

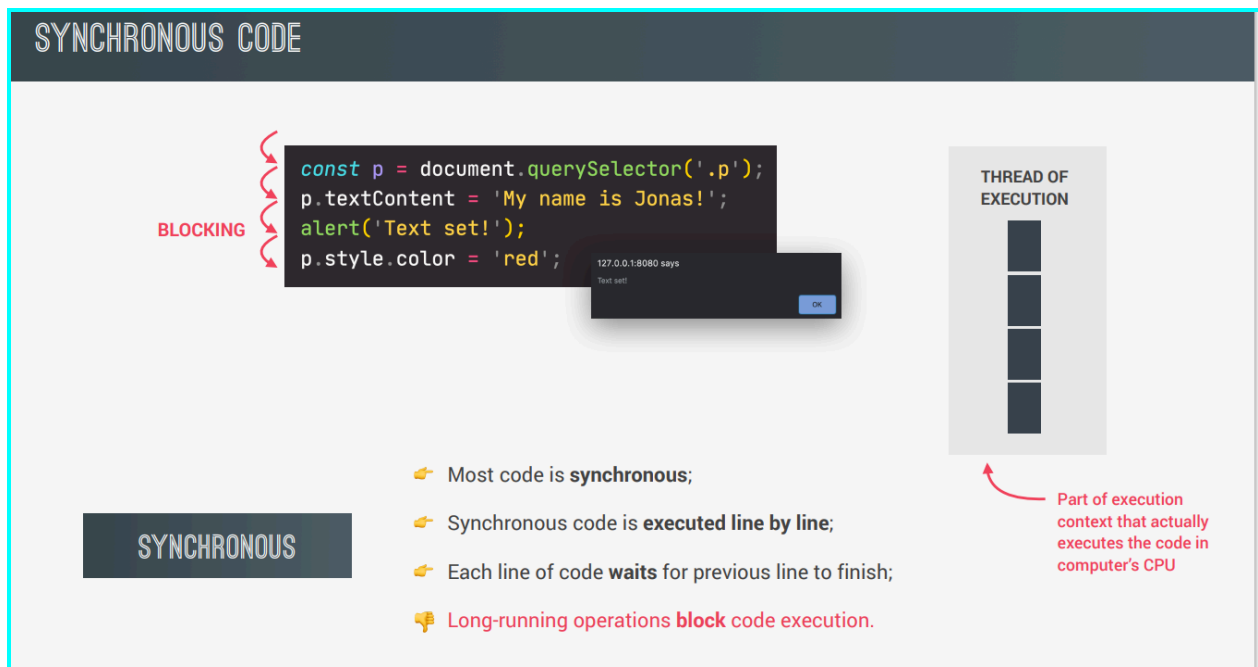
## Asynchronous JavaScript(Promises , Async/Await and AJAX)

### Synchronous

The code is executed line by line in the exact order as what we define in code.

Each line of codes waits for the previous line to finish. This leads to more time to execute the code .

Ex: In the following code, the `alert()` function is a prime example of synchronous behavior. When the alert box is displayed, the JavaScript engine pauses the execution of subsequent code until the user interacts with the alert by clicking "OK." so this leads to more time consuming and blocked .



**Ex2:** Here, each `console.log` is executed one after another.

```
console.log("Start");
console.log("Processing...");
console.log("End");
```

**Output:**

```
Start
Processing...
End
```

## ASynchronous

Asynchronous code allows tasks to run in the background without blocking the execution of the subsequent lines of code. It enables the program to handle time-consuming operations like fetching data, waiting for a timer, or reading files without pausing the entire program.

### Ex1:

In the following code, `setTimeout()` demonstrates asynchronous behavior.

```
console.log("Start"); // Executes immediately

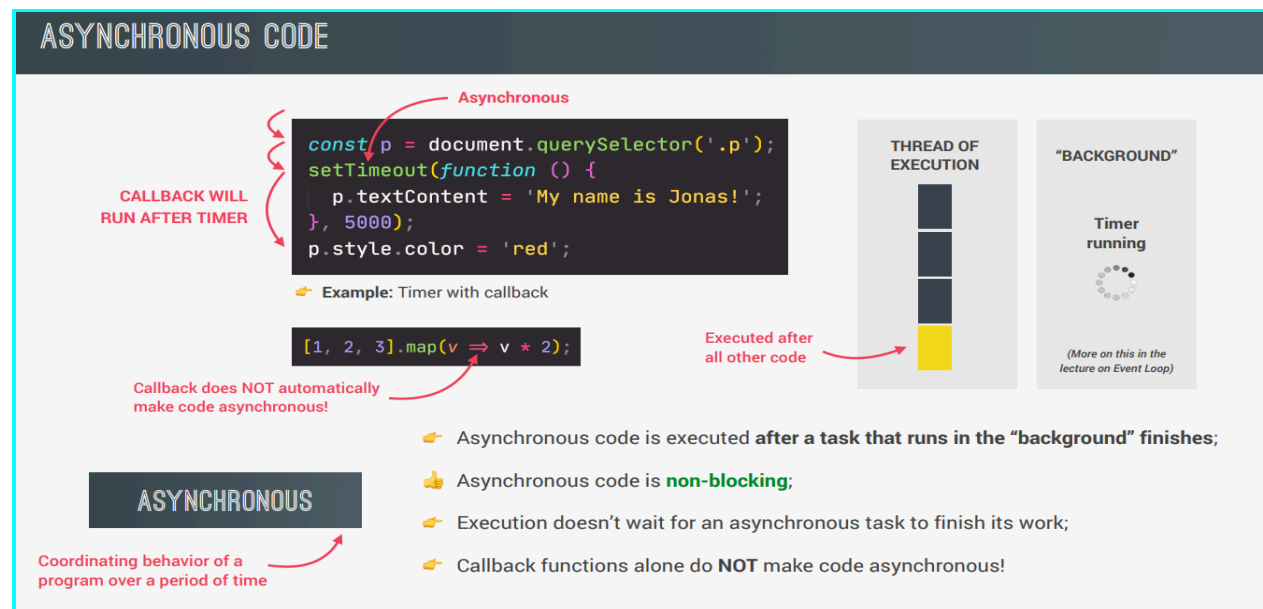
setTimeout(() => {
  console.log("Processing complete!"); // Queued to run after 2 seconds
}, 2000);

console.log("End"); // Executes immediately
```

### Output:

```
Start
End
Processing complete!
```

Here, the `setTimeout` function doesn't block the execution of the `console.log("End")`. Instead, it schedules the callback function to run after 2 seconds.



## Ex2:

```
const img = document.querySelector('dog'); // Selects an element (likely incorrect selector)
img.src = 'dog.png'; // Sets the source of the image element to 'dog.png'

img.addEventListener('load', function () {
  img.classList.add('fadeIn'); // Adds the 'fadeIn' class once the image is fully loaded
});

p.style.width = '300px'; // Sets the width of an element with variable 'p' to 300px
```

## Understanding Synchronous and Asynchronous Behavior in This Code

### 1. `const img = document.querySelector('dog');`

- This line is **synchronous**.
- The `document.querySelector` function immediately selects an element from the DOM if it matches the selector.

### 2. `img.src = 'dog.png';`

- Assigning a value to the `src` property of an image element is an instantaneous operation, but **loading the image** from the specified URL happens **asynchronously** in the background.

### 3. `img.addEventListener('load', function () { ... });`

- This line demonstrates **asynchronous behavior**.
- The `load` event listener is added to the image element. The code inside the callback function will execute only when the image has successfully finished loading.
- `addEventListener` does not automatically make code asynchronous. Instead, the behavior of the code depends on the type of event being listened to and the actions that trigger it. Let's clarify this in the context of our example and the `load` event:
- If the image fails to load (e.g., due to a broken URL), the `load` event will not trigger.
- When the image loads, the callback adds a CSS class `fadeIn`, possibly to apply a fade-in animation or style.

### 4. `p.style.width = '300px';`

- This line is **synchronous**.

- It immediately updates the **width** style of an element (**p**) to **300px**. However, **p** should ideally be selected or defined before this line; otherwise, it will throw an error.

## ASYNCHRONOUS CODE

CALLBACK WILL RUN AFTER IMAGE LOADS

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

Asynchronous

Example: Asynchronous image loading with event and callback

Other examples: Geolocation API or AJAX calls

addEventListener does NOT automatically make code asynchronous!

THREAD OF EXECUTION

"BACKGROUND"

Image loading

(More on this in the lecture on Event Loop)

ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- Asynchronous code is executed **after** a task that runs in the "background" finishes;
- Asynchronous code is **non-blocking**;
- Execution doesn't wait for an asynchronous task to finish its work;
- Callback functions alone do **NOT** make code asynchronous!

## AJAX Calls: ( ASynchronous JavaScript And XML)

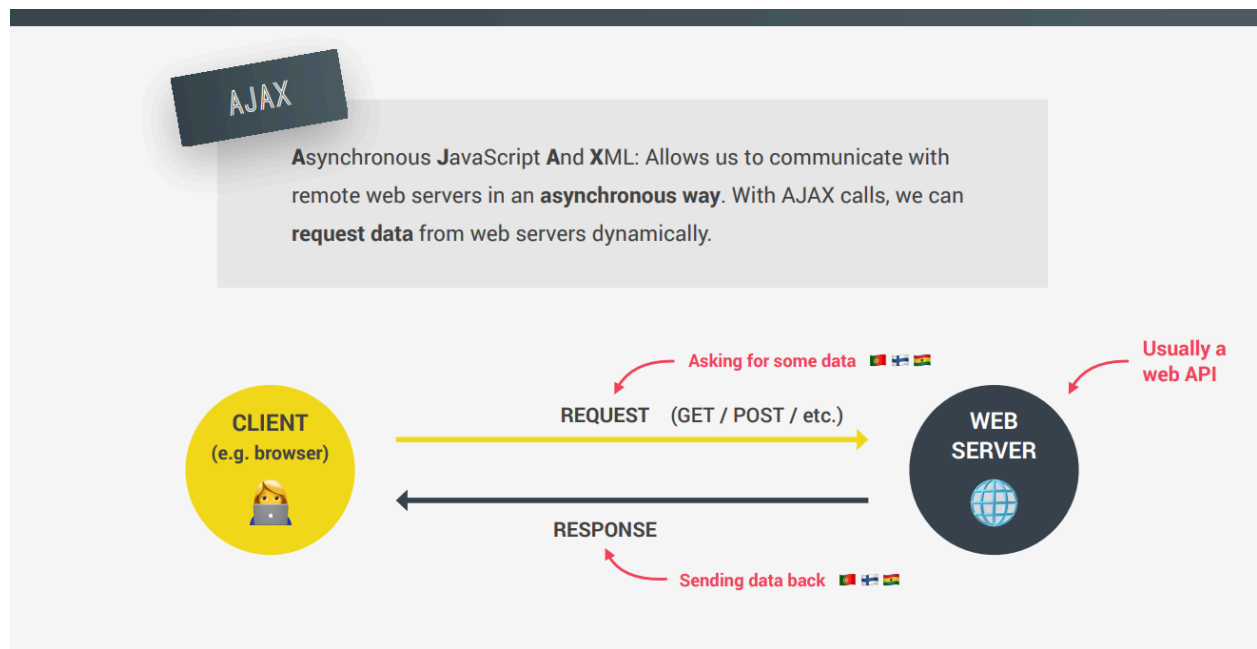
Allow us to communicate with remote web servers in an asynchronous way. With AJAX calls , we can request data from web servers dynamically.

Or

AJAX (Asynchronous JavaScript and XML) is a web development technique used to create dynamic and interactive web applications. **It allows web pages to update content asynchronously, meaning that data can be loaded in the background without needing to refresh the entire page.** This results in a smoother and faster user experience.

Or

AJAX (Asynchronous JavaScript and XML) is a method used in web development that allows web pages to load and update content without reloading the entire page. It enables web browsers to communicate with a server in the background to fetch data, so parts of a page can be updated on the fly without disturbing the user's experience



## How will use public apis

Open <https://github.com/public-apis/public-apis>

Click on respective api , go to documentation and use endpoint url (ex:

<https://restcountries.com/v3.1/name/{name}>)

In the below example, AJAX (Asynchronous JavaScript and XML) is used to send an HTTP request to a server, fetch data from it, and then dynamically update the webpage without reloading the entire page. The API used is the **REST Countries API**. Specifically, the endpoint: [https://restcountries.com/v2/name/\\${country}](https://restcountries.com/v2/name/${country})

This API provides detailed information about countries, such as their name, population, region, languages, flags, and currencies.

The example uses **AJAX** (specifically **XMLHttpRequest**) to fetch country data from an API asynchronously. This means the request is sent to the server without reloading the page, and the server responds with country details like name, population, and currency. The page is then dynamically updated with this data without a full page reload, providing a smooth and efficient user experience.

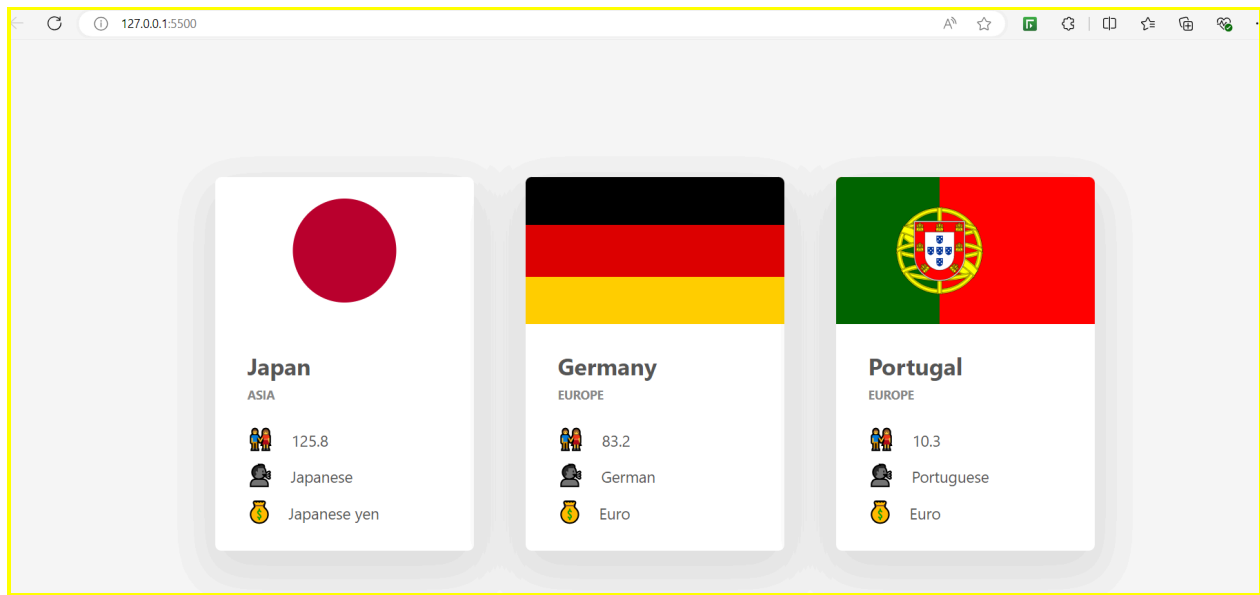
```
const countriesContainer = document.querySelector('.countries');
```

```

const getCountry = function (country) {
  const request = new XMLHttpRequest();
  request.open('GET', `https://restcountries.com/v2/name/${country}`);
  request.send();
  //console.log(request.responseText);
  request.addEventListener('load', function () {
    const [data] = JSON.parse(this.responseText);
    console.log(data);
    const html = `
      <article class="country">
        
        <div class="country__data">
          <h3 class="country__name">${data.name}</h3>
          <h4 class="country__region">${data.region}</h4>
          <p class="country__row"><span>👥</span>${(
            +data.population / 1000000
          ).toFixed(1)}</p>
          <p class="country__row"><span>🗣️</span>${data.languages[0].name}</p>
          <p class="country__row"><span>💰</span>${
            data.currencies[0].name
          }</p>
        </div>
      </article>
    `;
    countriesContainer.insertAdjacentHTML('beforeend', html);
    //the style for countriesContainer is set to be intially opacity=0; so invisible
    countriesContainer.style.opacity = 1;
  });
};

// Fetch details for Japan
getCountry('Japan');
getCountry('germany');
getCountry('portugal');

```



### API:

An API, or application programming interface, is a set of rules or protocols that enables software applications to communicate with each other to exchange data, features and functionality.

Few Data formats of API

Data Format Type	Definition	Example
JSON (JavaScript Object Notation)	A lightweight data format that is easy for humans to read and write, and easy for machines to parse and generate. It is often used in REST APIs.	<pre>{ "name": "John", "age": 30, "city": "New York" }</pre>
XML (eXtensible Markup Language)	A markup language used to encode documents in a format that is both human-readable and machine-readable.	<pre>&lt;person&gt;&lt;name&gt;John&lt;/name&gt;&lt;age&gt;30&lt;/age&gt; &lt;city&gt;New York&lt;/city&gt;&lt;/person&gt;</pre>
YAML (YAML Ain't Markup Language)	A human-readable data serialization standard that can be used for data exchange in APIs, often used for configuration files.	<pre>name: John age: 30 city: New York</pre>
CSV (Comma-Separated Values)	A simple file format used to store tabular data, where each line represents a row and values are separated by commas.	<pre>name,age,city John,30,New York</pre>
Form-Encoded	A data format used for submitting form data, typically in the <code>application/x-www-form-urlencoded</code> type.	<pre>name=John&amp;age=30&amp;city=New+York</pre>
HTML (HyperText Markup Language)	A standard markup language used to create web pages and web applications, often used to structure data for display in browsers.	<pre>&lt;html&gt;&lt;body&gt;&lt;h1&gt;Hello, World!&lt;/h1&gt; &lt;/body&gt;&lt;/html&gt;</pre>

## API Types:

### 1. Open APIs

It is also called public APIs which are available to any other users. **Open APIs** help external users to access data and services. It is an open-source application programming interface that we access with HTTP protocols. These types of APIs are available to the public by companies, allowing external users to access specific functionalities or data. It is a very popular type of API in modern API development, which involves consistency, reusability, and easy integration of services and data from different platforms. It has a developer-friendly environment but the developers who create these Open APIs have to be well aware of the documentation to the efficient usability of API.

Ex: **Google Maps API, Twitter API, Spotify API**

### 2. Internal APIs

**It is also known as a private API, only an internal system exposes this type of API.** These are designed for the internal use of the company rather than the external users. As Private APIs are used for internal use, data formats can be negotiated, depending on the use cases. These can



also be created for better collaboration of data and services among the different teams. It has a very controlled environment within the organization so the utilization of this API is more effective and promotes collaboration, and reusability, and ultimately leads to a more productive development process.

Ex: **User authentication APIs**

### 3. Composite APIs

**It is a type of API that combines different data and services.** The main reason to use Composites APIs is to improve the performance and speed up the execution process and improve the performance of the listeners in the web interfaces. Sometimes, interacting with some websites contains a very complex and time-consuming process, so to solve this problem, we use Composite API. **It allows developers to access a set of related functionalities through a single API endpoint which simplifies the complex interactions and provides a great user experience.**

Ex: **LinkedIn API:** A composite API that allows you to get user profile information and company data in one request. **Social Media Aggregation APIs .. etc**

### 4. Partner APIs

**It is a type of API in which a developer needs specific rights or licenses in order to access.** Partner APIs are not available to the public. These APIs are provided by third-party companies that allow external developers to access their services and integrate them into their applications. It is a very secure and controlled way for external systems to collaborate with the company and access the capabilities of our system. One of the best examples of Partner API is Amazon API which allows online businesses to connect with Amazon.com by providing various functionalities like inventory management, shipping options, and analytics of their business.

Ex: **Stripe API:** Used by businesses to process payments; it can be integrated into a company's website or app to allow payments through a third-party payment processor.

**PayPal API:** Allows integration of PayPal payment methods into other websites, apps, and systems for business transactions.

**There are many types of web API s in web development.**

EX: DOM API , Geolocation API , Own Class API and Online APIs etc..

**Online APIs :** Application running on server , that receives requests from data and send back as response; few Examples

## 1. Weather API

- **Client:** A mobile app or website.
- **API:** A weather service like OpenWeatherMap or WeatherStack.

**Request:** The client sends a GET request to fetch current weather data for a specific city.

GET [https://api.openweathermap.org/data/2.5/weather?q=London&appid=your\\_api\\_key](https://api.openweathermap.org/data/2.5/weather?q=London&appid=your_api_key)

**Response:**

```
{
  "weather": [
    {
      "description": "clear sky"
    }
  ],
  "main": {
    "temp": 289.92
  },
  "name": "London"
}
```

- This response provides the weather description and temperature in London.
- 

## 2. Social Media API (Twitter)

- **Client:** A social media analytics tool or a bot.
- **API:** Twitter API.

**Request:** The client sends a GET request to fetch tweets from a specific user.

GET [https://api.twitter.com/2/tweets?ids=1234567890&tweet.fields=created\\_at](https://api.twitter.com/2/tweets?ids=1234567890&tweet.fields=created_at)

**Response:**

```
{
  "data": [
    {
      "id": "1234567890",
      "text": "This is a sample tweet",
      "created_at": "2024-12-19T15:30:00Z"
    }
  ]
}
```

```
}  
]  
}
```

- This response includes the tweet's text and creation time.

### 3. Authentication API (OAuth2 - Google)

- **Client:** A website or mobile app trying to authenticate users via Google.
- **API:** Google OAuth 2.0 API.

**Request:** The client sends an authorization code to exchange it for a token.

POST <https://oauth2.googleapis.com/token>

body:

code=authorization\_code&client\_id=your\_client\_id&client\_secret=your\_client\_secret&redirect\_uri=your\_redirect\_uri&grant\_type=authorization\_code

**Response:**

```
{  
  "access_token": "ya29.a0AfH6SMC2h9xOWqMGKft3Bcc2RxZ5...",  
  "expires_in": 3600,  
  "token_type": "Bearer"  
}
```

- This response contains the access token that can be used for accessing protected resources.

### Cross Origin Resource Sharing (CORS)

CORS (Cross-Origin Resource Sharing) is a mechanism by which data or any other resource of a site could be shared intentionally to a third party website when there is a need. Generally, access to resources that are residing in a third party site is restricted by the browser clients for security purposes.

Client side code to make an HTTP Call would look like below,

```
function httpGetAction(urlLink)  
{  
    var xmlHttp = new XMLHttpRequest();  
    xmlHttp.open( "GET", urlLink, false );  
    xmlHttp.send();  
    return xmlHttp.responseText;  
}
```

}

This native JavaScript method is intended to make an HTTP call to the given link *urlLink* via the *GET* method and return the response text from the third party resource.

By default, a request to non-parent domains (domains other than the domain from which you are making the call) is blocked by the browser. If you try to do so, the console would throw the following error.,

Failed to load https://write.geeksforgeeks.org/: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'https://www.google.com' is therefore not allowed access.

Of course, there are some cases where you need to access a third party website like getting a public image, making a non-state changing API Call or accessing other domain which you own yourself. There are some ways to achieve this, as and when necessary.

One could get an idea from the error message that the *Access-Control-Allow-Origin* Header is not present on the requested resource. It simply means that the target website whose resource you are trying to access haven't specifically allowed you to get the resource from their site.

This could be done with an additional HTTP Header, *Access-Control-Allow-Origin*. This header could take the following values.,

- **Access-Control-Allow-Origin : [origin]**

*Example : Access-Control-Allow-Origin: https://write.geeksforgeeks.org*

This allows you to specifically allow one website to access your resource. In this case, https://write.geeksforgeeks.org can access your web resource, since it is explicitly allowed.

This requires an additional header called **ORIGIN** sent from the requesting client containing its hostname. This origin header is matched against the allowed origin and the access to the resource is decided.

### **Access-Control-Allow-Origin : \***

Example : Access-Control-Allow-Origin: \*

Wildcard character (\*) means that any site can access the resource you have in your site and obviously it's unsafe.

Based on the request methods (GET/PUT/POST/DELETE) and the request headers, the requests are classified into two categories.

- 1.Simple Requests
- 2.Non Simple/Complex Requests

#### **Simple Requests**

For Simple Requests, the CORS Works on the following way,

- 1.Request is made to a third party site with ORIGIN Header.
- 2.On the target site, the ORIGIN value is compared with the allowed origins.
- 3.If the source is an allowed one, then the resource is granted access, else denied.

### **Promises and the Fetch API**

**Promise:** An Object is used as a placeholder for the future result of an asynchronous operation.

Or

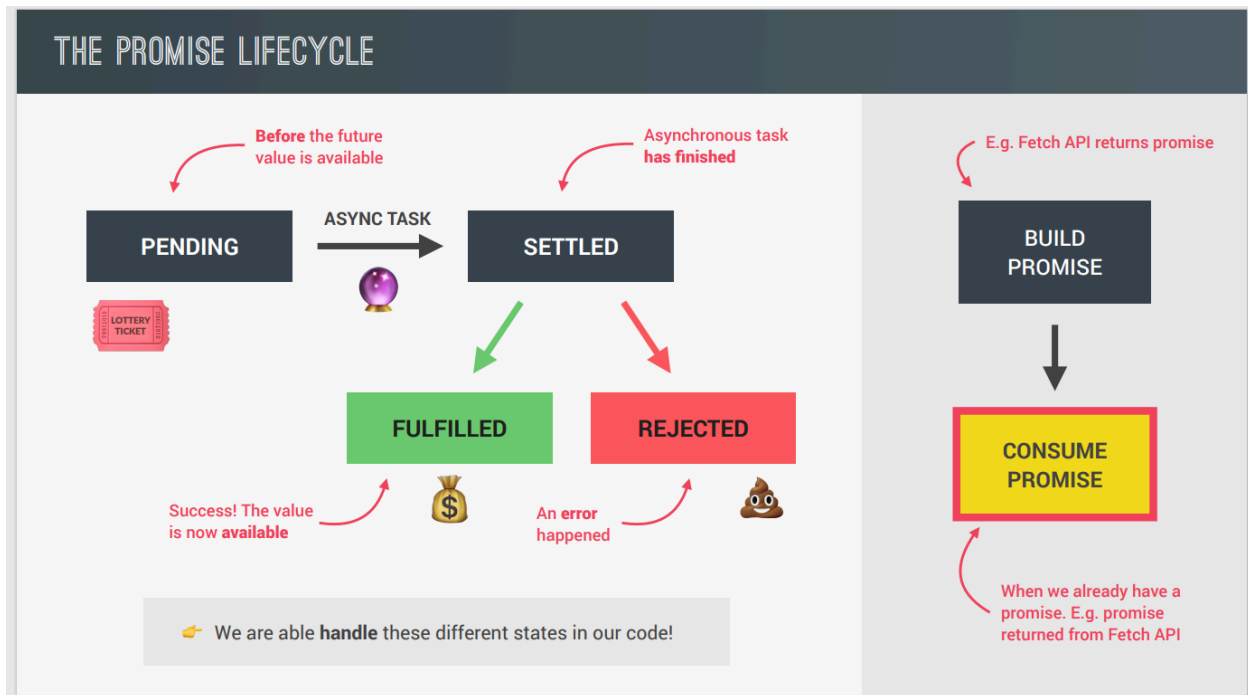
A container for an asynchronous delivered value.

Or

A container for future value.

In JavaScript, a **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to write asynchronous code in a cleaner and more manageable way compared to traditional callback functions.

#### **Life cycle**



**Ex:**

## Lottery Ticket Example for Promises

Let's take a real-world analogy: buying a lottery ticket. When you buy a lottery ticket, you don't immediately know if you've won or lost. Instead, you wait for the result to be announced. This process can be mapped to a Promise.

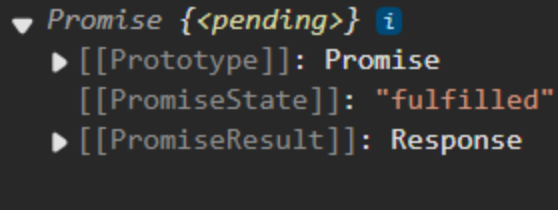
Here's how:

- **Pending**: You bought the lottery ticket and are waiting for the result.
- **Fulfilled**: You won the lottery (the promise is resolved).
- **Rejected**: You didn't win (the promise is rejected).

A promise can be in one of the following states:

1. **Pending**: The initial state, neither fulfilled nor rejected.
2. **Fulfilled**: The operation was completed successfully.
3. **Rejected**: The operation failed.

```
const request = fetch('https://restcountries.com/v2/name/india');
console.log(request);
```



```
▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
  ► [[PromiseResult]]: Response  
>
```

The `fetch` API returns a **Promise**, and your code is an example of a promise in action.

### Explanation:

- `fetch('https://restcountries.com/v2/name/india')` initiates an HTTP request to the specified URL and returns a promise that represents the eventual completion of the network request.
- The promise will either:
  - **Fulfill**: If the HTTP request is successful, the promise resolves to a `Response` object.
  - **Reject**: If there is a network error or some other issue, the promise rejects with an error.

```
const request = fetch('https://restcountries.com/v2/name/india');  
console.log(request);
```

#### 1. `fetch` returns a promise:

When you call `fetch`, it starts an asynchronous operation to fetch data from the given URL. The `request` variable holds this promise.

#### 2. Console log shows the promise:

Since the `fetch` function is asynchronous, the console will log a pending promise immediately after the call, as the operation hasn't completed yet.

### Handling promise:

```
const request = fetch('https://restcountries.com/v2/name/india');  
console.log(request);  
  
const getCountry = function (country) {  
  fetch(`https://restcountries.com/v2/name/${country}`)  
    .then(function (response) {
```

```

    console.log(response);
    return response.json();
  })
  .then(function (data) {
    console.log(data);
  });
};

getCountry('portugal');

//Same code using arrow function

const request = fetch('https://restcountries.com/v2/name/india');
console.log(request);

const getCountry = function (country) {
  fetch(`https://restcountries.com/v2/name/${country}`)
    .then(response => response.json())
    .then(data => console.log(data));
};

getCountry('portugal');

```

1. `const request = fetch('https://restcountries.com/v2/name/india');`

The `fetch` function sends an HTTP request to the given URL (`https://restcountries.com/v2/name/india`) to retrieve data about India.

`fetch` returns a Promise.

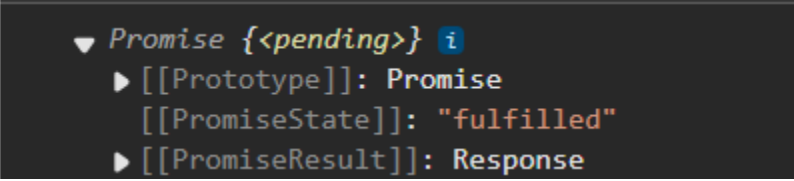
- When the request is successful, the promise resolves with a `Response` object.
- If the request fails (e.g., network error), the promise is rejected with an error.

2. Handling in your code:



Here, `console.log(request)` logs the promise object.

Since the promise has just been initiated, it will likely log something like:



```
▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
  ► [[PromiseResult]]: Response
```

3. `const getCountry = function (country) { ... }`

This is a function that takes a `country` name as an argument and fetches data about that country using the `fetch` API.

Inside `getCountry` function:

First `fetch` Call:

```
fetch('https://restcountries.com/v2/name/${country}')
```

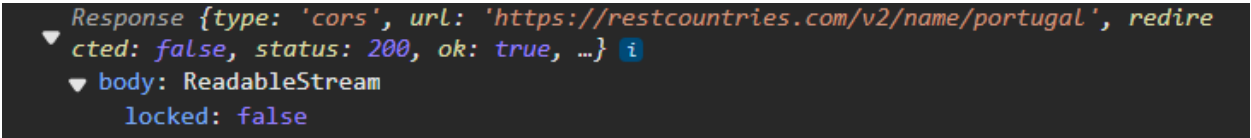
Sends an HTTP request to retrieve country data for the provided `country`.

## Handling the Promise:

`.then(function (response) { ... }):`

- This `.then()` block handles the resolved promise from the `fetch` call.
- `response` is the `Response` object returned by the `fetch`.
- `console.log(response)` logs the response object.

Example output of `response`:



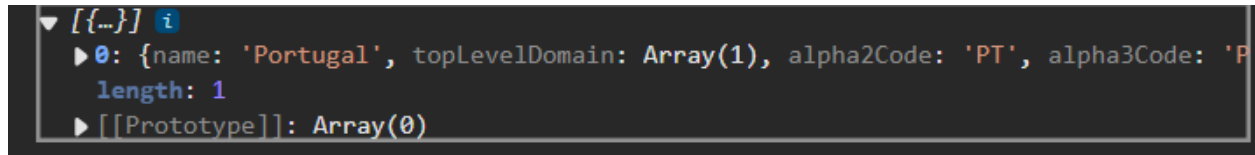
```
▼ Response {type: 'cors', url: 'https://restcountries.com/v2/name/portugal', redirected: false, status: 200, ok: true, ...} ⓘ  
  ▼ body: ReadableStream  
    locked: false
```

- `response.json()`:
- This is a method of the `Response` object that parses the response body as JSON and returns another Promise.
- The promise resolves with the parsed JSON data (the actual country data).

Chaining with Another `.then()`:

```
.then(function (data) {  
  console.log(data);  
});
```

- The second `.then()` handles the promise returned by `response.json()`.
- `data` is the actual JSON data fetched from the API.



```
▼ [{...}] ⓘ  
  ▶ 0: {name: 'Portugal', topLevelDomain: Array(1), alpha2Code: 'PT', alpha3Code: 'P'  
    length: 1  
  ▶ [[Prototype]]: Array(0)
```

## Summary:

- **What is a Promise here?**
  1. The promise is the object returned by the `fetch` function, which represents the eventual result of the HTTP request (success or failure).
  2. `response.json()` also returns a promise that resolves with the parsed JSON data.
- **How are Promises handled?**
  1. **Using `.then()`:**
    - The `.then()` method is used to handle the resolved value of a promise.
    - The first `.then()` handles the response object returned by `fetch`.
    - The second `.then()` handles the promise returned by `response.json()`.

## 1. `.catch()`

The `.catch()` method in JavaScript is used to handle **rejected promises**. When a promise is rejected (for example, due to a network failure, invalid data, or an error in the code), the `.catch()` method allows you to handle the error gracefully.

- **When to use:**
  - When you want to catch and handle any error that occurs in the promise chain.
- **What it does:**
  - It receives the error that was thrown or the rejected promise and allows you to handle it.
- **Where it's used:**
  - After `.then()`, to handle errors that might have occurred in the preceding promise.

**Example:**

```
fetch('https://restcountries.com/v2/name/unknownCountry')

.then(response => {

  if (!response.ok) {

    throw new Error('Country not found');

  }

  return response.json();

})

.catch(error => {

  console.error('Error:', error.message); // Catches any error in the promise chain

});
```

In this example:

- If `fetch()` fails or if `response.ok` is `false`, the promise will be rejected, and the `.catch()` block will handle the error by logging the message.

**`if (!response.ok):`**

- This condition checks the `response.ok` property, which is `true` if the HTTP status code is between `200` and `299` (indicating a successful request).
- If `response.ok` is `false`, it means something went wrong with the request (e.g., 404 Not Found, 500 Server Error, etc.).

**`throw new Error('Failed to fetch data');`**

- If the request fails (i.e., `response.ok` is `false`), this line throws a new **Error** with the message 'Failed to fetch data'.
- Throwing an error will cause the promise to be **rejected**. This means it will skip any subsequent `.then()` blocks and jump to the `.catch()` block to handle the error.

**`finally()`**

The `.finally()` method is used to execute a block of code **after a promise has settled**, regardless of whether it was fulfilled or rejected.

```

fetch('https://restcountries.com/v2/name/unknownCountry')
  .then(response => {
    if (!response.ok) {
      throw new Error('Country not found');
    }
    return response.json();
  })
  .catch(error => {
    console.error('Error:', error.message); // Catches any error in the promise chain
  })
  .finally(() => {
    console.log('Promise completed (success or error).'); // Runs regardless of outcome
  });

```

## Challenge: Asynchronous JavaScript

### Coding Challenge #1

In this challenge you will build a function 'whereAmI' which renders a country only based on GPS coordinates. For that, you will use a second API to geocode coordinates. So in this challenge, you'll use an API on your own for the first time 💎

Your tasks:

#### PART 1

1. Create a function 'whereAmI' which takes as inputs a latitude value ('lat') and a longitude value ('lng') (these are GPS coordinates, examples are in test data below).
2. Do “reverse geocoding” of the provided coordinates. Reverse geocoding means to convert coordinates to a meaningful location, like a city and country name. Use this API to do reverse

geocoding: <https://geocode.xyz/api>. The AJAX call will be done to a URL with this format: <https://geocode.xyz/52.508,13.381?geoit=json>. Use the `fetch` API and promises to get the data. Do not use the 'getJSON' function we created, that is cheating 💎

3. Once you have the data, take a look at it in the console to see all the attributes that you received about the provided location. Then, using this data, log a message like this to the console: "You are in Berlin, Germany"
4. Chain a `.catch` method to the end of the promise chain and log errors to the console
5. This API allows you to make only 3 requests per second. If you reload fast, you will get this error with code 403. This is an error with the request. Remember, `fetch()` does not reject the promise in this case. So create an error to reject the promise yourself, with a meaningful error message

## PART 2

6. Now it's time to use the received data to render a country. So take the relevant attribute from the geocoding API result, and plug it into the countries API that we have been using.
7. Render the country and catch any errors, just like we have done in the last lecture (you can even copy this code, no need to type the same code)

Test data:

Coordinates 1: 52.508, 13.381 (Latitude, Longitude)

Coordinates 2: 19.037, 72.873

Coordinates 3: -33.933, 18.474

```
const renderCountry = function (data, className = "") {  
  const html = `  
    <article class="country ${className}" >  
        
      <div class="country__data">  
        <h3 class="country__name">${data.name}</h3>  
      </div>  
    </article>  
  `;  
  return html;  
}
```

```

<h4 class="country__region">${data.region}</h4>

<p class="country__row"><span>👥</span>${(
  +data.population / 1000000
).toFixed(1)} people</p>

<p class="country__row"><span>🗣️</span>${data.languages[0].name}</p>

<p class="country__row"><span>💰</span>${data.currencies[0].name}</p>

</div>

</article>

`;

countriesContainer.insertAdjacentHTML('beforeend', html);

countriesContainer.style.opacity = 1;

};

```

```

///1
const whereAmI = function (lat, lng) {
  fetch(
    `https://api.bigdatacloud.net/data/reverse-geocode-client?latitude=${lat}&longitude=${lng}`
  )
    .then(res => {
      if (!res.ok) throw new Error(`Problem with geocoding ${res.status}`);
      return res.json();
    })
    .then(data => {
      console.log(data);
    })
};
///2

```

```

    console.log(`You are in ${data.city}, ${data.countryName}`);
    fetch(`https://restcountries.com/v2/name/${data.countryName}`);
  })
  .then(response => {
    if (!response.ok)
      throw new Error(`Problem with geocoding${response.status}`);
    return response.json();
  })
  .then(data => renderCountry(data[0]))
  .catch(err => console.log(`${err.message} 🤯`));
};

whereAmI(52.508, 13.381); //You are in Berlin, Germany
//whereAmI(19.037, 72.873); //You are in Mumbai, India
//whereAmI(-33.933, 18.474); //You are in Cape Town, South Africa

```

## Event Loop

The **event loop** in JavaScript is a mechanism that enables asynchronous programming by continuously checking the **call stack**, the **callback queue**, and the **microtask queue**. It ensures that tasks are executed in a non-blocking, efficient manner in a single-threaded environment.

Here's how the event loop works:

1. **Call Stack:** This is where the currently executing function or task resides. The event loop always checks if the call stack is empty. If it's not, it continues executing the synchronous code.
2. **Web APIs:** These are browser-provided APIs (such as `setTimeout`, `fetch`, and DOM events) that allow asynchronous operations. When an asynchronous operation is invoked, the browser delegates the task to the Web APIs, allowing the JavaScript code to continue executing other operations.
3. **Callback Queue (Task Queue):** Once a Web API finishes its task (e.g., a timer expires, or a network request completes), the associated callback function is placed in the callback queue. This queue holds tasks waiting to be executed when the call stack is empty.

4. **Microtask Queue:** This queue stores higher-priority tasks like **Promises** and **MutationObserver** callbacks. These tasks are executed before the tasks in the callback queue. Microtasks are processed after the currently executing function finishes, but before the event loop will first process all the tasks in the **microtask queue** before it starts processing tasks in the **callback queue**.

### Event Loop Process:

- The **event loop** continuously checks the **call stack**.
- If the call stack is empty, it first checks the **microtask queue** and processes any tasks inside it.
- After processing the microtasks, the event loop then checks the **callback queue** and processes any tasks there.

### Execution Flow:

1. **Synchronous code** (code in the call stack) executes first.
2. **Microtasks** (like resolved Promises) execute next.
3. **Tasks** from the callback queue (such as **setTimeout**) execute last.

```
// 1. Synchronous code
console.log('happy');

// 2. Asynchronous operation (fetch)
fetch('https://example.com').then(res => console.log(res)); // This goes to the microtask queue

// 3. Set a timeout with a very short delay
setTimeout(() => {
  console.log('time out');
}, 1); // This goes to the callback queue

// 4. Add event listener to an image
let el = document.querySelector('img');
el.src = 'dog.img';
el.addEventListener('load', () => {
  el.classList.add('fadeIn'); // This happens when the image finishes loading
});
```



```
// 5. More synchronous code
console.log('happy learning');
```

### Step-by-Step Execution:

#### 1. Start Execution:

- JavaScript begins executing the code synchronously, meaning the first line of code is pushed to the **call stack** and executed immediately.
- **Execution Order** (Synchronous Part):
  - **console.log("happy")**: Logs "happy" to the console.
  - **console.log("happy learning")**: Logs "happy learning" to the console.
- At this point, both of these synchronous operations have been completed, and the call stack is empty.

#### 2. Asynchronous Operations: After the synchronous code, asynchronous operations (like **fetch** and **setTimeout**) are encountered. These don't block the execution and are sent to their respective **web APIs**.

- **fetch("https://example.com")**:
  - This is an asynchronous operation, so JavaScript hands it off to the **web API** (browser's networking module).
  - The HTTP request is made, and once the response is received, the **then()** callback is placed in the **microtask queue**.
- **setTimeout(() => { console.log("time out"); }, 1)**:
  - This is also an asynchronous operation but is handled by the **web API** related to timers.
  - The **setTimeout** callback is placed in the **callback queue** once the timer expires (after 1 millisecond).
- **el.addEventListener('load', () => { el.classList.add('fadeIn'); })**:
  - This is setting an event listener on an image element.
  - When the image finishes loading, the callback is moved to the **callback queue** to be executed later.

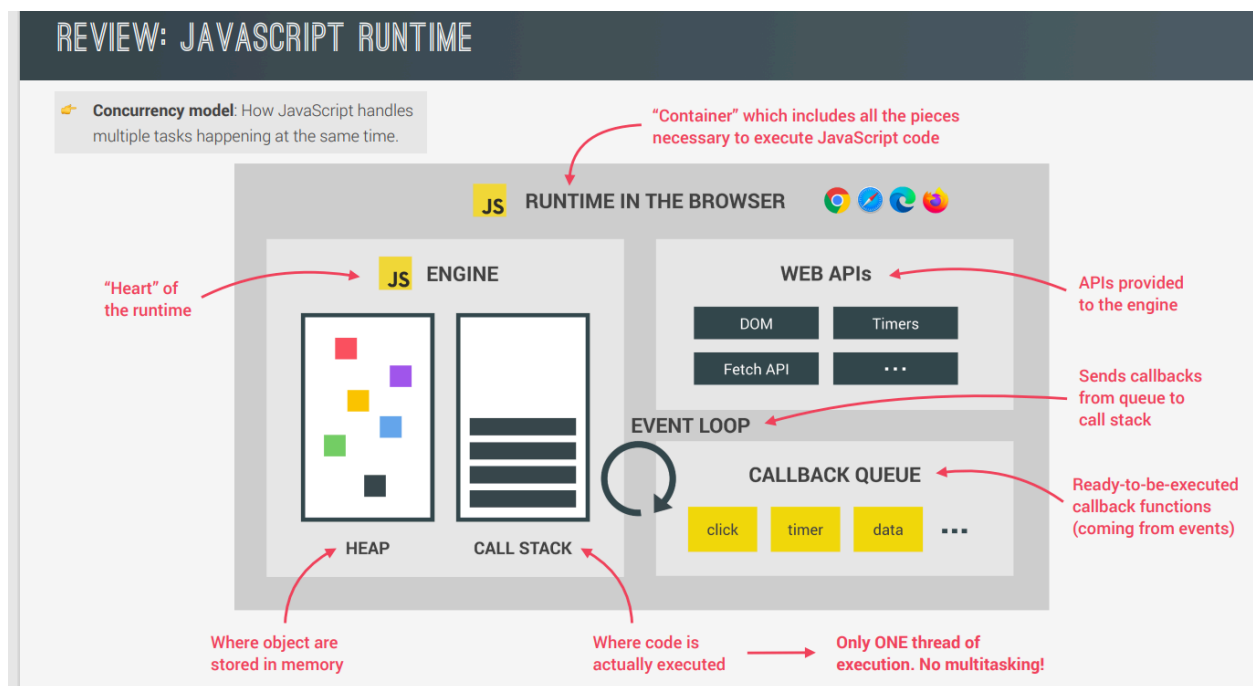
#### 3. Event Loop: After the synchronous code has finished executing, the **event loop** begins monitoring the call stack and the two queues (microtask queue and callback queue).

- The event loop first checks the **microtask queue**, which has higher priority than the callback queue. Ex: **Promises** (e.g., **.then()**, **.catch()**) and **Other tasks** like **MutationObserver** callbacks and **queueMicrotask()**.
- After the call stack is empty, it moves on to process the tasks in the **microtask queue** and then the **callback queue**.

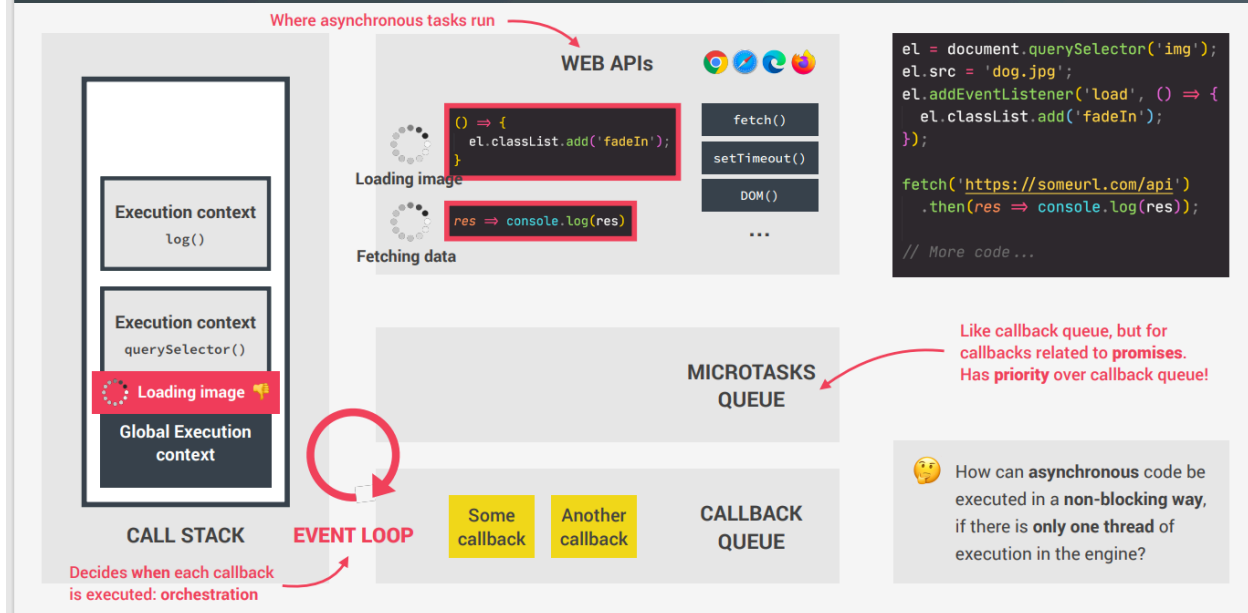
## Final Execution Order:

1. "happy" is logged (synchronous).
2. "happy learning" is logged (synchronous).
3. The fetch response is logged (after promise resolution, placed in the microtask **queue**).
4. "time out" is logged (after `setTimeout`, placed in the callback **queue**).
5. The image's "fadeIn" class is added (after the image load event, placed in the callback **queue**).

Once asynchronous functions complete run in the background on web apis , and their callbacks are moved to the respective queues. The event loop then takes care of executing them when the call stack is empty(the call **stack is empty** means that all **synchronous code** has finished executing.).



# HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES



Ex2:

```
console.log('start buddy');
setTimeout(function () {
  console.log('timer callback buddy');
}, 0);

Promise.resolve('Resolved a promise').then(res => console.log(res)); //create a promise
Promise.resolve('Resolved a promise 2').then(res => {
  for (let i = 0; i < 10000; i++) {}
  console.log(res);
});

console.log('end buddy');
```

```
start buddy
end buddy
Resolved a promise
Resolved a promise 2
timer callback buddy
```

## Building/Creating Promises

### Syntax:

```
let promise = new Promise((resolve, reject) => {
  // Perform async operation
  if (operationSuccessful) {
    resolve("Task successful");
  } else {
    reject("Task failed");
  }
});
```

- `resolve(value)`: Marks the promise as fulfilled and provides a result.
- `reject(error)`: Marks the promise as rejected with an error

### Ex1

```
let myPromise = new Promise((resolve, reject) => {
  // Some asynchronous operation, like an API call or setTimeout
  let success = true;

  if (success) {
    resolve('Operation was successful!');
  } else {
    reject('Operation failed.');
```

```
myPromise
  .then(result => {
    console.log(result); // Logs: "Operation was successful!"
  })
  .catch(error => {
    console.error(error); // Logs error if any.
  });
```

**Ex2:** You can create custom promises for your own asynchronous operations. Below is an example:

```
function fetchData(url) {
  return new Promise((resolve, reject) => {
    // Simulate async operation
    setTimeout(() => {
      if (url === 'validUrl') {
        resolve('Data fetched successfully');
      } else {
        reject('Invalid URL');
      }
    }, 1000);
  });
}

fetchData('validUrl')
  .then(data => console.log(data)) // "Data fetched successfully"
  .catch(error => console.log(error));
```

**EX3:**

```
// Building a Simple Promise
const lotteryPromise = new Promise(function (resolve, reject) {
```

```

console.log('Lottery draw is happening 🎰');
setTimeout(function () {
  if (Math.random() >= 0.5) {
    resolve('You WIN 💰');
  } else {
    reject(new Error('You lost your money 😭'));
  }
}, 2000);
});

lotteryPromise.then(res => console.log(res)).catch(err => console.error(err));

// Promisifying setTimeout
const wait = function (seconds) {
  return new Promise(function (resolve) {
    setTimeout(resolve, seconds * 1000);
  });
};

wait(1)
  .then(() => {
    console.log('1 second passed');
    return wait(1);
  })
  .then(() => {
    console.log('2 second passed');
    return wait(1);
  })
  .then(() => {
    console.log('3 second passed');
    return wait(1);
  })
  .then(() => console.log('4 second passed'));

```

## Promisifying Geolocation API

### Challenge2

For this challenge you will actually have to watch the video! Then, build the image loading functionality that I just showed you on the screen.

Your tasks:

Tasks are not super-descriptive this time, so that you can figure out some stuff by yourself. Pretend you're working on your own 💎

#### PART 1

1. Create a function 'createImage' which receives 'imgPath' as an input. This function returns a promise which creates a new image (use `document.createElement('img')`) and sets the `.src` attribute to the provided image path
2. When the image is done loading, append it to the DOM element with the 'images' class, and resolve the promise. The fulfilled value should be the image element itself. In case there is an error loading the image (listen for the 'error' event), reject the promise
3. If this part is too tricky for you, just watch the first part of the solution

#### PART 2

4. Consume the promise using `.then` and also add an error handler
5. After the image has loaded, pause execution for 2 seconds using the 'wait' function we created earlier
6. After the 2 seconds have passed, hide the current image (set display CSS property to 'none'), and load a second image (Hint: Use the image element returned by the 'createImage' promise to hide the current image. You will need a global variable for that 💎)
7. After the second image has loaded, pause execution for 2 seconds again
8. After the 2 seconds have passed, hide the current image

Test data:

Images in the img folder. Test the error handler by passing a wrong image path. Set the network speed to “Fast 3G” in the dev tools Network tab, otherwise images load too fast

```
//challenge 3

const imgConatiner = document.querySelector('.images');
/// Promisifying setTimeout
const wait = function (seconds) {
  return new Promise(function (resolve) {
    setTimeout(resolve, seconds * 1000);
  });
};

const createImage = function (imgPath) {
  return new Promise(function (resolve, reject) {
    const img = document.createElement('img');
    img.src = imgPath;
    img.addEventListener('load', function () {
      imgConatiner.append(img);
      resolve(img);
    });
    img.addEventListener('error', function () {
      reject(new Error('Image not Found'));
    });
  });
};

//
let currentImage;
createImage('img/img-1.jpg')
  .then(img => {
    currentImage = img;
    console.log('Image1 loaded');
    return wait(2);
  });
```



```

})
.then(() => {
  currentImage.style.display = 'none';
  return createImage('img/img-2.jpg');
})
.then(img => {
  currentImage = img;
  console.log('Image2 loaded');
  return wait(2);
})
.then(() => {
  currentImage.style.display = 'none';
})
.catch(err => console.error(err));

```

#### 1. Promisifying `setTimeout`:

The `wait` function wraps `setTimeout` in a Promise, allowing you to pause execution in a chainable `.then()` format.

#### 2. Image Loading with Promises:

The `createImage` function dynamically creates an image element, sets its source, and resolves or rejects the Promise based on whether the image loads successfully or fails.

#### 3. Chaining Promises:

The script:

- Loads the first image (`img/img-1.jpg`).
- Waits for 2 seconds.
- Hides the first image and loads the second one (`img/img-2.jpg`).
- Waits for another 2 seconds.
- Hides the second image.

#### 4. Error Handling:

The `.catch()` block handles any errors, such as when an image fails to load.

### Promises with Async/Await

`async` and `await` are part of JavaScript's asynchronous programming model introduced in ES2017 (ES8). They simplify working with promises, making asynchronous code look and behave more like synchronous code, which improves readability and maintainability.

Async and Await in JavaScript is used to simplify handling asynchronous operations using promises.

**async:** Declares an asynchronous function. It always returns a promise.

**Await :** The await keyword is used to wait for a promise to resolve. It can only be used within an async block. Await makes the code wait until the promise returns a result.

`await` pauses the execution of the **async function** it is inside, waiting for the promise to finish (either successfully or with an error). However, it doesn't stop the rest of your program from running—other parts of the code can keep going while it waits.

### Why `async` and `await`?

Before `async` and `await`, asynchronous code was primarily handled using:

1. **Callback functions:** Often led to "callback hell" (nested and hard-to-read code).
2. **Promises:** Improved readability but chaining `.then()` and `.catch()` could still become complex.

`async` and `await` allow developers to write asynchronous code in a way that feels synchronous, making it easier to reason about.

### Advantages of `async` and `await`

1. **Improved Readability:** Code appears linear and easier to follow.
2. **Error Handling:** Use of `try...catch` for managing errors is more intuitive.
3. **No Callback Hell:** Avoid deeply nested callbacks.
4. **Debugging:** Stack traces are easier to follow compared to promise chains.

```
//before
/*
const whereAmI = function (country) {
```

```

fetch(`https://restcountries.com/v2/name/${country}`)
  .then(response => response.json())
  .then(data => console.log(data));
};
whereAmI('india');
*/

//using async and await

const whereAmI = async function (country) {
  const res = await fetch(`https://restcountries.com/v2/name/${country}`);
  const data = await res.json();
  console.log(data);
};
whereAmI('india');
console.log('hello');

```

## Callback Hell to Promises to async/await

### Callback Hell

#### What is it?

In callback-based programming, functions are passed as arguments and executed when an asynchronous operation completes. When there are many nested operations, it becomes difficult to manage and debug, leading to "callback hell."

#### Example: Fetching user, their posts, and comments

```

// Simulating fetching a user from a database or API
function fetchUser(userId, callback) {
  setTimeout(() => {
    // After 1 second, the user object is returned with the provided userId
    callback({ id: userId, name: "John" });
  }, 1000);
}

```

```

}

// Simulating fetching posts associated with a user
function fetchPosts(userId, callback) {
  setTimeout(() => {
    // After 1 second, an array of posts for the user is returned
    callback([ { id: 1, title: "Post 1" }, { id: 2, title: "Post 2" } ]);
  }, 1000);
}

// Simulating fetching comments associated with a specific post
function fetchComments(postId, callback) {
  setTimeout(() => {
    // After 1 second, an array of comments for the post is returned
    callback(["Comment 1", "Comment 2"]);
  }, 1000);
}

// Callback Hell: A nested structure where each operation depends on the previous one

fetchUser(1, (user) => { // Step 1: Fetch the user with ID 1
  console.log("User fetched:", user); // Once the user is fetched, print the user info

  // Step 2: Fetch the posts for the user (fetchPosts is dependent on fetchUser)
  fetchPosts(user.id, (posts) => {
    console.log("Posts fetched:", posts); // Once posts are fetched, print the posts info

    // Step 3: Fetch comments for the first post (fetchComments is dependent on fetchPosts)
    fetchComments(posts[0].id, (comments) => {
      console.log("Comments fetched:", comments); // Once comments are fetched, print the
      comments
    });
  });
});

```

Each function calls another function within its own callback. As a result, the code becomes deeply indented and harder to follow as more operations are added.

fetchUser(1, callback):

- Fetches the user with ID 1.

- The user data { id: 1, name: "John" } is passed to the callback.

fetchPosts(user.id, callback):

- The user.id from the previous function (1) is passed to fetchPosts to fetch the user's posts.

fetchComments(posts[0].id, callback):

- The posts[0].id (ID of the first post) is passed to fetchComments to fetch comments for that post.

### Problems:

- Difficult to read and debug.
  - Hard to scale for more complex operations.
- 

### Using Promises

#### What is it?

Promises improve readability by chaining `.then()` and `.catch()` methods. This avoids deep nesting but can still become verbose with multiple `.then()` calls.

#### Example:

```
function fetchUser(userId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve({ id: userId, name: "John" });  
    }, 1000);  
  });  
}
```

```
function fetchPosts(userId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve([ { id: 1, title: "Post 1" }, { id: 2, title: "Post 2" } ]);  
    }, 1000);  
  });  
}
```

```
function fetchComments(postId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve(["Comment 1", "Comment 2"]);  
    }, 1000);  
  });  
}
```

// Chaining Promises

```
fetchUser(1)  
  .then((user) => {  
    console.log("User fetched:", user);  
    return fetchPosts(user.id);  
  })  
  .then((posts) => {  
    console.log("Posts fetched:", posts);  
    return fetchComments(posts[0].id);  
  })  
  .then((comments) => {  
    console.log("Comments fetched:", comments);  
  })  
  .catch((error) => {  
    console.error("Error:", error);  
  });
```

### Improvements:

- No deep nesting.
- Easier to manage error handling with `.catch()`.

---

## Using `async` and `await`

### What is it?

`async` and `await` simplify asynchronous code further by making it look like synchronous code. This improves readability and makes it easier to handle errors with `try...catch`.

### Example:

```
function fetchUser(userId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve({ id: userId, name: "John" });  
    }, 1000);  
  });  
}
```

```
function fetchPosts(userId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve([ { id: 1, title: "Post 1" }, { id: 2, title: "Post 2" } ]);  
    }, 1000);  
  });  
}
```

```
function fetchComments(postId) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve(["Comment 1", "Comment 2"]);  
    }, 1000);  
  });  
}
```

```
// Using async/await  
async function fetchAllData() {  
  try {  
    const user = await fetchUser(1);  
    console.log("User fetched:", user);  
  
    const posts = await fetchPosts(user.id);  
    console.log("Posts fetched:", posts);  
  
    const comments = await fetchComments(posts[0].id);  
    console.log("Comments fetched:", comments);  
  } catch (error) {  
    console.error("Error:", error);  
  }  
}
```

`fetchAllData();`

### Advantages:

- Code looks synchronous and linear.
- Easier to read and debug.
- Error handling with `try...catch` is straightforward.

## Running Promise in Parallel

### Promise.all()

The `Promise.all()` static method takes an iterable of promises as input and returns a single [Promise](#). The `Promise.all()` method returns a single Promise from a list of promises, when all promises fulfill.

**Promise.all() Returns a Single Promise:** Even though you pass multiple promises to `Promise.all()`, it returns a new single promise. This single promise represents the state of all the promises passed to it.

**All Promises Must Fulfill:** The new promise returned by `Promise.all()` resolves when **all** of the promises passed to it fulfill/resolve. If **any** of the promises are rejected, the returned promise will immediately reject with the reason of the first promise that fails.

### When is `Promise.all()` Recommended?

`Promise.all()` is recommended when:

1. **You want multiple promises to run concurrently** and you need the results of all the promises to proceed together.
2. **All promises must succeed** for the operation to be considered successful.
3. **You need to gather all results at once.** If any promise is rejected, you don't want to continue with incomplete data.

Common use cases for `Promise.all()` include:

- Fetching data from multiple APIs simultaneously and processing the results once all of them are successful.
- Performing multiple asynchronous tasks in parallel, like writing to several files or databases, and then combining the results.



## What Happens When One Promise Fails in `Promise.all()`?

If **any one promise fails** (i.e., it rejects) in `Promise.all()`, the following happens:

1. **Immediate Rejection:** The promise returned by `Promise.all()` immediately rejects with the error (or rejection reason) of the first promise that fails. It does **not** wait for the other promises to resolve or reject.
2. **Subsequent Promises Continue Running:** Even though `Promise.all()` has rejected, the other promises will continue to run. Their results are ignored because the whole `Promise.all()` operation has already been rejected.

if all promises are fulfilled, it wraps the results of all fulfilled promises into a single array and resolves that.

### Example with multiple resolved promises:

```
const promise1 = Promise.resolve(1);
```

```
const promise2 = Promise.resolve(2);
```

```
const promise3 = Promise.resolve(3);
```

```
Promise.all([promise1, promise2, promise3]).then(values => {  
  console.log(values); // [1, 2, 3]  
}).catch(error => {  
  console.error(error);  
});
```

In this case, since all promises are fulfilled, the result is an array of the resolved values.

**Example with a rejected promise:** If at least one promise rejects, it rejects with the error of the first rejected promise.

```
const promise1 = Promise.resolve(1);
```

```
const promise2 = Promise.reject('Error');
```

```
const promise3 = Promise.resolve(3);
```

```
Promise.all([promise1, promise2, promise3]).then(values => {  
  console.log(values);  
}).catch(error => {  
  console.error(error); // 'Error'  
});
```

In this case, since one of the promises (`promise2`) rejects, the entire `Promise.all()` call rejects immediately with `'Error'`, and the remaining promises (`promise3`) are not processed.

### tasks running individually (sequentially):

```
// Simple async functions that return promises  
const task1 = () =>  
  new Promise(resolve => setTimeout(() => resolve('Task 1 done'), 1000));  
const task2 = () =>  
  new Promise(resolve => setTimeout(() => resolve('Task 2 done'), 500));  
const task3 = () =>  
  new Promise(resolve => setTimeout(() => resolve('Task 3 done'), 1500));  
  
const runTasksIndividually = async () => {  
  // Run the first task and log its result  
  const result1 = await task1();  
  console.log(result1); // 'Task 1 done'  
  
  // Run the second task and log its result  
  const result2 = await task2();  
  console.log(result2); // 'Task 2 done'
```

```

// Run the third task and log its result
const result3 = await task3();
console.log(result3); // 'Task 3 done'
};

runTasksIndividually();

```

Running tasks concurrently with **Promise.all()**:

```

const task1 = () =>
  new Promise(resolve => setTimeout(() => resolve('Task 1 done'), 1000));
const task2 = () =>
  new Promise(resolve => setTimeout(() => resolve('Task 2 done'), 500));
const task3 = () =>
  new Promise(resolve => setTimeout(() => resolve('Task 3 done'), 1500));

const runTasksConcurrently = async () => {
  const results = await Promise.all([task1(), task2(), task3()]);
  console.log(results); // ['Task 1 done', 'Task 2 done', 'Task 3 done']
};

runTasksConcurrently();

```

**race, allSettled and any**

**Promise.race():**

The Promise.race() static method takes an iterable of promises as input and returns a single [Promise](#).

**Purpose:** Returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects.

Or

The **Promise.race()** method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

We may think of this particular method as in the form of a real-life example where several people are running in a race whoever wins comes first wins the race, the same scenario is here, which ever promise is successfully fulfills or rejects at early will be executed at first and rest one's results will not be displayed as an output.

**Ex1:**

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one');
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then(value => {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// Expected output: "two"
```

**Ex2**

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('five'), 500);
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('six')), 100);
});

Promise.race([promise1, promise2]).then(value => {
  console.log(value);
});
```

```
});
```

```
✖ ▶ Uncaught (in promise) Error: six  
    at script.js:392:27
```

```
>
```

## Promise.allSettled()

**Purpose:** Returns a promise that resolves/fulfills after all the given promises have either resolved or rejected. It always resolves with an array of objects, each representing the outcome of each promise.

**Behavior:** This method allows you to track the outcome (fulfilled or rejected) of each promise, regardless of whether they succeed or fail. It waits for all promises to either resolve or reject, and then returns an array of results showing the status of each promise (fulfilled or rejected).

"Wait for all promises to finish, no matter if they succeed or fail."

**When it's used:** You have multiple promises, and you want to know the outcome (fulfilled or rejected) of each one, even if some fail.

**Example 1:** In this example, we will use the Promise **allSettled()** Method

```
const p1 = Promise.resolve(50);  
  
const p2 = new Promise((resolve, reject) => setTimeout(reject, 100, 'geek'));  
  
const prm = [p1, p2];  
  
Promise.allSettled(prm).then(results =>  
  
    results.forEach(result => console.log(result.status, result.value))  
  
);
```

```
fulfilled 50
```

```
rejected undefined
```

```
>
```

In this example, we will use the Promise **allSettled()** Method

```

const tOut = t => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Completed in ${t}`);
    }, t);
  });
};

// Resolving a normal promise
tOut(1000).then(result => console.log(result));
// Completed in 1000

// Promise.allSettled
Promise.allSettled([tOut(1000), tOut(2000)]).then(result =>
  console.log(result)
);

```

Completed in 1000

```

▼ (2) [{...}, {...}] ⓘ
  ► 0: {status: 'fulfilled', value: 'Completed in 1000'}
  ► 1: {status: 'fulfilled', value: 'Completed in 2000'}
    length: 2
  ► [[Prototype]]: Array(0)

```

```

let first_promise = Promise.resolve(200);

let second_promise = Promise.reject('Rejected Promise');

let third_promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(500), 100);
});

let result = Promise.allSettled([first_promise, second_promise, third_promise]);
result.then(value => console.log(value));

```

```
▼ (3) [{...}, {...}, {...}] ⓘ  
  ▶ 0: {status: 'fulfilled', value: 200}  
  ▶ 1: {status: 'rejected', reason: 'Rejected Promise'}  
  ▶ 2: {status: 'fulfilled', value: 500}  
    length: 3  
  ▶ [[Prototype]]: Array(0)
```

## Promise.any()

The `Promise.any()` static method takes an iterable of promises as input and returns a single [Promise](#)

Returns a promise that resolves/fulfills as soon as any of the promises in the iterable resolves. If all the promises are rejected, it returns a rejection with an [AggregateError](#).

**Behavior:** It resolves with the value of the first fulfilled/resolved promise, or rejects if all promises are rejected.

Or

**JavaScript Promise any() method** is a static method that takes an array of promises as a parameter and returns the first fulfilled/resolved promise. It returns a rejected value when all of the promises in the array return rejects or if the array is empty. When all the promises are rejected an `AggregateError` is returned which contains the reason for rejection.

### Syntax:

```
Promise.any(iter)
```

**Parameters:** It takes only one parameter.

- **iter:** It is iterable of promises.

```
let prom1 = new Promise((resolve, reject) => {  
  reject('Failure');  
});  
let prom2 = new Promise((resolve, reject) => {  
  reject('Failed to load');
```

```

});
let prom3 = new Promise((resolve, reject) => {
  resolve('Worked');
});
let prom4 = new Promise((resolve, reject) => {
  resolve('Successful');
});

let prom = [prom1, prom2, prom3, prom4];

Promise.any(prom).then(val => {
  console.log(val);
});

```

The first two promises were rejected and the first successful promise which was prom3 was returned whose value is captured using then and printed on the console

Worked

Ex2: This example shows how the promise is rejected in any method.

```

let prom1 = new Promise((resolve, reject) => {
  reject('Failure');
});
let prom2 = new Promise((resolve, reject) => {
  reject('Failed to load');
});
let prom3 = new Promise((resolve, reject) => {
  reject('Unsuccessful');
});
let prom4 = new Promise((resolve, reject) => {
  reject('Rejected');
});

```



```
let prom = [prom1, prom2, prom3, prom4];
```

```
Promise.any([])
```

```
.then(val => console.log(val))
```

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

```
Promise.any(prom)
```

```
.then(val => console.log(val))
```

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

When we pass an empty array in any method it is automatically rejected and `AggregateError` is returned, also if all the promises passed in an array return a rejected same output is shown.

```
AggregateError: All promises were rejected
```

```
AggregateError: All promises were rejected
```

```
>
```

**Promise.all():** Resolves when all promises are fulfilled; rejects as soon as any promise is rejected.

**Promise.race():** Resolves/rejects as soon as the first promise settles (either resolved or rejected).

**Promise.allSettled():** Waits for all promises to settle and returns an array of results.

**Promise.any():** Resolves as soon as any promise is fulfilled; rejects only if all promises are rejected.

## Challenge

### Coding Challenge #3

Your tasks:

#### PART 1

1. Write an async function 'loadNPause' that recreates Challenge #2, this time using `async/await` (only the part where the promise is consumed, reuse the 'createImage' function from before)

2. Compare the two versions, think about the big differences, and see which one you like more

3. Don't forget to test the error handler, and to set the network speed to “Fast 3G” in the dev tools Network tab

## PART 2

1. Create an async function 'loadAll' that receives an array of image paths 'imgArr'
2. Use .map to loop over the array, to load all the images with the 'createImage' function (call the resulting array 'imgs')
3. Check out the 'imgs' array in the console! Is it like you expected?
4. Use a promise combinator function to actually get the images from the array 💎
5. Add the 'parallel' class to all the images (it has some CSS styles)

Test data Part 2: ['img/img-1.jpg', 'img/img-2.jpg', 'img/img3.jpg']. To test, turn off the 'loadNPause' function

```
//challenge 3
//part 1
const loadNPause = async function () {
  try {
    //1 load image 1
    let img = await createImage('img/img-1.jpg');
    console.log('Image 1 loaded');
    await wait(2);
    img.style.display = 'none';

    //load image 2
    img = await createImage('img/img-2.jpg');
    console.log('Image2 loaded');
    await wait(2);
    img.style.display = 'none';
  } catch (error) {
    console.error(error);
  }
};
loadNPause();
```

```
//Part 2
const loadAll = async function (imgArr) {
  try {
    const imgs = imgArr.map(async img => await createImage(img));
    console.log(imgs);
    const imgsEl = await Promise.all(imgs);
    console.log(imgsEl);
    imgsEl.forEach(img => img.classList.add('parallel'));
  } catch (error) {
    console.error(error);
  }
};
loadAll(['img/img-1.jpg', 'img/img-2.jpg', 'img/img-3.jpg']);
```

## Part 1: loadNPause

This function loads two images sequentially with a pause between loading each one. The process involves the following steps:

1. **Load Image 1:**
  - `let img = await createImage('img/img-1.jpg');` — This calls an asynchronous function `createImage` that returns a promise to load an image (`img/img-1.jpg`).
  - Once the image is loaded, it is assigned to the `img` variable, and `'Image 1 loaded'` is logged.
  - `await wait(2);` — A `wait` function is called to simulate a delay of 2 seconds before moving to the next step.
  - `img.style.display = 'none';` — The image is hidden by setting its `display` property to `'none'`.
2. **Load Image 2:**
  - The same steps are repeated for loading the second image (`img-2.jpg`).
  - Once it is loaded, `'Image 2 loaded'` is logged, followed by another 2-second delay, and then the image is hidden.
3. **Error Handling:**
  - If any error occurs in either part of the code (loading an image or waiting), it will be caught in the `catch` block and logged as an error.
4. **Function Invocation:**

- `loadNPause()`; — This function is called to execute the image loading and hiding sequence.

## Part 2: `loadAll`

This function loads multiple images in parallel (simultaneously) and then adds a CSS class to each image once they are all loaded. The steps are:

### 1. Mapping Image Paths:

- `const imgs = imgArr.map(async img => await createImage(img));` — For each image path in the `imgArr` array, an asynchronous `createImage` function is called, which returns an image promise. The result is an array of promises, stored in `imgs`.

### 2. Awaiting All Image Loads:

- `const imgsEl = await Promise.all(imgs);` — `Promise.all` is used to wait until all image promises have resolved, i.e., when all images are loaded. The resolved results (image elements) are stored in `imgsEl`.

### 3. Logging and Styling:

- `console.log(imgs);` — Logs the array of image promises (before they are resolved).
- `console.log(imgsEl);` — Logs the array of image elements (after they are resolved).
- `imgsEl.forEach(img => img.classList.add('parallel'));` — For each image element in `imgsEl`, the CSS class `'parallel'` is added, which could be used for styling the images (e.g., to arrange them in a grid or apply other visual effects).

### 4. Error Handling:

- If an error occurs during image loading, it will be caught in the `catch` block and logged.

### 5. Function Invocation:

- `loadAll(['img/img-1.jpg', 'img/img-2.jpg', 'img/img-3.jpg']);` — The function is called with an array of image paths to load (`img-1.jpg`, `img-2.jpg`, and `img-3.jpg`).