11 - Asynchronous JavaScript - callbacks, promise and async-await

Any computer program is nothing but a series of tasks we require the computer to execute. In JavaScript, tasks can be classified into synchronous and asynchronous types.

11.1 - Synchronous Programming

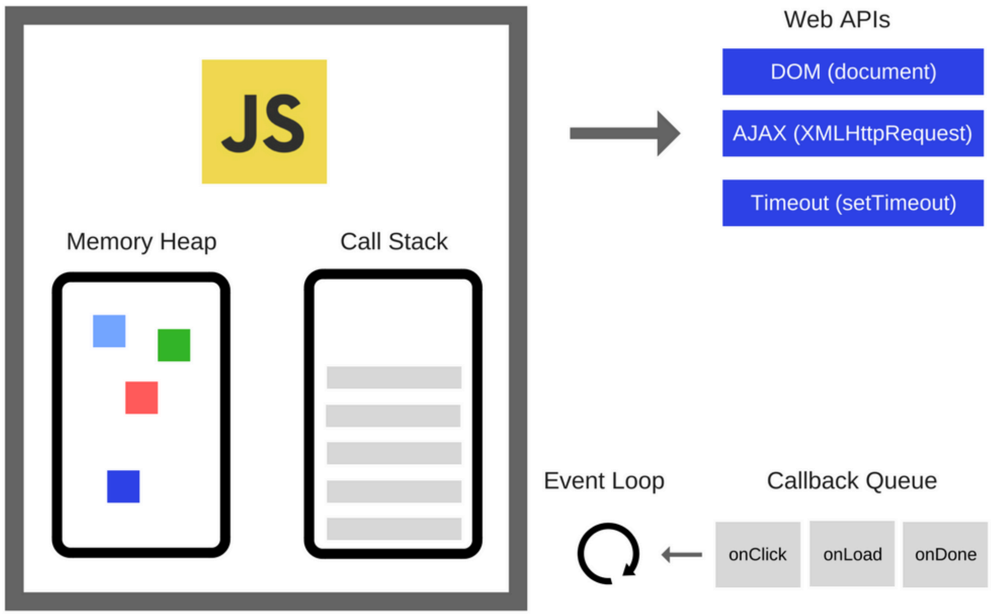
- **Definition**
  - **Synchronous** tasks are the ones that execute sequentially, one after the other, and while they're being executed nothing else is being done. At each line of the program, the browser waits for the task to finish before jumping to the next one.
  - These kinds of tasks are "blocking", because while they execute they block the execution thread, preventing it from doing anything else.

  - **Synchronous JS Example:**

```
const studentName = "Abebe";
const greeting = `Hello, my name is ${studentName}!`;
console.log(greeting);
```

    - In the above example
      1. Declares a string called studentName.
      2. Declares another string called greeting, which uses studentName.
      3. Outputs the greeting to the JavaScript console.
    - **Note** that the browser effectively steps through the program one line at a time, in the order we wrote it. At each point, the browser waits for the line to finish its work before going on to the next line. It has to do this because each line depends on the work done in the preceding lines. That makes this a **synchronous program**.

- **How does the Browser & JS engine work together to execute synchronous JS**



  - There are 5 major tools involved to make this happen
    1. Memory/JavaScript heap
    2. Call stack
    3. Web APIs
    4. Callback queue
    5. Event loop
  - Let's discuss the first **two** for now, since those are used to handle the synchronous JS

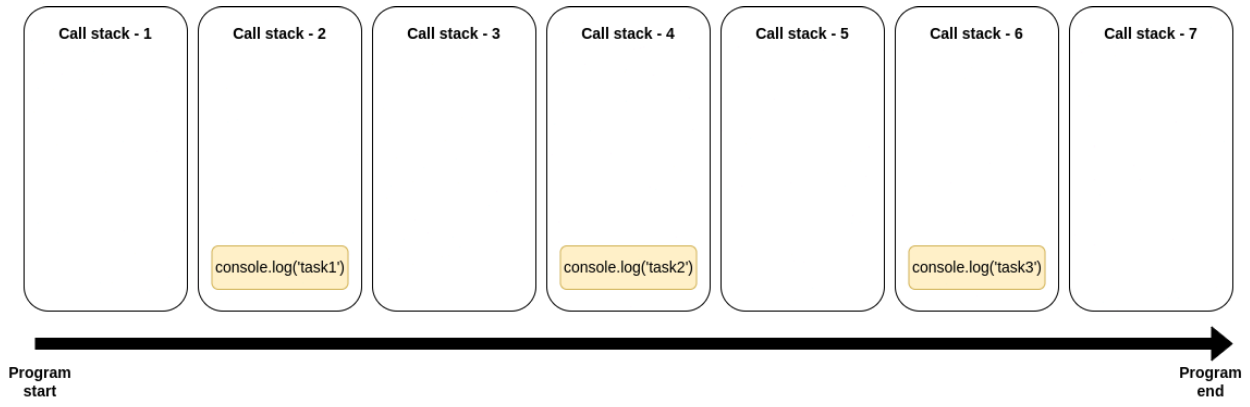
1. **Memory/JavaScript heap**
    - The JavaScript heap is where objects are stored when we define functions or variables.
2. **Call stack**
    - This is the synchronous part of the JavaScript engine component that tracks the current code being executed and lets the JavaScript engine know what code to execute next.
    - The way it works is quite simple. When a task is to be executed, it's added to the call stack. When it's finished, it's removed from the call stack. This same action is repeated for each and every task until the program is fully executed.

- **Note**: Everything that happens inside the call stack is sequential as this is the synchronous part of JavaScript. JavaScript's main thread makes sure that it takes care of everything in the stack before anything else.

- **Call stack Example 1 :**

```
console.log("task 1");
console.log("task 2");
console.log("task 3");
```

❖ **In the above example 1 the call stack would look like this:**

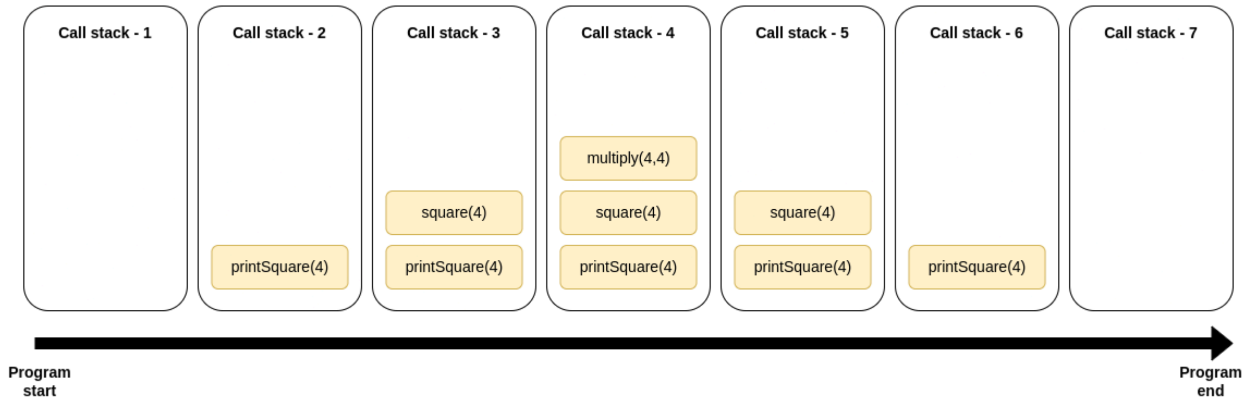


1. Call stack starts off empty at the start of the program.
2. The first task is added to the call stack and executed.
3. The first task is removed from the call stack once finished.
4. The second task is added to the call stack and executed.
5. The second task is removed from the call stack once finished.
6. The third task is added to the call stack and executed.
7. The third task is removed from the call stack once finished. End of the program.

- **Call stack Example 2 :**

```
const multiply = (a, b) => a * b;
const square = (n) => multiply(n, n);
const printSquare = (n) => console.log(square(n));
printSquare(4);
```

❖ **In the above example 2 the call stack would look like this:**



1. Call stack starts off empty at the start of the program.
2. printSquare(4) is added to the call stack and executed.
3. As printSquare(4) calls the function square(4), square(4) is added to the call stack and executed as well. Note that as the execution of printSquare(4) isn't finished yet, it's kept on the stack.
4. As square(4) calls multiply(4,4), multiply(4,4) is added to the call stack and executed as well.
5. multiply(4,4) is removed from the call stack once finished.
6. square(4) is removed from the call stack once finished.
7. printSquare(4) is removed from the call stack once finished. End of the program.

- **The problem with synchronous JS**
  - The problem with synchronous JS comes when we encounter a **long-running synchronous function.**
  - Consider the following **example**

```
console.log("Start");
function delay(seconds) {
   const start = Date.now();
   let x = true;
   while (x) {
       const current = Date.now();
       if (current - start >= seconds * 1000) {
           console.log("hi");
           x = false;
       }
   }
}
delay(5); // Delays for 5 seconds
console.log("Doing something completely unrelated to the
timer above");
```

❖ In the above example the **function delay** takes **5 seconds to execute**, and the console.logs we have on the very first and last line has nothing to do with the function. But **because of its synchronous nature the last console.log has been affected, it's forced to wait 5 seconds before printing to the console window.**

❖ This effect continues to your UI too. Imagine if you have something synchronous executing and taking 10 seconds or more. Your client **can not** use the interface for 10 seconds since it has been blocked by the long synchronous function.

❖ That's where asynchronous functions come to the rescue.

11.2 - Asynchronous Programming

- **Definition**
  - **Asynchronous programming** is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

  - Asynchronous tasks are the ones that, while they execute, they don't block the execution thread. So the program can still perform other tasks while the asynchronous task is being executed.

  - These kinds of tasks are "**non blocking**". This technique comes in handy specially for tasks that take a long time to execute, as by not blocking the execution thread the program is able to execute more efficiently.

  - Many functions provided by browsers, especially the most interesting ones, can potentially take a long time, and therefore, are asynchronous.
    - For example:
      - Making HTTP requests using fetch()
      - Accessing a user's camera or microphone using getUserMedia()
      - Asking a user to select files using showOpenFilePicker()

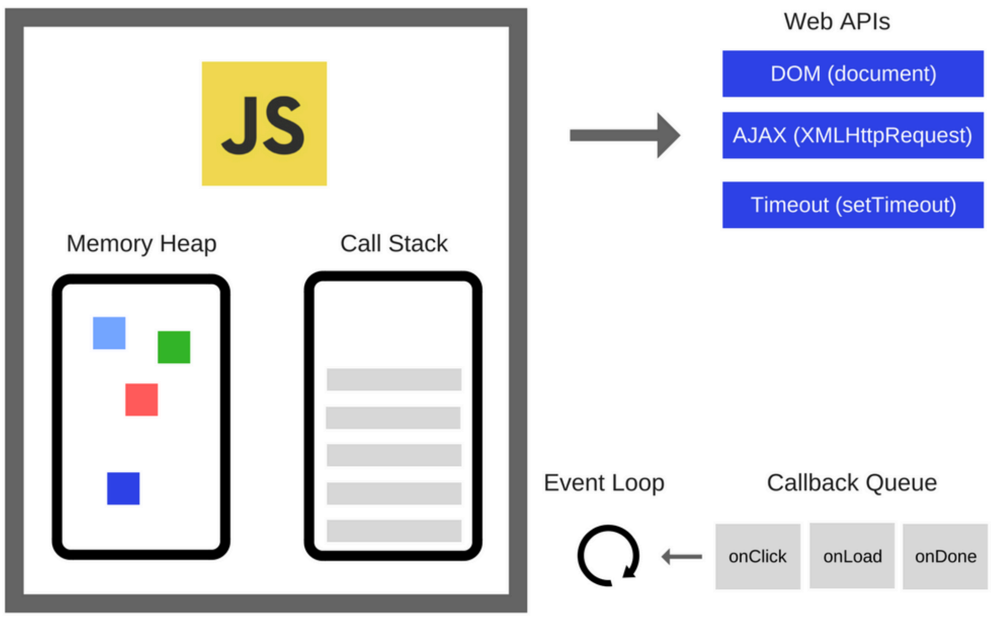
  - Even though you may not have to implement your own **asynchronous functions** very often, you are very likely to **need to use them correctly**.

- **Why Asynchronous Programming?**
  - Enhances performance by handling time-consuming operations efficiently.

  - Prevents blocking the user interface (UI) and allows concurrent operations.

  - Enables better handling of tasks like network requests, file operations, etc.

- **How does the Browser & JS engine work together to execute asynchronous JS**



  - As we discussed earlier, there are 5 major tools involved to make this happen
    1. Memory/JavaScript heap
    2. Call stack
    3. Web APIs
    4. Callback queue
    5. Event loop
  - Now Let's discuss the **last three**, since we discussed the **first two** with **synchronous** JS

3. **Web APIs**

- **Web APIs** are a set of features and functionalities that the browser uses to enable JavaScript to execute.

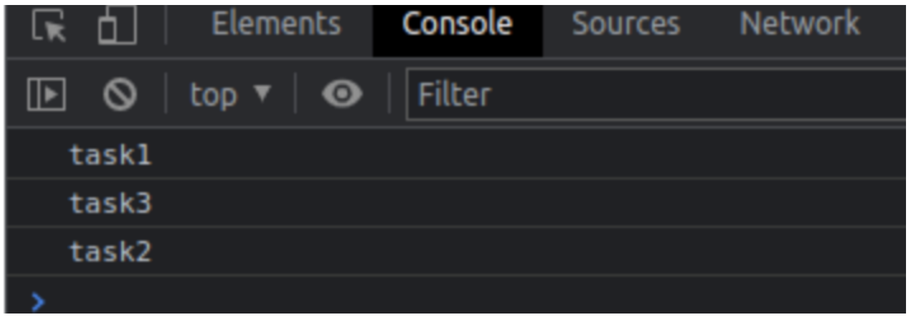
- These features include
  - **DOM manipulation**,
  - HTTP requests (**fetch** method),
  - **setTimeout** among other things, and
  - Here is a full list of web APIs from MDN website

- Think of it like a different "execution place" **rather than** the **call stack**. When the call stack detects that the task it's processing is web API-related, it asks the web API "Hey API, I need to get this done", and the web API takes care of it, allowing the call stack to continue with the next task in the stack.

- **Example using setTimeout() API**

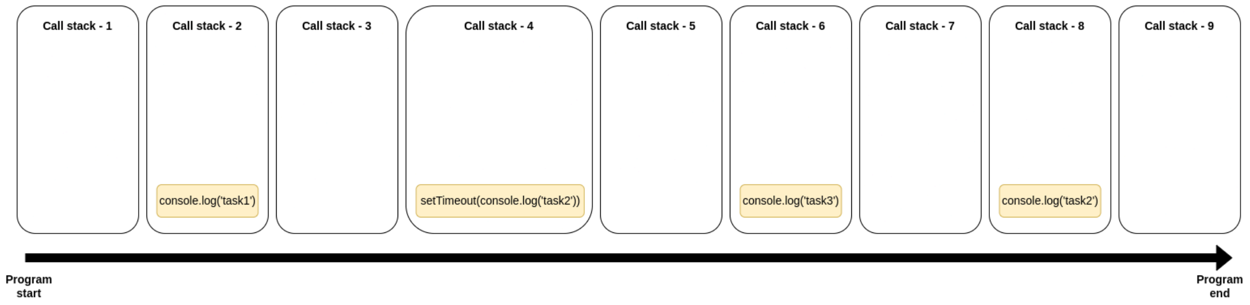
```
console.log("task1");
setTimeout(() => console.log("task2"), 0);
console.log("task3");
```

  - Here, we're logging three separate strings, and on the second one we're using setTimeout to log it after 0 milliseconds. Which should be, according to common logic, instantly. So one should expect the console to log: "task1", then "task2", and then "task3".
  - But that's not what happens:



```
task1
task3
task2
```

  - And if we had a look at our call stack during the program, It would look like this:



1. Call stack starts off empty at the start of the program.
2. console.log('task1') is added to the call stack and executed.
3. console.log('task1') is removed from the call stack once finished.
4. setTimeout(console.log('task2')) is added to the call stack, but it's **not executed.**
5. setTimeout(console.log('task2')) **disappears from the call stack, because it was sent to the web API**.

6. console.log('task3') is added to the call stack and executed.
7. console.log('task3') is removed from the call stack once finished.
8. console.log('task2') "mysteriously" hops into the call stack and is executed.
9. console.log('task2') is removed from the call stack once finished.

❖ To explain this "mysterious" reappearance of the setTimeout task, we need to introduce more components that are part of the browser runtime: the callback queue, and the event loop.

4. **Callback Queue**

- **Callback queue** is a queue that stores the tasks that the web APIs return.

- Once the JavaScript engine finishes executing the synchronous tasks on the call stack, it continues to execute the asynchronous code found on the callback queue.

- **Note**: Unlike the call stack, the callback queue follows the FIFO order (First In, First Out), meaning the callback function that first gets into the queue has the opportunity to go out first.

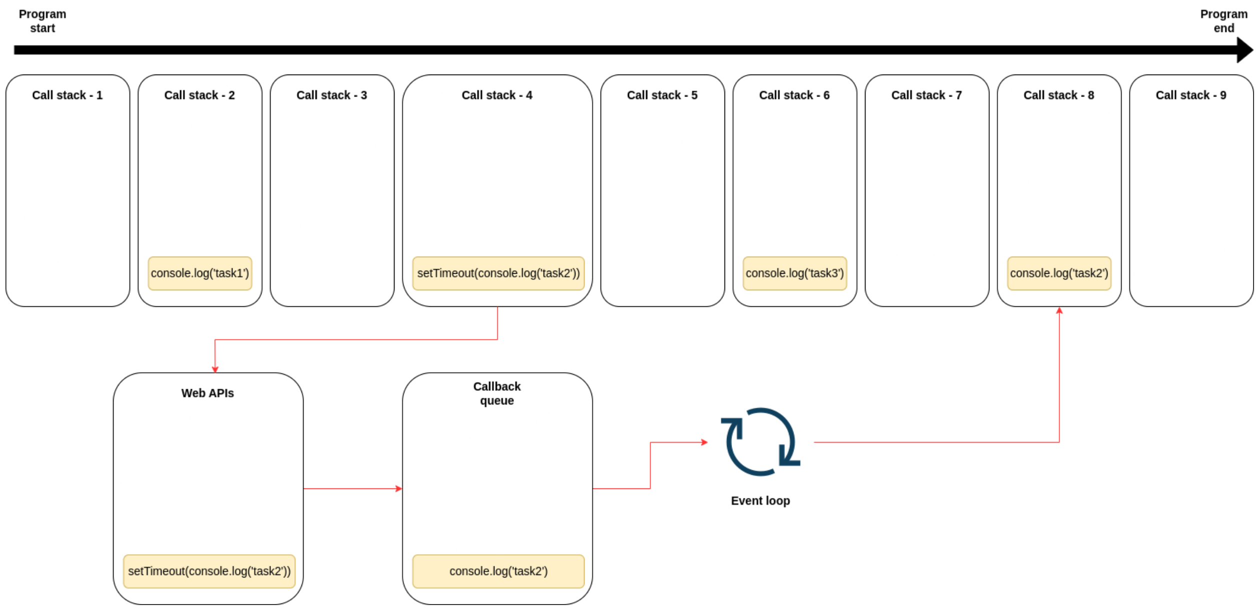
5. **Event Loop**

- **Event Loop** is a system to track tasks being executed at the external processor.

- Once the browser finishes processing the tasks under the synchronous stack (or call stack), the event loop will pass the tasks under the call back queue to the browser to be processed.

- It constantly checks two things:
  1. If the call stack is empty
  2. If there's any task present in the callback queue
  ★ If both of these conditions are met, then the task present in the callback queue is sent to the call stack to complete its execution.

❖ Now that we know about the callback queue, and the event loop, we can know what actually happened in our previous example:



★ Following the red lines, we can see that when the call stack identified that the task involved **setTimeout**, it sent it to the **web APIs** to process it.

- Once the web APIs processed the task, it inserted the callback into the **callback queue**.

- And once the **event loop detected** that the **call stack was empty** and that there was a callback present in the **callback queue**, it inserted the **callback in the call stack to complete its execution**.

- This is how **JavaScript makes asynchronism possible. Asynchronous tasks are processed by web APIs instead of the call stack, which handles only synchronous tasks.**

11.3 - How to consume/code asynchronous JS

Now that we have the theoretical foundation of how JavaScript makes asynchronism possible, let's see how all this can be implemented in code.

There are mainly three ways in which we can code asynchronism in JavaScript:

1. **callback functions,**
2. **promises, and**
3. **async-await.**

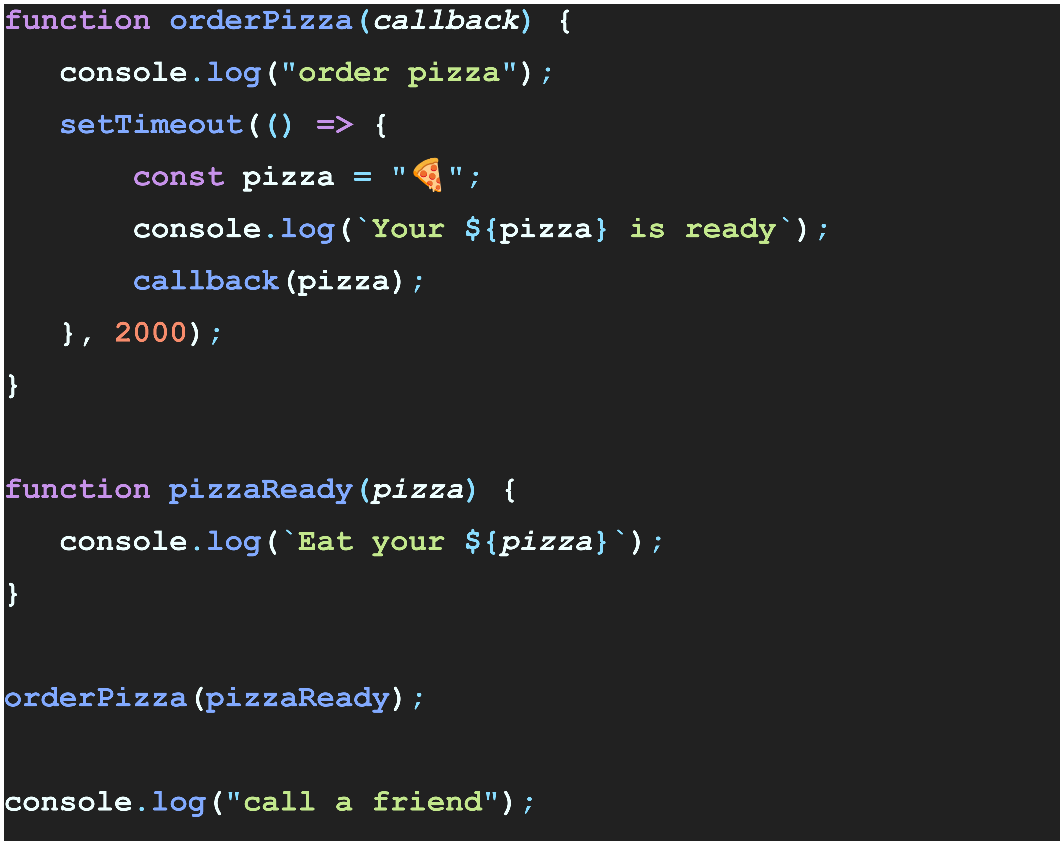
1. Callback functions

- A **callback** is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time.

- **Callbacks** are functions that are passed as arguments to other functions. The function that takes the argument is called a "Higher order function", and the function that is passed as an argument is called a "Callback".

- The difference between synchronous and asynchronous callbacks relies on the type of task that function executes (if it uses Web APIs).

- **Callback function Example :**



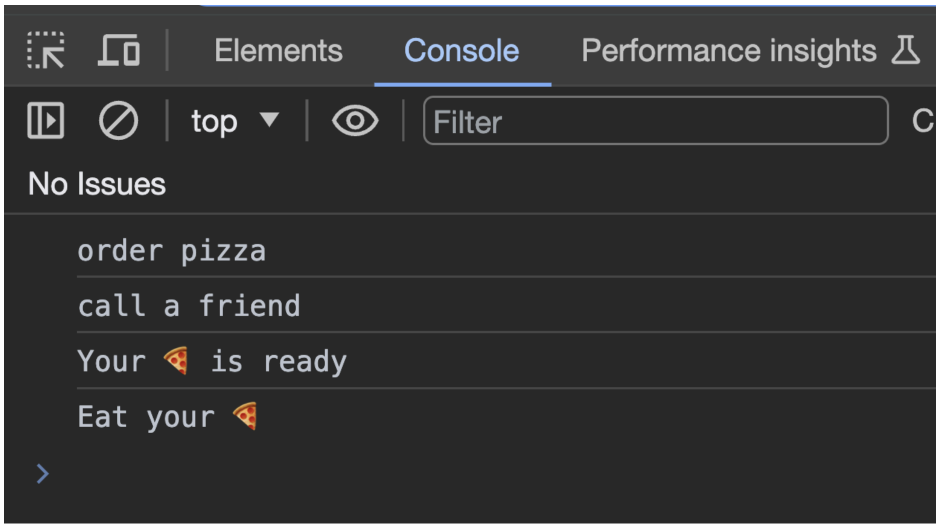
```
function orderPizza(callback) {
   console.log("order pizza");
   setTimeout(() => {
       const pizza = "🍕";
       console.log(`Your ${pizza} is ready`);
       callback(pizza);
   }, 2000);
}

function pizzaReady(pizza) {
   console.log(`Eat your ${pizza}`);
}

orderPizza(pizzaReady);

console.log("call a friend");
```

- **Example Output**



```
order pizza
call a friend
Your 🍕 is ready
Eat your 🍕
```

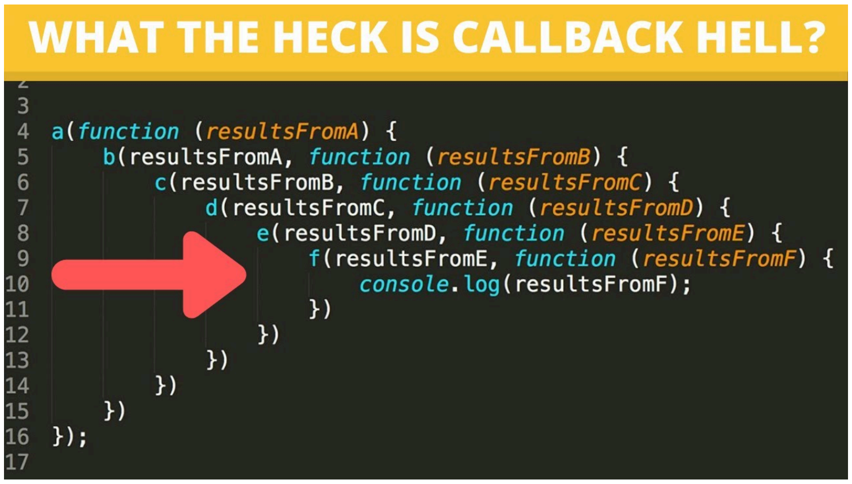
- **In the above example**,
    - “**orderPizza**” is taking a function as argument which makes it the **higher order function**
    - “**pizzaReady**” is passed as an argument which makes it the **callback function -** this function is being executed after 2 seconds, since it's inside a setTimeout method (Web APIs) of asynchronous nature.
    - It will log all synchronous tasks first, then continue to log asynchronous tasks.

- **Note:** Not all callbacks are asynchronous, what makes them asynchronous is if they are being called inside an asynchronous Web API

- **The Problem with callback function**
  - **Callback Hell/nesting a callback in a callback**
    - **Callback Hell** is a situation when callback-based code can get hard to understand when the callback itself has to call functions that accept a callback.

    - This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions.

    - Example



```
a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});
```

    - When we nest callbacks like this, it can also get very hard to handle errors: often you have to handle errors at each level of the "pyramid", instead of having error handling only once at the top level.

    - For these reasons, most modern asynchronous APIs don't use callbacks. **Instead, the foundation of asynchronous programming in JavaScript is the Promise**.

2. Promises

- **Promises** are the foundation of asynchronous programming in modern JavaScript.
- A promise is an object returned by an asynchronous function, which represents the current state of the operation.
- At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.
- **A promise** is a special kind of object in JavaScript that has 3 possible states:
    - **Pending**: It's the initial state, and it signifies that the corresponding task is yet to be resolved.
    - **Fulfilled**: Means the task has been completed successfully.
    - **Rejected**: Means the task has produced some kind of error.

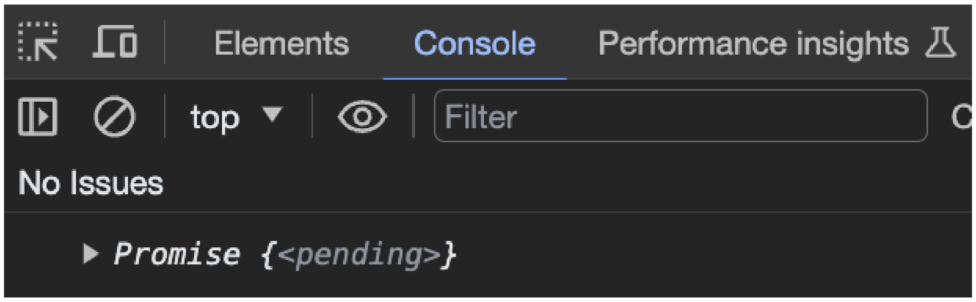
❖ To see this in practice, we'll use a realistic case in which we fetch some data from an API endpoint and log that data in our console. We'll use the fetch API provided by browsers and a public API that returns Chuck Norris jokes.

- **Using the fetch() API**
    - **fetch() API -** is an asynchronous Web API that is used to make HTTP requests to a server and respond with a response data.

Example :

```
console.log(fetch("https://randomuser.me/api"));
```

OutPut :



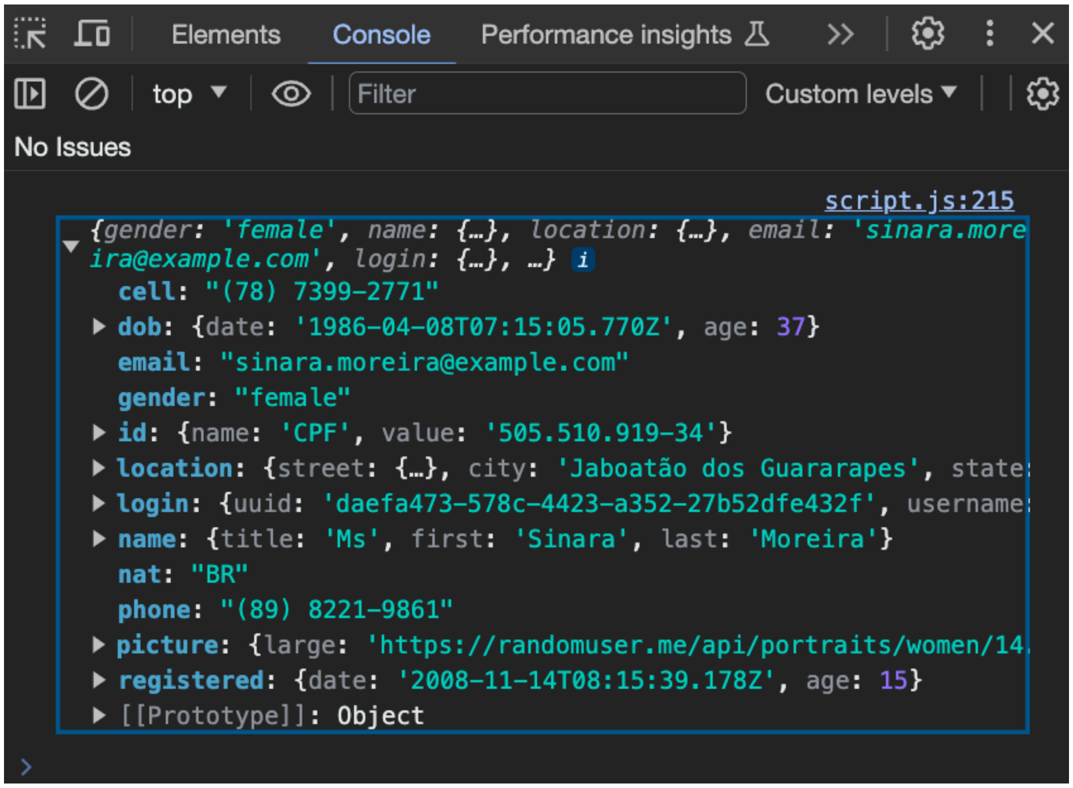
```
▶ Promise {<pending>}
```

- We see we get a promise with a pending state
    - We can **chain** other operations to it to be followed once it becomes **fulfilled**, using the “**.then()**” method

- **Chaining promises (.then( ) method)**
  - Using the **.then** method we can perform additional operations once the promise resolves/completes/fulfilled.
  - It can be applied on any promise returning function
  - **Example : apply ".then" method to be executed once the promise object is fulfilled.**

```
fetch("https://randomuser.me/api")
    .then((res) => res.json())
    .then((data) => console.log(data.results[0]));te
```

  - Output:



```
{gender: 'female', name: {…}, location: {…}, email: 'sinara.more
ira@example.com', login: {…}, …}
   cell: "(78) 7399-2771"
 ▶ dob: {date: '1986-04-08T07:15:05.770Z', age: 37}
   email: "sinara.moreira@example.com"
   gender: "female"
 ▶ id: {name: 'CPF', value: '505.510.919-34'}
 ▶ location: {street: {…}, city: 'Jaboatão dos Guararapes', state
 ▶ login: {uuid: 'daefa473-578c-4423-a352-27b52dfe432f', username
 ▶ name: {title: 'Ms', first: 'Sinara', last: 'Moreira'}
   nat: "BR"
   phone: "(89) 8221-9861"
 ▶ picture: {large: 'https://randomuser.me/api/portraits/women/14
 ▶ registered: {date: '2008-11-14T08:15:39.178Z', age: 15}
 ▶ [[Prototype]]: Object
```

  - In the above example,
    - We have seen that the fetch method returns a promise,
    - Now we are asking it to apply the .json method to its result once the promise is fulfilled.
    - But that's not the only thing we did, we have a second .then method applied, that is because the ".json( )" method itself returns a promise.
    - Finally, we are logging the output to the console window.

- **Returning a Promise in the .then() method:** When you return a value in the then() method, you are basically asking the then() method to return a new Promise that immediately resolves to the return value. In short, the following 2 example code give the same result. Please take a look at the first .then() code for both examples. Both of the first .then() methods return a new Promise, but the syntax is different.

  Example of returning value in the .then() method: Returns a new Promise

```
  let myPromise = new Promise(function (resolve, reject) {
  setTimeout(function () {
      resolve(10);
  }, 3000);
 });
myPromise.then(function (resultFromPromise) {
  return resultFromPromise * 3;
})
.then(function (resultFromPromise) {
  console.log(resultFromPromise / 2);
});
```

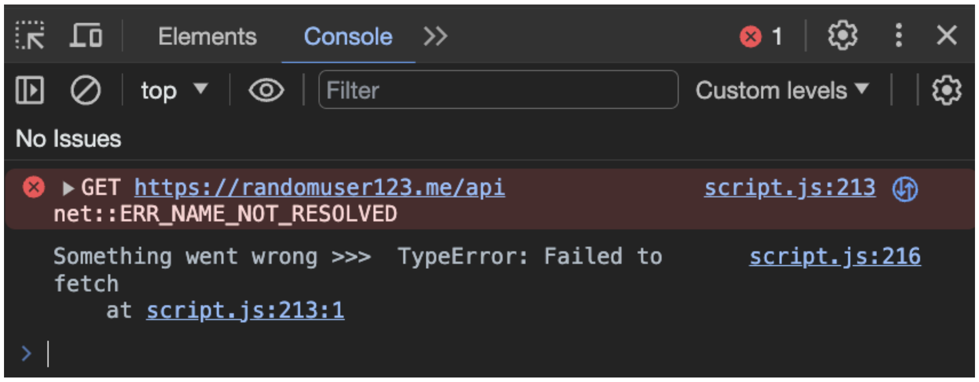
- **Catching errors (.catch( ) method)**
  - The fetch() API can throw an error for many reasons (for example, because there was no network connectivity or the URL was malformed in some way) and we are throwing an error ourselves if the server returned an error.
  - To support error handling, Promise objects provide a **catch()** method.
  - **This is a lot like then()**: you call it and pass in a handler function. However, while the handler passed to then() is called when the asynchronous operation succeeds, **the handler passed to catch() is called when the asynchronous operation fails/rejected**.
  - If you **add catch() to the end** of a promise chain, then it will be called when any of the asynchronous function calls fail. So you can implement an operation as

several consecutive asynchronous function calls, and have a single place to handle all errors.

- **Example : apply “.catch” method to be executed once the promise object is Rejected.**

```
fetch("https://randomuser123.me/api")
   .then((res) => res.json())
   .then((data) => console.log(data.results[0]))
   .catch((err) => console.log("Something went wrong
>>> ", err));
```

  - Output:



```
▶GET https://randomuser123.me/api                     script.js:213
net::ERR_NAME_NOT_RESOLVED

Something went wrong >>>  TypeError: Failed to        script.js:216
fetch
    at script.js:213:1
>
```

  - In the above example
    - We are mimicking a network call error by messing up its request URL address
    - Doing so, it caused the fetch promise object to return a rejection.
    - Finally, our “.catch()” method handled the rejection gracefully.

- **Note1:** If the callback in .catch() returns a value, .catch() will return a new promise with resolved status: If the handler function in catch() returns a value, it means that .catch() is returning a new Promise with a resolved value. Please note that this new Promise is resolved and has the returned value as its value.

- Note: Promises created when a .cathc() method returns a value will never be handled by .cathc(), but by .then(). Example:

```
  let myPromise = new Promise(function (resolve, 
reject) {
   setTimeout(function () {
       reject(10);
   }, 3000);

   });
  myPromise.catch(function (result) {
   return result * 2;
// code below is same as the returned value above  //
   return new Promise(function (resolve, reject) {
       // resolve(result * 2); //
   });
  })
   .catch(function (returnedReulst) {
       console.log(returnedReulst); // prints nothing
   })
   .then(function (returnedReulst) {
       console.log(returnedReulst); // Prints 20
   });
```

▪ **Note 2:** If the callback in .catch() throws an error, .catch() will return a new promise with reject(): If the handler function in catch() throws an error, it means that .catch() is returning a new Promise with reject(). If reject() is the status of a promise, it can only by .catch(). Example:

```
let myPromise = new Promise(function (resolve, reject) {
  setTimeout(function () {
      reject(10);
  }, 3000);
});
myPromise
  .catch(function (result) {
      throw new Error(result * 3);
   // new Promise below is same as the above thrown  value
      // return new Promise(function (resolve, reject) {
      // reject(result * 3);

 // });
  }).then(function (result) {
      console.log(result); //nothing prints here
  })
  .catch(function (result) {
      console.log(result); // prints 30
  });
```

- **Promise terminology**
  - As discussed earlier, a promise can be in one of three states:
    - **pending**: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to fetch(), and the request is still being made.
    - **fulfilled**: the asynchronous function has succeeded. When a promise is fulfilled, its then() handler is called.
    - **rejected**: the asynchronous function has failed. When a promise is rejected, its catch() handler is called.

- **Note** that what "succeeded" or "failed" means here is up to the API in question: for example, fetch() considers a request successful if the server returned an error like 404 Not Found, but not if a network error prevented the request being sent.

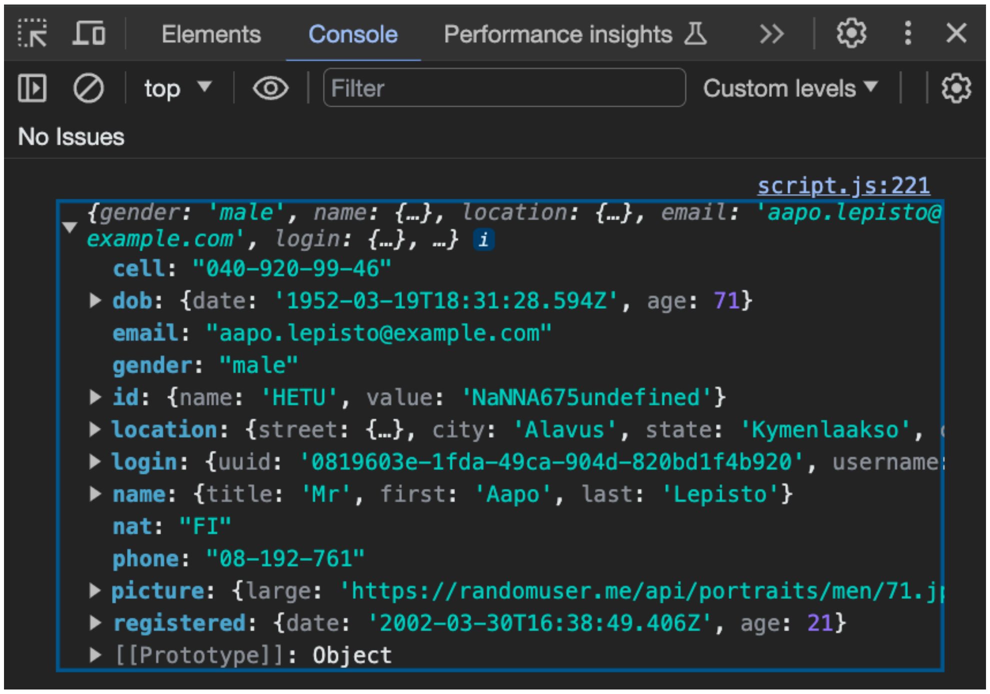
3. async and await

- **Async-await** is the latest way of dealing with asynchronism provided by JavaScript.
- The async keyword gives you a simpler way to work with asynchronous promise-based code.
- Adding async at the start of a function makes it an async function
- Inside an async function, you can use the await keyword before a call to a function that returns a promise. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown.
- This enables you to write code that uses asynchronous functions but looks like synchronous code.
- To catch the error or rejection we can make use of try/catch blocks
- **Example** : handling fetch method using async-await

```
async function logData() {
   try {
let response = await fetch("https://randomuser.me/api");
       let data = await response.json();
       const user = data.results[0];
       console.log(user)
   } catch (error) {
       console.log(error);
   }
}

logData();
```

- Output:



```
▼ {gender: 'male', name: {…}, location: {…}, email: 'aapo.lepisto@
  example.com', login: {…}, …} i
    cell: "040-920-99-46"
  ▶ dob: {date: '1952-03-19T18:31:28.594Z', age: 71}
    email: "aapo.lepisto@example.com"
    gender: "male"
  ▶ id: {name: 'HETU', value: 'NaNNA675undefined'}
  ▶ location: {street: {…}, city: 'Alavus', state: 'Kymenlaakso', 
  ▶ login: {uuid: '0819603e-1fda-49ca-904d-820bd1f4b920', username
  ▶ name: {title: 'Mr', first: 'Aapo', last: 'Lepisto'}
    nat: "FI"
    phone: "08-192-761"
  ▶ picture: {large: 'https://randomuser.me/api/portraits/men/71.j
  ▶ registered: {date: '2002-03-30T16:38:49.406Z', age: 21}
  ▶ [[Prototype]]: Object
```

  - In the above example,
    - We were able to wait for the promise to finish / complete using the await keyword and perform any additional operations we wanted afterwards.
    - We handled any possible rejections from the promise using the "try catch" block.

11.4 - How to implement our own promise-based API (Promise Constructor)

- **Promise() constructor function**
  - The promise constructor function is a constructor function that exists in JS.
  - It lets us create a promise object.
  - It is initiated like any other constructor function using the "new" keyword followed by the "Promise" constructor function.
  - The Promise() constructor takes a single function as an argument. We'll call this function the **executor**.

- When you create a new promise you supply the implementation of the executor.

- **The executor function**
    - Takes two arguments, which are both also functions, and which are conventionally called **resolve** and **reject**.
    - In your executor implementation, you call the underlying asynchronous function.
        - If the asynchronous function **succeeds**, you call **resolve**, and
        - If it **fails**, you call **reject**.
    - If the executor function **throws an error, reject is called automatically**. You can pass a single parameter of any type into resolve and reject.

- **Example :**
    - **Let's create our own promise based API called alarm**

```
function alarm(person, delay) {
   return new Promise((resolve, reject) => {
       if (delay < 0) {
           reject("Alarm delay can not be negative");
       }
       setTimeout(() => {
           resolve(`Wake up, ${person}!`);
       }, delay);
   });
}
```

    - **Consume/use the alarm API using .then( ) method**

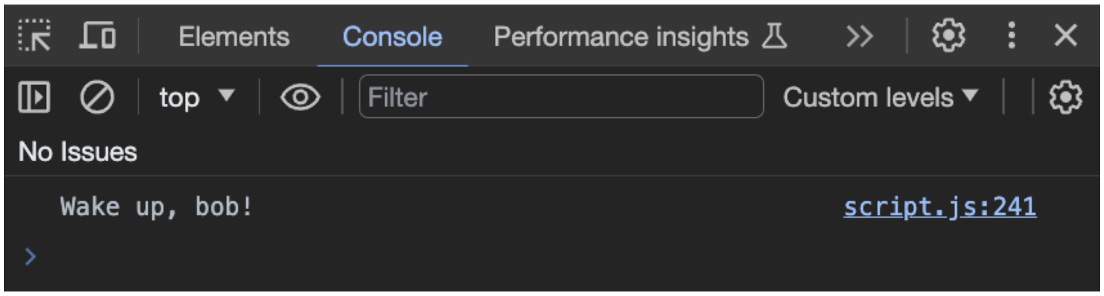
```
alarm("bob", 2000)
```

```
    .then((data) => console.log(data))
    .catch((err) => console.log(err));
```

- **Consume/use the alarm API using async-await**

```
async function getUp() {
    try {
        let data = await alarm("bob", 2000)
        console.log(data)
    } catch (error) {
        console.log(error)
    }
}
getUp()
```

- **Output:**



```
Wake up, bob!                                   script.js:241
```

  - In the above example,
    - We are consuming the alarm API using both .then() & async-await methods, we get exactly the same result in both cases.

Here is a nice link to practice your understanding of Promises

- https://www.codingame.com/playgrounds/347/javascript-promises-masteringthe-asynchronous/its-quiz-time

Here is a nice link to practice and visualize how JavaScript's call stack/event loop/callback queue interact with each other.

- http://latentflip.com/loupe/