Lesson - 24

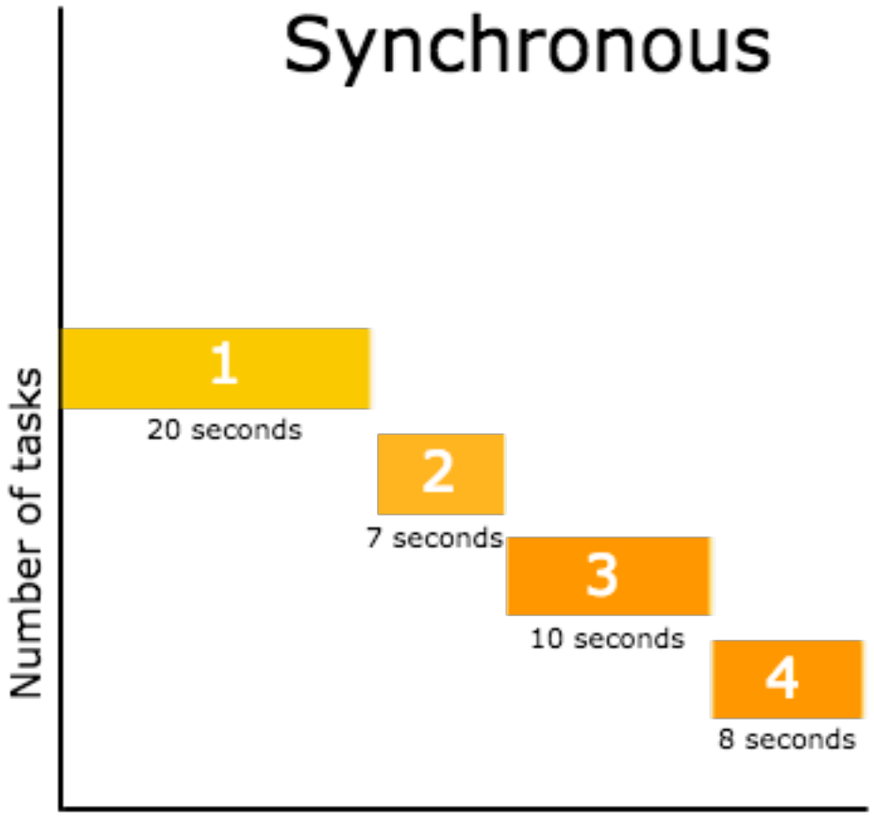
Asynchronous Javascript

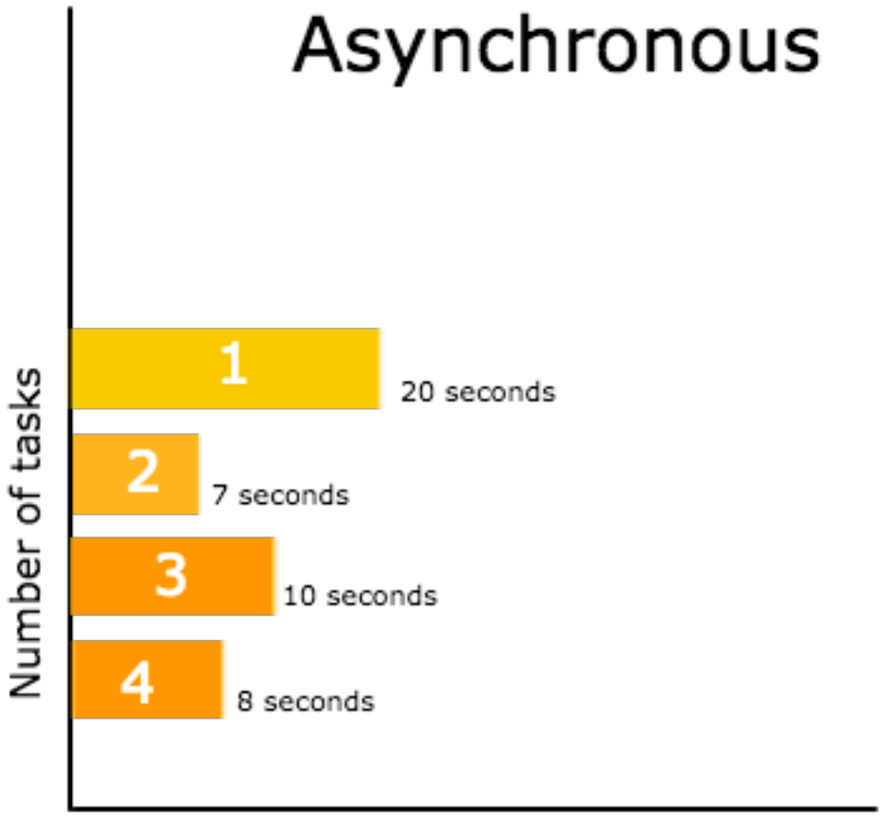
SkillUp by Dobrea Vladislav

Lesson Plan

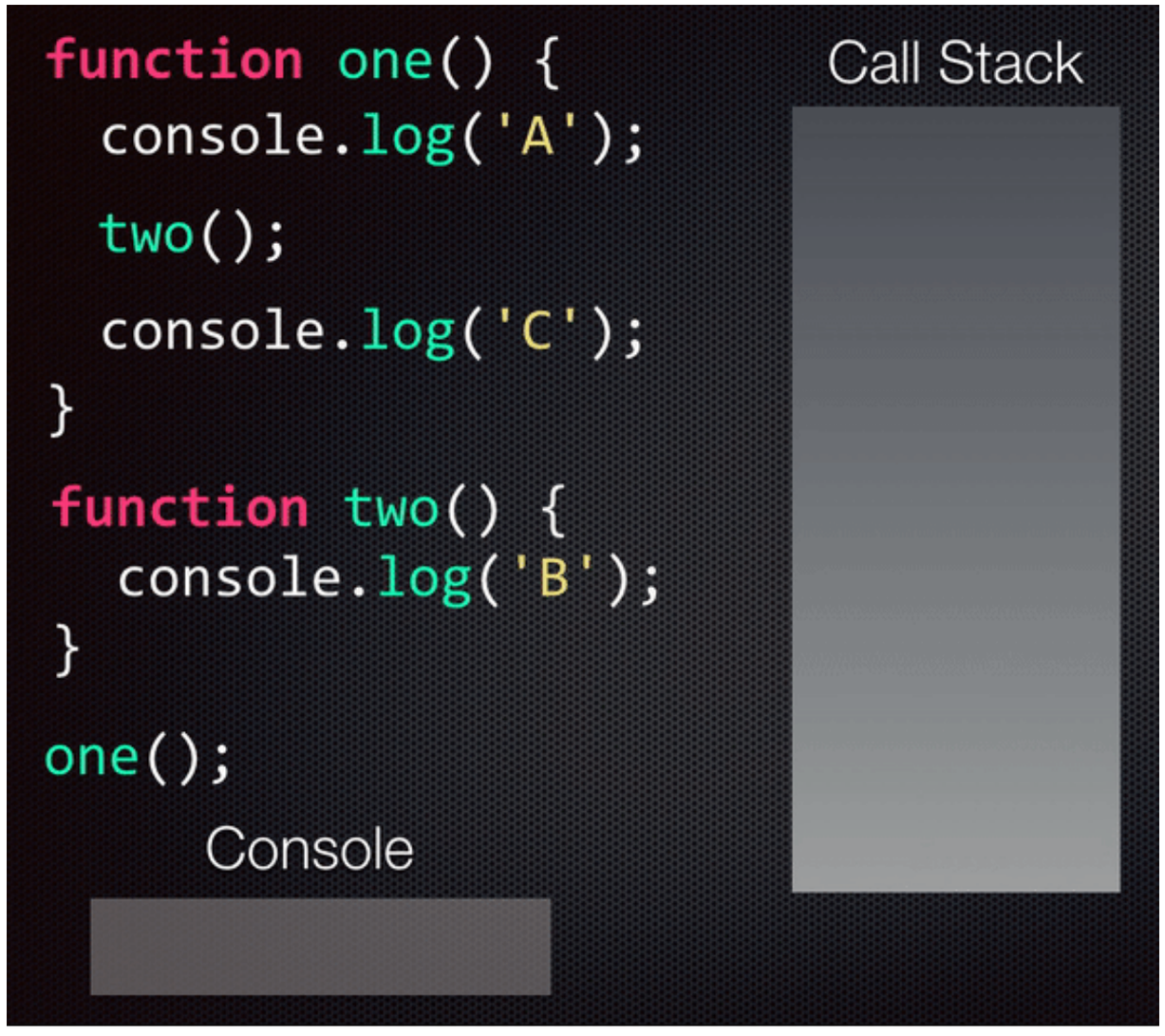
- Discuss HW
- What is Call Stack?
- Introduction to asynchronous javascript
- Event Loop
- Callback’s
- Promises

Sync vs Async





How does sync work?



To understand how the above code executes inside the JavaScript engine, we have to understand the concept of the execution context and the call stack (also known as execution stack).

What is Execution Context?

An Execution Context is an abstract concept of an environment where the JavaScript code is evaluated and executed. Whenever any code is run in JavaScript, it's run inside an execution context.

The function code executes inside the function execution context, and the global code executes inside the global execution context. Each function has its own execution context.

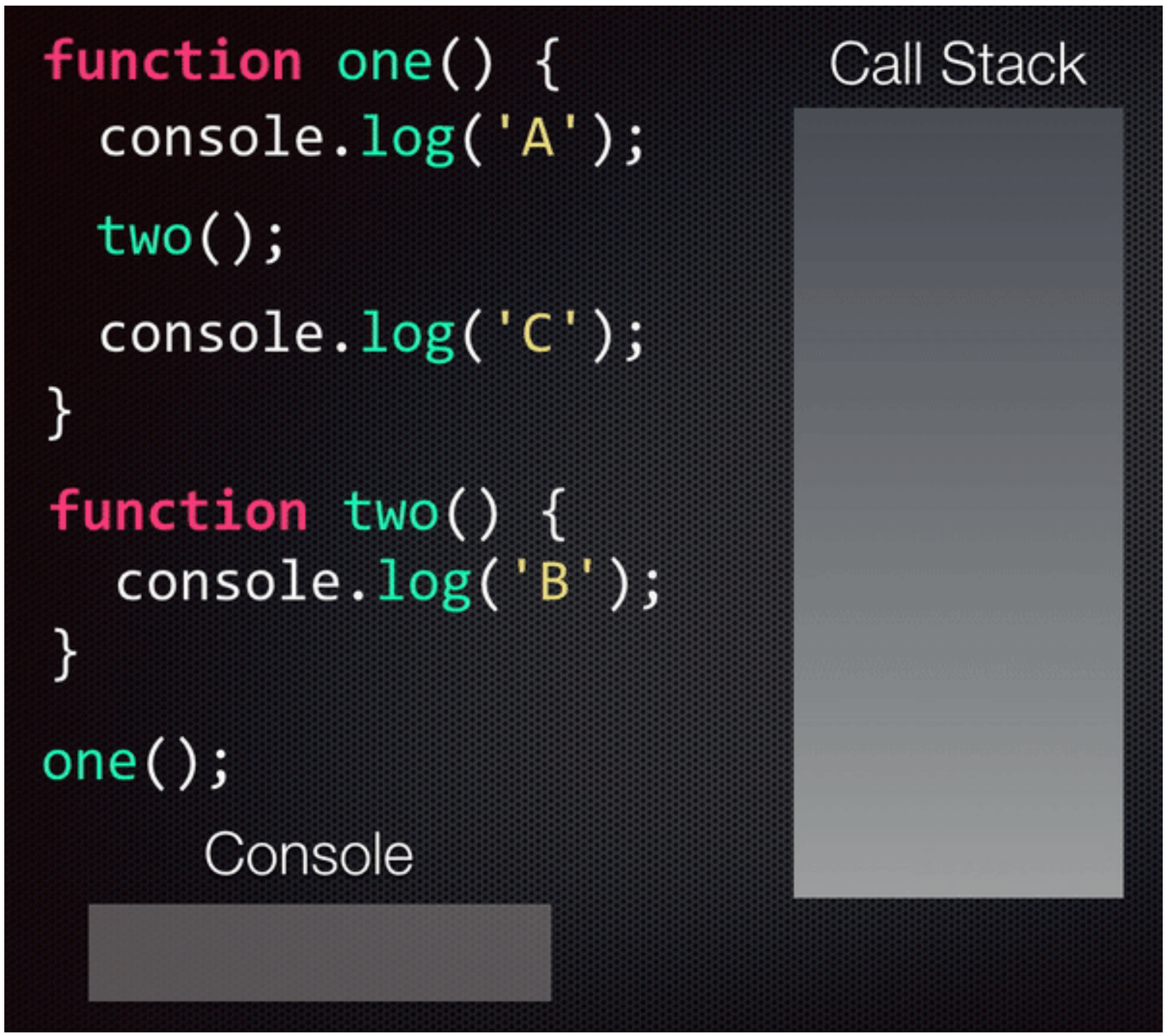
What is Call Stack?

The call stack as its name implies is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.

JavaScript has a single call stack because it's a single-threaded programming language.

The call stack has a LIFO structure which means that the items can be added or removed from the top of the stack only.

Execution Context and Call Stack in action



How does async work?

Now that we have a basic idea about the call stack, and how the synchronous JavaScript works, let's get back to the asynchronous JavaScript.

What is Blocking?

Let's suppose we are doing an image processing or a network request in a synchronous way. For example:

```
const processImage = (image) => {
  /**
  * doing some operations on image
  **/
  console.log('Image processed');
}
const networkRequest = (url) => {
  /**
  * requesting network resource
  **/
  return someData;
}
const greeting = () => {
  console.log('Hello World');
}
processImage(logo.jpg);
networkRequest('www.somerandomurl.com');
greeting();
```

The problem

Doing image processing and network request takes time. So when `processImage()` function is called, it's going to take some time depending on the size of the image.

When the `processImage()` function completes, it's removed from the stack. After that the `networkRequest()` function is called and pushed to the stack. Again it's also going to take some time to finish execution.

At last when the `networkRequest()` function completes, `greeting()` function is called and since it contains only a `console.log` statement and `console.log` statements are generally fast, so the `greeting()` function is immediately executed and returned.

So you see, we have to wait until the function (such as `processImage()` or `networkRequest()`) has finished. This means these functions are blocking the call stack or main thread. So we can't perform any other operation while the above code is executing which is not ideal.

The solution!

The simplest solution is asynchronous callbacks. We use asynchronous callbacks to make our code non-blocking.

For example:

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
};
console.log('Hello World');
networkRequest();
```

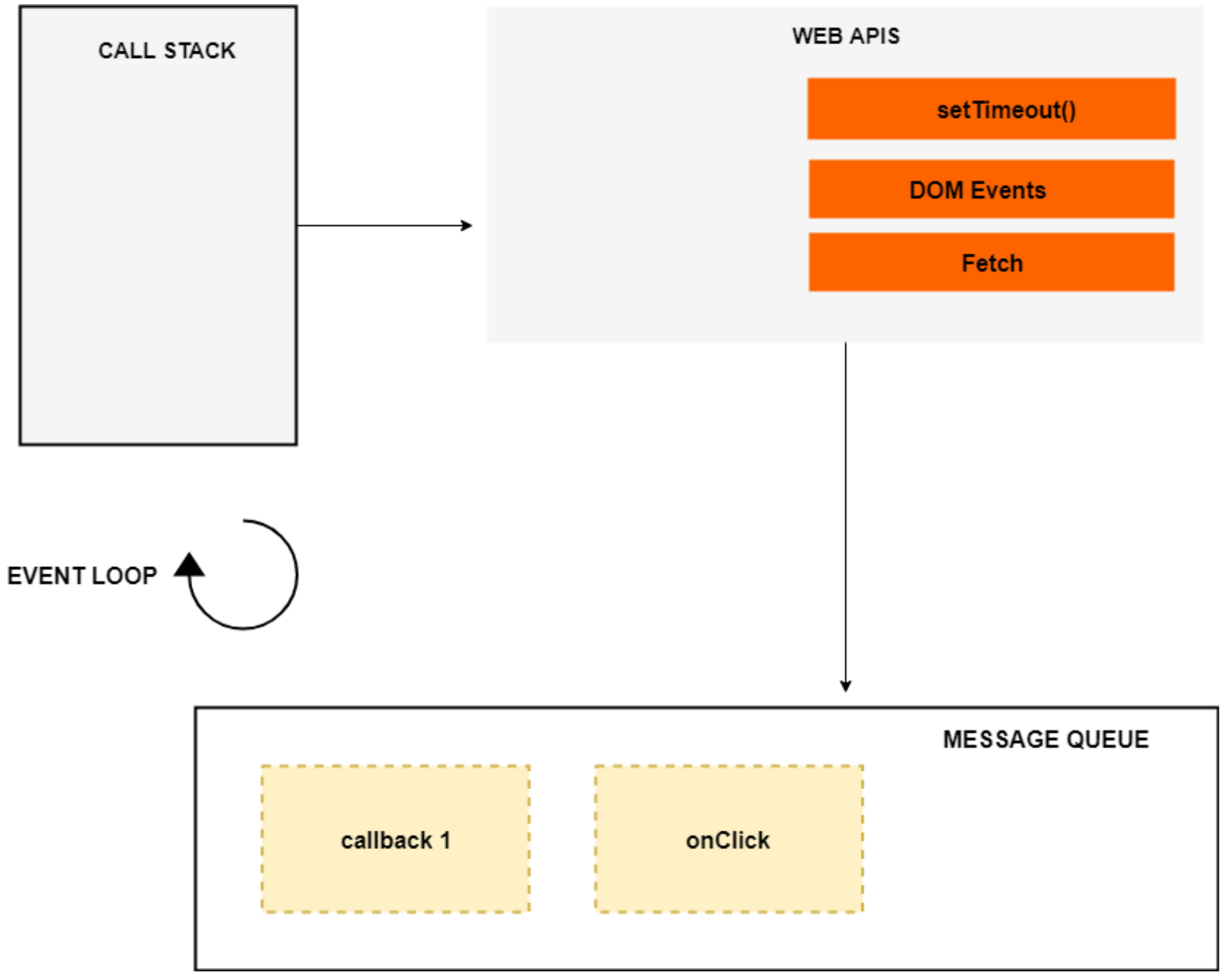
Here I have used `setTimeout` method to simulate the network request.

What is a callback?

Simply put: A callback is a function that is to be executed after another function has finished executing — hence the name ‘call back’.

More complexly put: In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called higher-order functions. Any function that is passed as an argument is called a callback function.

Event loop and message queue



Example

```
const networkRequest = () => {
  setTimeout(() => {
    console.log('Async Code');
  }, 2000);
};
console.log('Hello World');
networkRequest();
console.log('The End');
```

What happens?

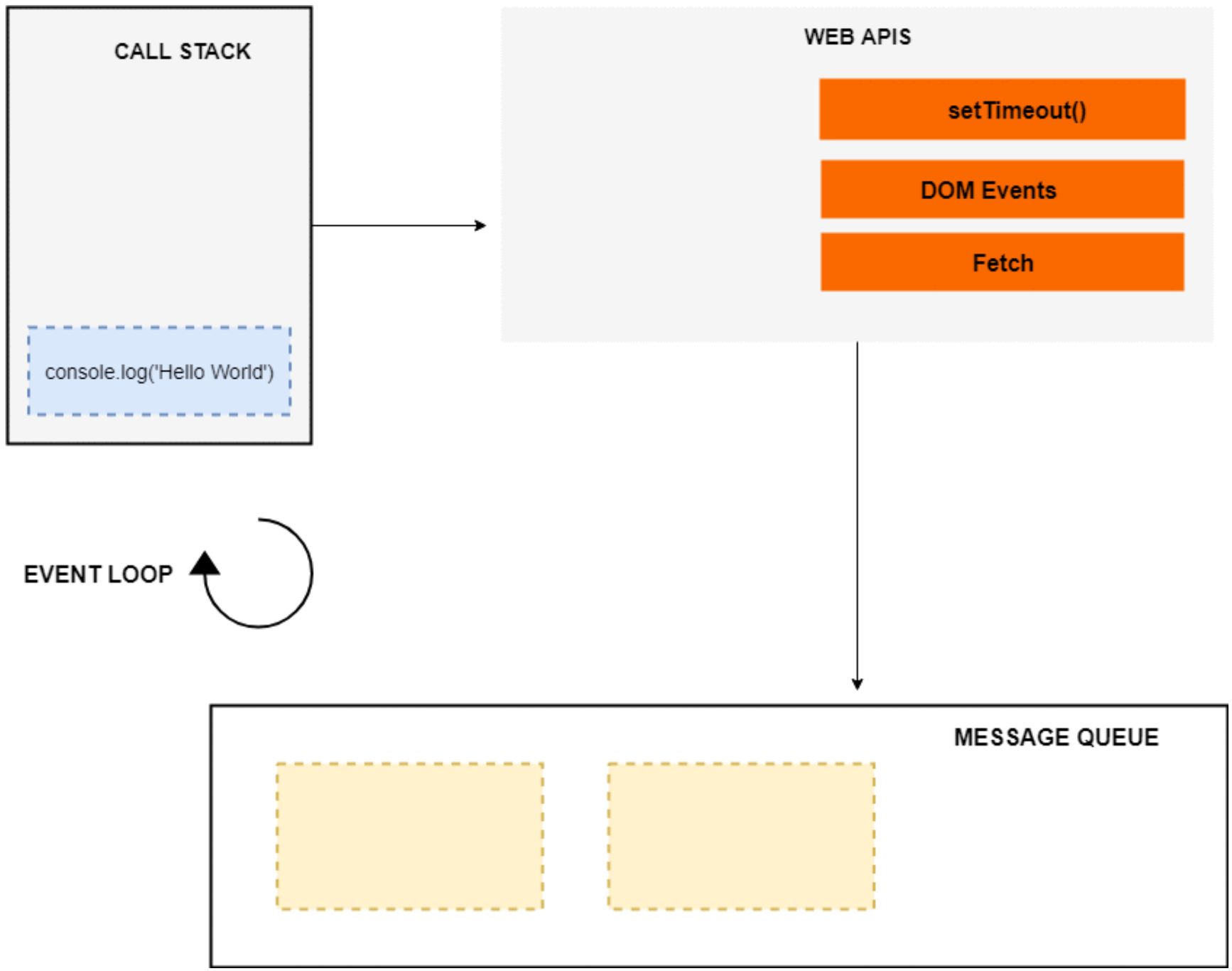
When the above code loads in the browser, the `console.log('Hello World')` is pushed to the stack and popped off the stack after it's finished. Next, a call to `networkRequest()` is encountered, so it's pushed to the top of the stack.

Next `setTimeout()` function is called, so it's pushed to the top of the stack. The `setTimeout()` has two arguments: 1) callback and 2) time in milliseconds (ms).

The `setTimeout()` method starts a timer of `2s` in the web APIs environment. At this point, the `setTimeout()` has finished and it's popped off from the stack. After it, `console.log('The End')` is pushed to the stack, executed and removed from the stack after its completion.

Meanwhile, the timer has expired, now the callback is pushed to the **message queue**. But the callback is not immediately executed, and that's where the event loop kicks in.

Illustration



Event Loop

The job of the Event loop is to look into the call stack and determine if the call stack is empty or not. If the call stack is empty, it looks into the message queue to see if there's any pending callback waiting to be executed.

In this case, the message queue contains one callback, and the call stack is empty at this point. So the Event loop pushes the callback to the top of the stack.

After that the `console.log('Async Code')` is pushed to the top of the stack, executed and popped off from the stack. At this point, the callback has finished so it's removed from the stack and the program finally finishes.

Callback's

Many functions are provided by JavaScript host environments that allow you to schedule *asynchronous* actions. In other words, actions that we initiate now, but they finish later.

For instance, one such function is the `setTimeout` function.

There are other real-world examples of asynchronous actions, e.g. loading scripts and modules (we'll cover them in later chapters).

Take a look at the function `loadScript(src)`, that loads a script with the given `src`:

```
function loadScript(src) {
  // creates a <script> tag and append it to the page
  // this causes the script with given src to start loading and run when comp
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

It appends to the document the new, dynamically created, tag `<script src="…">` with given `src`. The browser automatically starts loading it and executes when complete.

We can use this function like this:

```
// load and execute the script at the given path
loadScript('/my/script.js');
```

Callback’s

The script is executed “asynchronously”, as it starts loading now, but runs later, when the function has already finished.

If there’s any code below `loadScript(…)`, it doesn’t wait until the script loading finishes.

```
loadScript('/my/script.js');
// the code below loadScript
// doesn't wait for the script loading to finish
// ...
```

Let’s say we need to use the new script as soon as it loads. It declares new functions, and we want to run them.

But if we do that immediately after the `loadScript(…)` call, that wouldn’t work:

```
loadScript('/my/script.js'); // the script has "function newFunction() {…}"

newFunction(); // no such function!
```

Callback's

Naturally, the browser probably didn't have time to load the script. As of now, the `loadScript` function doesn't provide a way to track the load completion. The script loads and eventually runs, that's all. But we'd like to know when it happens, to use new functions and variables from that script.

Let's add a `callback` function as a second argument to `loadScript` that should execute when the script loads:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function() {
  // the callback runs after the script is loaded
  newFunction(); // so now it works
  ...
});
```

Callback's

That's the idea: the second argument is a function (usually anonymous) that runs when the action is completed.

Here's a runnable example with a real script:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js'
  alert(`Cool, the script ${script.src} is loaded`);
  alert( _ ); // function declared in the loaded script
});
```

That's called a "callback-based" style of asynchronous programming. A function that does something asynchronously should provide a `callback` argument where we put the function to run after it's complete.

Here we did it in `loadScript`, but of course it's a general approach.

Callback in Callback

How can we load two scripts sequentially: the first one, and then the second one after it?

The natural solution would be to put the second `loadScript` call inside the callback, like this:

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

After the outer `loadScript` is complete, the callback initiates the inner one.

What if we want one more script...?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });

  })

});
```

Handling errors

In the above examples we didn't consider errors. What if the script loading fails? Our callback should be able to react on that.

Here's an improved version of `loadScript` that tracks loading errors:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

It calls `callback(null, script)` for successful load and `callback(error)` otherwise.

The usage:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

Once again, the recipe that we used for `loadScript` is actually quite common. It's called the "error-first callback" style.

The convention is:

1. The first argument of the `callback` is reserved for an error if it occurs. Then `callback(err)` is called.
2. The second argument (and the next ones if needed) are for the successful result. Then `callback(null, result1, result2…)` is called.

So the single `callback` function is used both for reporting errors and passing back results.

Pyramid of doom

From the first look, it's a viable way of asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.

But for multiple asynchronous actions that follow one after another we'll have code like this:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    })
  }
});
```

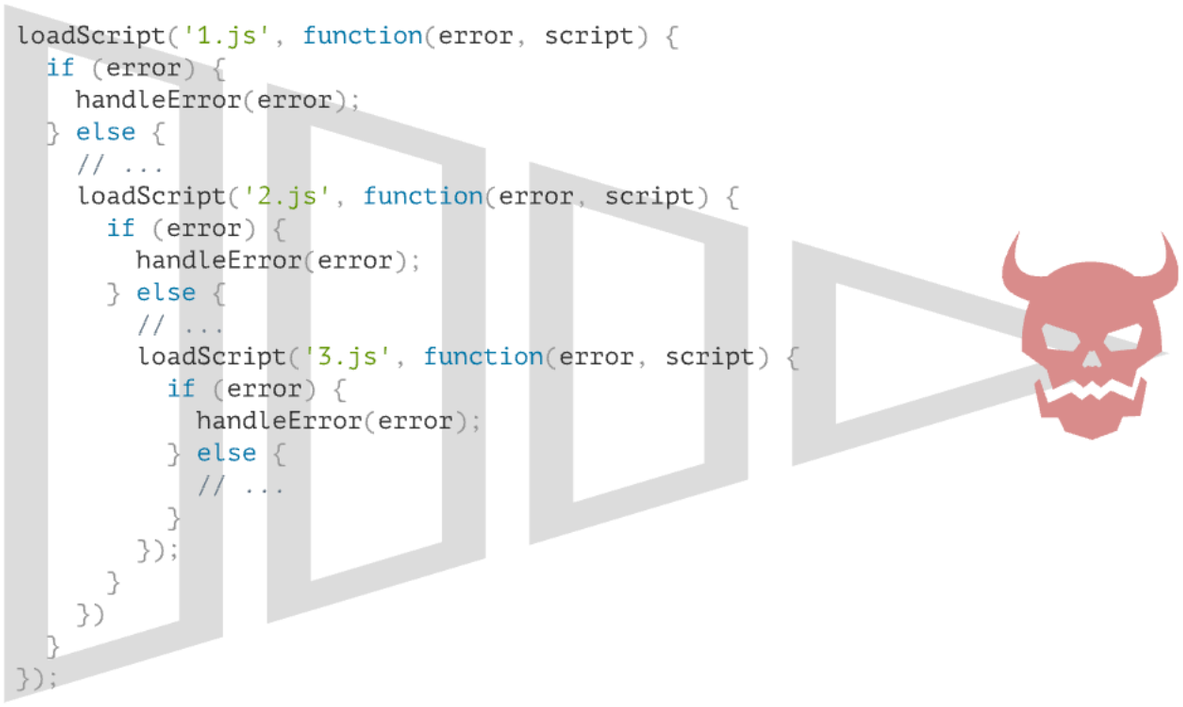
Pyramid of doom

In the code above:

1. We load `1.js`, then if there's no error.
2. We load `2.js`, then if there's no error.
3. We load `3.js`, then if there's no error – do something else `(*)`.

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of `...` that may include more loops, conditional statements and so on.

That's sometimes called "callback hell" or "pyramid of doom."



```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

The "pyramid" of nested calls grows to the right with every asynchronous action. Soon it spirals out of control.

So this way of coding isn't very good.

Pyramid of doom

We can try to alleviate the problem by making every action a standalone function, like this:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded (*)
  }
};
```

Callback's conclusion

See? It does the same, and there's no deep nesting now because we made every action a separate top-level function.

It works, but the code looks like a torn apart spreadsheet. It's difficult to read, and you probably noticed that one needs to eye-jump between pieces while reading it. That's inconvenient, especially if the reader is not familiar with the code and doesn't know where to eye-jump.

Also, the functions named `step*` are all of single use, they are created only to avoid the "pyramid of doom." No one is going to reuse them outside of the action chain. So there's a bit of namespace cluttering here.

We'd like to have something better.

Luckily, there are other ways to avoid such pyramids. One of the best ways is to use "promises," described in the next chapter.

Promises

Imagine that you're a top singer, and fans ask day and night for your upcoming single.

To get some relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can't publish the song, they will still be notified.

Everyone is happy: you, because the people don't crowd you anymore, and fans, because they won't miss the single.

This is a real-life analogy for things we often have in programming:

1. A "producing code" that does something and takes time. For instance, some code that loads the data over a network. That's a "singer".
2. A "consuming code" that wants the result of the "producing code" once it's ready. Many functions may need that result. These are the "fans".
3. A *promise* is a special JavaScript object that links the "producing code" and the "consuming code" together. In terms of our analogy: this is the "subscription list". The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.

Promises

The analogy isn't terribly accurate, because JavaScript promises are more complex than a simple subscription list: they have additional features and limitations. But it's fine to begin with.

The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

The function passed to `new Promise` is called the *executor*. When `new Promise` is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the "singer".

Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself. Our code is only inside the executor.

Promises

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

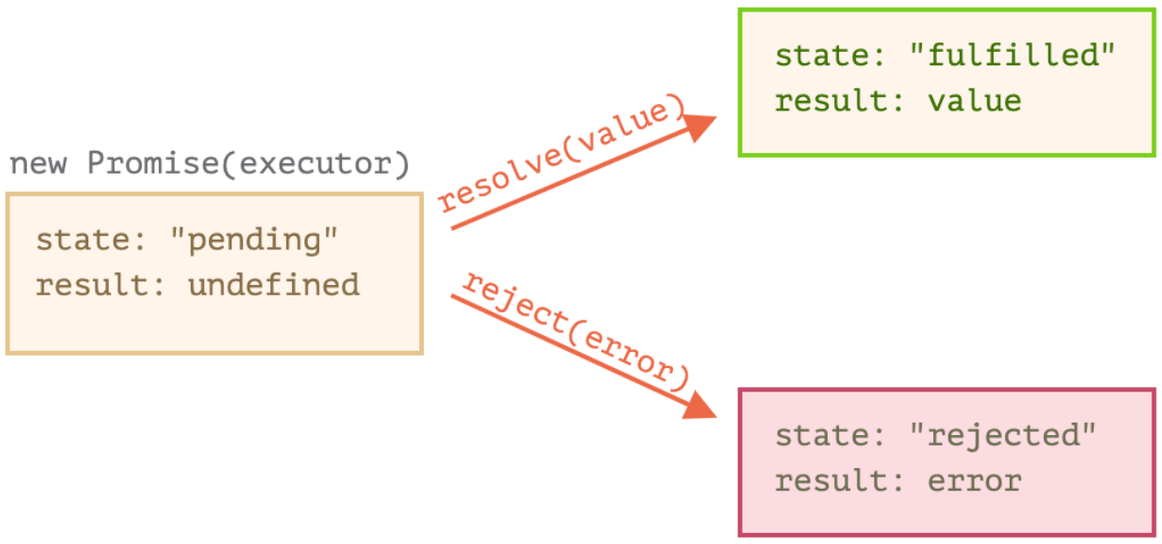
- `resolve(value)` — if the job finished successfully, with result `value`.
- `reject(error)` — if an error occurred, `error` is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt it calls `resolve` if it was successful or `reject` if there was an error.

The `promise` object returned by the `new Promise` constructor has these internal properties:

- `state` — initially `"pending"`, then changes to either `"fulfilled"` when `resolve` is called or `"rejected"` when `reject` is called.
- `result` — initially `undefined`, then changes to `value` when `resolve(value)` called or `error` when `reject(error)` is called.

So the executor eventually moves `promise` to one of these states:



Promises

Here's an example of a promise constructor and a simple executor function with "producing code" that takes time (via `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

We can see two things by running the code above:

1. The executor is called automatically and immediately (by `new Promise`).
2. The executor receives two arguments: `resolve` and `reject`. These functions are pre-defined by the JavaScript engine, so we don't need to create them. We should only call one of them when ready.

   After one second of "processing" the executor calls `resolve("done")` to produce the result. This changes the state of the `promise` object:

That was an example of a successful job completion, a "fulfilled promise".

Promises

Here's an example of a promise constructor and a simple executor function with "producing code" that takes time (via `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```

We can see two things by running the code above:

1. The executor is called automatically and immediately (by `new Promise`).
2. The executor receives two arguments: `resolve` and `reject`. These functions are pre-defined by the JavaScript engine, so we don't need to create them. We should only call one of them when ready.

   After one second of "processing" the executor calls `resolve("done")` to produce the result. This changes the state of the `promise` object:

That was an example of a successful job completion, a "fulfilled promise".

Consumers: then, catch, finally

The most important, fundamental one is `.then`.

The syntax is:

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

The first argument of `.then` is a function that runs when the promise is resolved, and receives the result.

The second argument of `.then` is a function that runs when the promise is rejected, and receives the error.

For instance, here's a reaction to a successfully resolved promise:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

The first function was executed.

Consumers: then

And in the case of a rejection, the second one:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

If we're interested only in successful completions, then we can provide only one function argument to `.then`:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

Consumers: catch

catch

If we're interested only in errors, then we can use `null` as the first argument: `.then(null, errorHandlingFunction)`. Or we can use `.catch(errorHandlingFunction)`, which is exactly the same:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

The call `.catch(f)` is a complete analog of `.then(null, f)`, it's just a shorthand.

Consumers: finally

Just like there's a `finally` clause in a regular `try {...} catch {...}`, there's `finally` in promises.

The call `.finally(f)` is similar to `.then(f, f)` in the sense that `f` always runs when the promise is settled: be it resolve or reject.

`finally` is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed anymore, no matter what the outcome is.

Like this:

```
new Promise((resolve, reject) => {
  /* do something that takes time, and then call resolve/reject */
})
  // runs when the promise is settled, doesn't matter successfully or not
  .finally(() => stop loading indicator)
  .then(result => show result, err => show error)
```

Consumers: finally

It's not exactly an alias of `then(f,f)` though. There are several important differences:

1. A `finally` handler has no arguments. In `finally` we don't know whether the promise is successful or not. That's all right, as our task is usually to perform "general" finalizing procedures.
2. A `finally` handler passes through results and errors to the next handler.

   For instance, here the result is passed through `finally` to `then`:

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then handles the result
```

   And here there's an error in the promise, passed through `finally` to `catch`:

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err));  // <-- .catch handles the error object
```

   That's very convenient, because `finally` is not meant to process a promise result. So it passes it through.

   We'll talk more about promise chaining and result-passing between handlers in the next chapter.

3. Last, but not least, `.finally(f)` is a more convenient syntax than `.then(f, f)`: no need to duplicate the function `f`.

LoadScript rewrited

The new function `loadScript` will not require a callback. Instead, it will create and return a Promise object that resolves when the loading is complete. The outer code can add handlers (subscribing functions) to it using `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Script load error for ${src}`));

    document.head.append(script);
  });
}
```

Usage:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Another handler...'));
```

We can immediately see a few benefits over the callback-based pattern:

Comparison

Promises	Callbacks
Promises allow us to do things in the natural order. First, we run `loadScript(script)`, and `.then` we write what to do with the result.	We must have a `callback` function at our disposal when calling `loadScript(script, callback)`. In other words, we must know what to do with the result *before* `loadScript` is called.
We can call `.then` on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list". More about this in the next chapter: Promises chaining.	There can be only one callback.

So promises give us better code flow and flexibility. But there's more. We'll see that in the next chapters.

Learning Resources

1. **Callbacks**
    1. https://javascript.info/callbacks
2. **Promises**
    1. https://javascript.info/promise-basics
    2. https://javascript.info/promise-chaining
    3. **VIDEO:** https://www.youtube.com/watch?v=2d7s3spWAzo
    4. **VIDEO:** https://www.youtube.com/watch?v=QO4NXhWo_NM

HomeWork

Go through learning resources, and watch videos to better understanding of promises.