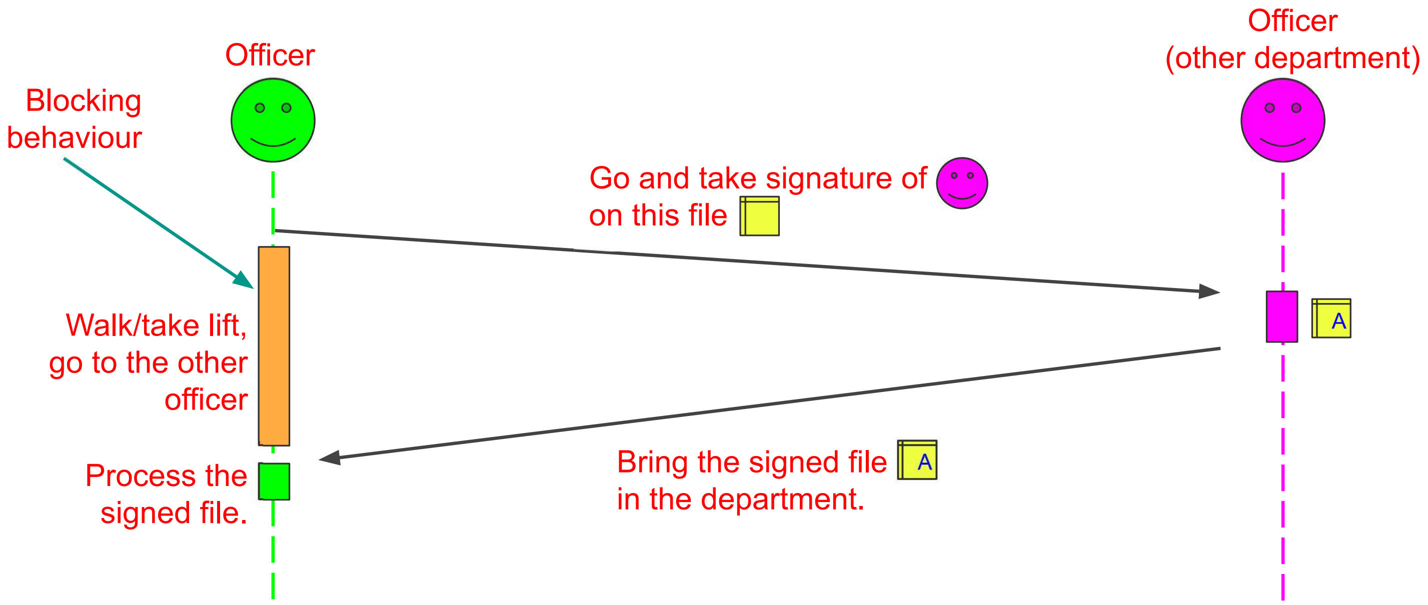
Advanced Concepts: Promise, Async, Await

Dr Harshad Prajapati
26 Nov 2023

Synchronous vs Asynchronous Call

Analogy of Blocking Work of Officer



Officer

Officer
(other department)

Blocking
behaviour

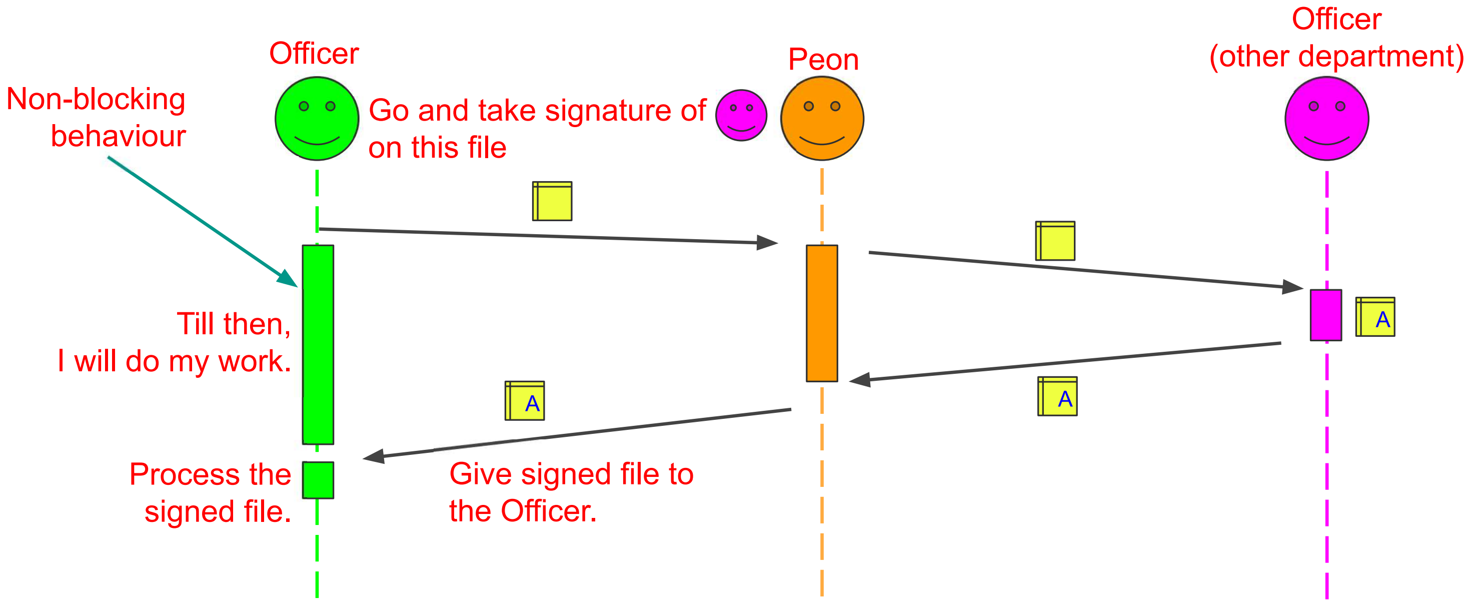
Go and take signature of
on this file

Walk/take lift,
go to the other
officer

Process the
signed file.

Bring the signed file
in the department.

Analogy of Non-blocking Work of Officer



Officer

Peon

Officer
(other department)

Non-blocking
behaviour

Go and take signature of
on this file

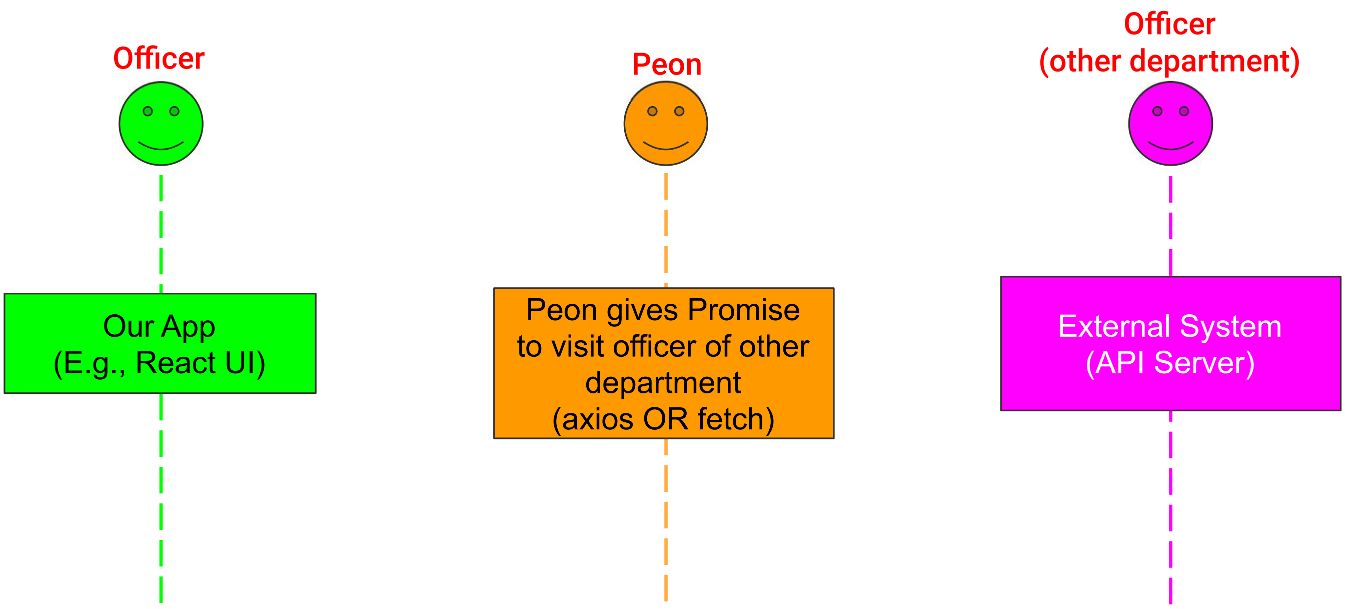
Till then,
I will do my work.

Process the
signed file.

Give signed file to
the Officer.

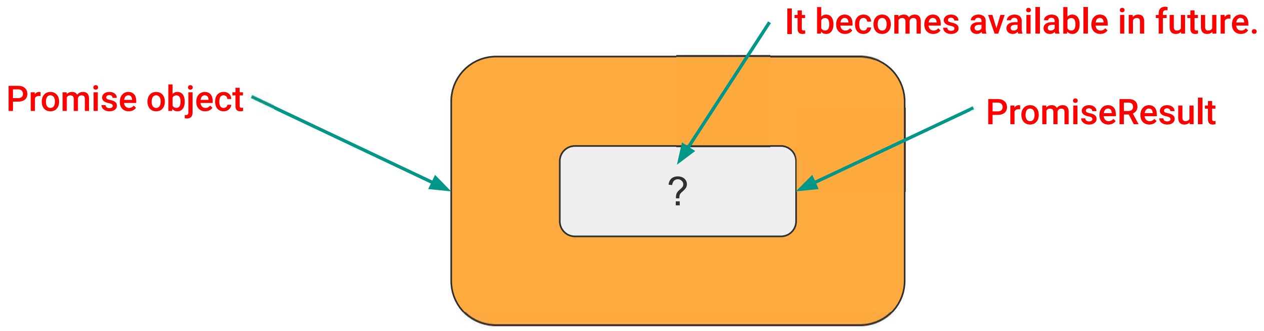
✓ Blocking call is called Synchronous Call
✓ Non-blocking call is Asynchronous Call

Analogy of non-blocking Work of Officer



Promise is an Object

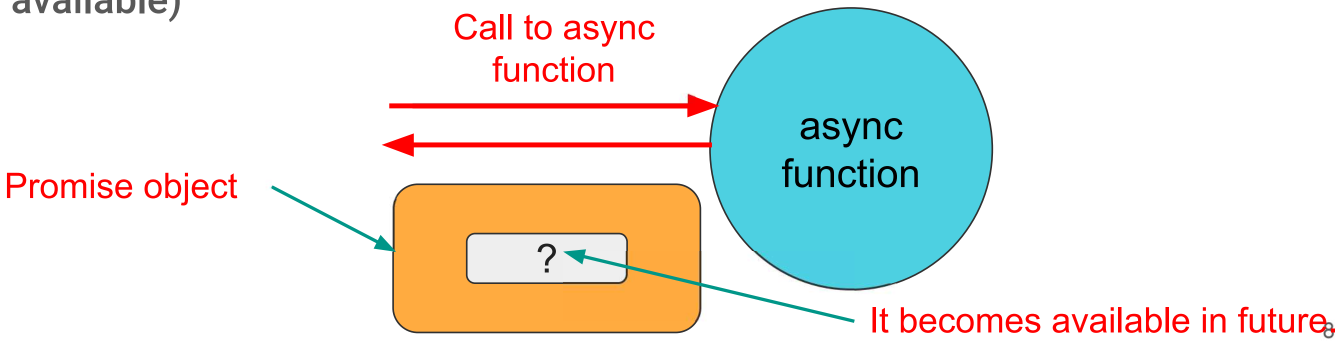
- Promise is an object that represents future event.
- Promise is a proxy for a value not necessarily known when the promise is created.
    - Promise is a wrapper (box) on a value that will come in future.



Analogy: Father promises his son that if you score more than 80% in 12th Science exam, I will get you a motorcycle.

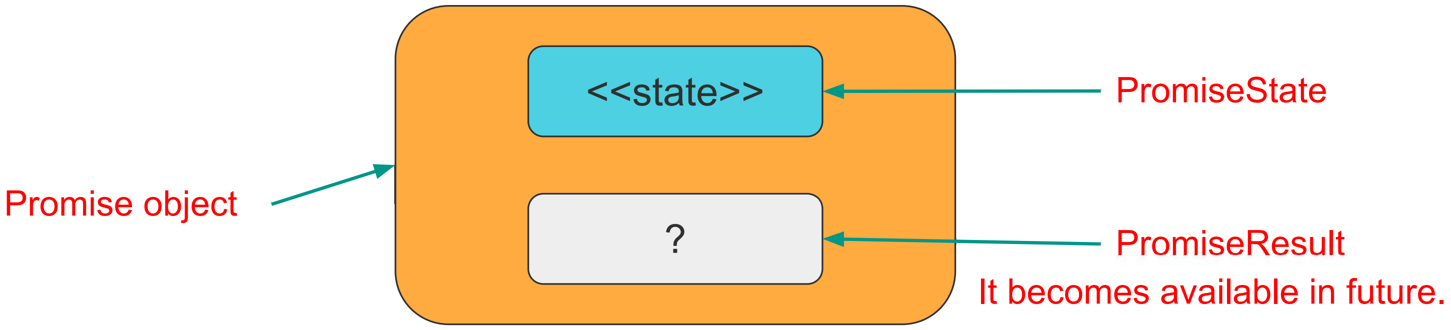
Promise as a Return Value of Async Call

- Asynchronous (async) methods can return values like synchronous methods return.
    - Since the result is not available immediately, an asynchronous method returns a promise.
    - The Promise promises to provide result in future (whenever, it becomes available)



Promise Object Contains State also

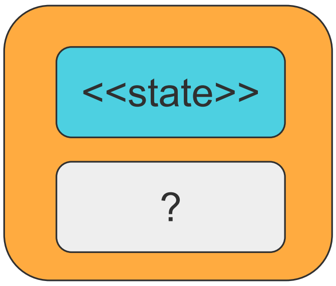
- **Promise** object also contains **PromiseState** variable that represents **current state** of the promise object.



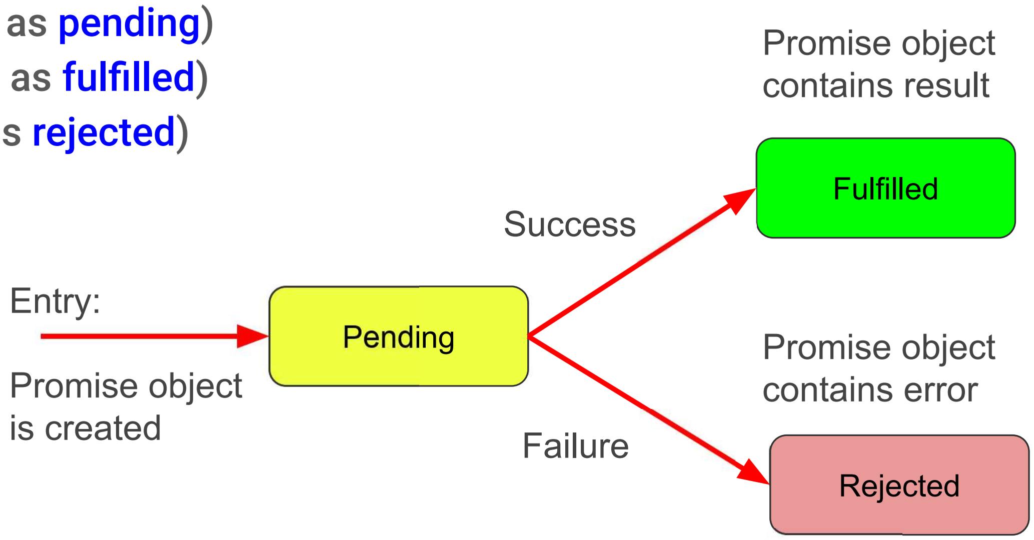
It becomes available in future.

States of Promise

- There are **three** possible **states** of a promise object:
  - Pending (named as **pending**)
  - Success (named as **fulfilled**)
  - Failure (named as **rejected**)

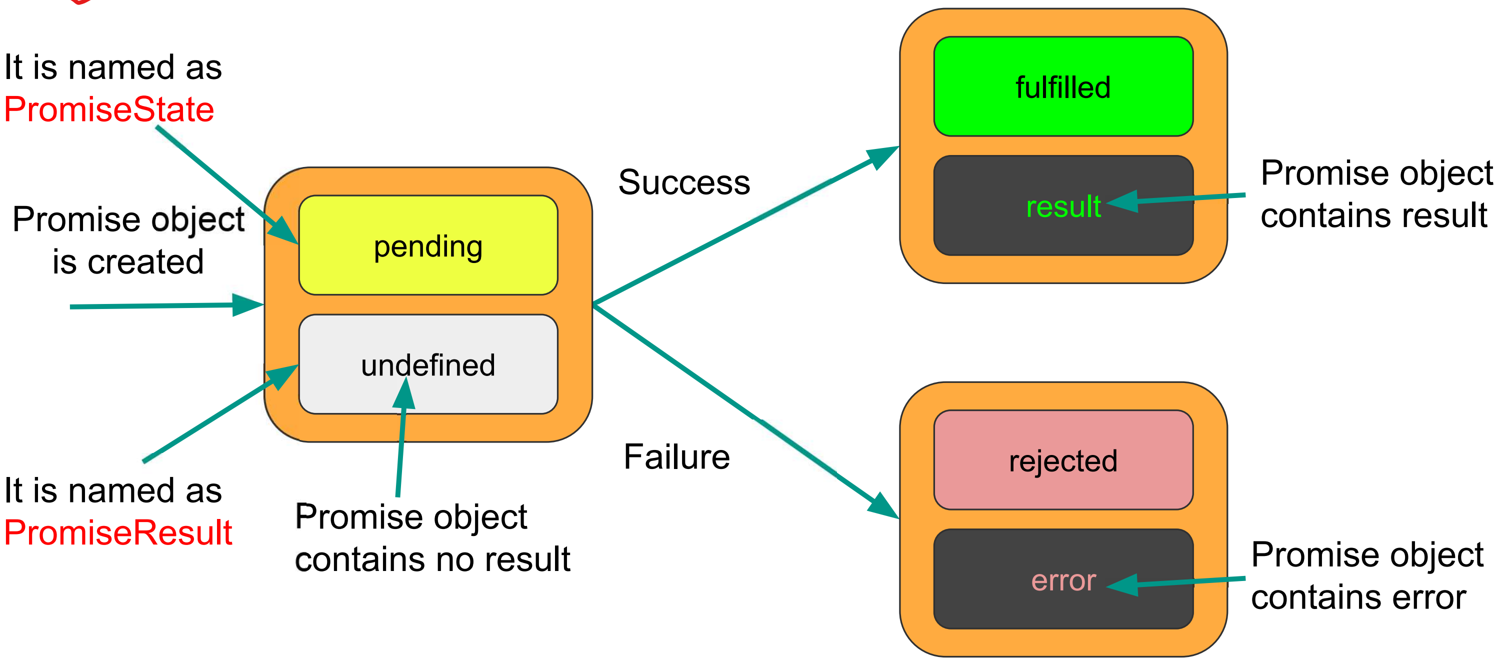


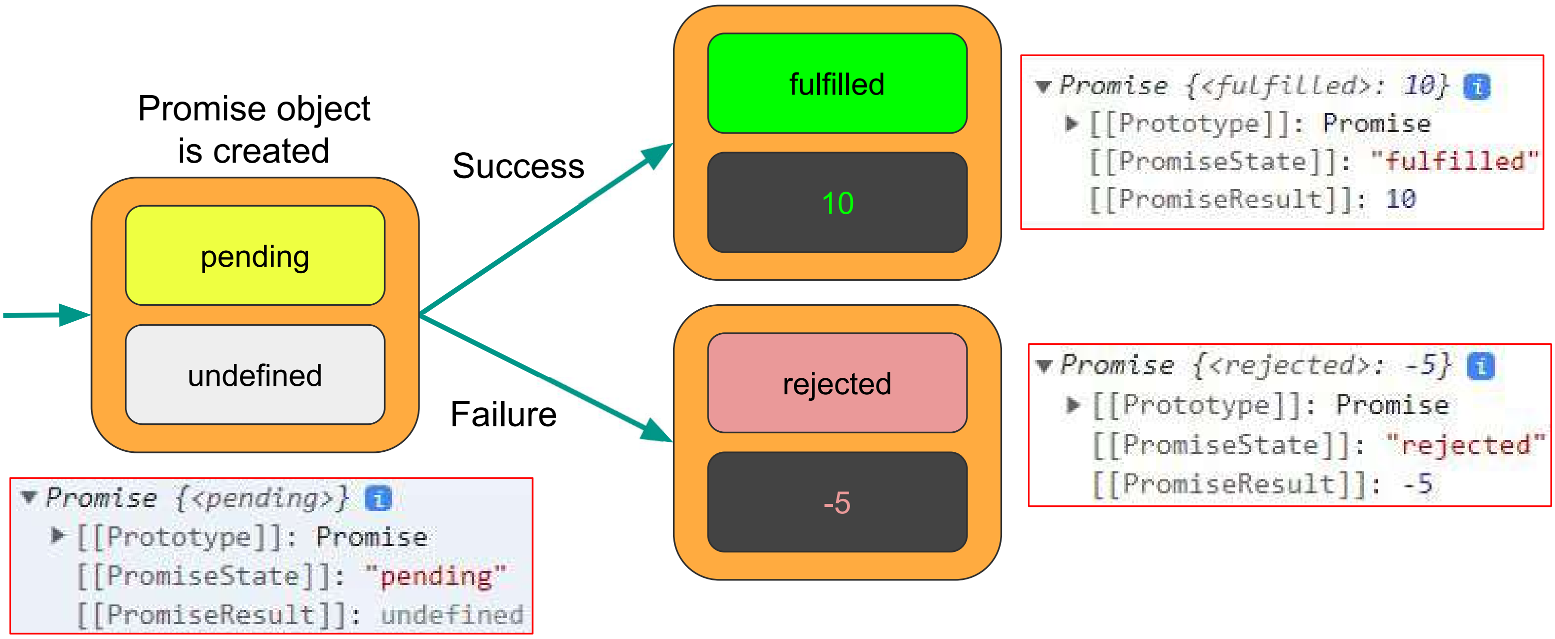
Promise object


Entry:
Promise object is created
Pending
Success
Failure
Promise object contains result
Fulfilled
Promise object contains error
Rejected

Three possible states of Promise object

Value of PromiseResult Property (Result of Promise)


It is named as PromiseState
Promise object is created
pending
undefined
It is named as PromiseResult
Promise object contains no result
Success
Failure
fulfilled
result
Promise object contains result
rejected
error
Promise object contains error



```
▼ Promise {<pending>}
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

```
▼ Promise {<fulfilled>: 10}
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 10
```

```
▼ Promise {<rejected>: -5}
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: -5
```

How to Create a Promise

How to Create a Promise

- There are **two ways** we can create a promise?
  - Implicitly
  - Explicitly
- **Implicitly**:
  - **Mark** a **function** as **async**.
- **Explicitly**:
  - **Create** and **return Promise object**.
    - Promise.resolve() or Promise.reject().
    - Promise() constructor.

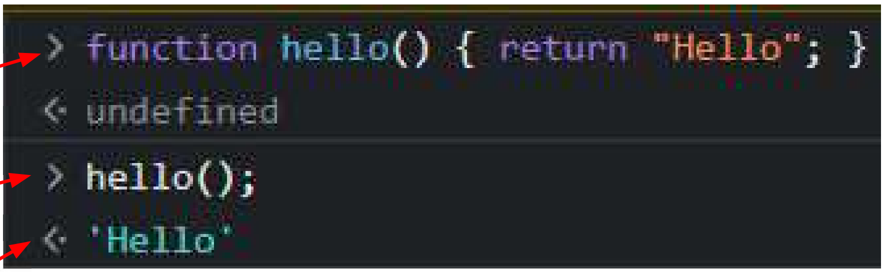
How to Create a Promise Implicitly?

- First, we understand behaviour of a normal (synchronous) function.
- We write and call a normal function in console of Web Browser.
  - Console is available under Developer tools.

Create a normal function.

Call the function.

Return value of the function.



```
> function hello() { return "Hello"; }
< undefined
> hello();
< 'Hello'
```

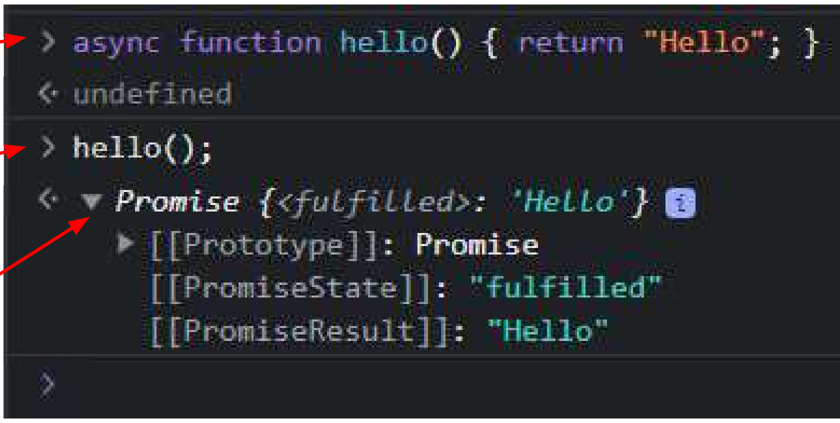
How to Create a Promise Implicitly?

- Now, we write and call an async function in console of Web Browser.

Create an async function.

Call the async function.

Return value of an async function is a promise object.

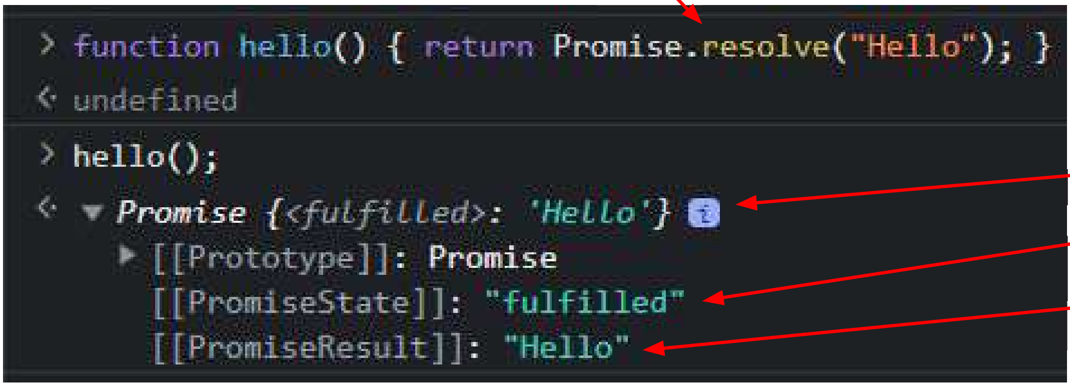


```
> async function hello() { return "Hello"; }
< undefined
> hello();
< ▼ Promise {<fulfilled>: 'Hello'}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "Hello"
>
```

How to Create a Promise Explicitly?

- We **explicitly return** a **promise object**.

Explicitly return a Promise object.



```
> function hello() { return Promise.resolve("Hello"); }
< undefined
> hello();
< ▼ Promise {<fulfilled>: 'Hello'} ℹ
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "Hello"
```

Promise object

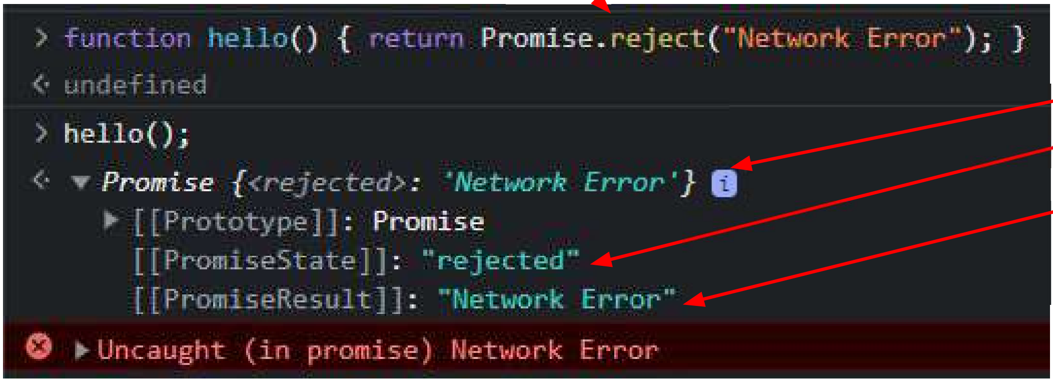
Promise state

Promise error

How to Create a Promise Explicitly?

- We **explicitly return** a **promise object**.

Explicitly return a Promise object.



```
> function hello() { return Promise.reject("Network Error"); }
< undefined
> hello();
< ▼ Promise {<rejected>: 'Network Error'} ℹ
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "rejected"
      [[PromiseResult]]: "Network Error"
⊗ ▶Uncaught (in promise) Network Error
```

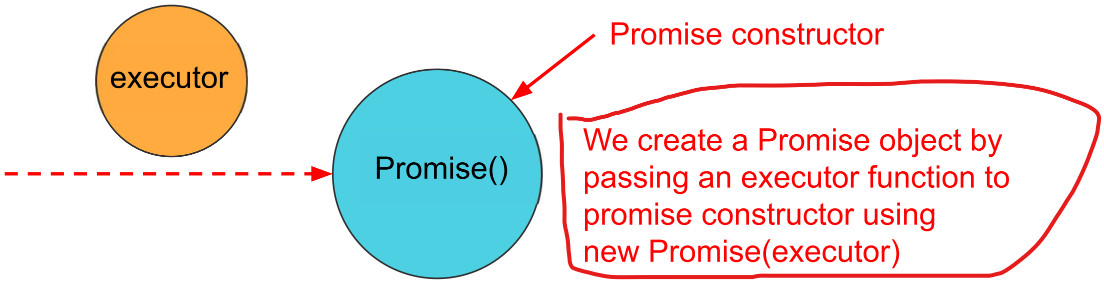
Promise object

Promise state

Promise error

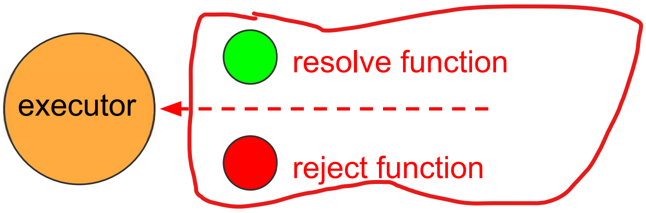
How to Create a Promise Explicitly using constructor?

- Input to Promise constructor is executor function.



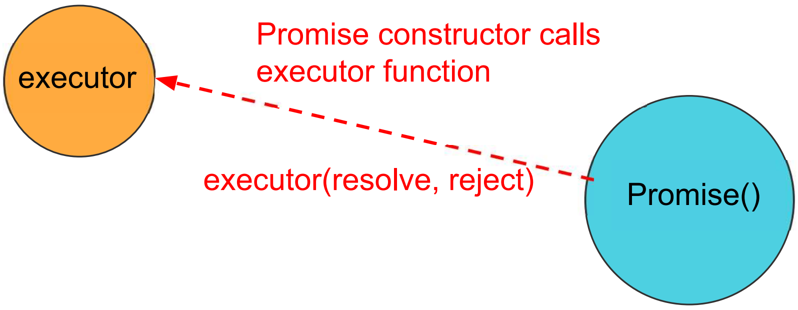
We create a Promise object by passing an executor function to promise constructor using new Promise(executor)

Form of executor function



How to Create a Promise Explicitly using constructor?

- What does the Promise constructor do?

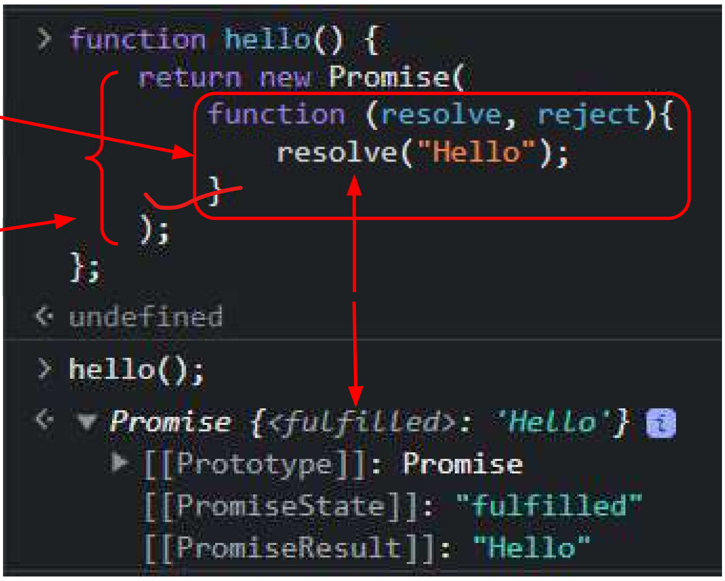

executor
Promise constructor calls executor function
executor(resolve, reject)
Promise()

Promise constructor calls executor function and passes two functions as arguments:

- resolve
- reject

How to Create a Promise Explicitly using Constructor?

- **Resolve Promise**:



```
> function hello() {
      return new Promise(
          function (resolve, reject){
              resolve("Hello");
          }
      );
  };
< undefined
> hello();
< ▼ Promise {<fulfilled>: 'Hello'}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "Hello"
```

Executor function

Create a Promise object.

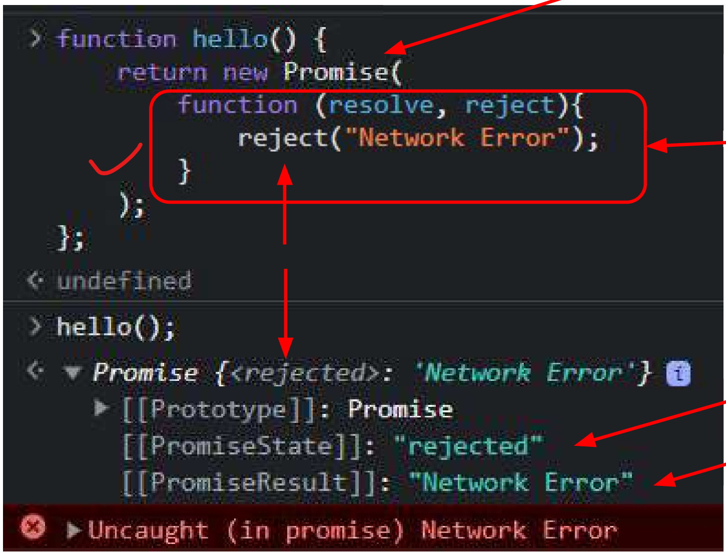
- We need to pass executor function to Promise constructor.
- This executor function will be executed immediately, when we create an object of Promise.
- Executor function gets two functions:
  - resolve to resolve a promise (for successful operation)
  - reject to reject a promise (generate error)

How to Create a Promise Explicitly using constructor?

- **Reject Promise**:

Create a Promise object.

- We return a rejected promise using reject function that we get via executor function.



```
> function hello() {
      return new Promise(
          function (resolve, reject){
              reject("Network Error");
          }
      );
  };
< undefined
> hello();
< ▼ Promise {<rejected>: 'Network Error'}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "rejected"
      [[PromiseResult]]: "Network Error"
⊗ ▶Uncaught (in promise) Network Error
```

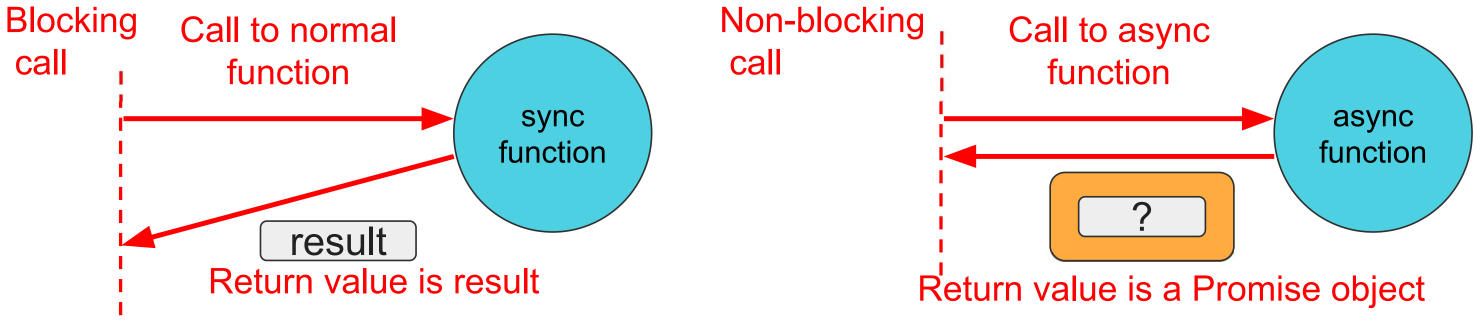
Executor function

Promise state is rejected.

Promise result is an error.

Difference between returned value is promise vs result

- If a function is not async, then the caller of that function will get blocked until the called function completes its execution.
- However, if a function is an async function, then the caller can decide whether it wants to wait for the result or want to continue its other work.
  - So async function does not return result, rather promise and that is returned immediately.



Availability of Result in Promise

- If a function returns a Promise, then result is not available immediately.
  - If a function returns a Promise object, the promise object may be returned immediately. (But, not result, i.e., in the promise object, result may not be available).
  - However, that promise object will not contain result.
  - Because the promise has not yet been fulfilled.
- When the promise has been fulfilled, the promise object contains result (in PromiseResult property)

But how do we get result?

- If with Promise, the result is not available immediately, how do we get result and when do we get result?
  - We need to register callback functions for success and failure events.
  - So when the promise becomes successful,
    - Our callback function for success will be called.
  - And when the promise gets failed,
    - Our callback function for failure will be called.

How do we pass our callback functions?

- We pass our callback functions to promise object using:
  - then() for registration of success event handler.
  - catch() for registration of failure event handler.
  - finally() for registration of finally event handler.

How to use then(), catch(), and finally()

- then() is used to register callback function to be called for success event (when the promise resolves and produces a value).
  - (response) => { process response }
- catch() is used to register callback function for failure event.
  - (error) => { process error }
- finally() is used to register callback function for finally of promise.
  - (anything) => { process cleanup }

Who returns Promise object?

- If a function is decorated with async keyword, that function implicitly or explicitly returns a Promise object.
- All API functions also return Promise object.

How to create Promise object?

- We can **explicitly create** a Promise object using **Promise constructor** function.
- We can **implicitly create** a Promise object if we **return** from **async function**.

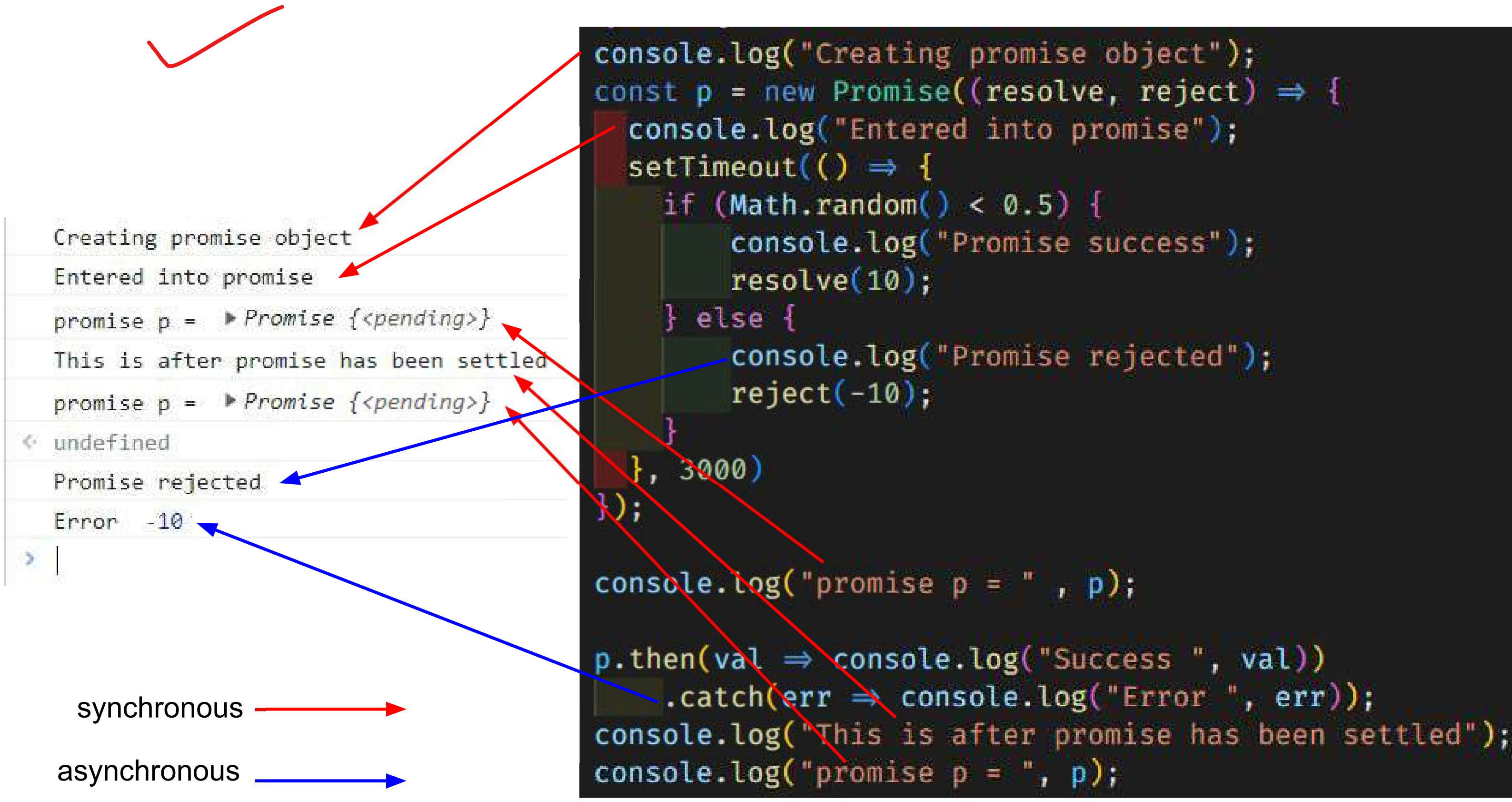
How to create Promise object explicitly?

- We can **explicitly create** a **Promise** object using
- new Promise(function(resolve, reject){

```
new Promise(function(resolve, reject){
        if(successFull)
                resolve(successValue)
        else if(failure)
                reject(failureValue)
})
```

- We get those values via our callbacks
    - successValue via then()
    - failureValue via catch()

- To Promise constructor we pass a callback function taking resolve and reject.
- The implementation of Promise constructor calls our callback function immediately/synchronously.
- When the Promise implementation calls our callback function, it provides resolve and reject functions.
- Whatever value we pass to resolve function becomes available in the parameter of callback function that we pass to then().
- Whatever value we pass to reject function becomes available in the parameter of callback function that we pass to catch().



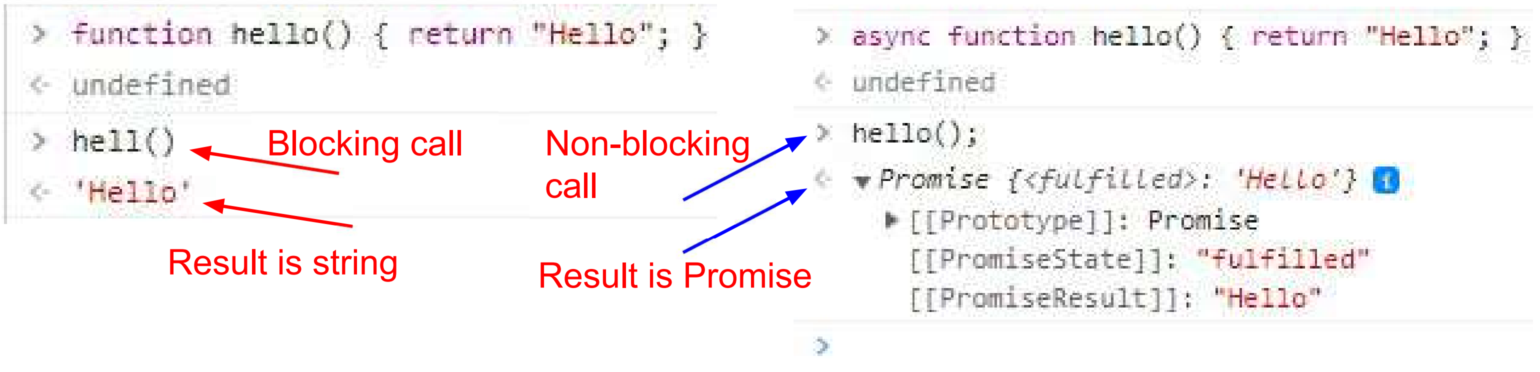
```
console.log("Creating promise object");
const p = new Promise((resolve, reject) => {
  console.log("Entered into promise");
  setTimeout(() => {
    if (Math.random() < 0.5) {
      console.log("Promise success");
      resolve(10);
    } else {
      console.log("Promise rejected");
      reject(-10);
    }
  }, 3000)
});

console.log("promise p = " , p);

p.then(val => console.log("Success ", val))
  .catch(err => console.log("Error ", err));
console.log("This is after promise has been settled");
console.log("promise p = ", p);
```

```
Creating promise object
Entered into promise
promise p =  ▶ Promise {<pending>}
This is after promise has been settled
promise p =  ▶ Promise {<pending>}
<· undefined
Promise rejected
Error  -10
> |
```

How to create Promise object implicitly? Return value of async function

Returned type and value of **normal function**.

Returned type and value of **async function**.



```
> function hello() { return "Hello"; }
<· undefined
> hell()
<· 'Hello'
```

```
> async function hello() { return "Hello"; }
<· undefined
> hello();
<· ▼ Promise {<fulfilled>: 'Hello'} ℹ
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "Hello"
>
```

async function

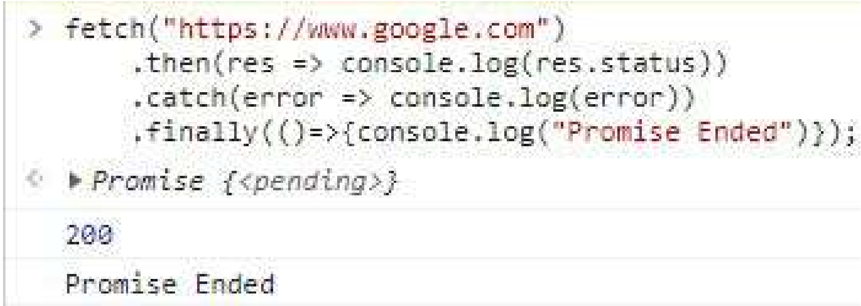
- An asynchronous function in JavaScript is defined using async keyword.
- Difference between synchronous and asynchronous functions:
  - A synchronous function blocks its caller until the called function completes.
  - An asynchronous function does not block its caller.

Two important characteristics of async function

- When we call normal function, the caller of the function gets blocked, i.e., the caller cannot do anything till the called function completes its execution.
- For async function that may not happen.
  - An async function always returns a promise, future completion of operation.
  - An async function is not synchronous, so it can return without blocking the caller.

Using Promise in Console of Browser

```
fetch("https://www.google.com")
        .then(res => console.log(res.status))
        .catch(error => console.log(error))
        .finally(()=>{console.log("Promise Ended")});
```

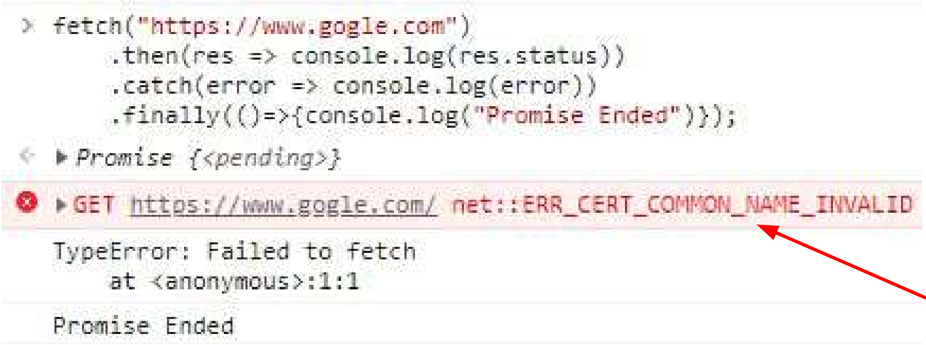


```
> fetch("https://www.google.com")
       .then(res => console.log(res.status))
       .catch(error => console.log(error))
       .finally(()=>{console.log("Promise Ended")});
< ▸ Promise {<pending>}
  200
  Promise Ended
```

Using Promise in Console of Browser

typing mistake

```
fetch("https://www.gogle.com")
        .then(res => console.log(res.status))
        .catch(error => console.log(error))
        .finally(()=>{console.log("Promise Ended")});
```



```
> fetch("https://www.gogle.com")
       .then(res => console.log(res.status))
       .catch(error => console.log(error))
       .finally(()=>{console.log("Promise Ended")});
< ▸ Promise {<pending>}
✖ ▸GET https://www.gogle.com/ net::ERR_CERT_COMMON_NAME_INVALID
  TypeError: Failed to fetch
      at <anonymous>:1:1
  Promise Ended
```

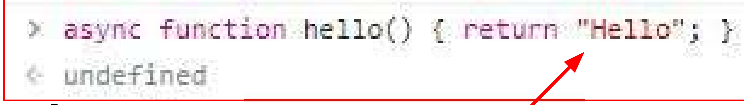
Error

Important points about async and await

- Which keyword to use for which situation?
    - The async is used to define or declare a function.
    - The await is used to while calling a function.
- What does exactly async mean?
    - It means that the function is asynchronous (Completion time is not known).
    - That means that the function will not return the result immediately and to get the result the caller has to wait.

Important points about async and await

- What does exactly await mean?
    - The await means that the caller will wait till async function returns.
    - Thus use of await function call inside a (outer) function also makes that (outer) function async.
    - That is await call can be made only inside async function.



```
> async function hello() { return "Hello"; }
< undefined
```

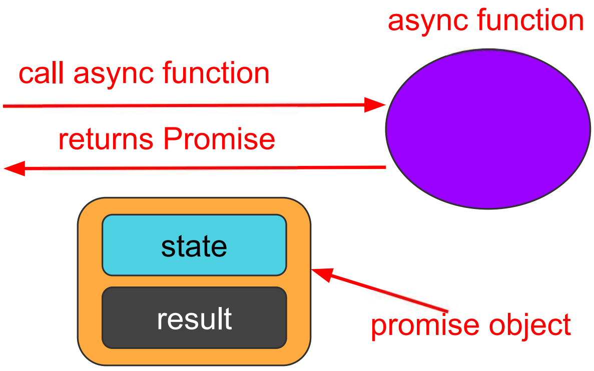
Important points about async and await

- An **async** function **always returns** a **promise**.
- For **async** functions, a **non-promise return value** is **wrapped** in a **Promise**.
  - That is in our earlier hello() async function, when **we returned "Hello"**, it **automatically** becomes **Promise.resolve("Hello")**;
- Use of **await pauses** an **async** function **call** until a promise is complete (resolved).
- The **await** will do one of the following:
  - **Yield** the value of a **fulfilled promise**.
  - **Throw** an exception from a **rejected** promise.
- The **await** can **only** be used **in** an **async function**.

What happens if we do not use await keyword?

- If we do **not** write **await** keyword while calling async function, the **result** of **async** function **call** will be a **Promise**.

const promise = asyncFunction();



async function

call async function

returns Promise

state

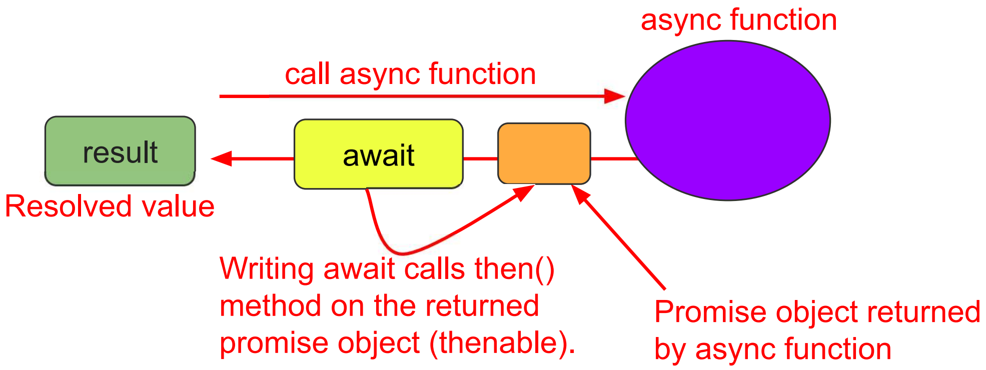
result

promise object

What happens if we use await keyword?

- If we write **await** keyword while calling async function, the **result** of the **async** function call will be a **resolved value**.
  - Whenever, we put an **await** keyword before any **object** (returned value of async), JS just tries to call the **then() method** of that object.

```
const result = await asyncFunction();
```



async function

call async function

result

await

Resolved value

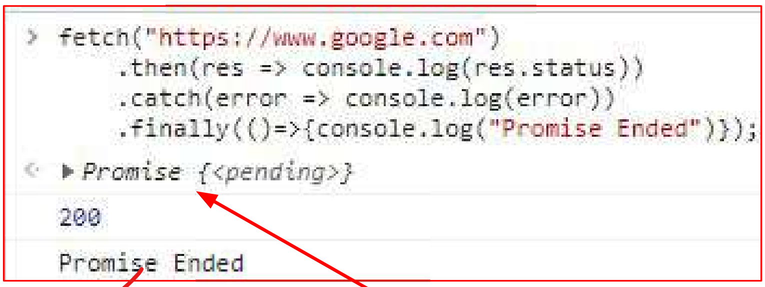
Writing await calls then() method on the returned promise object (thenable).

Promise object returned by async function

What happens if we use or do not use await keyword?

- That is the following two are equivalent for **success** (**resolved** case):

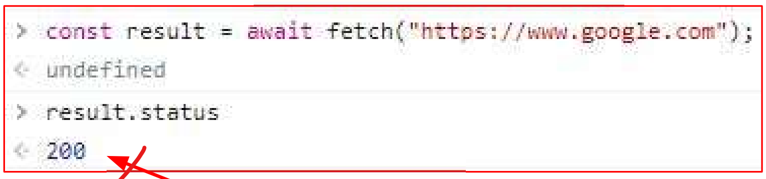
Do not use await



```
> fetch("https://www.google.com")
      .then(res => console.log(res.status))
      .catch(error => console.log(error))
      .finally(()=>{console.log("Promise Ended")});
< ▸ Promise {<pending>}
  200
  Promise Ended
```

Returned value is promise

Use await



```
> const result = await fetch("https://www.google.com");
< undefined
> result.status
< 200
```

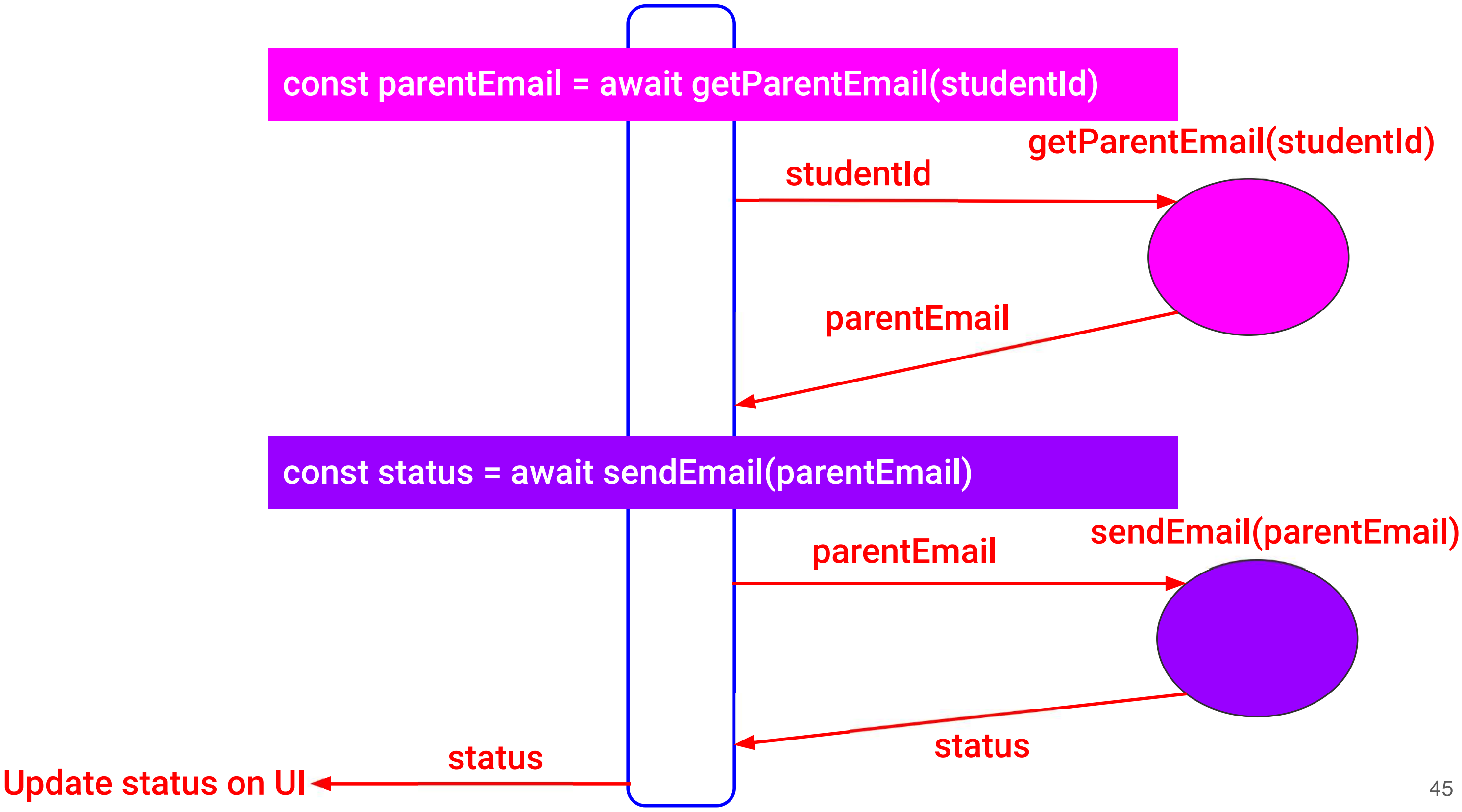
Returned value is resolved value

Why to Make async call Synchronous using await?

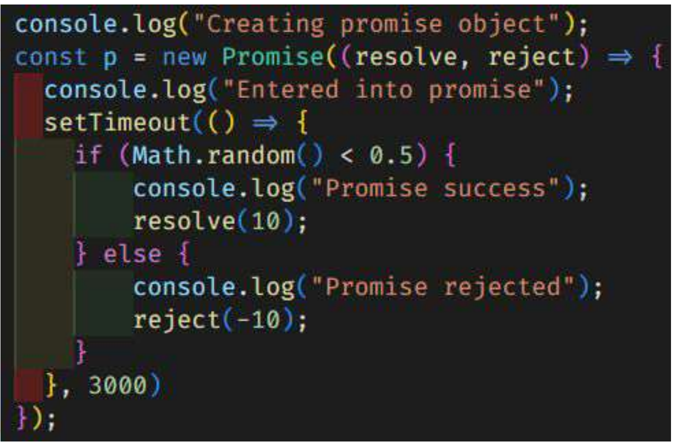
- We create async function to perform asynchronous operations.
- But, then we make call to async function synchronous using await keyword.
  - Why?
- Assume a scenario:
  - We have one API that returns us some information.
  - Now, based on received information, we want to make another API call.
- In this situation, we do not make call to second API until response of the first API is received.
  - So in this situation, we use await.

Practical Situation

- We have a list of students who has less than 75% attendance.
- We want to send an email to the student's parent.
- We have two APIs:
  - One API provides email address of parent for given student.
  - Another API sends an email to the provided email address.
- In this situation, we have to convert async call into synchronous call using await.


const parentEmail = await getParentEmail(studentId)
getParentEmail(studentId)
studentId
parentEmail
const status = await sendEmail(parentEmail)
sendEmail(parentEmail)
parentEmail
status
status
Update status on UI

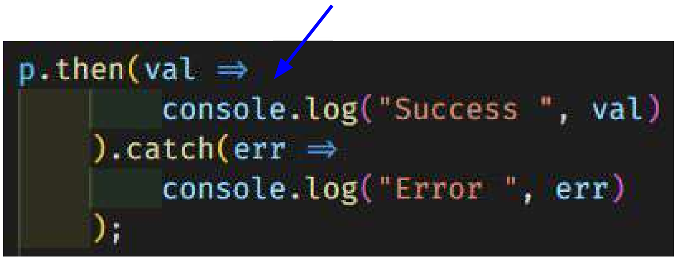
Equivalent then/catch vs await



```
console.log("Creating promise object");
const p = new Promise((resolve, reject) => {
  console.log("Entered into promise");
  setTimeout(() => {
    if (Math.random() < 0.5) {
      console.log("Promise success");
      resolve(10);
    } else {
      console.log("Promise rejected");
      reject(-10);
    }
  }, 3000)
});
```

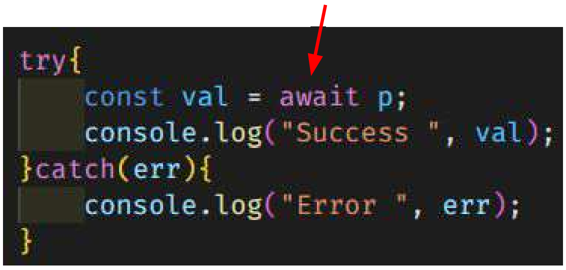
Consider this promise object p.

- The code is asynchronous.



```
p.then(val =>
        console.log("Success ", val)
    ).catch(err =>
        console.log("Error ", err)
    );
```

- This can be used in only async function.
- The code is synchronous.



```
try{
    const val = await p;
    console.log("Success ", val);
}catch(err){
    console.log("Error ", err);
}
```

Macro Tasks and Micro Tasks

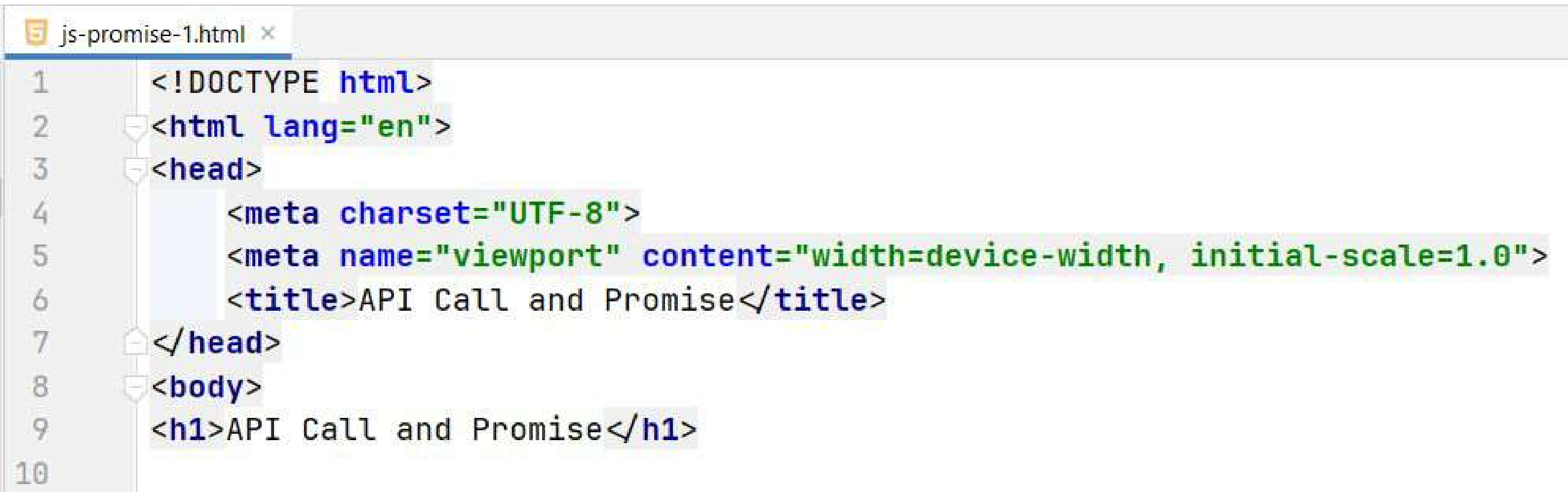
- JavaScript engine is single threaded, so how does it make asynchronous calls?
  - API calls and setTimeout() or setInterval() are executed by browser and not by JavaScript.
- There are two queues that browser uses to inform to the JavaScript engine about the tasks given to them.
  - Macro tasks queue.
    - Used for setTimeout() or setInterval().
  - Micro tasks queue (Have higher priority)
    - Used for promises.

Priority of tasks

- For each round of the event loop, JavaScript Engine does the following:
  - Run synchronous code.
  - Run Promise or microtask callbacks.
  - Run async task callbacks (e.g., setTimeout(), setInterval(), etc.).

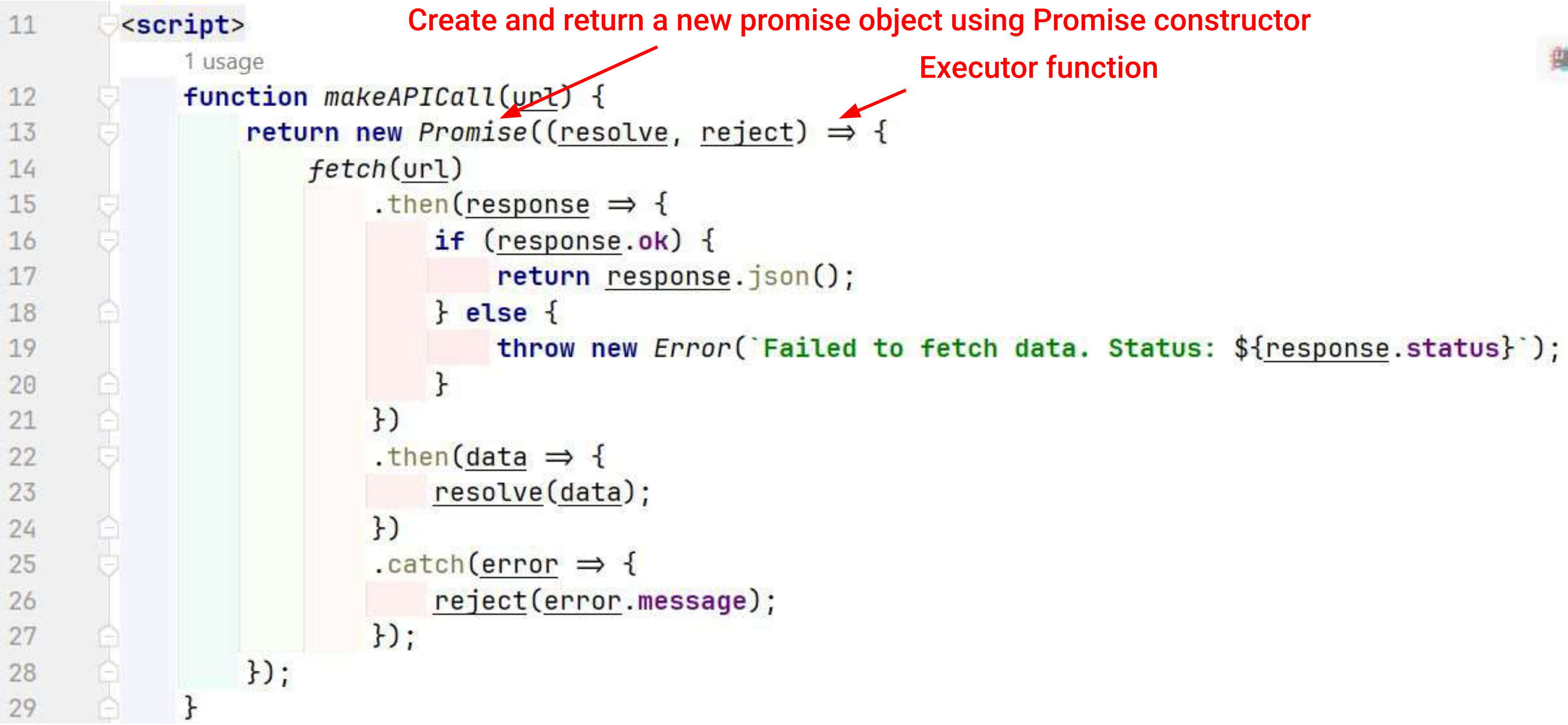
Example: API Call and Promise

js-promise-1.html



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>API Call and Promise</title>
</head>
<body>
<h1>API Call and Promise</h1>

```

js-promise-1.html

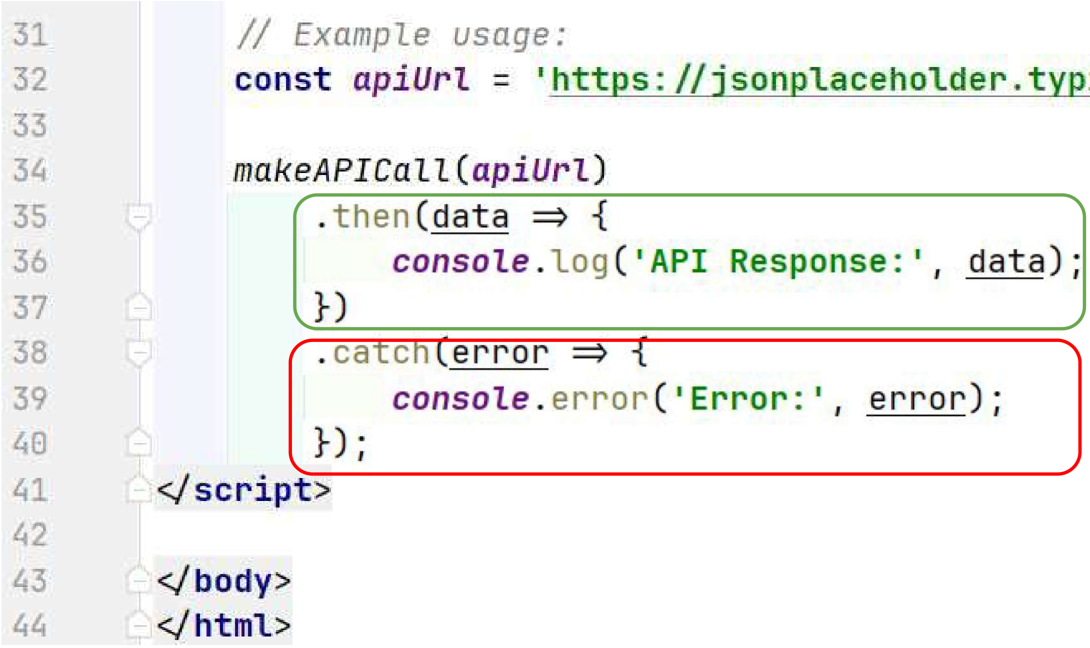


Create and return a new promise object using Promise constructor

Executor function

```
<script>
    1 usage
    function makeAPICall(url) {
        return new Promise((resolve, reject) => {
            fetch(url)
                .then(response => {
                    if (response.ok) {
                        return response.json();
                    } else {
                        throw new Error(`Failed to fetch data. Status: ${response.status}`);
                    }
                })
                .then(data => {
                    resolve(data);
                })
                .catch(error => {
                    reject(error.message);
                });
        });
    }
```

js-promise-1.html

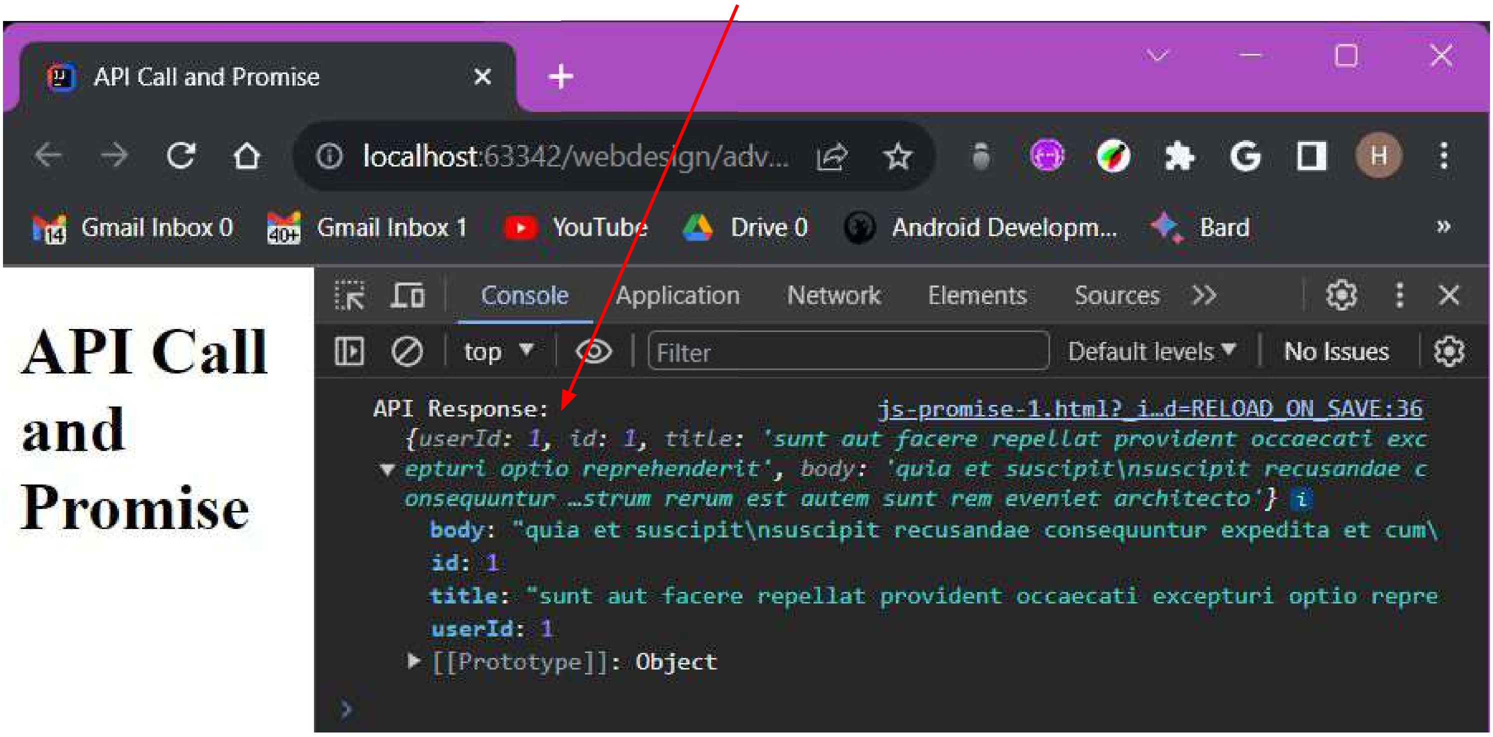


```
    // Example usage:
    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

    makeAPICall(apiUrl)
        .then(data => {
            console.log('API Response:', data);
        })
        .catch(error => {
            console.error('Error:', error);
        });
</script>

</body>
</html>
```

Registered callback to execute if the promise was resolved.

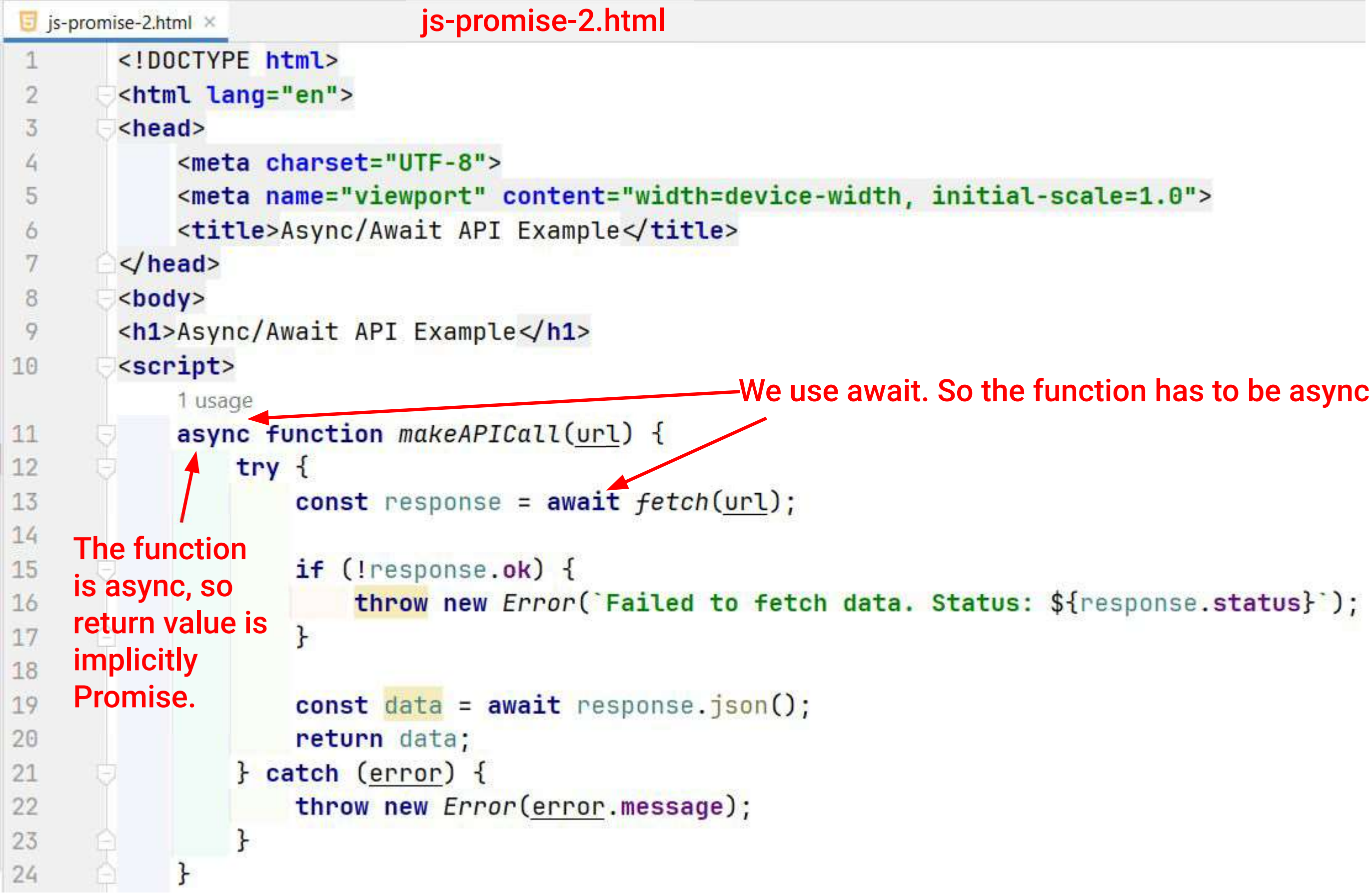
Registered callback to execute if the promise was rejected.

API response is received after some time, few milliseconds.



API Call and Promise

```
API Response:                              js-promise-1.html?_i…d=RELOAD_ON_SAVE:36
  {userId: 1, id: 1, title: 'sunt aut facere repellat provident occaecati exc
▼ epturi optio reprehenderit', body: 'quia et suscipit\nsuscipit recusandae c
  onsequuntur …strum rerum est autem sunt rem eveniet architecto'} i
    body: "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\
    id: 1
    title: "sunt aut facere repellat provident occaecati excepturi optio repre
    userId: 1
  ▶ [[Prototype]]: Object
>
```



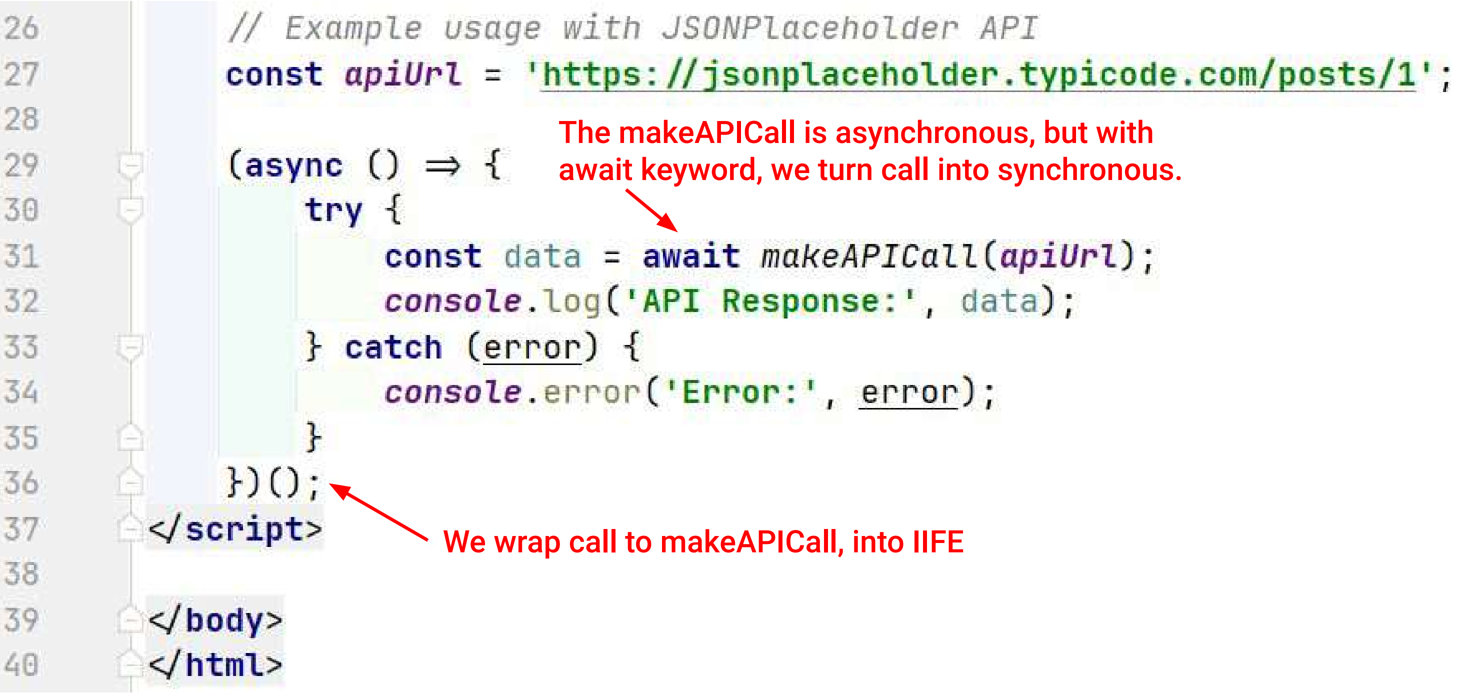
js-promise-2.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Async/Await API Example</title>
</head>
<body>
<h1>Async/Await API Example</h1>
<script>
    async function makeAPICall(url) {
        try {
            const response = await fetch(url);

            if (!response.ok) {
                throw new Error(`Failed to fetch data. Status: ${response.status}`);
            }

            const data = await response.json();
            return data;
        } catch (error) {
            throw new Error(error.message);
        }
    }
```

We use await. So the function has to be async

The function is async, so return value is implicitly Promise.

js-promise-2.html

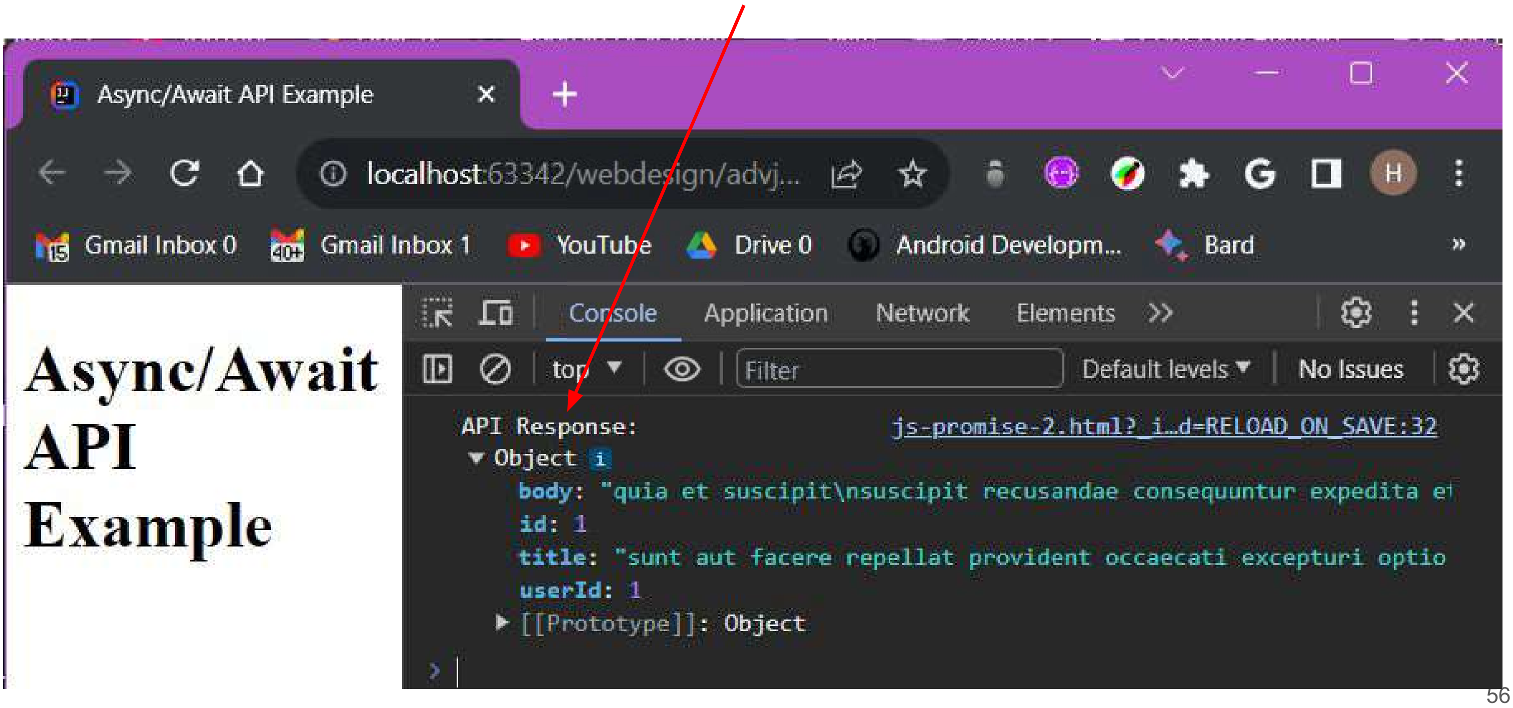


```
    // Example usage with JSONPlaceholder API
    const apiUrl = 'https://jsonplaceholder.typicode.com/posts/1';

    (async () => {
        try {
            const data = await makeAPICall(apiUrl);
            console.log('API Response:', data);
        } catch (error) {
            console.error('Error:', error);
        }
    })();
</script>

</body>
</html>
```

The makeAPICall is asynchronous, but with await keyword, we turn call into synchronous.

We wrap call to makeAPICall, into IIFE

API response is received after some time, few milliseconds.



Async/Await API Example

```
API Response:                    js-promise-2.html?_i…d=RELOAD_ON_SAVE:32
▼ Object i
    body: "quia et suscipit\nsuscipit recusandae consequuntur expedita e
    id: 1
    title: "sunt aut facere repellat provident occaecati excepturi optio
    userId: 1
  ▶ [[Prototype]]: Object
```

Methods of Promise

Methods of Promise

- Promise.all() takes an array of promises and return a new promise.
- This new promise resolves when all promises in the array have resolved.

```
Promise.all([
  axios.get('https://api.example.com/data1'),
  axios.get('https://api.example.com/data2'),
  axios.get('https://api.example.com/data3')
])
  .then(responses => {
    console.log(responses[0].data);
    console.log(responses[1].data);
    console.log(responses[2].data);
  })
  .catch(error => {
    console.log(error);
  });
```

Makes three requests in parallel.

- This responses will be an array.
- It contains response of each API call in the same sequence.

Promise.race

- Promise.race() takes an array of promises and return a new promise.
- This new promise resolves or rejects as soon as one of the promises in the array resolves or rejects.

```
const request1 = axios.get('https://api.example.com/data1');
const request2 = axios.get('https://api.example.com/data2');
const request3 = axios.get('https://api.example.com/data3');

Promise.race([request1, request2, request3])
 .then(response => {
   console.log(response.data);
 })
 .catch(error => {
   console.log(error);
 });
```

- This response will not be an array.
- It contains response of the first promise that resolved.

- The error contains an error object from the first promise that rejects.

What is the use of Promise.race?

- Promise.race is useful in situations where we want to quickly **retrieve data** from **multiple sources** and want to **use** the **data** from the **first source** that responded.

Promise.resolve and Promise.reject

- Promise.resolve and Promise.reject are used to create new resolved or rejected promises, respectively.
- Promise.resolve(value)
  - It creates a promise that immediately resolves with a specific value.
- Promise.reject(reason)
  - It creates a promise that immediately rejects with a specific reason.

Promise.any

- This method Promise.any takes an array of promises and returns a new promise that resolves as soon as one of the promises in the array resolves.
- If all promises reject, the returned promise will reject with an array of rejection reasons.

Promise.allSettled

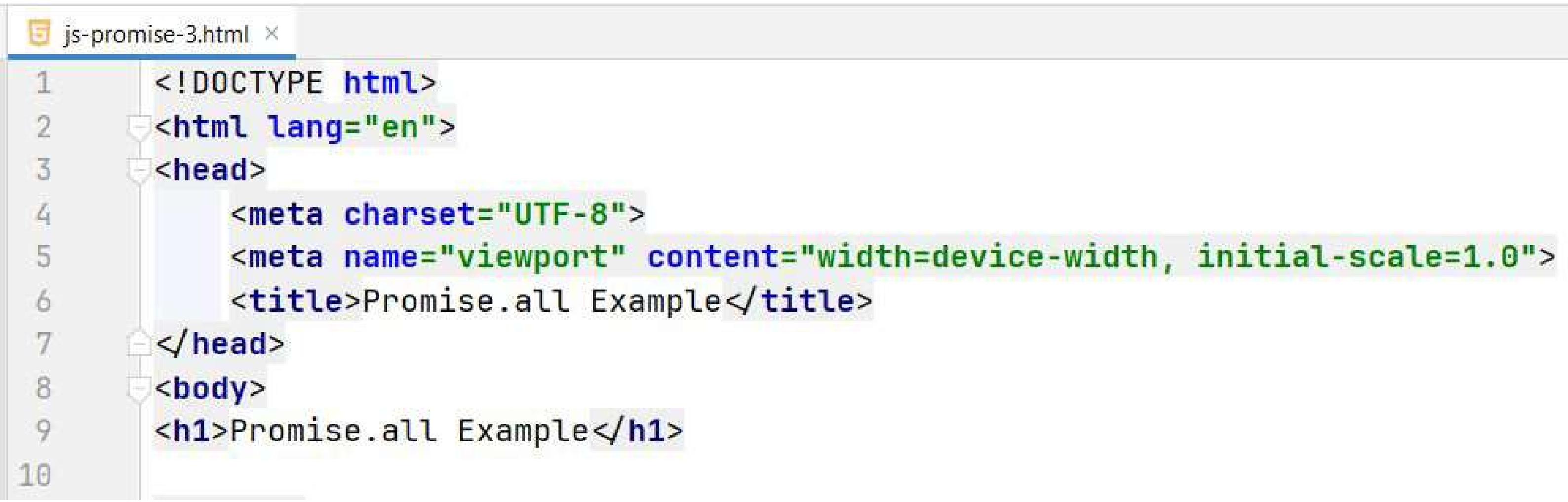
- This method Promise.allSettled takes an array of promises and returns a new promise.
- That new promise resolves with an array of objects once all the promises have settled (either resolved or rejected).
- Each object in the array represents the outcome of each promise.
- It includes
  - status: either fulfilled OR rejected
  - value OR reason

Promise.allSettled

- Difference between Promise.all() and Promise.allSettled
  - Promise.all() returns a new promise when all promises have resolved.
  - Promise.allSettled() returns a new promise when all promises have settled (either fulfilled OR rejected).

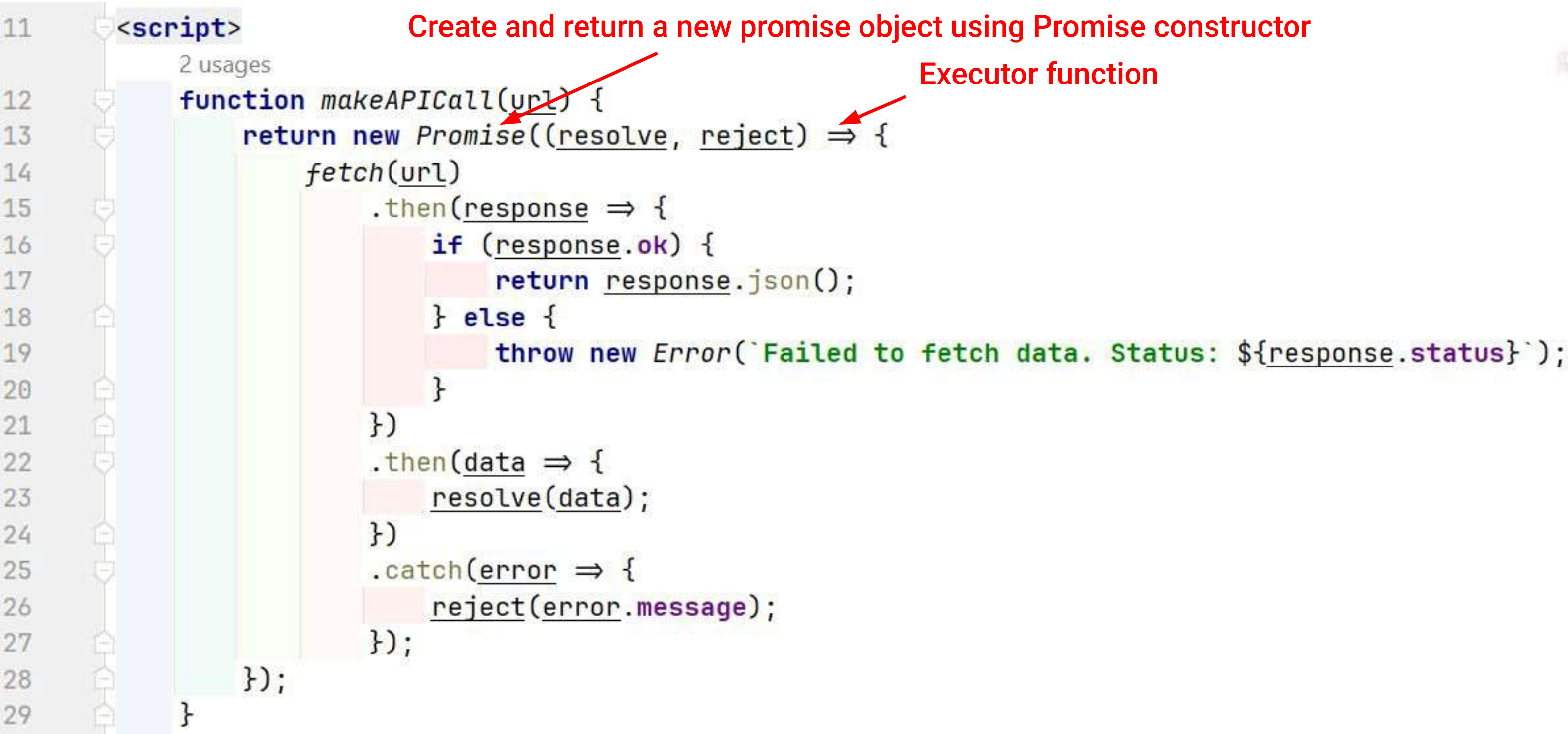
Example: Use of Promise.all

js-promise-3.html



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Promise.all Example</title>
</head>
<body>
<h1>Promise.all Example</h1>

```

js-promise-3.html

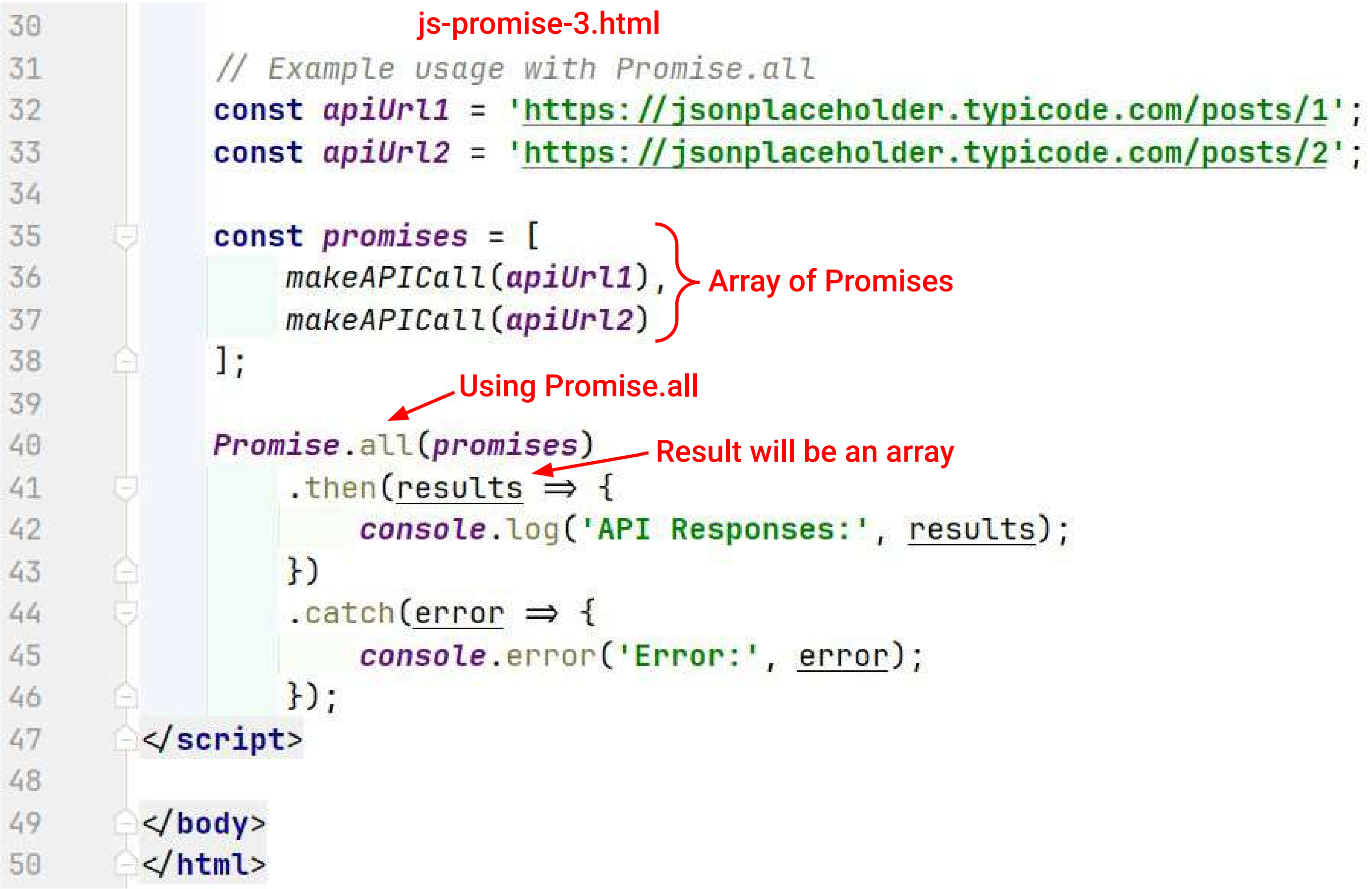


Create and return a new promise object using Promise constructor

Executor function

```
<script>
    2 usages
    function makeAPICall(url) {
        return new Promise((resolve, reject) => {
            fetch(url)
                .then(response => {
                    if (response.ok) {
                        return response.json();
                    } else {
                        throw new Error(`Failed to fetch data. Status: ${response.status}`);
                    }
                })
                .then(data => {
                    resolve(data);
                })
                .catch(error => {
                    reject(error.message);
                });
        });
    }
```

js-promise-3.html

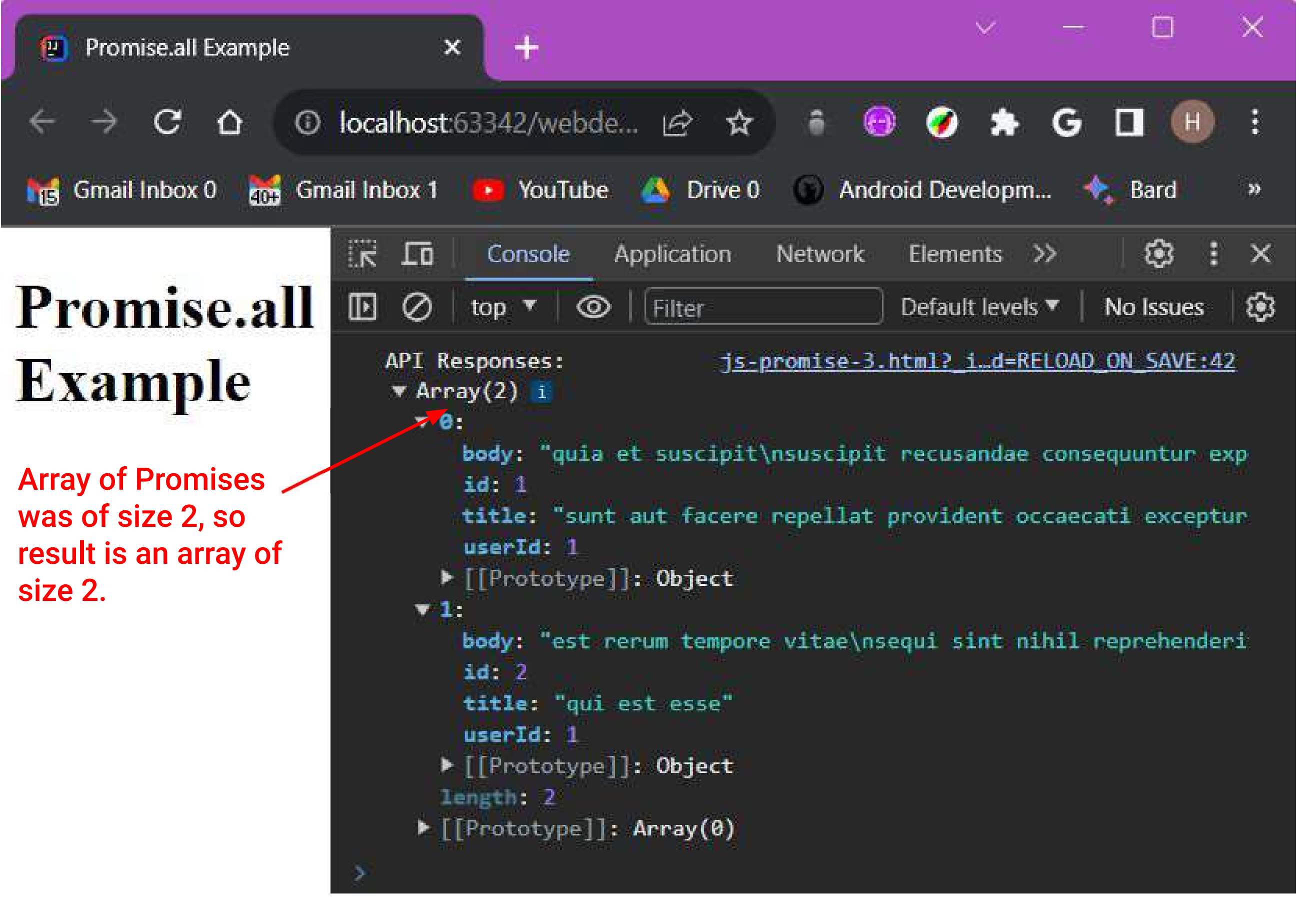


```

    // Example usage with Promise.all
    const apiUrl1 = 'https://jsonplaceholder.typicode.com/posts/1';
    const apiUrl2 = 'https://jsonplaceholder.typicode.com/posts/2';

    const promises = [
        makeAPICall(apiUrl1),
        makeAPICall(apiUrl2)
    ];

    Promise.all(promises)
        .then(results => {
            console.log('API Responses:', results);
        })
        .catch(error => {
            console.error('Error:', error);
        });
</script>

</body>
</html>
```

Array of Promises

Using Promise.all

Result will be an array



Promise.all Example

```
API Responses:                    js-promise-3.html?_i…d=RELOAD_ON_SAVE:42
▼ Array(2) i
  ▼ 0:
      body: "quia et suscipit\nsuscipit recusandae consequuntur exp
      id: 1
      title: "sunt aut facere repellat provident occaecati exceptur
      userId: 1
    ▶ [[Prototype]]: Object
  ▼ 1:
      body: "est rerum tempore vitae\nsequi sint nihil reprehenderi
      id: 2
      title: "qui est esse"
      userId: 1
    ▶ [[Prototype]]: Object
    length: 2
  ▶ [[Prototype]]: Array(0)
>
```

Array of Promises was of size 2, so result is an array of size 2.

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript