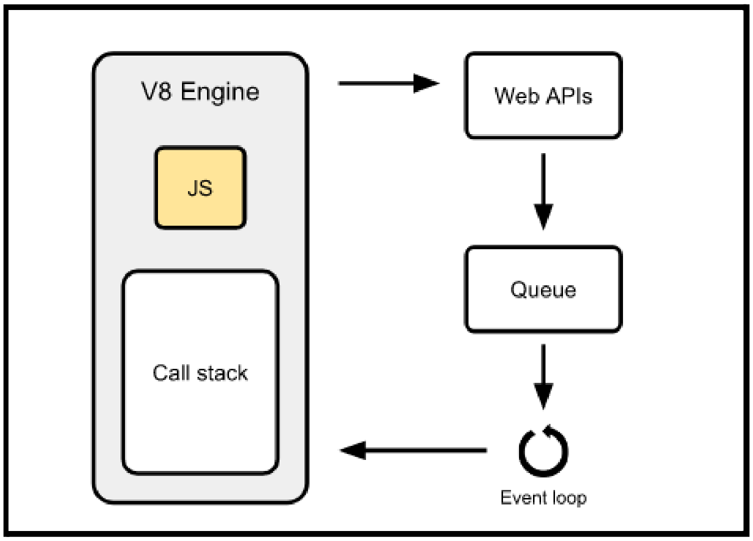
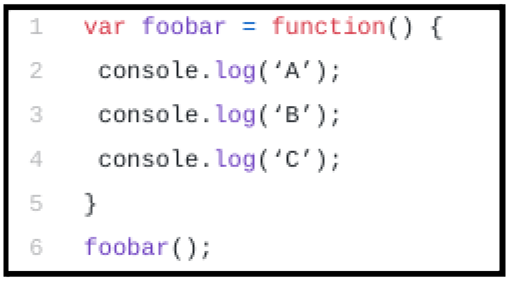
Javascript Execution Model (The Event Loop):

- **Blocking code** is code that runs one instruction after another, and makes next instructions wait.
- **Non-blocking code** means that we don't have to wait for some code to run before running other code.
  I.e. Blocking refers to operations that block further execution until that operation finishes while non-blocking refers to code that doesn't block execution.
- Blocking methods execute synchronously and non-blocking methods execute asynchronously.
- There are two types of function calls:
  - Synchronous calls are pushed to the call stack.
  - Asynchronous calls are pushed to the event queue.
- Example of asynchronous calls are:
  - DOM events (browser)
  - Ajax requests (browser)
  - Timer (browser and NodeJs)
  - Any non-blocking I/O (NodeJs)
  - Promises and async/await
- JavaScript must be compiled and interpreted. It needs a runtime environment.
  In Chrome, the JS runtime is the V8 Engine. Furthermore, Javascript is an event-driven language. It uses the **event loop** to keep track of all of these events. The event loop is a way of scheduling events one after the other. It often gets help from the platform the engine is running on. It is important to note that JS is single-threaded, but that browsers are multi-threaded. Non-blocking JS functions aren't built into the V8 engine. They live in the platform JS is running on, the browser. For example, Chrome contains the instructions for setTimeout, a non-blocking function in JS.
- V8 is a popular open source JS Engine. It is the engine inside Chrome that runs the JavaScript code. But when running JavaScript in the browser, there's more involved than just the JS Engine. Most JavaScript code makes use of web APIs like DOM, AJAX, and Timeout (setTimeout). These APIs are provided by the browser.
- In the picture below we can see an overview of the elements involved in the JS execution in the browser. The V8 Engine contains a call stack. The engine communicates with the web APIs, and there is also a queue and a mysterious event loop.

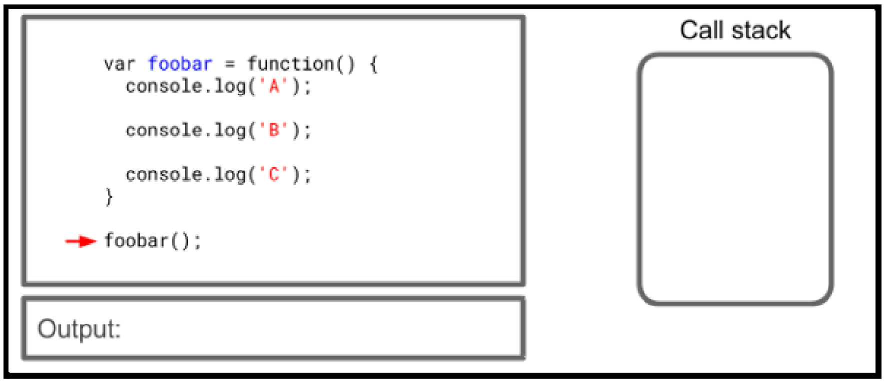


- Javascript is a single threaded programming language, which means it has a single call stack and can do one thing at a time.
- The call stack is a mechanism so the interpreter knows its place in a script. When a script calls a function, the interpreter adds it on the top of the call stack. When the current function is finished, the interpreter takes it off the stack.
- Some terminology:
    - **Call stack:** It's a simple data structure that records where in the code we are currently. So if we step into a function that is a function invocation it is pushed to the call stack. When we return from a function it is popped out of the stack.
    - **Heap:** It's where objects are allocated. Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.
    - **Queue/Callback Queue:** It stores all the callbacks.
      It stores a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.
      At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.
      The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue if there is one.
    - **Event Loop:** This is where all these things come together. The event loop simply checks the call stack, and if it is empty (which means there are no functions in the stack) it takes the oldest callback from the callback queue and pushes it into the call stack which eventually executes the callback.
- E.g. Consider this example:
  We have this code:



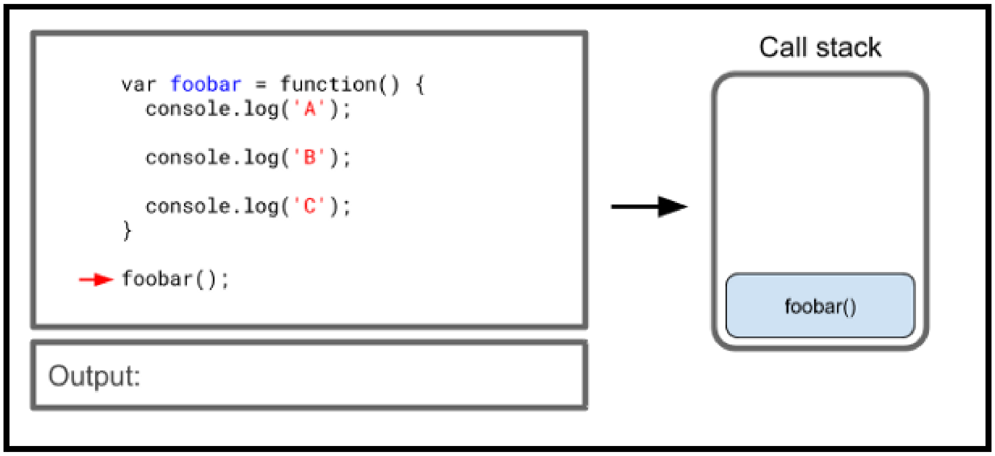
```
var foobar = function() {
 console.log('A');
 console.log('B');
 console.log('C');
}
foobar();
```

Step 1:
The interpreter starts calling foobar.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

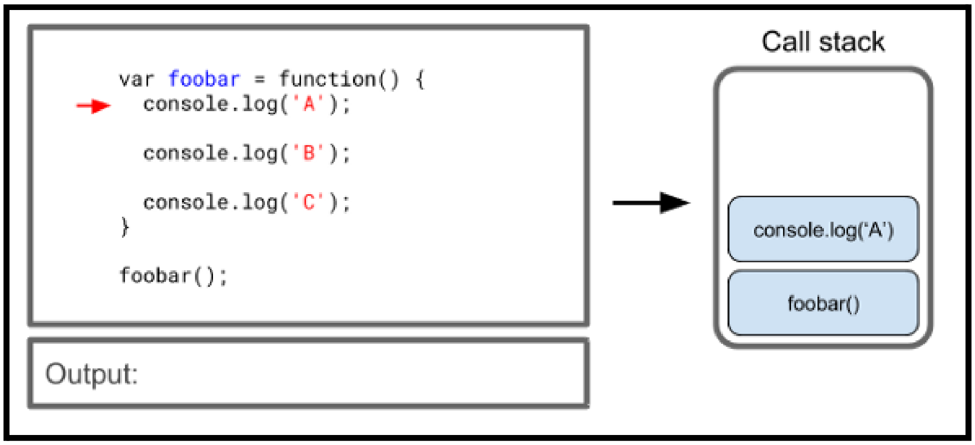
Step 2:
foobar is added to the stack.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

Output:

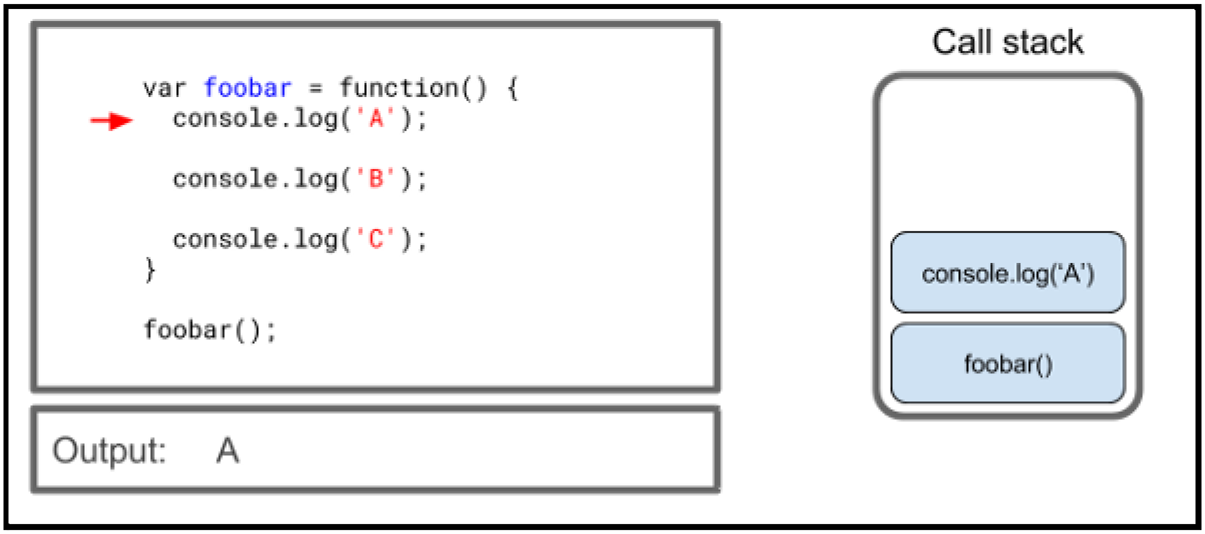
Step 3:
The interpreter steps into the function, and console.log(‘A’) is called and added to the top of the call stack.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

Output:

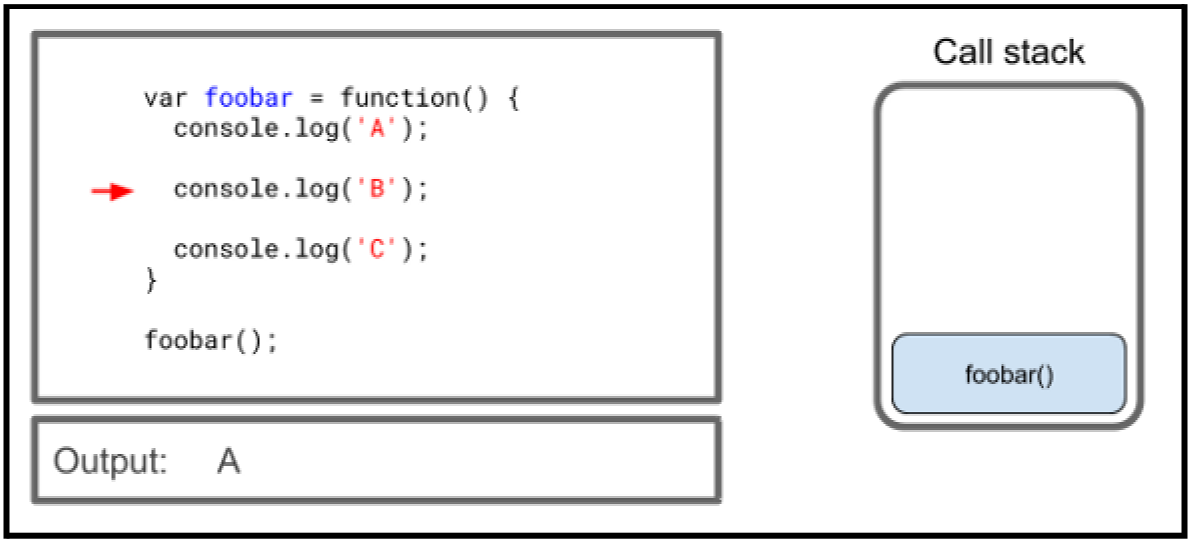
Step 4:
console.log(‘A’) is executed, and the letter “A” is printed in the console.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

Output: A

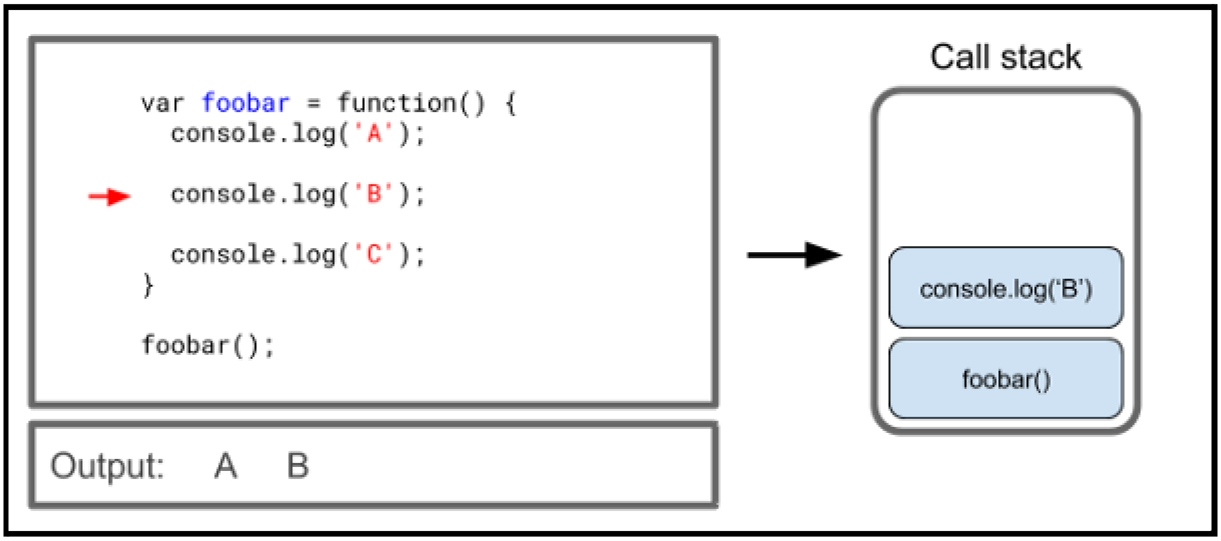
Step 5:
console.log('A') is removed from the call stack since its execution has been completed, and the interpreter moves on to the next command.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A
```

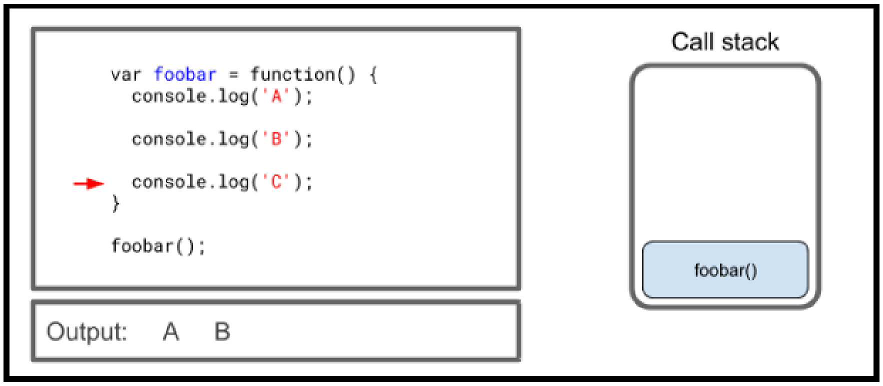
Step 6:
console.log('B') is processed the same way as the previous one, and the letter 'B' is printed in the console.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A    B
```

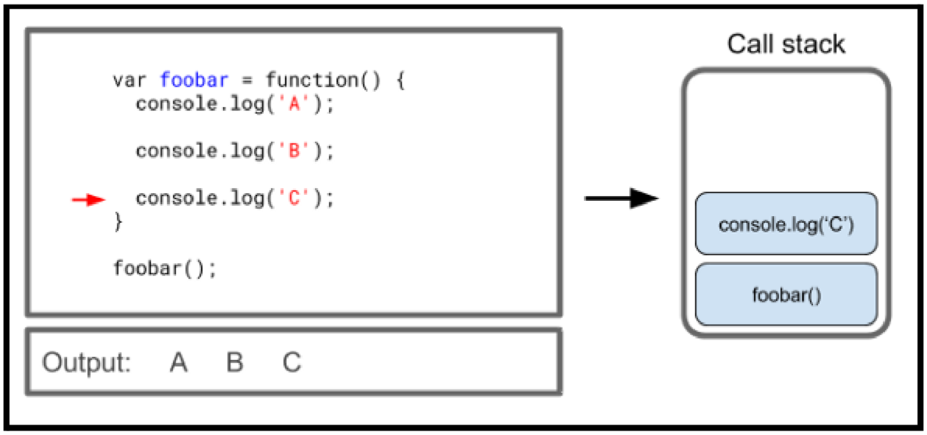
Step 7:
console.log(‘B’) is removed from the stack, and the interpreter moves on.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A    B
```

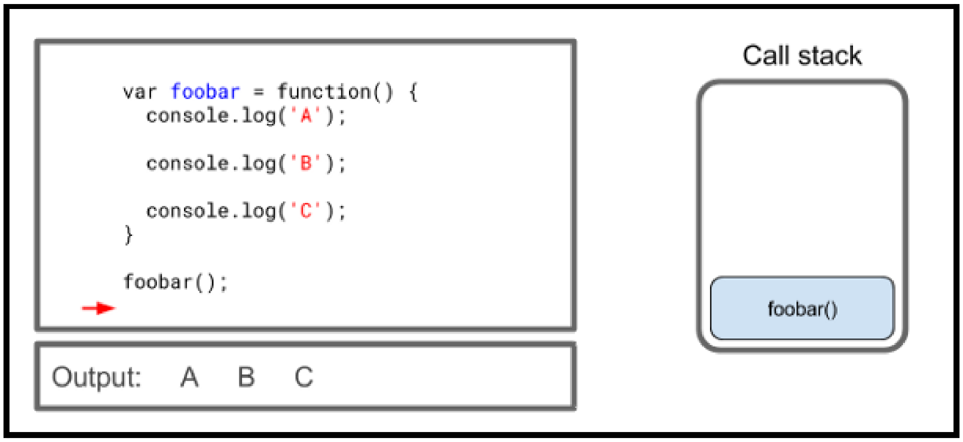
Step 8:
console.log(‘C’) is processed the same way as the previous ones, and the letter 'C' is printed in the console.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A    B    C
```

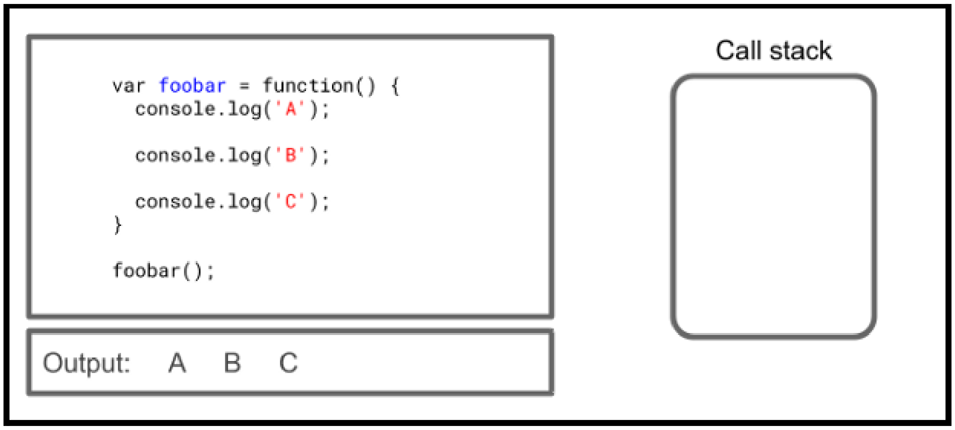
Step 9:
console.log(‘C’) is removed from the stack, and the end of the foobar function is reached.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A    B    C
```

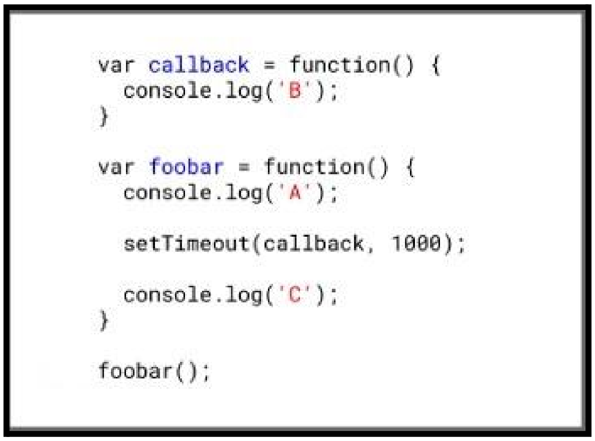
Step 10:
foobar is also removed from the call stack, which becomes empty again.



```
var foobar = function() {
  console.log('A');

  console.log('B');

  console.log('C');
}

foobar();
```

```
Output:    A    B    C
```

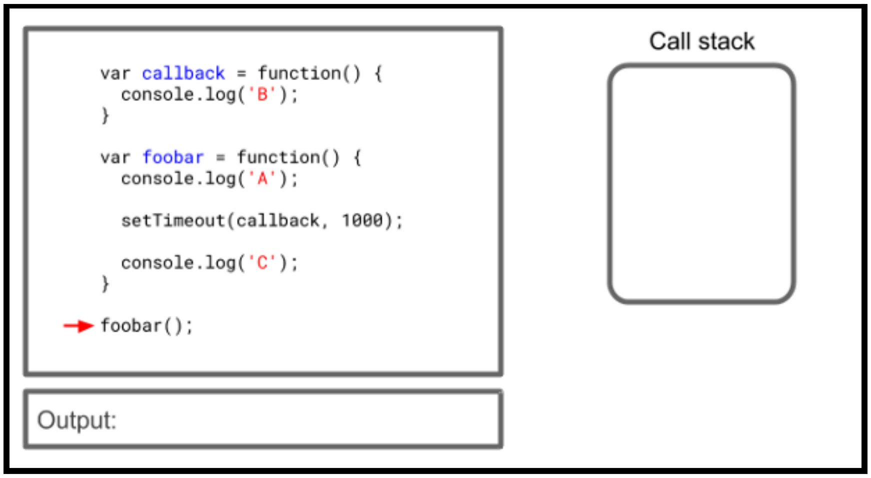
- E.g. Consider this example:
  We have this code:



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

Note: This time, instead of directly doing “console.log(“B”);” in foobar, we have a callback function that does it after waiting 10 seconds.

Step 1:
Now, instead of calling console.log(‘B’) directly, we are putting it inside a callback function that will be called later. The setTimeout is an asynchronous function that we are using to simulate a slow operation. Here again, the interpreter starts by calling foobar.

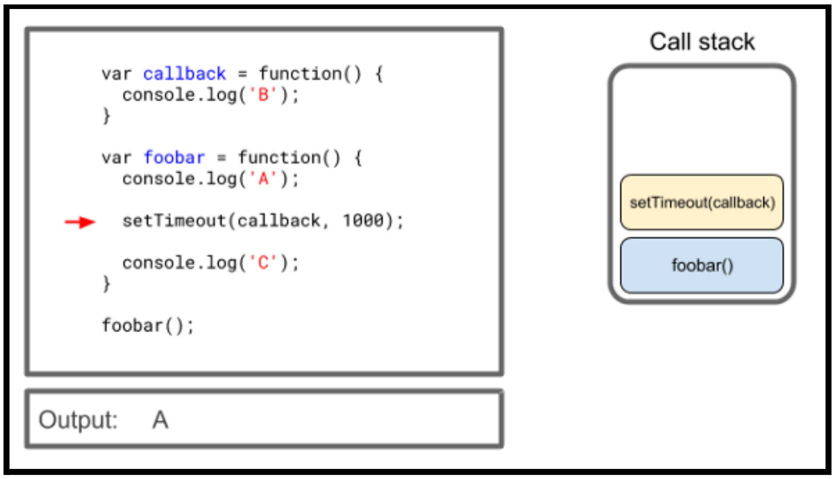


```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

→ foobar();
```

```
Output:
```

Step 2:

The first steps are the same: foobar is added to the call stack, the interpreter steps in, adds console.log(‘A’) to the stack, prints the letter 'A,' and removes the call from the stack. Then the interpreter reaches the setTimeout function and adds it to the call stack.

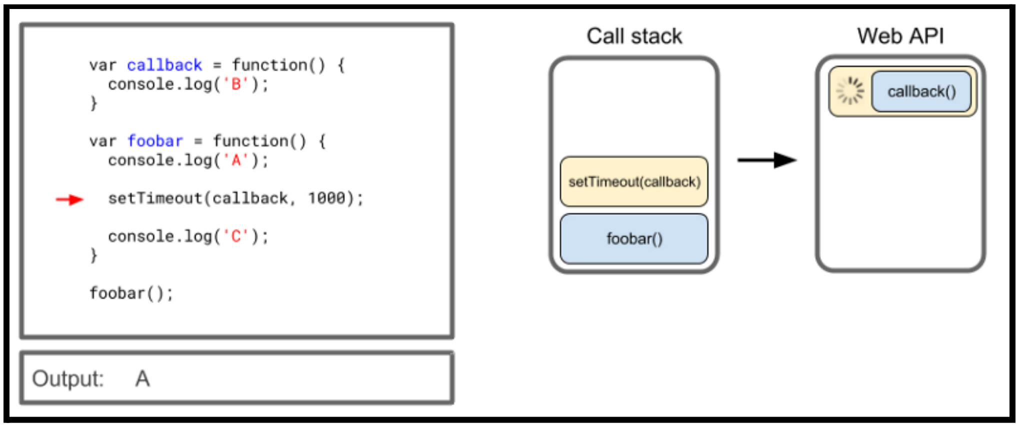


```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:    A
```

Step 3:

Since setTimeout is provided for us by the Web API, it receives the setTimeout call alongside the callback and starts processing it.

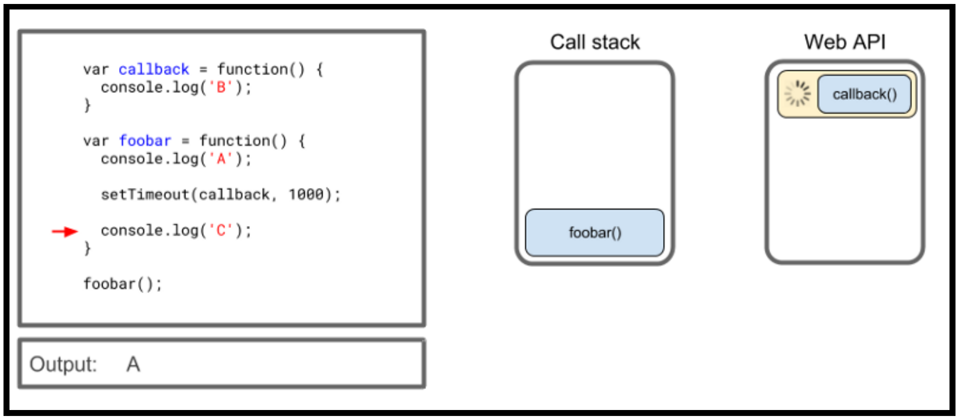


```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:    A
```

Step 4:

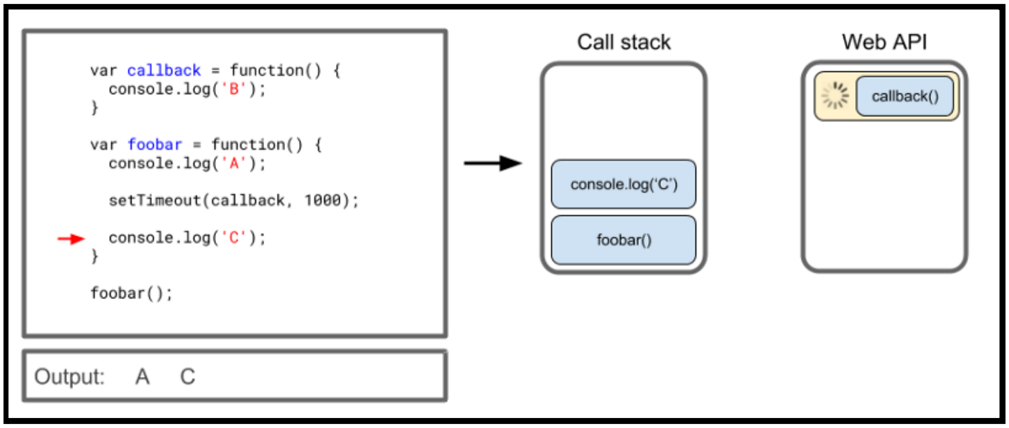
Now the call to setTimeout itself is completed, so it can be removed from the stack. The Web API continues the processing, but the call stack is not blocked anymore, so the interpreter can move to the next command.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:    A
```

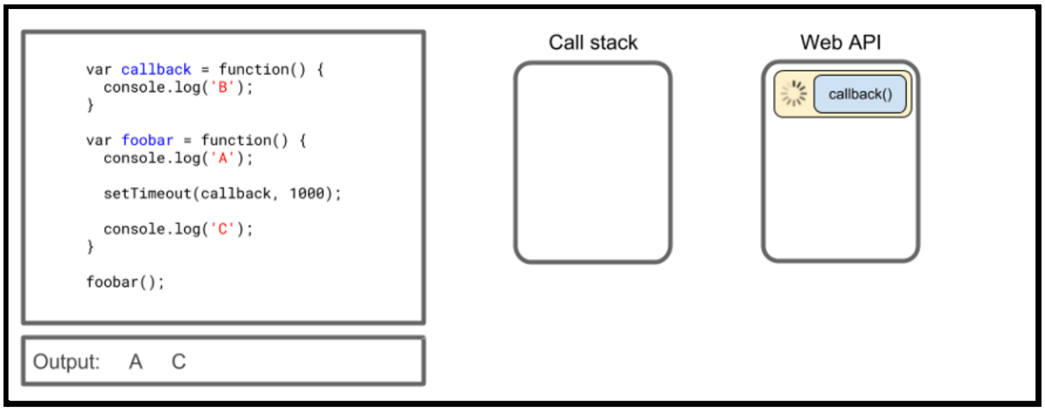
Step 5:
This next step is executed normally, and the letter 'C' is printed to the console.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:   A   C
```

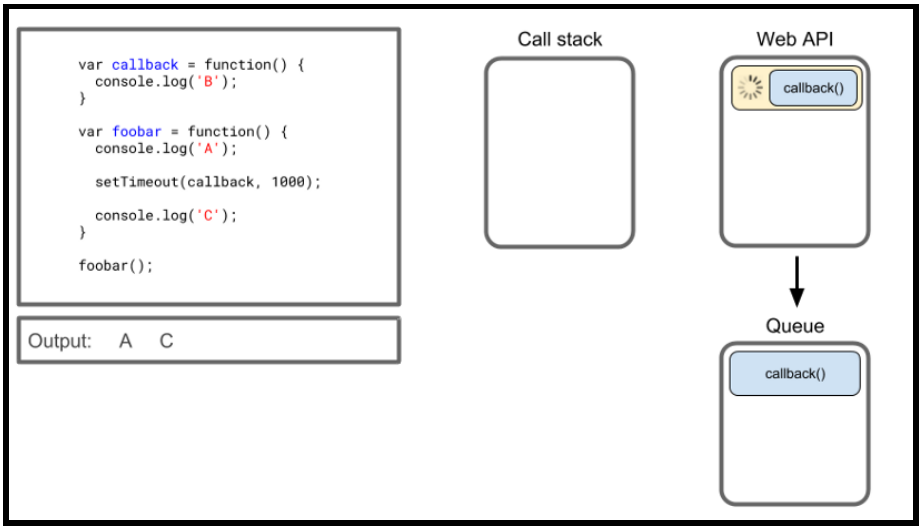
Step 6:
console.log('C') is removed from the call stack, and so is foobar since the interpreter has reached the end of the function. The call stack is empty again.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:   A   C
```

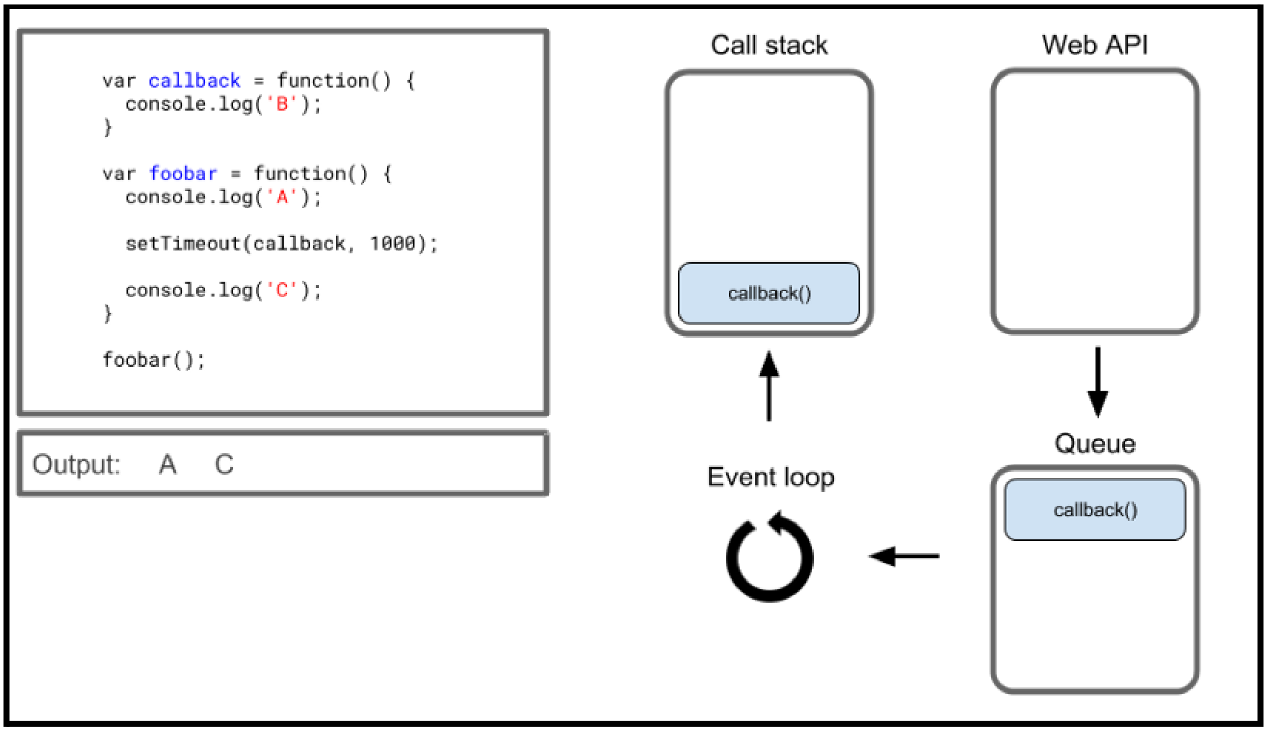
Step 7:
When an asynchronous call is completed, the callback function is pushed to the queue.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:   A   C
```

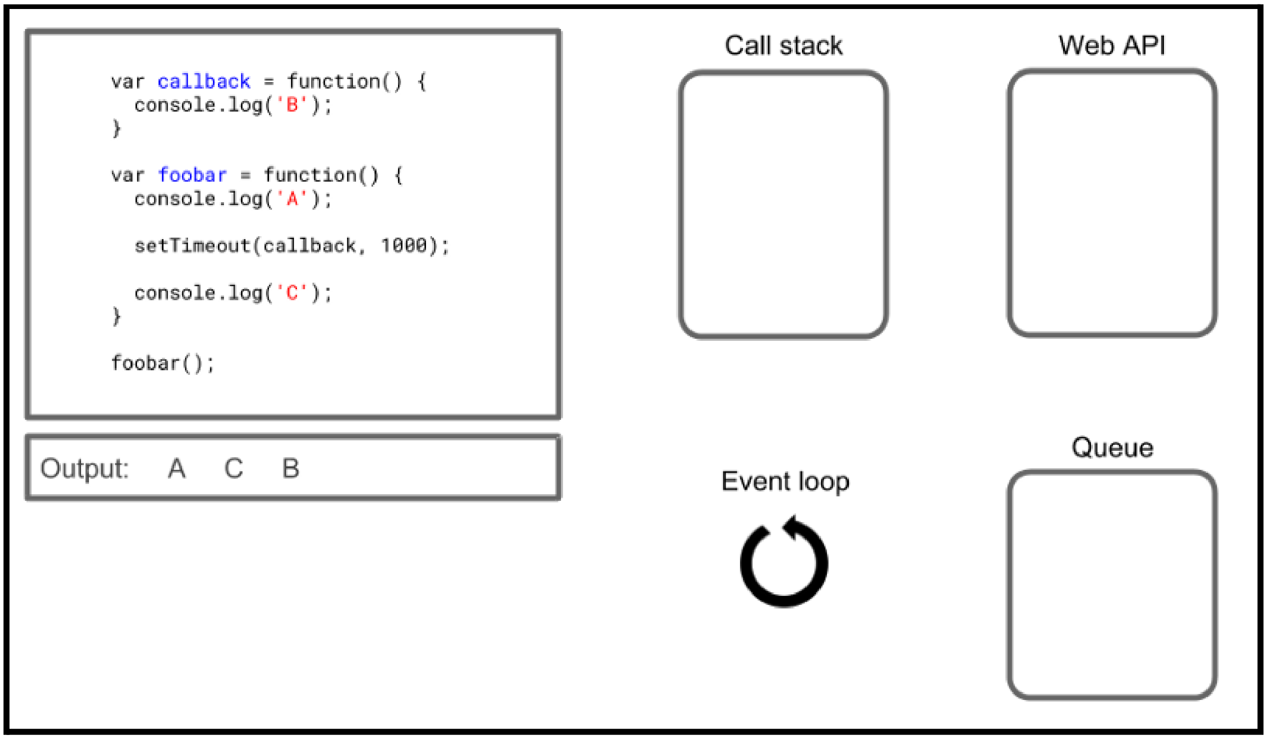
Step 8:
This is where the last piece of the puzzle comes in--the event loop. The event loop has one simple job: it looks at the call stack and the task queue, and if the stack is empty, it takes the first item in the queue and sends it back to the call stack.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:   A   C
```

Step 9:
Finally, the callback is executed, the letter "B" is printed in the console, and the callback is removed from the call stack.



```
var callback = function() {
  console.log('B');
}

var foobar = function() {
  console.log('A');

  setTimeout(callback, 1000);

  console.log('C');
}

foobar();
```

```
Output:   A   C   B
```

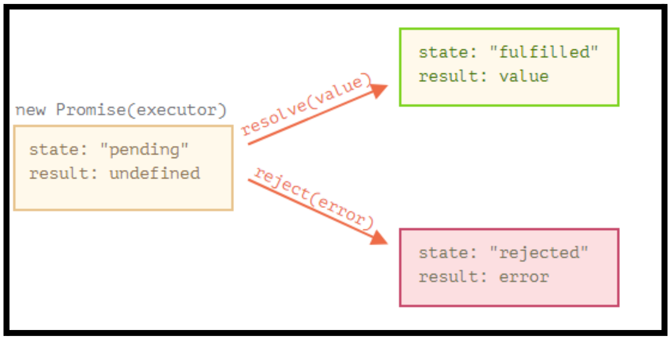
- Here is a good resource that lets you visualize the process: http://latentflip.com/loupe/.
- **Note:** The event loop will look at the queue if the stack is empty.

Multi-Threaded vs Single-Threaded:

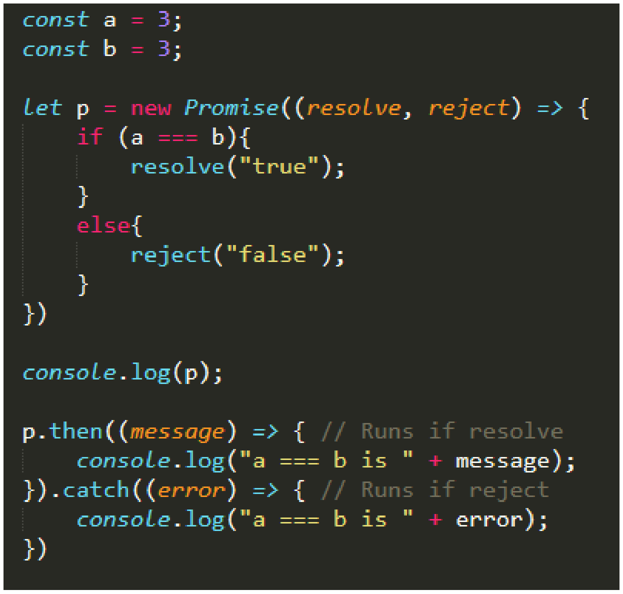
- Multi-threading does not necessarily mean things are executed in parallel. Remember, we only have one CPU.
- We need multi-threading because programming languages have blocking I/O, and by default, programs wait for the I/O to be completed.
- But multi-threading is expensive in terms of software design (synchronization) and in terms of performances (context switch).
- An alternative to multi-threading is single-threaded with non-blocking I/O.
- Running a single-threaded server will have good performance, as long as the requests handlers:
  - Do some asynchronous I/O such as filesys, database, cache, network, etc.
  - Do not do any heavy but yet synchronous computations such as complex math, intensive data processing, etc.
- If needed, Javascript can be multi-threaded.
- Examples of multi-threaded Javascript are Node cluster (NodeJS only) and Web Workers (Browser and NodeJS). They are good for heavy but yet synchronous computations and take advantage of multicore machines.

Dealing with Asynchronism:

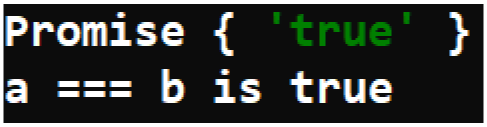
- **Promises:**
- The **Promise** object represents the eventual completion or failure of an asynchronous operation and its resulting value.
- A Promise is in one of these states:
  1. pending: initial state, neither fulfilled nor rejected.
  2. fulfilled: meaning that the operation was completed successfully.
  3. rejected: meaning that the operation failed.
- A pending promise can either be fulfilled with a value or rejected with a reason (error).
- Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.
- As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.
- A pending promise may transition into a fulfilled or rejected state.
- A fulfilled or rejected promise is settled, and must not transition into any other state.
- Once a promise is settled, it must have a value (which may be undefined). That value must not change.
- The arguments resolve and reject are callbacks provided by JavaScript itself. Our code is only inside the executor. When the executor obtains the result, it should call one of these callbacks:
  1. resolve(value) - State if fulfilled.
  2. reject(error) - State is rejected.



- **Note:** Even with resolve or reject, the value could still be undefined.
- E.g.



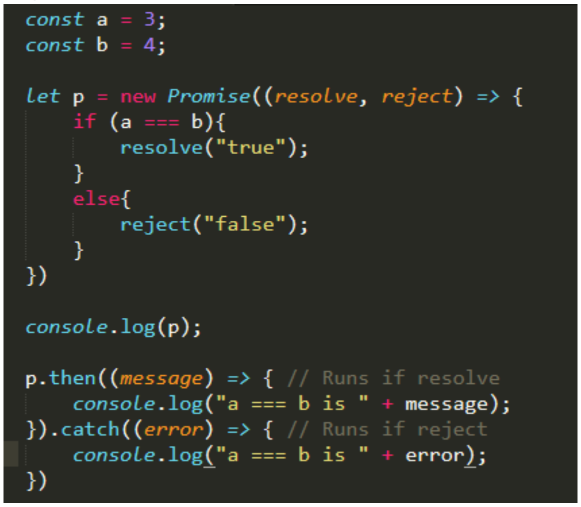
```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve("true");
    }
    else{
        reject("false");
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```



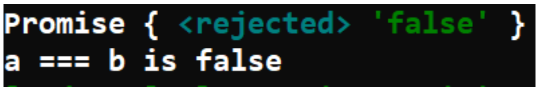
```
Promise { 'true' }
a === b is true
```

Note: resolve and reject are callback functions.
Here, since a === b, the promise runs resolve function, which will run the .then function.
Furthermore, we see that the promise is fulfilled.

- E.g.



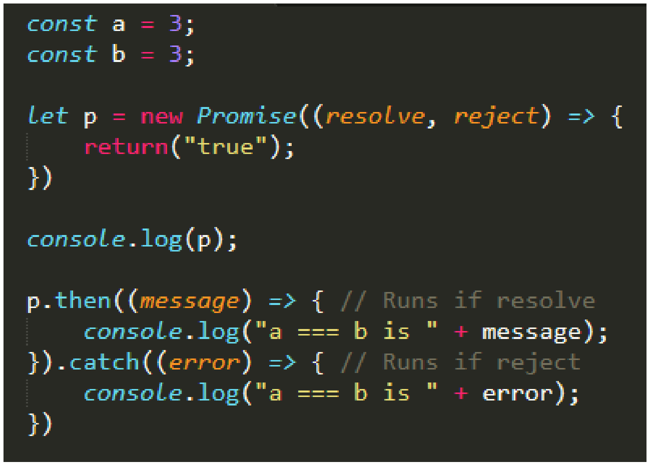
```
const a = 3;
const b = 4;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve("true");
    }
    else{
        reject("false");
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```



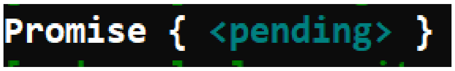
```
Promise { <rejected> 'false' }
a === b is false
```

Here, since a !== b, the promise runs resolve function, which will run the .catch function.
Furthermore, we see that the promise is rejected.

- E.g.



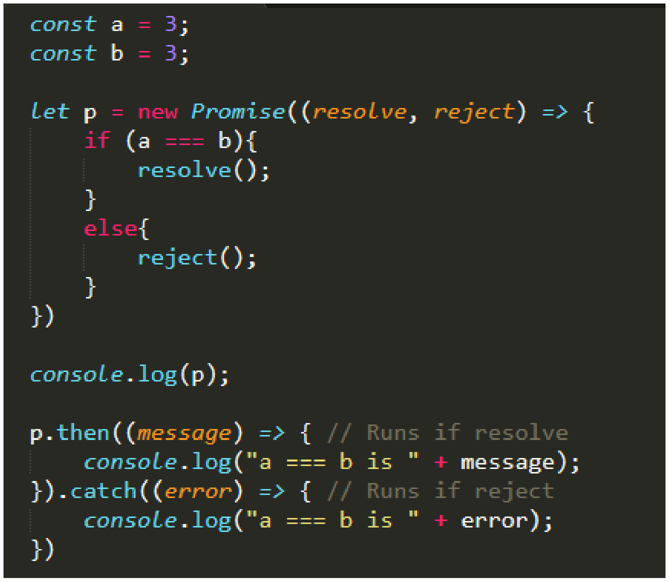
```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    return("true");
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```



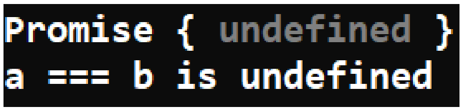
```
Promise { <pending> }
```

Here, since we don't use resolve or reject, the promise is pending.

- E.g.



```
const a = 3;
const b = 3;

let p = new Promise((resolve, reject) => {
    if (a === b){
        resolve();
    }
    else{
        reject();
    }
})

console.log(p);

p.then((message) => { // Runs if resolve
    console.log("a === b is " + message);
}).catch((error) => { // Runs if reject
    console.log("a === b is " + error);
})
```



```
Promise { undefined }
a === b is undefined
```

Here, because resolve doesn't give any value, we get undefined.

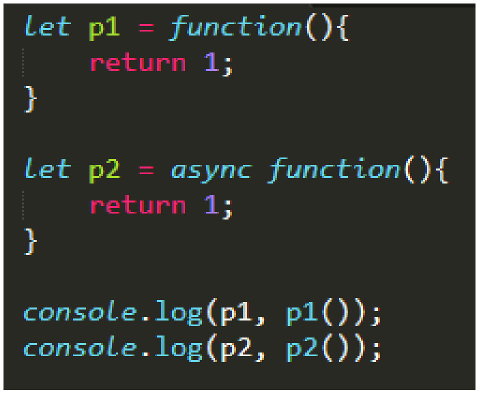
- **Async/Await:**
- The keyword **async** before a function makes the function return a promise.
- E.g.

```
async function foo() {
  return 1;
}
```

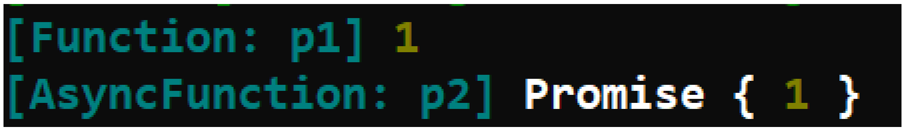
is equivalent to

```
function foo() {
  return Promise.resolve(1);
}
```

- E.g.



```
let p1 = function(){
    return 1;
}

let p2 = async function(){
    return 1;
}

console.log(p1, p1());
console.log(p2, p2());
```



```
[Function: p1] 1
[AsyncFunction: p2] Promise { 1 }
```

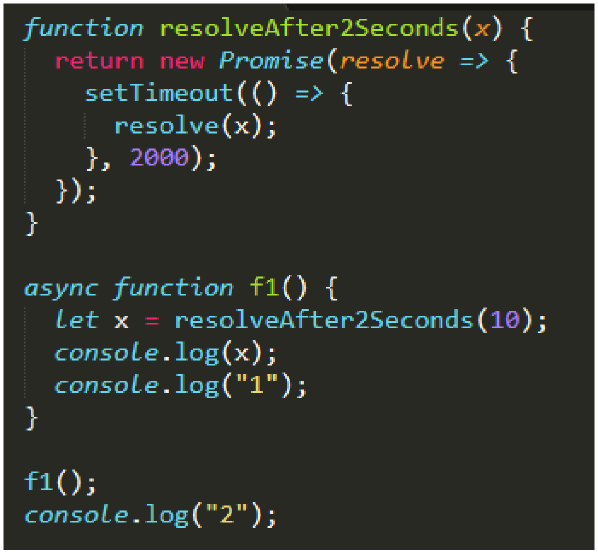
- The keyword **await** before a function makes the function wait for a promise.
- The await keyword can only be used inside an async function.
- The await expression causes an async function execution to pause until a Promise is settled (that is, fulfilled or rejected), and to resume execution of the async function after fulfillment. When resumed, the value of the await expression is that of the fulfilled Promise.
- If the Promise is rejected, the await expression throws the rejected value.
- If the value of the expression following the await operator is not a Promise, it's converted to a resolved Promise.
- An await splits the execution flow, allowing the caller of the async function to resume execution. After the await defers the continuation of the async function, execution of subsequent statements ensues. If this await is the last expression executed by its function, execution continues by returning to the function's caller a pending Promise for completion of the await's function and resuming execution of that caller.
- await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills/rejects, then return the resulting value.
- You can use await when calling any function that returns a Promise, including web API functions.
- The body of an async function can be thought of as being split by zero or more await expressions. Top-level code, up to and including the first await expression (if there is one), is run synchronously. In this way, an async function without an await expression will run synchronously. If there is an await expression inside the function body, however, the async function will always complete asynchronously.
- Code after each await expression can be thought of as existing in a .then callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.
- E.g.

```
async function foo() {
  await 1;
}
```

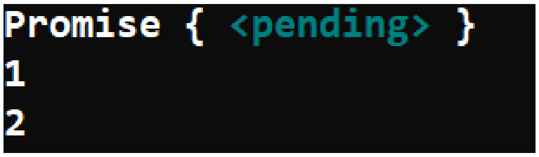
is equivalent to

```
function foo() {
  return Promise.resolve(1).then(() => undefined);
}
```

- E.g.



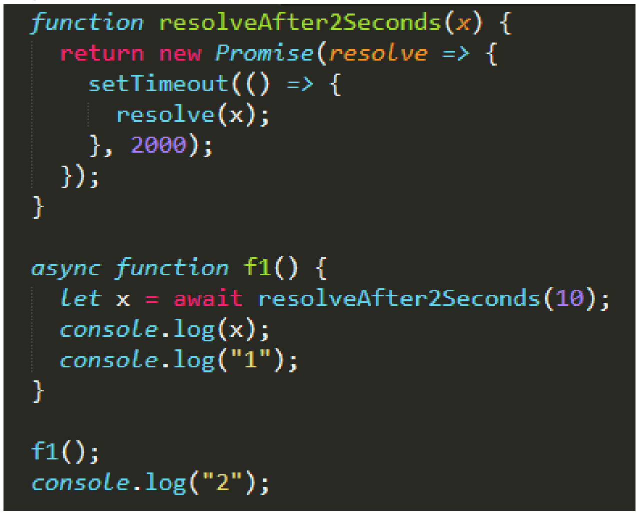
```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  let x = resolveAfter2Seconds(10);
  console.log(x);
  console.log("1");
}

f1();
console.log("2");
```



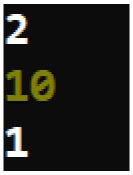
```
Promise { <pending> }
1
2
```

Notice that x is still pending (because of the setTimeout function) and that 1 is printed before 2.

- E.g.



```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  let x = await resolveAfter2Seconds(10);
  console.log(x);
  console.log("1");
}

f1();
console.log("2");
```



```
2
10
1
```

Now, when we use await, we see that we wait for resolveAfter2Seconds to return a settled value, either fulfilled or rejected, before continuing. Furthermore, notice that 2 is printed before 10 and 1. This is because await causes the code to run asynchronously. So while it's waiting for the result of resolveAfter2Seconds, the other parts of the code, not in f1, will run. Hence, 2 is printed first.

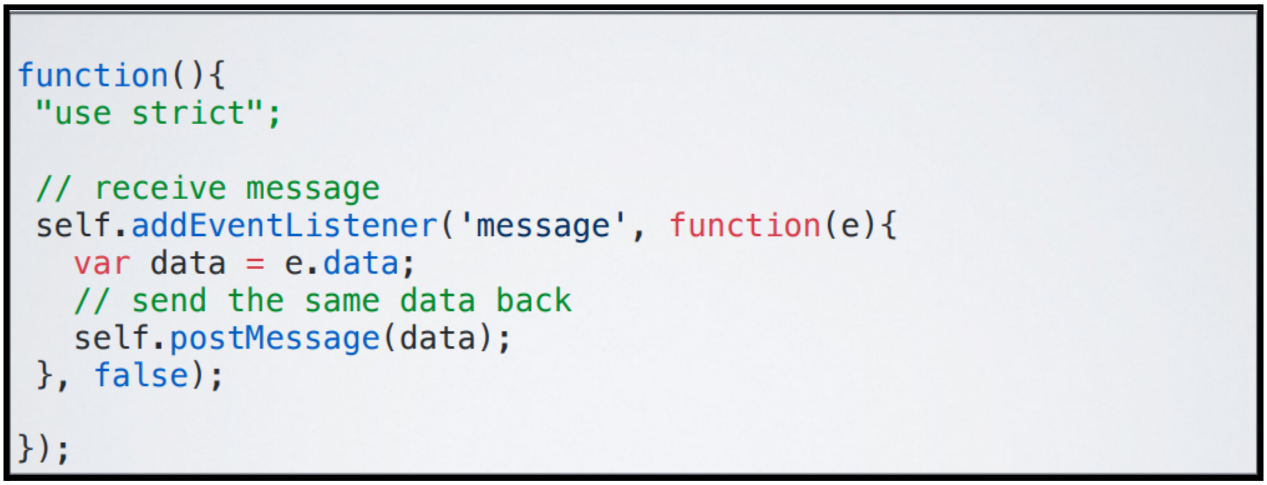
Web Workers:

- **Web workers** makes it possible to run a script operation in a background thread separate from the main execution thread of a web application. The advantage of this is that laborious processing can be performed in a separate thread, allowing the main (usually the UI) thread to run without being blocked/slowed down.
- Web workers create threads in Javascript (frontend and backend).
- These threads can run in parallel (separate event loop).

- What a web worker can/cannot do on the frontend

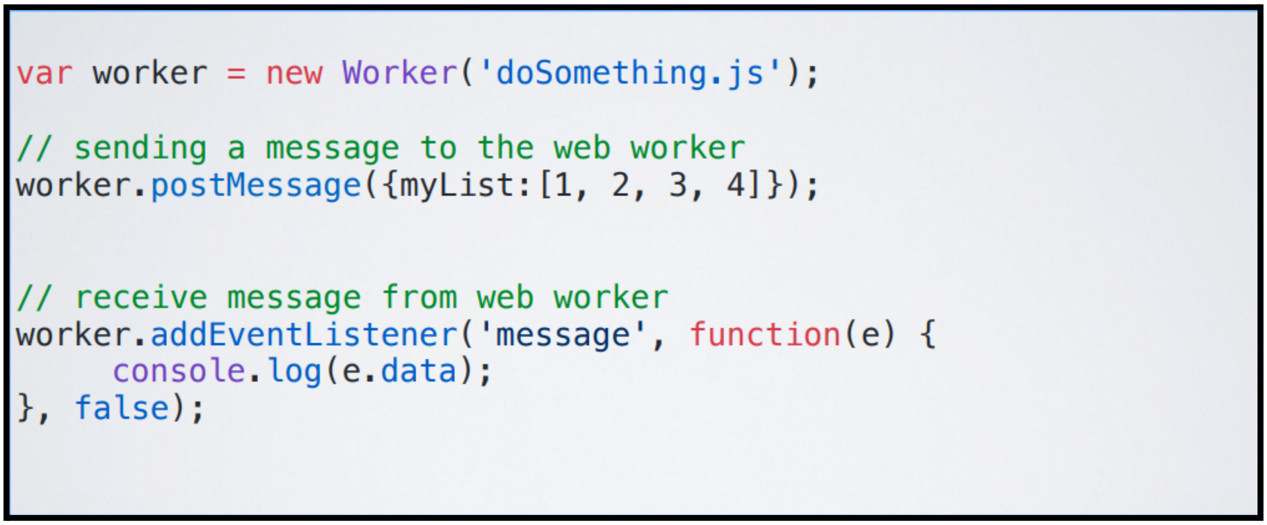
Can do	**Cannot do**
XMLHttpRequest	window
indexedDB	document (not thread safe)
location (read only)	

- Create a web worker



```
function(){
 "use strict";

 // receive message
 self.addEventListener('message', function(e){
   var data = e.data;
   // send the same data back
   self.postMessage(data);
 }, false);

});
```

- Instantiate a web worker



```
var worker = new Worker('doSomething.js');

// sending a message to the web worker
worker.postMessage({myList:[1, 2, 3, 4]});


// receive message from web worker
worker.addEventListener('message', function(e) {
     console.log(e.data);
}, false);
```