Hello

Twitter

@Codersinhoods

Slack

Youtube

/Codersinhoods

HELLO!

I am Sam Barker

- *Lead Instructor at Flatiron School*

samuel.e.barker@gmail.com

Coders In Hoods World

2900+

In Twitter

530+ users

In slack

20

Time zones

Change Your life

Our main goal is to help people change their lives, and make software education affordable for everyone.

JavaScript

Lesson 4

Asynchronous Javascript

=> synchronous code and blocking
=> asynchronous code
=> Callbacks
=> Promises
=> Fetch API
=> Async/Await

A few things you need to know

Many computers now have multiple cores, so can do multiple things at once. Programming languages that can support multiple threads can use multiple cores to complete multiple tasks simultaneously.

JavaScript is single-threaded. *Even with multiple cores, you could only get it to run tasks on a single thread, called the main thread.*

In the browser JavaScript runs from the top of the file to the bottom, with only one thing happening at once.

If a function relies on the result of another function, it has to wait for the other function to finish and return, and until that happens, the entire program is essentially stopped from the perspective of the user.

When an app runs and it executes an intensive chunk of code, the application can appear to be frozen. This is called blocking. The application is blocked from continuing to handle user input and perform other tasks until the app returns control of the processor.

Example

```
let x = 10
while (x > 0) {
    setTimeout(() => {
        console.log(`Number inside setTimeout: ${x}`);
    }, 2000)
    x -= 1;
}
```

One more example

```
1 const buttonEl = document.querySelector('button');
2 buttonEl.addEventListener('click', () ⇒ {
3     console.log('You clicked a button');
4
5     let pEl = document.createElement('p');
6     pEl.textContent = 'You new paragraph!';
7     document.querySelector('body').appendChild(pEl);
8 });
```

Asynchronous code

The most simple examples of async code are `**setTimeout()**`
and `**setInterval()**`

setTimeout

setTimeout() executes a particular block of code once after a specified time has elapsed. It takes the following parameters:

setTimeout

=> A function to run, or a reference to function defined elsewhere.

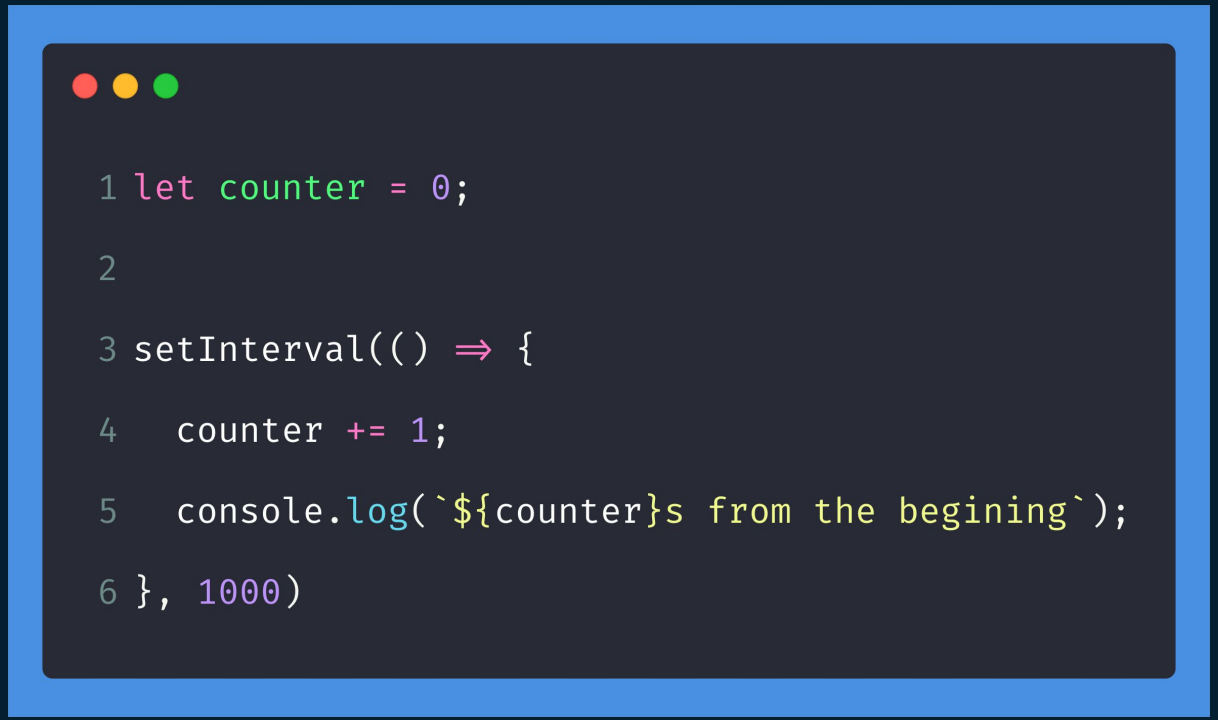
=> A number representing the time interval in milliseconds to wait before executing the code. If you specify a value of 0 (or omit this value altogether), the function will run as soon as possible (see the note below on why it runs "as soon as possible" and not "immediately"). More on why you might want to do this later.

=> Zero or more values that represent any parameters you want to pass to the function when it is run.


setTimeout(() ⇒ {
console.log('Hello, Mr. Universe!');
}, 2000)

setInterval()

This works in a very similar way to setTimeout(), except that the function you pass to it as the first parameter is executed repeatedly at no less than the number of milliseconds given by the second parameter apart, rather than at once. You can also pass any parameters required by the function being executed in as subsequent parameters of the setInterval() call.


let counter = 0;
setInterval(() ⇒ {
counter += 1;
console.log(`${counter}s from the begining`);
}, 1000)

But how to stop it?

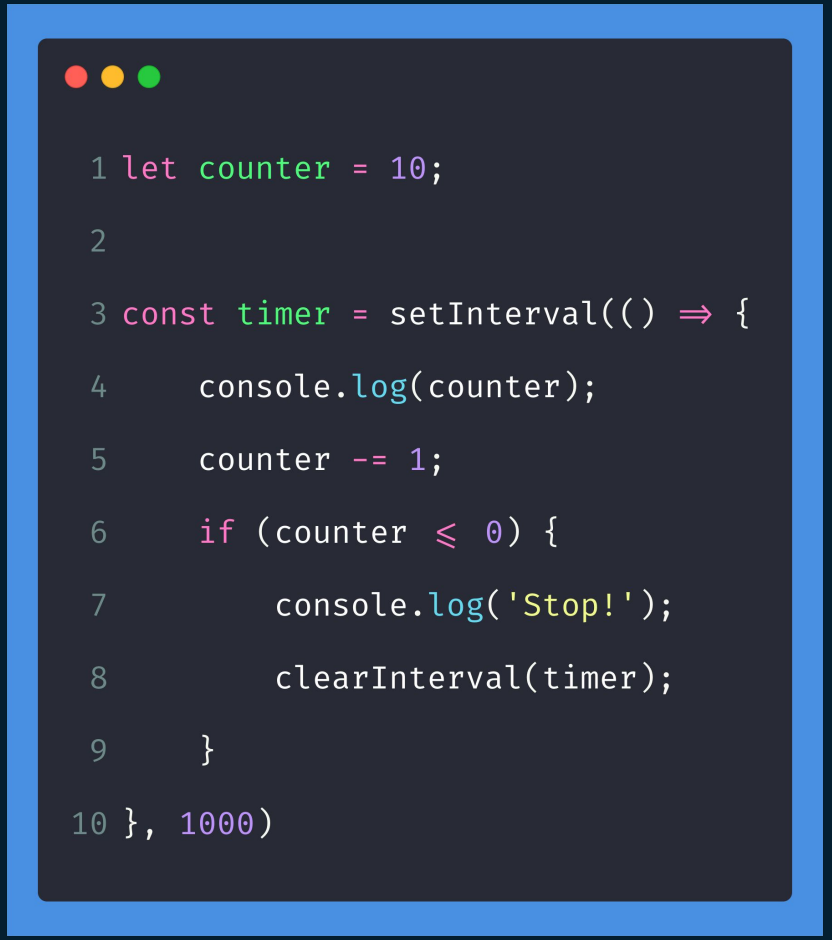
clearInterval()

```
let counter = 0;

const timer = setInterval(() => {
  counter += 1;
  console.log(`${counter}s from the begining`);
}, 1000)

clearInterval(timer);
```

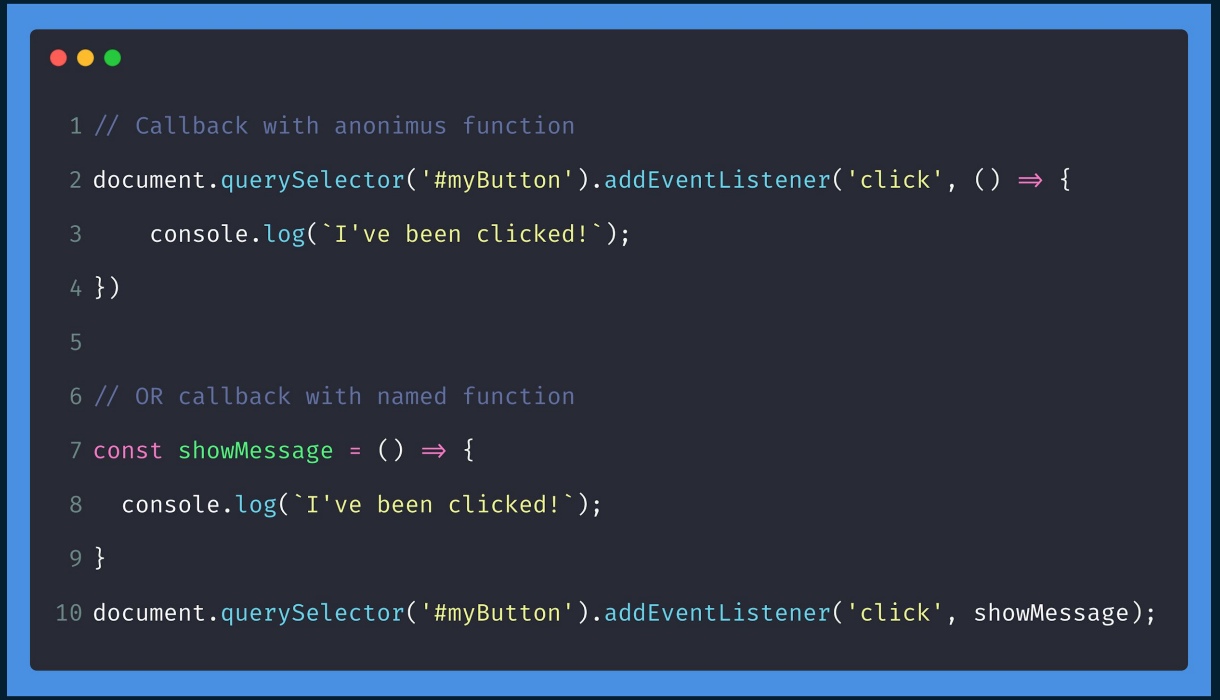
Let’s Build a stopwatch


let counter = 10;
const timer = setInterval(() ⇒ {
console.log(counter);
counter -= 1;
if (counter ⩽ 0) {
console.log('Stop!');
clearInterval(timer);
}
}, 1000)

Callbacks

Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background. When the background code finishes running, it calls the callback function to let you know the work is done.

- MDN



```
// Callback with anonimus function
document.querySelector('#myButton').addEventListener('click', () => {
    console.log(`I've been clicked!`);
})

// OR callback with named function
const showMessage = () => {
  console.log(`I've been clicked!`);
}
document.querySelector('#myButton').addEventListener('click', showMessage);
```

Promises

Promise is an object that represents an intermediate state of an operation — in effect, a promise that a result of some kind will be returned at some point in the future. You don't know when the operation will complete and the result will be returned, but when the result is available, or the promise fails, the code you provide will be executed in order to do something else with a successful result, or to gracefully handle a failure case.

Basic structure

```
yourAsyncOperation(yorParams)
    .then((result) => {
        // Do something with the result
    })
    .catch((error) => {
        // Handle error
    });
```

Example

```
1 const number = 10;
2
3 const myAsyncFunction = new Promise((resolve, reject) => {
4     setTimeout(() => {
5         (number > 5) ? resolve('success') : reject('Error!!!');
6     }, 3000);
7 });
8
9 myAsyncFunction
10     .then(message => console.log(message))
11     .catch(error => console.log(error))
```

Long chains can be annoying

```
wakeUp
  .then(commuteToWork)
  .then(turnOnYourComputer)
  .then(workTillLunch)
  .then(chillAnHour)
  .then(pretendYouAreBusy)
  .then(commute)
  .then(sleep)
  .catch(error ⇒ coughtOnPretending)
```

We will talk about a bit nicer implementation in a few minutes.

Fetch API

JavaScript can send network requests to the server and load new information whenever is needed.

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.

– MDN

*By default you are doing **GET** request.*
We use this method to request data from your source.

```
1 fetch('http://example.com/kittens.json')
2     .then(response ⇒ response.json())
3     .then(data ⇒ {
4         console.log(data)
5     })
```

Full list of HTTP methods

=> GET
=> POST
=> PUT
=> HEAD
=> DELETE
=> PATCH
=> OPTIONS
=> CONNECT
=> TRACE

Good to read

https://javascript.info/fetch

Response headers

The response headers are available in a Map-like headers object in response.headers that means that you can iterate through headers.

```
fetch('http://example.com/kittens.json')
  .then(response ⇒ {
    // get one header
    response.headers.get('Content-Type')); // application/json; charset=utf-8
    // iterate over all headers
    for (let [key, value] of response.headers) {
      console.log((`${key}: ${value}`));
    }
  })
```

Request headers

To set a request header in fetch, we can use the headers option. It has an object with outgoing headers, like this


fetch(URL, {
headers: {
Authentication: 'YOUR_KEY'
}
});

Forbidden properties

We have a list of properties that you can't change for security reasons. They are controlled by the browser.

https://fetch.spec.whatwg.org/#forbidden-header-name

POST method

We use to send data to the server create or replace resource.

```
fetch('https://kittensbay.org/kitten/create', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({name: 'Tom', age: '3'})
  });
```

Async/Await

This is special syntax to work with promises in a more comfortable fashion.

To use await your function should be async

async Function

Real life example

```
1 const getData = async () ⇒ {
2     const response = await fetch('YOUR_URL');
3     const data = response.json();
4
5     return data;
6 }
```

What's next?

~~**Lesson 0** — JavaScript basics: Part 1~~ 

~~**Lesson 1** — JavaScript basics: Part 2~~

~~**Lesson 2** — DOM manipulation~~

~~**Lesson 3** — Let's build a **Project**~~

~~**Lesson 4** — Asynchronous Javascript~~

Lesson 5 — Classes in javascript

Lesson 6 — **Final Project.**

THANKS!

Twitter

@Codersinhoods

Slack

Youtube

/Codersinhoods