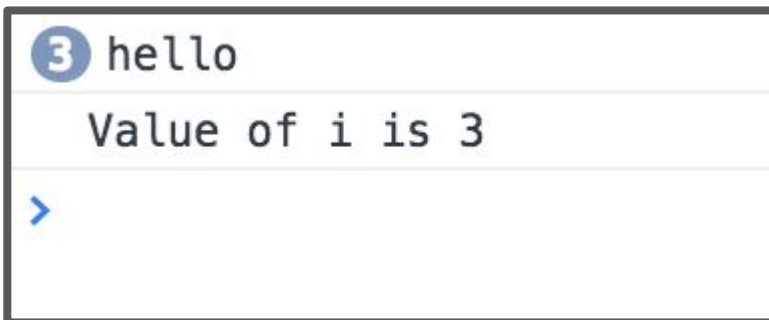
```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
}
```

Les paramètres de fonctions ne sont pas déclarés à l'aide de `let`, `const` ou `var`

Comprendre var

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```

```
printMessage('hello', 3);
```



A screenshot of a JavaScript console window. It shows three lines of output: a blue circle with the number 3 followed by the text 'hello', the text 'Value of i is 3', and a blue prompt character '>' on the third line.

```
3 hello  
Value of i is 3  
>
```

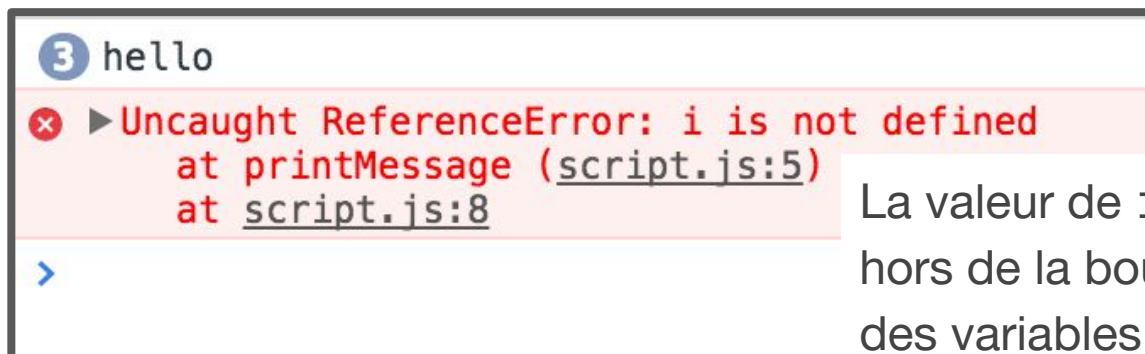
La valeur de `i` est accessible hors de la boucle `for` car `var` déclare des variables avec un scope *fonction*

Comprendre let

```
function printMessage(message, times) {  
  for (let i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```



```
printMessage('hello', 3);
```



The screenshot shows a browser console with a blue circle containing the number 3 and the text 'hello'. Below this, a red error message is displayed: 'Uncaught ReferenceError: i is not defined'. The error message includes the location 'at printMessage (script.js:5)' and 'at script.js:8'. A blue prompt character '>' is visible at the bottom of the console.

La valeur de `i` n'est pas accessible hors de la boucle `for` car `let` déclare des variables avec un scope *block*

Comprendre const

```
const y = 10;  
y = 0;           // error!  
y++;            // error!  
const list = [1, 2, 3];  
list.push(4);    // OK
```

Les variables déclarées avec `const` ne peuvent pas être réaffectées

Cependant il reste possible de modifier l'objet sous jacent

- Ressemble au `final` de Java

Bonnes pratiques

- Utiliser `const` partout où vous pouvez
- Si vous avez besoin d'une variable réaffectable, utilisez `let`
- **N'utilisez pas `var`.**
 - `const` et `let` sont maintenant bien supportés par les browsers

Types

Les **variables** JS n'ont pas de types, mais leurs **valeurs** si

Il y a plusieurs types primitifs:

- **Boolean** : true et false
- **Number** : tout est de type double (pas d'entiers)
- **String**: avec 'single' ou "double-quotes"
- **Null**: null une valeur qui signifie “ceci n’a pas de valeur”
- **Undefined**: la valeur d’une variable non affectée

Il y a aussi les types Object, comme Array, Date, et même Function!

Transtypage booléen

Les valeurs non booléennes peuvent être utilisées dans les instructions de contrôle, elles sont converties en "truthy" ou "falsy"

- `null`, `undefined`, `0`, `NaN`, `' '` évaluent à `false`
- Les autres valeurs évaluent à `true`

```
if (username) {  
    // username is defined  
}
```

Egalité

`==` et `!=` font une conversion implicite de type avant comparaison

```
' ' == '0' // false
' ' == 0   // true
0  == '0'  // true
NaN == NaN // false
[ ' ' ] == ' ' // true
false == undefined // false
false == null // false
null == undefined // true
```

Opérateurs === et !==

=== et !== sont les véritables opérateurs de comparaison,
toujours les utiliser!

```
' ' === '0'    // false
' ' === 0      // false
0 === '0'     // false
NaN === NaN    // still weirdly false
[ ' ' ] === ' ' // false
false === undefined // false
false === null  // false
null === undefined // false
```

null et undefined

Quelle différence?

- `null` est la valeur représentant l'absence de valeur (comme `null` en Java)
- `undefined` est la valeur d'une variable n'ayant pas reçu de valeur

```
let x = null;  
let y;  
console.log(x);  
console.log(y);
```

`null`

`undefined`



Tableaux

Les tableaux sont les objets pour définir des listes

```
// Creates an empty list
const list = [];
const groceries = ['milk', 'cocoa puffs'];
groceries[1] = 'kix';

// For each loop
for (let item of groceries) {
  console.log(item);
}
```

Objets

Collection de paires clé - valeur, peut être utilisé comme une hashmap

```
const prices = {};  
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};  
console.log(scores['peach']);    // 100  
console.log(scores.peach);      // 100  
scores.peach = 20;  
console.log(scores.peach);      // 20
```

Itérer les propriétés d'un objet

Il est possible d'itérer sur les propriétés d'un objet

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};  
  
for (let name in scores) {  
  console.log(name + ':' + scores[name]);  
}
```


Fonctions de premier ordre

En JavaScript, les fonctions sont des objets comme les autres

```
var add = function(a, b) {  
    return a + b;  
}
```

```
add(2, 2); // 4
```

Ajout d'une fonction sur un objet

On peut ajouter une fonction dans un objet, `this` désigne l'objet receveur

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91,  
  total: function() {  
    return this.peach + this.mario + this.luigi;  
  }  
};
```

Callbacks

Les fonctions peuvent être passées en paramètres d'autres fonctions

```
const groceries = ['milk', 'cocoa puffs'];

// For each loop
for (let item of groceries)
  console.log(item);

// Callback style
groceries.forEach(function(item) {
  console.log(item);
});

// Nerdy callback style (use that!)
groceries.forEach(item => {
  console.log(item);
});
```

Mais que fait forEach?

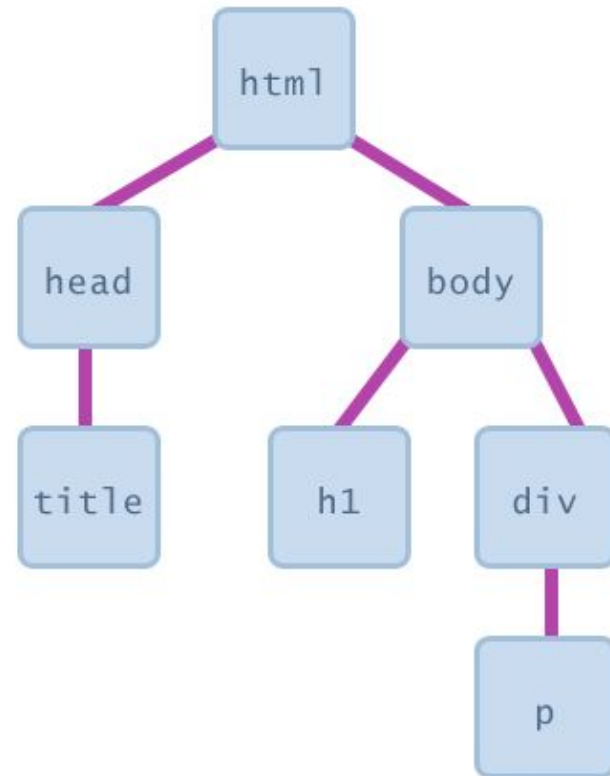
```
class Array {  
  forEach(callback) {  
    for (let i = 0; i < this.length; i++) {  
      callback(this[i], i, this);  
    }  
  }  
}
```

forEach itère sur tous les éléments du tableau, pour chaque élément, elle applique la fonction callback (passée en paramètre de forEach) en lui fournissant en paramètre l'élément courant **forEach calls callback back!**

DOM

On accède aux éléments composant la page web en JavaScript au travers du **Document Object Model**

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1></h1>
    <div>
      <p></p>
    </div>
  </body>
</html>
```



DOM et JavaScript

JavaScript peut :

- **examiner** les noeuds du DOM pour en inspecter l'état (ex. voir le texte saisi par un utilisateur)
- **editer** les attributs des noeuds du DOM (ex. changer le style d'un élément <h1>)
- **ajouter ou supprimer** des noeuds du DOM (ex. ajouter un texte de statut quelque part)

Accéder aux noeuds du DOM

L'accès aux noeuds du DOM se fait via la fonction `querySelector`:

```
document.querySelector('css selector');
```

- Retourne le **premier** noeud qui matche la règle CSS

```
document.querySelectorAll('css selector');
```

- Retourne **tous** les éléments qui matchent la règle CSS

Quelques propriétés des noeuds du DOM

Property	Description
id	la valeur de l'attribut id de l'élément
innerHTML	le HTML entre le noeud ouvrant et fermant de l'élément vu comme une chaîne de caractères
textContent	Le contenu texte d'un noeud ainsi que celui de ses descendants
classList	Un objet contenant toute la liste des classes dans lesquelles l'élément se situe

Créer des éléments dans le DOM

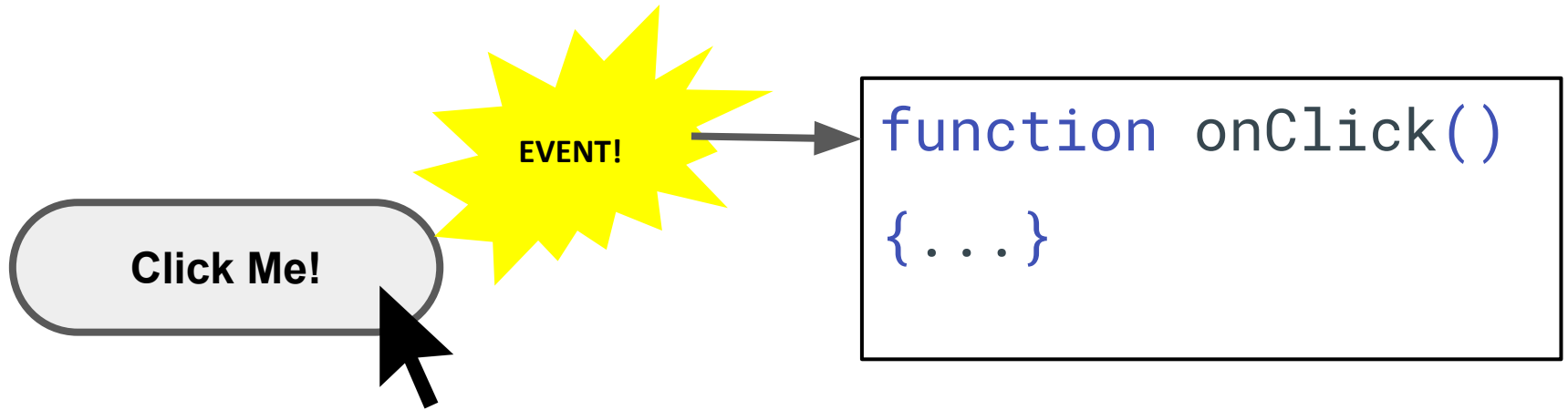
Il est possible de créer des éléments dynamiquement via `createElement` et `appendChild`:

```
document.createElement(tag string)  
    element.appendChild(element);
```

On peut supprimer des éléments via `remove`

```
element.remove();
```

Évènements



Exemple : un élément de page avec lequel on peut interagir. Quand on clique sur le bouton, un événement est déclenché. L'utilisateur peut enregistrer des callbacks sur les événements de son choix

```
<html>
  ▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer</script>
  </head>
  ▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

Click Me!



Elements

Console



top

clicked

