Programming JavaScript

Santosh Kalwar, 27.02.2022

JavaScript Topics

- General Discussions

- Working with Fetch / JSON
  - GET/PUT/POST/DELETE
  - Practice / Classroom coding

- Some other topics
  - Legacy var
  - Hoisting
  - Closures
- Practice / Classroom coding

- Modules
  - Import from libraries
  - Parcel guide
- Practice / Classroom coding

General Discussions

- General Discussions

- ToDo app + Final project (discussion with **Margit** ongoing)
  - Some kind of “Recipe” app or “Food tracker app” using real API (just like your mini projects) using plain JS, HTML and CSS (no libraries or framework allowed) or using Firebase API

- Final project Q&A

Remaining Lessons

- 1.03 (Remaining JS topics, and usual class)
- 7.03 (Remaining JS topics, practice as usual)
- 8.03 (Remaining JS topics, final project work introduction)
  - If there is some confusion in some JS topics, we will go through those together again
- 14.03 (No JS topics only practice, project work and recap)
  - If there is some confusion in some JS topics, we will go through those together again
- 15.03 (Last lesson of Programming JS. Project submission and the END)

Fetch and FetchWrapper

Let's start with the basics:

```
fetch( URL)
```

This is how you fetch a URL. You need to replace the URL with an actual URL string. So let's take an example, with a **sample API** which I have built for you to use:

https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/notifications.json

Open this URL in a new tab and notice that it returns JSON. This is how we communicate with external services.

This API lets us know that there are 3 new notifications.

Fetch and FetchWrapper

Now, we need to do the same process but with a piece of JavaScript code. So, we need to be able to send a request to this URL and get back its result. To do that, we use the fetch API .

So, in our example, we'll need to fetch this URL that we linked to above. This is how it will look like:

```
fetch( " https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/notifications.json" )
```

This is the **first** step to working with the fetch API.
It will send a request (sometimes called AJAX request or XMLHttpRequest for historical reasons). This request will go to the Internet, reach the URL you specified and finally come back to you with the response (result) from that URL.

Fetch returns a promise

This is extremely important. Fetch always returns a **promise** .

This is because fetch has to go to the network, this could take anywhere between a couple of milliseconds and a second (on average). So, we cannot freeze the entire browser while the fetch request is working.

Thus, by design, fetch returns a promise that we can resolve once the request has finished.

This means that we'll have to add a .then() after the fetch() call.

Why have a fetch wrapper

A fetch wrapper is not absolutely necessary, but, it does make working with fetch easier and more eloquent.

A fetch wrapper is a class that wraps the `fetch` API in a way that works specifically for our scenario. When building a website as web developers, you and I will often work with the following:

- mostly the same API (same base URL but different endpoints)
- a JSON API (an API that returns JSON)

This means that for every `fetch` request we make, we'll have quite some repetition. For example, assuming the following API documentation:

Base URL: **https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/**

Endpoint :

- **GET** `/notifications.json`
- **GET** `/chapters.json`

Why have a fetch wrapper

If we have to send 2 fetch requests to both endpoints, we'll have quite some repetition:

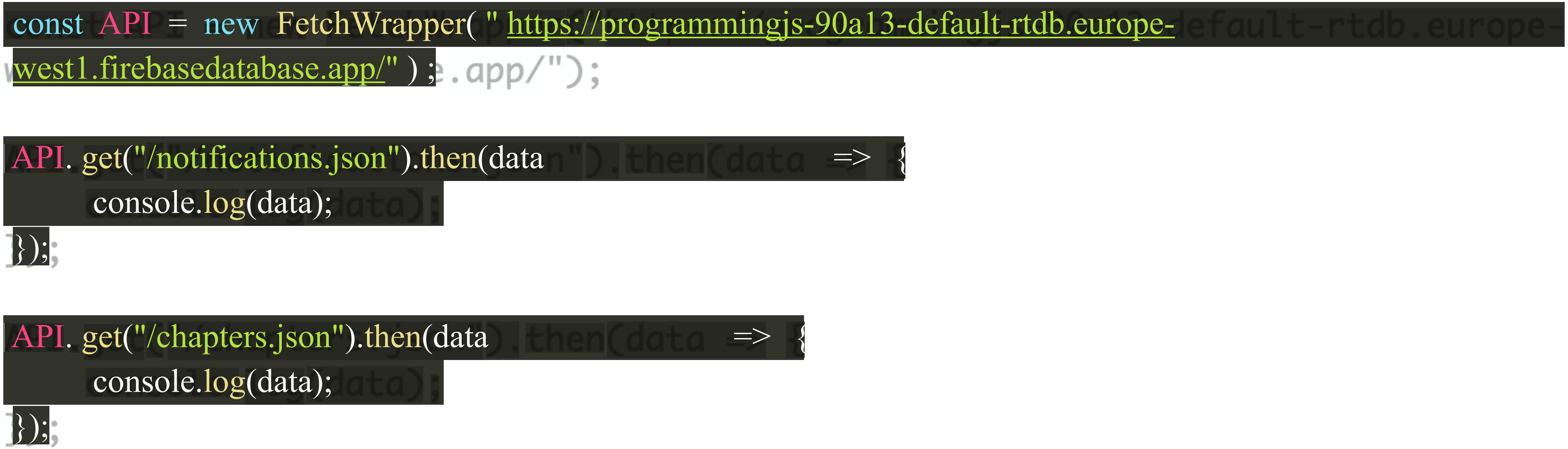
```
fetch( `https://programmingjs-90a13-default-rtdb.europe-
west1.firebasedatabase.app/notification.json`)
 . then(response => response.json())
 . then(data => {
        console.log(data);
});
```

```
fetch( `https://programmingjs-90a13-default-rtdb.europe-
west1.firebasedatabase.app/chapters.json`)
 . then(response => response.json())
 . then(data => {
        console.log(data);
});
```

A real-life app would have anywhere between 20 and 100+ fetch requests to the same API (same base URL but different endpoints). This is a lot of repetition.

Why have a fetch wrapper

This is exactly why a fetch wrapper class can come in handy. Assuming we already built this fetch wrapper, we can re-write the code above such that it looks like the following:



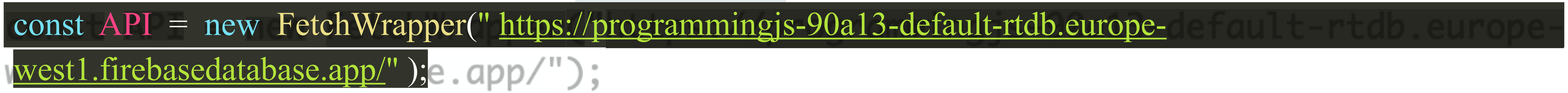
```
const API = new FetchWrapper( " https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/" ) ;

API. get("/notifications.json").then(data => {
        console.log(data);
});

API. get("/chapters.json").then(data => {
        console.log(data);
});
```

This FetchWrapper class has the following benefits :

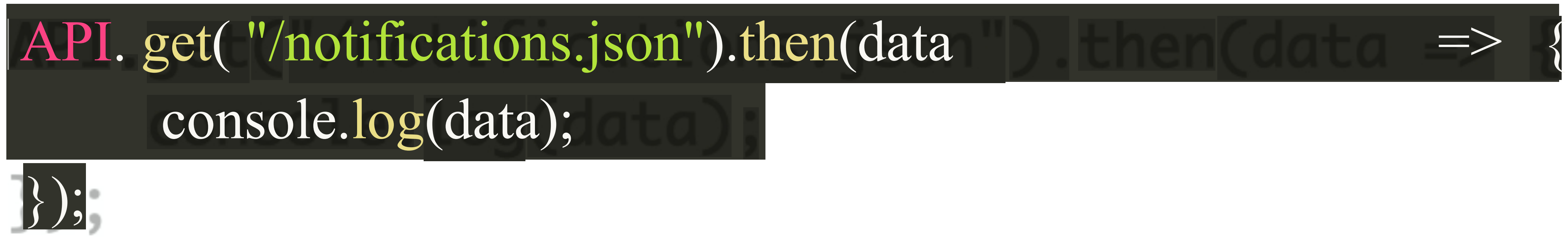
- we only need to set the **base URL** once (we pass it to the constructor).
- It always converts the response to JSON (response => response.json()), since our API always returns JSON.

Implementing GET

By looking at the sample usage of this FetchWrapper class:



```
const API = new FetchWrapper("https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/");
```

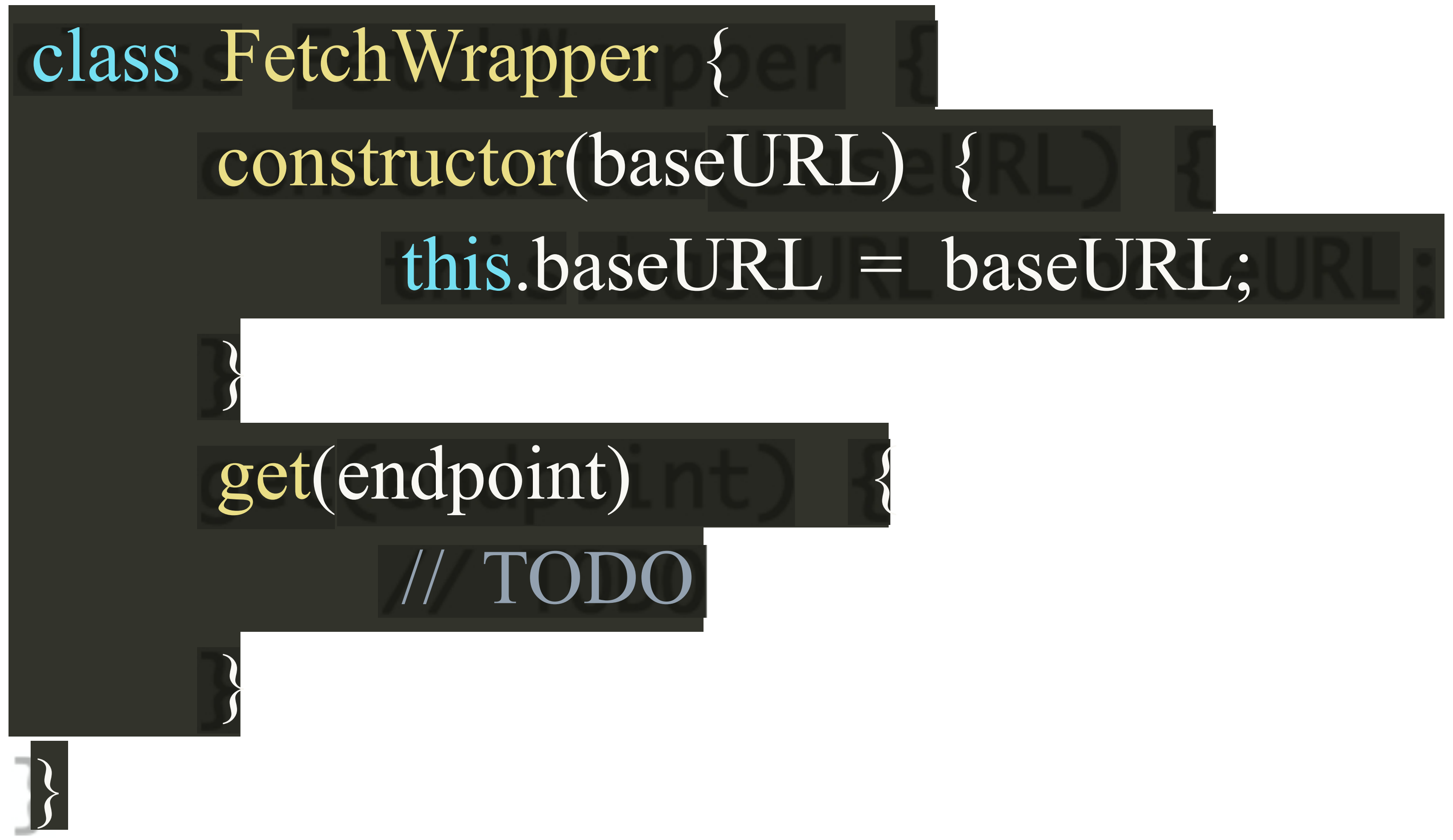


```
API.get("/notifications.json").then(data => {
    console.log(data);
});
```

We can deduce that we need the following:

- A class called FetchWrapper
- A constructor() that accepts the baseURL. We need to capture this baseURL as an **instance variable** so that we can use it in the **instance methods** of this class.
- An instance method called get() that accepts the endpoint.

So, after implementing the above, we end up with the following:

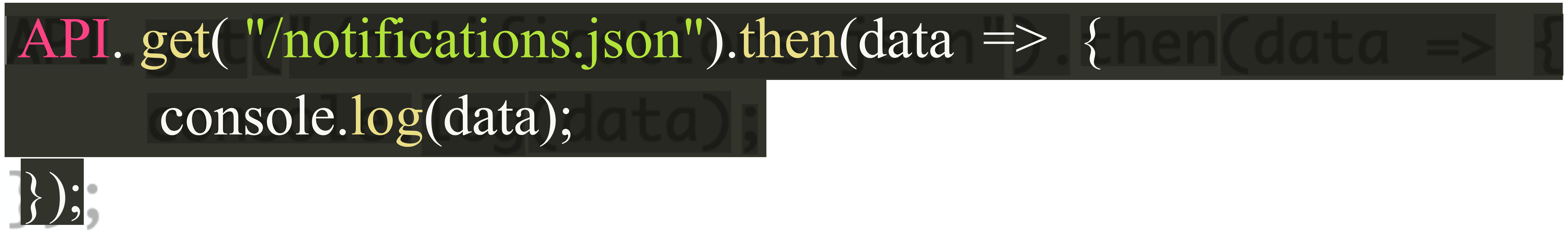


```
class FetchWrapper {
    constructor(baseURL) {
        this.baseURL = baseURL;
    }
    get(endpoint) {
        // TODO
    }
}
```

Implementing get(endpoint)

The last necessary step is to implement the get(endpoint) instance method.

Let's take another look at the sample usage:



```
API. get( "/notifications.json").then(data => {
        console.log(data);
});
```

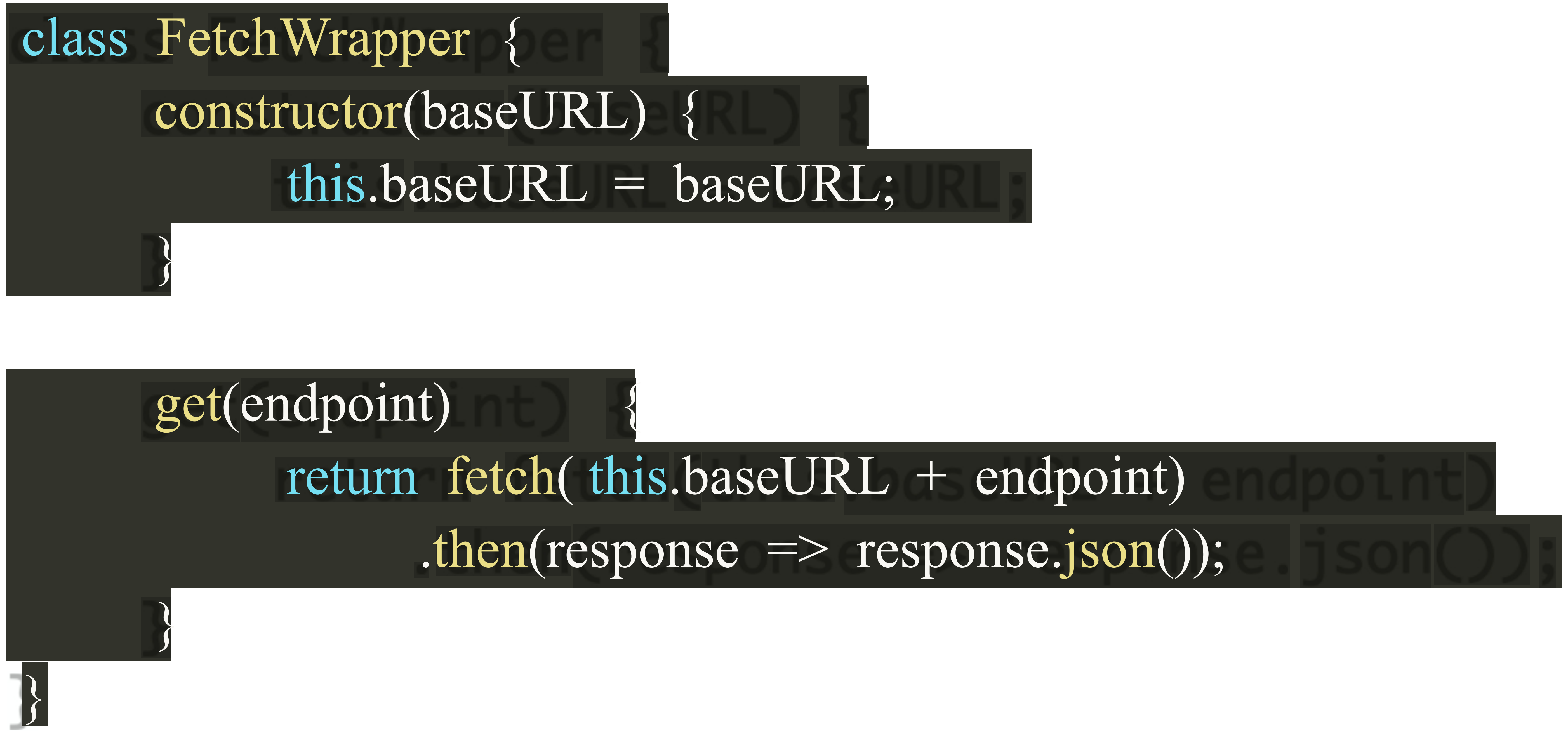
The code above is sending a fetch request to **https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/notification.json** . This means that the fetch request is being sent to baseURL + endpoint.

Also, the response is **automatically** converted to JSON (response => response.json()) .

And, finally, the get(endpoint) method returns a **promise** . When we **resolve** this promise, we get back the data. This means that the get(endpoint) method has to return the result of fetch(...).then(response => response.json()).

Implementing get(endpoint)

Taking all of this into consideration, here's how we can implement it:



```
class FetchWrapper {
    constructor(baseURL) {
        this.baseURL = baseURL;
    }

    get(endpoint) {
        return fetch(this.baseURL + endpoint)
            .then(response => response.json());
    }
}
```

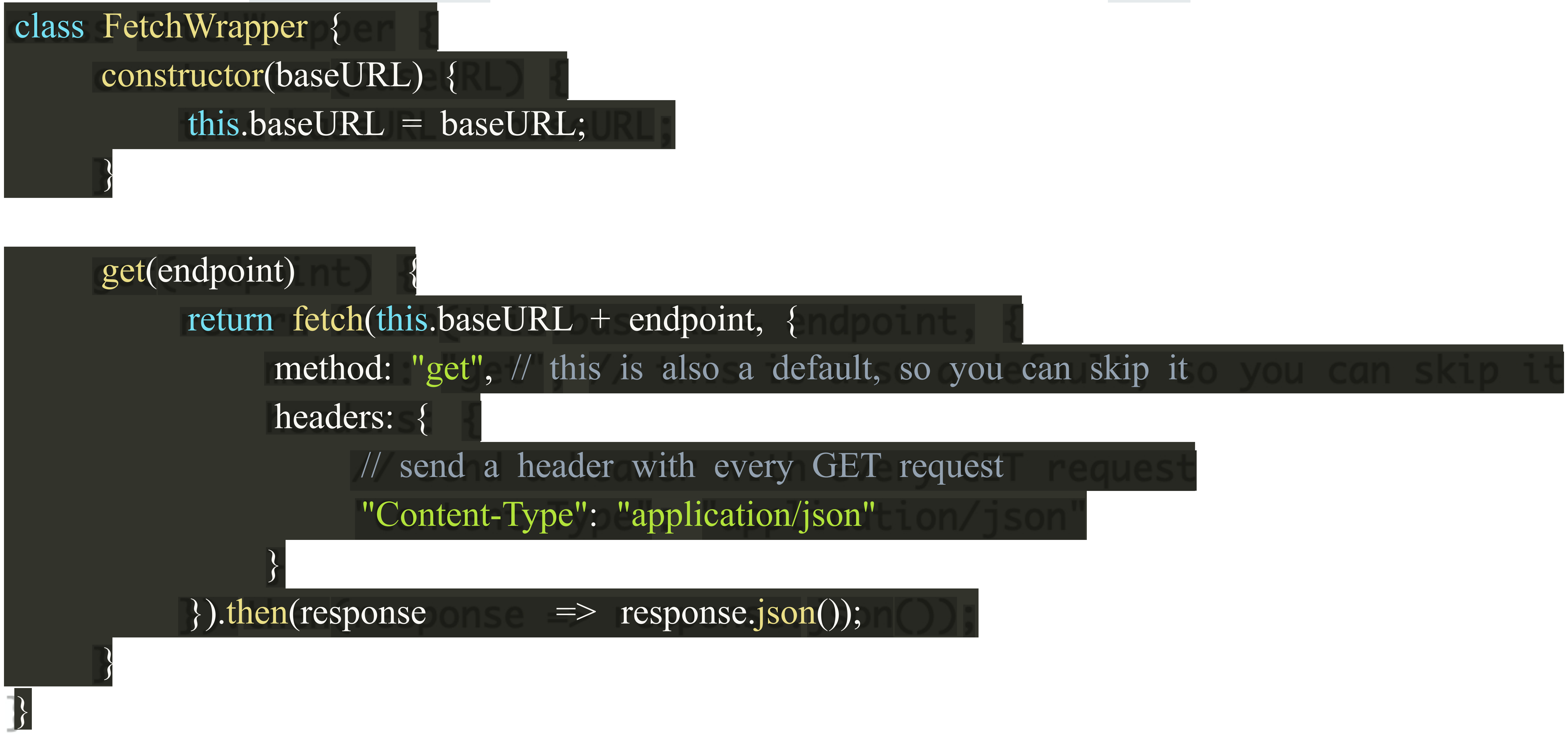
Important things to note:

- the return is **very** important. It is what allows us to call .then(data => ...) on API.get(...) . That's because it returns the result of fetch().then().
- don't forget to prefix the endpoint with the this.baseURL.
- since we know that the API is always going to return JSON, then we convert the response to JSON inside the get() method so that we don't have to do that outside the FetchWrapper class.

Customizable

This class is customizable. So, you will receive the full FetchWrapper that you can use in your own projects. This class might have to be customized to fit your needs. The benefit, however, is that you only have to customize it once and all the .get() calls will benefit from this customization .

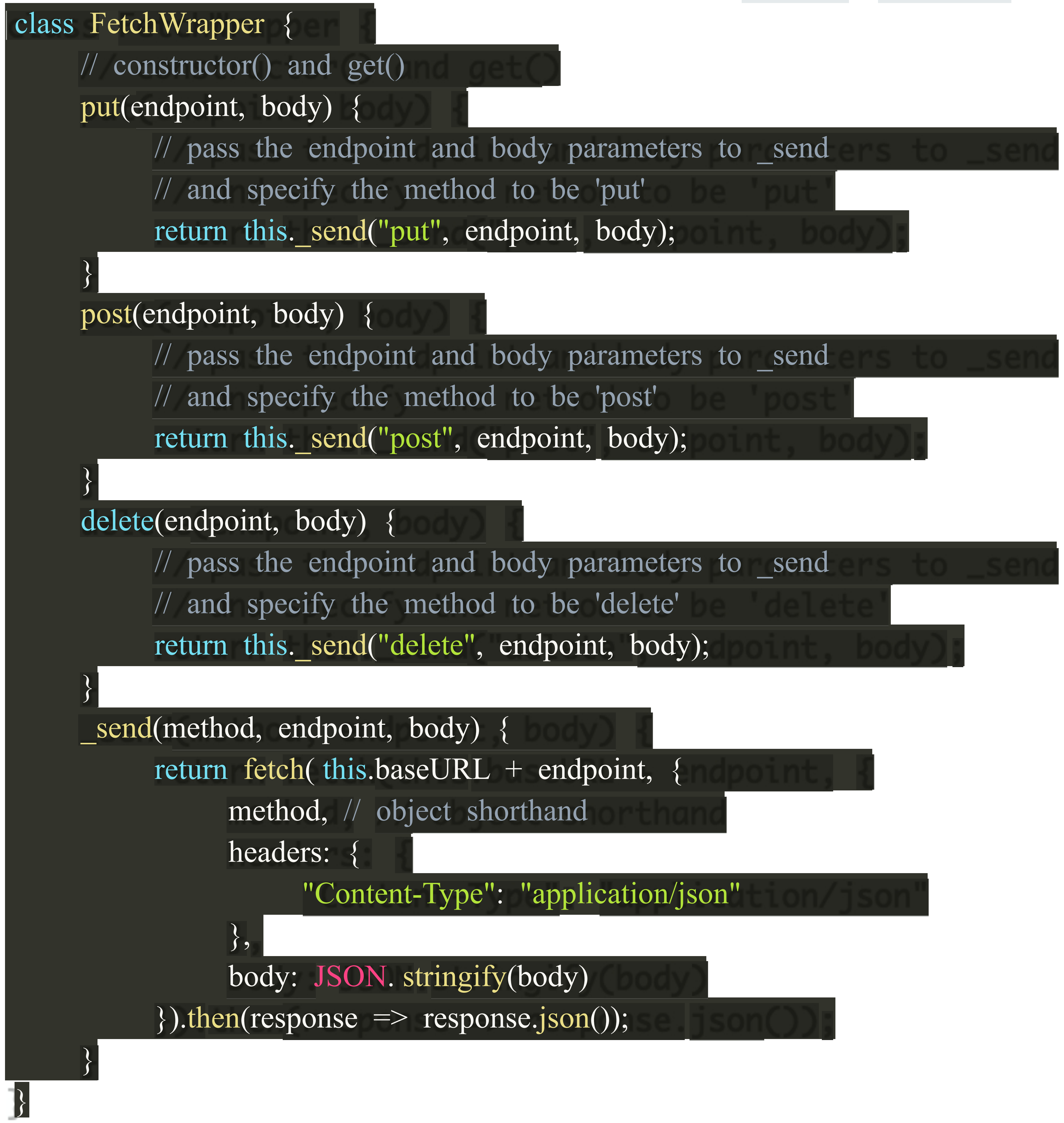
For example, your API might require a Content-Type header to be sent on every request. In that case, you can customize the FetchWrapper so that it sends the header on every GET request :



```
class FetchWrapper {
    constructor(baseURL) {
        this.baseURL = baseURL;
    }

    get(endpoint) {
        return fetch(this.baseURL + endpoint, {
            method: "get", // this is also a default, so you can skip it
            headers: {
                // send a header with every GET request
                "Content-Type": "application/json"
            }
        }).then(response => response.json());
    }
}
```

Customizable

The _send() method takes the method ("put", "post", or "delete") followed by the endpoint and the body. So, now, we can implement the put(), post(), and delete() methods that call this internal _send() method:



```
class FetchWrapper {
    // constructor() and get()
    put(endpoint, body) {
        // pass the endpoint and body parameters to _send
        // and specify the method to be 'put'
        return this._send("put", endpoint, body);
    }
    post(endpoint, body) {
        // pass the endpoint and body parameters to _send
        // and specify the method to be 'post'
        return this._send("post", endpoint, body);
    }
    delete(endpoint, body) {
        // pass the endpoint and body parameters to _send
        // and specify the method to be 'delete'
        return this._send("delete", endpoint, body);
    }
    _send(method, endpoint, body) {
        return fetch( this.baseURL + endpoint, {
            method, // object shorthand
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON. stringify(body)
        }).then(response => response.json());
    }
}
```

Practice / Classroom coding

- **fetch1.js**: Complete the `checkForNewNotifications` function such that it makes a `fetch` request to https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/notifications.json and return its result. Also, visualize that the result of fetch is a Promise. You should see `Promise` in the console.

- **fetch2.js**: Complete the `checkForNewNotifications` function such that it makes a `fetch` request to https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/notifications.json, converts the response to JSON format, and logs the `data` received to the console.

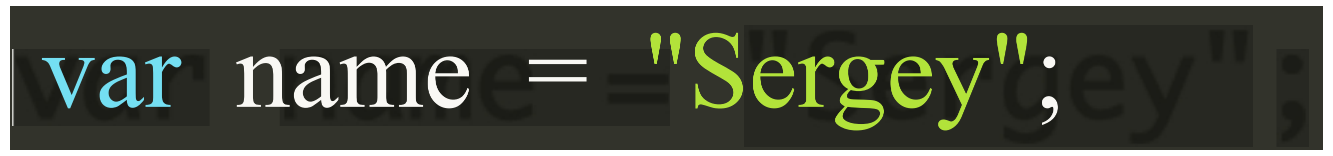
- **fetch3.js**: Implement the `FetchWrapper` class and its `get(endpoint)` instance method.

- **fetch4.js:** You can get the list of chapters by using the following endpoint https://programmingjs-90a13-default-rtdb.europe-west1.firebasedatabase.app/chapters.json. Call the `displayCompletedChapters` with only the chapters that have been completed.

Some other topics

At the beginning of the course, we mentioned how to define variables with let and const and we recommended that you avoid var. In this section, I'll explain why it should be avoided as well as how to convert old code from the Internet (for example StackOverflow) to let/const.

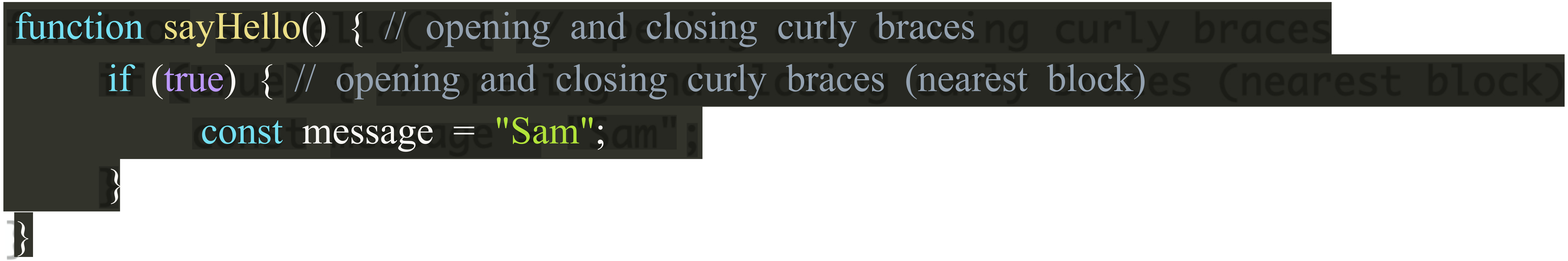
You should **NOT** use var, but here's how you define a variable with the var keyword:



```
var name = "Sergey";
```

let/const is block-scoped

When you define variables with let and const, they are block-scoped, which means they are only accessible in the nearest block. The nearest block is the nearest opening and closing curly braces you can find. For example:



```
function sayHello() { // opening and closing curly braces
    if (true) { // opening and closing curly braces (nearest block)
        const message = "Sam";
    }
}
```

The variable message is only available inside the if statement because it's block scoped. It's available inside the nearest opening and closing curly braces.
So if you try to access the variable message outside the if statement, it will not be defined.

You may not be surprised by this behavior as we've been more or less using it throughout the lessons. It's also why we recommend let and const over var because var is function scoped.

var is function scoped

When you define a variable with var, it will be scoped to the nearest function. Which means if we take the previous code and replace const with var:

```
function sayHello() { // nearest function
    // message is accessible here
    if (true) {
        var message = "Sam";
    }
    // message is also accessible here
}
```

Because the variable was defined with var, it will be accessible anywhere **inside the nearest function** . Please don't think of that as a feature, it's often regarded as a bad language design (think of it as a "mistake" in JavaScript).

Also, you might be surprised, how is message accessible **before** it was defined (the first line inside the function), this is called hoisting which I will explain in next section.

The gist of this chapter is that you should not use var and should always use let/const instead.

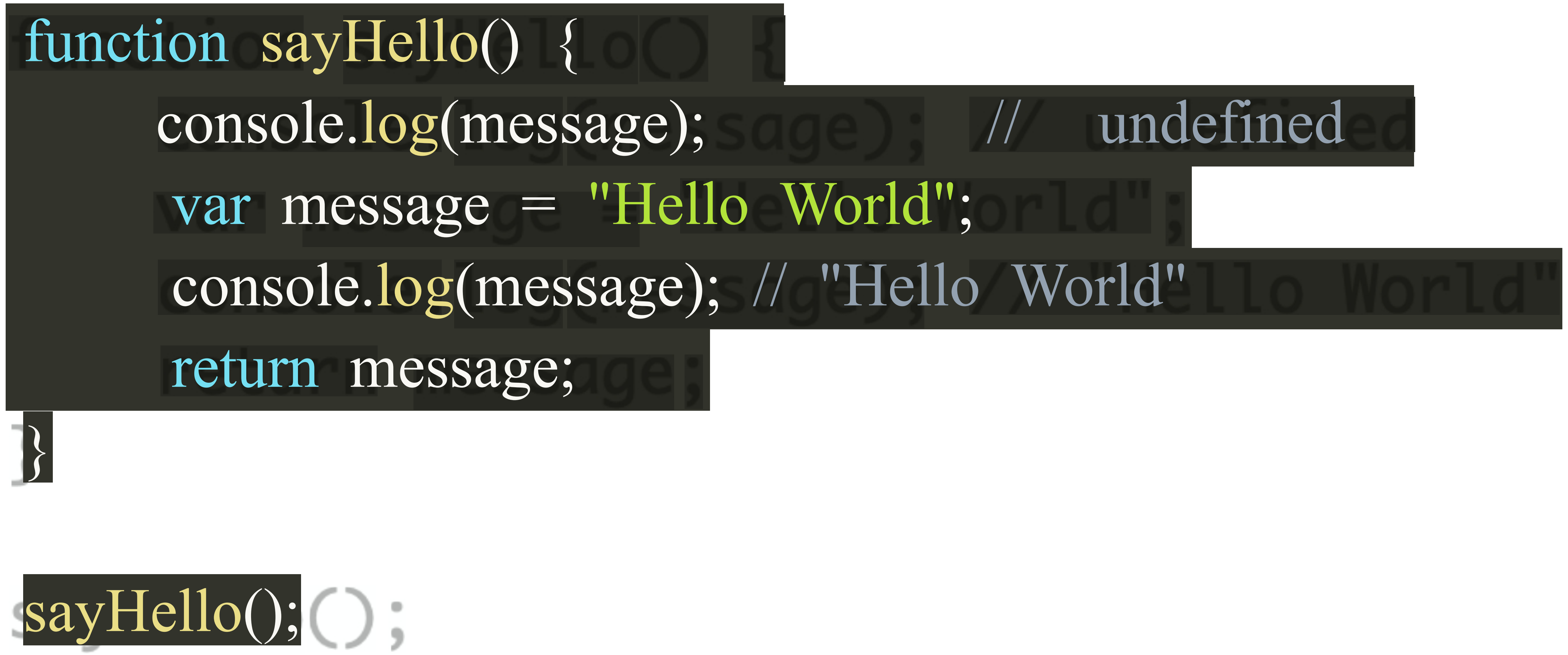
When developers were writing ES5, we only had var. But as of latest EcmaScript specs, let and const were introduced to fix the issues with var.

Hoisting and Temporal Dead Zone

Hoisting is another weird concept that you get with variables defined with var. It is also not recommended that you rely on this concept but I am explaining it in this lesson so that you can be aware that it exists.

What is hoisting ?

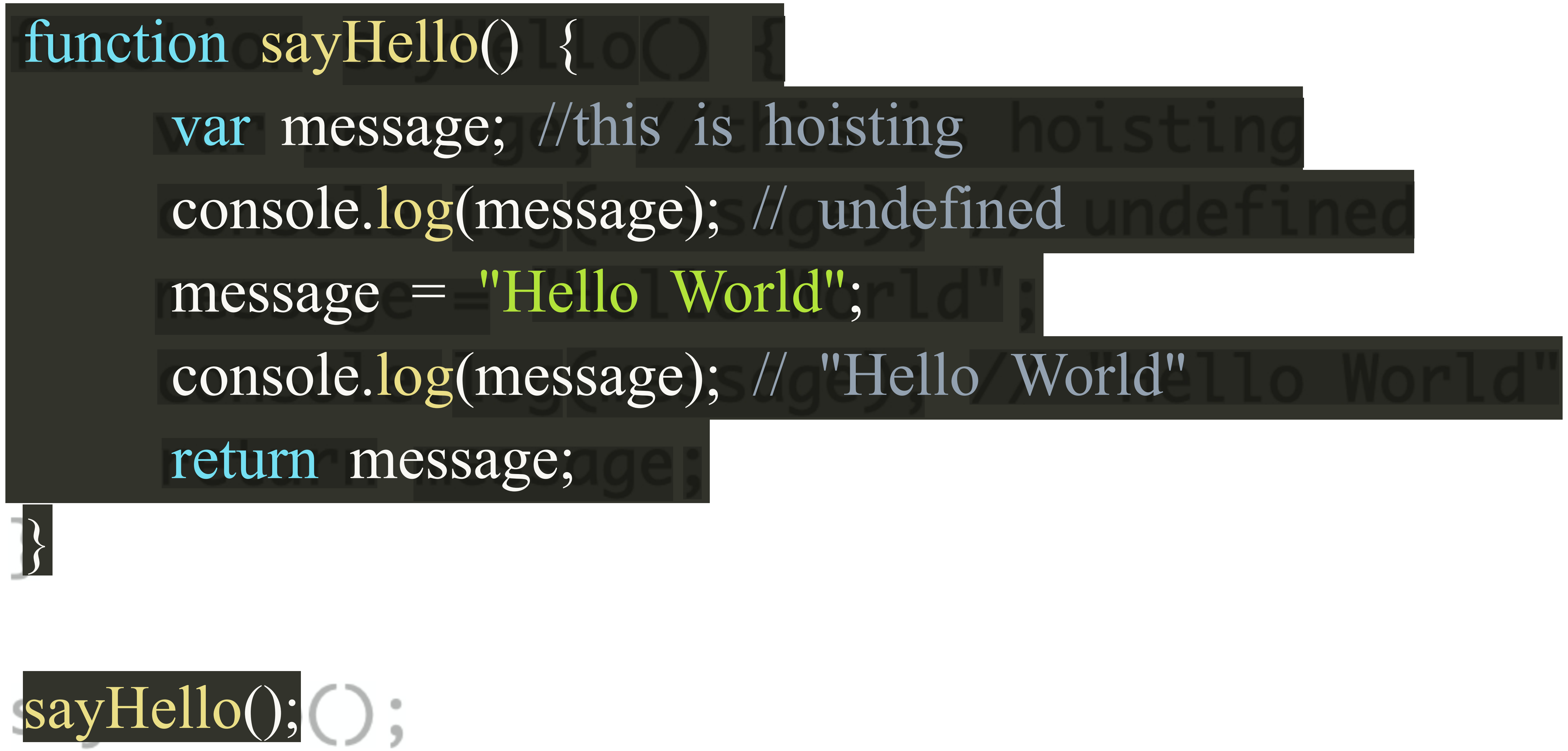
Hoisting in JavaScript is when the variables you define inside a function are moved to the top of the function. This happens every time you define a variable using var:



```
function sayHello() {
        console.log(message);  //  undefined
        var message = "Hello World";
        console.log(message); // "Hello World"
        return message;
}

sayHello();
```

The code you write above, will be transformed in the JavaScript compiler to the following:

Hoisting and Temporal Dead Zone



```
function sayHello() {
        var message; //this is hoisting
        console.log(message); // undefined
        message = "Hello World";
        console.log(message); // "Hello World"
        return message;
}

sayHello();
```

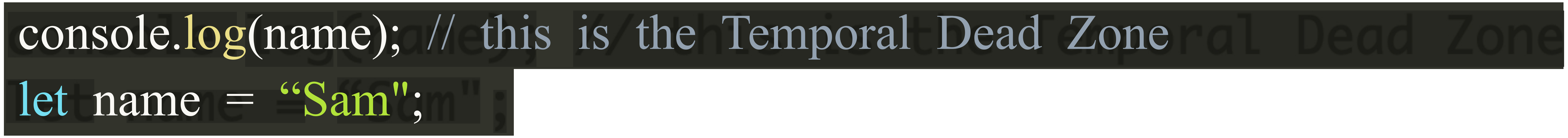
Notice how the JavaScript compiler will automatically move the variable declaration to the top of the function. Which explains why console.log(message) the first time is accessible before it was defined.

You should **not** rely on this behavior.

Note that variables defined with let and const are **NOT** hoisted.

Temporal Dead Zone

The Temporal Dead Zone is a fancy way of saying that variables defined with let and const cannot be accessed before they are initialized.



```
console.log(name); // this is the Temporal Dead Zone
let name = “Sam";
```

The code above will throw an error saying that the variable name cannot be accessed before it is declared. So any line of code that uses the variable name **before** it was defined, is called Temporal Dead Zone.

It's what you would expect from variables, this is normal behavior.

Converting old code

When you search for JavaScript questions, you might stumble upon some old and legacy code that still uses var.

In *most* cases, you can swap var with let and things will keep on working as expected.

If you prefer, you can also check which variables are not being re-assigned and change them to const.

Function hoisting

Functions defined with the function keyword are also hoisted. This allows you to call functions before they were defined, for example:

```
sayHello(); // call the function before it was defined

function sayHello() {
    console.log( "Hello World!") ;
}
```

However, with functions defined with let and const, they are **NOT** hoisted. This means the code below breaks:

```
sayHello(); // Cannot access 'sayHello' before initialization

const sayHello = function() {
    console.log( "Hello World!") ;
}
```

Even though the function keyword was used, the function sayHello was declared with const, then it is not hoisted.

This is confusing and like the rest of the course we'd like to recommend you best practices, and the best practice is to **avoid relying on the behavior of hoisting** .

It is **okay** to define functions with the function keyword, however, avoid relying on the behavior of hoisting, so always call functions **after** they were defined.

Closures

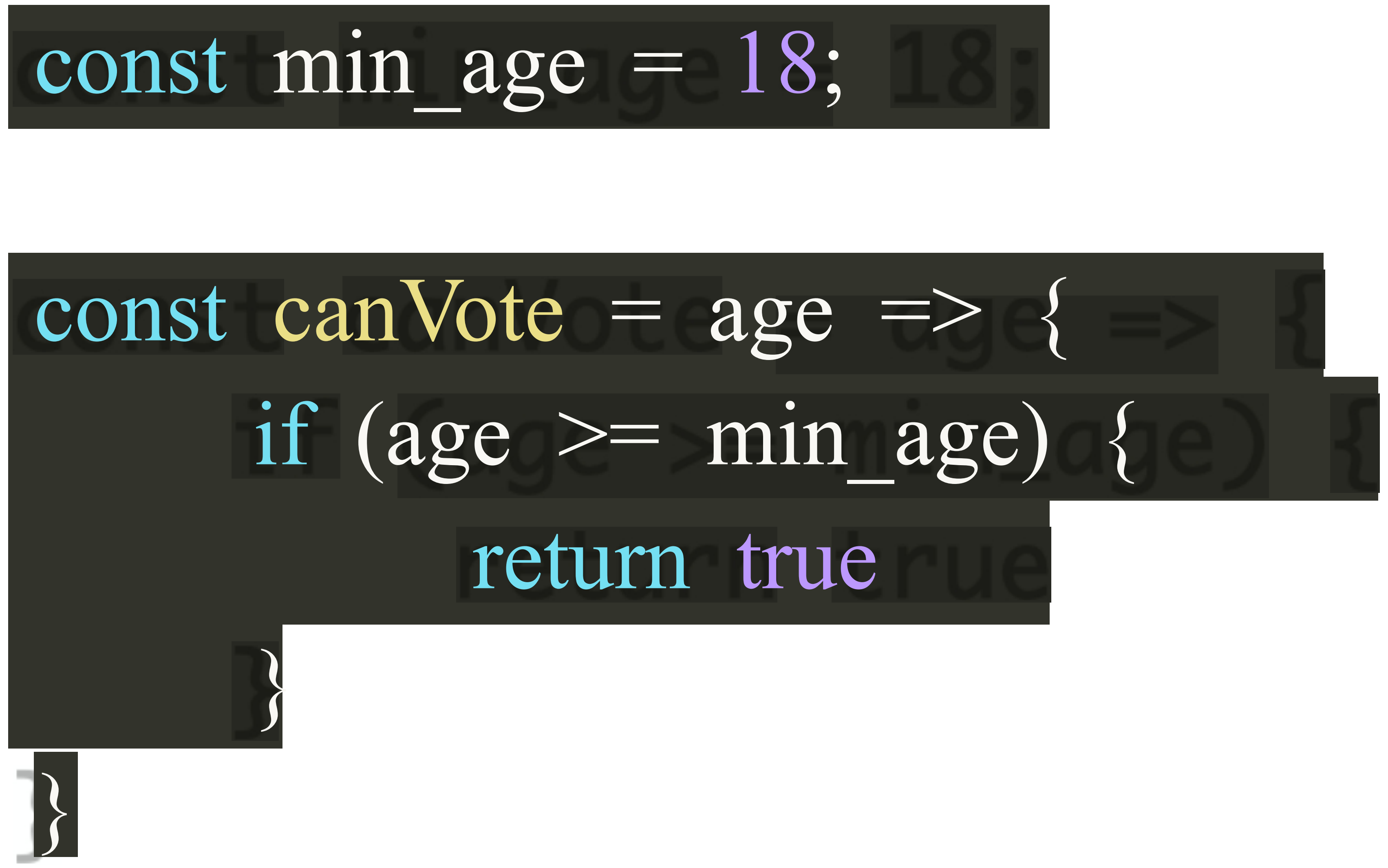
The concept of **closures** is one of those concepts that seem complicated but, in fact, it's not a complicated one. Let's break it down.

Every time you create a function in JavaScript, you create a **closure**.

A closure is where an inner function has access to the outer function's variables.

We're going to break down this definition over 3 lessons so that you understand it without being overwhelmed. The third lesson contains real-life examples of when closures can be useful.

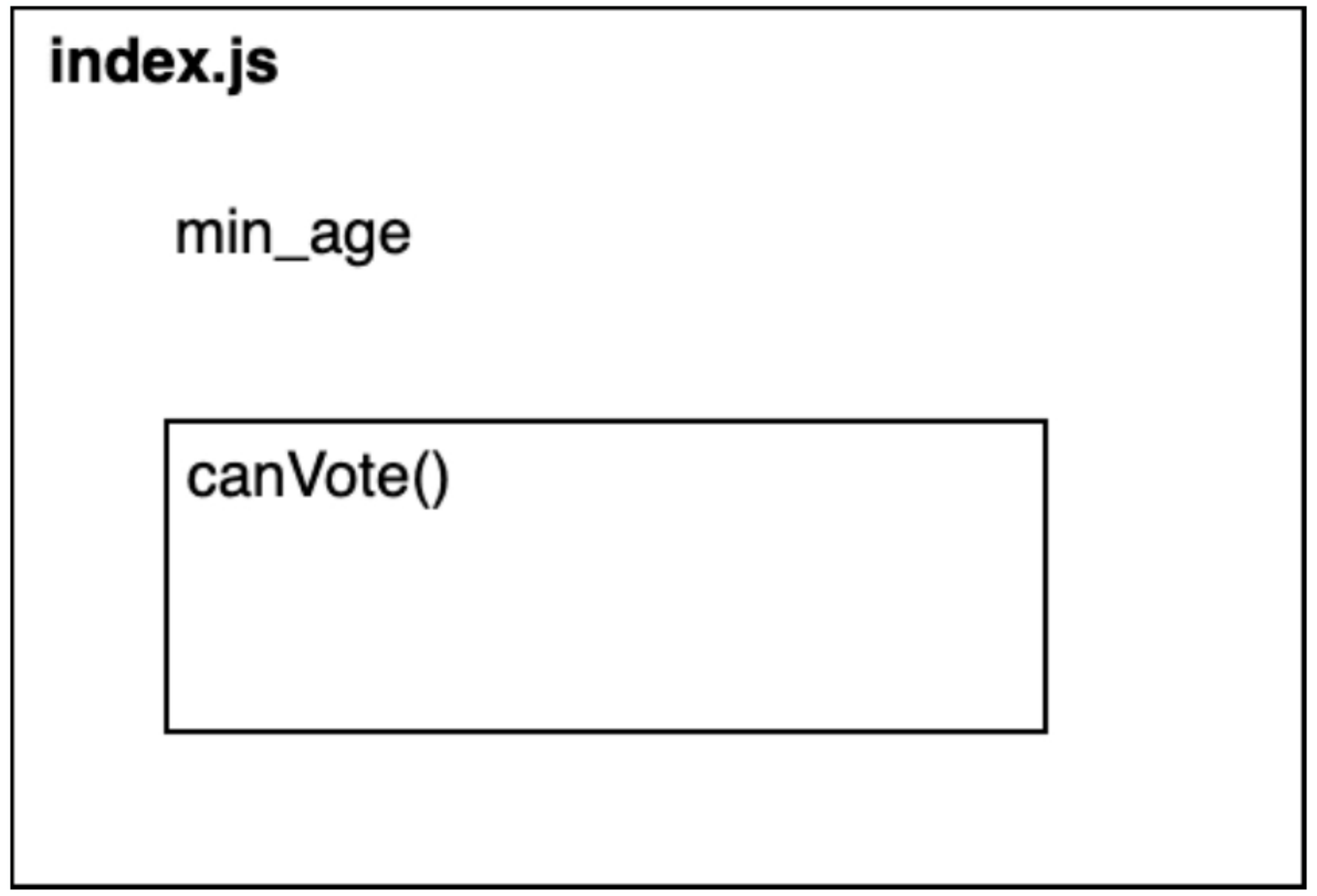
Let's say you open a new file (for example index.js) and write the following:



```
const min_age = 18;

const canVote = age => {
    if (age >= min_age) {
        return true
    }
}
```

The function canVote **is** able to access variables in the outer scope.

Closures



This is one of the cases of closures. When JavaScript encounters the variable min_age it starts looking to see if it's defined inside the current function.

That's not the case, so it goes up. Is it defined outside? Yes, then you will have access to it.

It may seem like closures is something you decide to use but in fact, closures is what you get (automatically) every time you define a function.

So, it's not a feature you decide to use but rather an explanation of how variable scoping works with regards to functions.

Practice / Classroom coding

- **misc1.js**: There is an old piece of code on StackOverflow that calculates the perimeter of a square. Start by running the code to visualize hoisting. The first console.log will output `undefined` . Then, get rid of the `var` declarations and feel free to remove unnecessary console logs if they break.

- **misc2.js**: Every time you click on the **Start game** button, it is currently calling the `startGame()` and `logAnalytics()` functions. Update the event listener such that it only calls the `startGame()` once, but keeps calling `logAnalytics()` every time .

Modules

When you link a JavaScript file to your HTML file, you do that with a script tag :

```
< script  src=" index.js"></script>
```

However, this type of script **does not support** the import/export syntax.
This means if the index.js had import or export statements, they will break with a syntax error.

If you've ever seen an error in your browser saying that the import is causing a syntax error, it's because import/export does not work in traditional scripts.

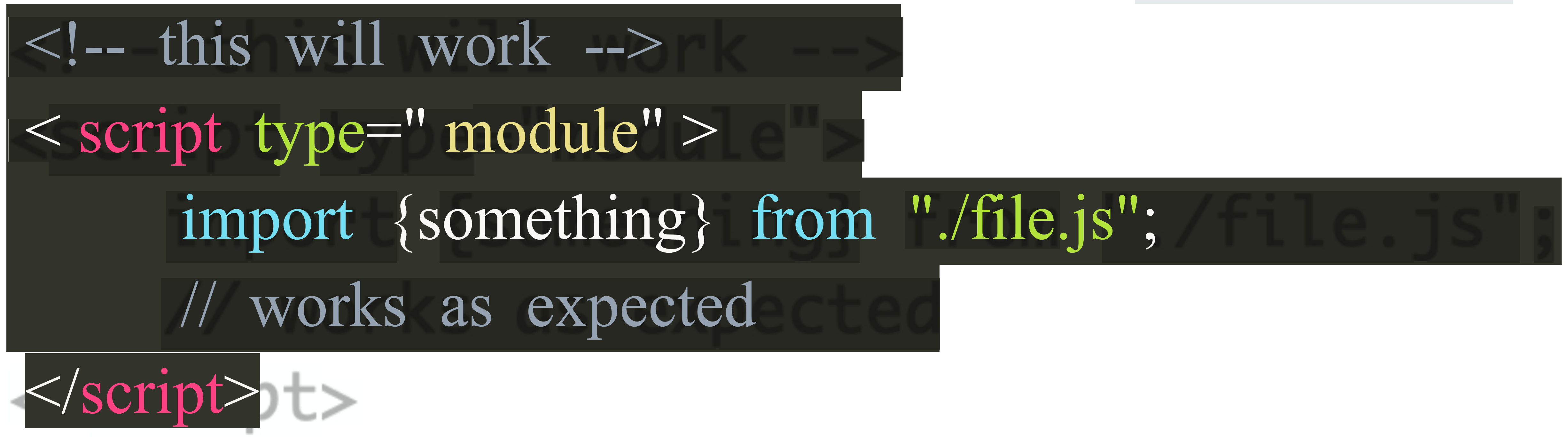
The same applies to inline scripts, such as:

```
<!--  this  will  NOT  work  -->
< script>
        import    {something}    from    "./file.js";
        // Syntax  error
</script>
```

Modules

A new kind of script has been recently introduced and **is supported** on all major browsers. It's called JavaScript Modules.

This kind of script does indeed support import/export syntax, thus the following works:



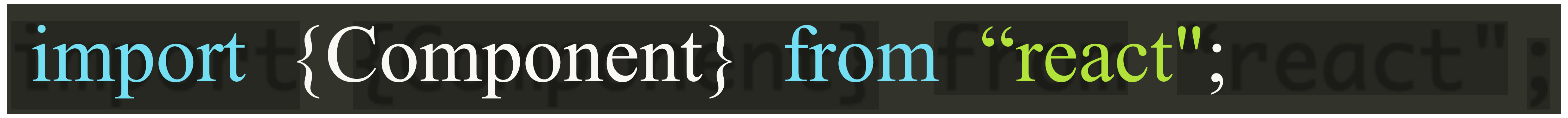
```
<!-- this will work -->
<script type="module">
    import {something} from "./file.js";
    // works as expected
</script>
```

Make sure however that you're serving from a **web server** , which means you should be on the http:// or https:// protocol, but not the file:// protocol.

Modules

Bare imports are when you import a package rather than a file name.

For example:



```
import {Component} from “react";
```

Notice how we're importing from a library called react rather than a file name such as before which had a path in the beginning (./file.js) .

At the time of writing, bare imports **do not work** in the browser, there is an ongoing proposal to bring them to the Web, in the meantime, you will have to use **module bundler** which also brings other benefits .

Modules Bundlers

Bare imports are an important part of web development, as developers often have to rely on libraries published by other developers.

Some of these libraries are helper libraries (such as **lodash**) others help you build web applications (such as **react**, **vue**, **angular**, etc.)

To make these imports work, developers use a **module bundler** such as webpack or parcel.

A module bundler (sometimes called a **build tool**) is a tool that understands your imports and is able to efficiently merge your files together.

Depending on how you configure it, it might merge all of your source files into a single final file which you can then deploy to a web server.

You can also configure it to **merge** several files together into multiple files, for example, 1 file per route (1 file for the homepage, 1 file for the settings page, etc..).

Modules Bundlers

There are several benefits of using a module bundler, let's cover the most common ones:

Dependency Resolution

These tools understand all of your imports and thus they can **resolve your dependencies** which means when you `import "lodash"` (an example library), `webpack` will resolve that import and find the actual file that needs to be imported (for example, `./node_modules/lodash/dist/lodash.min.js`).

Merge files

Your source code will have tens or even hundreds of files. A module bundler will merge these files into a single file, often called `main.js` or `app.js` which you can deploy.

You can also merge files based on the URL, which is often called: route-based code splitting.
For example, the homepage will have a homepage.js file, the settings will have a settings.js file, etc.

Running automated scripts before/after build

Most projects are configured to start by deleting the `dist/` folder so that new files can be generated there. That's an example of an automated command running before the build.
Tools that generate service workers need to run **after** a build has been completed, which can also be run using the module bundler.

Modules Bundlers

Cache busting

Cache busting is a technique that allows you to generate files with a **hash** in their name to maximize the caching capability of the browser.
For example, your `app.js` becomes: `app-9d0bc8147e2da823.js` after building with the module bundler.
Then, only when the content of `app.js` changes, the hash (`9d0bc8147e2da823`) would change.
This allows the browser to cache `app-9d0bc8147e2da823.js` for a long period of time knowing that its content won't update.
And that if its content has been updated, it will have a new URL. For example, app-3cd24fb0d6963f7d.js.

Another benefit of using a module bundler is that these tools build a **dependency graph** of all your imports, and are able to optimize the final result as they can tell which files are duplicate and which files will not be used.

Configuration

If you choose a front-end library/framework (such as Angular, Vue, React, LitElement, etc.) they will all have a module bundler already configured for you, which means you won't have to worry about configuring it yourself.

These libraries/frameworks often rely on webpack.

Even though you won't have to configure it yourself, we'd like to get you up to speed with the basics of 2 of the most common module bundlers:

1. Parcel
2. Webpack

`Parcel` is known for its ease of use as you can use it without having to configure it.
`Webpack` is known for its complicated configuration, however, it's widely used in the JavaScript community.

If you're starting your own personal project, we recommend `parcel` due to its ease of use.

Parcel guide

Parcel is a fast, zero-configuration web application bundler. https://parceljs.org

To get started with Parcel, you can follow the instructions on their website by clicking on the **getting started** button .

We're also going to provide you with our own guide explaining most of the steps:

Step 1: Create folder

Create a folder for your application and then open a terminal inside it.

On Linux/macOS you can use the following commands:

```
mkdir  your-project
cd  your-project
```

Step 2: Create package.json

Then, we need to create a `package.json file` (that will store your dependencies).

If you'd like to fill the values yourself, run `npm init` otherwise run `npm init -y` which will answer all the basic questions on your behalf.

This will create a `package.json file` inside your project.

Step 3: Install parcel locally

Let's start by installing parcel locally, the name of the package is `parcel` which will allow you to run the `parcel` command from anywhere.

```
npm  install  parcel@next  --save-dev
```

The `@next` installs version 2 of parcel which is currently in development and has reached a release candidate.

Parcel guide

Step 4: Create index.html index.js

Then we need to create 2 files: index.html and index.js, you can do so using the following command:

```
touch index.html index.js
```

Inside the index.html, add the following HTML which links the HTML file to the JavaScript file.

```
<!DOCTYPE html>
<html>
<head>
    <title>My App</title>
</head>
<body>
  <script src="./index.js"></script>
</body>
</html>
```

Step 5: Run the server

Almost ready, we still need to run the local web server which allows you to host your files on http:// rather than file://.

```
npx parcel index.html
```

You can now open **http://localhost:1234** in your browser and make changes to your HTML & JS files which will automatically update in the browser.