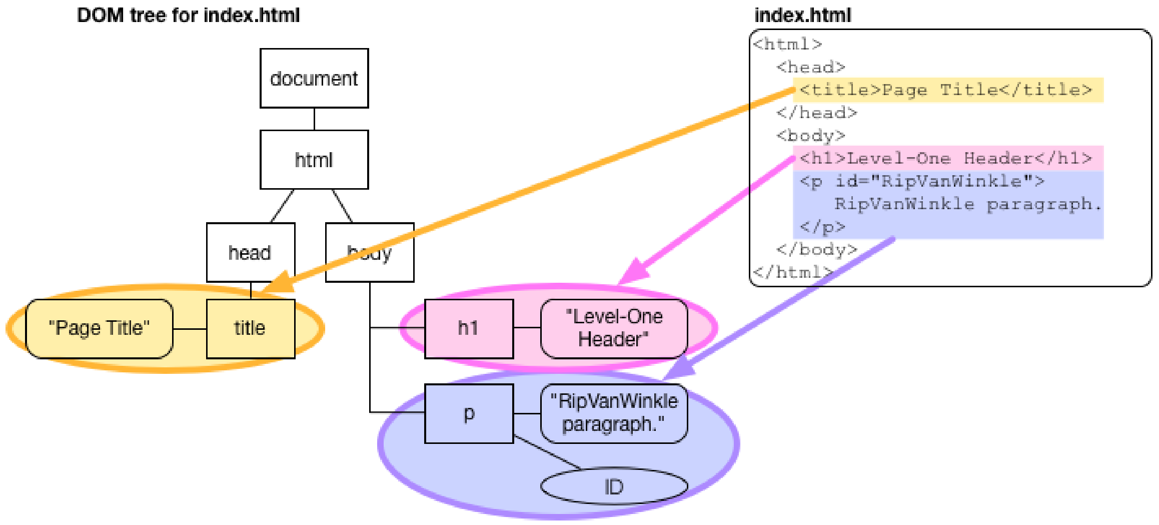
# HYPERMEDIA
# WEB AND MULTIMEDIA

Js advanced

# DOM manipulation

The DOM is created from the HTML and it contains information about each element and its attributes.

With JavaScript it is possible to get access to these nodes and manipulate them.



```
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1>Level-One Header</h1>
    <p id="RipVanWinkle">
      RipVanWinkle paragraph.
    </p>
  </body>
</html>
```

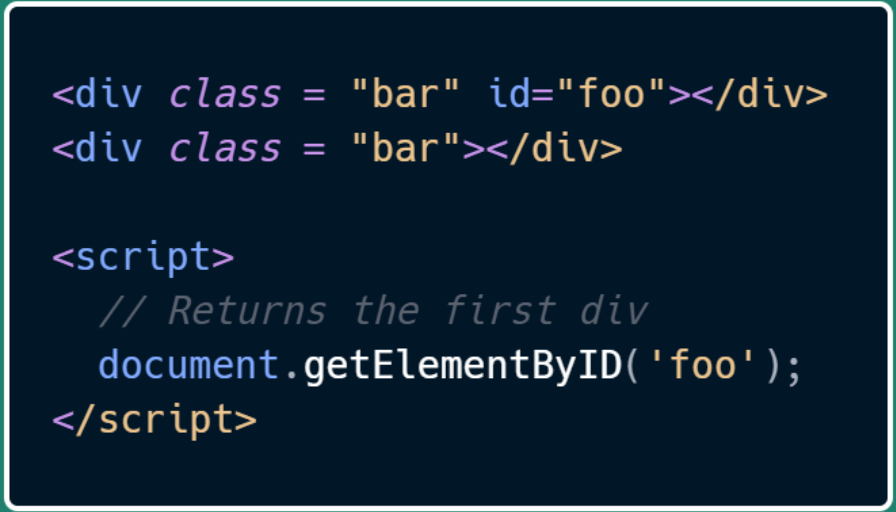
## DOM manipulation

JS gives access to a special object that represents the DOM in memory: the document object.

If you want to get access to any HTML element, you always start with accessing the document object.

# DOM manipulation

**document.getElementByID(x) retrieves a node in the DOM with the given ID**



```
<div class = "bar" id="foo"></div>
<div class = "bar"></div>

<script>
  // Returns the first div
  document.getElementByID('foo');
</script>
```

# DOM manipulation

**document.getElementByClass(x) retrieves all the nodes in the DOM with the given class**

```
<div class = "bar" id="foo"></div>
<div class = "bar"></div>
<div class = "anotherClass"></div>

<script>
  // Returns all the div of class bar
  // Returns the first two div
  document.getElementByClass('bar');
</script>
```

## DOM manipulation

**There is an easier way to do the same thing:**

- **document.querySelector( x )**
- **document.querySelectorAll( x )**

**The parameter passed to these methods follows the same rules as CSS selector:**

- **Name of the element (a, div, p…)**
- **Class of the element (.*name*)**
- **ID of the element (#*name*)**

```
<div id="foo"></div>
<div class="bar"></div>

<script>
  // Returns the first div
  document.querySelector('#foo');

  // Returns the second div
  document.querySelector('.bar');

  // Returns the first div
  document.querySelector('div');

  // Returns both
  document.querySelectorAll('div');
</script>
```

## DOM manipulation

Once an element is retrieved we can:

- change the text in the element (.innerText)
- change the html structure in the element (.innerHTML)
- add child elements (.appendChild(element))
- remove a child element(.removeChild(element))
- add events handler (.addEventListener(event, handler))

List of properties: Element, HTMLElement

## Window Interaction

This manipulation can be brought to a higher level by using another special object: the window object.

This object refers to the BOM (Browser Object Model).

Note that the document object that was used before is contained inside the window object.

## Window interaction

Using the window object we can:

- get information about the current URL (window.location)
- get the size of the browser viewport (window.innerHeight or innerWidth)
- open a new window/tab (window.open())
- Show an alert (window.alert())

Other parameters/method are available here.

CODEPEN

DOM manipulation

## Classes

An object is a special kind of “container” that can hold both values and functions (methods).

Downside of using generic object:
in case we want multiple objects with the same structure, we need to manually define what are the key:value pairs every time.

## Classes

Classes are templates for creating objects.

They define which variables and methods are available for all the objects that belongs to that class.

# Class declaration

**To create a class we use the *class* keyword.**

**The constructor defines what values are required for the object to be built.**

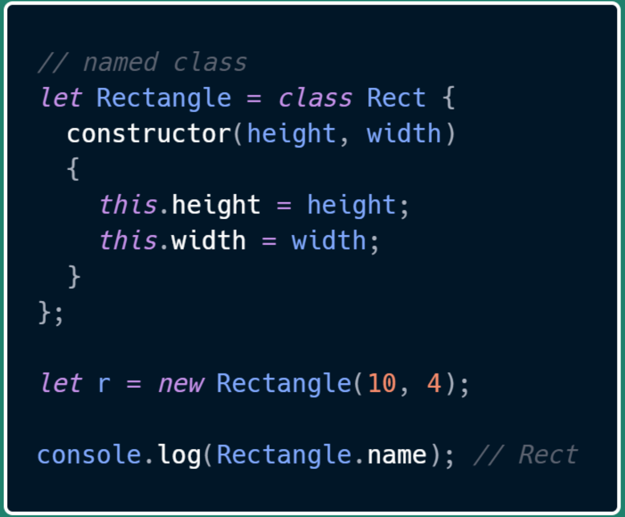
**When we want to create an object of the class we use the *new* keyword, followed by the class and the parameters.**

```
// Class declaration
class Rectangle
{
  constructor(height, width)
  {
    // Definition of properties
    this.height = height;
    this.width = width;
  }
}

// Declaration of object of class Rectangle
const r = new Rectangle(10, 4);

/*
r is now an object of class Rectangle
r =
    {
        height: 10,
        width: 4
    }
*/
```

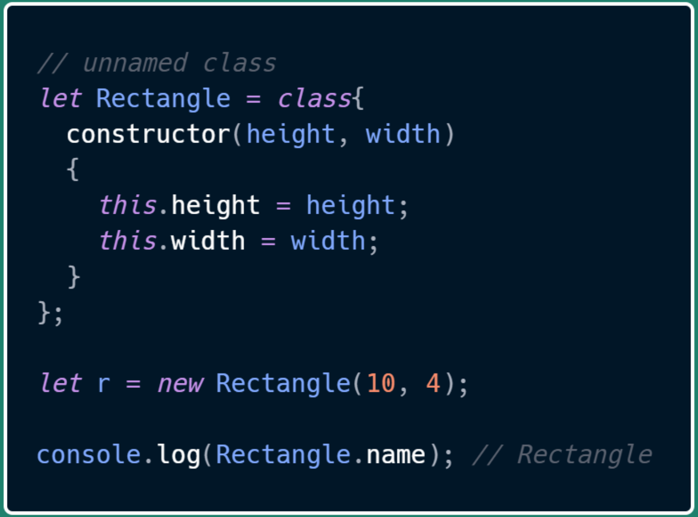
# Class expression

**A class can also be defined by using class expressions.**
**In this case we assign the definition to a variable and use that to create objects.**

**Unlike variables and function, classes declaration cannot be hoisted.**
**So, always define the class before you use it.**



```javascript
// named class
let Rectangle = class Rect {
  constructor(height, width)
  {
    this.height = height;
    this.width = width;
  }
};

let r = new Rectangle(10, 4);

console.log(Rectangle.name); // Rect
```



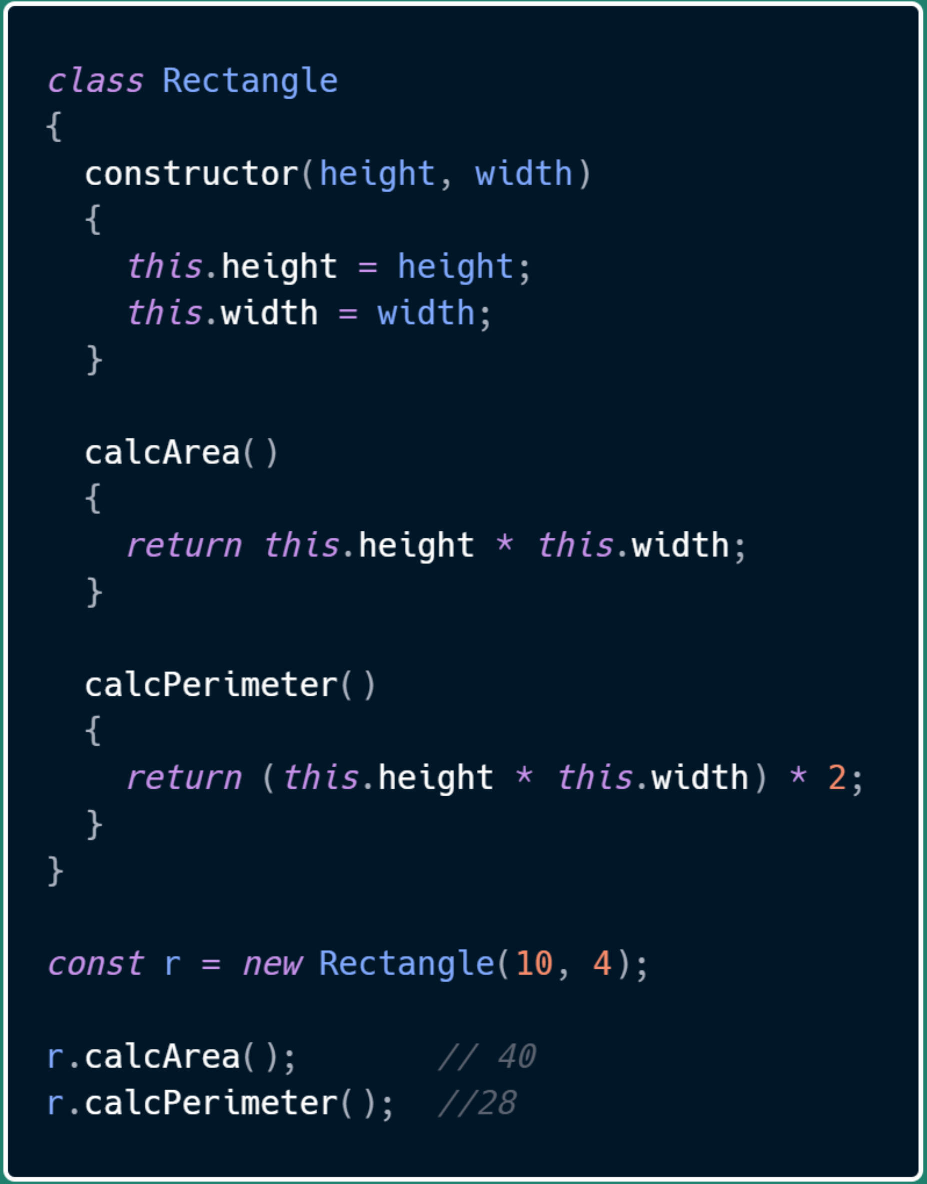
```javascript
// unnamed class
let Rectangle = class{
  constructor(height, width)
  {
    this.height = height;
    this.width = width;
  }
};

let r = new Rectangle(10, 4);

console.log(Rectangle.name); // Rectangle
```

# Class method

**Methods are declared by defining:**

- **Name**
- **Parameters**
- **The body of the function inside curly braces { }**

**Unlike object declaration, there is no need to separate each method with commas ( , ).**



```
class Rectangle
{
  constructor(height, width)
  {
    this.height = height;
    this.width = width;
  }

  calcArea()
  {
    return this.height * this.width;
  }

  calcPerimeter()
  {
    return (this.height * this.width) * 2;
  }
}

const r = new Rectangle(10, 4);

r.calcArea();        // 40
r.calcPerimeter();   //28
```

## Subclass

Classes can be used as a “building block” for other classes. So, we can *inherit* all the properties of the original class and extend them with new one.

In this case we say that the new class is a ***subclass*** of the previous class.

## Subclass

To define a subclass we use the keyword ***extends***.

**The subclass get access to a special keyword super that allows to get access to the method of the inherited class.**

**In the constructor it is required to use super() otherwise the subclass cannot be created.**

```
class Polygon
{
  constructor(height, width)
  {
    this.height = height;
    this.width = width;
  }

  describe()
  {
    return "I'm a polygon";
  }
}
```

```
class Square extends Polygon
{
  constructor(length)
  {
    super(length, length);
    this.name = "Square";
  }

  describe()
  {
    return "I'm a square";
  }
}
```

CODEPEN

## Classes

## Callback

Function accepts variables as parameters but they are not limited to only that.

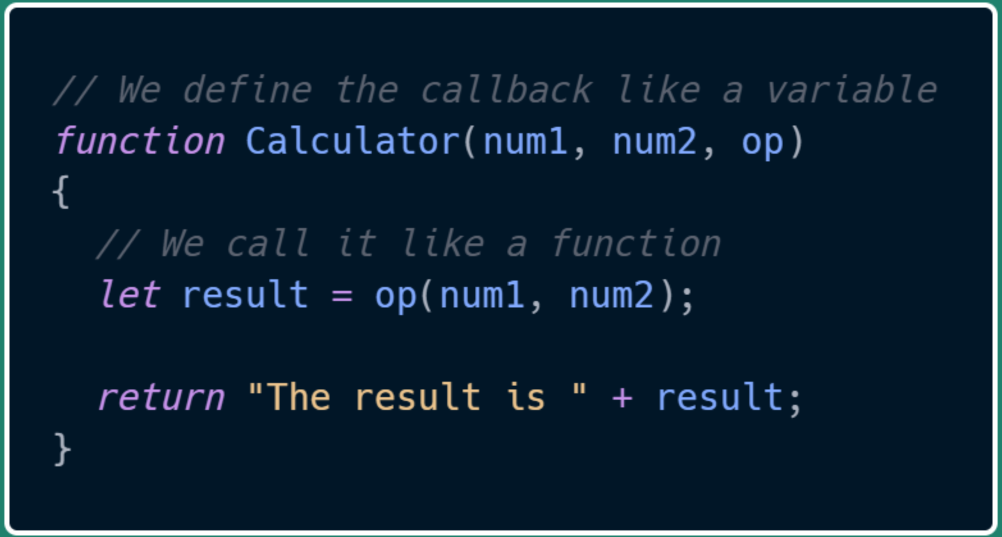
Other functions can be passed as parameter.

A function passed as a parameter is called a callback function.

# Callback

During the declaration, a callback function looks like exactly like other variables.

But, when we use it, we use the typical way to call a function.

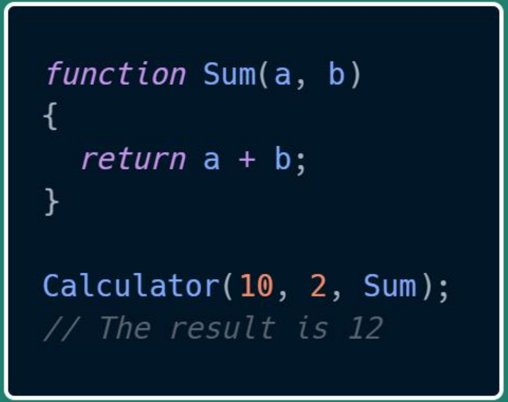


```javascript
// We define the callback like a variable
function Calculator(num1, num2, op)
{
  // We call it like a function
  let result = op(num1, num2);

  return "The result is " + result;
}
```

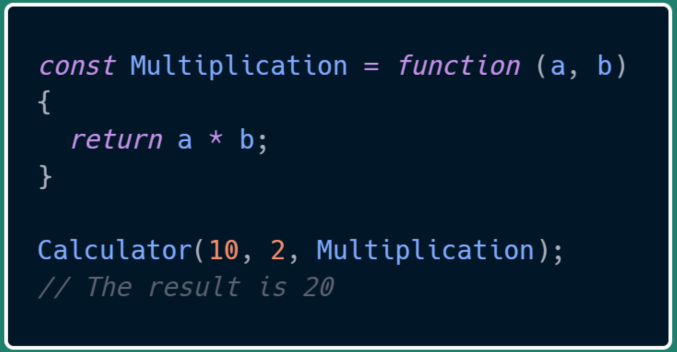
## Callback

**A callback function can be defined by:**

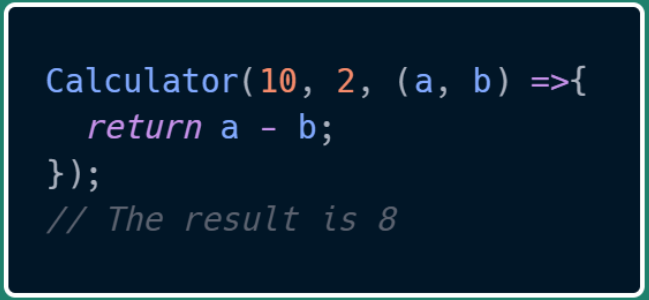
- **Passing the name of the function**
- **Create a function expression and passing the variable holding the function**
- **Using an arrow function**



```
function Sum(a, b)
{
  return a + b;
}

Calculator(10, 2, Sum);
// The result is 12
```



```
const Multiplication = function (a, b)
{
  return a * b;
}

Calculator(10, 2, Multiplication);
// The result is 20
```

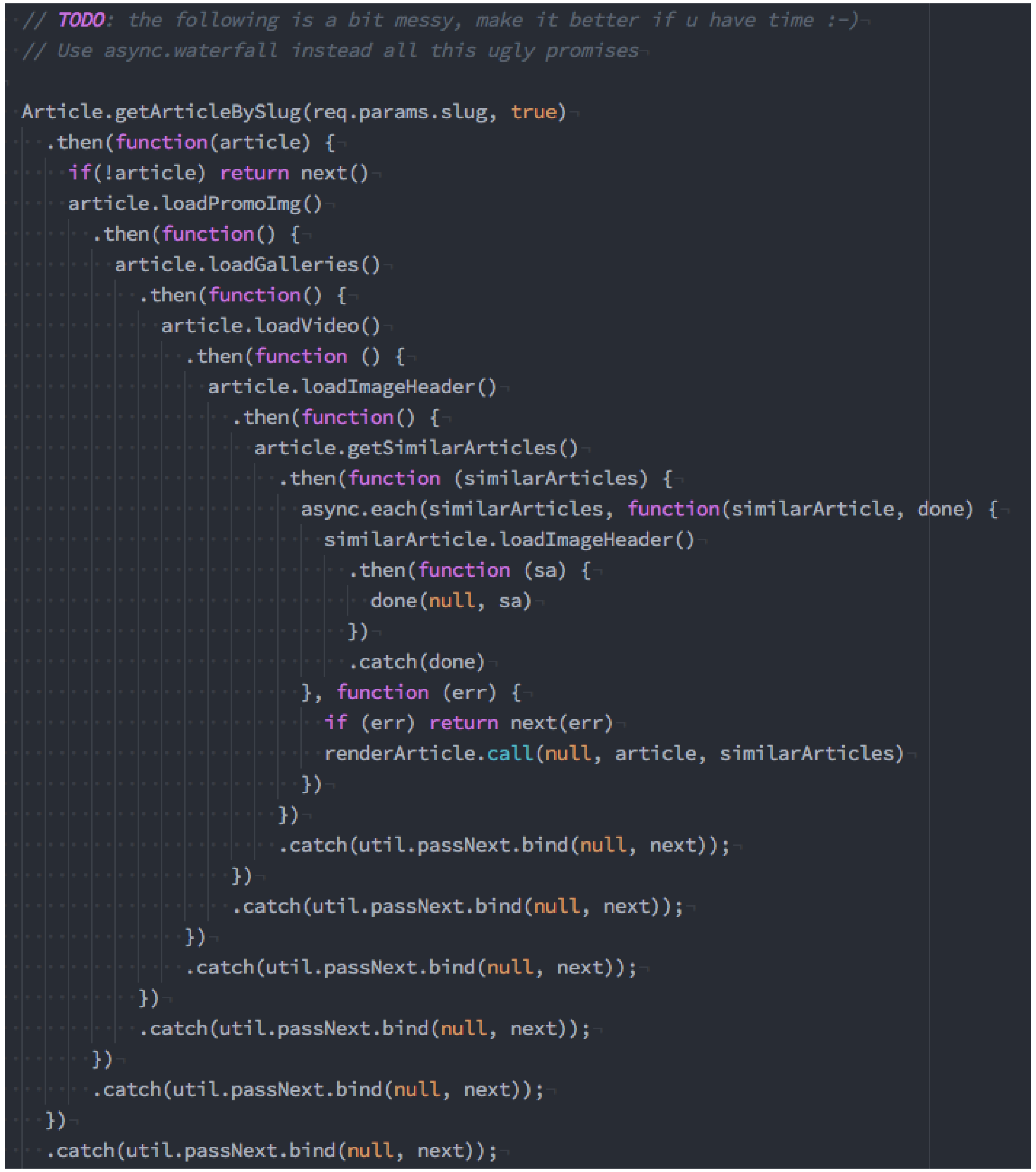


```
Calculator(10, 2, (a, b) =>{
  return a - b;
});
// The result is 8
```

## Callback hell

Many functions in JS require callback to operate properly: they are simple to implement and give a bit of customization of the behaviour.

It is easy to exaggerate with the callbacks.



```
// TODO: the following is a bit messy, make it better if u have time :-)
// Use async.waterfall instead all this ugly promises

Article.getArticleBySlug(req.params.slug, true)
  .then(function(article) {
    if(!article) return next()
    article.loadPromoImg()
      .then(function() {
        article.loadGalleries()
          .then(function() {
            article.loadVideo()
              .then(function () {
                article.loadImageHeader()
                  .then(function() {
                    article.getSimilarArticles()
                      .then(function (similarArticles) {
                        async.each(similarArticles, function(similarArticle, done) {
                          similarArticle.loadImageHeader()
                            .then(function (sa) {
                              done(null, sa)
                            })
                            .catch(done)
                        }, function (err) {
                          if (err) return next(err)
                          renderArticle.call(null, article, similarArticles)
                        })
                      })
                      .catch(util.passNext.bind(null, next));
                  })
                  .catch(util.passNext.bind(null, next));
              })
              .catch(util.passNext.bind(null, next));
          })
          .catch(util.passNext.bind(null, next));
      })
      .catch(util.passNext.bind(null, next));
  })
  .catch(util.passNext.bind(null, next));
```

## How to avoid callback hell

There are many ways to avoid callback hell:

- Split callbacks into normal functions
- Use promises
- Use async/await

If neither of these is possible: use comments.

Example of avoiding callback hell.

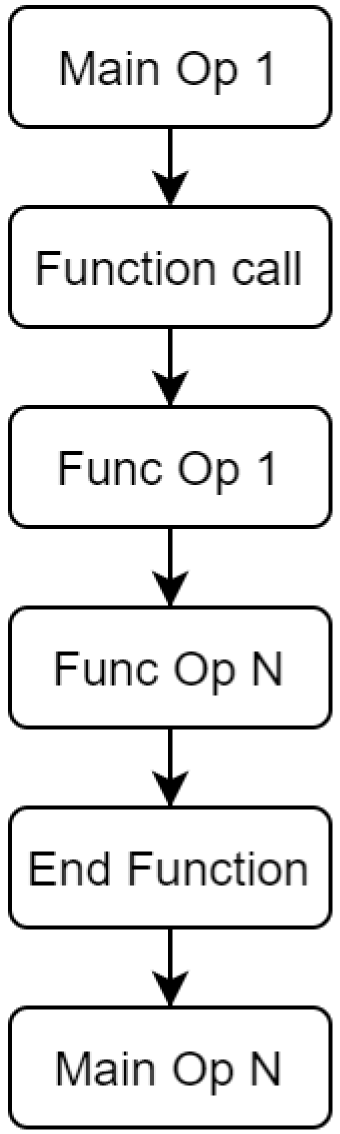
CODEPEN

Callback

## Synchronous

So far, the execution of the instructions was sequential:
the operations are always executed in the same order and you have to wait for the operation to finish before moving on to another task.

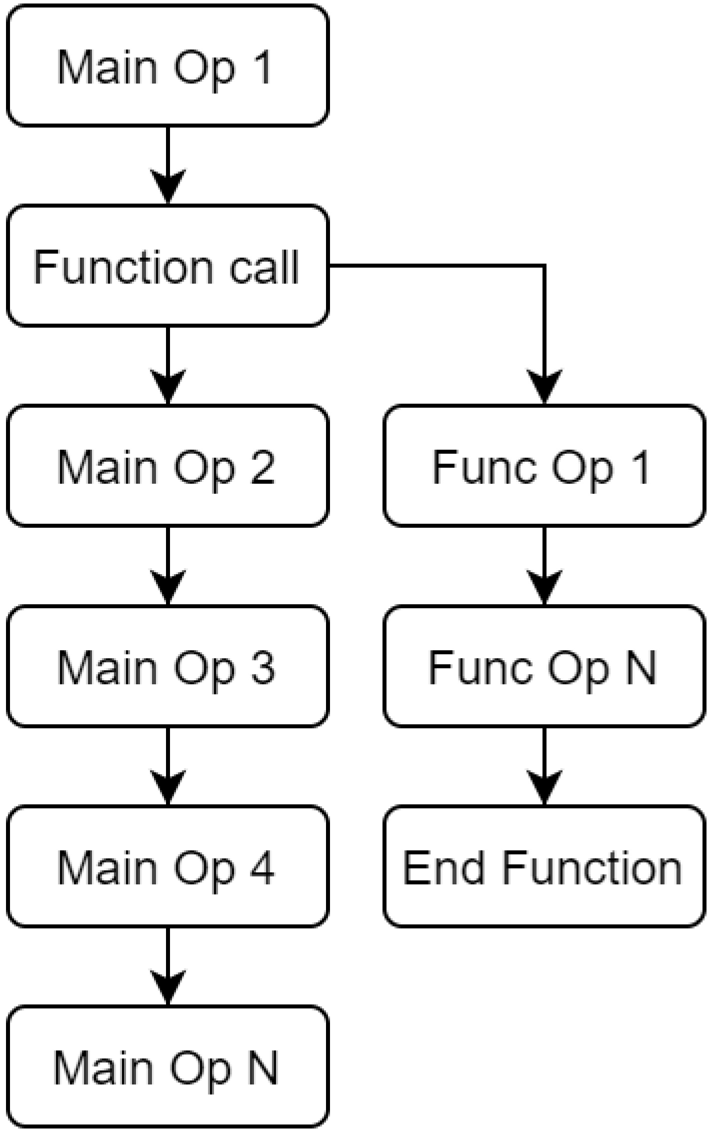
In this case we talk about **synchronous** execution.



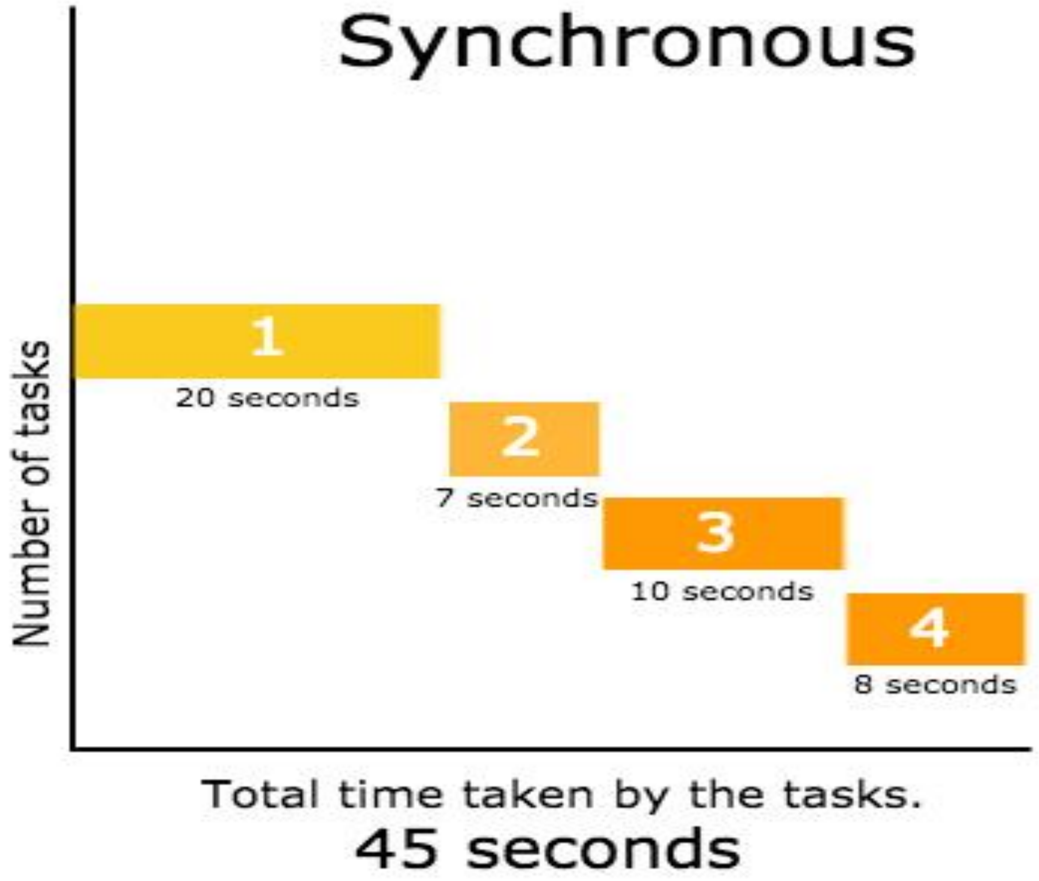
## Asynchronous

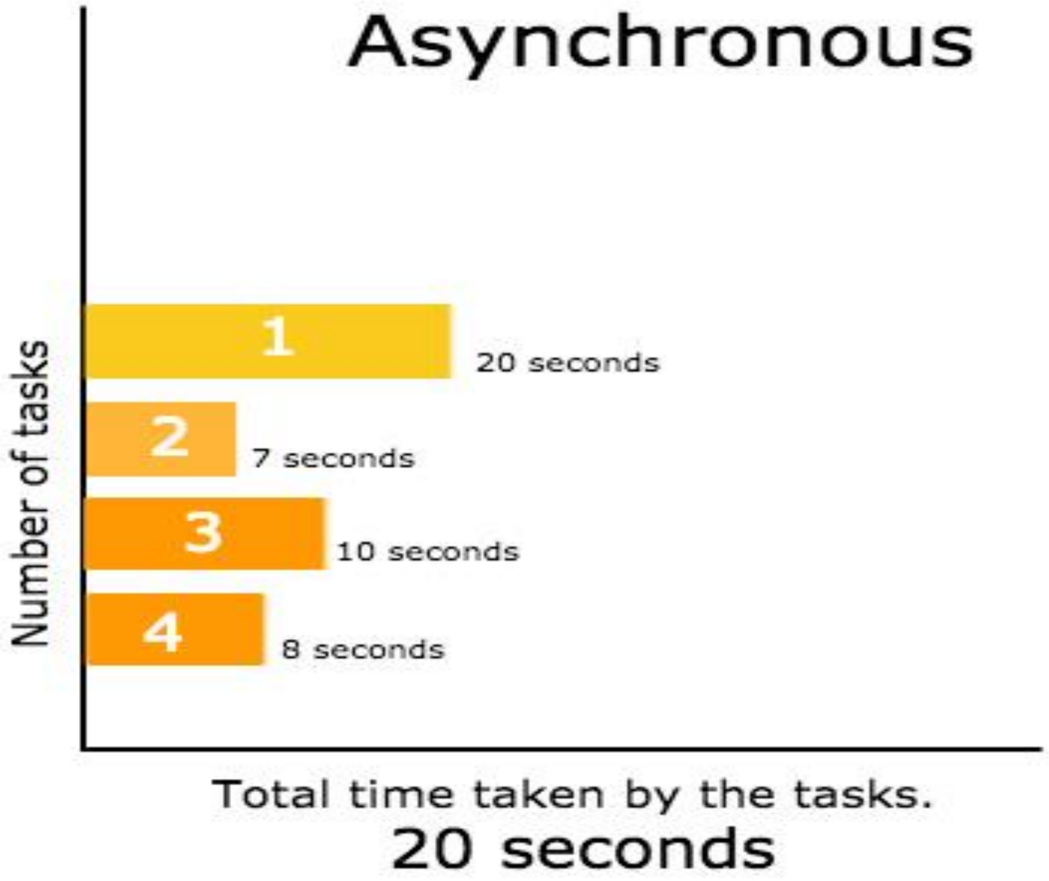
The code can also be executed **asynchronously**:
the execution of the operations can be overlapped meaning that multiple operation can be performed “at the same time”.

Thanks to **asynchronous** execution you are not forced to wait for some operations to be concluded before performing other tasks.



## Comparison


Synchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
45 seconds


Asynchronous
Number of tasks
1
20 seconds
2
7 seconds
3
10 seconds
4
8 seconds
Total time taken by the tasks.
20 seconds

## Promise

A promise is an object related to an asynchronous task.

When you create a Promise, a task will be started and you are “promised” that the operations will be completed at some point in time.

## Promise

During its execution, a Promise can be in one of these states:

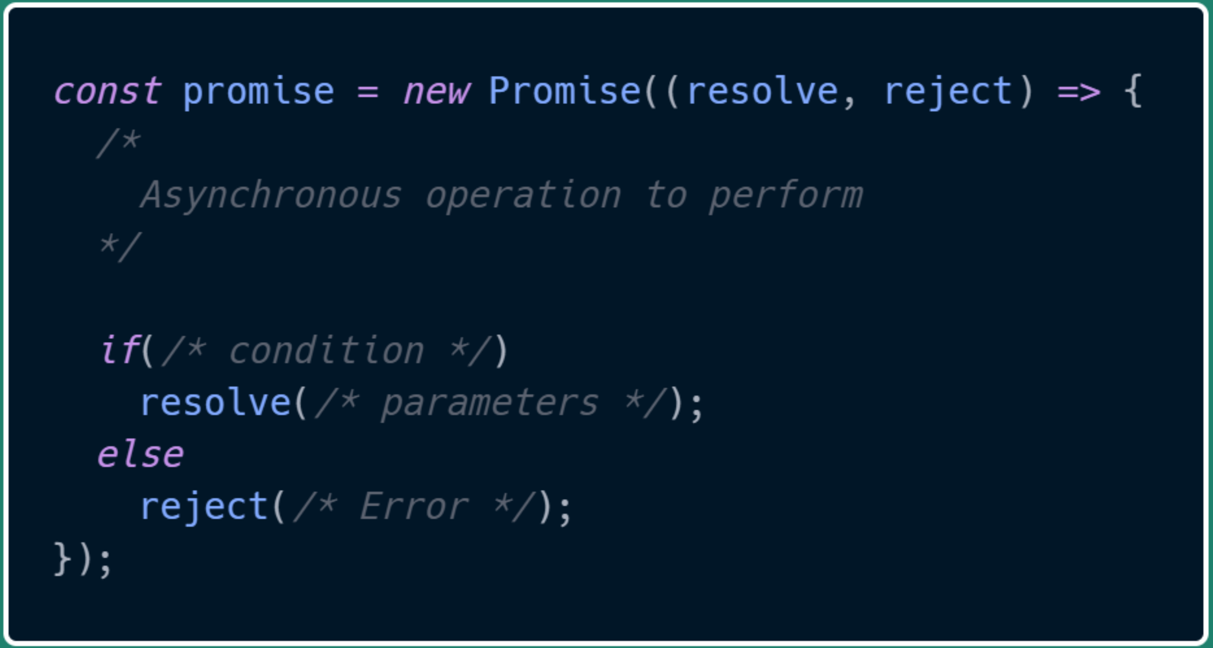
- pending: initial state, neither fulfilled nor rejected
- fulfilled: the operation was completed successfully
- rejected: the operation failed

# Promise

**A Promise can be created by defining a new Promise object.**

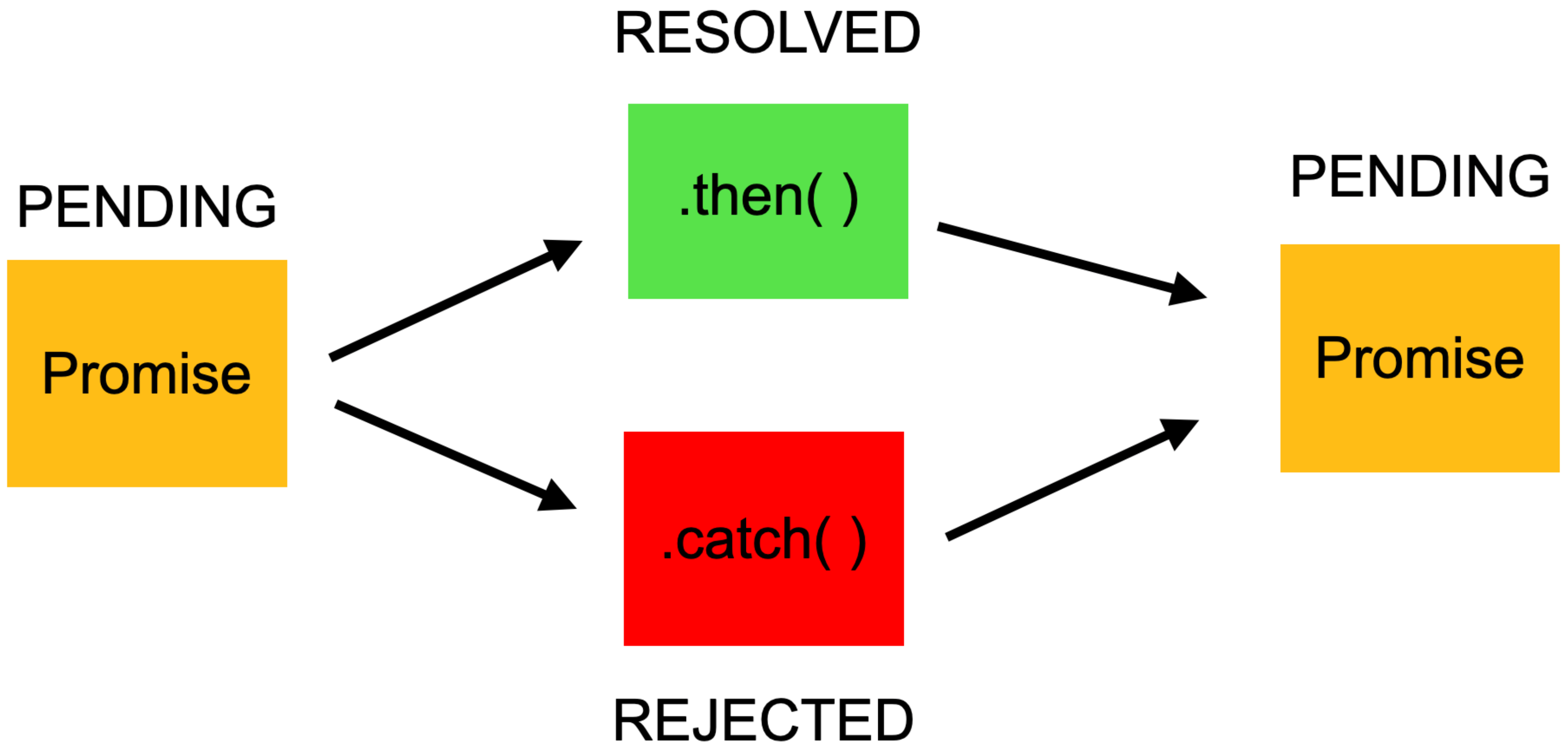
**A Promise requires a callback function that specify what are the asynchronous operation to perform.**

**The object offers two functions that allow to set the state of the Promise as either fulfilled or rejected.**



```
const promise = new Promise((resolve, reject) => {
  /*
    Asynchronous operation to perform
  */

  if(/* condition */)
    resolve(/* parameters */);
  else
    reject(/* Error */);
});
```

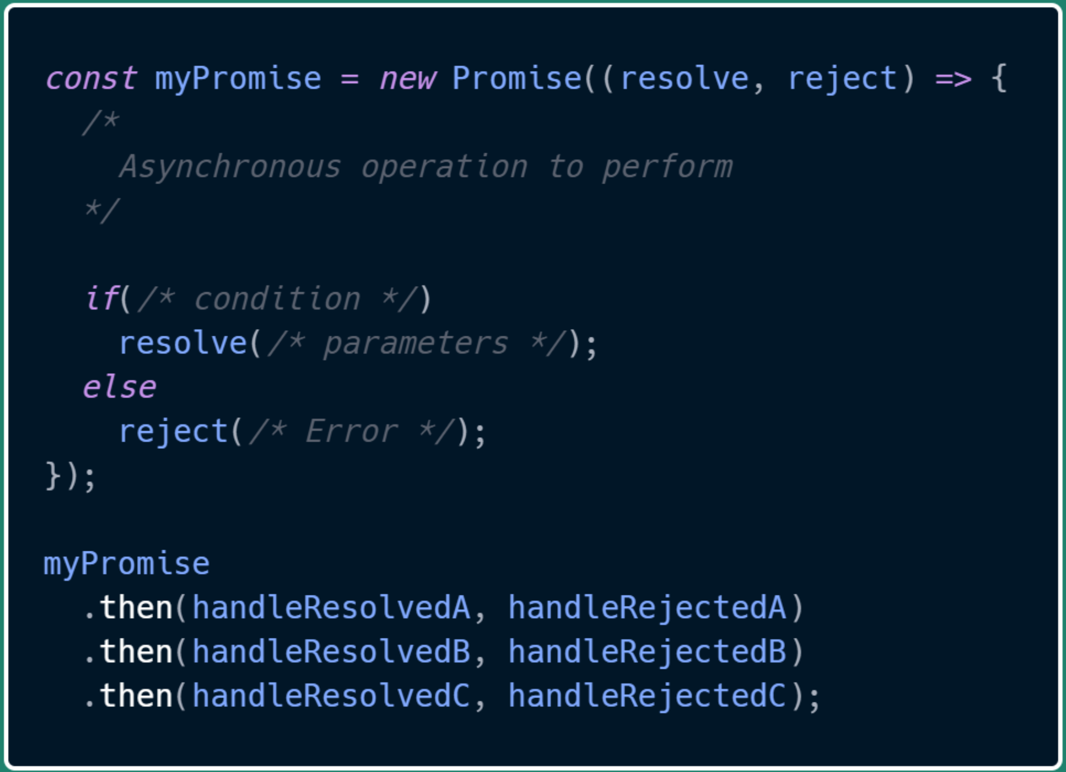
## Promise chaining



# Promise chaining

**Promises can be chained in order to have multiple asynchronous operation that works in a sequence.**

**We use the .then() (like in the case of the async function) to define what operations should be performed.**



```
const myPromise = new Promise((resolve, reject) => {
  /*
    Asynchronous operation to perform
  */

  if(/* condition */)
    resolve(/* parameters */);
  else
    reject(/* Error */);
});

myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

## Promise chaining

**When a Promise is defined, there is no need to pass both callback functions.**
**The reject function can be omitted but in this case, nothing will be done in case of rejection.**

**To avoid this, we can use .catch() to pick up any rejection that happened during the execution.**

```
const myPromise = new Promise((resolve, reject) => {
  /*
    Asynchronous operation to perform
  */

  if(/* condition */)
    resolve(/* parameters */);
  else
    reject(/* Error */);
});

myPromise
  .then(handleResolvedA)
  .then(handleResolvedB)
  .then(handleResolvedC)
  .catch(handleRejected);
```

## Async

**A Promise can also be generated by using functions declared as asynchronous.**

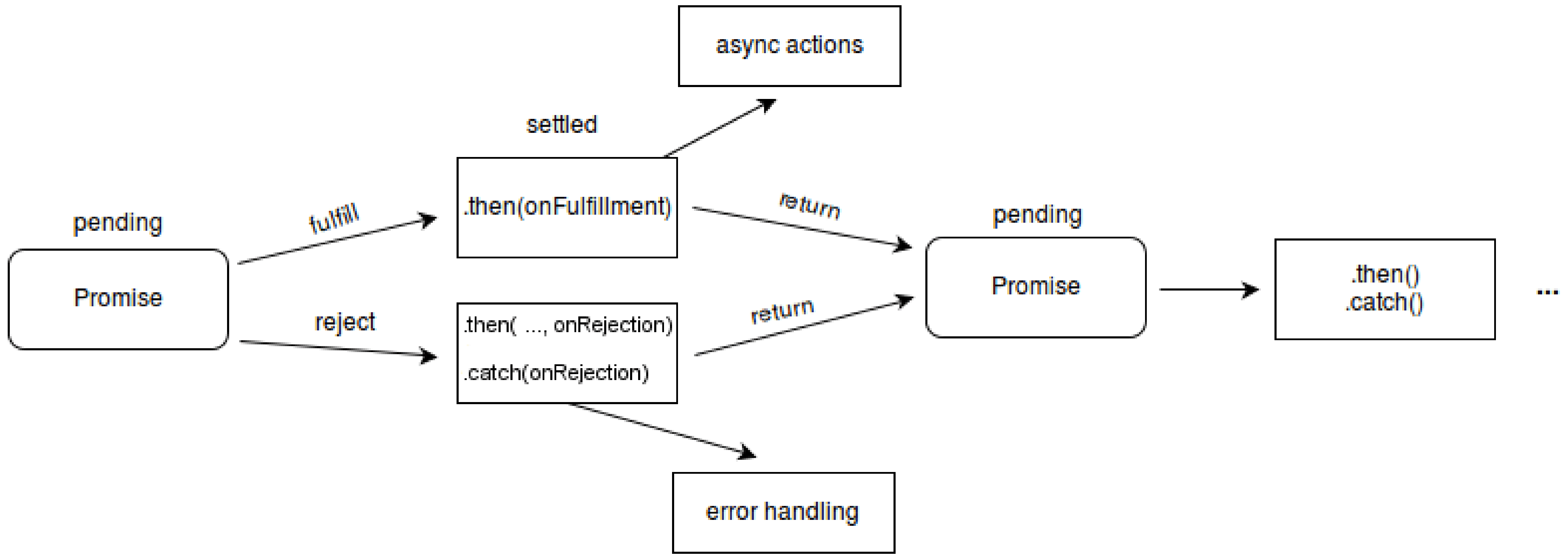
**We use the *async* keyword to define this behaviour.**

**From this we can do chaining like we did by using the Promise object.**

```
async function doAsynchronous()
{
  /*
    All the operation defined here
    will be performed asynchronously
    to the main code
  */
}
```

```
function resolve()
{
  /*
    What needs to be done in case of fulfilled
  */
}

function reject()
{
  /*
    What needs to be done in case of rejected
  */
}

doAsynchronous().then(resolve, reject);
```

# Promise execution



CODEPEN

Promise

## Await

The ***await*** operator is used to wait for a Promise.

It blocks the execution of the rest of the code in order to wait for the Promise to be completely evaluated.

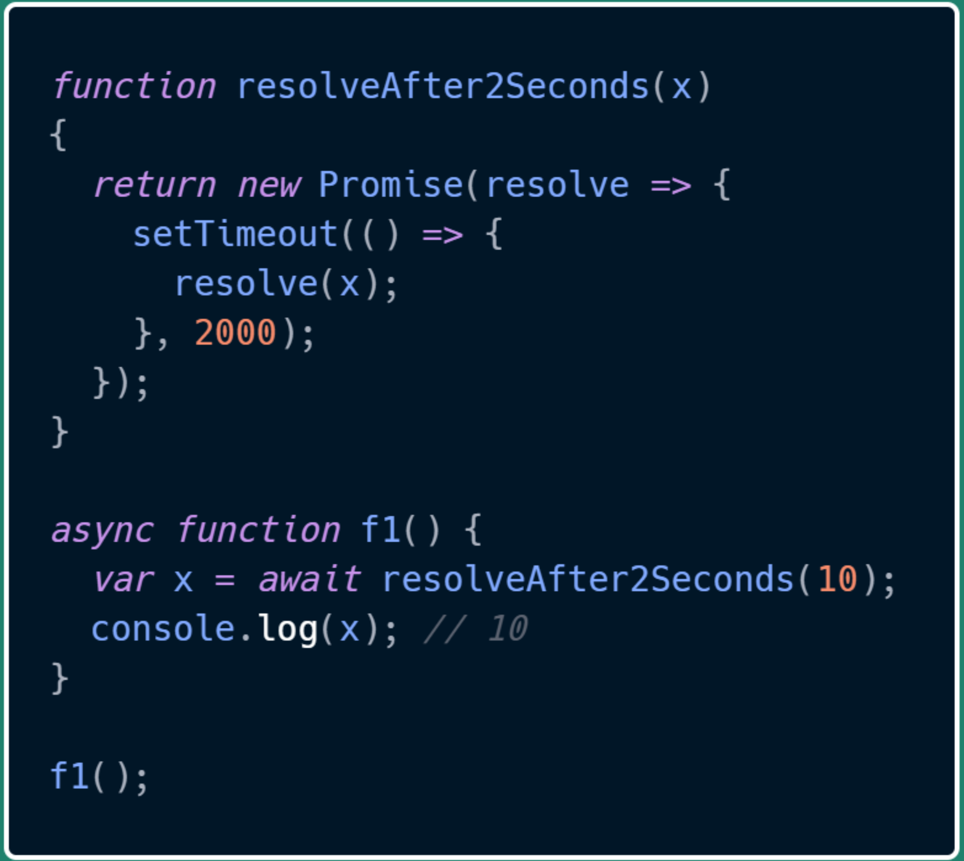
It works only inside async functions (and with JS modules).

# Await

**We use the await operator before calling an async function or creating a new Promise object.**

**The await operator changes the return value of a Promise into the actual result of the Promise.**

**NB: if you want to return a value after rejection, you need a way to handle the rejection in the chain (e.g. using a catch), otherwise no value will be returned.**

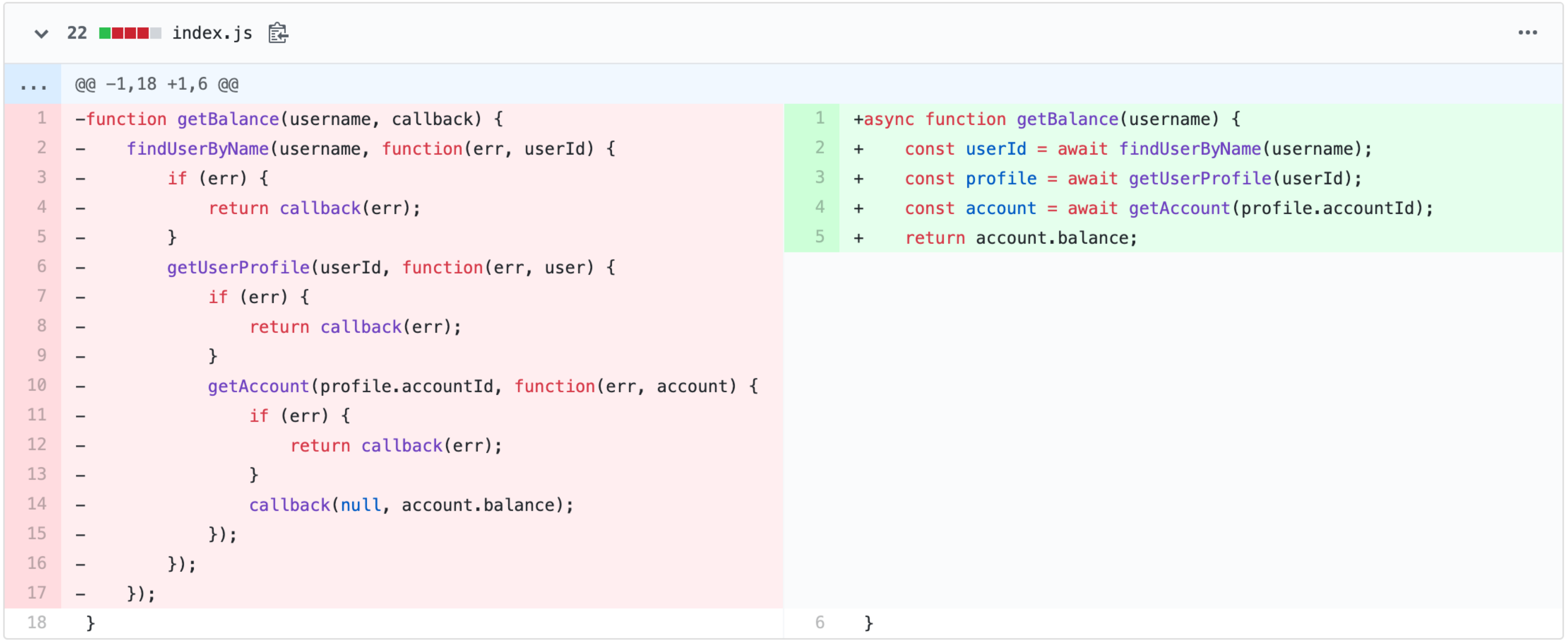


```
function resolveAfter2Seconds(x)
{
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function f1() {
  var x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}

f1();
```

# Await

By using the *await* keyword, there is no need to use Promise chaining.

We receive the value directly and we don't have to pass it to a callback meaning that we are able to avoid callback hell.



22 index.js

```
@@ -1,18 +1,6 @@
-function getBalance(username, callback) {
-    findUserByName(username, function(err, userId) {
-        if (err) {
-            return callback(err);
-        }
-        getUserProfile(userId, function(err, user) {
-            if (err) {
-                return callback(err);
-            }
-            getAccount(profile.accountId, function(err, account) {
-                if (err) {
-                    return callback(err);
-                }
-                callback(null, account.balance);
-            });
-        });
-    });
 }
```

```
+async function getBalance(username) {
+    const userId = await findUserByName(username);
+    const profile = await getUserProfile(userId);
+    const account = await getAccount(profile.accountId);
+    return account.balance;
 }
```

CODEPEN

Await

## API

An API (Application Programming Interface) is a kind of library that gives access to a set of already made function.

APIs allow developers to take “shortcut” when developing something by reusing something that someone else already made.

## API

The client APIs fall in two categories:

- Browser APIs (or web APIs) that are built into the web browser and are directly available with JS.
- Third-party APIs that usually require to retrieve data from the web.

## Web API

Browser APIs are usually accessible as objects which contains the data the API uses and all the functionality available.

Whenever you want to access a web API, you should first know where the “entry point” is.

## Web API

Some example of Browser API:

- DOM API → document
- BOM API → window
- Canvas API → HTMLCanvasElement.getContext()
- XMLHttpRequest → new XMLHttpRequest()
- Fetch → fetch()

List of available Browser API.

## Third-party API

To access third-party APIs we need to send a request to the server that hosts the service using its URL.

We can send this request using:

- XMLHttpRequest
- Fetch

## XMLHttpRequest

This object is somewhat deprecated by Fetch but it is still used for three reason:

- Need to support existing scripts that use it
- Need to support old browsers
- Need something that Fetch can’t do (yet): track upload progress

# XMLHttpRequest

**The XMLHTTPRequest is an object that require to “open” a request to an URL.**

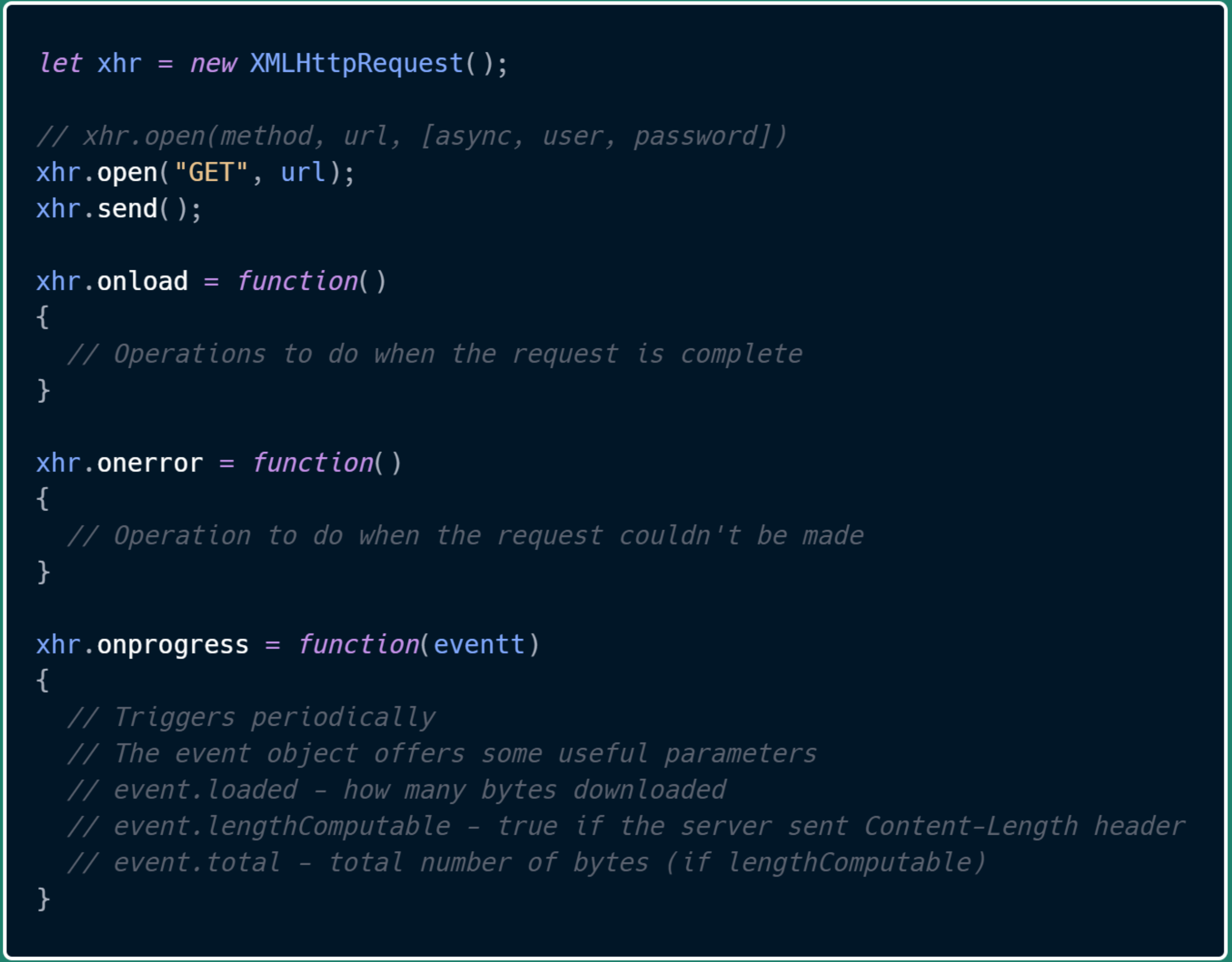
**Once the object is ready it is possible to “send” the request.**

```
let xhrGet = new XMLHttpRequest();

// xhr.open(method, url, [async, user, password])
xhrGet.open("GET", url);
xhrGet.send();

let xhrPost = new XMLHttpRequest();

xhrPost.open("POST", url);
// A post request needs an object containing the data
xhrPost.send(dataObject);
```

## XMLHttpRequest

**The XMLHttpRequest offers the possibility to listen to some events:**

- **load – triggers when the request is complete**
- **error – triggers when the request couldn’t be made**
- **progress – triggers periodically while the response is being downloaded**



```
let xhr = new XMLHttpRequest();

// xhr.open(method, url, [async, user, password])
xhr.open("GET", url);
xhr.send();

xhr.onload = function()
{
  // Operations to do when the request is complete
}

xhr.onerror = function()
{
  // Operation to do when the request couldn't be made
}

xhr.onprogress = function(eventt)
{
  // Triggers periodically
  // The event object offers some useful parameters
  // event.loaded - how many bytes downloaded
  // event.lengthComputable - true if the server sent Content-Length header
  // event.total - total number of bytes (if lengthComputable)
}
```

## Fetch

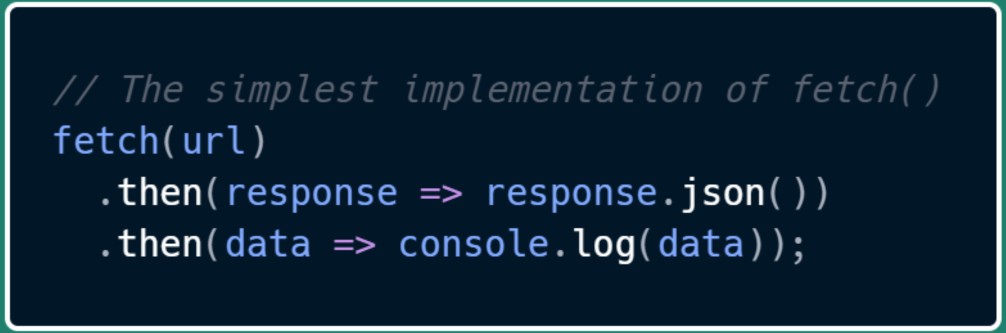
The Fetch API is another way to manage request and response.

Compared with XMLHttpRequest it is easier, more streamlined and uses Promise to manage the operations instead of callbacks.

## Fetch

**To send a request using the Fetch API, we only use the fetch() method.**

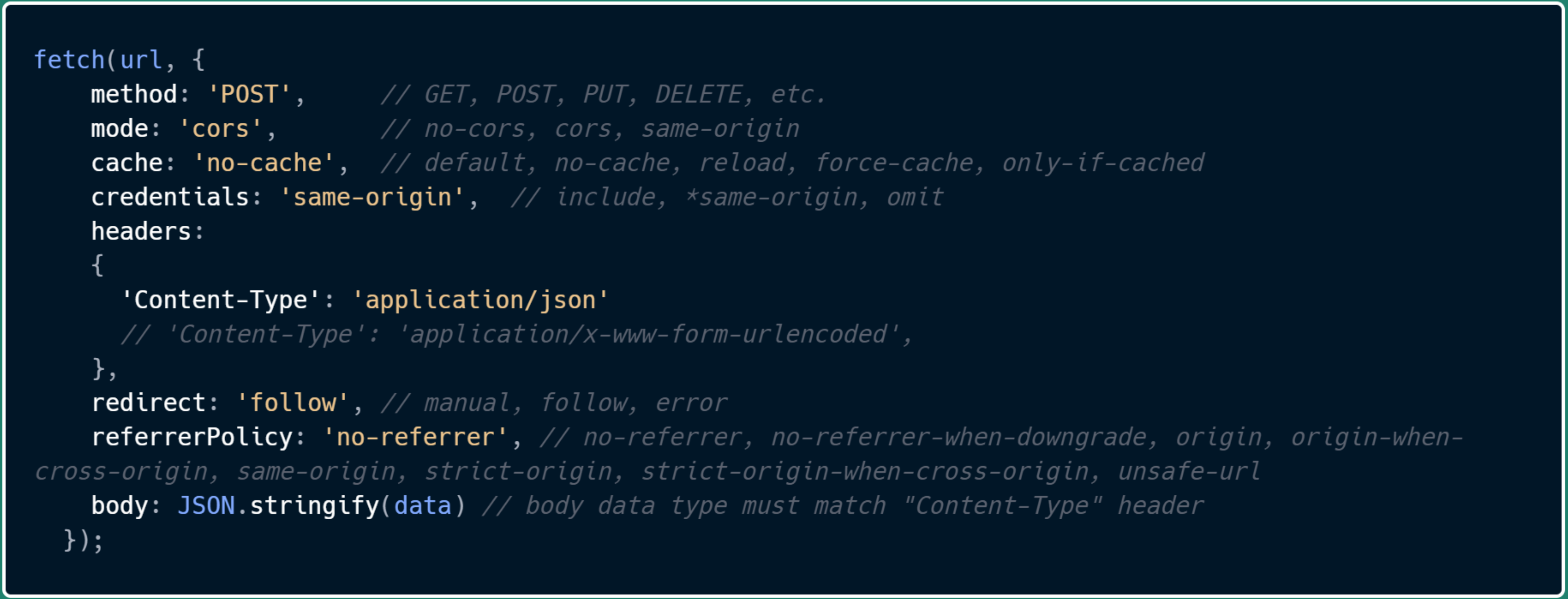
**Any additional operation is done by chaining Promises**



```
// The simplest implementation of fetch()
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data));
```

# Fetch

**In case we need to set up additional parameters, we can just add them as an object to the request. [Here is a list of possible values.]()**



```
fetch(url, {
    method: 'POST',      // GET, POST, PUT, DELETE, etc.
    mode: 'cors',        // no-cors, cors, same-origin
    cache: 'no-cache',  // default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin',  // include, *same-origin, omit
    headers:
    {
      'Content-Type': 'application/json'
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, follow, error
    referrerPolicy: 'no-referrer', // no-referrer, no-referrer-when-downgrade, origin, origin-when-
cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url
    body: JSON.stringify(data) // body data type must match "Content-Type" header
  });
```

CODEPEN

API

## Shadow DOM

Every time we submit some elements into the DOM tree, we have to be cautious of existing IDs, css selector that can cause conflicts.

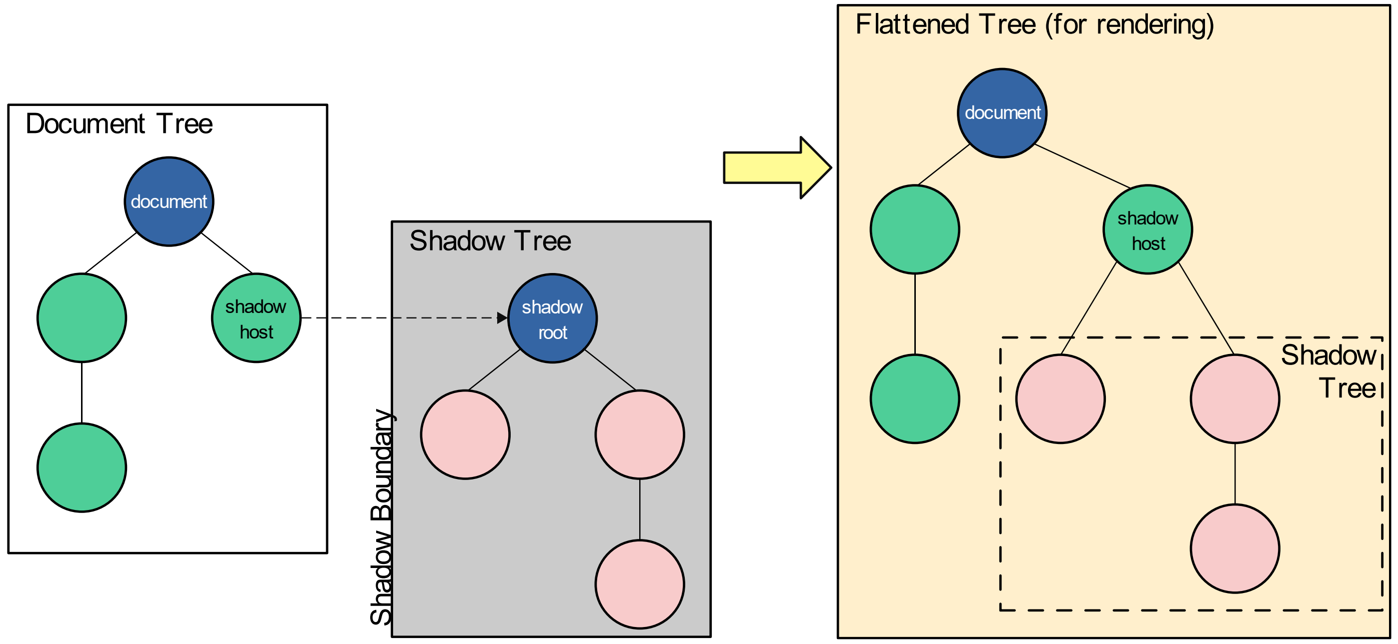
One way to avoid these problems is shadow DOM.

A shadow DOM is an independent tree-like structure that can be created from any node in the original DOM.

# Shadow DOM

Some terminology:

- **Shadow host**
  The regular DOM node that the shadow DOM is attached to.
- **Shadow tree**
  The DOM tree inside the shadow DOM.
- **Shadow boundary**
  the place where the shadow DOM ends, and the regular DOM begins.
- **Shadow root**
  The root node of the shadow tree.



## Shadow DOM

But, what are the benefit?

- It’s isolated, a component’s DOM is self-contained
- It simplifies CSS thanks to the fact that you don’t have to worry about naming conflicts
- It allows the creation and management of custom elements

# Shadow DOM

**To create a shadow DOM we need two things:**

- **pick the shadow host**
- **create the shadow root.**

**Once it is created, we can use the shadow root like the document object to find nodes and append elements.**

```javascript
// We defined a div element with id "shadow" in the HTML file
var div = document.querySelector("#shadow");
const shadowRoot = div.attachShadow({mode: 'open'});

/*
 Now the shadowRoot works like the document object.
*/

var node = document.createElement("p");
node.innerText = "I'm in the shadow DOM";
shadowRoot.appendChild(node);

shadowRoot.querySelector("p");
//returns the element that was just created
```

CODEPEN

Shadow DOM