Programmation Asynchrone Callbacks, Promises, async/await, Fetch

Dernière mise à jour : 30/01/2025 18:32

Callbacks (fonctions de rappel)

- Fonctions JavaScript sont des objet de 1ère classe (*first class objects*)
  - peuvent être affectées à des variables
  - passées en paramètre

```
function runThis(otherFctn) {
    console.log("Running ...");
    otherFctn();
}
```

le paramètre **`otherFctn`** de **`runThis`** est une fonction

quand la fonction **`runThis`** est exécutée elle rappelle (*calls back*) la fonction **`otherFctn`**

Callbacks (fonctions de rappel)

- Fonctions JavaScript sont des objet de 1ère classe (*first class objects*)
  - peuvent être affectées à des variables
  - passées en paramètre

```
function runThis(otherFctn) {
    console.log("Running ...");
    otherFctn();
}

function function1() {
    console.log("Function 1...");
}

runThis(function1);
```

la fonction de rappel (*callback*) est la fonction **`function1`**

```
Running ...
Function 1...
```

Callbacks (fonctions de rappel)

- Fonctions JavaScript sont des objet de 1ère classe (*first class objects*)
  - peuvent être affectées à des variables
  - passées en paramètre

```
function runThis(otherFctn) {
    console.log("Running ...");
    otherFctn();
}

function function1() {
    console.log("Function 1...");
}

runThis(function1);

runThis(
   function () {
       console.log("Function 2...");
   }
);
```

la fonction de rappel (*callback*) est la fonction **`function1`**

```
Running ...
Function 1...
```

la fonction de rappel (*callback*) est une fonction anonyme

```
Running ...
Function 2...
```

Callbacks (fonctions de rappel)

- Fonctions JavaScript sont des objet de 1ère classe (*first class objects*)
  - peuvent être affectées à des variables
  - passées en paramètre

```
function runThis(otherFctn) {
    console.log("Running ...");
    otherFctn();
}

function function1() {
    console.log("Function 1...");
}

runThis(function1);

runThis(
   function () {
       console.log("Function 2...");
   }
);

runThis(
   () => { console.log("Function 3...") }
);
```

la fonction de rappel (*callback*) est la fonction **`function1`**

```
Running ...
Function 1...
```

la fonction de rappel (*callback*) est une fonction anonyme

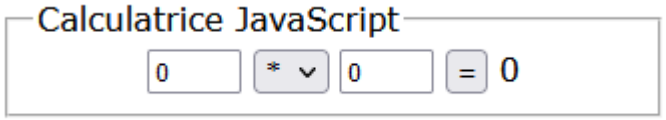
```
Running ...
Function 2...
```

la fonction de rappel est une fonction fléchée (*arrow function*) *

```
Running ...
Function 3...
```

* une autre manière d'écrire des fonctions anonymes

- gestionnaires d'événement fonctions de rappel d'un type particulier




```
<html>
 <head>
    ...
 </head>
 <body>
   ...
   <button id="btnCalculer">=</button>
   ...
   <script src="./js/calculatrice.js"></script>
</body>
```

```
let btnCalculer = document.querySelector("#btnCalculer");

btnCalculer.addEventListener("click", function() {
   let operande1 = parseFloat(document.querySelector("#op1").value);
   let operande2 = parseFloat(document.querySelector("#op2").value);
   let operateur = document.querySelector("#operateur").value;
   switch (operateur) {
      ...
   }
   document.querySelector("#resultat").innerHTML = res;
 }
);
```

Lorsque l'utilisateur effectue une action (ex click sur bouton), un événement est mis en file d'attente.

Si un événement a été crée la fonction callback associée sera exécutée par la boucle des événements du moteur JavaScript

La boucle d'événement tire principalement son nom de son implémentation. Celle-ci ressemble à :

fonction synchrone qui attend un message même s'il n'y en a aucun à traiter.

```
while (queue.attendreMessage()){
  queue.traiterProchainMessage();
}
```

Chaque message sera traité complètement avant tout autre message. (la fonction ne peut être interrompue, contrairement à d'autres langages, comme Java, C#, Go, Rust… supportant, le *multhreading*)

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

Synchrone vs. Asynchrone

Programmation synchrone

- Les traitements sont exécutés de manière séquentielle les uns après les autres

```
let img1 = charger("im1.jpg");
afficher(img1);
let img2 = charger("im2.jpg");
afficher(img2);
let img1 = charger("im3.jpg");
afficher(img1);
```

Les images sont chargées
et affichées les unes après les autres

Programmation asynchrone

- possibilité de démarrer une traitement sans que le traitement précédent ait été terminé

```
chargerAsync("im1.jpg")
     .then(afficher);
chargerAsync("im2.jpg")
     .then(afficher);
chargerAsync("im3.jpg")
     .then(afficher);
```

Les images sont chargées en parallèle
et affichées dès que leur chargement est terminé

Fonction asynchrone

- fonction asynchrone
  - permet de démarrer une opération (potentiellement longue) et rend la main immédiatement, afin que le programme puisse continuer et puisse réagir aux autres évènements
  - recevoir une notification à la fin de l'opération, pour mettre en place un éventuel post-traitement en fonction du résultat

Comment mettre cela en œuvre en JavaScript ?

les fonctions callback ont un rôle central dans la gestion de la programmation asynchrone en JS

Le post-traitement est une fonction de rappel (*callback*) passée en paramètre de la fonction qui effectue l'opération asynchrone

exemple : la fonction globale `setTimeout()` qui définit un minuteur qui exécute une fonction ou un code donné après la fin du délai indiqué.

```
setTimeout( traitement, 5000); //fonction asynchrone
console.log("l'exécution continue");
```

le traitement asynchrone est délégué à un objet, le post-traitement est un gestionnaire d'événements* (*event handler*) associé à l'objet

* ici les événements sont déclenchés non par une action de l'utilisateur, mais par un changement d'état de l'objet)

exemple : un objet `FileReader` permet à des applications web de lire le contenu de fichiers de façon asynchrone.

fonctions callback appelées à la fin de l'opération asynchrone

```
const reader = new FileReader();
reader.addEventListener('load', traiterLeFichier);
reader.addEventListener('error', traiterErreur);
...
reader.readAsDataURL(selectedFile); //fonction asynchrone
console.log("l'exécution continue");
```

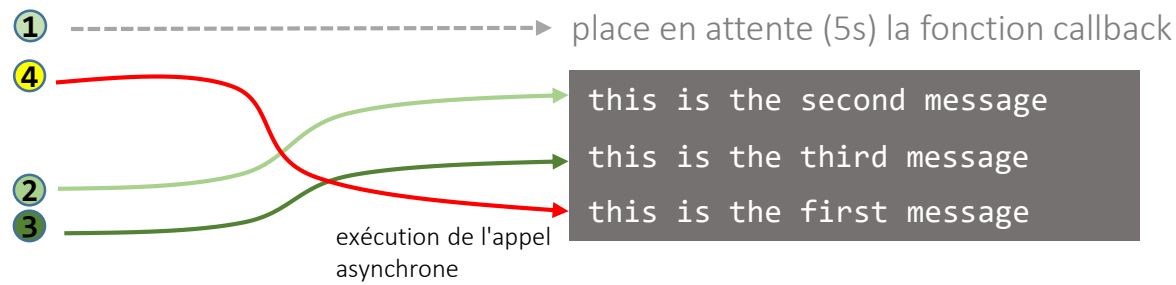
JavaScript et Programmation asynchrone

- exemple `setTimeout(callback, delay)`

délai (en millisecondes)

la fonction exécutée après que le délai soit écoulé

```
setTimeout(
    () => console.log("this is the first message") ,
    5000
);
console.log("this is the second message");
console.log("this is the third message");
```




```
setTimeout(
    () => console.log("this is the first message") ,
    5000
);
setTimeout(
    () => console.log("this is the second message") ,
    3000
);
setTimeout(
    () => console.log("this is the third message") },
    1000
);
```

JavaScript et Programmation asynchrone

```
function fonction1(mess) {
    console.log(mess);
}

function fonction2(mess, nb) {
    console.log("début iteration")
    for (let i = 0; i < nb; i++) {
        if (i % 500_000_000 === 0) {
            console.log("*");
        }
    }
    fonction1(mess);
}

setTimeout(
    () => console.log("execution callback du timer"),
    100
);
fonction1("Hello");
fonction2("World", 2_000_000_000);
```

3 7

5

affiche * tous les 500 millions d'itérations :
(environ toutes les 371 ms)

6

1 ----------▸ place en attente (0.1s) la fonction callback

8

2

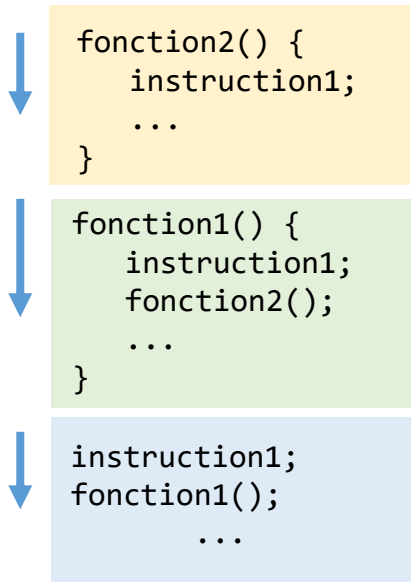
4

```
Hello
debut itération
*
*
*
*
World
execution callback du timer
```

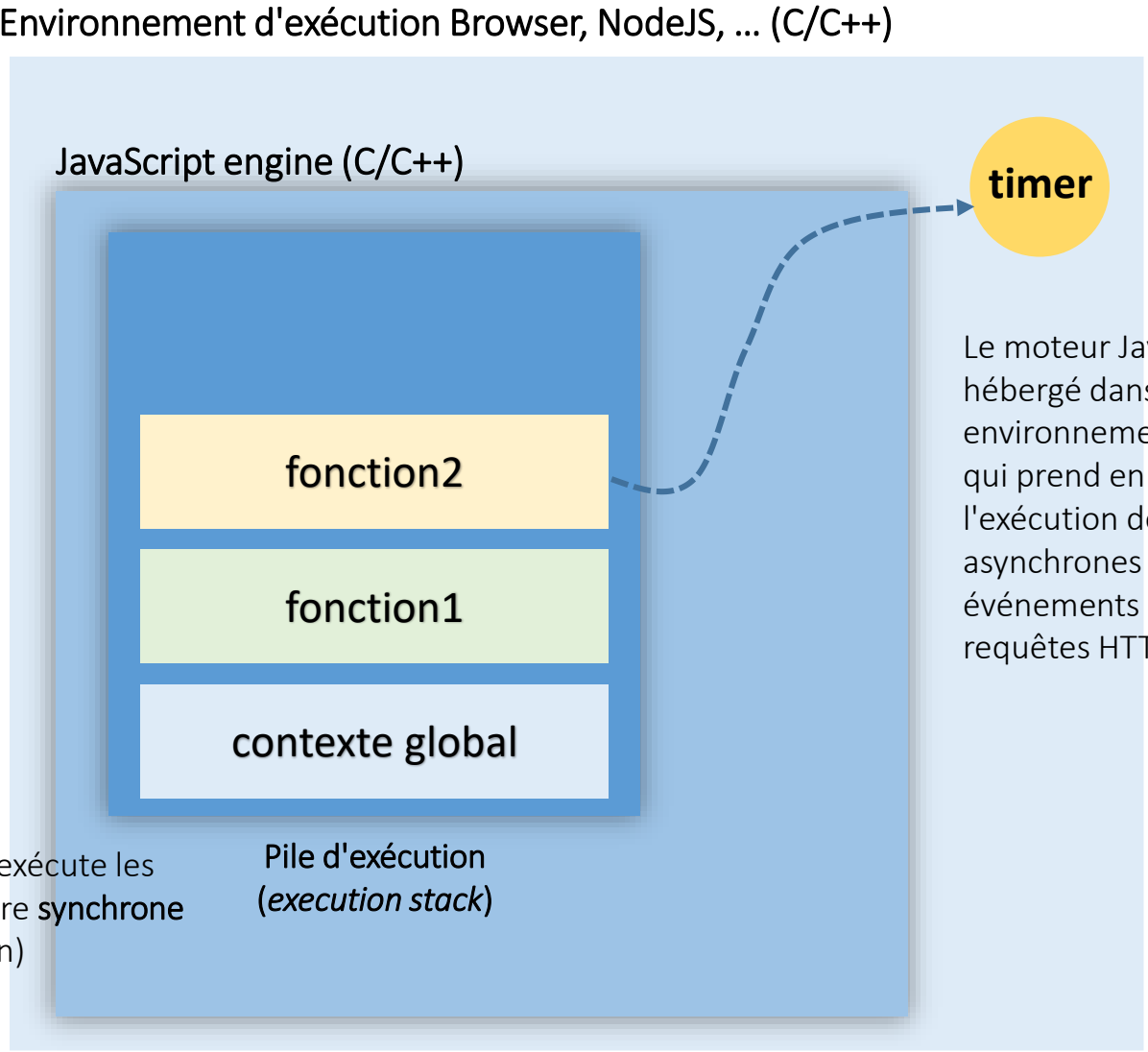
le code asynchrone n'est exécuté qu'une fois que l'exécution du code synchrone terminée

JavaScript et Programmation asynchrone

- JavaScript par essence est un langage synchrone avec un seul flot d'exécution (*single threaded*)
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript



Le moteur JavaScript exécute les instructions de manière **synchrone** (via sa pile d'exécution)



Le moteur JavaScript est hébergé dans un environnement d'exécution qui prend en charge l'exécution des opérations asynchrones (timers, événements d'interaction, requêtes HTTP...)

JavaScript et Programmation asynchrone

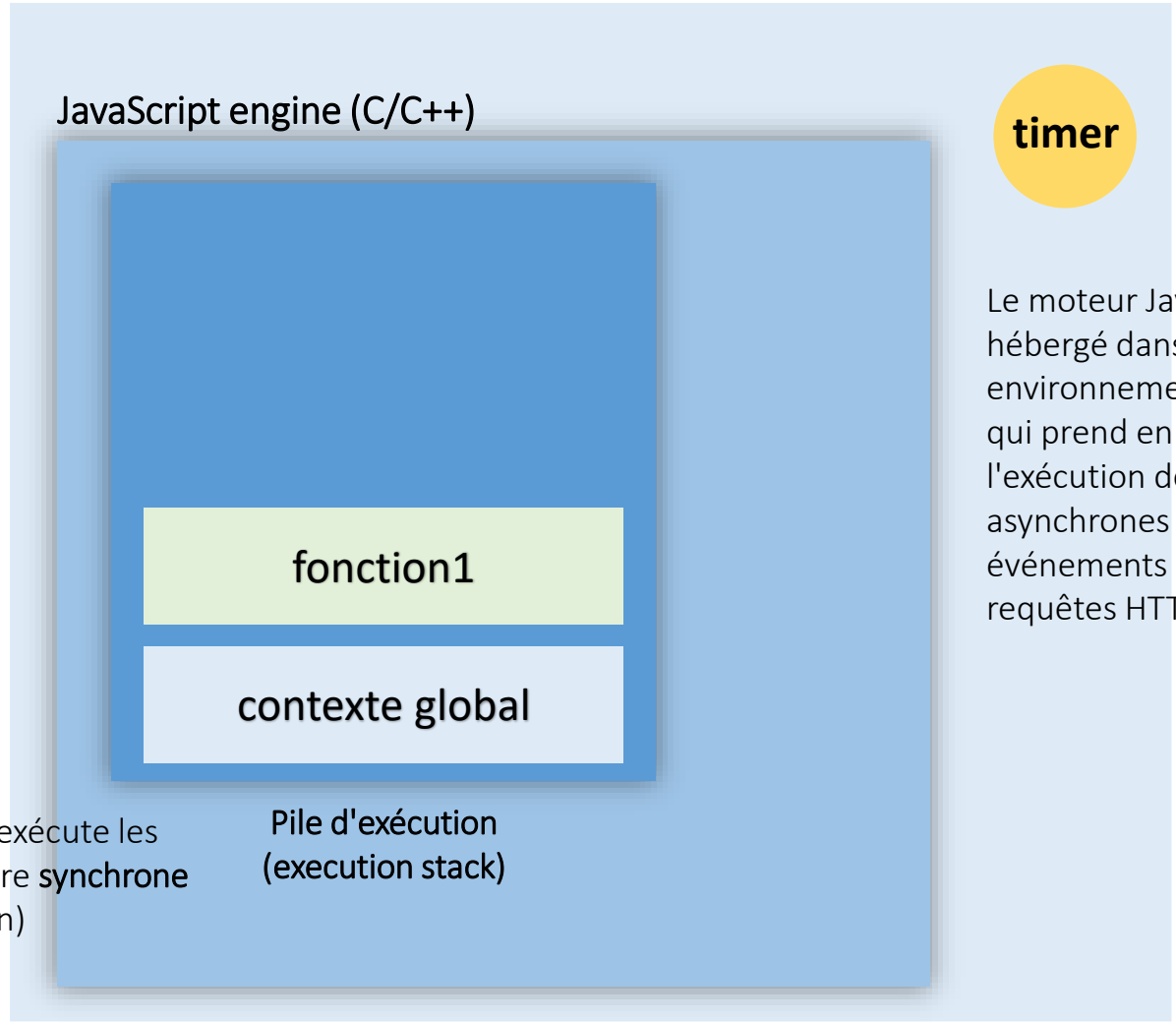
- JavaScript par essence est un langage synchrone et single threaded
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript

```
fonction2() {
   instruction1;
   ...
}
```

```
fonction1() {
   instruction1;
   fonction2();
   ...
}
```

```
instruction1;
fonction1();
      ...
```

Le moteur JavaScript exécute les instructions de manière **synchrone** (via sa pile d'exécution)



Le moteur JavaScript est hébergé dans un environnement d'exécution qui prend en charge l'exécution des opérations asynchrones (timers, événements d'interaction, requêtes HTTP…)

JavaScript et Programmation asynchrone

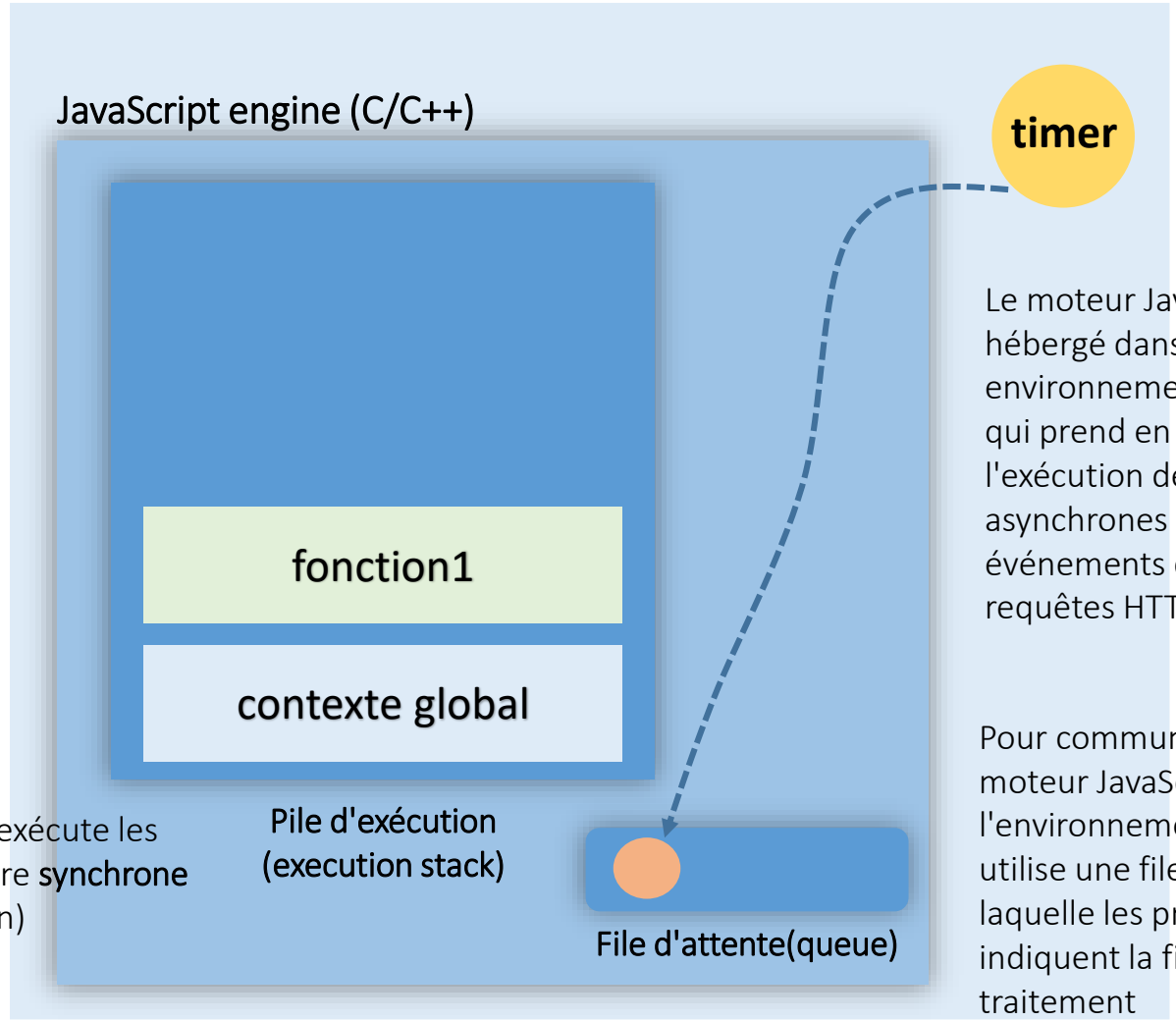
- JavaScript par essence est un langage synchrone et single threaded
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript

```
fonction2() {
   instruction1;
   ...
}
```

```
fonction1() {
   instruction1;
   fonction2();
   ...
}
```

```
instruction1;
fonction1();
       ...
```

Le moteur JavaScript exécute les instructions de manière **synchrone** (via sa pile d'exécution)



Le moteur JavaScript est hébergé dans un environnement d'exécution qui prend en charge l'exécution des opérations asynchrones (timers, événements d'interaction, requêtes HTTP…)

Pour communiquer avec le moteur JavaScript, l'environnement d'exécution utilise une file d'attente dans laquelle les processus externes indiquent la fin de leur traitement

JavaScript et Programmation asynchrone

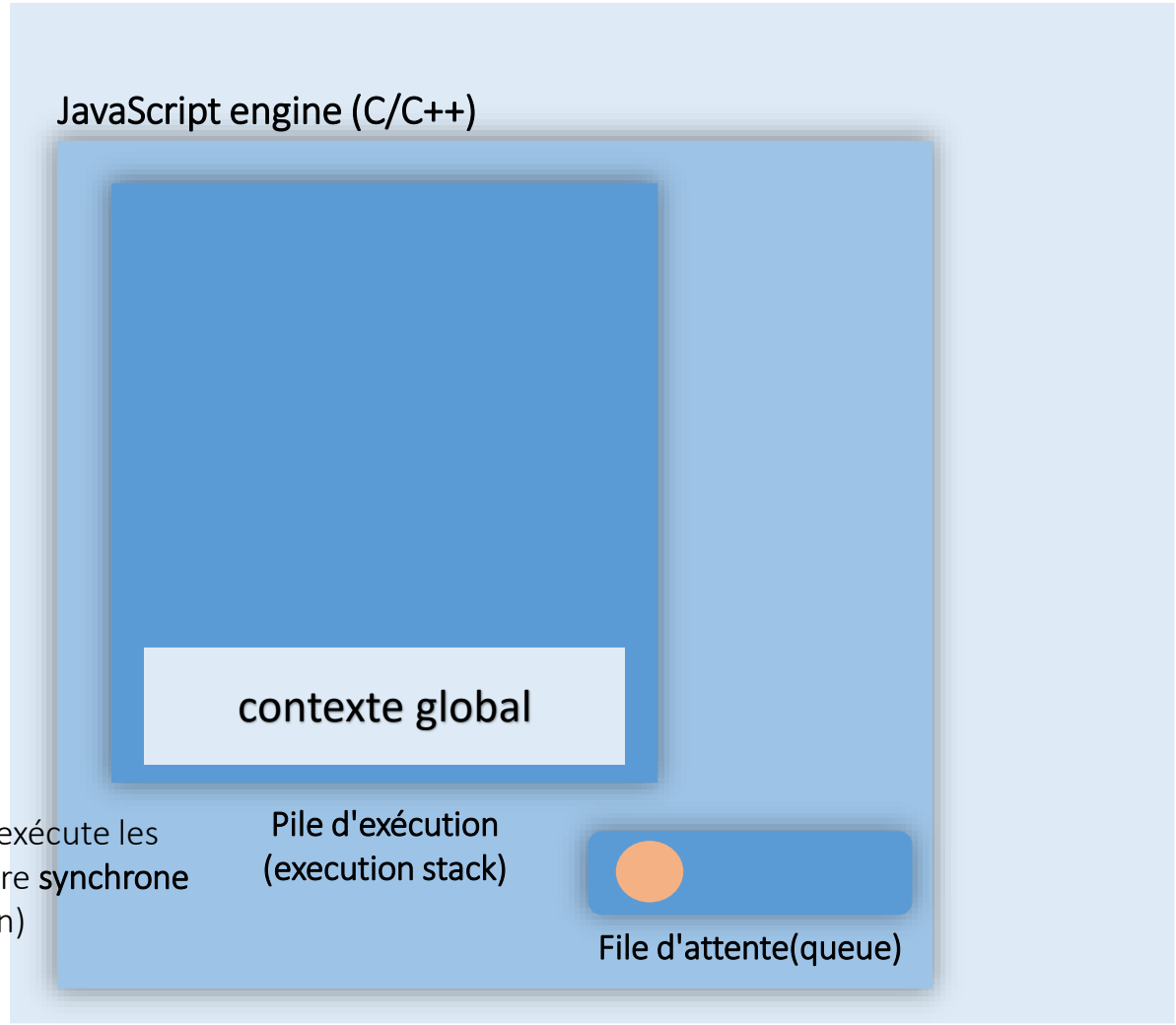
- JavaScript par essence est un langage synchrone et single threaded
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript

```
fonction2() {
   instruction1;
   ...
}
```

```
fonction1() {
   instruction1;
   fonction2();
   ...
}
```

```
instruction1;
fonction1();
       ...
```

Le moteur JavaScript exécute les instructions de manière **synchrone** (via sa pile d'exécution)



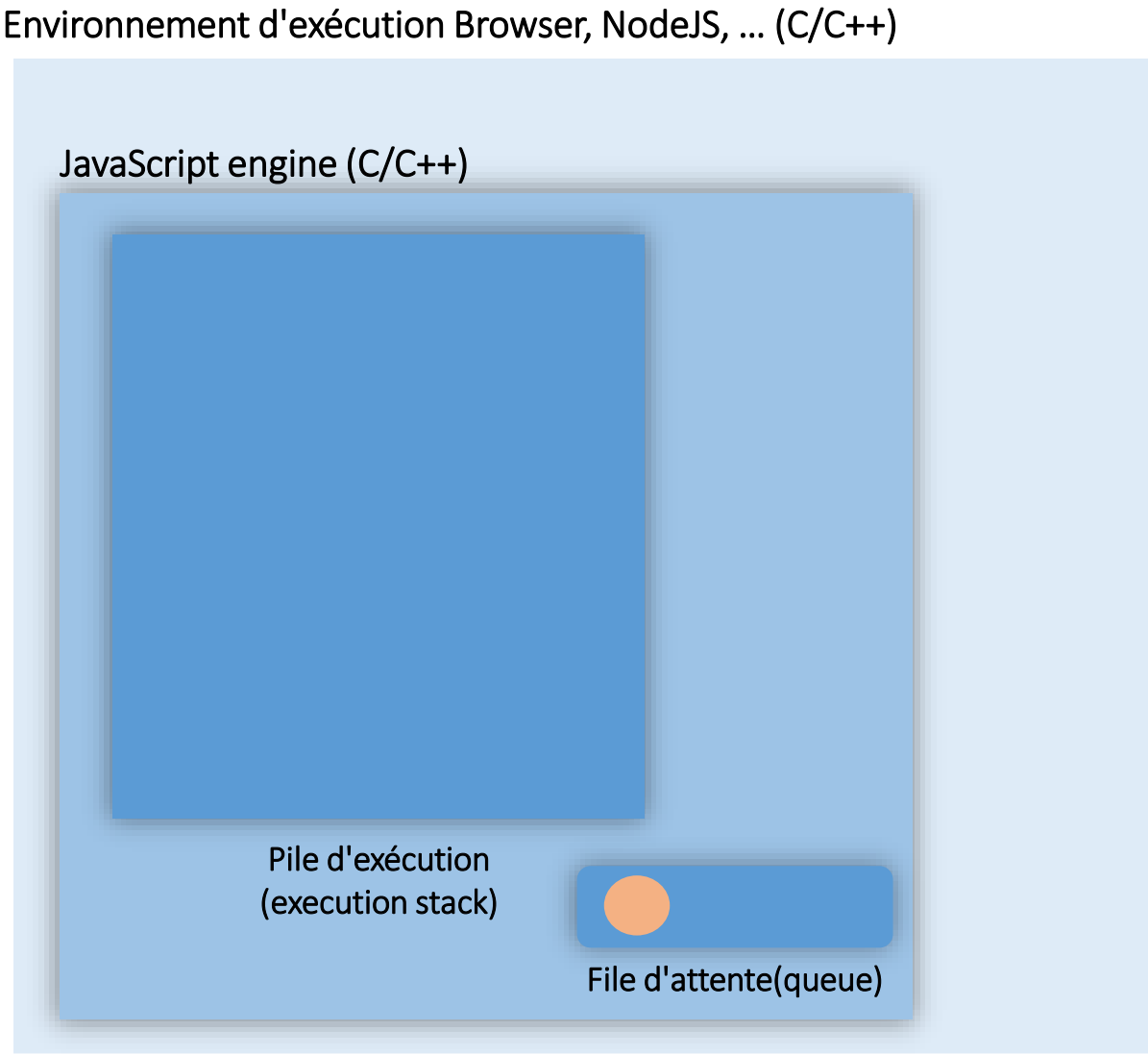
JavaScript et Programmation asynchrone

- JavaScript par essence est un langage synchrone et single threaded
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript

```
fonction2() {
   instruction1;
   ...
}
```

```
fonction1() {
   instruction1;
   fonction2();
   ...
}
```

```
instruction1;
fonction1();
       ...
```



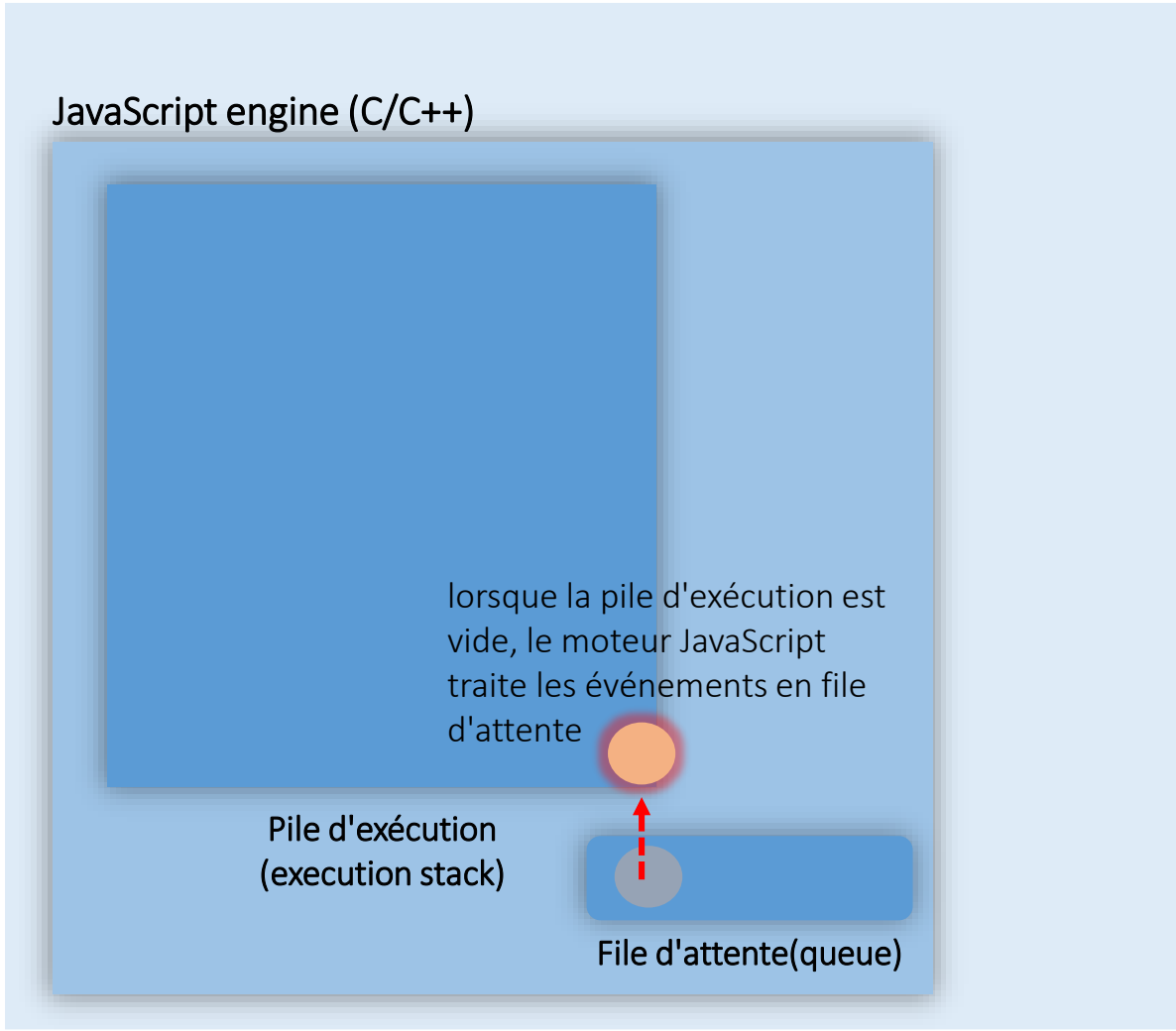
JavaScript et Programmation asynchrone

- JavaScript par essence est un langage synchrone et single threaded
  - il ne fait qu'une chose à la fois
  - les traitements asynchrones sont effectués en dehors du moteur d'exécution JavaScript

```
fonction2() {
   instruction1;
   ...
}
```

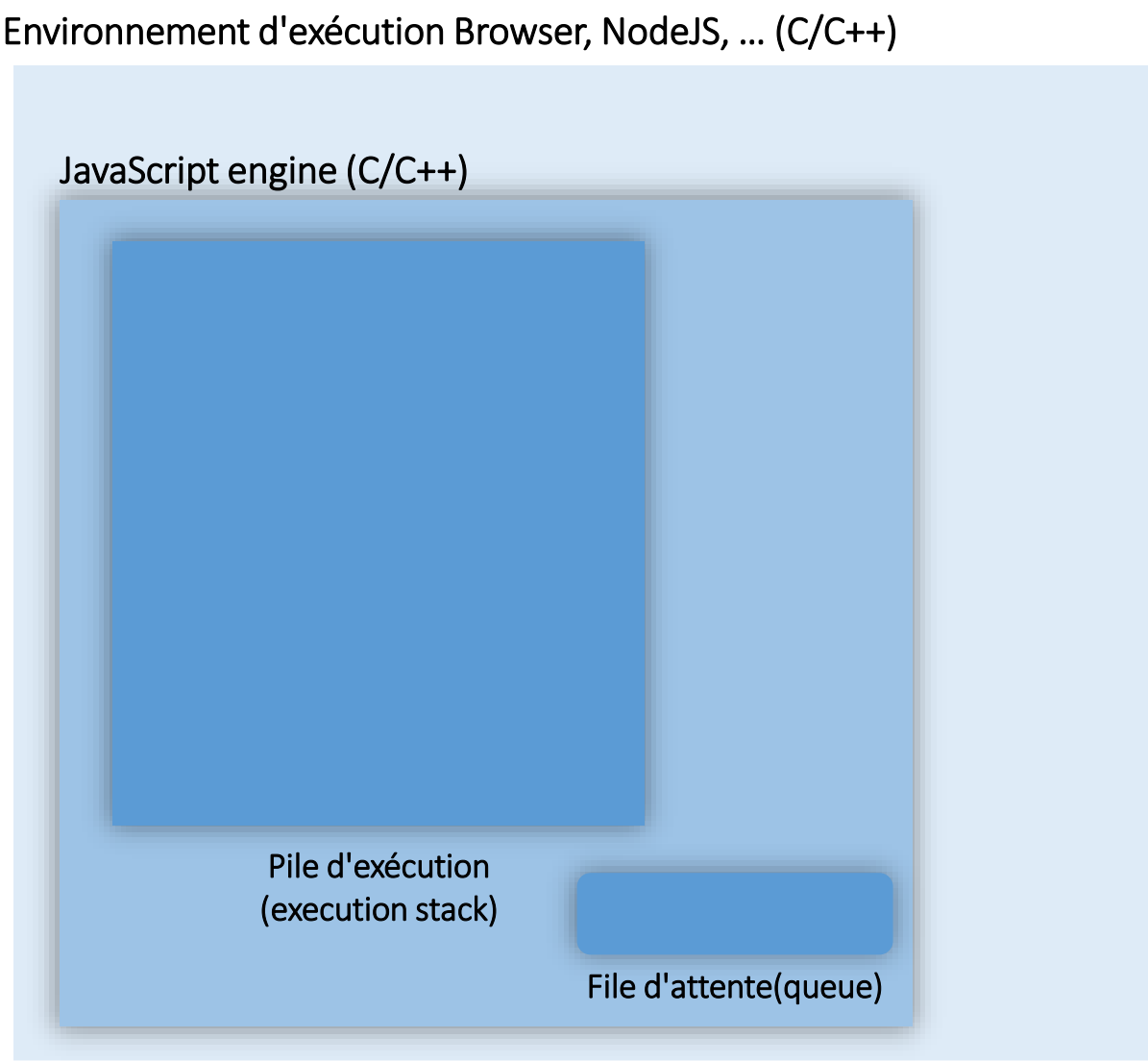
```
fonction1() {
   instruction1;
   fonction2();
   ...
}
```

```
instruction1;
fonction1();
       ...
```



JavaScript et Programmation asynchrone

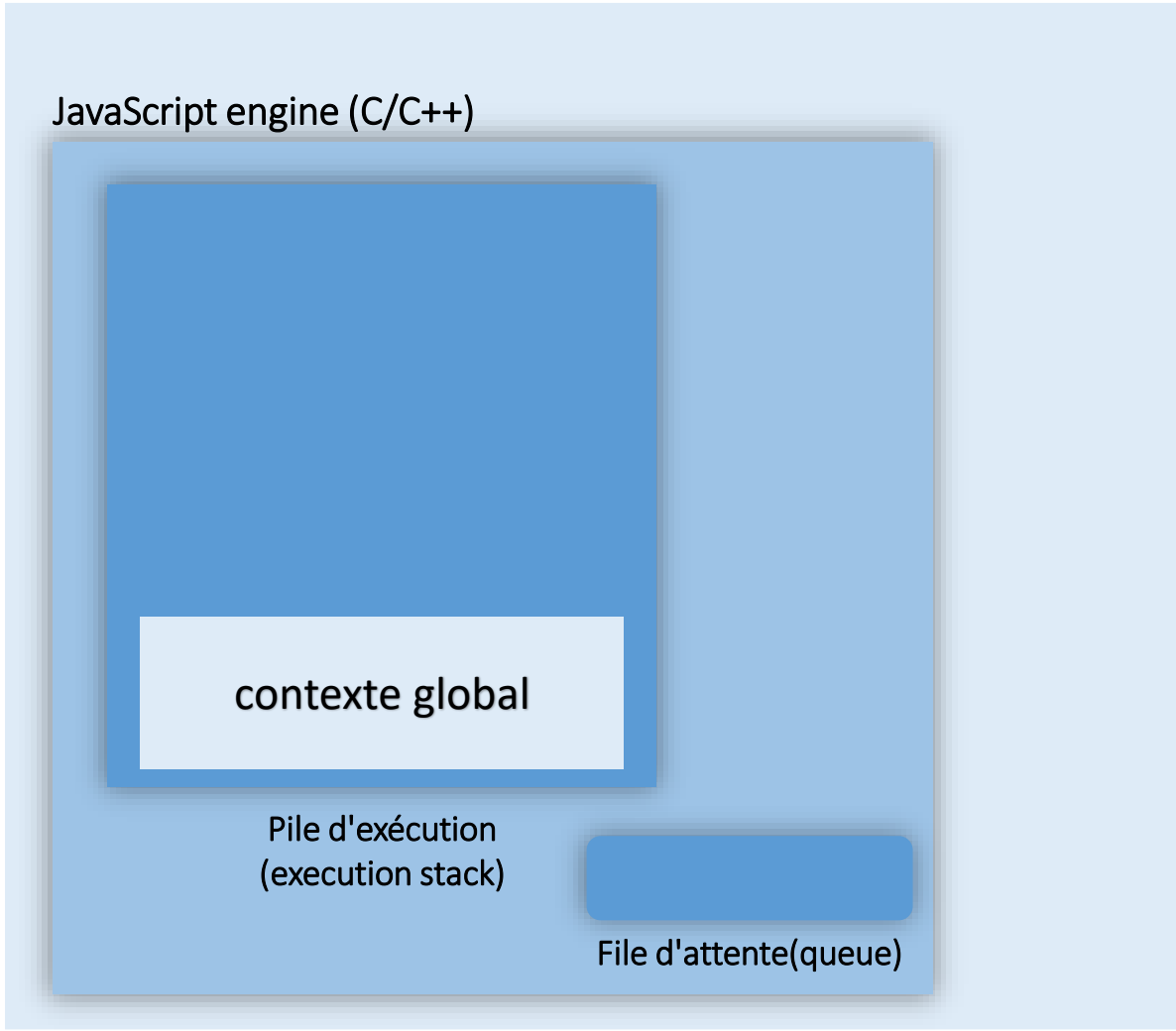
```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer")
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```



JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer")
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

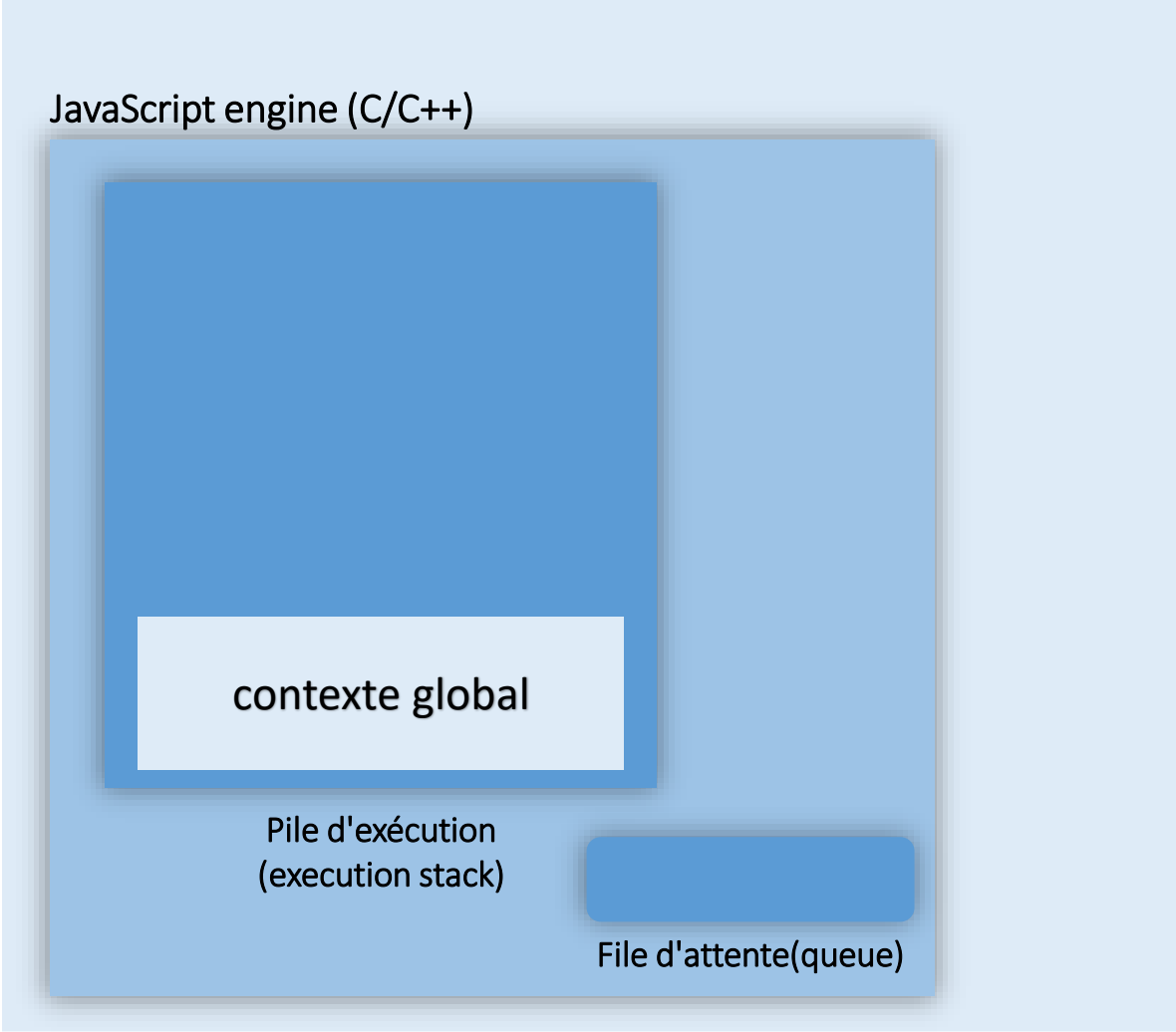
Environnement d'exécution Browser, NodeJS, … (C/C++)



JavaScript et Programmation asynchrone

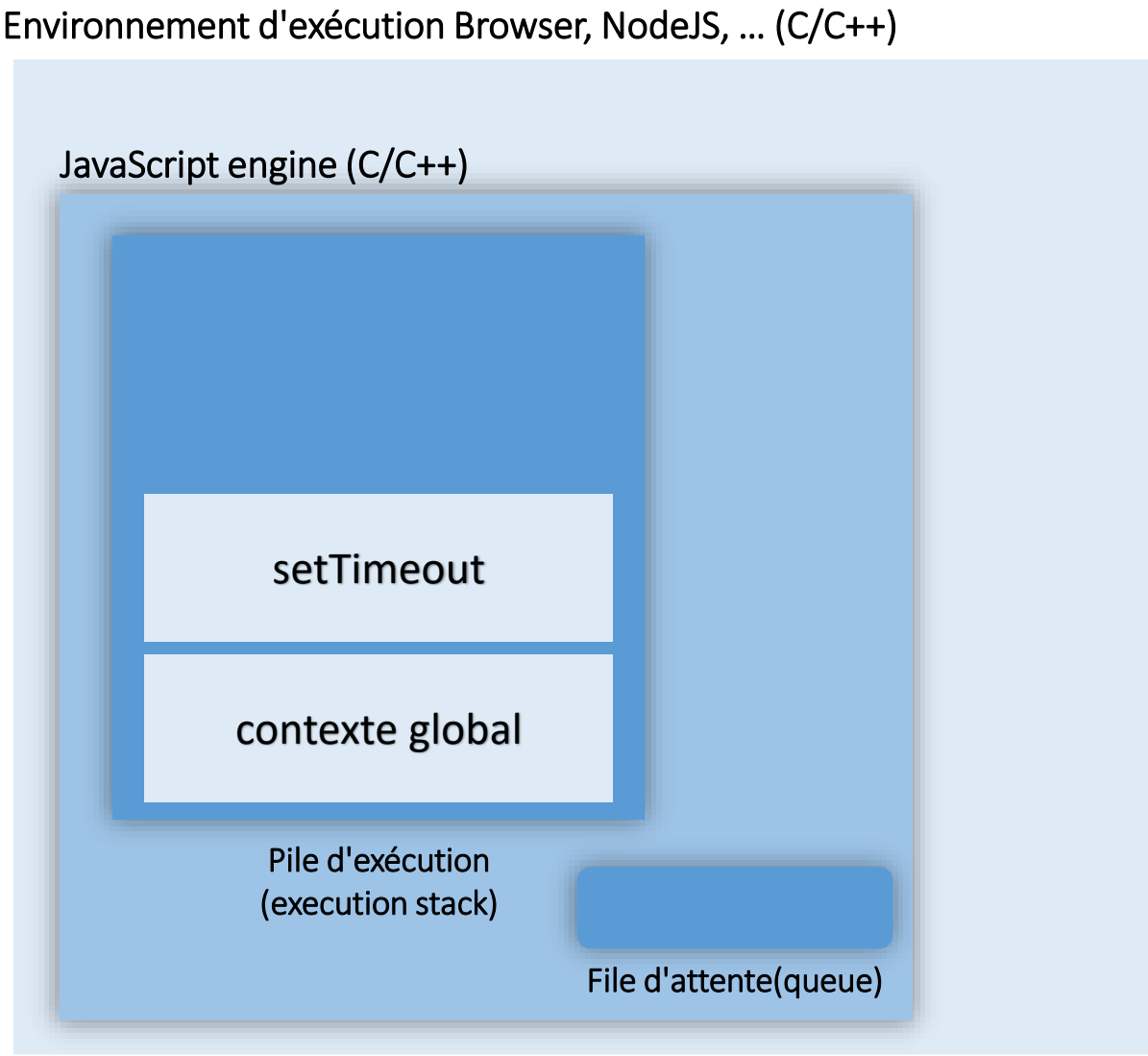
```
1  function fonction1(mess) {
2      console.• log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.• log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

Environnement d'exécution Browser, NodeJS, … (C/C++)



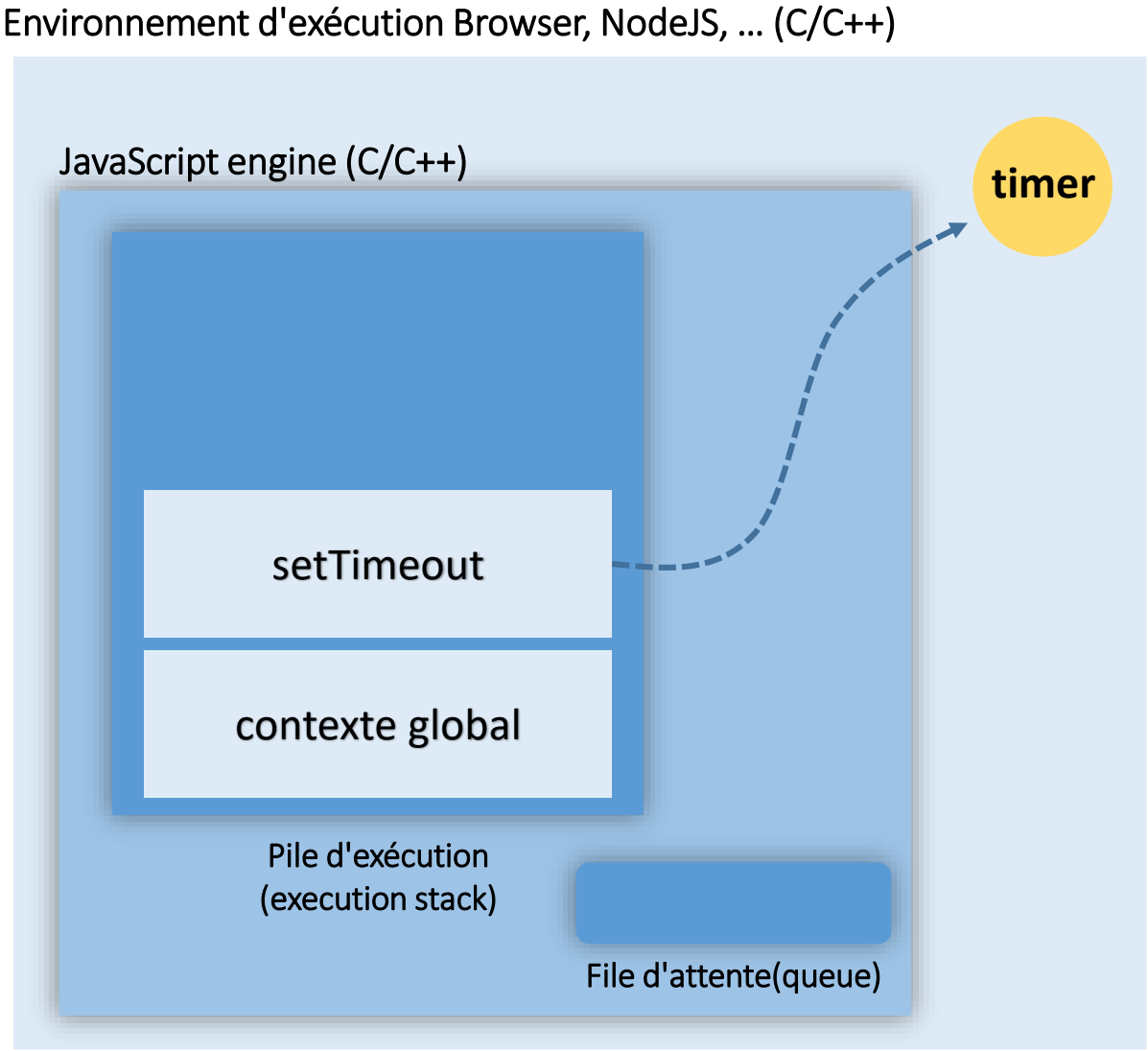
JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.● log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.● log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```



JavaScript et Programmation asynchrone

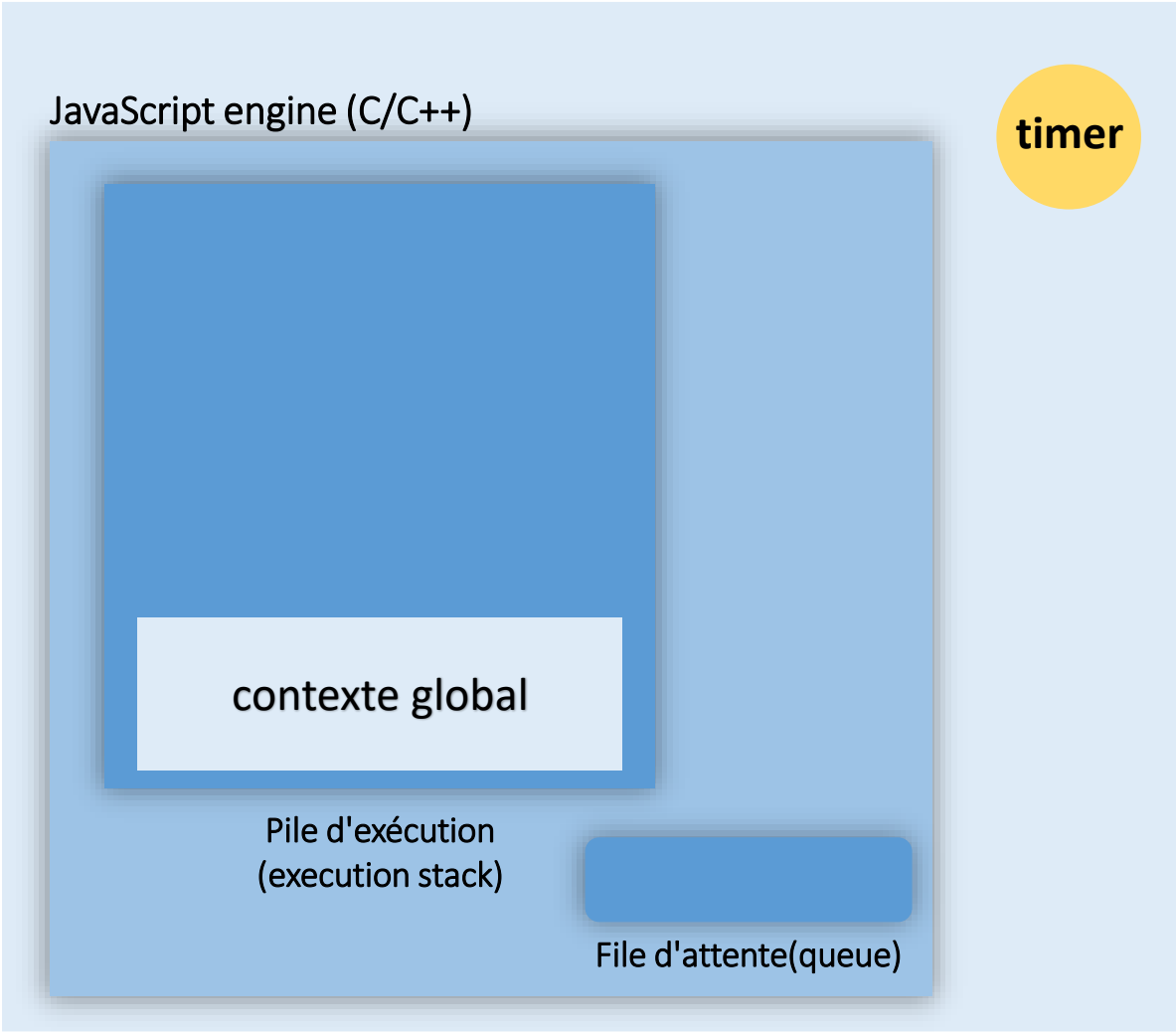
```
1  function fonction1(mess) {
2      console.• log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.• log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```



JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.• log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.• log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

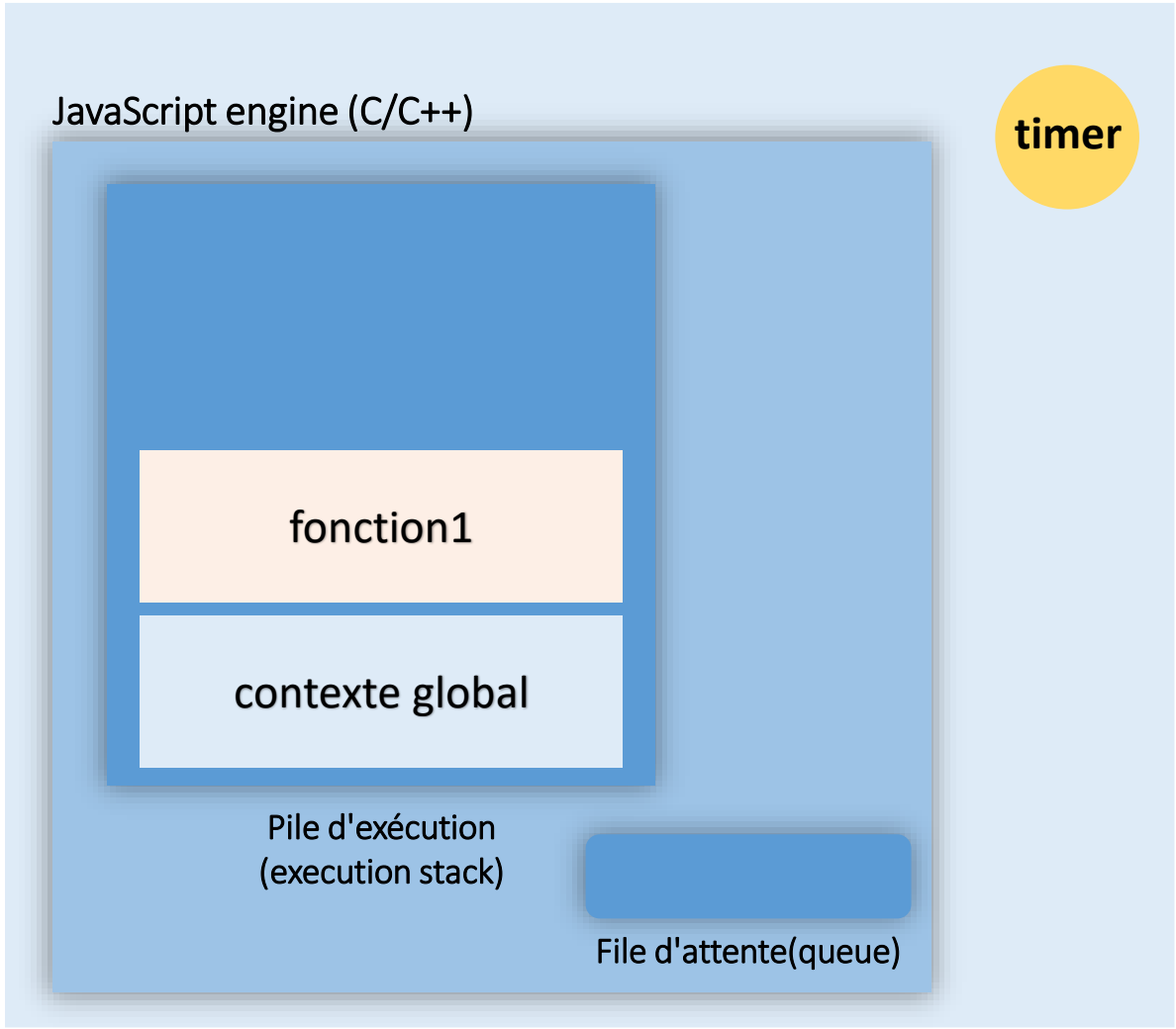


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

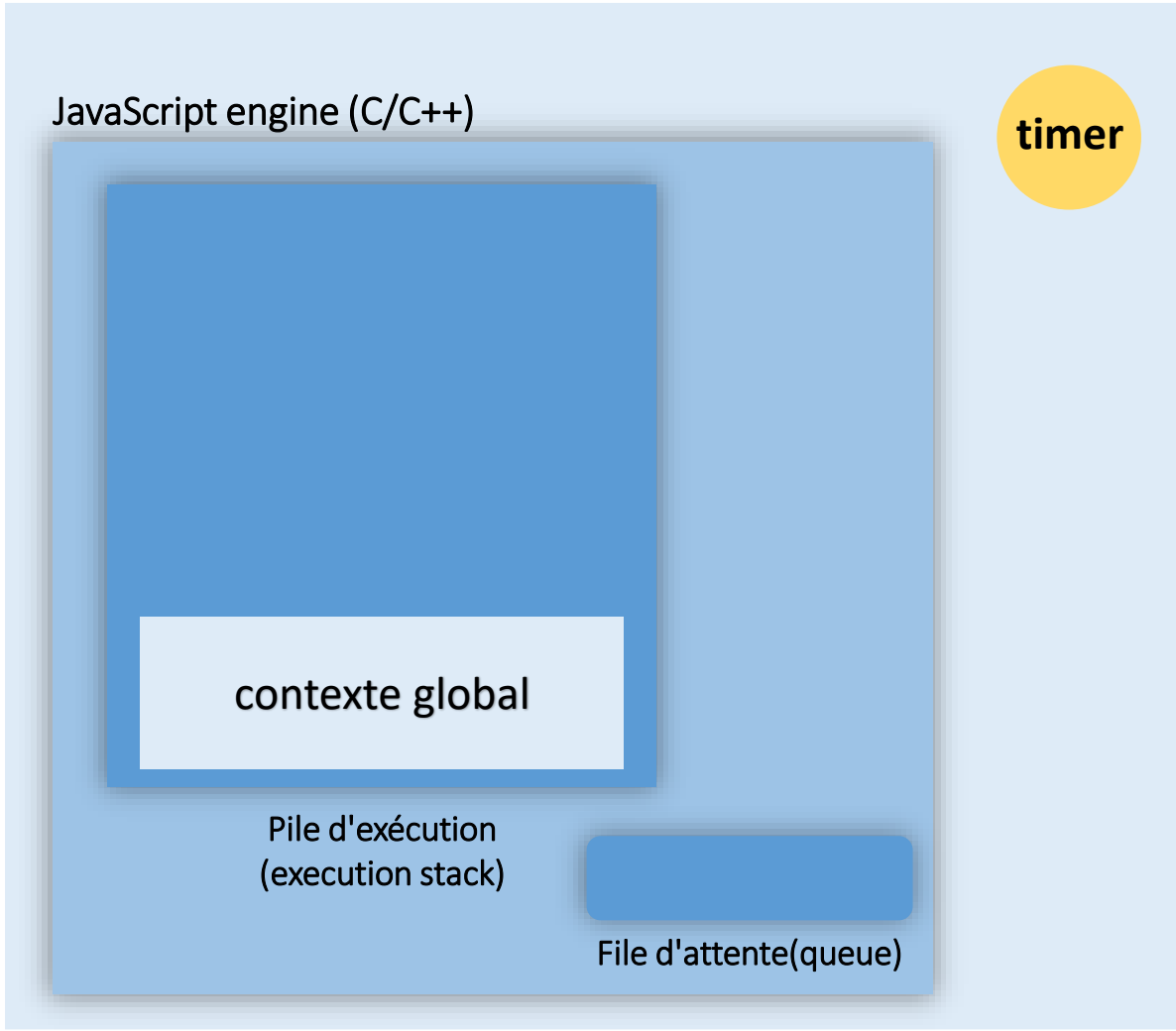


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.• log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.• log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.• log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

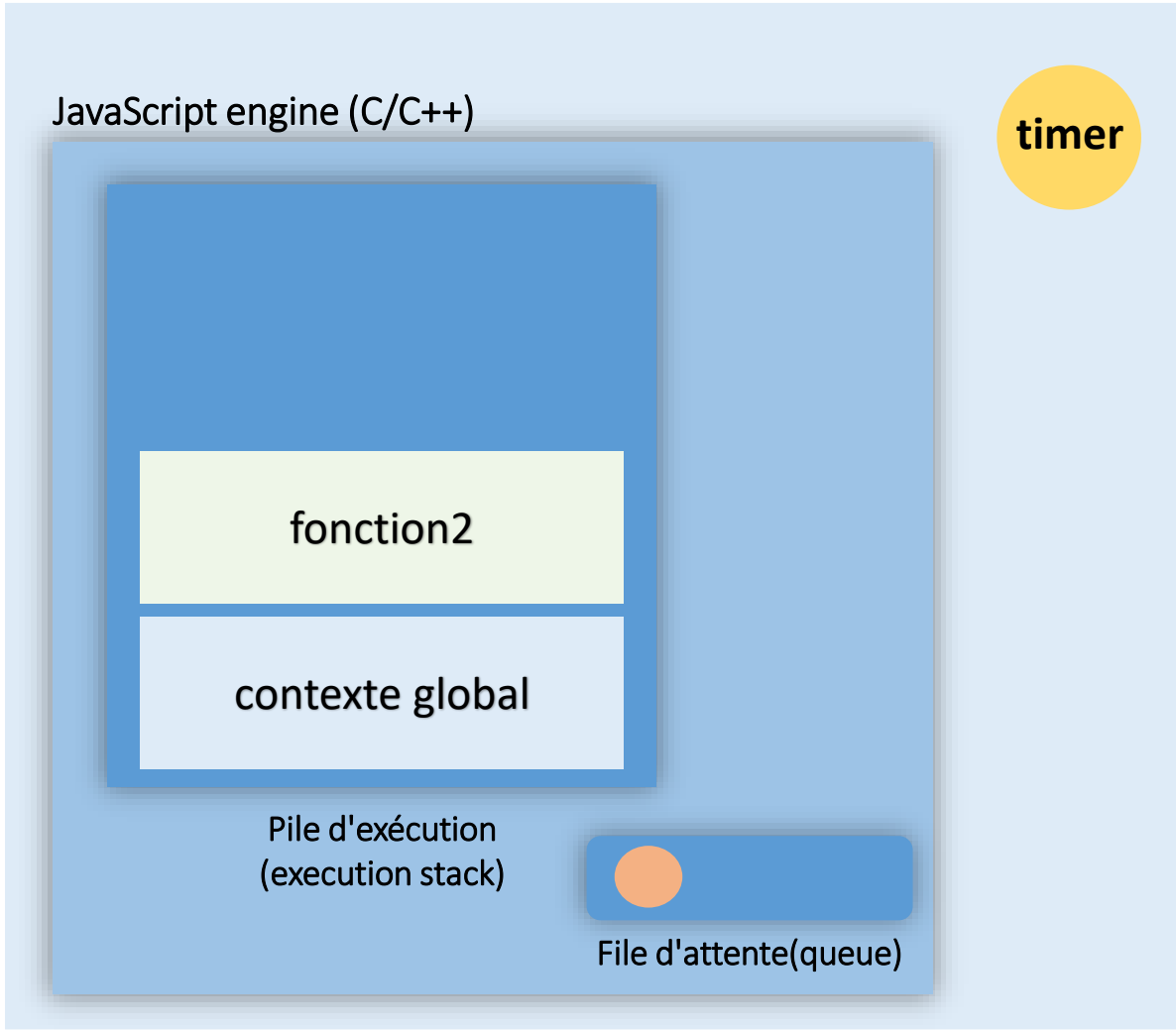


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
```

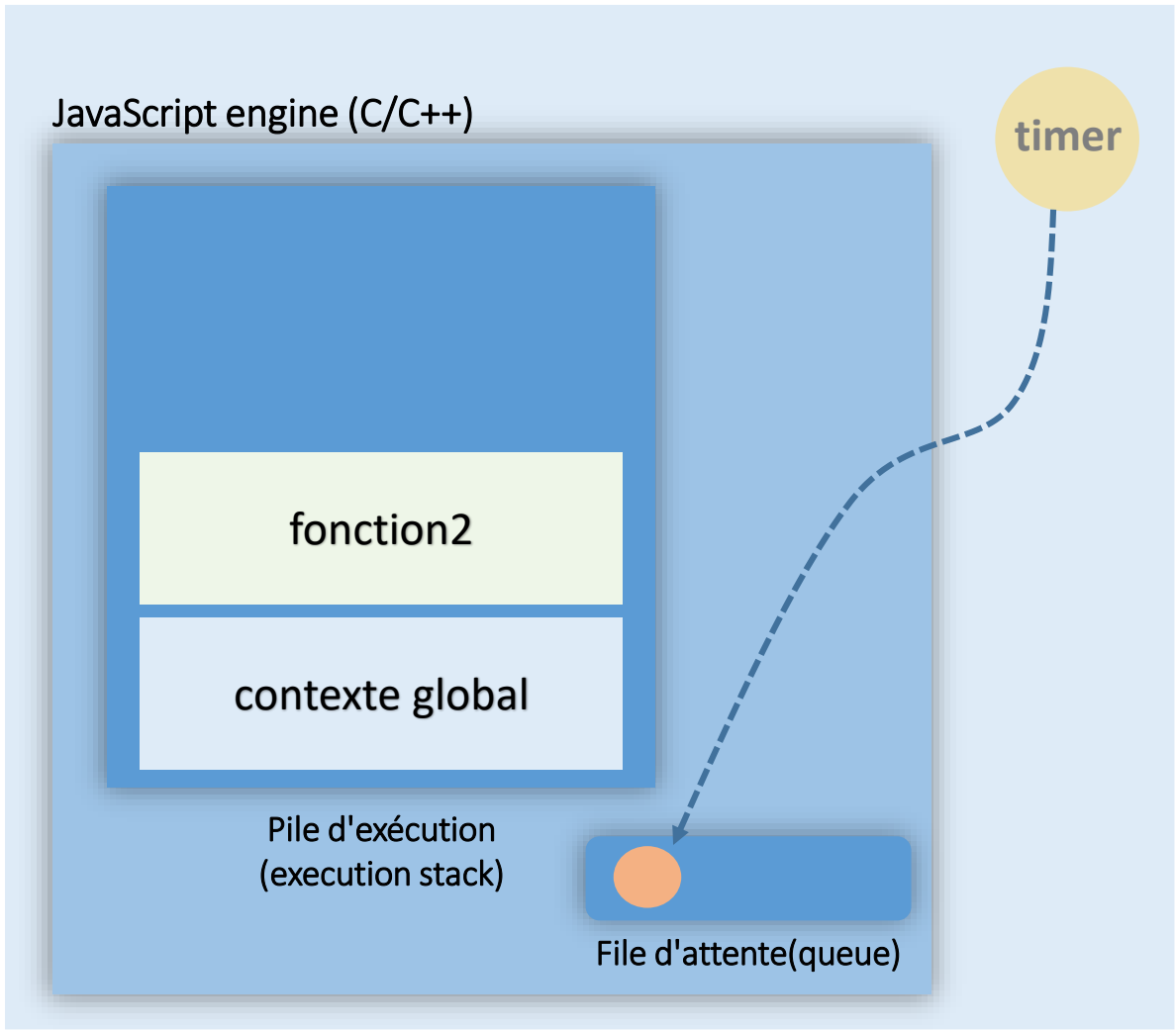
Environnement d'exécution Browser, NodeJS, ... (C/C++)



JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
```

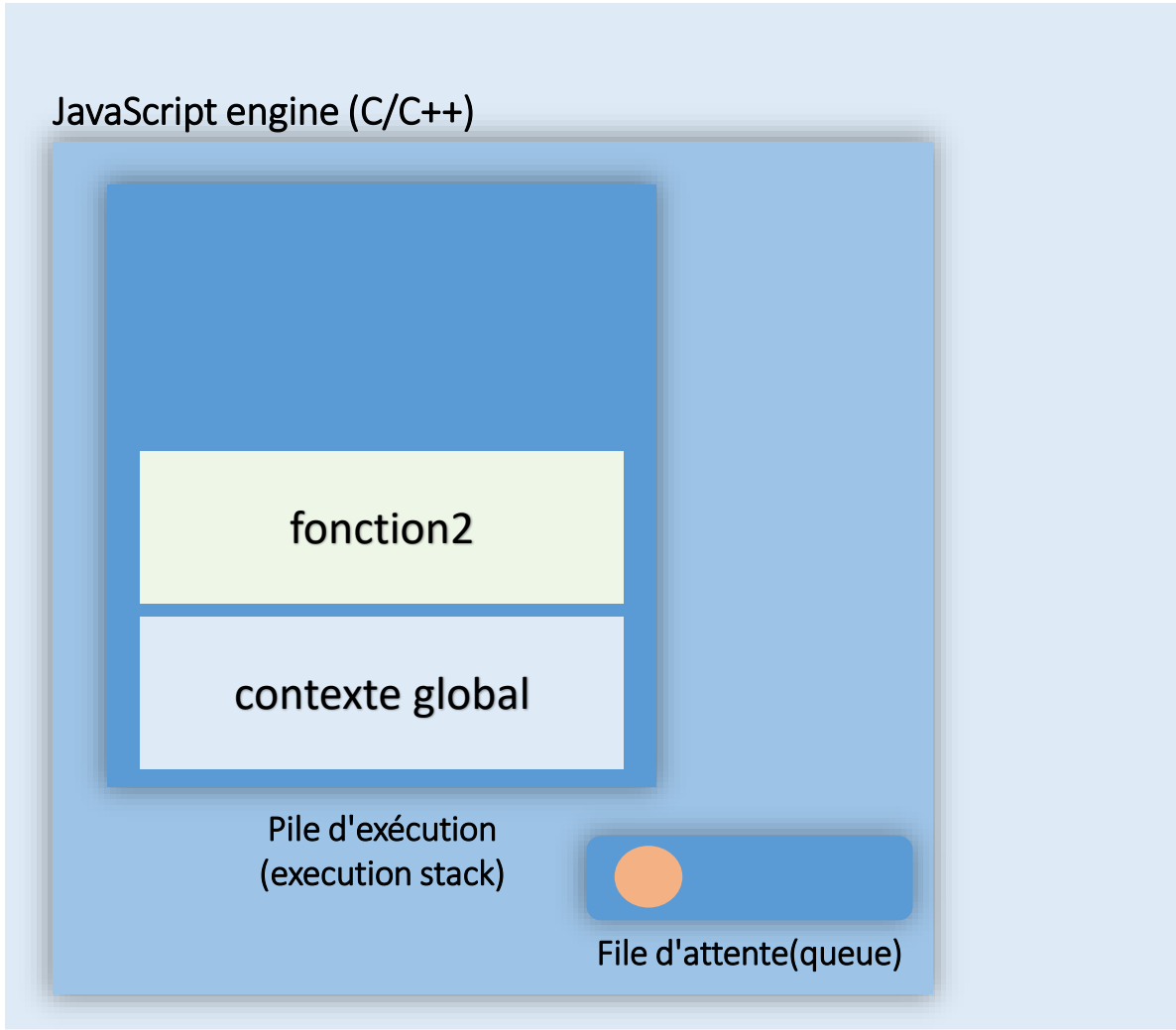


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

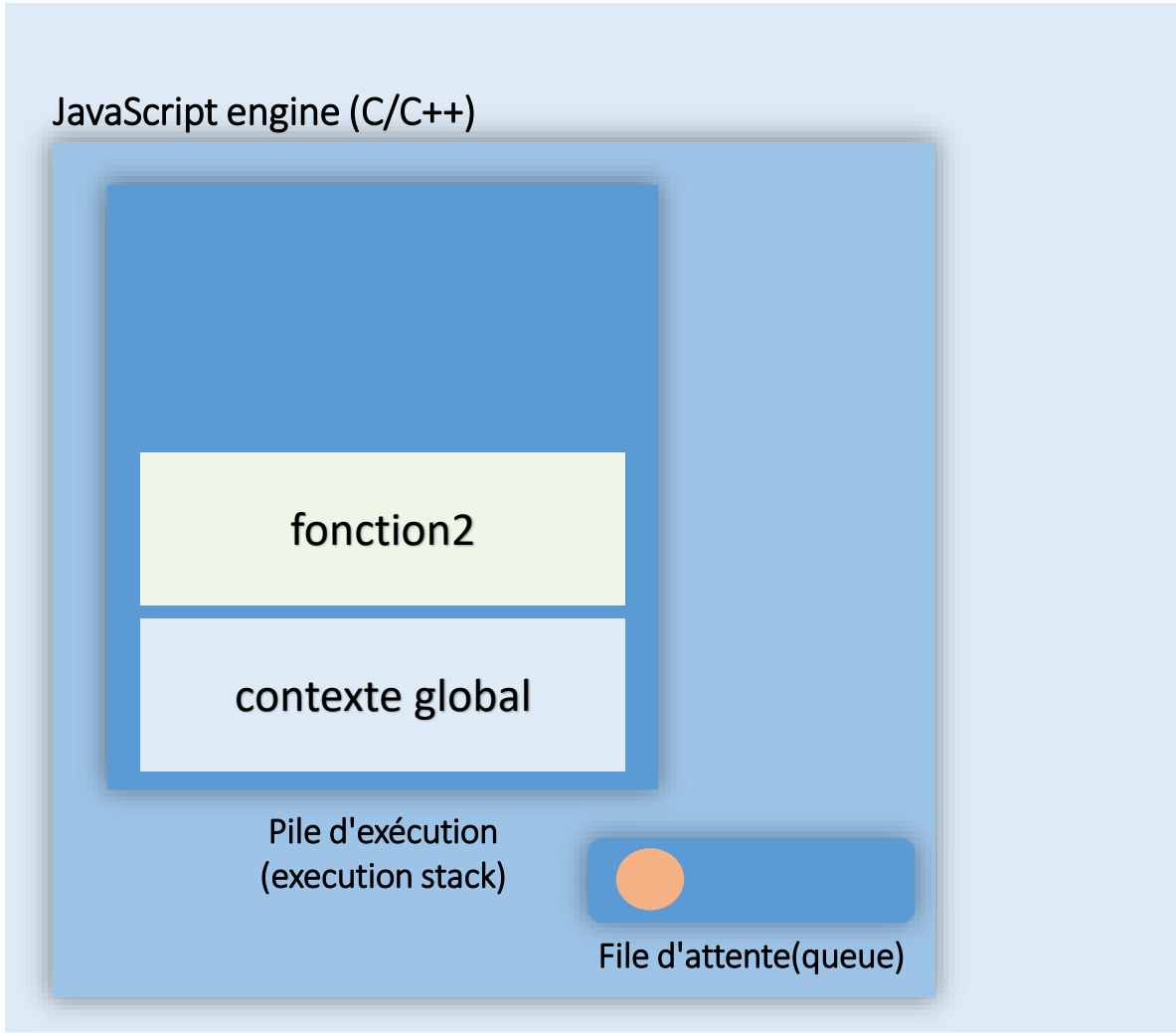


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

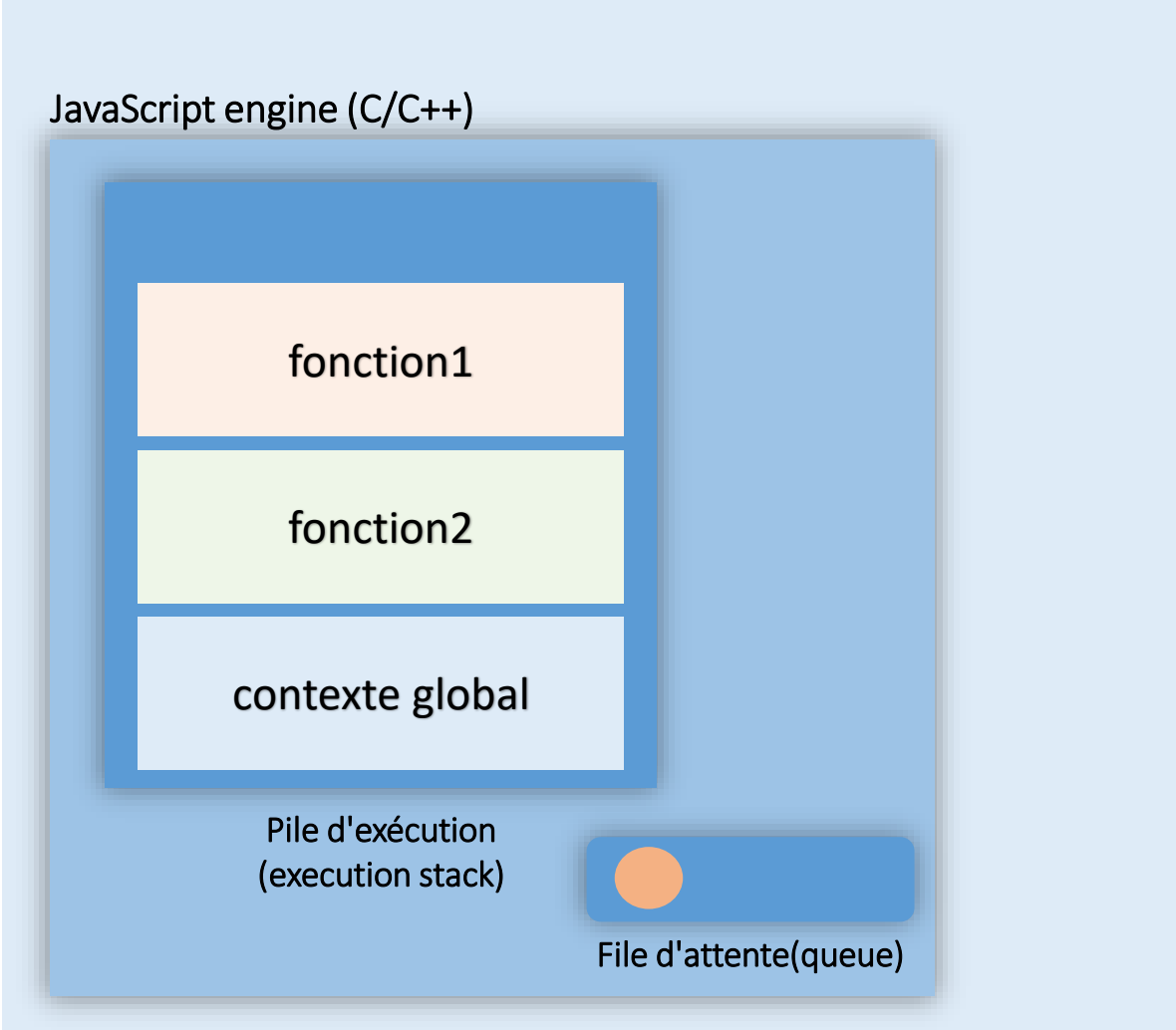


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
World
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

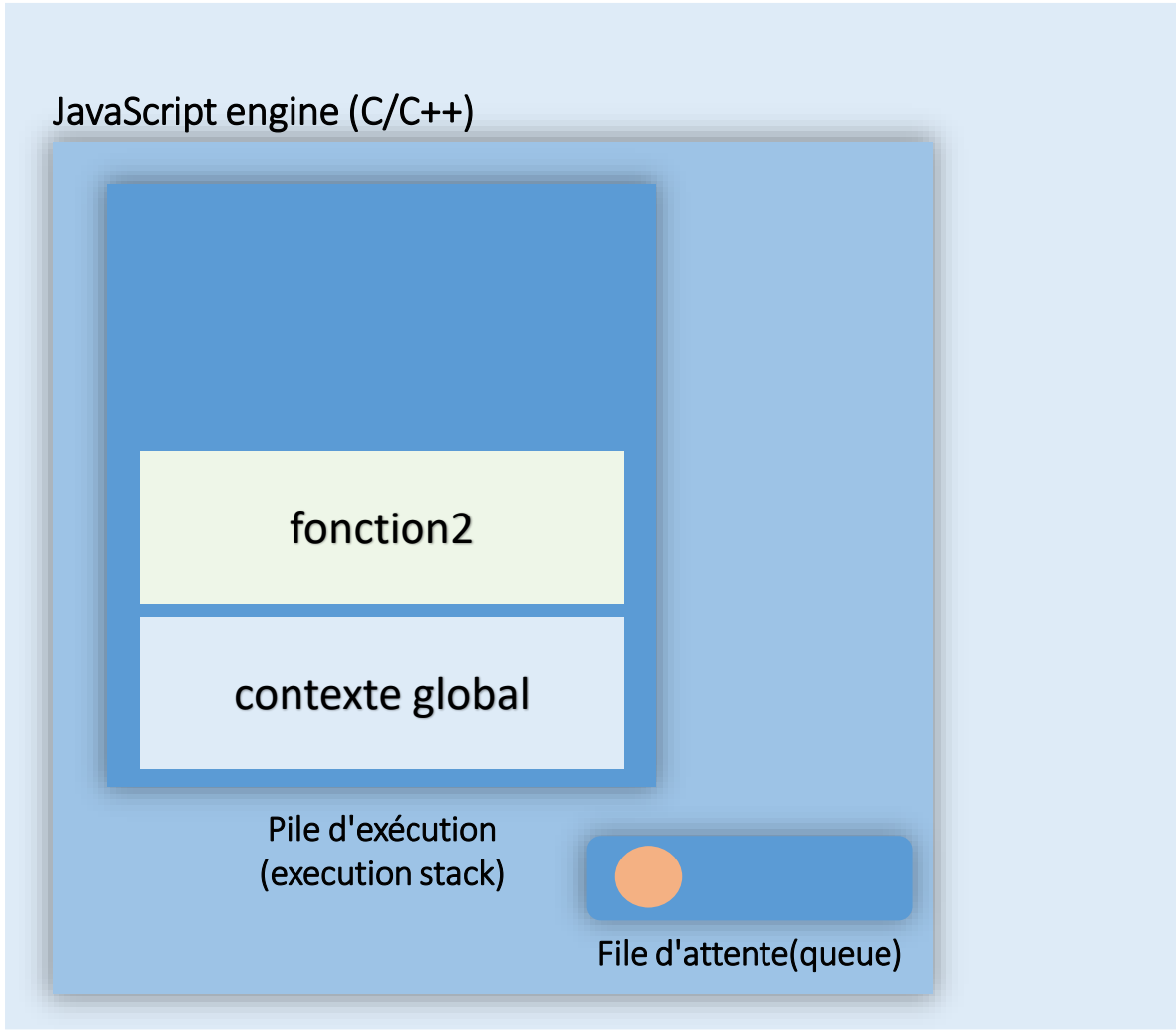


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.• log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.• log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.• log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
World
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

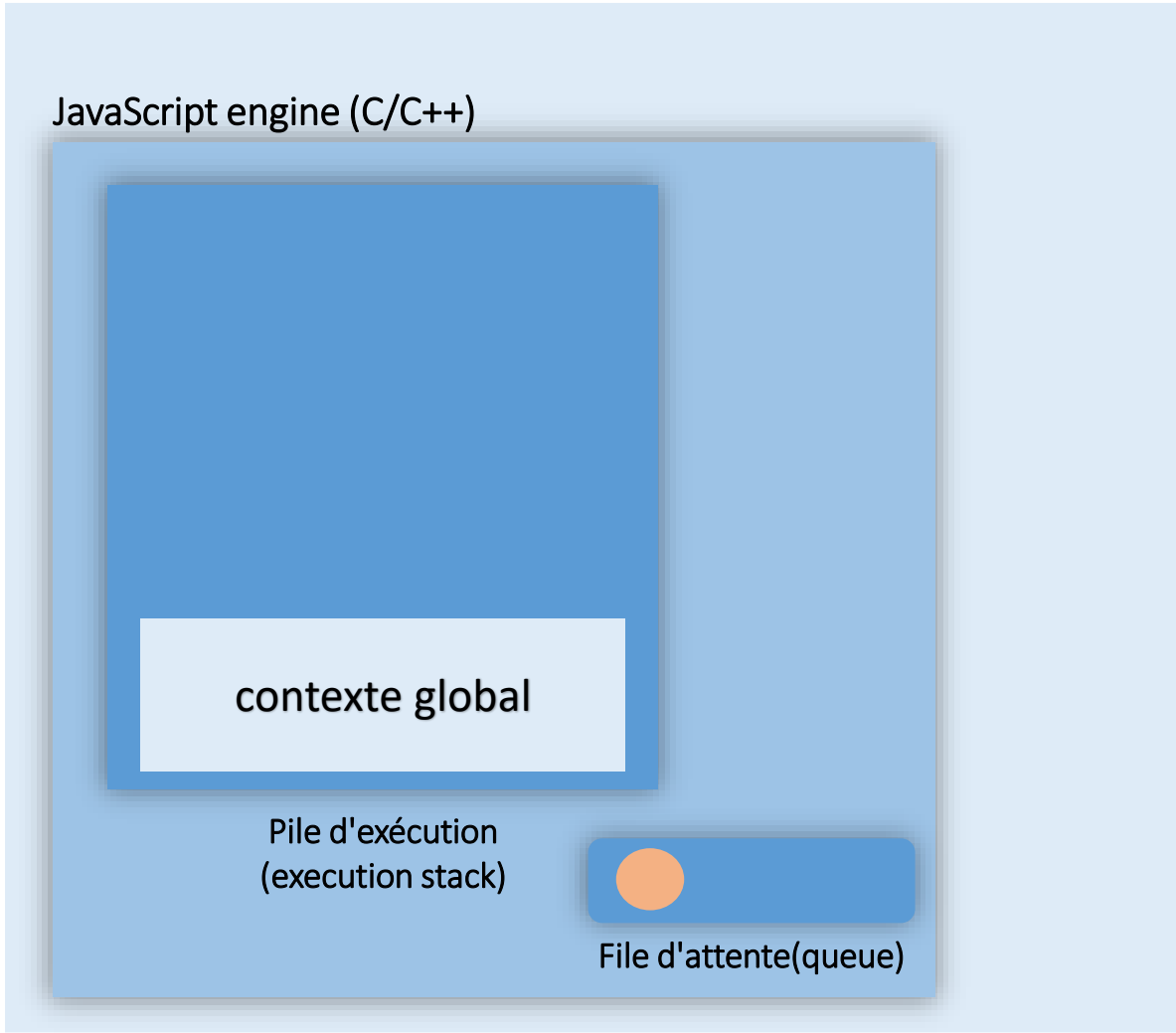


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
World
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

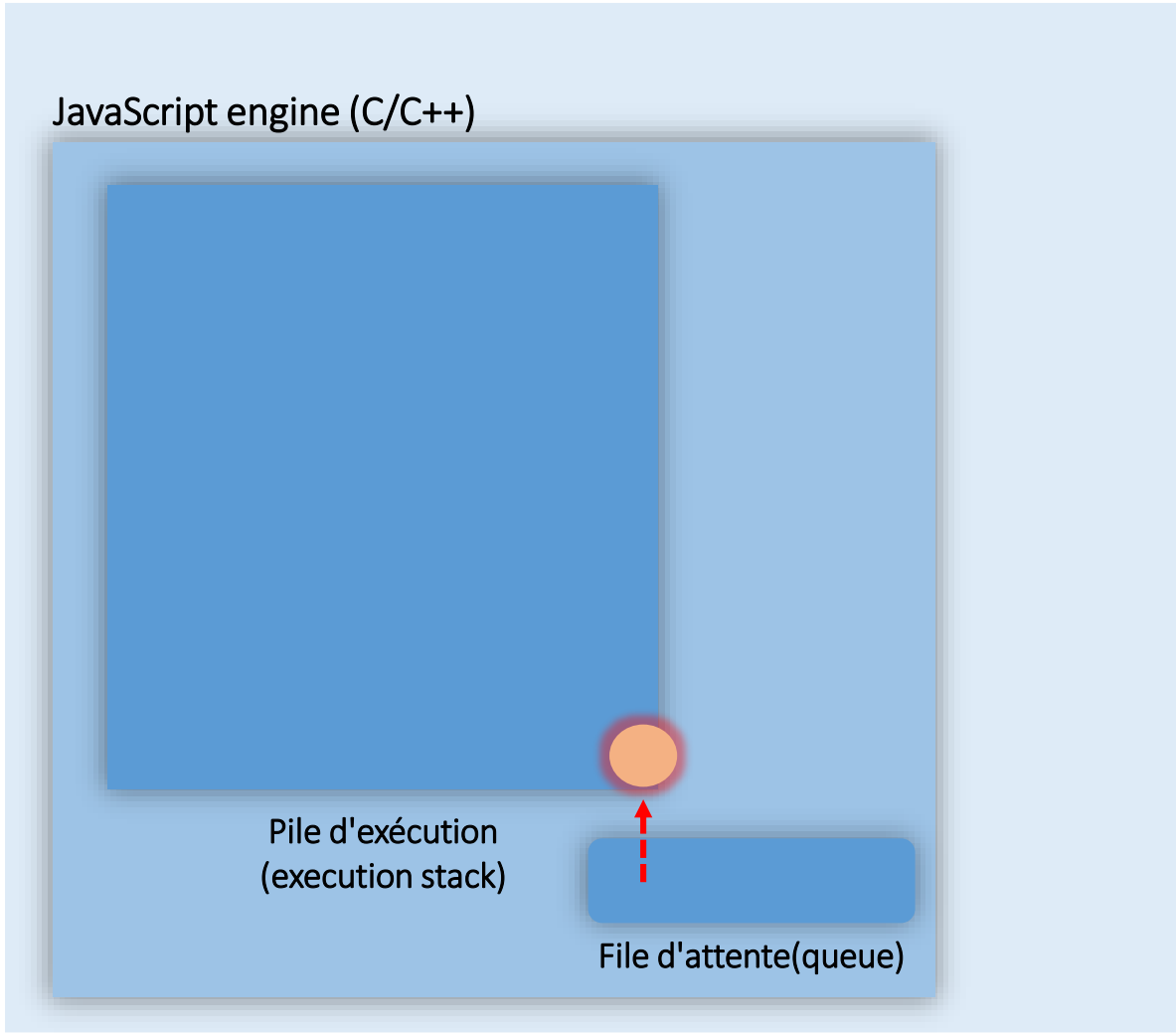


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer")
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
```

```
Hello
début itération
*
*
*
*
World
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

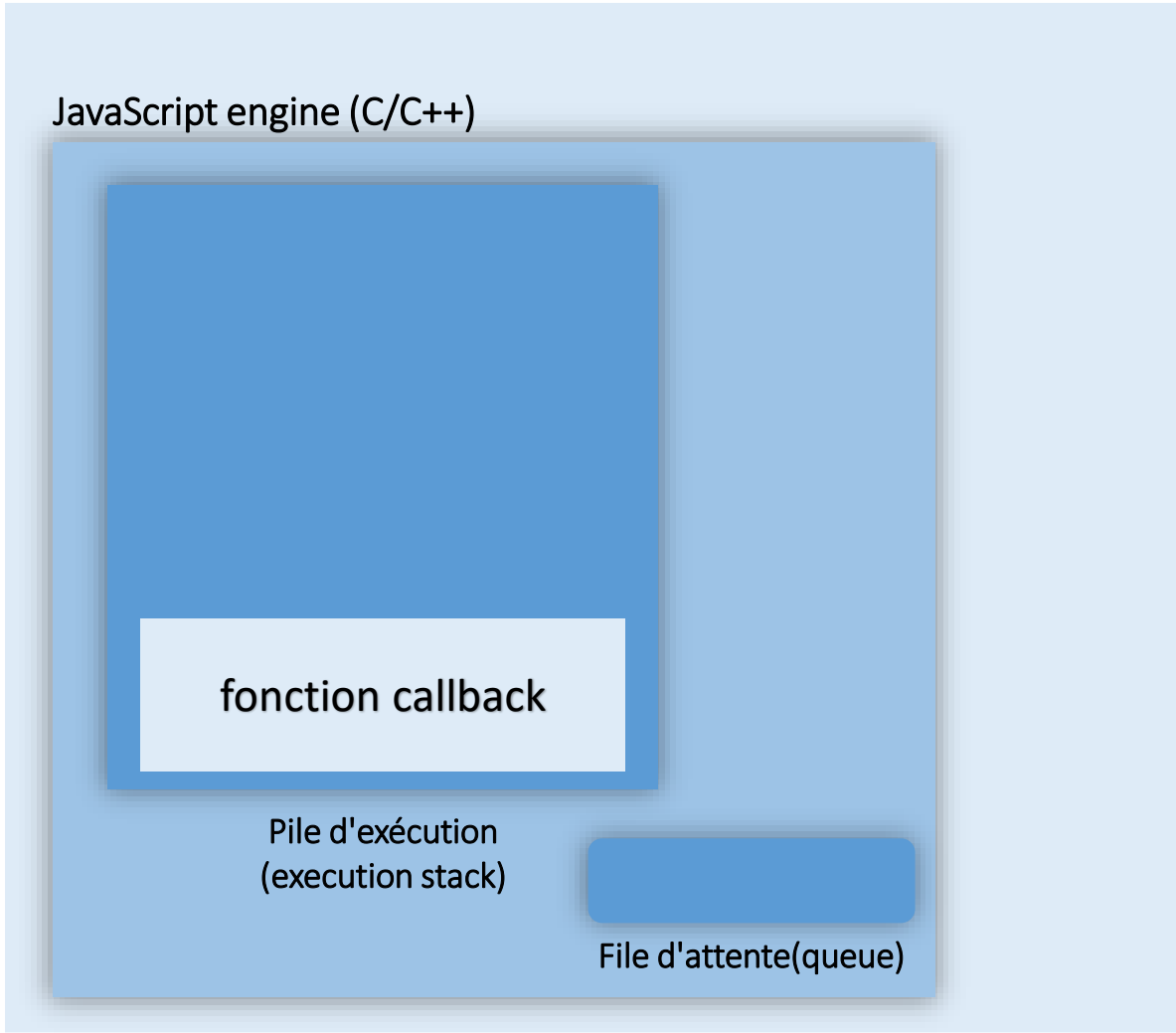


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
World
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

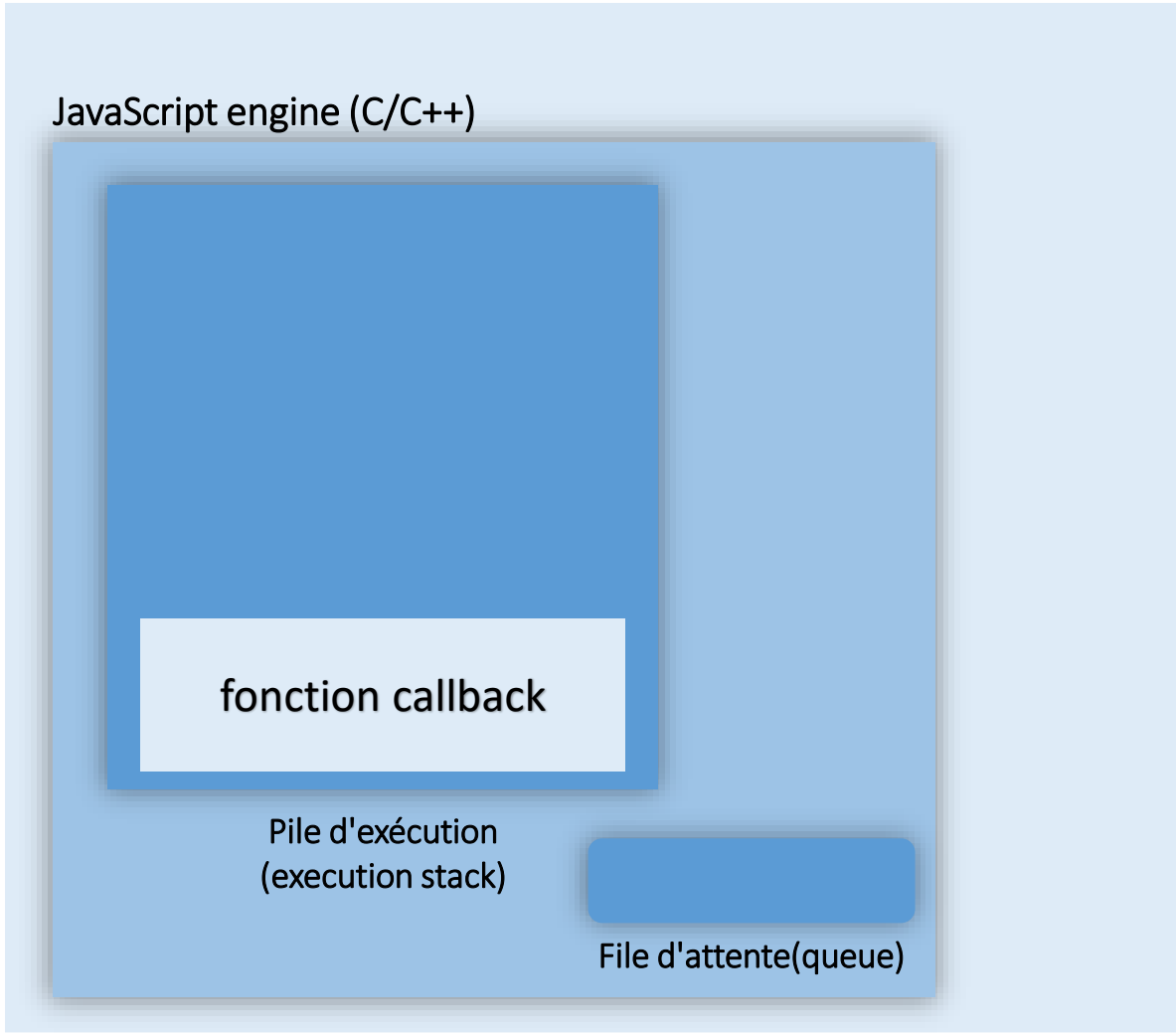


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer"
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
début itération
*
*
*
*
World
exécution callback du timer
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)

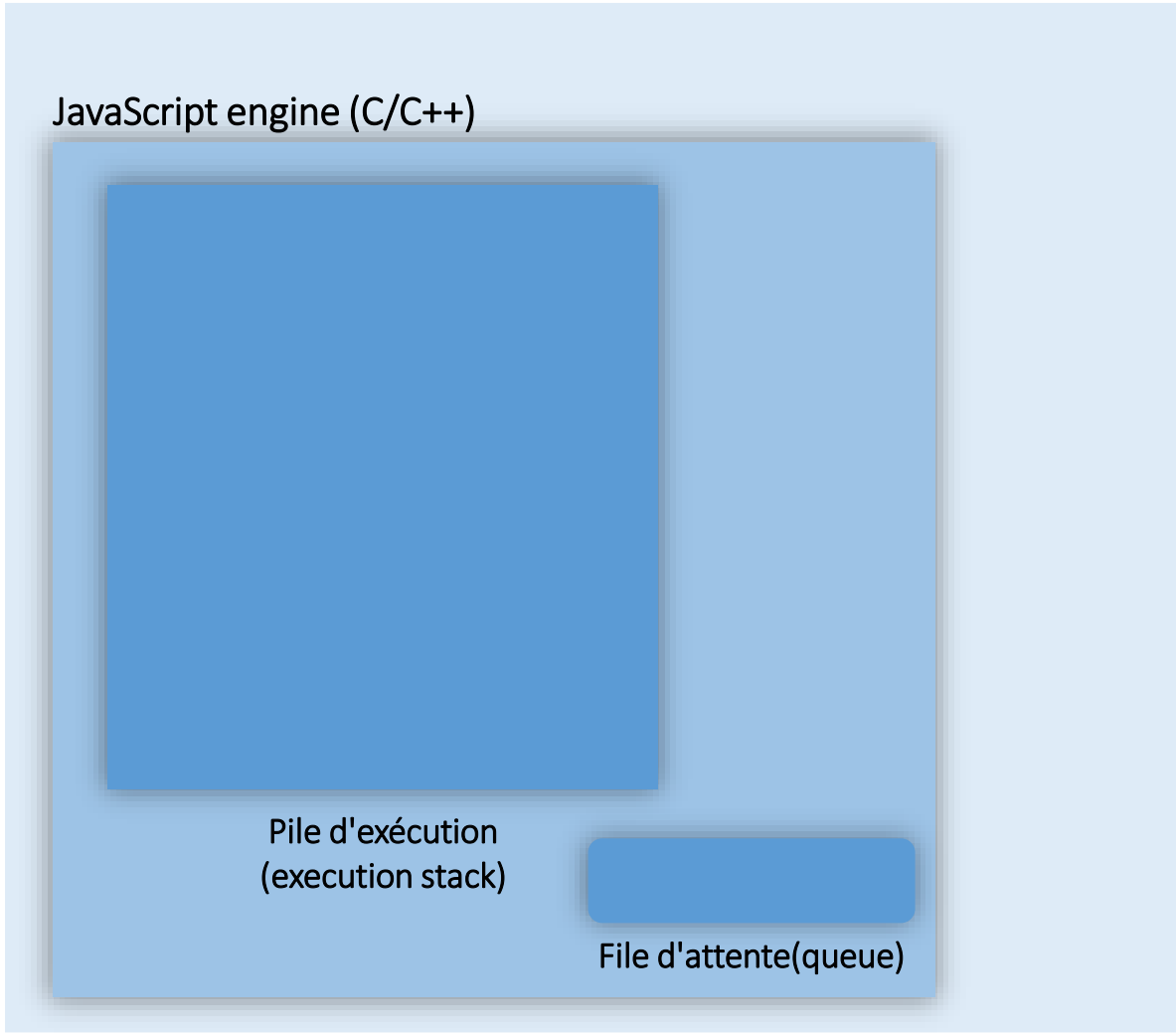


JavaScript et Programmation asynchrone

```
1  function fonction1(mess) {
2      console.log(mess);
3  }
4
5  function fonction2(mess, nb) {
6      console.log("debut itération")
7      for (let i = 0; i < nb; i++) {
8          if (i % 500_000_000 === 0) {
9              console.log("*");
10         }
11     }
12     fonction1(mess);
13 }
14
15 setTimeout(
16     () => {
17         console.log("execution callback du timer")
18     },
19     10
20 );
21 fonction1("Hello");
22 fonction2("World", 2_000_000_000);
23
```

```
Hello
debut itération
*
*
*
*
World
execution callback du timer
```

Environnement d'exécution Browser, NodeJS, ... (C/C++)



JavaScript et Programmation asynchrone

- Comment enchaîner séquentiellement des traitements asynchrones ?

```
setTimeout(
  () => {
    console.log("first message")
  },
  5000
);
setTimeout(
  () => {
    console.log("second message")
  },
  3000
);
setTimeout(
  () => {
    console.log("third message")
  },
  1000
);
```

```
third message
second message
first message
```

?

```
first message
second message
third message
```

t = 5 s
t = 8 s
t = 9 s

JavaScript et Programmation asynchrone

- Comment enchaîner séquentiellement des traitements asynchrones ?

```
setTimeout(
  () => {
    console.log("first message")
  },
  5000
);
setTimeout(
  () => {
    console.log("second message")
  },
  3000
);
setTimeout(
  () => {
    console.log("third message")
  },
  1000
);
```

t = 1 s
t = 3 s
t = 5 s

```
third message
second message
first message
```

Callback imbriqués

```
setTimeout(
  () => {
      console.log("first message");
      setTimeout(
         () => {
             console.log("second message");
             setTimeout(
                () => {
                    console.log("third message");
                },
                1000
             )
         },
         3000
      );
  },
  5000
);
```

```
first message
second message
third message
```

t = 5 s
t = 8 s
t = 9 s

JavaScript et Programmation asynchrone

- Comment enchaîner séquentiellement des traitements asynchrones ?

Callback imbriqués

```
setTimeout(
  () => {
      console.log("first message");
      setTimeout(
         () => {
             console.log("second message");
             setTimeout(
                () => {
                    console.log("third message");
                },
                1000
             )
         },
         3000
      );
  },
  5000
);
```

Callback Hell ou
Pyramid of doom

plus il y a de traitements asynchrones à enchainer plus la pyramide d'appels imbriqués croit vers la droite, plus l'écriture et la compréhension du code deviennent difficiles

Atténuer le problème en faisant de
chaque action une fonction séparée de haut niveau

```
function message(mess, delay, callback) {
    setTimeout(
        () => {
            console.log(mess);
            if (callback) callback();
        }, delay);
}

function message1() {
    message("first message", 5000, message2);
}

function message2() {
    message("second message", 3000, message3);
}

function message3() {
    message("third message", 1000);
}

message1();
```

plus d'imbrication profonde mais le code reste difficile à lire (il faut passer d'un morceau de code à l'autre pour comprendre l'enchainement des traitements) pas pratique surtout quand le code devient gros

➔pour répondre à ces difficultés introduction dans ES6 d'un nouveau type d'objets : `Promises` (promesses)

Promises (promesses)

- Promesses (**Promises**) à la base de la programmation asynchrone en JavaScript moderne.
- Une promesse est un objet renvoyé par une fonction asynchrone, qui représente l'état actuel de l'opération.
  - Opération en cours (promesse **en attente** - *pending*) ;
  - Opération terminée (promesse **acquittée** - *settled*))
    - avec succès (promesse **tenue** - *fulfilled :* un résultat est disponible) ;
    - stoppée après un échec (promesse **rompue** - *rejected :* un erreur indique la cause de l'échec)
- Au moment où la promesse est renvoyée à l'appelant, l'opération n'est souvent pas terminée, mais l'objet promesse fournit des méthodes permettant de lui attacher des gestionnaires (*handlers*) qui seront exécutés lorsque la promesse sera acquittée pour gérer le succès ou l'échec éventuel de l'opération.

Avec fonctions callback

```
function asyncOp1(callback1) {
    // ... traitement asynchrone
    // quand le traitement est terminé
    // le fonction asynchrone appelle
    // la fonction callback avec les
    // résultats du traitement
    callback1(res1);
}

function op2(params) {
    // ...
}

asyncOp1(op2);
```

Avec les promesses

```
function asyncOp1() {
    // lance le traitement asynchrone
    // et retourne un objet Promesse qui,
    // lorsqu'elle sera tenue,
    // contiendra le résultat du traitement
}

function op2(params) {
    //…
}

let p1 = asyncOp1();
p1.then(op2);
```

le traitement asynchrone est lancé et une promesse est retournée

le fonction op2 est enregistrée auprès de p1 et sera exécutée avec la valeur retournée par op1 lorsque celle-ci aura terminé son traitement

```
asyncOp1()
    .then(op2);
```

pas besoin de passer par une variable intermédiaire

L'objet Promise

- Un objet `Promise`
  - représente l'état d'une opération asynchrone.
  - possède deux propriétés internes (non accessibles directement)
    - `state` : définit l'état d'avancement de l'opération
      - `pending` : opération non terminée ,
      - `fulfilled` : opération terminée avec succès (promesse **tenue**)
      - `rejected` : opération terminée par un échec (promesse **rompue**)

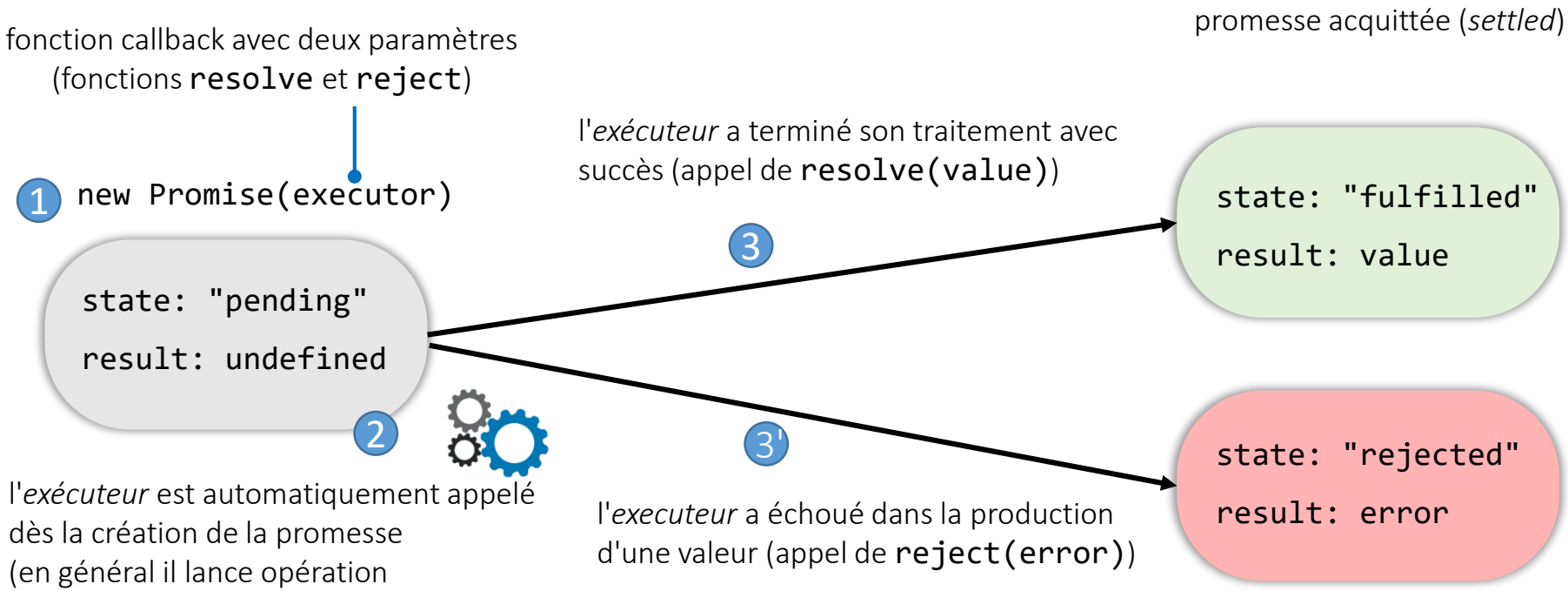
      } promesse **acquittée** (*fulfilled*)
    - `result` : le résultat de l'opération quand la promesse est acquittée (valeur de succès ou raison de l'échec)

- Constructeur

  `Promise(executor)`
  - `executor` une fonction à exécuter lors de la construction du nouvel objet `Promise`
  - quand on créée une promesse on doit lui fournir l'implémentation de l'*executor* qui se chargera d'appeler le traitement asynchrone et quand celui-ci se termine de modifier l'état de la promesse

Promises : modèle objet

- lorsque le constructeur de `Promise` appelle l'*exécuteur* il lui transmet deux fonctions en paramètres (*`resolve`* et *`reject`*)
- à la charge de l'exécuteur de lancer le traitement asynchrone et quand celui-ci sera terminé de communiquer à l'objet promesse le résultat où la raison de l'échec en utilisant les fonctions *`resolve`* et *`reject`* (qui prennent un seul paramètre pouvant être de n'importe quel type)



il ne peut y avoir qu'un seul résultat (ou erreur).
Tout changement d'état est final.
Dans l'*exécuteur* `resolve` ou `reject` ne peuvent être appelées qu'une fois, tout appel suivant est sans effet

utiliser un objet `Error` comme paramètre de `reject`

L'objet Promise

- Méthodes
  - `then(siTenue [, siRejetée])`
    - utilisée pour obtenir et exploiter le résultat d'une promesse.
    - *`siTenue`* : fonction de rappel invoquée lorsque la promesse est **tenue** (*fulfilled)*.
      - un seul argument, la valeur qui a permis de résoudre la promesse (i.e. la valeur passée en paramètre de l'appel à `resolve` dans l'*exécuteur*).
    - *`siRejetée`* : fonction de rappel invoquée lorsque la promesse est **rompue** (*rejected*).
      - un seul argument, la raison pour laquelle la promesse a été rejetée (i.e. la valeur passée en paramètre de l'appel à `reject` dans l'*exécuteur,* en général un objet `Error`).

les fonctions *`siTenue`* et *`siRejetée`* sont appelées de manière **asynchrone**,
elles ne seront exécutées qu'une fois la promesse **acquittée** (settled) c'est-à-dire une fois que la promesse a été **tenue** (*fulfilled*) ou **rompue** (*rejected*)

- appel de `then` avec un seul argument `promise.then((result) => { ... } )`
  - utilisé si on est intéressé que par les complétions réussies

L'objet Promise

- Méthodes (suite)

  - `catch(`*`gestionErreur`*`)`
    - utilisée si on est intéressé uniquement par les cas d'erreur (⇔ `then(null, gestionErreur)`)
    - *`gestionErreur`* : fonction de rappel avec un seul argument, la raison pour laquelle la promesse a été rejetée.
    - ex : `promise.then(`*`succeshandler`*`).catch((error) => { ... } )`

  - `finally(`*`finalisation`*`)`
    - utilisée lorsque la promesse a été acquittée (qu'elle ait été tenue ou rejetée), évite ainsi de dupliquer du code entre les gestionnaires `then()` et `catch()`.
    - *`finalisation`* fonction de rappel sans paramètres, exécutée quelle que soit la terminaison de la promesse

Modèle des promesses (Promises)

création de la promesse

```
let aPromise = new Promise(
  function (resolve, reject) {
      setTimeout(
         () => {
             const nb = Math.random();
             if ( nb < 0.5) {
                 resolve(nb);
             }
             else {
                 reject(new Error(nb));
             }
         },
         1000
      )
  }
);
```

executor, fonction **de rappel** exécutée dès la création de la promesse

consommation de la promesse

```
aPromise.then(
    (result) => console.log (`Succes[${result}]`),
    (error) => console.log(`Failed [${error.message}]`)
);
```

fonction de rappel exécutée si la promesse a été tenue (un résultat est disponible)

fonction de rappel exécutée si la promesse a été rompue (une erreur produite)

```
console.log("Hello promise !");
```

les fonctions de rappel sont asynchrones. Elles ne seront exécutées que quand la promesse est *acquittée* (*tenue* ou *rompue*) **et que la pile d'exécution est vide**
=> `console.log` exécutée en 1er

```
Hello promise !
Succes[0.2623066331027655]
```

```
Hello promise !
Failed [0.6043363237748061]
```

Modèle des promesses (Promises)

création de la promesse

```
let aPromise = new Promise(
  function (resolve, reject) {
      setTimeout(
         () => {
             const nb = Math.random();
             if ( nb < 0.5) {
                 resolve(nb);
             }
             else {
                 reject(new Error(nb));
             }
         },
         1000
      )
  }
);
```

executor, fonction **de rappel** exécutée dès la création de la promesse

consommation de la promesse

```
aPromise.then(
    (result) => console.log(`Succes[${result}]`))
 .catch(
    (error) => console.log(`Failed [${error.message}]`))
 .finally(
     () => console.log("finalisation")
  );

console.log("Hello promise !");
```

fonction de rappel exécutée si la promesse a été tenue (un résultat est disponible)

fonction de rappel exécutée si la promesse a été rompue (une erreur produite)

fonction de rappel pour finalisation des traitements

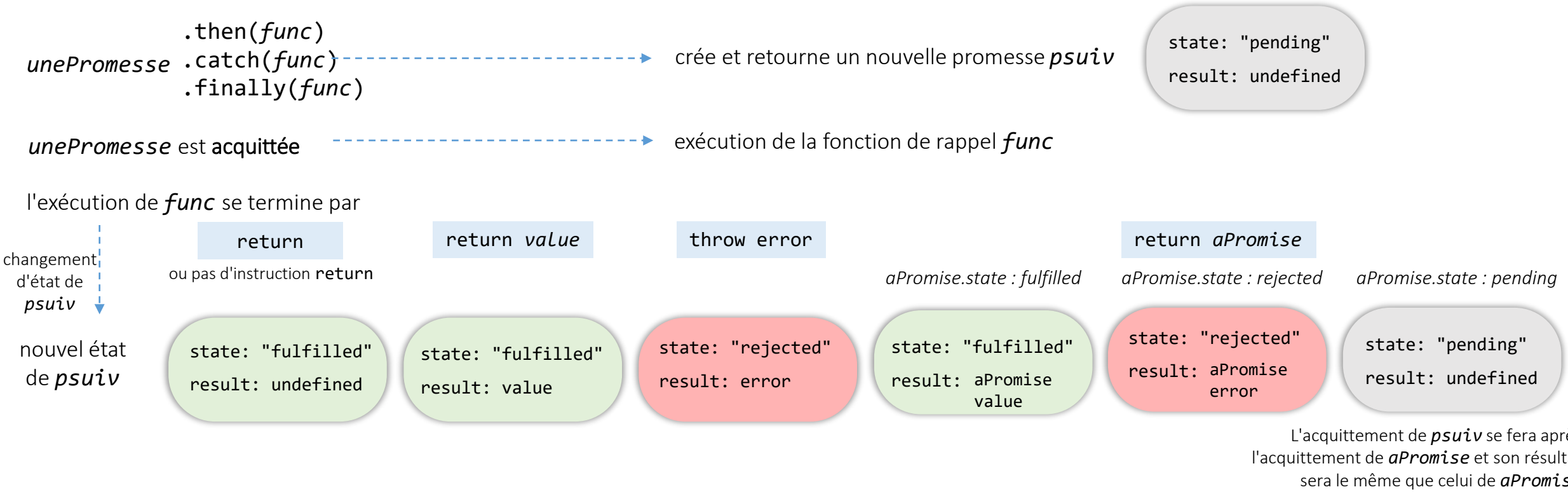
les fonctions de rappel sont asynchrones. Elles ne seront exécutées que quand la promesse est *acquittée* (*tenue* ou *rompue*) **et que la pile d'exécution est vide**
=> `console.log` exécutée en 1er

```
Hello promise !
Succes[0.2623066331027655]
finalisation
```

```
Hello promise !
Failed [0.6043363237748061]
finalisation
```

Chaînage des promesses

- **`then`**, **`catch`** et **`finally`** retournent toutes une nouvelle **`Promise`** ➔ Permet d'enchaîner une suite de traitements asynchrones dans un ordre donné.
- Cette **`Promise`** lorsqu'elle est créée est attente (*pending*) , elle ne sera acquittée qu'après que la fonction de rappel associée (*handler*) à la méthode qui l'a créée (**`then`**, **`catch`** ou **`finally`**) ait été exécutée.
- Ce que contient cette **`Promise`** lorsqu'elle sera acquittée va dépendre de la manière dont l'exécution de la fonction de rappel se termine



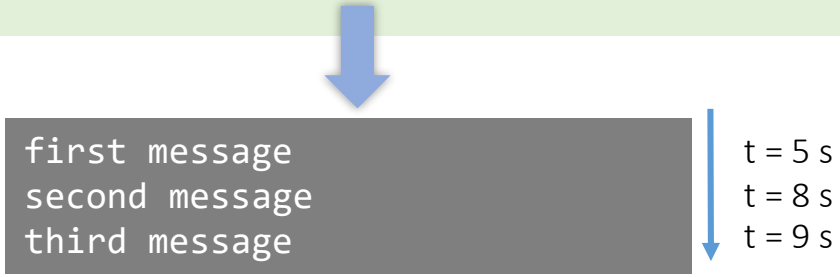
L'acquittement de ***psuiv*** se fera après l'acquittement de ***aPromise*** et son résultat sera le même que celui de ***aPromise***

Chaînage des promesses

- enchaîner séquentiellement des traitements asynchrones

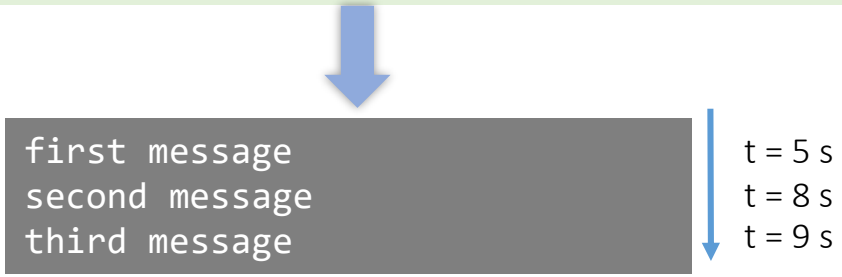
Avec fonctions callback

```
setTimeout(
  () => {
      console.log("first message");
      setTimeout(
         () => {
             console.log("second message");
             setTimeout(
                () => {
                    console.log("third message");
                },
                1000
             )
         },
         3000
      );
  },
  5000
);
```



Avec les promesses

```
function messageAssync(mess, delay) {
    return new Promise( (resolve, reject) => {
        setTimeout(() => { console.log(mess); resolve(true) }, delay);
    })
}

messageAssync("first message", 5000)
.then( () => messageAssync("second message",3000))
.then( () => messageAssync("third message", 1000));
```



Chaînage des promesses

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
})
.then(function (result) {
    console.log("1er then --> " + result);
    return result * 2;
})
.then(function (result) {
    console.log("2ème then --> " + result);
    return result * 2;
})
.then(function (result) {
    console.log("3ème then --> " + result);
    return result * 2;
});
console.log("Hello promesses !");
```

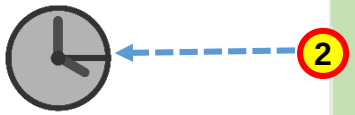
hello promesses !
1er then --> 1
2ème then --> 2
3ème then --> 4

```
let p = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

p.then(function (result) {
  console.log("1er then --> " + result);
  return result * 2;
});

p.then(function (result) {
  console.log("2ème then --> " + result);
  return result * 2;
});

p.then(function (result) {
  console.log("3ème then --> " + result);
  return result * 2;
});
console.log("Hello promesses !");
```

hello promesses !
1er then --> 1
2ème then --> 1
3ème then --> 1

Chaînage des promesses

- Etape 1 : exécution synchrone du code principal



création de
d'un timer (1s)

```
new Promise(function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
})
.then(function (result) {
    console.log("1er then --> " + result);
    return result * 2;
})
.then(function (result) {
    console.log("2ème then --> " + result);
    return result * 2;
})
.then(function (result) {
    console.log("3ème then --> " + result);
    return result * 2;
});
console.log("Hello promesses !");
```

1 → status: pending, value : undefined — création de la promesse initiale

2 — création d'un timer (1s)

3 → status: pending, value : undefined — création de la promesse produite par 1er **then**

4 → status: pending, value : undefined — création de la promesse produite par 2ème **then**

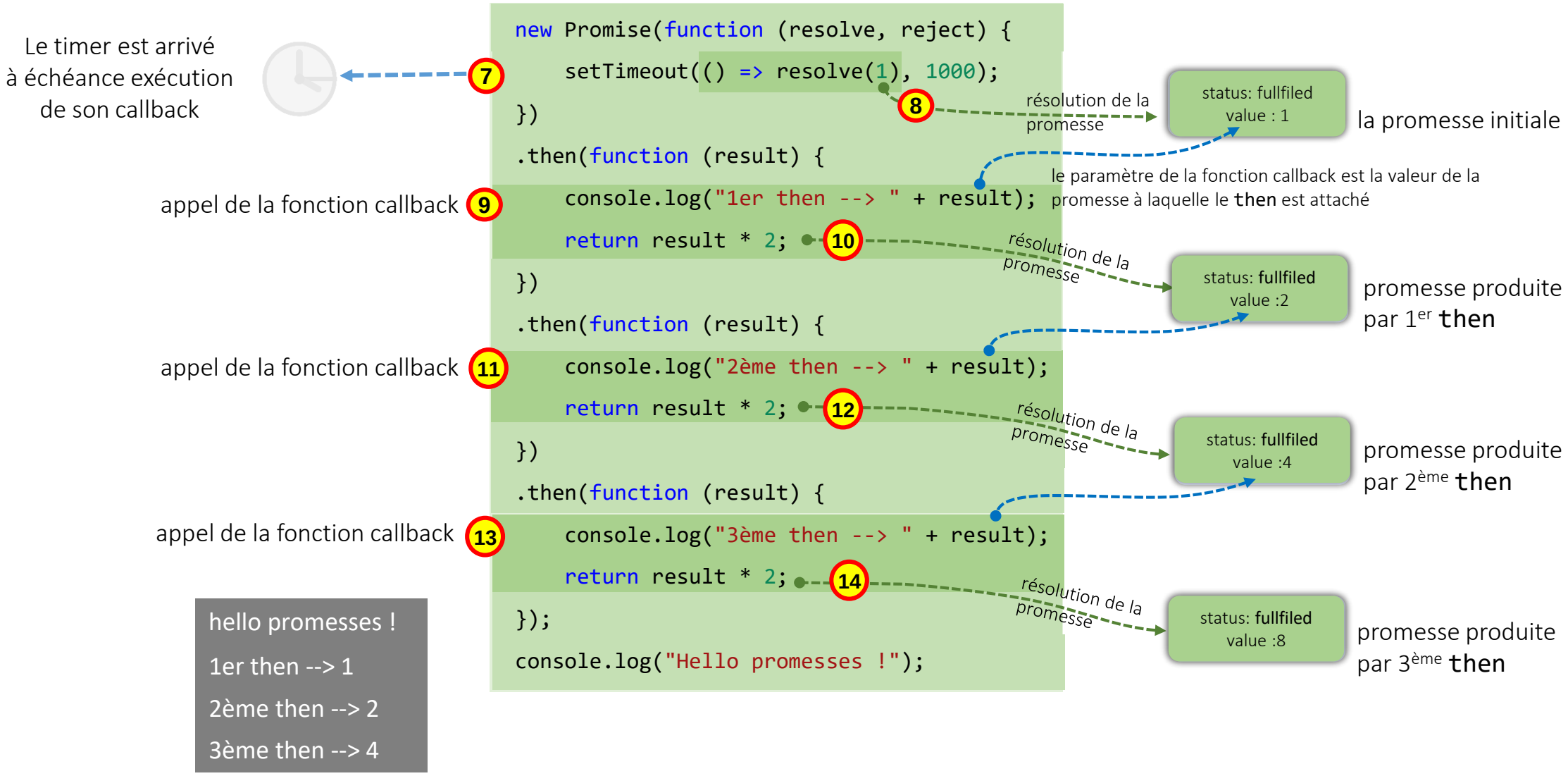
5 → status: pending, value : undefined — création de la promesse produite par 3ème **then**

6 — affiche le message sur la console

hello promesses !

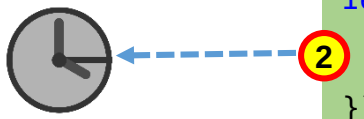
Chaînage des promesses

- Etape 2 : exécution du code des callback



Chaînage des promesses

- Etape 1 : exécution synchrone du code principal



création de
d'un timer (1s)

```
let p = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

p.then(function (result) {
  console.log("1er then --> " + result);
  return result * 2;
});

p.then(function (result) {
  console.log("2ème then --> " + result);
  return result * 2;
});

p.then(function (result) {
  console.log("3ème then --> " + result);
  return result * 2;
});
console.log("Hello promesses !");
```

2

1 → status: pending
value : undefined

création de
la promesse initiale

3 → status: pending
value : undefined

création de la
promesse produite
par 1er **then**

4 → status: pending
value : undefined

création de la
promesse produite
par 2ème **then**

5 → status: pending
value : undefined

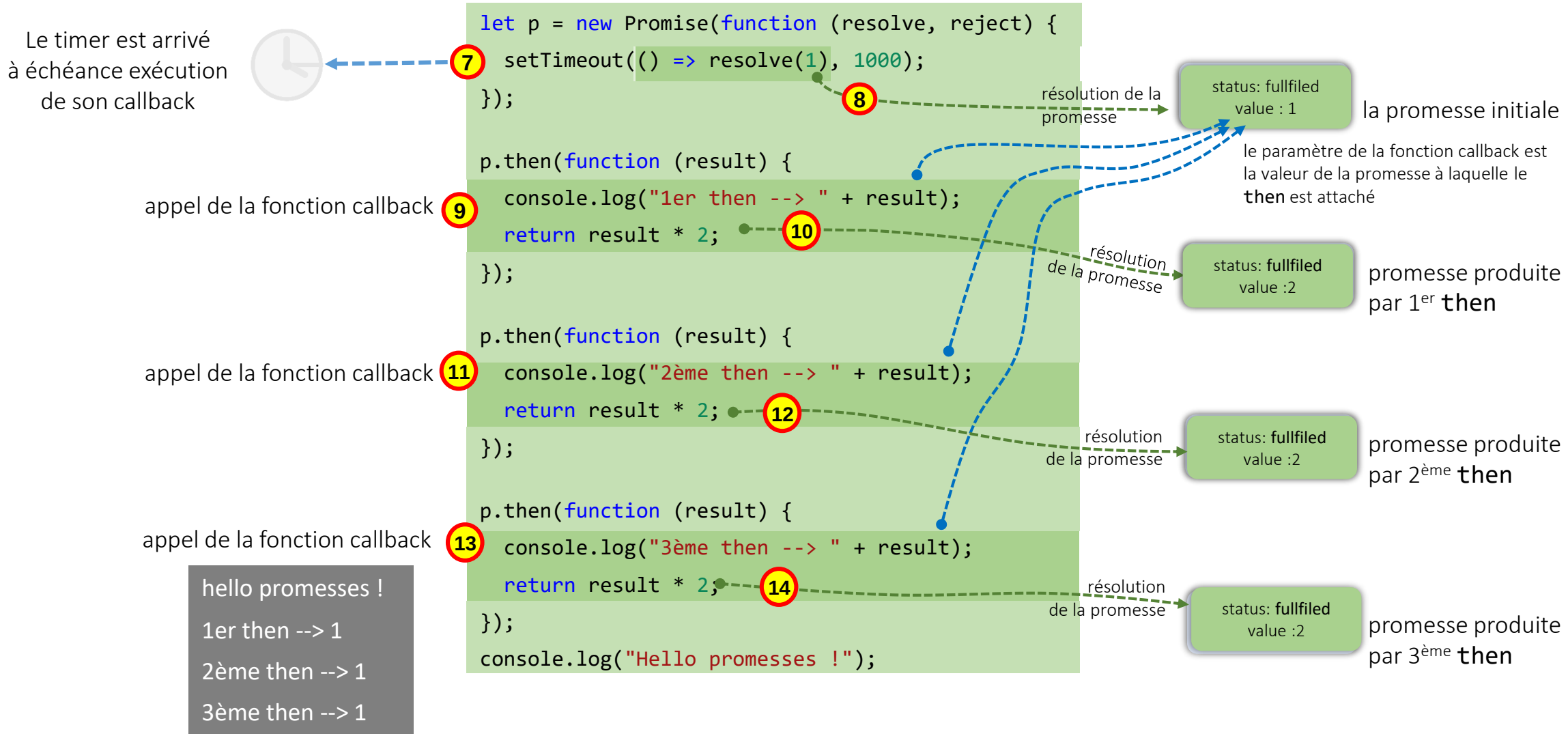
création de la
promesse produite
par 3ème **then**

hello promesses !

affiche le message sur la console 6

Chaînage des promesses

- Etape 2 : exécution du code des callback

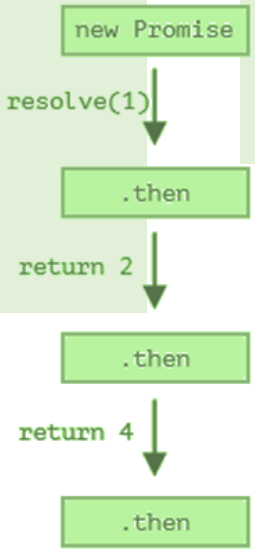


Chaînage des promesses

```
new Promise(
  function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
  }
)
  .then(
    function (result) {
      console.log("then 1 : " + result);
      return result * 2;
    }
  )
  .then(
    function (result) {
      console.log("then 2 : " + result);
      return result * 2;
    }
  )
  .then(
    function (result) {
      console.log("then 3 : " + result);
      return result * 2;
    }
  );

console.log("Hello promises chaînées");
```

⇔

```
let p1 = new Promise(
  function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
  }
);

let p2 = p1.then(
  function (result) {
    console.log("then de p1 : " + result);
    return result * 2;
  }
);

let p3 = p2.then(
  function (result) {
    console.log("then de p2 : " + result);
    return result * 2
  }
);

p3.then(
  function (result) {
    console.log("then de p3 : " + result);
    return result * 2;
  }
);

console.log("Hello promises chaînées");
```



chaque **then** est attaché à la promesse précédente dans la chaîne

*

attente 1s puis

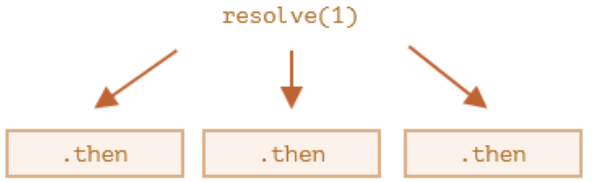
```
Hello promises chaînées
then de p1 : 1
then de p2 : 2
then de p3 : 4
```

≠

```
let p1 = new Promise(
  function (resolve, reject) {
    setTimeout(() => resolve(1), 1000);
  }
);

p1.then(
  function (result) {
    console.log("then de p1 : " + result);
    return result * 2;
  }
);
p1.then(
  function (result) {
    console.log("then de p2 : " + result);
    return result * 2
  }
);

p1.then(
  function (result) {
    console.log("then de p3 : " + result);
    return result * 2;
  }
);

console.log("Hello promises chaînées");
```

```
Hello promises chaînées
then de p1 : 1
then de p2 : 1
then de p3 : 1
```

les **then** sont tous attachés à la même promesse



*

* images d'après https://javascript.info/promise-chaining

Chaînage des promesses

```
1  new Promise(
2    function (resolve, reject) {
       setTimeout(() => resolve(1), 3000);
9    }
3  ).then(
     function (result) {
       console.log("then de p1 : " + result);
       return new Promise((resolve, reject) => {
         setTimeout(() => resolve(result * 2), 2000);
10     });
     }
4  ).then(
     function (result) {
       console.log("then de p2 : " + result);
       return new Promise((resolve, reject) => {
         setTimeout(() => resolve(result * 2), 1000);
12     });
     }
5  ).then(
     function (result) {
       console.log("then de p3 : " + result);
       return result * 2;
     }
   );

6  console.log("Hello promises chaînées");
```

8

11

13

l'*exécuteur* contient une action asynchrone qui sera terminée au bout de trois secondes.

Pour s'exécuter les gestionnaires (*handlers*) attachés à cette promesse (fonctions paramètres des méthodes **then** et **catch**) devront attendre sa résolution

le gestionnaire contient une action asynchrone qui sera terminée au bout de deux secondes: il retourne une promesse.

Pour s'exécuter les gestionnaires (handlers) qui suivent devront attendre la résolution de cette promesse

idem (avec attente de de 1 seconde)

attente 3s... puis

attente 2s... puis

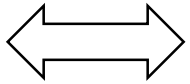
attente 1s... puis

```
Hello promises chaînées
then de p1 : 1
then de p2 : 2
then de p3 : 4
```

Promises : gestion des erreurs

- Si une exception intervient dans un *exécuteur* ou un *handler* (fonction de rappel de `then` ou `catch` ou `finally`) elle est implicitement attrapée et traité comme un rejet

```
function afficheErreur(err) {
  console.log("Error: " + err.message);
}

new Promise((resolve, reject) => {
     throw new Error("Whoops!");
  }
).catch(afficheErreur); // Error: Whoops!
```



```
function afficheErreur(err) {
  console.log("Error: " + err.message);
}

new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
  }
).catch(afficheErreur); // Error: Whoops!
```

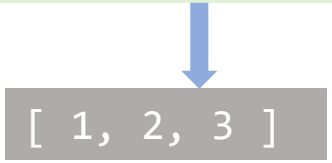
- Quand dans un chaîne de promesses une erreur intervient le contrôle est transmis au catch le plus proche

```
new Promise(executor)
  .then(...)
  .then(...)
  .then(...)
  .then(...)
  .catch(...)
  .then(...)
  .then(...)
  .then(...)
  .catch(...);
```

Promises : méthodes statiques

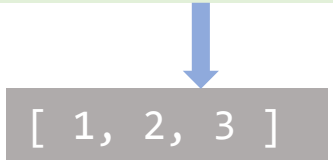
- `Promise.all(`*`itérable`*`)`
  - pour exécuter plusieurs promesses en parallèle, et attendre qu'elles soient toutes **tenues**.
    - Par exemple, télécharger plusieurs URLs en parallèle et traiter le contenu lorsque tout a été téléchargé
  - paramètre
    - *`itérable`* : un objet itérable (tel qu'un tableau (`Array`)) contenant des promesses.
  - Valeur de retour
    - Un objet `Promise` qui sera
      - **tenue** lorsque les promesses contenues dans *itérable* auront été **tenues** (*fulfilled*) avec comme valeur de résultat un tableau des résultats de chacune des promesses passée dans *itérable*
      - **rompue** (*rejected*) , dès que l'une des promesses d'*itérable* est **rompue** avec comme raison celle de la promesse **rompue**.

```
Promise.all([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise(resolve => setTimeout(resolve, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(result))
    .catch(error => console.log(`Erreur : ${error} !`));
```



toutes les promesses ont été **tenues** : la promesse retournée par `Promise.all` est **tenue** , result est un tableau des résultats des promesses (`result[i]` contient le résultat de la i[ème] promesse de l'*itérable* passé en argument)

```
Promise.all([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise((resolve,reject) => setTimeout(reject, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(result))
    .catch(error => console.log(`Erreur : ${error} !`));
```



toutes les promesses ont été **tenues** : la promesse retournée par `Promise.all` est **tenue** , result est un tableau des résultats des promesses (`result[i]` contient le résultat de la i[ème] promesse de l'*itérable* passé en argument)

Promises : méthodes statiques

- `Promise.any(`*`itérable`*`)`
  - pour exécuter plusieurs promesses en parallèle et attendre que l'une d'elle soit **tenue** (*fulfilled*).
  - paramètre
    - *`itérable`* : un objet itérable (tel qu'un tableau (`Array`)) contenant des promesses.
  - Valeur de retour
    - Un objet `Promise` qui sera
      - **tenue** dès que l'une promesses contenues dans *itérable* aura été **tenue** (*fulfilled*) avec comme valeur de résultat le résultat de la promesse **tenue**
      - **rompue** (*rejected*) , si toutes les promesses d'*itérable* ont été **rompues** (*rejected*) avec comme raison une erreur de type `AgragateError` (objet qui stocke les erreurs de chacune des promesses d'*itérable*)

```
Promise.any([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise(resolve => setTimeout(resolve, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(`Resultat : ${result}`))
    .catch(error => console.log(`Erreur : ${error} !`));
```

```
Résultat : 2
```

La valeur de la première promesses qui a été **tenue**.

```
Promise.any([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise((resolve,reject) => setTimeout(reject, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(`Resultat : ${result}`))
    .catch(error => console.log(`Erreur : ${error} !`));
```

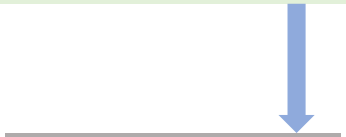
```
Résultat : 3
```

La valeur de la première promesses qui a été **tenue**.

Promises : méthodes statiques

- `Promise.race(`*`itérable`*`)`
  - pour exécuter plusieurs promesses en parallèle et attendre que l'une d'elle soit **acquittée** (setteld) (**tenue** *(fulfilled)* ou **rompue** (*rejected*)).
  - paramètre
    - *`itérable`* : un objet itérable (tel qu'un tableau (`Array`)) contenant des promesses.
  - Valeur de retour
    - Un objet `Promise` qui sera
      - **tenue** dès que l'une promesses contenues dans *itérable* aura été **tenue** (*fulfilled*) avec comme valeur de résultat le résultat de la promesse **tenue**
      - **rompue** (*rejected*) , dès que l'une promesses contenues dans a été **rompue** (*rejected*) avec comme raison l'erreur de la promesse **rompue**

```
Promise.race([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise(resolve => setTimeout(resolve, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(`Resultat : ${result}`))
    .catch(error => console.log(`Erreur : ${error} !`));
```



```
Résultat : 2
```

La valeur de la première promesse qui a été **acquittée** (ici la deuxième qui est **tenue**).

```
Promise.race([
    new Promise(resolve => setTimeout(resolve, 3000, 1)),
    new Promise((resolve,reject) => setTimeout(reject, 1000, 2)),
    new Promise(resolve => setTimeout(resolve, 2000, 3))
  ]).then(result => console.log(`Resultat : ${result}`))
    .catch(error => console.log(`Erreur : ${error} !`));
```

```
Erreur : 2 !
```

La valeur de la première promesse qui a **qui a été acquittée** (ici la deuxième qui est **rompue**).

Promises : méthodes statiques

- `Promise.resolve(`*`value`*`)`
  - crée une promesse résolue avec le résultat *`value`*.
- `Promise.reject(error)`
  - crée une promesse rejetée avec comme raison *`error`*.

Faire des requêtes HTTP depuis JavaScript

- historiquement
  - objet `XMLHttpRequest`
    - API basée sur les fonction callbacks
    - bas niveau
    - difficulté d'enchaîner les traitements (*callback hell – pyramid of doom*)
    - voir démo sur MDN
      - https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing#event_handlers

    - utilisation de librairies tierces pour simplifier l'écriture des requêtes HTTP (JQuery, axios …)
      - API AJAX *legacy* de JQuery https://api.jquery.com/category/ajax/
      - API Axios : https://axios-http.com/fr/docs/intro
- avec l'introduction des promesses apparition d'une API standard
  - `fetch`

L'API standard `fetch`

- Le moyen moderne de faire des requêtes HTTP, intégré dans tous les navigateurs modernes.
- syntaxe de base

quand **`fetch`** est invoquée le navigateur débute la requête et retourne un objet **`Promise`** que le code appelant devra utiliser pour récupérer le résultat

```
let promise = fetch(url, [options]);
```

L'URL à laquelle la requête est envoyée

paramètres optionnels permettant de configurer la requête (méthode, headers ...)

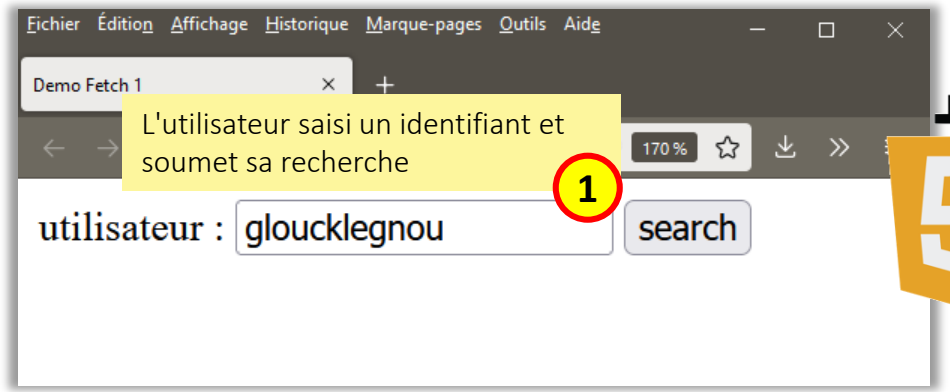
sans **`options`** la requête est un simple **`GET`** chargeant le contenu de l'URL

L'API standard `fetch`

- fonctionnement de `fetch`
  - récupérer la réponse se fait en général en deux temps
    1. dès que le serveur à envoyé les en tête de réponse, la promesse retournée par `fetch` est **tenue**, sa valeur est alors un objet de type `Response` (type prédéfini *built-in class*) .
       - A ce stade possibilité de vérifier le code HTTP de statut de la réponse et de consulter les en-têtes de réponse , mais le corps (body) de la réponse n'est pas encore accessible
       - la promesse retournée par `fetch` n'est **rompue** (provoque une erreur) que si la requête réseau n'a pas aboutie (problème réseau, site de l'URL inexistant…). Les codes d'erreur HTTP (4xx, 5xxx) doivent eux être traités dans la résolution de la requête.
    2. pour obtenir le corps de la réponse, il est nécessaire de faire un appel de méthode supplémentaire.
       - l'objet réponse propose plusieurs méthodes basées sur les promesses permettant d'accéder au corps de la réponse selon différents formats
         - `response.text()` lit le corps de la réponse et le renvoie sous forme de chaîne de caractères.
           - Elle renvoie une promesse qui se résout avec le texte de la réponse.
           - Utile pour traiter des réponses qui ne sont pas au format JSON, comme du HTML ou du texte brut.
         - `response.json()` lit le corps de la réponse et le parse en JSON.
           - Elle renvoie une promesse qui si elle est **tenue** (fulfilled) se résout avec le résultat de la conversion JSON.
           - Si le corps de la réponse n'est pas un JSON valide, la promesse est **rompue** (*rejected*) avec une erreur.
         - `response.blob()` retourne la réponse sous une forme binaire (*Blob* : *Binary Large Object*), exemple une image
         - …

L'API standard fetch

- exemple – aller chercher la localité associée à un utilisateur de GitHub via l'API GitHub



Le code JavaScript récupère la valeur de l'input et envoie un requête à l'API GitHub

2

GET https://api.github.com/users/glouck legnou

https://docs.github.com/en/rest

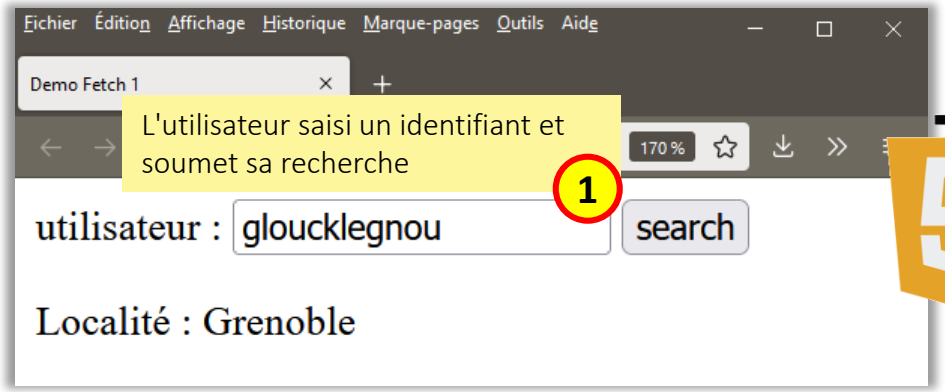
GitHub

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Demo Fetch 1</title>
</head>

<body>
    <div class="container">

        <label>utilisateur : </label> <input id="userId" type="text">
        <button id="searchBtn">search</button>
        <br>
        <p id="resultat"></p>
    </div>
    <script src="demoFetchPromise.js"></script>
</body>

</html>
```

L'API standard fetch

- exemple – aller chercher la localité associée à un utilisateur de GitHub via l'API GitHub

https://docs.github.com/en/rest



Le code JavaScript récupère la valeur de l'input et envoie un requête à l'API GitHub

2

GET https://api.github.com/users/glougloulegnou

GitHub

État 200 OK
Version HTTP/2

3

GitHub API renvoie la réponse sous la forme d'un objet JSON

content-security-policy: default-src 'none'
content-type: application/json; charset=utf-8
date: Thu, 03 Mar 2022 16:51:52 GMT

en tête de la réponse

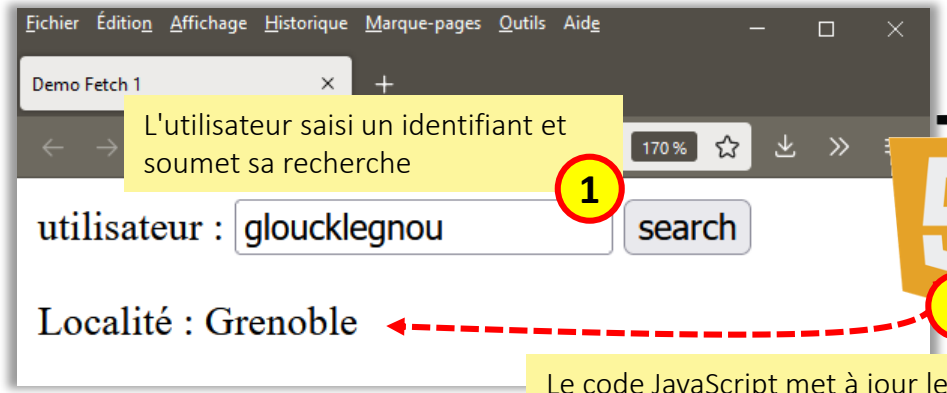
```
{
  "login": "glougloulegnou",
  "id": 7489084,
  "node_id": "MDQ6VXNlcjc0ODkwODQ=",
  "avatar_url": "https://avatars.githubusercontent.com/u/7489084?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/glougloulegnou"
  "blog": "",
  "location": "Grenoble",
  "email": null,
  "hireable": null
```

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Demo Fetch 1</title>
</head>

<body>
    <div class="container">

        <label>utilisateur : </label> <input id="userId" type="text">
        <button id="searchBtn">search</button>
        <br>
        <p id="resultat"></p>
    </div>
    <script src="demoFetchPromise.js"></script>
</body>

</html>
```

L'API standard fetch

- exemple – aller chercher la localité associée à un utilisateur de GitHub via l'API GitHub

https://docs.github.com/en/rest



Le code JavaScript récupère la valeur de l'input et envoie un requête à l'API GitHub

2

GET https://api.github.com/users/glougloulegnou

État 200 OK
Version HTTP/2

3

GitHub API renvoie la réponse sous la forme d'un objet JSON

content-security-policy: default-src 'none'
content-type: application/json; charset=utf-8
date: Thu, 03 Mar 2022 16:51:52 GMT

en tête de la réponse

4

Le code JavaScript met à jour le contenu de la page

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Demo Fetch 1</title>
</head>

<body>
    <div class="container">

        <label>utilisateur : </label> <input id="userId" type="text">
        <button id="searchBtn">search</button>
        <br>
        <p id="resultat"></p>
    </div>
    <script src="demoFetchPromise.js"></script>
</body>

</html>
```

```
{
  "login": "glougloulegnou",
  "id": 7489084,
  "node_id": "MDQ6VXNlcjc0ODkwODQ=",
  "avatar_url": "https://avatars.githubusercontent.com/u/7489084?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/glougloulegnou"
  "blog": "",
  "location": "Grenoble",
  "email": null,
  "hireable": null
```

L'API standard `fetch`

- exemple – aller chercher la localité associée à un utilisateur de gitHub via l'API gitHub

en utilisant la syntaxe des promesses

```
const userInput = document.getElementById("userId");

document.getElementById("searchBtn").addEventListener("click", () => {
    fetch(`https://api.github.com/users/${userInput.value}`)
        .then(response => {
            if (response.ok) {
                return response.json();
            }
            else {
                throw new Error("Error HTTP " + response.status);
            }
        })
        .then(user => {
            document.getElementById("resultat").innerHTML =
                "Localité : " + user.location;
        })
        .catch(
            e => {
                document.getElementById("resultat").innerHTML = e.message;
            }
        );
});
```

on verra plus tard comment encore simplifier cette écriture en utilisant la syntaxe **`async await`**

L'API standard `fetch`

une seule des méthode de consommation d'une réponse d'un `fecth` peut être utilisée à la fois

```
let p = fetch("https://api.github.com/users/gloucklegnou")
p.then(function(response) {
        ...
    ✅ response.json (); // consommation du corps de la réponse
  });
p.then(function(response) {
        ...
    ❌ response.text();  // échec, la réponse a déjà été consommée
  });
```

- accéder aux en-têtes des réponses
  - utilisation de l'objet `response.headers` qui se comporte comme une `Map`

```
fetch("https://api.github.com/users/gloucklegnou")
  .then(function(response) {

        console.log(response.headers.get('Content-Type')); // application/json; charset=utf-8

        for (let [key, value] of response.headers) {
           console.log(`${key} = ${value}`);
        }
        ...
   })
```

L'API standard `fetch`

- Fixer les en-tête des requêtes
  - utiliser le paramètre **`options`**

```
fetch(protectedUrl, {
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer your-token-here',
    'Accept': 'application/json'  }
}).then(...)
```

  - certains en-têtes ne peuvent être initialisés avec le paramètre options (ils sont contrôlés exclusivement par le navigateur)
    - `Accept-Charset, Accept-Encoding, Access-Control-Request-Headers, Access-Control-Request-Method, Connection, Content-Length, Cookie, Cookie2, Date, DNT, Expect, Host, Keep-Alive, Origin, Referer, TE, Trailer, Transfer-Encoding, Upgrade, Via, Proxy-*, Sec-*`

L'API standard `fetch`

- Pour faire une requête autre que **`GET`** il faut utiliser le champ **`method`** de l'objet **`options`**
- Si la requête a un corps il faut utiliser le champ **`body`** de l'objet **`options`** et préciser son type MIME avec un en-tête de requête **`Content-Type`**

- exemple d'une requête **POST**

```
let user = {
  name: 'Joe',
  surname: 'Moose'
};

fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
}).then(...)
```

Async / Await

- Promises ont permis d'encadrer le développement de code asynchrone,
- utilisées dans de nombreuses API
- mais n'ont pas permis de résoudre pleinement les problèmes de lisibilité du code lié à l'utilisation des fonctions callback

➔ introduction dans ES7 (2017) d'une syntaxe spéciale pour travailler de manière plus confortable avec les Promises : mots clés **`async`** et **`await`**

sucre syntaxique (*syntactic sugar*) : extensions à la syntaxe d'un langage de programmation qui

1. ne modifient pas son expressivité (n'ajoutent pas de nouvelles fonctionnalités)
2. facilitent l'écriture et augmentent la lisibilité

ex : en C **`tableau[i]`** au lieu de **`*(tableau+i)`**

Async / Await

- fonctions asynchrones
- `async` placé devant la déclaration d'une fonction (ou une expression de fonction, ou encore une fonction fléchée) pour la transformer en fonction asynchrone
  - la fonction va toujours retourner une promesse
  - cette promesse sera **tenue** avec la valeur renvoyée par la fonction ou **rompue** si il y a une exception non interceptée émise depuis la fonction asynchrone

Dans le cas où la fonction retourne une valeur qui n'est pas une promesse, alors cette valeur sera automatiquement enveloppée dans une promesse.

```
async function hello(nom) {
    return "hello " + nom;
}
```

⇔

```
function hello(nom) {
    return new Promise(
        (resolve, reject) => resolve("hello " + nom)
    );
}
```

hello() retourne une promesse, on peut appeler **then**

```
hello("world")
    .then((result) => console.log(result));
```

→ hello world

Async / Await

- `await`
  - ne peut être utilisé que dans les fonctions définies avec `async`
  - permet d'interrompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée.

```
async function testAwait(){
    console.log("début test await");
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve('Je suis résolue !'), 2000)
    });

    let result = await promise;
    console.log(result);
}

testAwait();
console.log("****");
```

Attend que la promesse soit résolue ou rejetée puis retourne le résultat de la promesse

le code équivalent avec les promesses

```
function testAwait(){
    console.log("début test await");
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve('Je suis résolue !'), 2000)
    });

    promise.then(result => console.log(result))

}

testAwait();
console.log("****");
```

```
début test await
****
Je suis résolue !
```

`await` est une syntaxe alternative à `then()`, plus facile à lire, à comprendre et à écrire : nous pouvons écrire séquentiellement des traitements qui sont en fait asynchrones

ne consomme aucune ressource supplémentaire, le moteur JavaScript peut effectuer d'autres tâches en attendant : exécuter d'autres scripts, gérer des événements, etc.

Async / Await : gestion des erreurs

- dans un fonction `async` l'expression `await promise`
  - retourne le résultat de la promesse si celle-ci est **tenue** (*fulfilled*),
  - lance une erreur (exception) si la promesse est **rompue** (*rejected*).
    - c'est comme si il y avait une instruction `throw` sur cette ligne
  - au sein de la fonction, l'erreur peut être interceptée de manière classique par un bloc `try/catch`
  - si l'erreur n'est pas traitée au sein de la fonction `async`, la promesse générée par l'appel à cette fonction devient **rompue** (*rejected*). On peut adjoindre à cet appel un `.catch()` pour gérer ce rejet.

Async /Await : gestion des erreurs

```
const fetch = require('node-fetch');

async function testUrl(url) {
    let response = await fetch(url);
    console.log(`ressource ${url} trouvée`);
}
```

```
testUrl('http://no-such-url');
testUrl('https://www.univ-grenoble-alpes.fr');
```

```
async function f() {
  await asyncFetch('http://no-such-url');
  await asyncFetch('https://www.univ-grenoble-alpes.fr');
}

f();
```

pour séquencer les traitements passer par un fonction asynchrone

`testURL` est asynchrone, l'exécution des appels à `testUrl` n'est pas pas forcément séquentielle

```
λ node awaitFetchError1.js
ressource https://www.univ-grenoble-alpes.fr trouvée
(node:2076) UnhandledPromiseRejectionWarning: FetchError: request to http://no-such-url/ failed, reason: getaddrinfo
ENOTFOUND no-such-url
    at ClientRequest.<anonymous> (P:\ENSEIGNEMENT\JavaScript\VueJS\JSTutoAsync\setTimeout\node_modules\node-
fetch\lib\index.js:1491:11)
    at ClientRequest.emit (events.js:314:20)
    at Socket.socketErrorListener (_http_client.js:427:9)
    at Socket.emit (events.js:314:20)
    at emitErrorNT (internal/streams/destroy.js:92:8)
    at emitErrorAndCloseNT (internal/streams/destroy.js:60:3)
    at processTicksAndRejections (internal/process/task_queues.js:84:21)
```

Async /Await : gestion des erreurs

```
const fetch = require('node-fetch');

async function testUrl(url) {

    try {
      let response = await fetch(url);
      console.log(`ressource ${url} trouvée`);
    } catch(err) {
       console.log(err.message); // TypeError: failed to fetch
       console.log(`ressource ${url} non trouvée`);
    }
  }

async function f() {
  await testUrl('http://no-such-url');
  await testUrl('https://www.univ-grenoble-alpes.fr');
}

f();
```

utilisation d'un bloc `try/catch` pour gérer l'exception

```
λ node awaitFetchError2.js
ressource https://www.univ-grenoble-alpes.fr trouvée
request to http://no-such-url/ failed, reason: getaddrinfo ENOTFOUND no-such-url
ressource http://no-such-url non trouvée
```

Async /Await : gestion des erreurs

```
const fetch = require('node-fetch');

async function testUrl(url) {
    let response = await fetch(url);
    console.log(`ressource ${url} trouvée`);
}

async function f() {
    await testUrl('http://no-such-url');
    await testUrl('https://www.univ-grenoble-alpes.fr');
}

try {
    f();
} catch (error) {
    console.log("Erreur dans f() " + error.message);
}
```

l'exception n'est pas attrapée, pourquoi ?

```
λ node awaitFetchError3.js
(node:7332) UnhandledPromiseRejectionWarning: FetchError: request to http://no-such-url/ failed
    at ClientRequest.<anonymous> (P:\ENSEIGNEMENT\JavaScript\VueJS\JSTutoAsync\setTimeout\node_
    at ClientRequest.emit (events.js:314:20)
    at Socket.socketErrorListener (_http_client.js:427:9)
    at Socket.emit (events.js:314:20)
    at emitErrorNT (internal/streams/destroy.js:92:8)
    at emitErrorAndCloseNT (internal/streams/destroy.js:60:3)
    at processTicksAndRejections (internal/process/task_queues.js:84:21)
(node:7332) Unhandled promise rejection warning. Unhandled promise rejection. This error originat
```

```
const fetch = require('node-fetch');

async function testUrl(url) {
    let response = await fetch(url);
    console.log(`ressource ${url} trouvée`);
}

async function f() {
    await testUrl('http://no-such-url');
    await testUrl('https://www.univ-grenoble-alpes.fr');
}

f().catch(
    (error) =>
        console.log("Erreur dans f() " + error.message)
);
```

pour gérer l'erreur au niveau global, il faut utiliser `catch()` de l'API **Promise**

```
λ node awaitFetchError4.js
Erreur dans f() request to http://no-such-url/ failed, reason: getaddrinfo ENOTFOUND no-such-url
```

les appels d'une fonction asynchrone renvoient une promesse :
Résolue (fulfilled) : si il n'y pas eu d'erreur
Rompue (rejected) : en cas d'erreur

L'API standard `fetch`

- exemple – aller chercher la localité associée à un utilisateur de gitHub via l'API gitHub

en utilisant la syntaxe des promesses

```
const userInput = document.getElementById("userId");

document.getElementById("searchBtn").addEventListener("click", () => {
    fetch(`https://api.github.com/users/${userInput.value}`)
        .then(response => {
            if (response.ok) {
                return response.json();
            }
            else {
                throw new Error("Error HTTP " + response.status);
            }
        })
        .then(user => {
            document.getElementById("resultat").innerHTML =
                "Localité : " + user.location;
        })
        .catch(
            e => {
                document.getElementById("resultat").innerHTML = e.message;
            }
        );
});
```

en utilisant la syntaxe **`async await`**

```
const userInput = document.getElementById("userId");

document.getElementById("searchBtn").addEventListener("click", async () => {
    let response =
        await fetch(`https://api.github.com/users/${userInput.value}`);
    if (response.ok) {
        let user = await response.json();
        document.getElementById("resultat").innerHTML =
            "Localité : " + user.location;
    }
    else {
        document.getElementById("resultat").innerHTML =
            "Error HTTP " + response.status;
    }
});
```

Promises (promesses)

- Promesses (**Promises**) à la base de la programmation asynchrone en JavaScript moderne.
- Une promesse est un objet renvoyé par une fonction asynchrone, qui représente l'état actuel de l'opération.
  - Opération en cours (promesse **en attente** - *pending*) ;
  - Opération terminée (promesse **acquittée** - *settled*))
    - avec succès (promesse **tenue** - *fulfilled :* un résultat est disponible) ;
    - stoppée après un échec (promesse **rompue** - *rejected :* un erreur indique la cause de l'échec)
- Au moment où la promesse est renvoyée à l'appelant, l'opération n'est souvent pas terminée, mais l'objet promesse fournit des méthodes permettant de lui attacher des gestionnaires (*handlers*) qui seront exécutés lorsque la promesse sera acquittée pour gérer le succès ou l'échec éventuel de l'opération.

Avec fonctions callback

```
function asyncOp1(callback1) {
    // ... traitement asynchrone
    // quand le traitement est terminé
    // appel de la fonction callback avec
    // résultats du traitement
    callback1(res1);
}

function op2(params) {
    // ...
}

asyncOp1(op2);
```

Avec les promesses

```
function asyncOp1() {
    // lance le traitement asynchrone
    // et retourne un objet Promesse qui,
    // lorsqu'elle sera tenue,
    // contiendra le résultat du traitement
}

function op2(params) {
    //…
}

let p1 = asyncOp1();
p1.then(op2);
```

le traitement asynchrone est lancé et une promesse est retournée

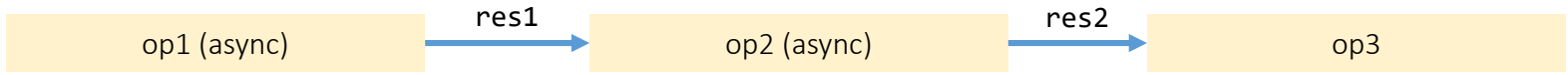
le fonction op2 est enregistrée auprès de p1 et sera exécutée avec la valeur retournée par op1 lorsque celle-ci aura terminé son traitement

```
asyncOp1()
  .then(op2);
```

pas besoin de passer par une variable intermédiaire

Promises(promesses)

- Enchainer des traitements asynchrones



Avec fonctions callback

```
function asyncOp1(callback1) {
    // ... traitement asynchrone
    // qd le traitement est terminé
    // appel de la fonction callback avec
    // résultats du traitement
    callback1(res1);
}

function asyncOp2(params, callback2) {
    // ...
    callback2(res2);
}

function op3(data) {
   //...
}

function op2PuisOp3(params) {
    asyncOp2(params, op3);
}

asyncOp1(op2PuisOp3);
```

Avec les promesses

```
function asyncOp1() {
    // lance le traitement asynchrone op1
    // et retourne un objet Promesse qui contiendra lorsqu'elle sera
    // tenue le résultat du traitement
}

function asyncOp2(op) {
    // lance le traitement asynchrone op2
    // et retourne un objet Promesse qui contiendra lorsqu'elle sera
    // tenue le résultat du traitement
}

function op3(data) {
   //...
}

asyncOp1()                    ⇔    let p1 = asyncOp1();
  .then(asyncOp2)                  let p2 = p1.then(asyncOp2)
  .then(op3);                      p2.then(op3);
```

then renvoie elle-même un promesse
ce qui permet d'enchaîner facilement les traitements