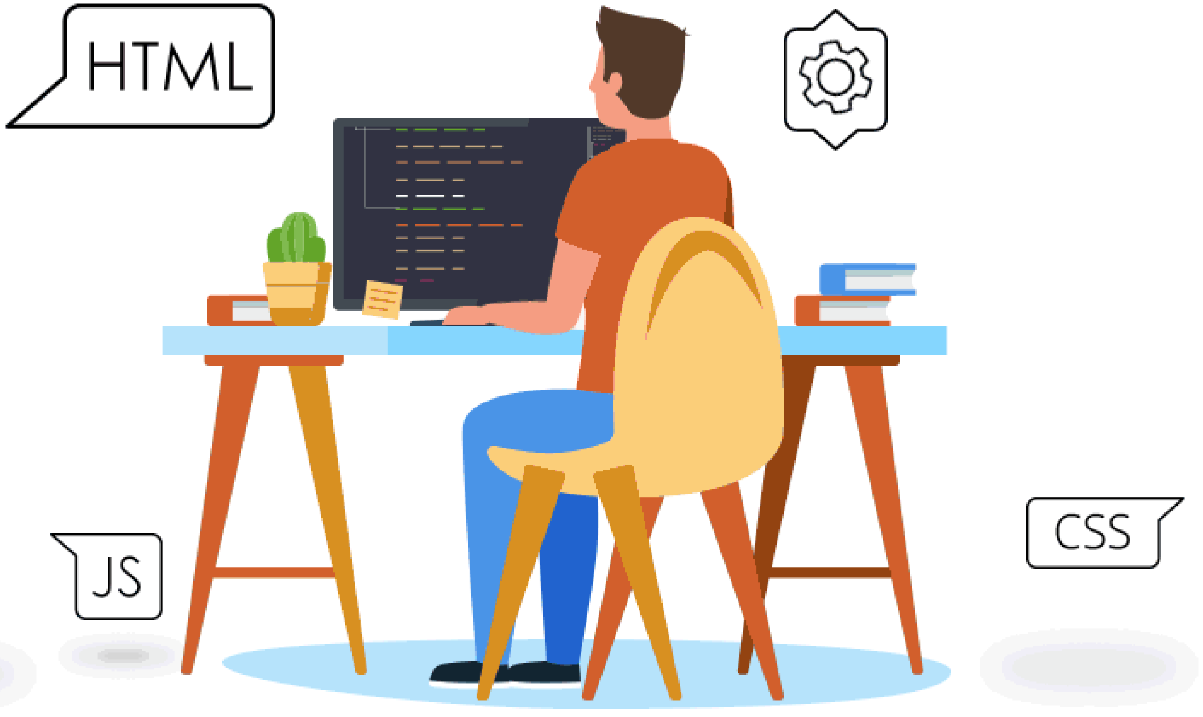
Session-04

Asynchronus Javascript

Thanos is on a mission to make his website standout from his rest of universe with Javascript pesto

Web Fundamentals



Session-04

Asynchronus Javascript

Agenda : Asynchronus Javascript

01 Sync Vs Async

02 JS Event Loop and Task Queue

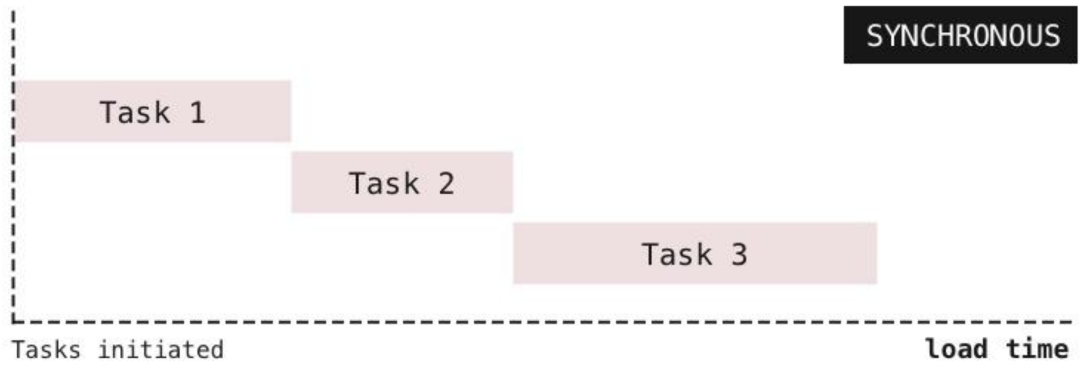
03 SetTimeout and SetInterval

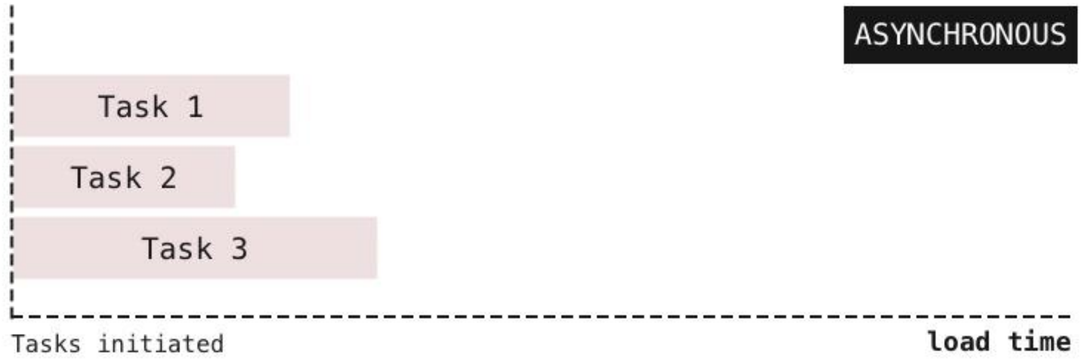
04 Callbacks

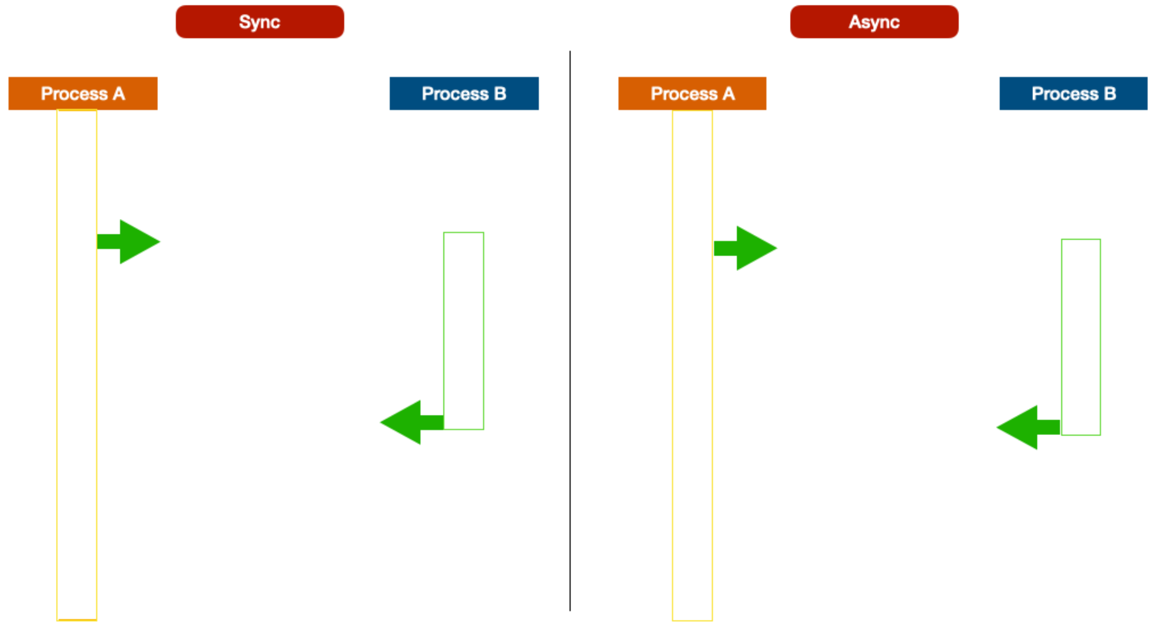
05 Promises, Async/Await

06 Hands-On

Synchronous Vs Asynchronous

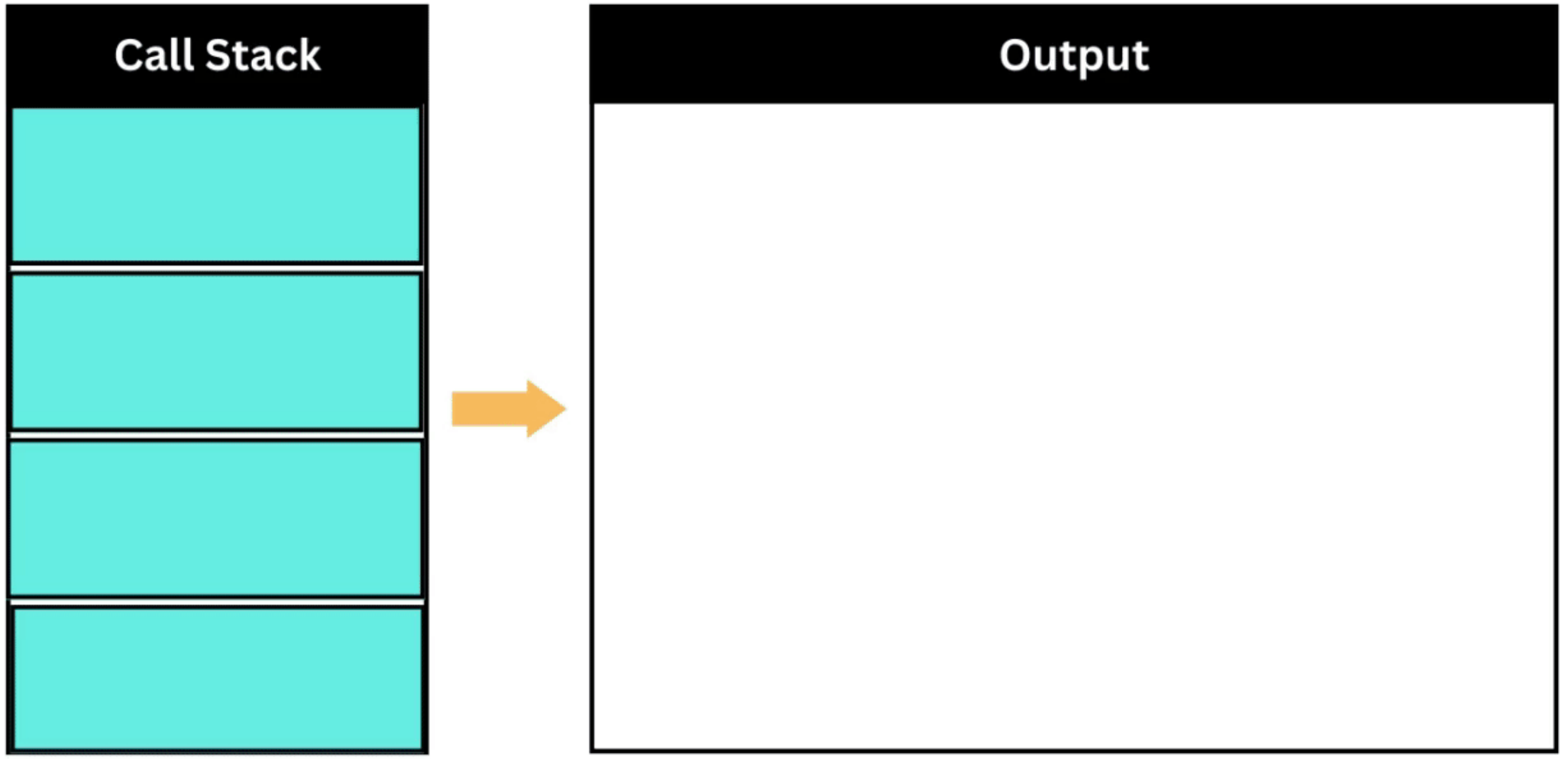






Asynchronous programming is key for Performance and Load time of any application

JavaScript is **single-threaded programming** language

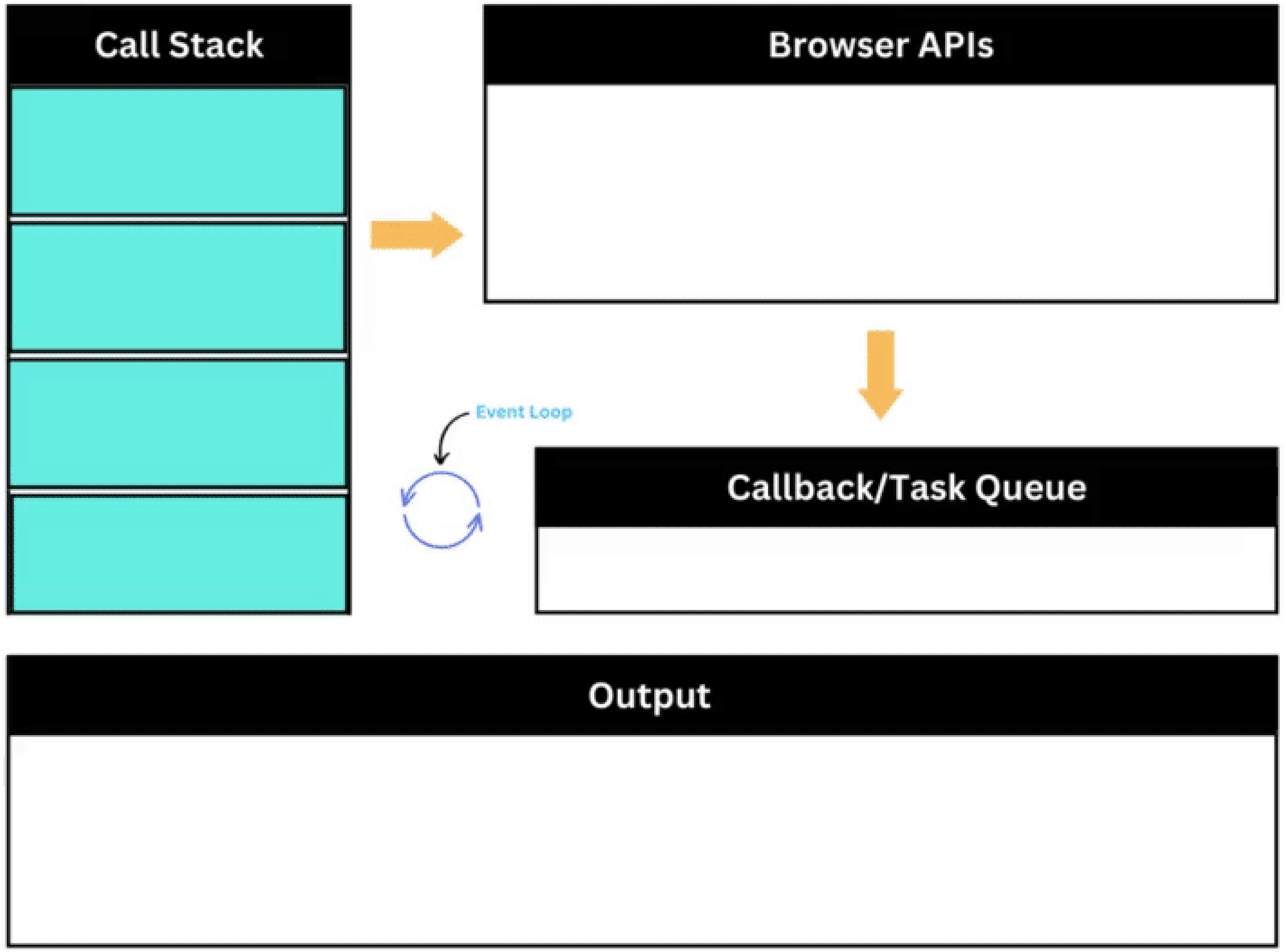


Then How JS handles Asynchronous programming??

Event Loop/ Callback Queue – The magic

Asynchronous programming in JavaScript : Instead of waiting for a task to complete,

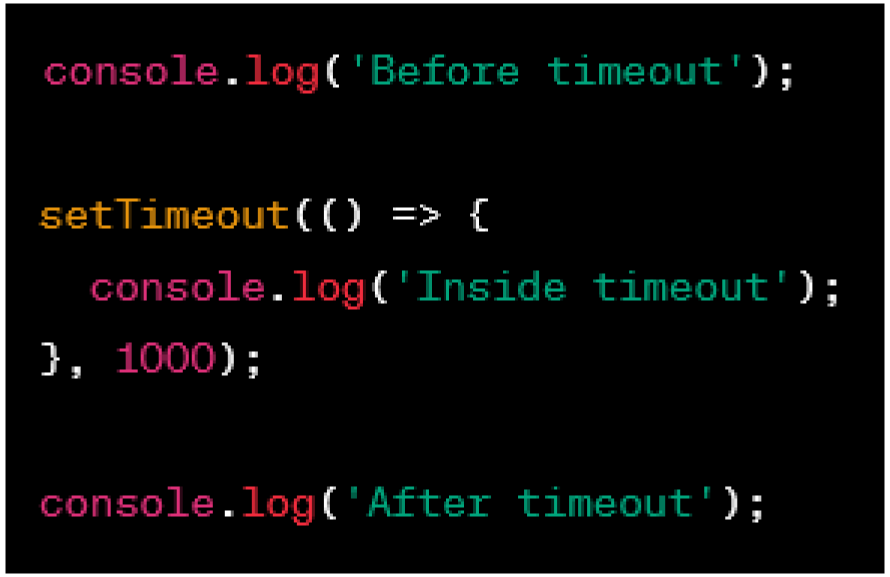
JavaScript engine handle the task **in the background** using **Event Loop**



setTimeout ( ) - Using Timers for Asynchronous JS

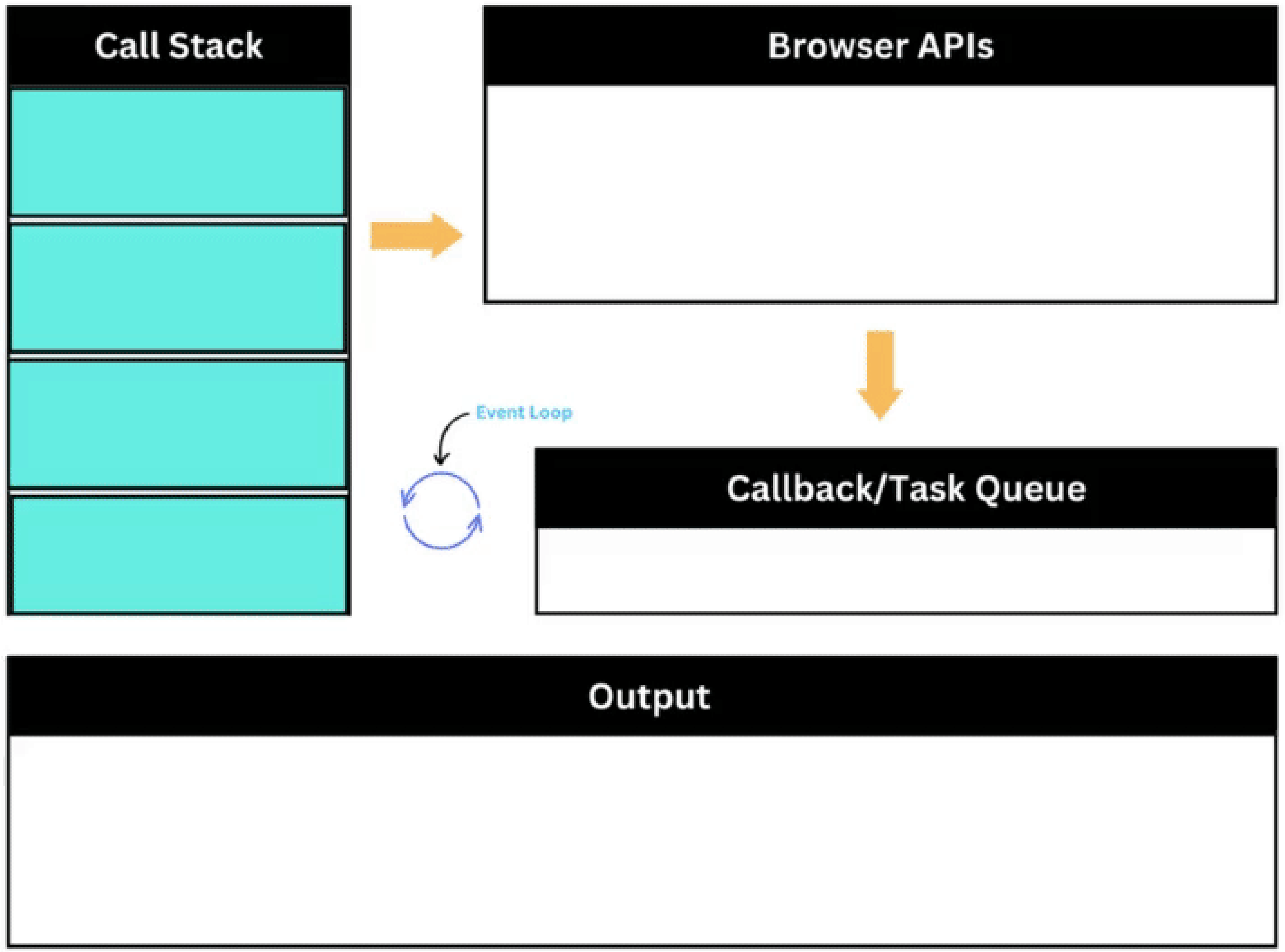
setTimeout(), to perform asynchronous operations in JavaScript.

While the timer is counting down, **other tasks can continue to execute on the main thread**



```
console.log('Before timeout');

setTimeout(() => {
  console.log('Inside timeout');
}, 1000);

console.log('After timeout');
```

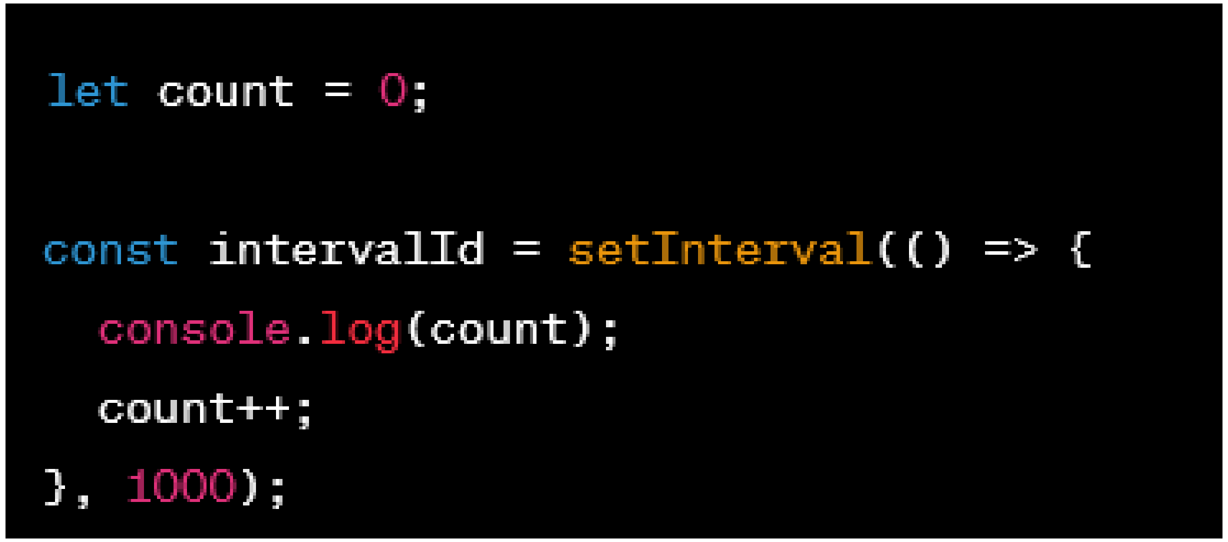
setTimeout ( ) – Execution in Asynchronous JS



setInterval ( ) – Repeatedly running code
Asynchronous JS

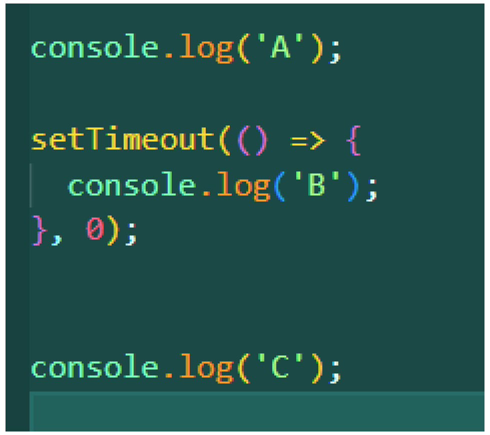
setInterval ( )- is a built-in method that allows us to repeatedly execute a function at a specified interval

While the interval is running, **other tasks can continue to execute on the main thread**

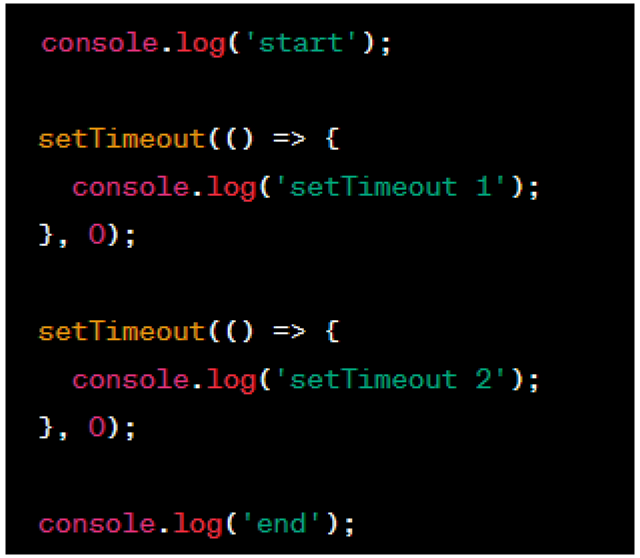


```
let count = 0;

const intervalId = setInterval(() => {
  console.log(count);
  count++;
}, 1000);
```

Knowledge check: What is the output?



```
console.log('A');

setTimeout(() => {
  console.log('B');
}, 0);


console.log('C');
```



```
console.log('start');

setTimeout(() => {
  console.log('setTimeout 1');
}, 0);

setTimeout(() => {
  console.log('setTimeout 2');
}, 0);

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

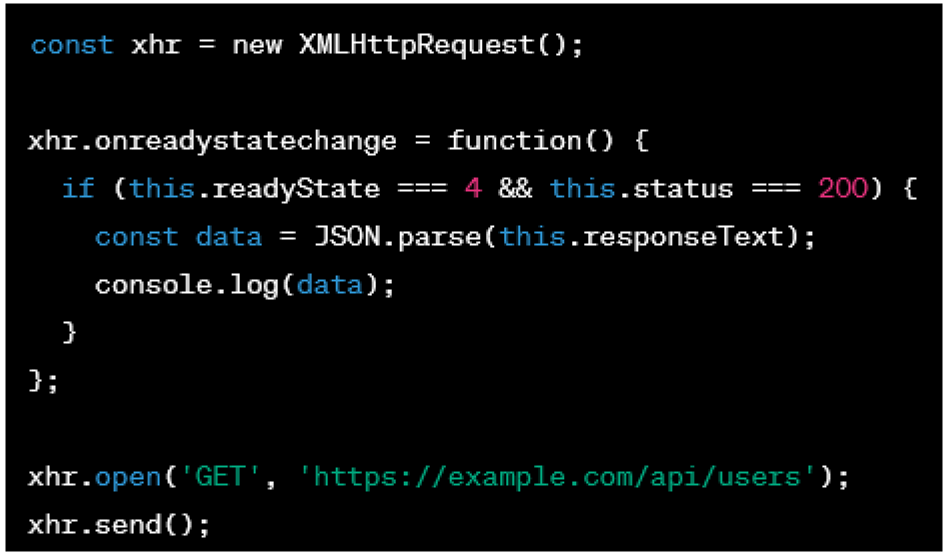
A C B
// setTimeOut- Async code goes to callback queue and event loop

Ans: start end setTimeout 1 setTimeout 2
// sync code executes first

XMLHTTPRequest in Asynchronous Programming

-XMLHttpRequest (XHR) is a browser API that allows us to send and receive HTTP requests and responses asynchronously.

-It is a core component of AJAX (Asynchronous JavaScript and XML)

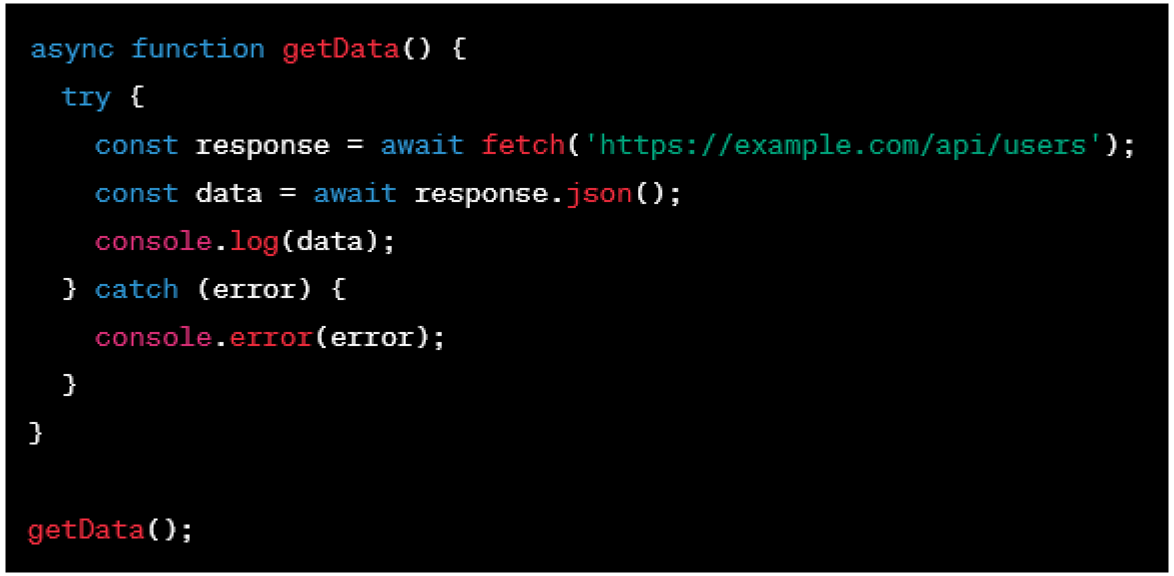


```
const xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
  if (this.readyState === 4 && this.status === 200) {
    const data = JSON.parse(this.responseText);
    console.log(data);
  }
};

xhr.open('GET', 'https://example.com/api/users');
xhr.send();
```

Knowledge Check: Since this is async, does it execute on CallStack or uses Event Loop?

Fetch API – Making Async Requests (file, data, image)

Fetch API is a built-in method to make **asynchronous HTTP requests** to retrieve data from a server.

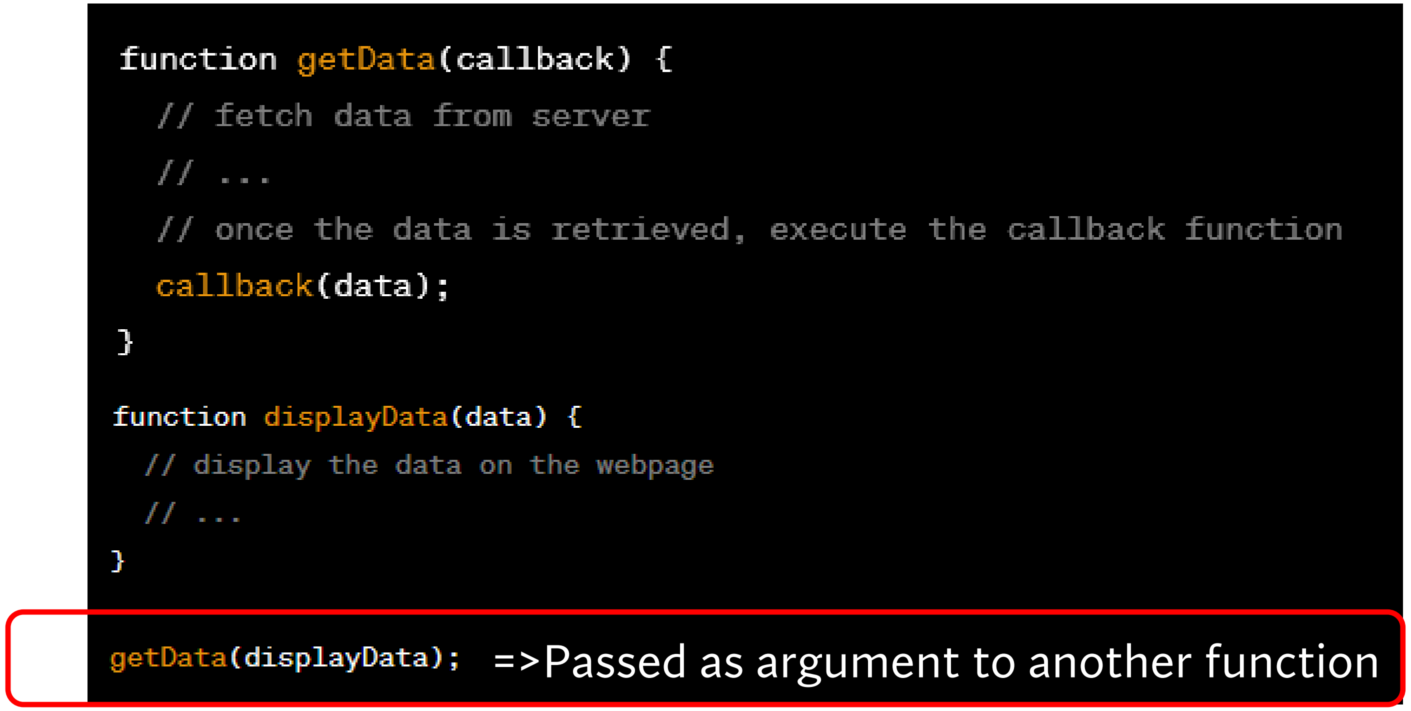


```
async function getData() {
  try {
    const response = await fetch('https://example.com/api/users');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

getData();
```

Knowledge Check: Since this is async, does it execute on CallStack or uses Event Loop?

Callbacks – Traditional Async JS

A callback is a function that is passed as **an argument to another function** and is executed once the first function has completed its task.



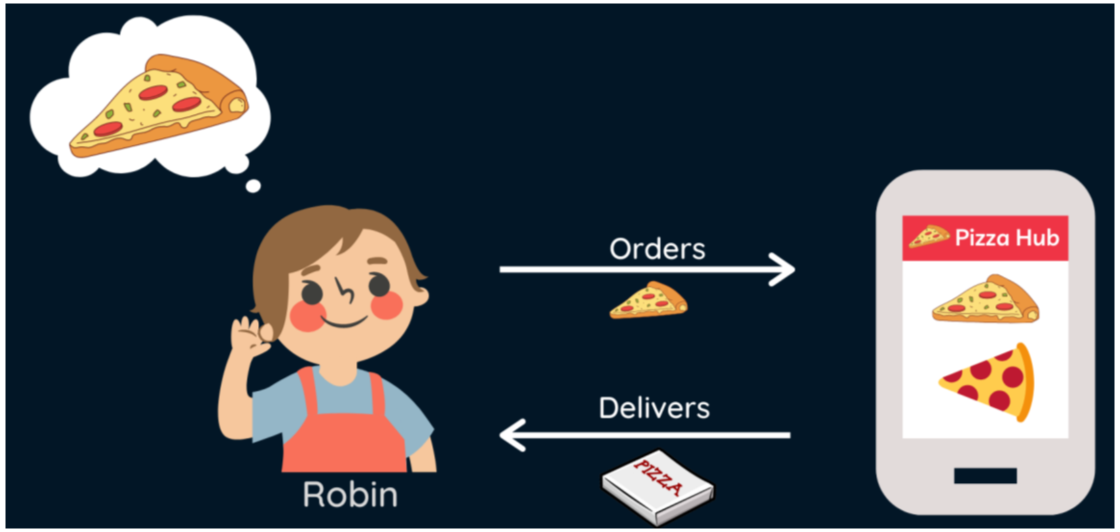
```
function getData(callback) {
  // fetch data from server
  // ...
  // once the data is retrieved, execute the callback function
  callback(data);
}

function displayData(data) {
  // display the data on the webpage
  // ...
}

getData(displayData);
```

=>Passed as argument to another function

Knowledge Check: Since this is async, does it execute on CallStack or uses Event Loop?

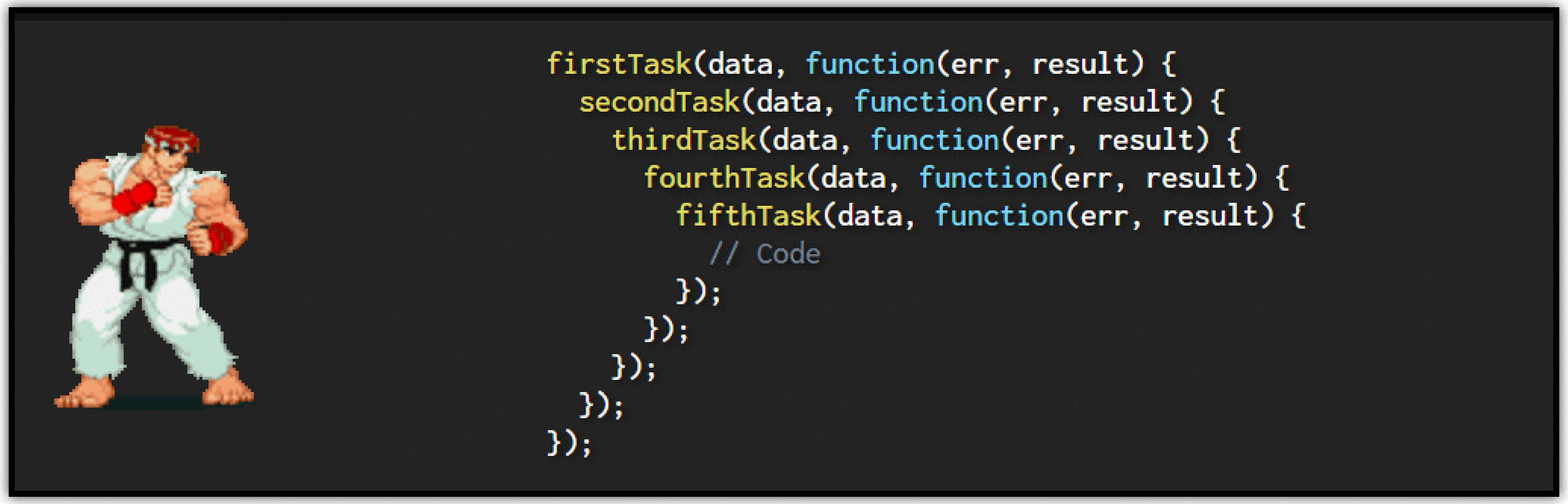
Callbacks Simplified– A Delivery boy

A **callback is like a delivery boy**. You place an order (call a function), and when the food (response) is ready, the delivery boy (callback function) brings it to your doorstep (code execution)



Callback Hell

Callback Hell is a problem in asynchronous programming, where we end up with deeply nested callbacks that **make code difficult to read and maintain**

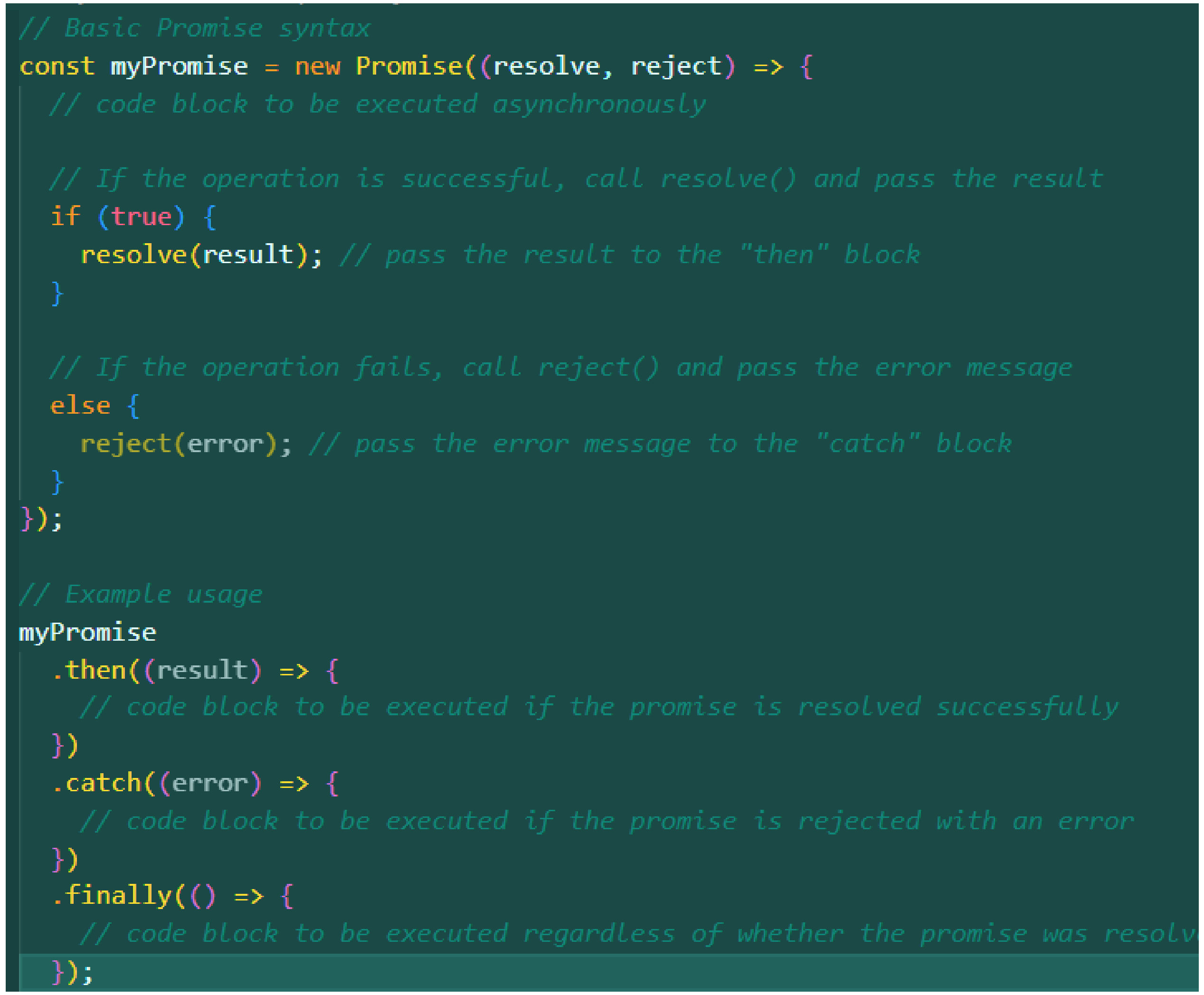


```
firstTask(data, function(err, result) {
  secondTask(data, function(err, result) {
    thirdTask(data, function(err, result) {
      fourthTask(data, function(err, result) {
        fifthTask(data, function(err, result) {
          // Code
        });
      });
    });
  });
});
```

Hence the existence of Promises in ES6!!

Promises- A Modern Approach to Asynchronous JS

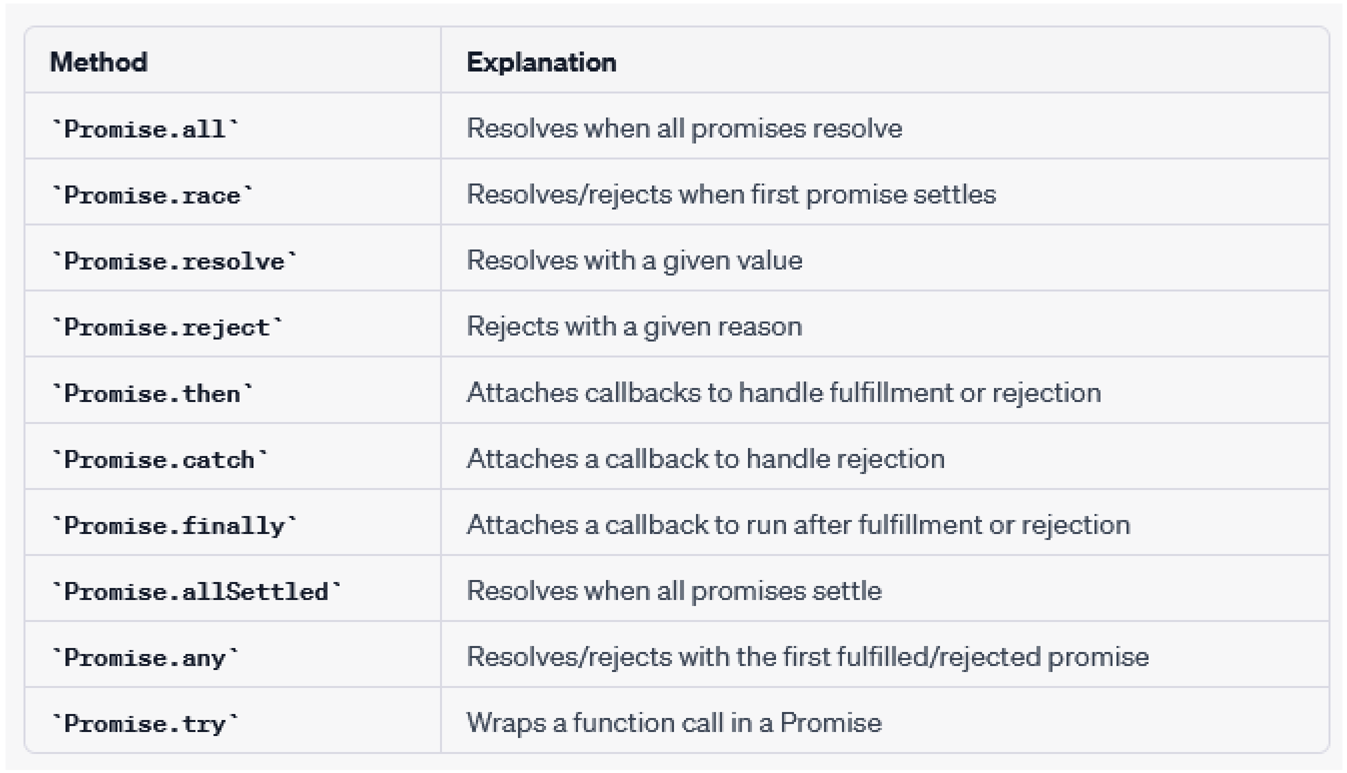
- **A promise is an object** that represents the eventual completion (or failure) of an asynchronous operation.
- A promise can be in one **of three states: pending, fulfilled, or rejected**.
- It allows you to **chain multiple asynchronous operations together** in a **more readable and maintainable** way.



```
// Basic Promise syntax
const myPromise = new Promise((resolve, reject) => {
  // code block to be executed asynchronously

  // If the operation is successful, call resolve() and pass the result
  if (true) {
    resolve(result); // pass the result to the "then" block
  }

  // If the operation fails, call reject() and pass the error message
  else {
    reject(error); // pass the error message to the "catch" block
  }
});

// Example usage
myPromise
  .then((result) => {
    // code block to be executed if the promise is resolved successfully
  })
  .catch((error) => {
    // code block to be executed if the promise is rejected with an error
  })
  .finally(() => {
    // code block to be executed regardless of whether the promise was resolv
  });
```

Promise Methods

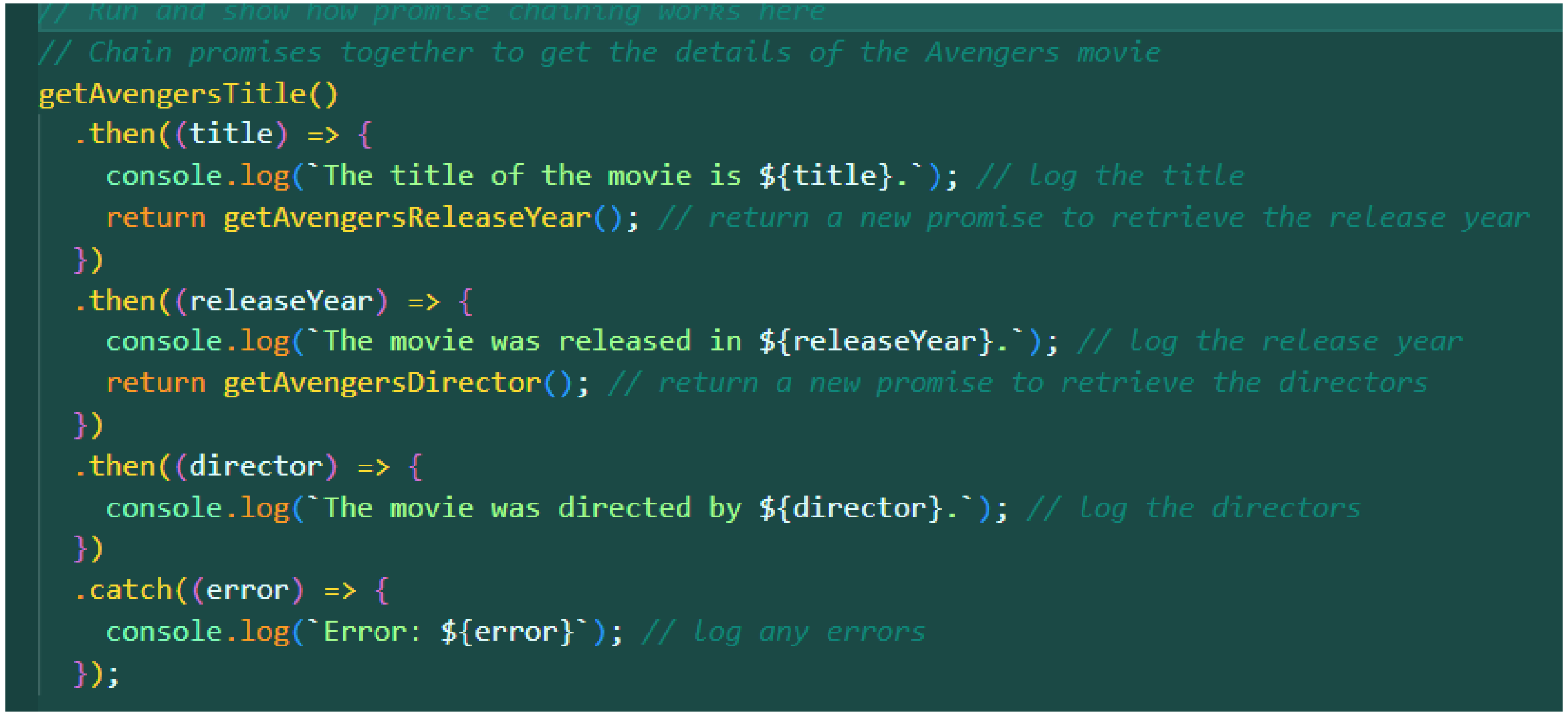
JavaScript **Promises** provide **several methods that can be used to handle the outcome** of an asynchronous operation



Method	Explanation
`Promise.all`	Resolves when all promises resolve
`Promise.race`	Resolves/rejects when first promise settles
`Promise.resolve`	Resolves with a given value
`Promise.reject`	Rejects with a given reason
`Promise.then`	Attaches callbacks to handle fulfillment or rejection
`Promise.catch`	Attaches a callback to handle rejection
`Promise.finally`	Attaches a callback to run after fulfillment or rejection
`Promise.allSettled`	Resolves when all promises settle
`Promise.any`	Resolves/rejects with the first fulfilled/rejected promise
`Promise.try`	Wraps a function call in a Promise

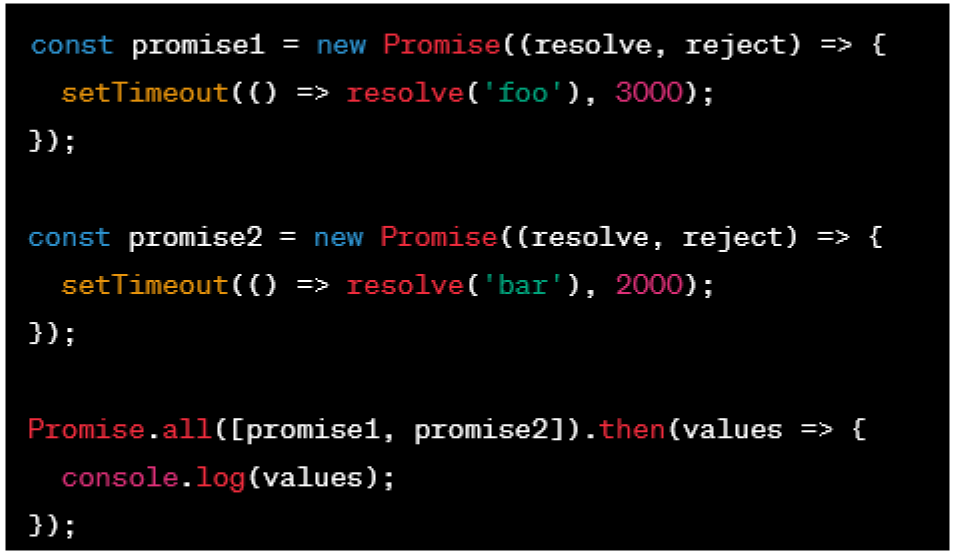
Promise Chaining

- **Promises can be chained together**, allowing for more complex asynchronous operations to be **performed in a readable and maintainable**
- The **output of one promise is passed as the input to the next promise**, allowing for a series of operations to be performed in sequence.



```javascript
// Run and show how promise chaining works here
// Chain promises together to get the details of the Avengers movie
getAvengersTitle()
  .then((title) => {
    console.log(`The title of the movie is ${title}.`); // log the title
    return getAvengersReleaseYear(); // return a new promise to retrieve the release year
  })
  .then((releaseYear) => {
    console.log(`The movie was released in ${releaseYear}.`); // log the release year
    return getAvengersDirector(); // return a new promise to retrieve the directors
  })
  .then((director) => {
    console.log(`The movie was directed by ${director}.`); // log the directors
  })
  .catch((error) => {
    console.log(`Error: ${error}`); // log any errors
  });
```

Knowledge check: What is the output?

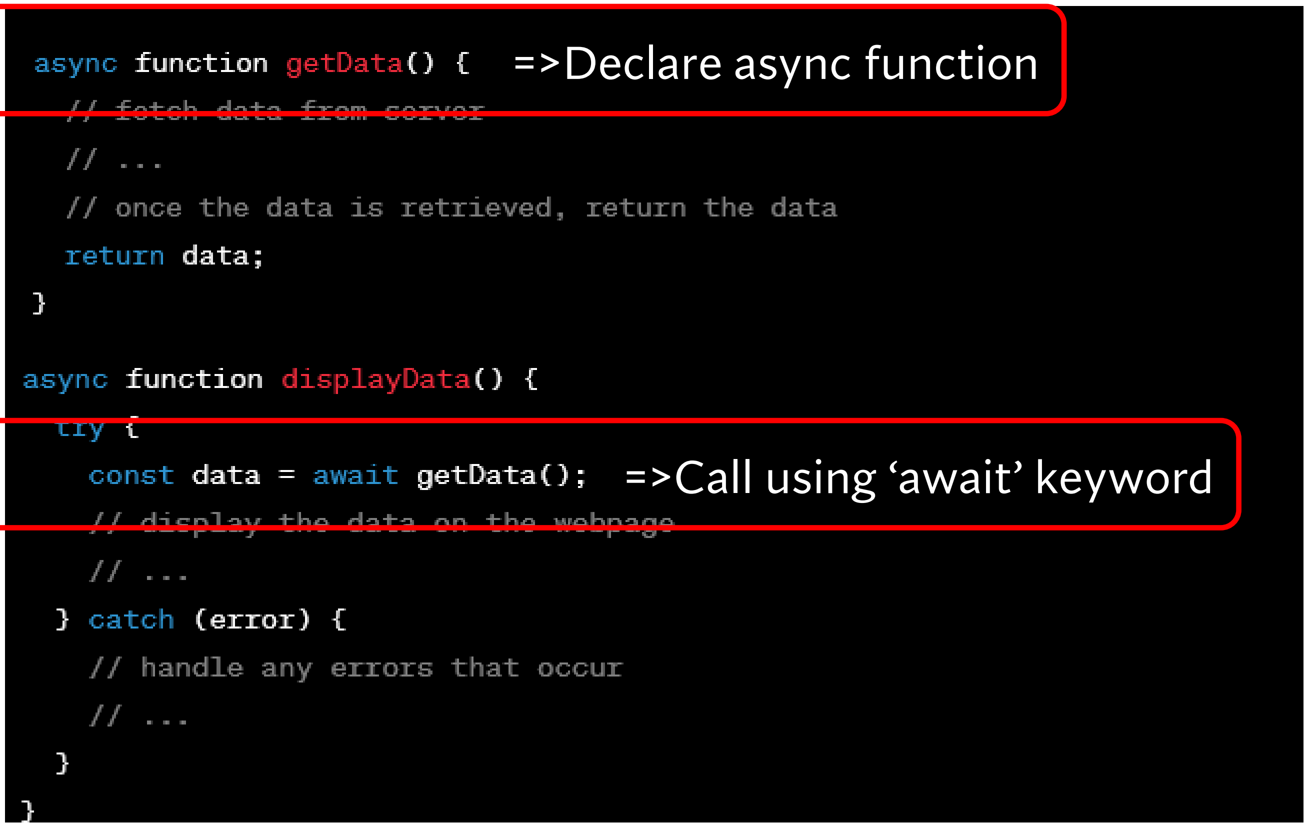


```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('foo'), 3000);
});

const promise2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('bar'), 2000);
});

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

Ans: [foo, bar]
// Always prints in the order of
they called

Async/Await- The Modern Standard for Asynchronous JavaScript

- Async/await is built on top of JS promises
- the most modern approach to asynchronous programming in JavaScript and is now the stand
  for writing asynchronous code



```
 async function getData() {
   // fetch data from server
   // ...
   // once the data is retrieved, return the data
   return data;
 }

async function displayData() {
  try {
    const data = await getData();
    // display the data on the webpage
    // ...
  } catch (error) {
    // handle any errors that occur
    // ...
  }
}
```

=>Declare async function

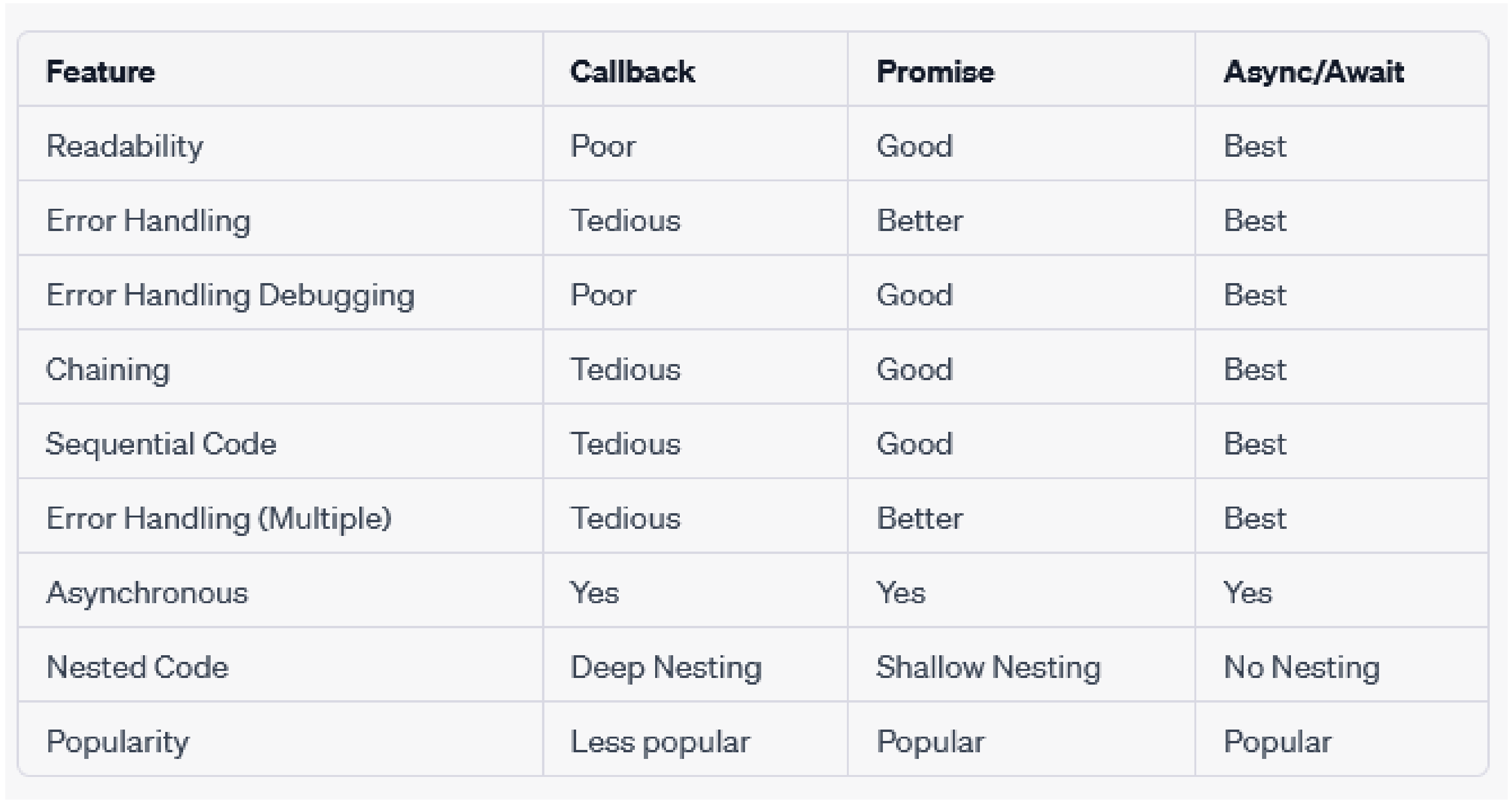
=>Call using ‘await’ keyword

Combine Async/Await and Promise

Async/await and Promises can be combined to make JS more powerful

```
async function getAvengersDirector() {
  return new Promise(resolve => {
    setTimeout(() => {
      const director = "Anthony Russo and Joe Russo";
      resolve(director);
    }, 4000);
  });
}

// Call all three functions in parallel using Promise.all()
async function getAvengersDetails() {
  const titlePromise = getAvengersTitle();
  const releaseYearPromise = getAvengersReleaseYear();
  const directorPromise = getAvengersDirector();

  const [title, releaseYear, director] = await Promise.all([
    titlePromise,
    releaseYearPromise,
    directorPromise,
  ]);
```

Callback Vs Promises Vs Async



Feature	Callback	Promise	Async/Await
Readability	Poor	Good	Best
Error Handling	Tedious	Better	Best
Error Handling Debugging	Poor	Good	Best
Chaining	Tedious	Good	Best
Sequential Code	Tedious	Good	Best
Error Handling (Multiple)	Tedious	Better	Best
Asynchronous	Yes	Yes	Yes
Nested Code	Deep Nesting	Shallow Nesting	No Nesting
Popularity	Less popular	Popular	Popular

Use Cases: Callback Vs Promises Vs Async

1.Callbacks: execute a function when another function has finished

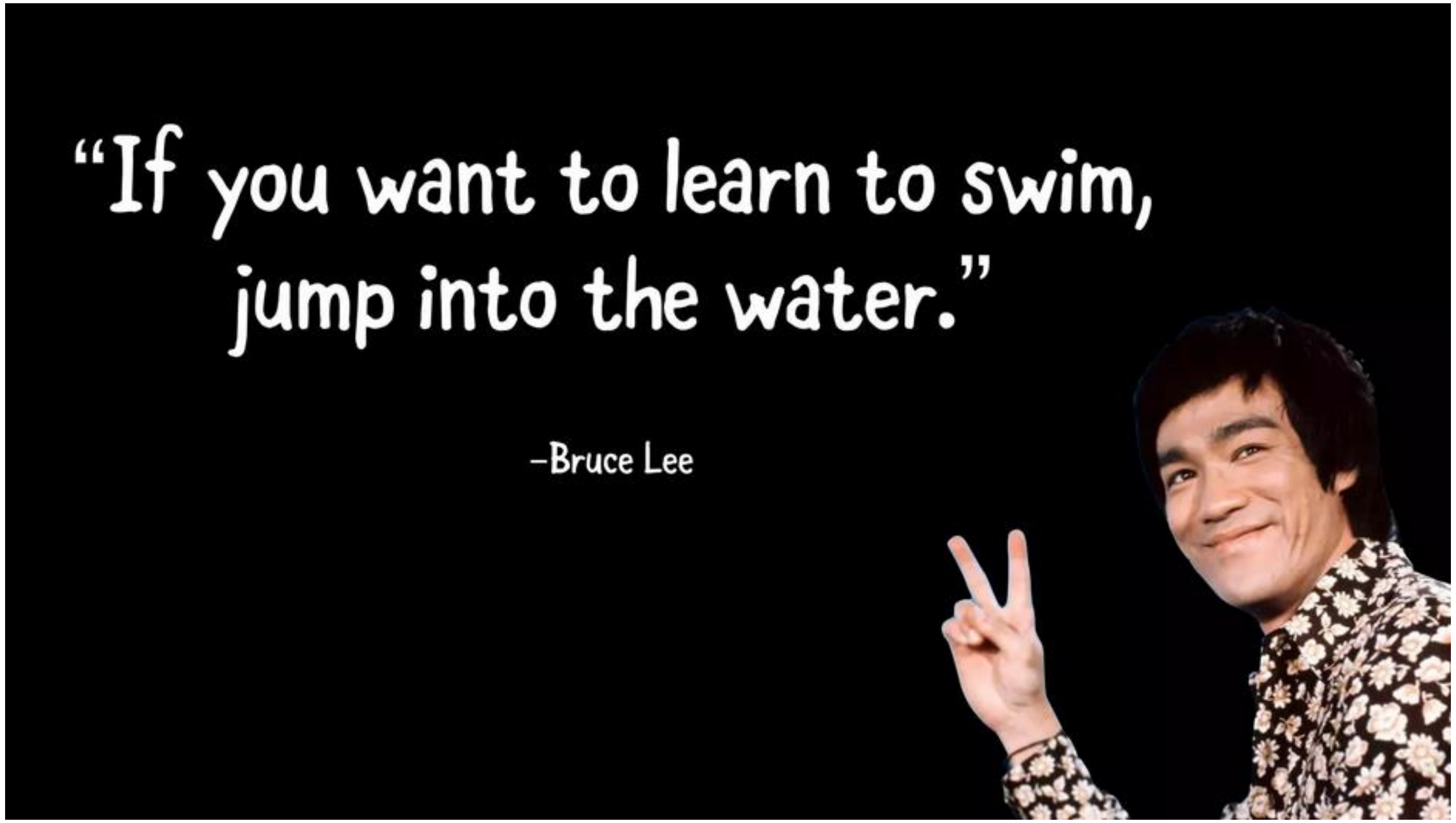
1. Handling user input events in a web page (e.g. button clicks)
2. Reading and writing files in a Node.js server

2.Promises: when we need to perform an asynchronous operation and want to handle its result when it is ready.

1. Fetching data from an API and updating the UI when it's ready
2. Loading multiple resources asynchronously, such as images or scripts

3.Async/Await: is useful when we want to write asynchronous code that looks like synchronous code.

1. Making multiple HTTP requests in a sequence, such as login and fetching user data
2. Waiting for a user action to resolve a Promise, such as filling in a form and submitting it.


“If you want to learn to swim, jump into the water.”

–Bruce Lee

Q & A