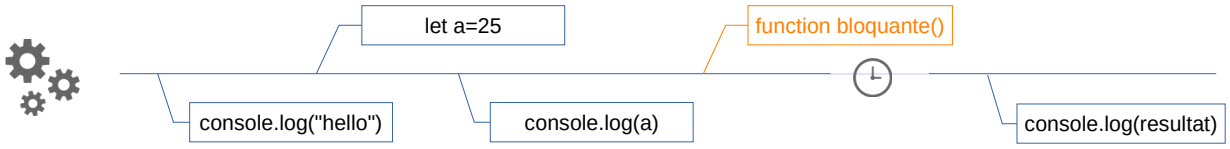
1.1 BOUCLE D'ÉVÉNEMENTS

JavaScript n'utilise qu'un seul thread pour fonctionner, ce qui signifie que les choses n'arrivent que les unes après les autres, dans l'ordre des instructions.
Ce mode de fonctionnement augmente le risque de blocage du programme par une seule instruction ou fonction qui serait en attente d'un élément long (un téléchargement par exemple).



Dans un navigateur, chaque onglet dispose de sa propre boucle, afin qu'une page ne bloque pas l'ensemble des onglets. Il s'agit d'une mesure de sécurité pour isoler les scripts de chaque page, mais elle ne protège pas du blocage d'un script qui s'exécute sur une seule page qui comporte plusieurs scripts (exemple : site web avec des publicités et des informations provenant d'autres sites).

Pour éviter les blocages, les moteurs JavaScript implémentent des primitives[1] non-bloquantes, utilisant des fonctions de retour (ou de rappel) : les "callbacks".
Par exemple, les fonctions d'entrées/sorties (I/O), les fonctions réseaux, la gestion des fichiers, etc.

Ces fonctions de retour ne s'exécuteront que lorsque un événement surviendra : un déclencheur.
Un exemple simple d'utilisation des fonctions de retour, est celui de l'usage du bouton et du clic.

```
// création d'un événement
bouton.onclick = function () {
      console.log("bouton cliqué à l'instant")
}
```

ou

```
// création d'un événement (méthode 2)
bouton.addEventListener("click", function () {
      console.log("bouton cliqué à l'instant")
}
```

Ainsi, le moteur JavaScript utilise une boucle d'événements qui scrute en permanence la pile d'appels, *call stack*.

1 instructions de base

1.1.1 Pile d'appels

Il s'agit d'une pile de type LIFO. Une pile est réellement un empilement de fonctions appelées par les instructions du programme, dans l'ordre des lignes. Pour rappel :

- LIFO : Last In, First Out
- FIFO : First In, First Out

Le code suivant permet de comprendre le fonctionnement de cette pile.

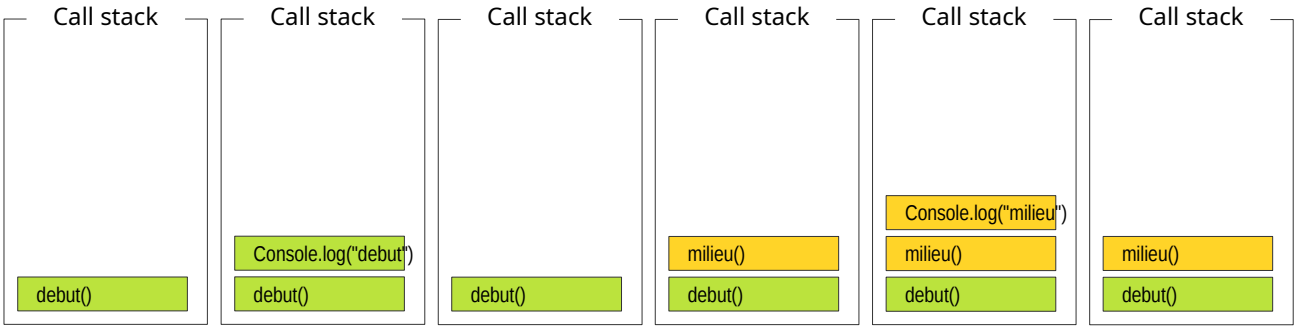
```
// création des fonctions
const milieu =      ()=>   console.log('milieu')
const fin    =      ()=>   console.log('fin')
const debut  =      ()=>   {
      console.log('début')
      milieu()
      fin()
}

// exécution du code
debut()
```

Ainsi, lorsque la fonction debut() est appelée, elle va lancer la fonction console.log() puis la fonction milieu() et enfin la fonction fin().

Le schéma ci-dessous représente les étapes du début du programme :

- JavaScript place la fonction debut() dans la pile et l'ouvre
  - il place la première et seule fonction console.log("début")
  - console.log() ne contenant aucune fonction, est exécuté (et donc enlevé de la pile)
- de retour dans debut(), JavaScript passe à la commande suivante : milieu()
  - la fonction milieu() est lue et la première fonction est empilée : console.log("milieu")
  - console.log() ne contenant aucune fonction, est exécuté (et donc enlevé de la pile)
  - milieu() n'ayant plus d'autres fonctions, milieu() sera dépilé
- et ainsi de suite (avec la fonction fin())



- …

Le programme se termine lorsque la pile est vide.

Le code est ici très simple et les fonctions sont synchrones. Si nous introduisons une fonction asynchrone (par exemple un timer, comme *setTimeout(fonction, intervalle)*), l'empilement ne sera plus correct.
Le code suivant permet de comprendre le fonctionnement modifié de cette pile.

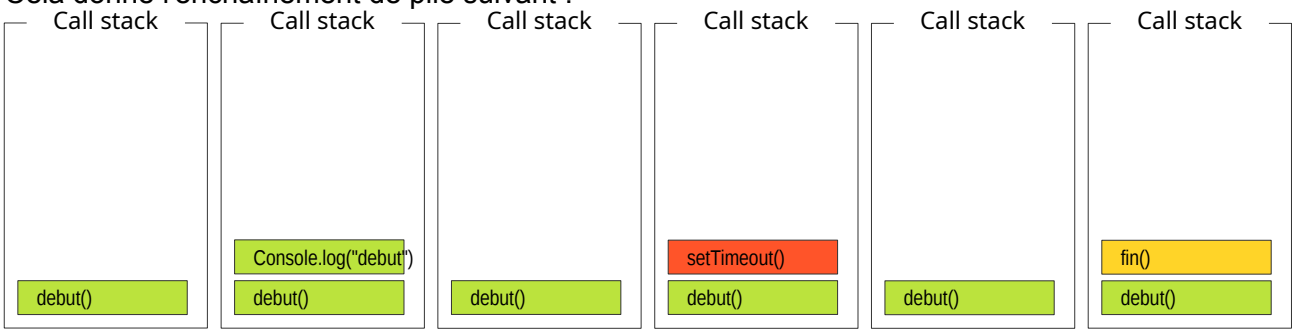
```
// création des fonctions
const milieu =      ()=>   console.log('milieu')
const fin    =      ()=>   console.log('fin')
const debut  =      ()=>   {
      console.log('début')
      setTimeout(milieu)
      fin()
}

// exécution du code
debut()
```

Ainsi, lorsque la fonction debut() est appelée, elle va lancer la fonction console.log() puis la fonction setTimeout(), ensuite la fonction fin() et seulement après, la fonction milieu().
Ce code affichera :

```
début
fin
milieu
```

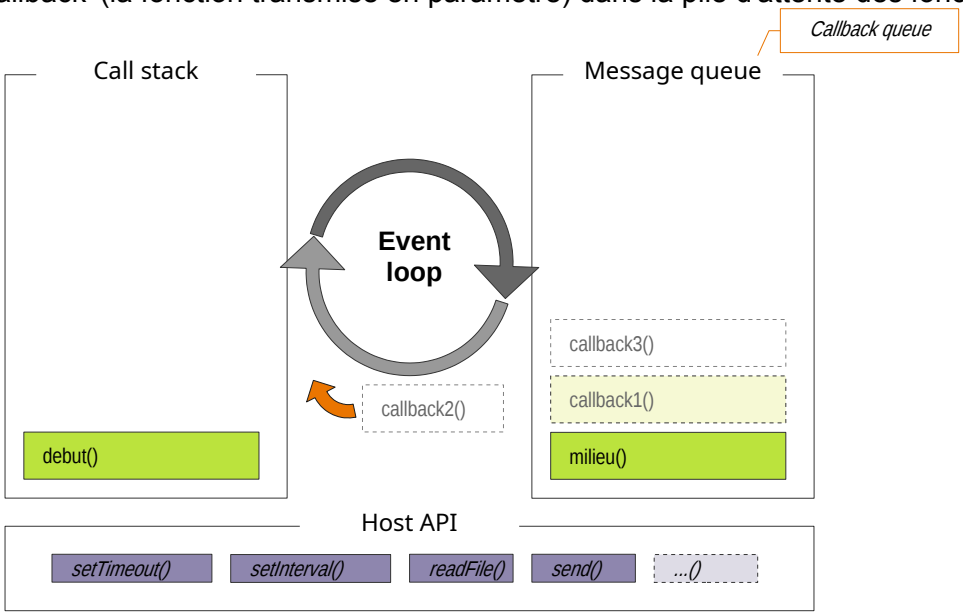
Cela donne l'enchaînement de pile suivant :



La fonction setTimeout() place en effet un événement dans une autre file d'attente : la file d'attente des messages (*message queue*). Cette file gère les événements : clic souris, focus, clavier… mais elle n'est pas prioritaire sur la pile d'attente des fonctions !

1.1.2 Message queue

Cette pile accumule les événements et lorsque l'un d'eux survient, la boucle des événements replace le 'callback' (la fonction transmise en paramètre) dans la pile d'attente des fonctions.



Évidemment, les fonctions asynchrones sont proposées par le moteur JavaScript sous l'appellation "Host API". Une superbe explication vidéo est faite par Philip Roberts↗ à la JSConf.

Cette vidéo explique aussi que la file d'attente est prioritaire sur la callback queue.
https://youtu.be/8aGhZQkoFbQ?t=895

1.2 EXEMPLES

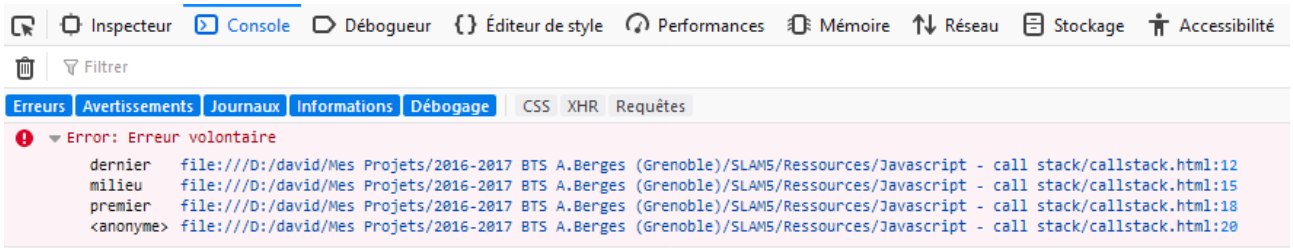
1.2.1 Erreur sur exception

Voici un tout petit script dans une page HTML qui affiche une erreur et… le contenu de la pile d'appel :

callstack.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
	<meta charset="UTF-8">
	<title>Javascript Call Stack error</title>
	<meta name="viewport" content="width=device-width, initial-scale=1.0">
	<link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
	<script>
	function dernier() {
		throw new Error('Erreur volontaire')
	}
	function milieu() {
		dernier()
	}
	function premier() {
		milieu()
	}
	premier()
	</script>
</body>
</html>
```

ouvrez la page dans un navigateur, puis affichez le débogueur (F12) et dépliez l'erreur. Notez quelle est la toute première fonction appelée, il s'agit en fait de notre fonction main() identique à C# ou Java.



1.2.2 Erreur sur dépassement de pile

Voici un deuxième petit script dans une page HTML qui affiche une erreur et… le contenu de la pile d'appel :

callstackloop.html

```
<!DOCTYPE html>
<html lang="fr">
<head>
	<meta charset="UTF-8">
	<title>Javascript Call Stack error</title>
	<meta name="viewport" content="width=device-width, initial-scale=1.0">
	<link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
	<script>
	function encore(n) {
		document.write("["+n+" ]")
		encore(n+1)
	}

	encore(1)
	</script>
</body>
</html>
```

Ouvrez la page dans un navigateur, puis affichez le débogueur (F12) et dépliez l'erreur. Notez la quantité d'appel est affiché dans la page avant d'avoir un message d'erreur.