Conception Et Programmation Web

LIFWEB - http://lifweb.pages.univ-lyon1.fr/

Semestre printemps 2024-2025
L3 - UCBL

Aurélien Tabard - https://tabard.fr/, basé sur les cours de Romuald Thion

JavaScript et asynchrone

Aurélien Tabard - 2024-2025 - basé sur les cours de Romuald Thion

JavaScript et asynchrone

Plan

1. Événements JavaScript
2. Portée des variables et fermetures en JavaScript
3. Programmation asynchrone en JavaScript

“Les événements DOM sont déclenchés pour notifier au code des « changements intéressants » qui peuvent affecter l’exécution du code. Ces changements peuvent résulter d’interactions avec l’utilisateur, […], de changements dans l’état de l’environnement […], et d’autres causes.”

MDN - Event reference

Propagation des événements

Bubbling : les événements remontent le long de l'arbre DOM des feuilles (plus précis, les plus imbriqués) à la racine.

Capture : les parents peuvent capturer les événements et ne pas les transmettre aux enfants. (*inverse du bubbling*)

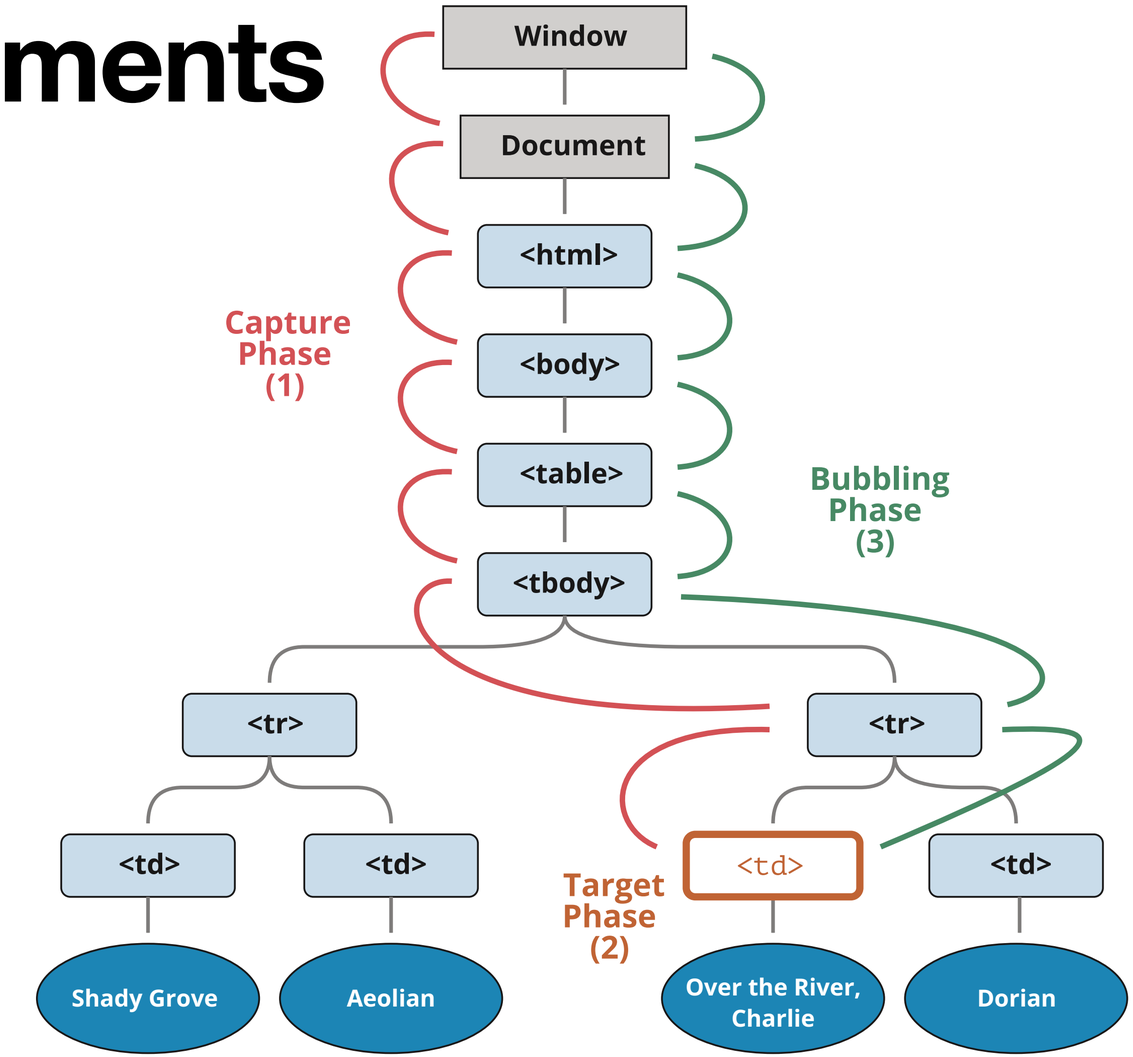
Voir MDN et javascript.info.

Propagation des événements

Bubbling est la phase “normale” de gestion des événements

`Capture` est très rarement utilisé

- Pour surveiller les rares événements qui ne *bubble* pas (`focus`, `load`…)
- Bloquer la propagation aux enfants



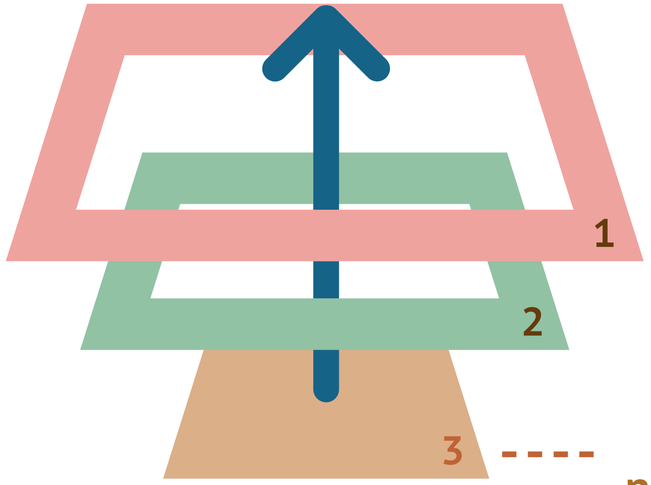
https://javascript.info/bubbling-and-capturing

Exemple de bubbling

https://javascript.info/bubbling-and-capturing

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```



Gérer les événements

“De nombreux éléments DOM peuvent être paramétrés afin d’accepter (« d’écouter ») ces évènements et d’exécuter du code en réaction pour les traiter (« gérer »).”

MDN - Event

- Un objet DOM (interface **EventTarget**) peut *écouter* (listen) certains événements.
- On attache une fonction appelée **handler** ou **listener** à cet événement.
- Le **handler** est appelé à chaque occurrence de l’événement.

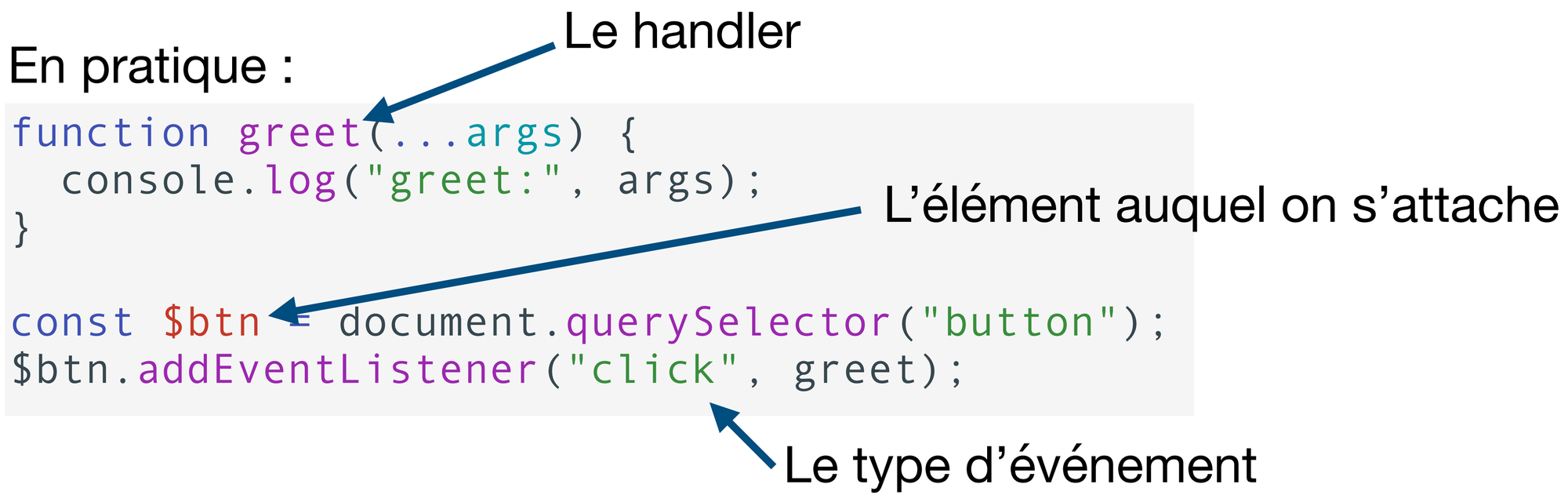
MDN - Event handling (overview)

Attacher un handler

Attacher ou détacher un handler :

```
EventTarget.addEventListener(type, handler, opts).
EventTarget.removeEventListener(type, handler, opts).
```

En pratique :




```
function greet(...args) {
  console.log("greet:", args);
}

const $btn = document.querySelector("button");
$btn.addEventListener("click", greet);
```

Attacher un handler

Méthode alternative

Attribut spécial **onevent** comme **EventTarget.onclick** :

- onevent est un attribut
- limitée à un seul handler
- plus simple à utiliser

```
function greet(...args) {
  console.log("greet:", args);
}

const $btn = document.querySelector("button");
// ajout du handler
$btn.onclick = greet;
// suprression
$btn.onclick = null;
```

On évite

Méthode inline HTML

```
<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

☣ À bannir (MDN) ☣

☠ le handler est une fonction ☠

`element.onclick = handlerFunct()`

- n’est exécuté qu’**une seule fois**
  - au moment de l’assignation
  - c’est son résultat qui sera assigné
- ceci n’a pas de sens (sauf curryfication)
- provoquera (généralement) une erreur

`element.onclick = handlerFunct`

- L’appel handlerFunct() sera exécuté à chaque occurrence de l’événement "click".

On aura besoin de fermetures -> on va faire de la programmation fonctionnelle.

La source

this dans les handlers

“When a function is used as an event handler, its **this** is set to the **element on which the listener is placed**”

this and event handler | MDN

```
const $btn = document.querySelector("button");

function greet(event) {
  // affiche le texte du premier bouton de la page
  // et deux attributs de l'événement "click"
  console.log(this.innerText, event.type, event.timeStamp);
}
$btn.addEventListener("click", greet);
```

☣️ this ≠ currentTarget ≠ target ☣️

Démo

http://lifweb.pages.univ-lyon1.fr/2024/CM2-DOM/exemples/gallerie.html

```
const $output = document.querySelector("#output");
const $div = document.querySelector("#container");
const $button = document.querySelector("button");

const handle = (event) => {
  $output.textContent += `Clicked ${event.currentTarget.tagName}\n`;
};

document.body.addEventListener("click", handle);
$div.addEventListener("click", handle);
$button.addEventListener("click", handle);
```

JavaScript et asynchrone

Plan

1. Événements JavaScript
2. Portée des variables et fermetures en JavaScript
3. Programmation asynchrone en JavaScript

Portée des variables JS

☠️ ~~var : portée fonction~~ ☠️

⚠️ let/const : portée lexicale (bloc) ⚠️

Portée des variables

👍 let/const, la portée de i est la boucle for

```
for (let i = 0; i < 3; i++) {
  console.log(i); // 0, then 1, then 2
}
console.log(i); // Uncaught ReferenceError: i is not defined
```

👎 var, la portée de i est la fonction englobante

```
for (var i = 0; i < 3; i++) {
  console.log(i); // 0, then 1, then 2
}
console.log(i); // affiche 3 😱
```

Déclaration de fonction

par instruction

```
// déclaration de funct
function funct(arg1, arg2, ..., argN) {
  // instructions
  return expression;
}

// appel
const res = funct(p1, p2, ..., pN);
```

Ici, on crée une fonction avec une instruction (statement 🇬🇧) function qui ne produit pas de valeur de retour (MDN).

Déclaration fonctionnelle

```
// NFE affectée à funct
const funct = function (arg1, arg2, ..., argN) {
  // instructions
  return expression;
};

// appel
const res = funct(p1, p2, ..., pN);
```

👉 Ici, on crée une fonction comme une expression avec laquelle on initialise une variable funct, c'est une Named Function Expression - (NFE) (javascript.info, MDN).

Expression fléchée

ou arrow functions, fat arrows ou lambdas (MDN)

```
// arrow avec bloc d'instructions et return explicite
const funct = (arg1, arg2, ..., argN) => {
  // instructions
  return expression;
};

// appel
const res = funct(p1, p2, ..., pN);
```

⚠️ Sans `return`, `funct` renvoie `undefined`. ⚠️

Expression fléchée

ou arrow functions, fat arrows ou lambdas (MDN)

```
// arrow affectée à funct
const funct = (arg1, arg2, ..., argN) => expression;

// appel
const res = funct(p1, p2, ..., pN);
```

Quasi-équivalente à la NFE suivante :

```
let funct = function(arg1, arg2, ..., argN) {
  return expression;
};
```

Et à l'expression du λ-calcul : $\lambda a_1 \ldots \lambda a_n.e$

`this` dans une fonction

⚠️ NFE et arrows ne sont **pas équivalentes**,

- une arrow a `this` dans sa fermeture statique, mais
- la NFE l'obtient dynamiquement.

"Arrow functions do not have `this`. If `this` is accessed, it is taken from the outside".

voir Arrow functions, the basics et Arrow functions revisited

`this` **dans une fonction**

> “Inside a function, the value of `this` depends on how the function is called. Think about `this` as a **hidden parameter of a function** - just like the parameters declared in the function definition, `this` is a binding that the language creates for you **when the function body is evaluated**.”

voir MDN: this > function context

 Si vous utilisez `this` dans un handler avec une arrow, `this` ne sera pas l’élément auquel on attache le handler. 

Exemple

```
const fNFE = function (x) {
  return console.log(`this=${this} id=${this.id} x=${x}`);
};
const fArrow = (x) => console.log(`this=${this} id=${this.id} x=${x}`);

const object = { id: 42, fNFE, fArrow };
object.fNFE(3); // this = object
object.fArrow(3); // this = globalThis

const testButton = document.querySelector("#test");
testButton.addEventListener("click", fNFE); // this = testButton
testButton.addEventListener("click", fArrow); // this = globalThis
testButton.click();
```

Résultat :

```
this=[object Object] id=42 x=3
this=[object Window] id=undefined x=3
this=[object HTMLButtonElement] id=test x=[object MouseEvent]
this=[object Window] id=undefined x=[object MouseEvent]
```

Les fermetures

closures en anglais

Terme utilisé en JS pour $(\lambda x.e)v$: l'expression e dans laquelle x prend la valeur à v

```
function hello(person) {
  return function (content) {
    return `Hi ${person}, ${content}`;
  };
}

const helloBuddy = hello("Buddy");
const helloGuys = hello("Guys");
console.log(helloBuddy("c'mon."));
console.log(helloBuddy("welcome."));
console.log(helloGuys("how do you do?"));
```

En λ-calcul

avec ++ pour la concaténation

revoir LIFLC - CM6 et LIFPF - CM1.

```
      hello  ≡ λp.λc.p++c
helloBuddy   ≡ (hello Buddy)
             ≡ (λp.λc.p++c) Buddy
             ≡ (λc.p++c)[p:=Buddy]

helloBuddyOK ≡ ((λc.p++c)[p:=Buddy])
          OK ≡ (p++c)[p:=Buddy, c:=OK]
```

Exemple de fermeture

avec le DOM

```
function fabriqueRedimensionneur(taille) {
  return function () {
    // 🚨 taille est capturée 🚨
    document.body.style.fontSize = taille + "px";
  };
}

const taille12 = fabriqueRedimensionneur(12);
const taille14 = fabriqueRedimensionneur(14);
document.querySelector("#t-12").onclick = taille12;
document.querySelector("#t-14").onclick = taille14;
```

💡 La variable `taille` est **liée**.

Exemple de fermeture #2

déjà vu en TP

```
const $anchors = document.querySelectorAll("#images-list li > a");
const $display = document.querySelector("#image-container");

for (const $a of $anchors) {
  $a.addEventListener("click", function clickButton(event) {
    // 🚨 $a est capturée car portée block 🚨
    event.preventDefault();
    const $img = document.createElement("img");
    $img.src = $a.href;
    $display.replaceChildren($img);
  });
}
```

Exemple Tableau de fonctions

```
const fsMap = [1, 2, 3].map((x) => (y) => y + x);
console.log(fsMap.map((f) => f(1)));
// [ 2, 3, 4 ] 👍

const fsConst = [];
for (const a of [1, 2, 3]) fsConst.push((x) => x + a);
console.log(fsConst.map((f) => f(1)));
// [ 2, 3, 4 ] 👍

const fsVar = [];
for (var a of [1, 2, 3]) fsVar.push((x) => x + a);
console.log(fsVar.map((f) => f(1)));
// [ 4, 4, 4 ] 🤯
```

Immediately Invoked Function Expression (IIFE)

🤔 `let/const` n'ont été introduits qu'avec ES6 / ECMAScript 2015, comment faisait-on depuis 1995 ?

```
(function () {
  var x1 = ... ;
  var x2 = ... ;
  /* … */
  return ...
})();
```

"An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined"

MDN

📜 Permet de limiter la portée des variables avec une closure : méthode avant la standardisation des modules ESM (ES6) pour encapsuler.

Une IIFE pour encapsuler des var

```
const functs = [];
for (var a of [1, 2, 3])
  (function (y) {
    functs.push((x) => x + y);
  })(a); // IIFE

console.log(functs.map((f) => f(1)));
// [ 2, 3, 4 ] 😌
```

Ici, on introduit y pour capturer a via l'IIFE : $(\lambda x.x+y)[y:=a]$

“Closures and classes behave differently in JavaScript with a fundamental difference: closures support encapsulation, while JavaScript classes don’t support encapsulation. Closures offer simplicity, since we don’t have to worry about the context that `this` is referring to.”

How to decide between classes v. closures in JavaScript

Class vs. Closure

Voir class-vs-closure.js

Class, style ES6+

```
class CountClass {
  constructor() {
    this.count = 0;
  }
  up() {
    return ++this.count;
  }
}
// ! new crée un nouvel objet
const obj1 = new CountClass();
const obj2 = new CountClass();
console.log(obj1.up()); // 1
console.log(obj1.up()); // 2
console.log(obj2.up()); // 1
```

Closure, style FP

```
function countClosure() {
  // closure
  let count = 0;

  return () => ++count;
}

// ! pas de new
// ! pas de this crée
const funct1 = countClosure();
const funct2 = countClosure();
console.log(funct1()); // 1
console.log(funct1()); // 2
console.log(funct2()); // 1
```

JavaScript et asynchrone

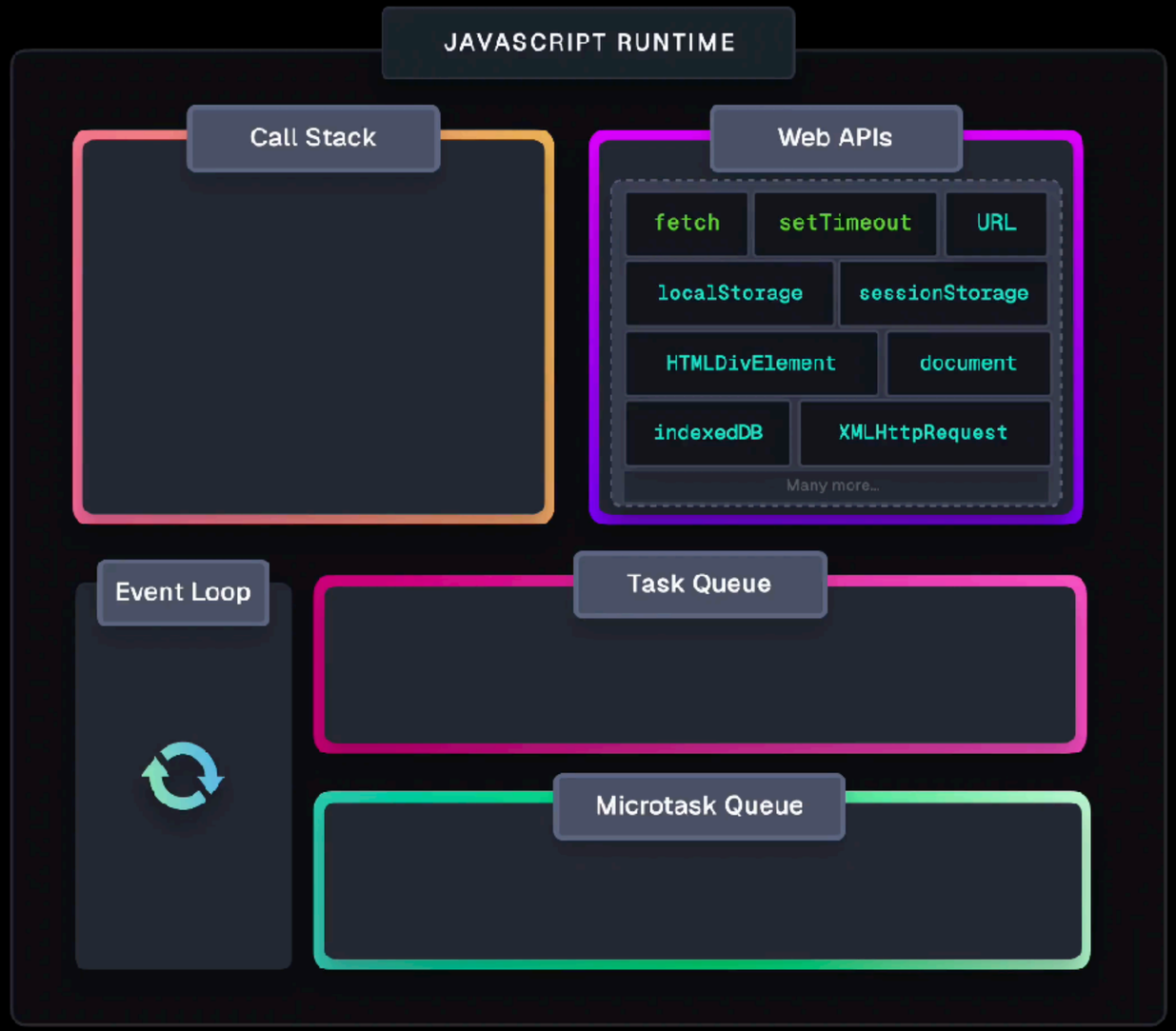
Plan

1. Événements JavaScript
2. Portée des variables et fermetures en JavaScript
3. Programmation asynchrone en JavaScript

Programmation événementielle ≋ asynchrone

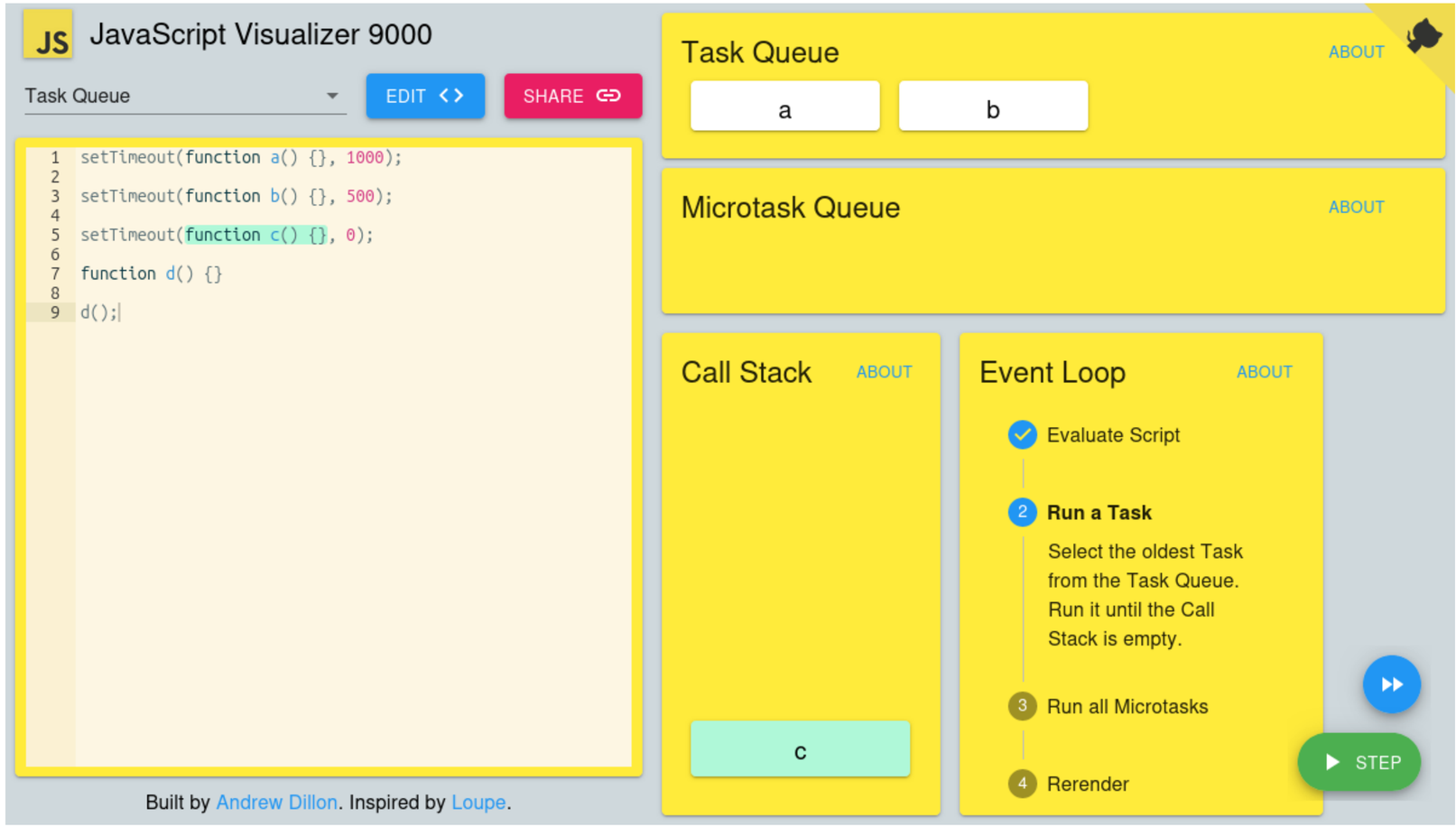
La boucle d’événement

https://www.lydiahallie.com/blog/event-loop



JAVASCRIPT VISUALIZER 9000

https://jsv9000.app/



Handler onevent

```
const $output = document.querySelector("#output");
const $input = document.querySelector("#input");
const $eval = document.querySelector("#eval");

function work() {
  // 💡 ici, $input.value n'est pas dans la closure
  // la variable n'est PAS dans la fermeture
  $output.textContent += `${2n ** BigInt($input.value)}\n`;
}
$eval.addEventListener("click", work);
```

exemple

Timeouts

au alarmes

demo

```
console.log("Start"); // (A)
setTimeout(
  // (T1)
  () => console.log("Call back #1"), // (CB1)
);
console.log("Middle"); // (B)
setTimeout(
  // (T2)
  () => console.log("Call back #2"), // (CB2)
);
console.log("End"); // (C)
```

🧠 `setTimeout(fct, delay=0, ...params)` rend `fct` asynchrone : les alarmes sont déclenchées au début du tour suivant de la boucle d'événements.

Attention au code bloquant

```
const s = new Date().getSeconds();
setTimeout(function () {
  console.log(`Elapsed: ${new Date().getSeconds() - s} seconds`);
}, 0);

while (true) {
  if (new Date().getSeconds() - s >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

⚠️ Ici, l’alarme sera déclenchée après au moins deux secondes (exemple).

💀 Ne réalisez pas vous-même cette cascade.

Fonctions Asynchrones

Voir sur javascript.info, MDN ou Node.js.

📝 Les fonctions synchrones sont bloquantes :

- le thread d'exécution attend la fin du traitement

💡 Les fonctions asynchrones ne sont pas bloquantes

- Le thread d'exécution continue après l'appel.
- On passe en paramètre la suite du traitement :
  - Paramètre appelé handler ou callback.
- 🔁 Le traitement sera déclenché ultérieurement :
  - dans un autre tour de la boucle d'événements.

De l’asynchrone partout en JS

👀 Les gestionnaires d’événements pour HTMLElement, Document, ou Window.

👉 Les applications AJAX (Asynchronous JavaScript + XML) Fetch API (et XHR - XMLHttpRequest).

👉 La bibliothèque standard Node.js comme FS - File System, HTTP ou l’API stream.

Mais aussi Web Workers, l’API Crypto, etc.

Passage de paramètre en asynchrone

Version callback

```
function work(x) {
  console.info(`work(${x})`);
  return x ** 2;
}

function asyncWork(n, callback) {
  return setTimeout(() => {
    const r = work(n);
    callback(r);
  }, 3000);
}

const r1 = asyncWork(3, (r) => console.log(r));
console.log(r1);
```

✅ on a bien exécuté `(r) => console.log(r)` avec la valeur de retour de **`work(n)`** après 3000 ms (exemple) ! Attention `r1` est l'alarme pas le résultat

Enchainement de callback

Aussi nommé *callback hell*

```
function asyncWork(n, callback) {
  return setTimeout(() => callback(n ** 2), 3000);
}

asyncWork(3, (r1) => {
  console.log(r1);
  asyncWork(r1, (r2) => {
    console.log(r2);
    asyncWork(r2, (r3) => console.log(r3));
  });
});
```

✅ On parvient à chaîner les `asyncWork`, mais il faut imbriquer des fonctions

Les promesses et `async/await` résoudront ce problème

JavaScript et asynchrone

Plan

1. Événements JavaScript
2. Portée des variables et fermetures en JavaScript
3. Programmation asynchrone en JavaScript

Conclusion

🗒️ Asynchrone = exécution de fonction dans le futur

- Ajout à la queue sur déclenchement d’un event.
- Puis traitement par l’event loop.

⚠️ return renvoie une valeur synchrone :

- ❎ ne peut pas renvoyer une valeur future :
  - seulement une promesse de valeur.
- ✅ permet d’interrompre le flux d’exécution :
  - comme throw, mais pour un succès.

💡 On doit passer la suite du traitement en paramètre.

- Principe du Continuation Passing Style (CPS).