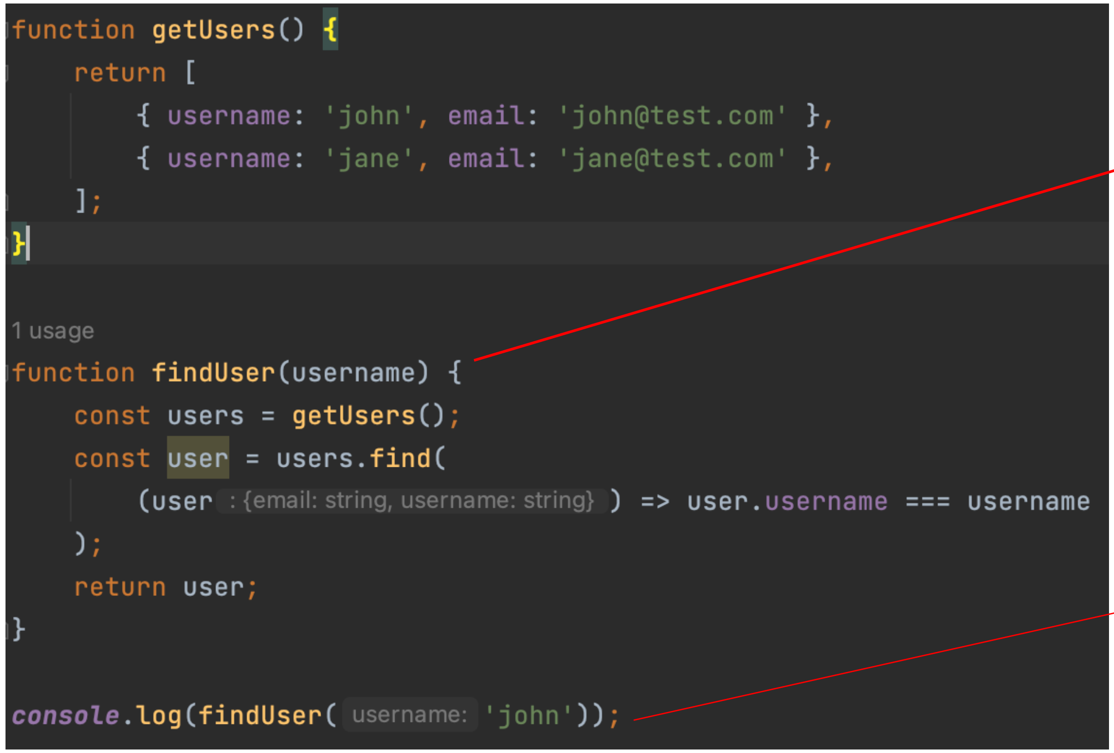
JavaScript Fundamentals

Instructor: Bui Binh Giang
buibinhgiang@vanlanguni.vn

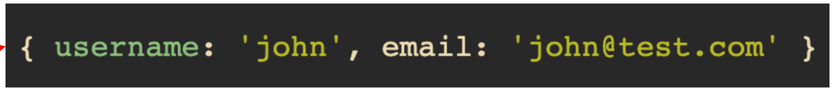
PROMISES

JavaScript Asynchronous Problem



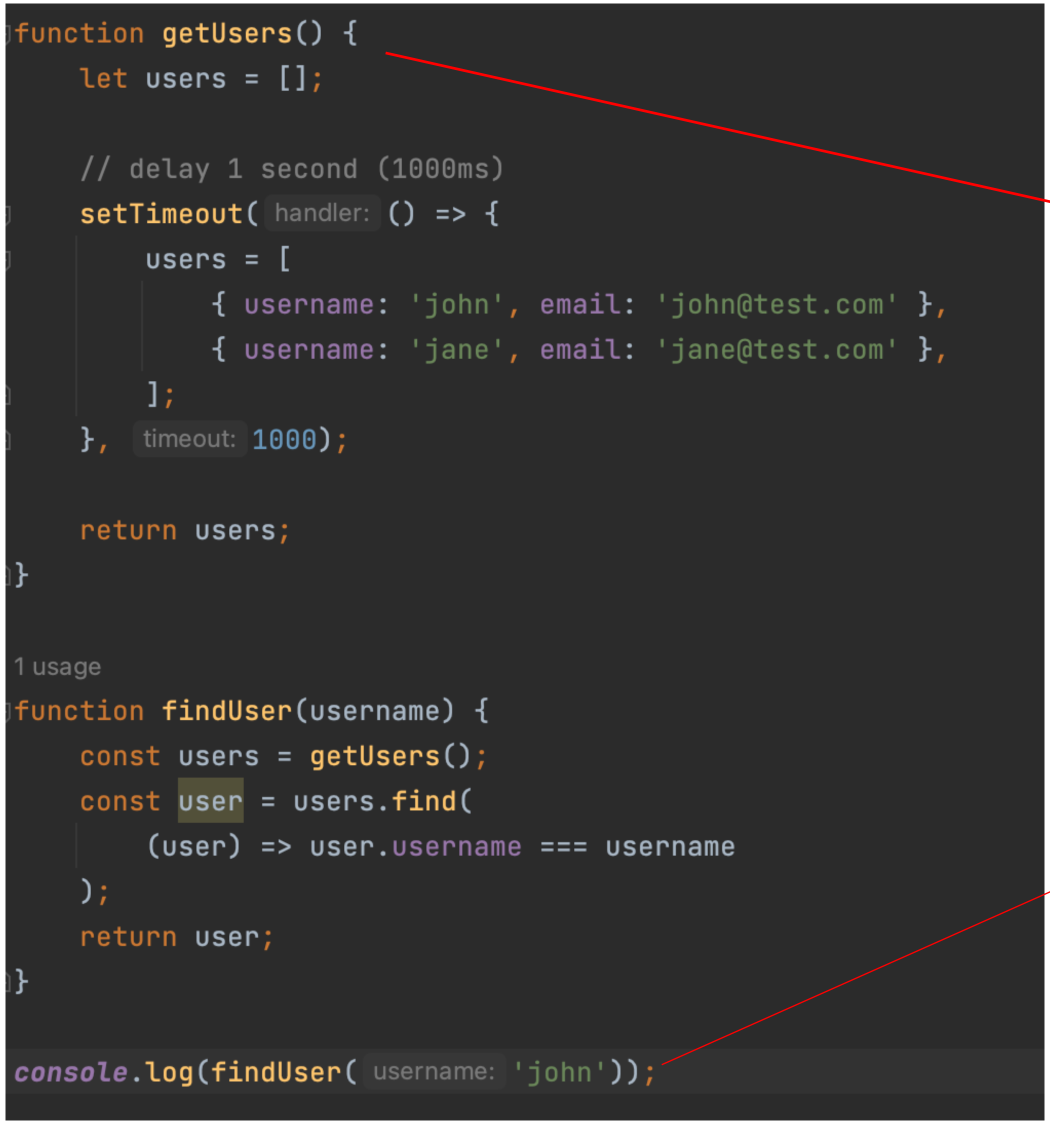
```
function getUsers() {
    return [
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
    ];
}

1 usage
function findUser(username) {
    const users = getUsers();
    const user = users.find(
        (user : {email: string, username: string} ) => user.username === username
    );
    return user;
}

console.log(findUser( username: 'john'));
```

The code in the findUser() function is **synchronous** and **blocking**. The findUser() function executes the getUsers() function to get a user array, calls the find() method on the users array to search for a user with a specific username, and returns the matched user.



```
{ username: 'john', email: 'john@test.com' }
```

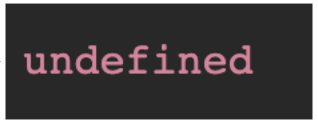
JavaScript Asynchronous Problem



```
function getUsers() {
    let users = [];

    // delay 1 second (1000ms)
    setTimeout( handler: () => {
        users = [
            { username: 'john', email: 'john@test.com' },
            { username: 'jane', email: 'jane@test.com' },
        ];
    }, timeout: 1000);

    return users;
}

1 usage
function findUser(username) {
    const users = getUsers();
    const user = users.find(
        (user) => user.username === username
    );
    return user;
}

console.log(findUser( username: 'john'));
```

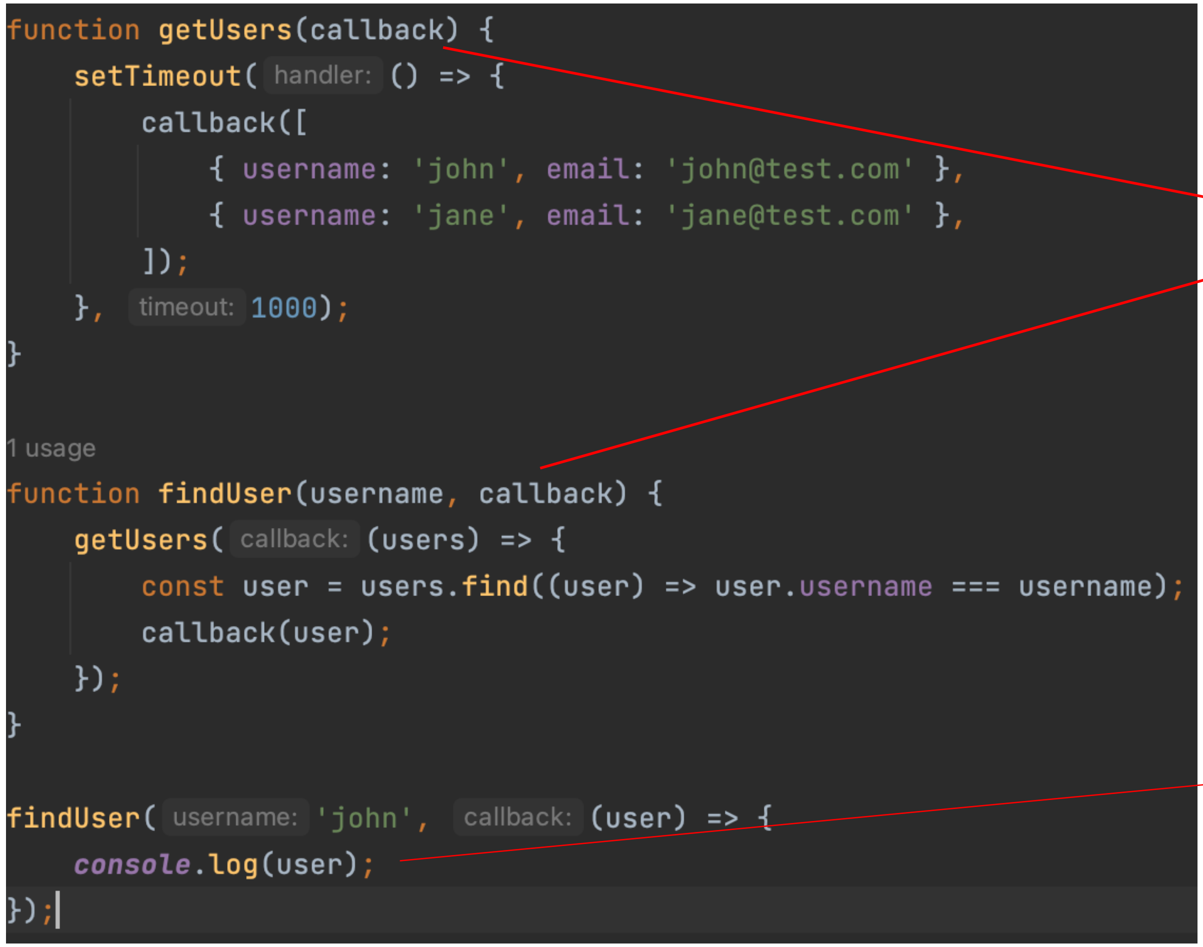
In practice, the getUsers() function may **access a database** or **call an API** to get the user list. Therefore, the getUsers() function will have a delay.

To simulate the delay, we use the **setTimeout**() function



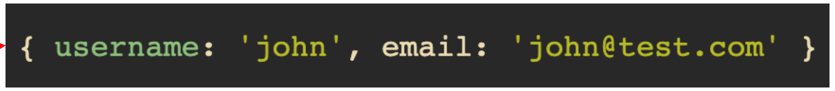
```
undefined
```

Using callbacks to deal with an asynchronous operation



```
function getUsers(callback) {
    setTimeout( handler: () => {
        callback([
            { username: 'john', email: 'john@test.com' },
            { username: 'jane', email: 'jane@test.com' },
        ]);
    }, timeout: 1000);
}

1 usage
function findUser(username, callback) {
    getUsers( callback: (users) => {
        const user = users.find((user) => user.username === username);
        callback(user);
    });
}

findUser( username: 'john',  callback: (user) => {
    console.log(user);
});
```

the getUsers() function accepts a **callback function** as an argument and invokes it with the users array inside the setTimeout() function. Also, the findUser() function accepts a **callback function** that processes the matched user



```
{ username: 'john', email: 'john@test.com' }
```

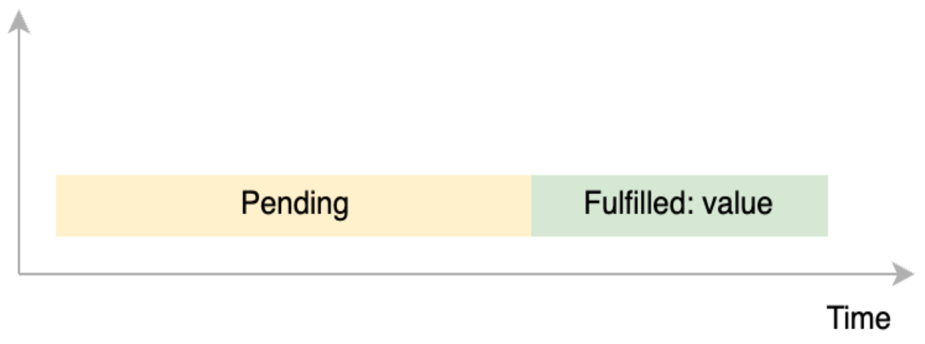
JavaScript Promises

A **promise** is an object that encapsulates the result of an asynchronous operation. Promise object has a state that can be one of the following:

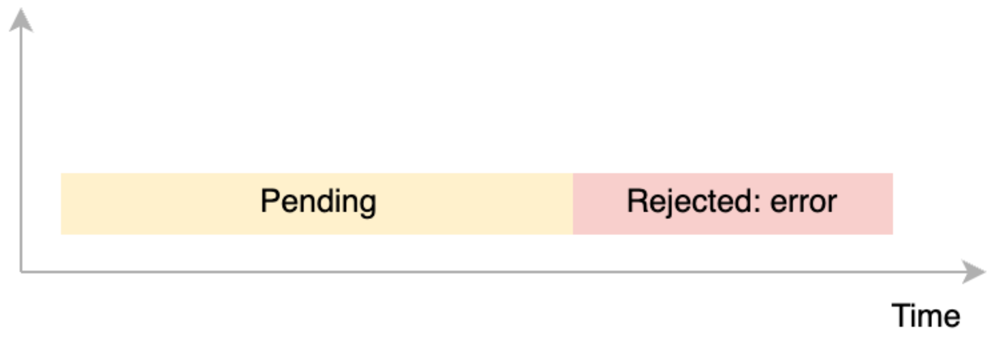
- **Pending**
- **Fulfilled** with a value
- **Rejected** for a reason

In the beginning, the state of a promise is **pending**, indicating that the asynchronous operation is in progress. Depending on the result of the asynchronous operation, the state changes to either **fulfilled** or **rejected**

JavaScript Promises



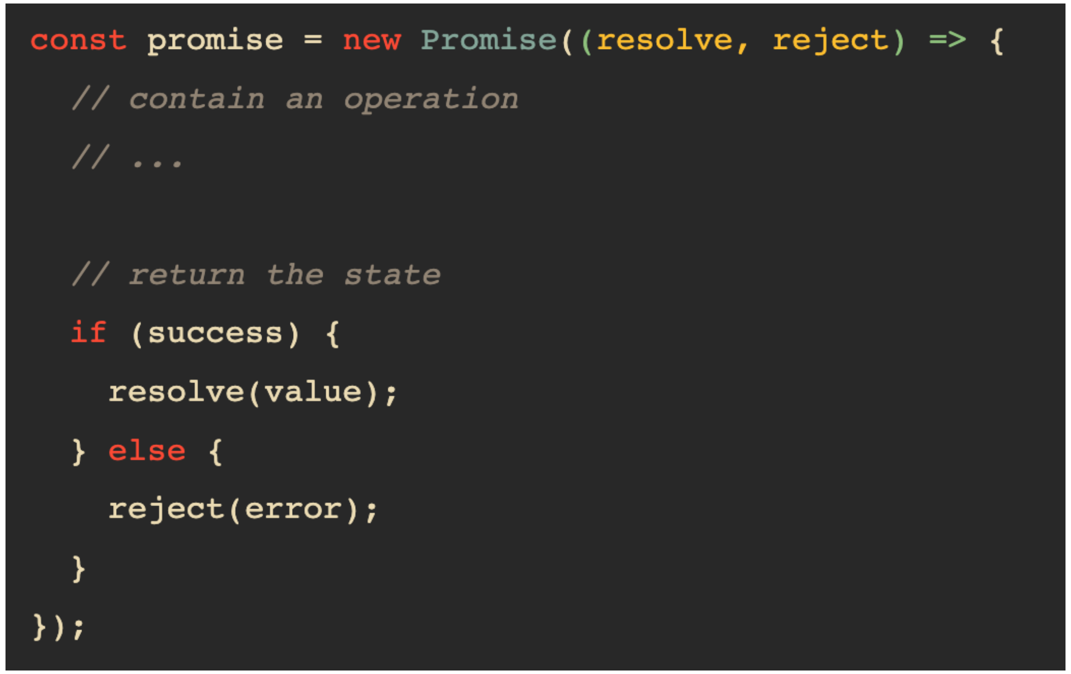
The **fulfilled** state indicates that the asynchronous operation was completed successfully



The **rejected** state indicates that the asynchronous operation failed

JavaScript Promises

Creating a promise with Promise() constructor



```
const promise = new Promise((resolve, reject) => {
  // contain an operation
  // ...

  // return the state
  if (success) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

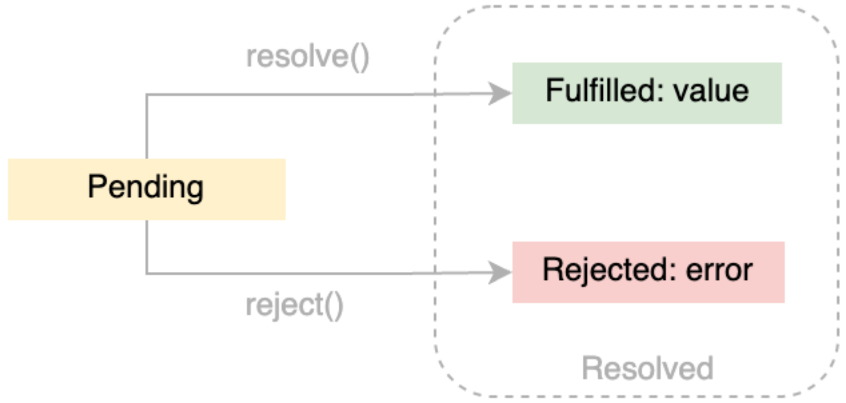
The **promise constructor** accepts a **callback function** that typically performs an **asynchronous operation**. This function is often referred to as an **executor**.

In turn, the **executor** accepts **two callback functions** with the name **resolve** and **reject:**

- If the asynchronous operation completes successfully, the executor will call the **resolve**() function to change the state of the promise from pending to fulfilled with a value.
- In case of an error, the executor will call the **reject**() function to change the state of the promise from pending to rejected with the error reason.

JavaScript Promises

- A promise cannot go from the **fulfilled** state to the **rejected** state and vice versa.
- Once a new **Promise** object is created, its state is **pending**. If a promise reaches **fulfilled** or **rejected** state, it is **resolved**.



Consuming a Promise: then, catch, finally

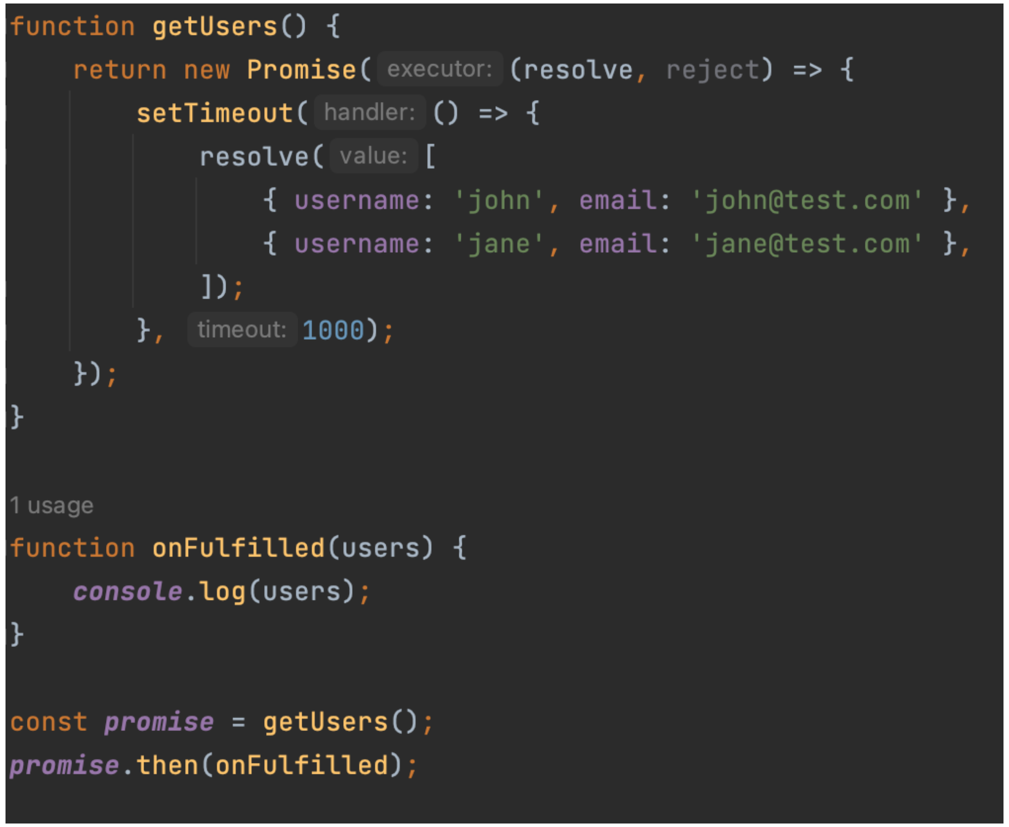
The then() method:

- To get the value of a promise when it’s **fulfilled**, you call the **then**() method of the promise object
- The **then**() method accepts two callback functions: **onFulfilled** and **onRejected**
- The **then**() method calls the **onFulfilled**() with a value, if the promise is fulfilled or the **onRejected**() with an error if the promise is rejected

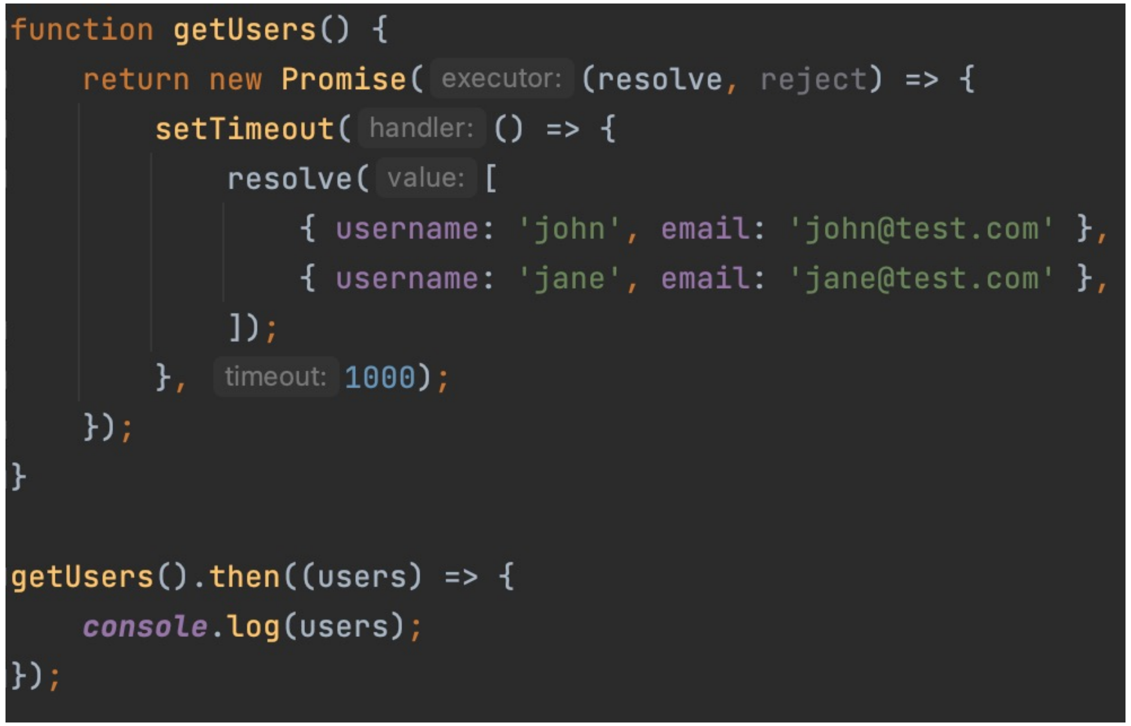
Note: both **onFulfilled** and **onRejected** arguments are optional.

```
promise.then(onFulfilled,onRejected);
```

Consuming a Promise: then, catch, finally

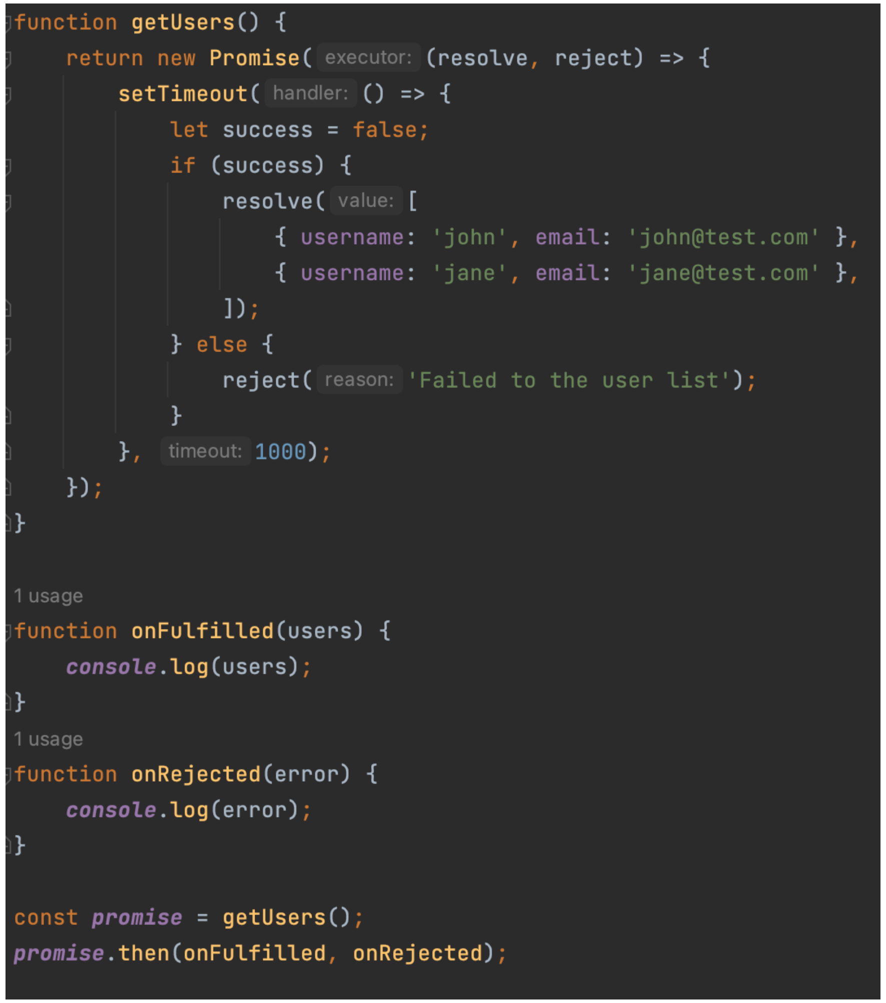


```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            resolve( value: [
                { username: 'john', email: 'john@test.com' },
                { username: 'jane', email: 'jane@test.com' },
            ]);
        }, timeout: 1000);
    });
}

1 usage
function onFulfilled(users) {
    console.log(users);
}

const promise = getUsers();
promise.then(onFulfilled);
```

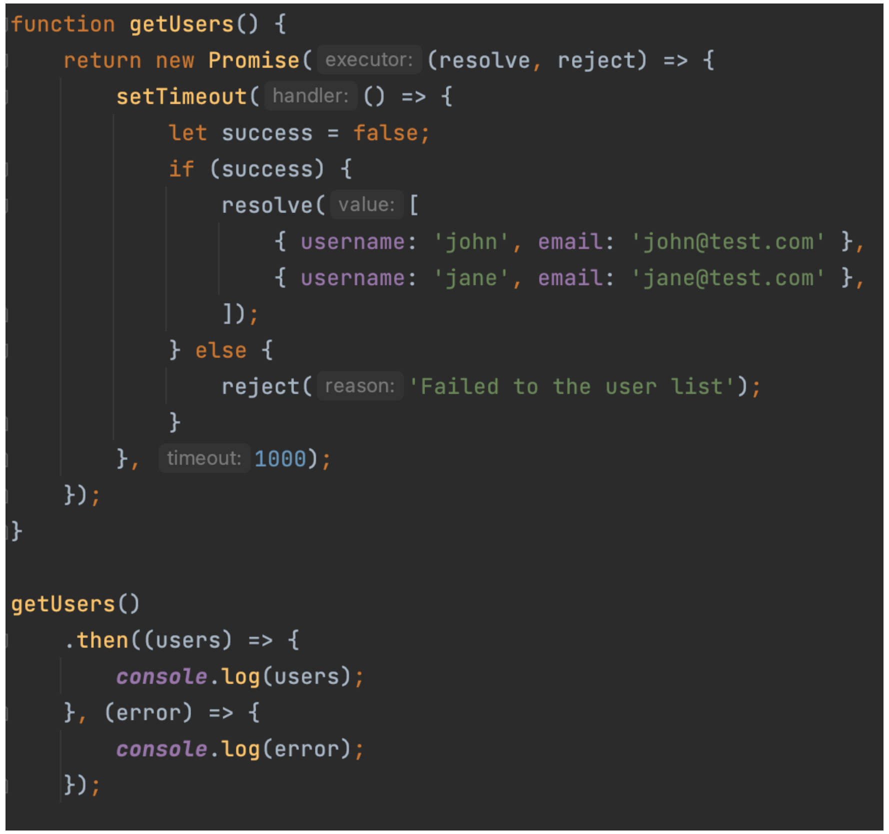


```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            resolve( value: [
                { username: 'john', email: 'john@test.com' },
                { username: 'jane', email: 'jane@test.com' },
            ]);
        }, timeout: 1000);
    });
}

getUsers().then((users) => {
    console.log(users);
});
```

Consuming a Promise: then, catch, finally



```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            let success = false;
            if (success) {
                resolve( value: [
                    { username: 'john', email: 'john@test.com' },
                    { username: 'jane', email: 'jane@test.com' },
                ]);
            } else {
                reject( reason: 'Failed to the user list');
            }
        },  timeout: 1000);
    });
}

1 usage
function onFulfilled(users) {
    console.log(users);
}
1 usage
function onRejected(error) {
    console.log(error);
}

const promise = getUsers();
promise.then(onFulfilled, onRejected);
```

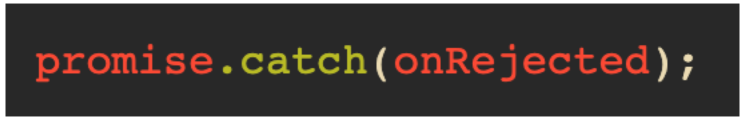


```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            let success = false;
            if (success) {
                resolve( value: [
                    { username: 'john', email: 'john@test.com' },
                    { username: 'jane', email: 'jane@test.com' },
                ]);
            } else {
                reject( reason: 'Failed to the user list');
            }
        },  timeout: 1000);
    });
}

getUsers()
    .then((users) => {
        console.log(users);
    }, (error) => {
        console.log(error);
    });
```

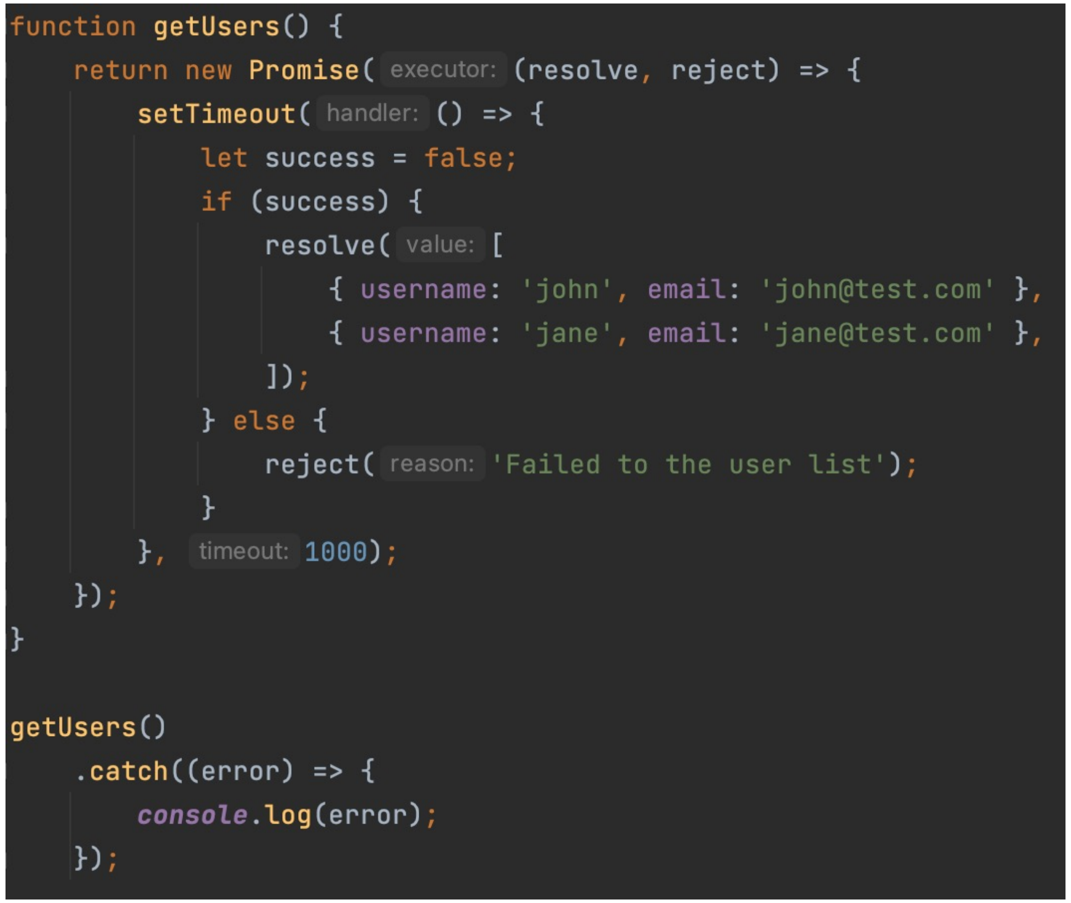
Consuming a Promise: then, catch, finally

The catch() method

- If you want to **get the error only** when the state of the promise is **rejected**, you can use the **catch**() method of the Promise object
- Internally, the **catch**() method invokes the **then(undefined, onRejected)** method



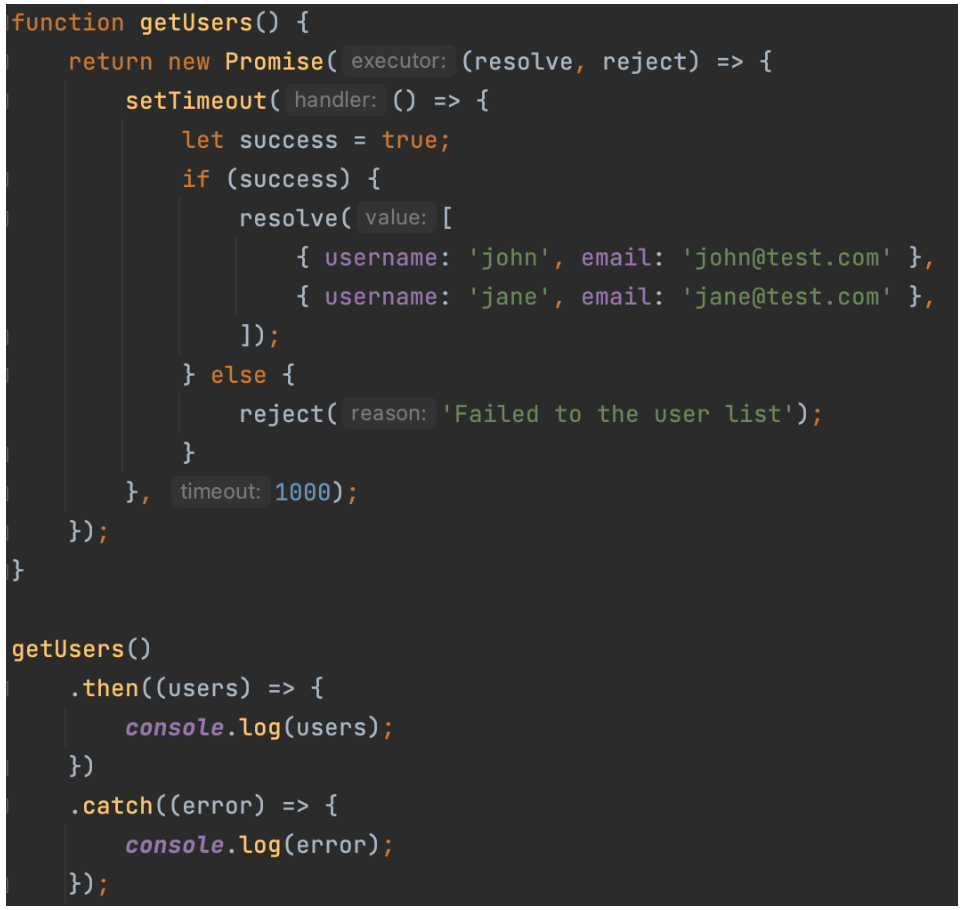
```
promise.catch(onRejected);
```



```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            let success = false;
            if (success) {
                resolve( value: [
                    { username: 'john', email: 'john@test.com' },
                    { username: 'jane', email: 'jane@test.com' },
                ]);
            } else {
                reject( reason: 'Failed to the user list');
            }
        },  timeout: 1000);
    });
}

getUsers()
    .catch((error) => {
        console.log(error);
    });
```

Consuming a Promise: then, catch, finally

then() + catch()

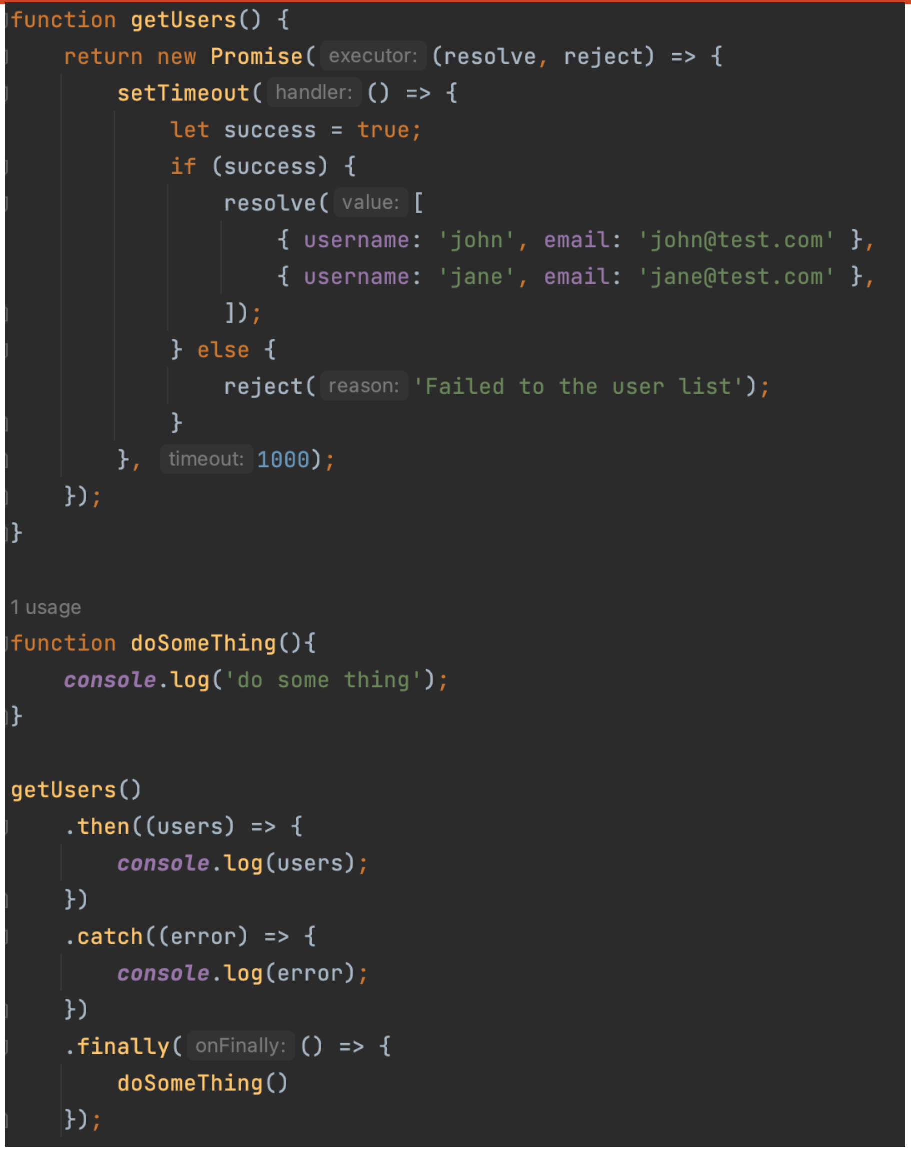


```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            let success = true;
            if (success) {
                resolve( value: [
                    { username: 'john', email: 'john@test.com' },
                    { username: 'jane', email: 'jane@test.com' },
                ]);
            } else {
                reject( reason: 'Failed to the user list');
            }
        },  timeout: 1000);
    });
}

getUsers()
    .then((users) => {
        console.log(users);
    })
    .catch((error) => {
        console.log(error);
    });
```

Consuming a Promise: then, catch, finally

The finally() method

- If you want to execute some piece of code no matter what the promise is **fulfilled** or **rejected**, you can use the **finally()** method of the **Promise** object

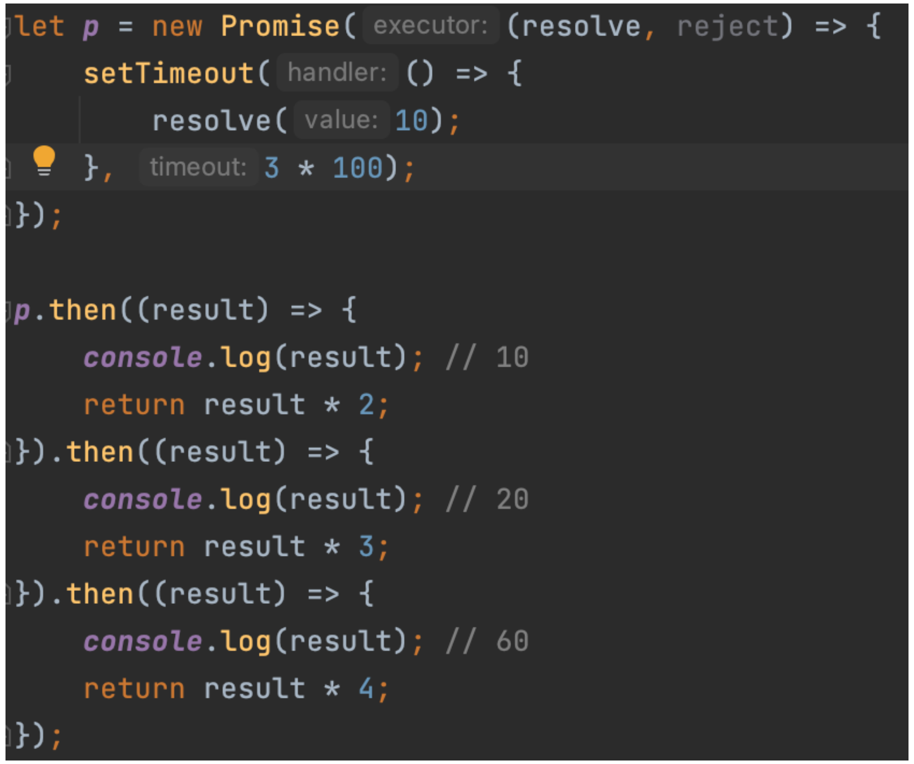


```
function getUsers() {
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            let success = true;
            if (success) {
                resolve( value: [
                    { username: 'john', email: 'john@test.com' },
                    { username: 'jane', email: 'jane@test.com' },
                ]);
            } else {
                reject( reason: 'Failed to the user list');
            }
        },  timeout: 1000);
    });
}

1 usage
function doSomeThing(){
    console.log('do some thing');
}

getUsers()
    .then((users) => {
        console.log(users);
    })
    .catch((error) => {
        console.log(error);
    })
    .finally( onFinally: () => {
        doSomeThing()
    });
```

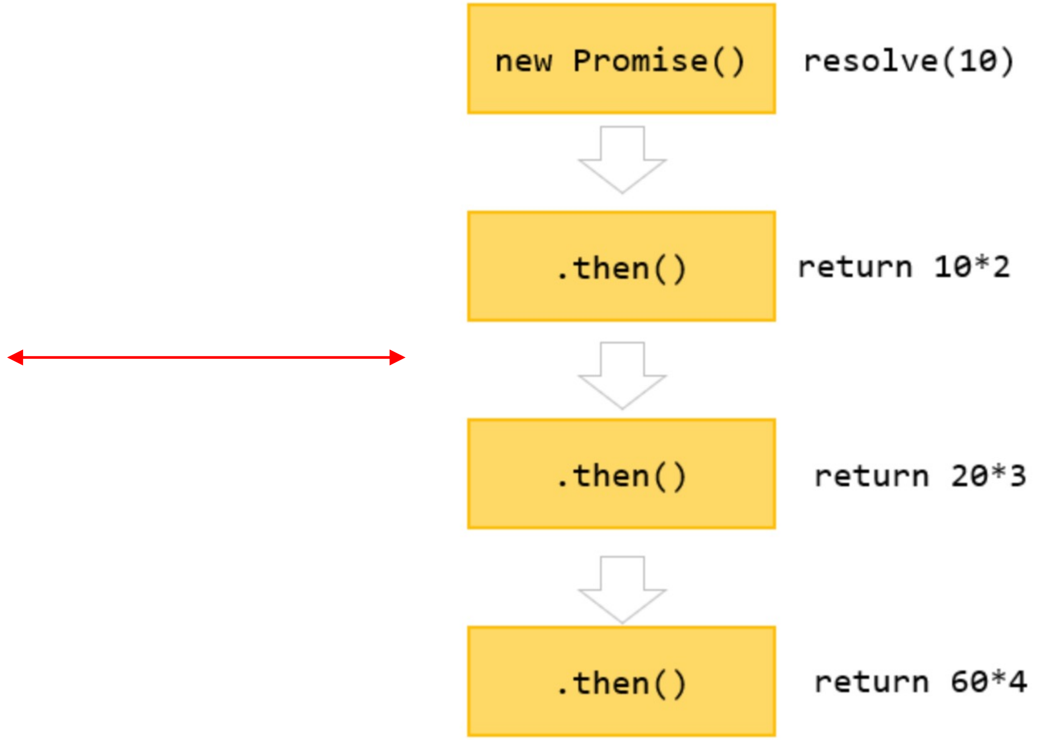
PROMISES CHAINING

JavaScript Promises chaining

Sometimes, you want to execute two or more related asynchronous operations, where the next operation starts with the result from the previous step → **return a value in the then**() method, the **then**() method returns a new Promise that immediately resolves to the **return value**.

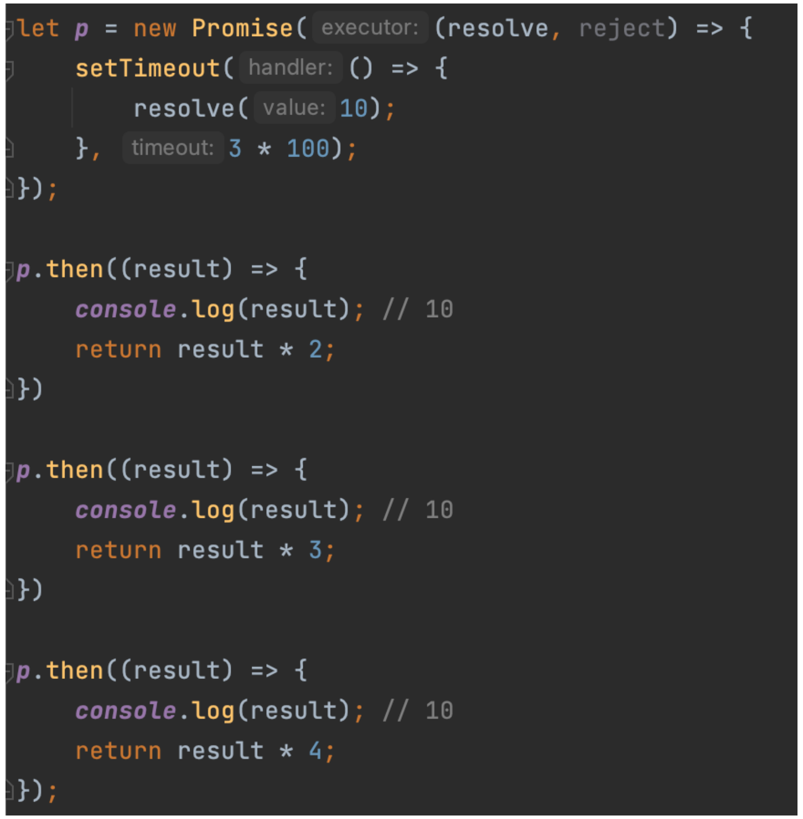


```
let p = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        resolve( value: 10);
    },  timeout: 3 * 100);
});

p.then((result) => {
    console.log(result); // 10
    return result * 2;
}).then((result) => {
    console.log(result); // 20
    return result * 3;
}).then((result) => {
    console.log(result); // 60
    return result * 4;
});
```

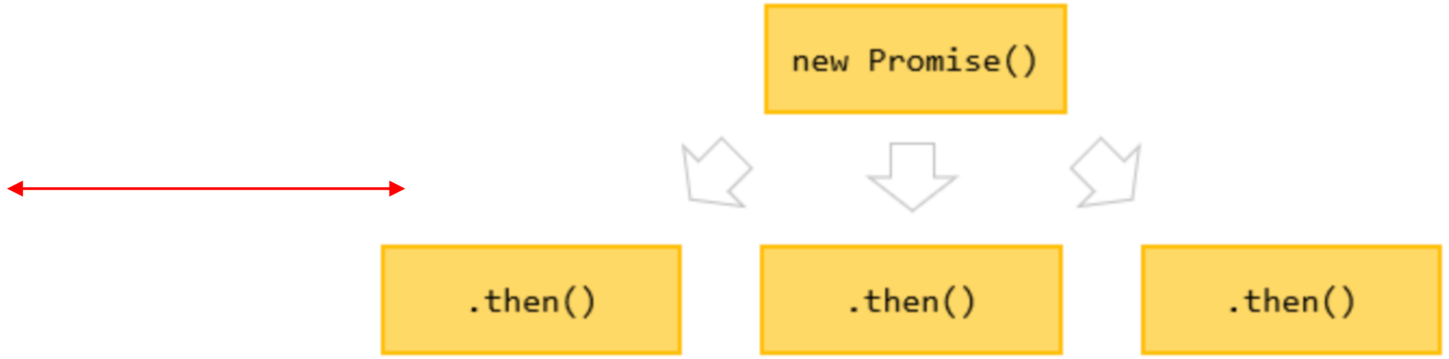


JavaScript Promises chaining

NOTE: When you call the **then**() method **multiple times on a promise**, it is **not** the promise chaining

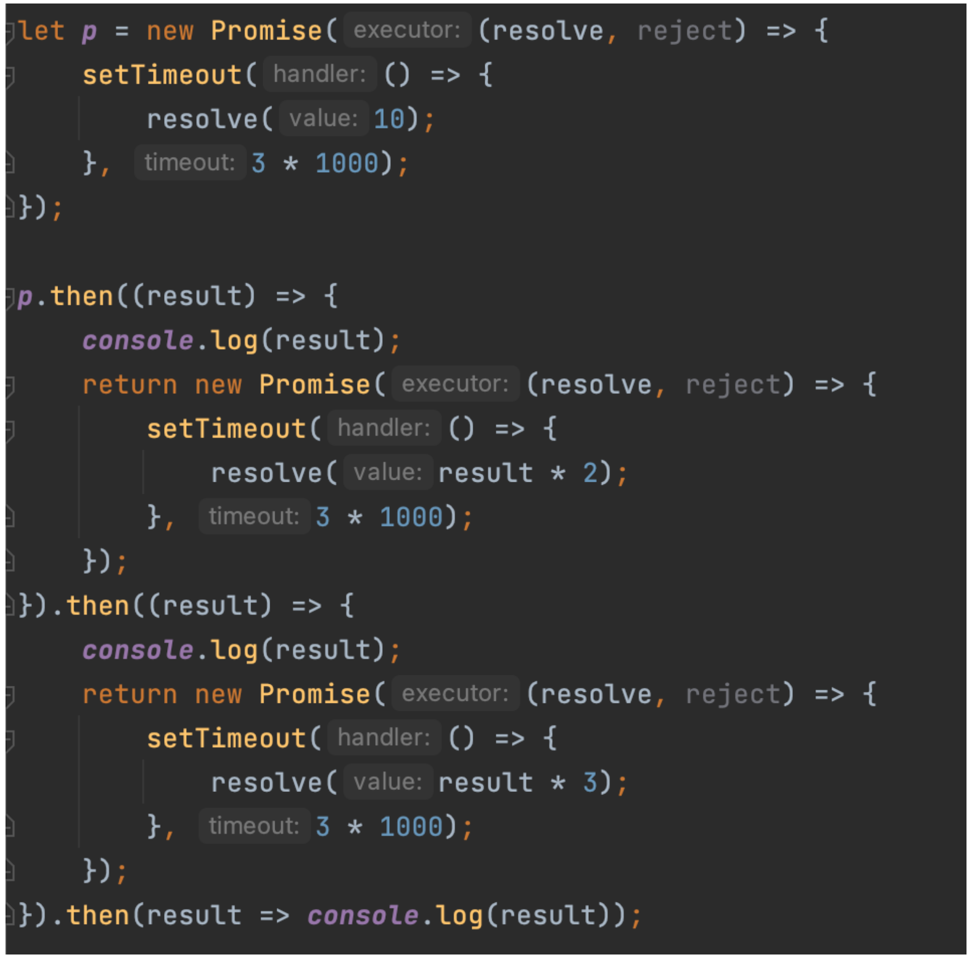


```
let p = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        resolve( value: 10);
    }, timeout: 3 * 100);
});

p.then((result) => {
    console.log(result); // 10
    return result * 2;
})

p.then((result) => {
    console.log(result); // 10
    return result * 3;
})

p.then((result) => {
    console.log(result); // 10
    return result * 4;
});
```



JavaScript Promises chaining

Returning a Promise



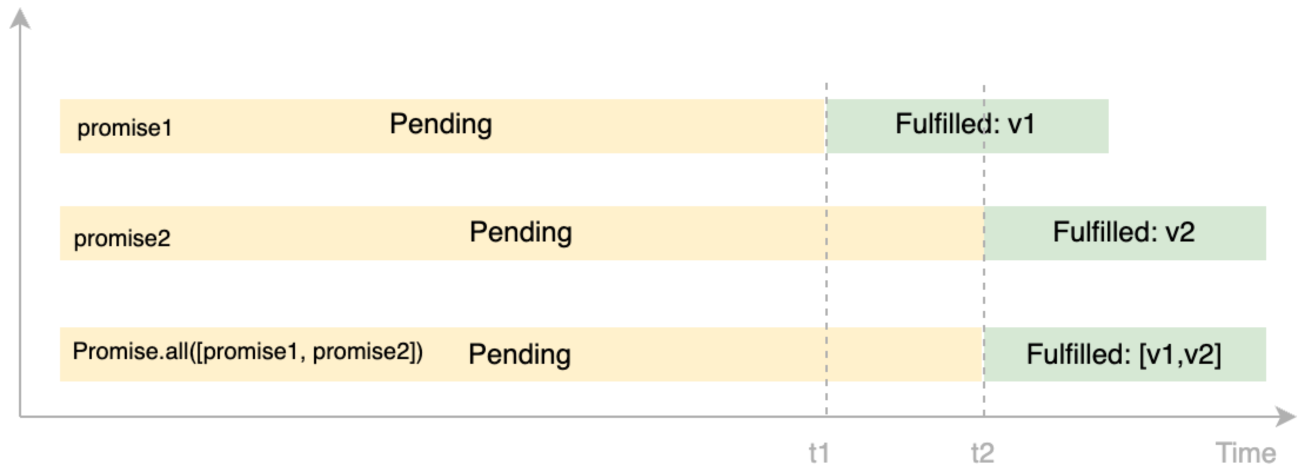
```
let p = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        resolve( value: 10);
    },  timeout: 3 * 1000);
});

p.then((result) => {
    console.log(result);
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            resolve( value: result * 2);
        },  timeout: 3 * 1000);
    });
}).then((result) => {
    console.log(result);
    return new Promise( executor: (resolve, reject) => {
        setTimeout( handler: () => {
            resolve( value: result * 3);
        },  timeout: 3 * 1000);
    });
}).then(result => console.log(result));
```

Promise.all()

JavaScript Promise.all()

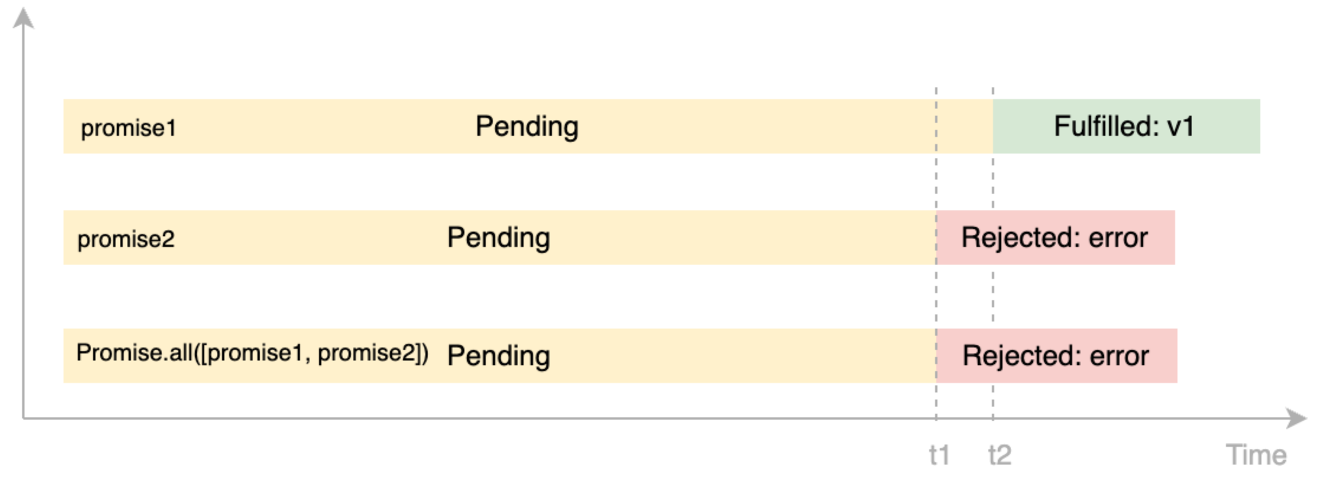
The **Promise.all**() method returns a single promise that resolves when all the input promises have been **resolved**. The returned promise resolves to an **array of the results of the input promises**

```
Promise.all(iterable);
```

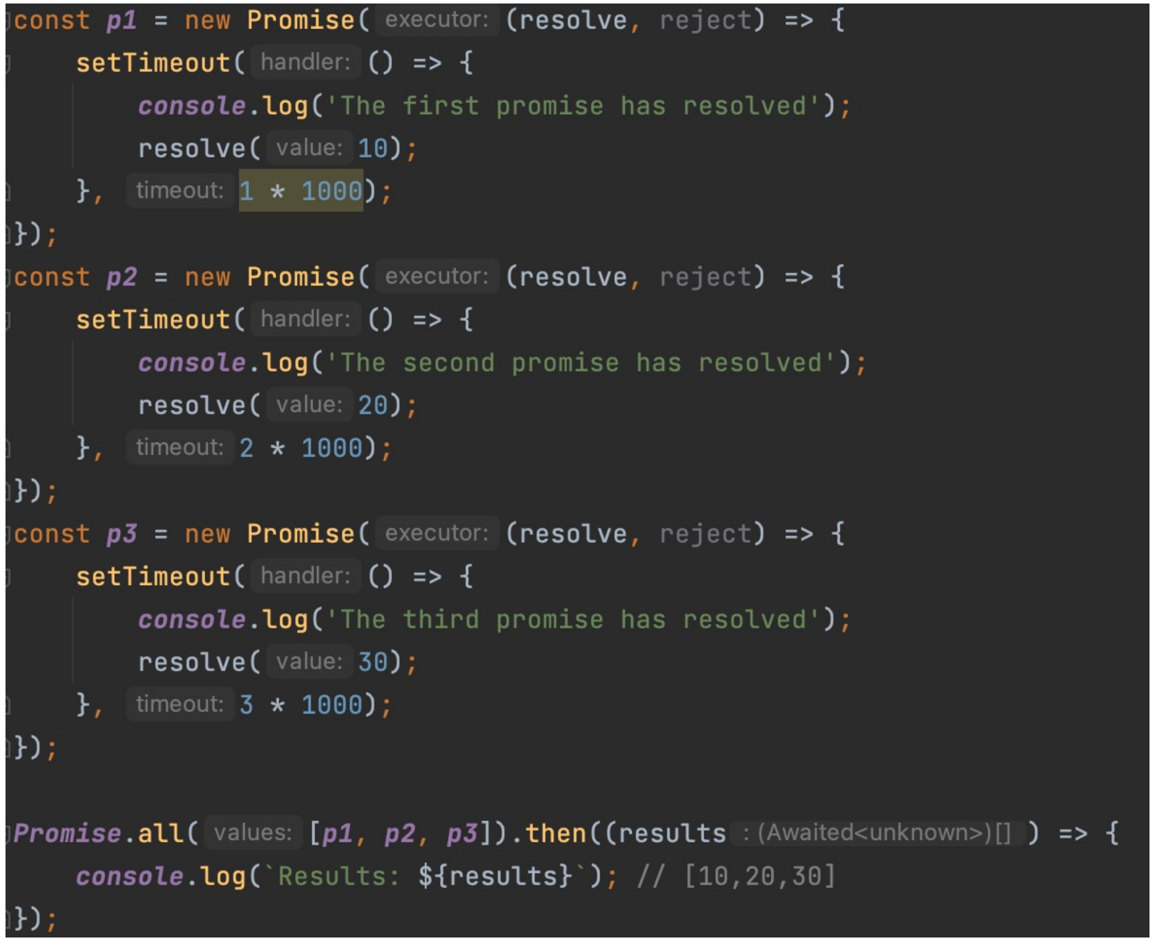


JavaScript Promise.all()

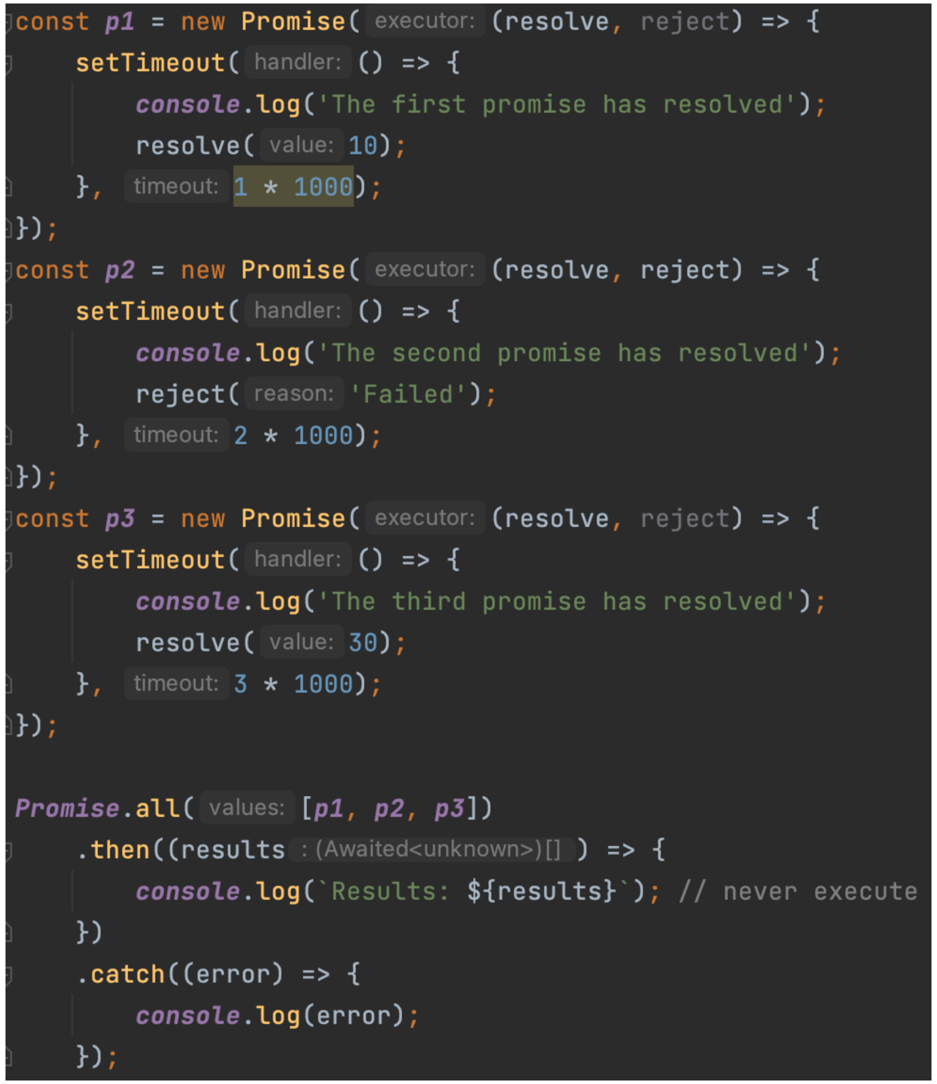
If one of the input promise rejects, the **Promise.all**() method immediately returns a promise that rejects with an error of the first rejected promise



JavaScript Promise.all()



```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The first promise has resolved');
        resolve( value: 10);
    }, timeout: 1 * 1000);
});
const p2 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The second promise has resolved');
        resolve( value: 20);
    }, timeout: 2 * 1000);
});
const p3 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The third promise has resolved');
        resolve( value: 30);
    }, timeout: 3 * 1000);
});

Promise.all( values: [p1, p2, p3]).then((results : (Awaited<unknown>)[] ) => {
    console.log(`Results: ${results}`); // [10,20,30]
});
```



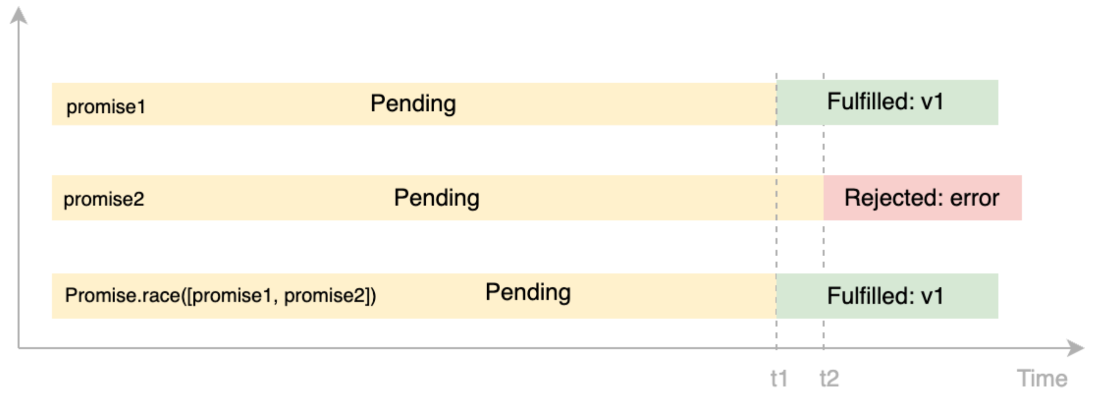
```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The first promise has resolved');
        resolve( value: 10);
    }, timeout: 1 * 1000);
});
const p2 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The second promise has resolved');
        reject( reason: 'Failed');
    }, timeout: 2 * 1000);
});
const p3 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The third promise has resolved');
        resolve( value: 30);
    }, timeout: 3 * 1000);
});

Promise.all( values: [p1, p2, p3])
    .then((results : (Awaited<unknown>)[] ) => {
        console.log(`Results: ${results}`); // never execute
    })
    .catch((error) => {
        console.log(error);
    });
```

Promise.race()

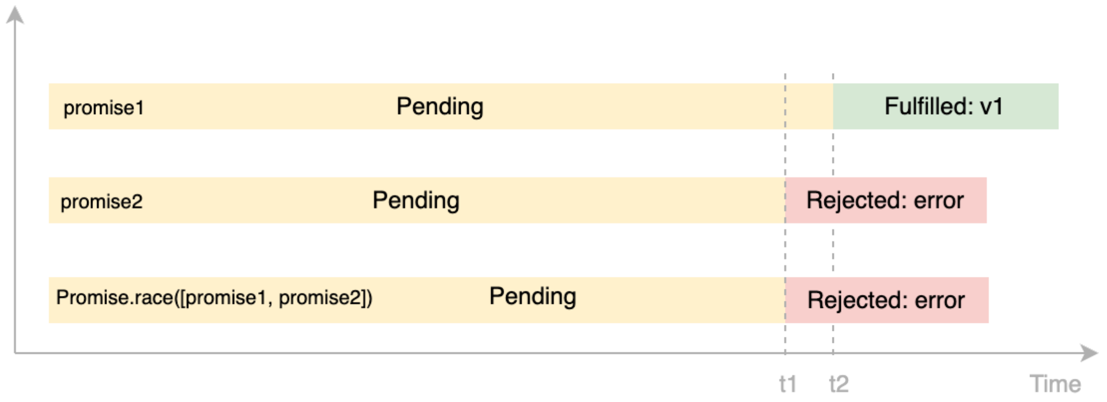
JavaScript Promise.race()

The **Promise.race**() method accepts a list of promises and returns a new promise that **fulfills** or **rejects** as soon as there is one promise that **fulfills** or **rejects**, with the value or reason from that promise

```
Promise.race(iterable)
```

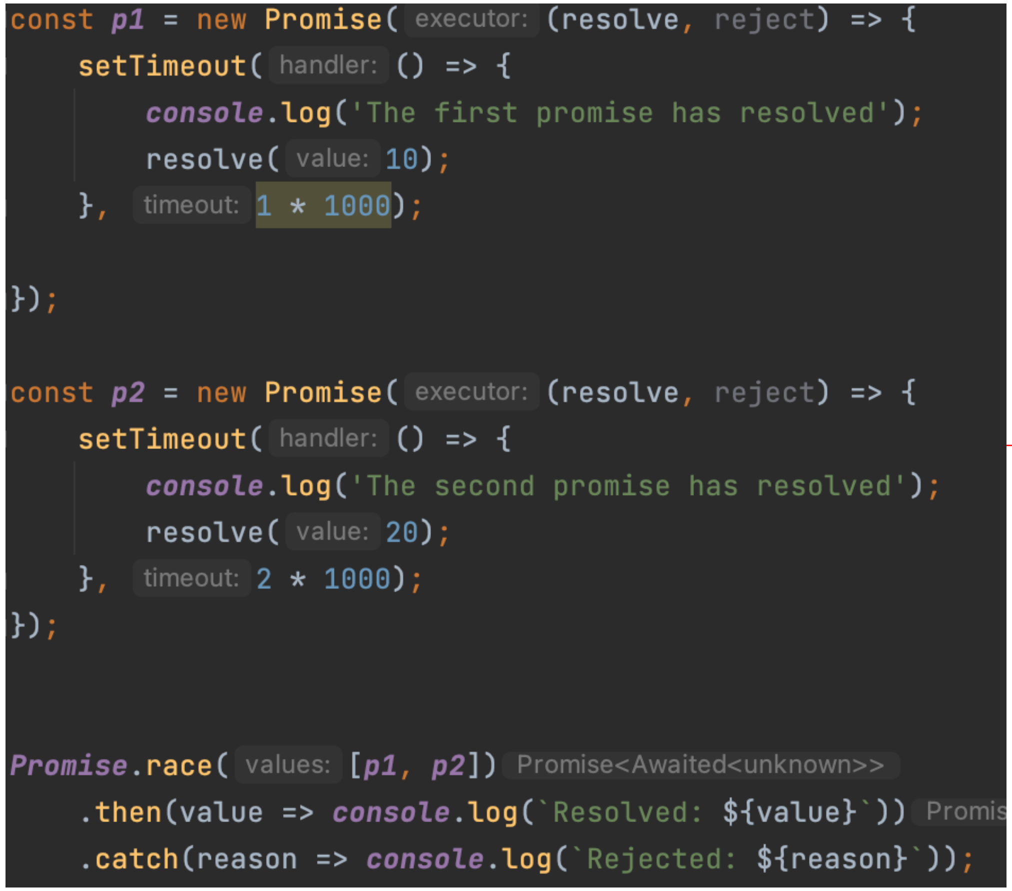


Promise.race([promise1,promise2]) returns a new promise that is **fulfilled** with the value **v1** at **t1**

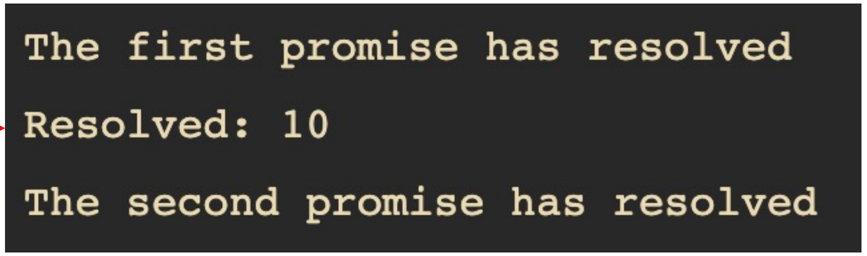


Promise.race([promise1,promise2]) returns a new promise that is **rejected** with the **error** at **t1**

JavaScript Promise.race()



```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The first promise has resolved');
        resolve( value: 10);
    }, timeout: 1 * 1000);

});

const p2 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The second promise has resolved');
        resolve( value: 20);
    }, timeout: 2 * 1000);
});


Promise.race( values: [p1, p2]) Promise<Awaited<unknown>>
    .then(value => console.log(`Resolved: ${value}`)) Promis
    .catch(reason => console.log(`Rejected: ${reason}`));
```



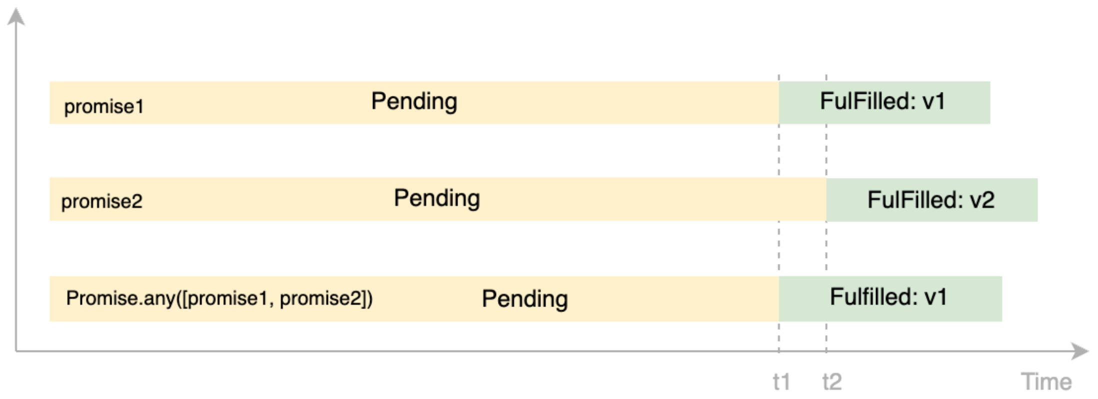
```
The first promise has resolved
Resolved: 10
The second promise has resolved
```

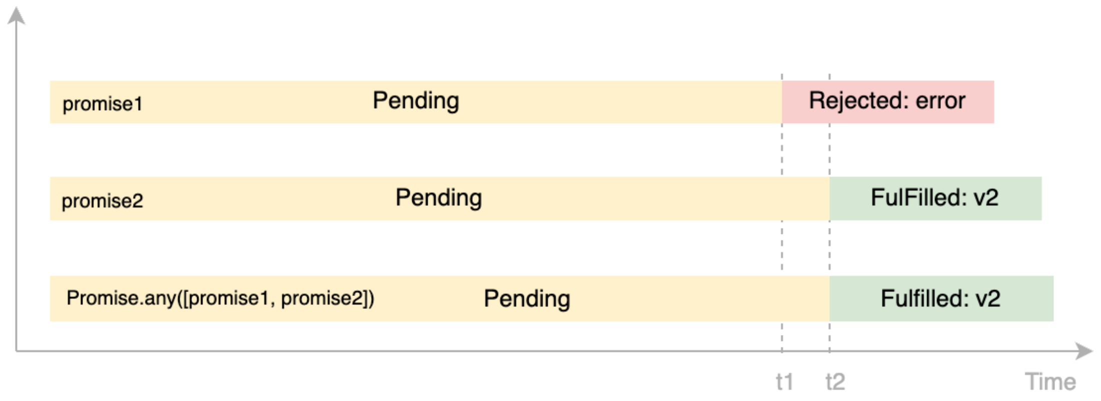
Promise.any()

JavaScript Promise.any()

The **Promise.any**() method accepts a list of promises. If one of the promises is fulfilled, the **Promise.any**() returns a single promise that resolves to a value which is the result of the fulfilled promise

```
Promise.any(iterable);
```

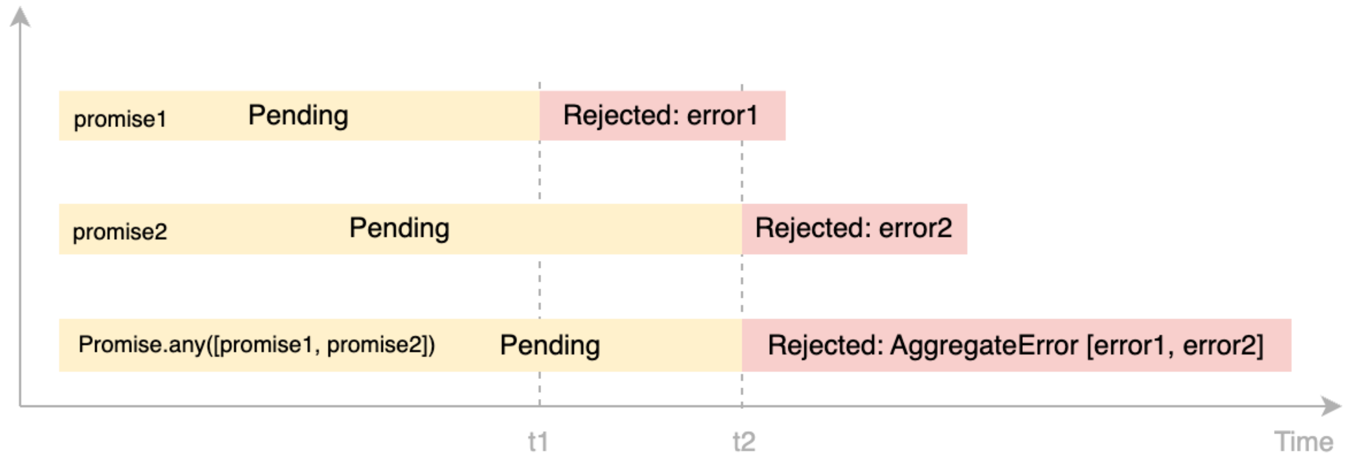




Promise.any([promise1,promise2]) returns a new promise that is **fulfilled** with the value **v1** at **t1**

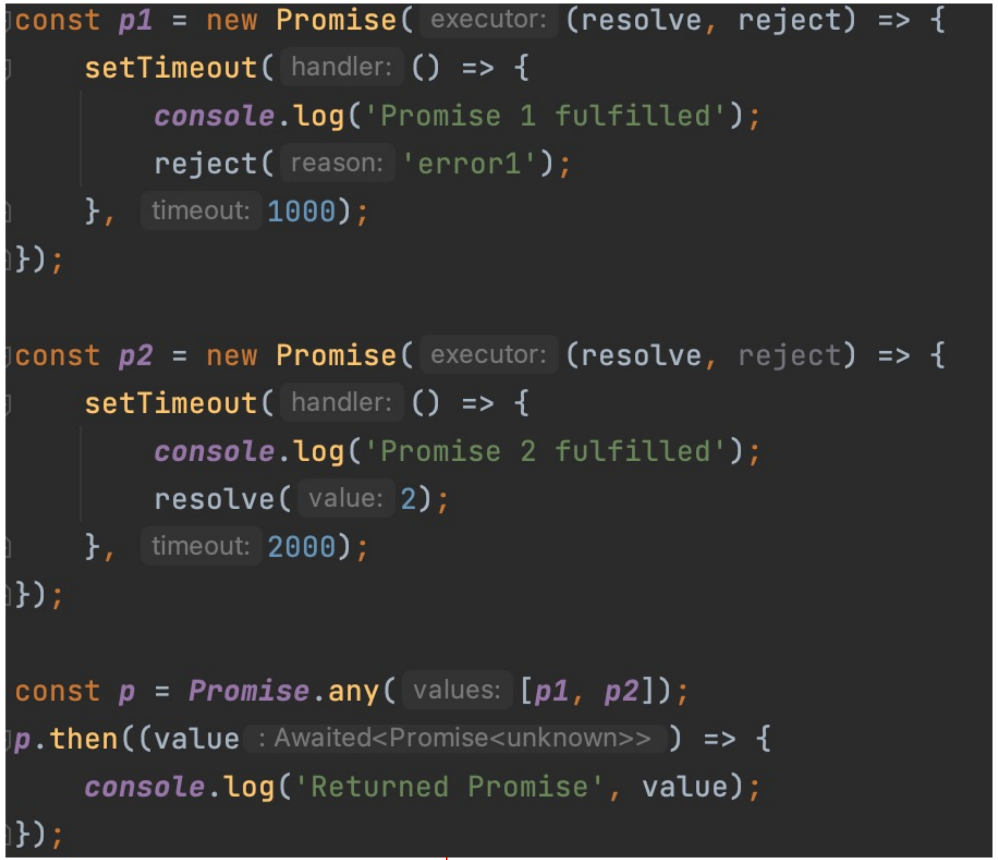
Promise.any([promise1,promise2]) returns a new promise that is **fulfilled** with the value **v2** at **t2** (Ignores the rejected of promise 1)

JavaScript Promise.any()

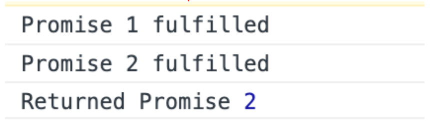


Promise.any([promise1,promise2]) returns an **AggregateError** containing the **error1** and **error2** of all the rejected promises at **t2**

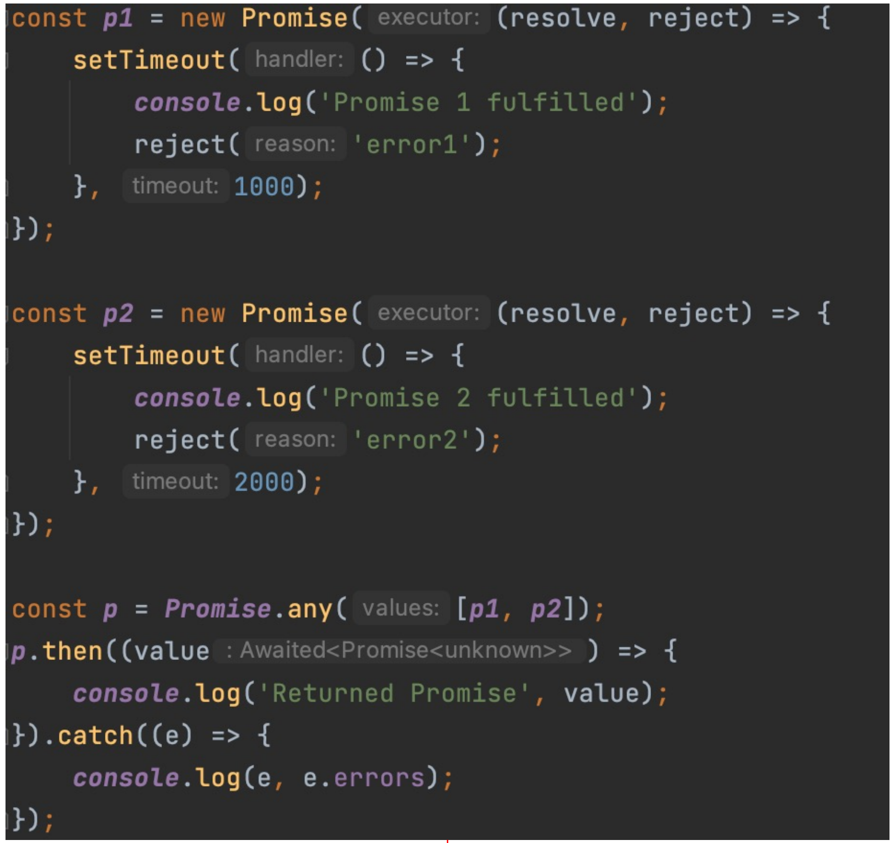
JavaScript Promise.any()



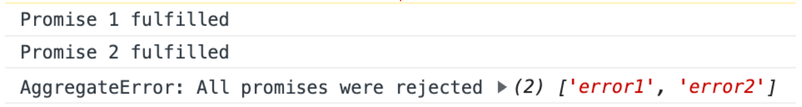
```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('Promise 1 fulfilled');
        reject( reason: 'error1');
    },  timeout: 1000);
});

const p2 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('Promise 2 fulfilled');
        resolve( value: 2);
    },  timeout: 2000);
});

const p = Promise.any( values: [p1, p2]);
p.then((value : Awaited<Promise<unknown>> ) => {
    console.log('Returned Promise', value);
});
```



```
Promise 1 fulfilled
Promise 2 fulfilled
Returned Promise 2
```



```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('Promise 1 fulfilled');
        reject( reason: 'error1');
    },  timeout: 1000);
});

const p2 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('Promise 2 fulfilled');
        reject( reason: 'error2');
    },  timeout: 2000);
});

const p = Promise.any( values: [p1, p2]);
p.then((value : Awaited<Promise<unknown>> ) => {
    console.log('Returned Promise', value);
}).catch((e) => {
    console.log(e, e.errors);
});
```



```
Promise 1 fulfilled
Promise 2 fulfilled
AggregateError: All promises were rejected ▶(2) ['error1', 'error2']
```

Promise.allSettled()

JavaScript Promise.allSettled()

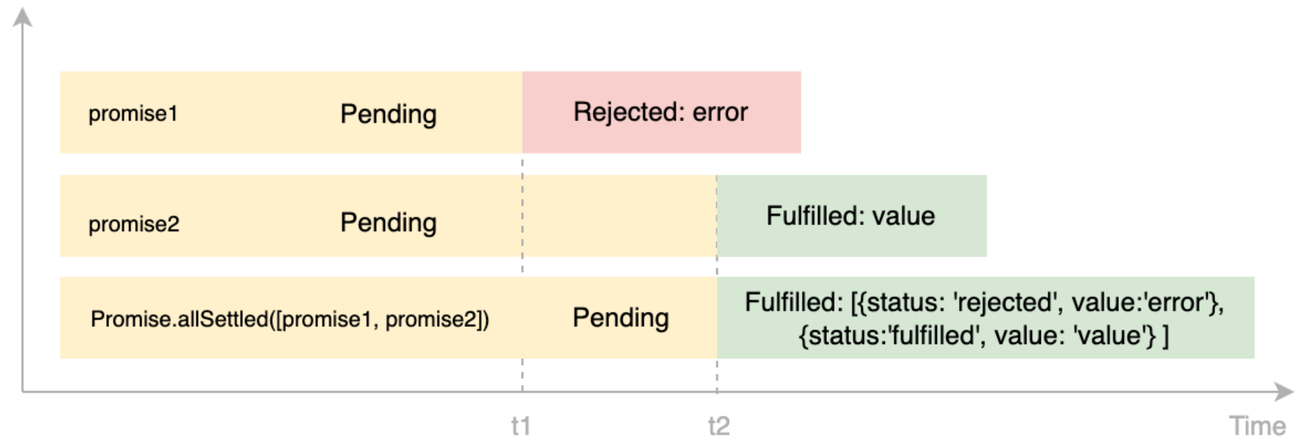
The **Promise.allSettled**() method accepts a list of promises and returns a new promise that resolves after all the input promises have settled, either resolved or rejected

The **Promise.allSettled**() method returns a promise that resolves to **an array of objects** that each describes the result of the input promise. Each object has two properties: **status** and **value** (or **reason**).

- The **status** can be either fulfilled or rejected
- The **value** in case the promise is fulfilled or **reason** if the promise is rejected.

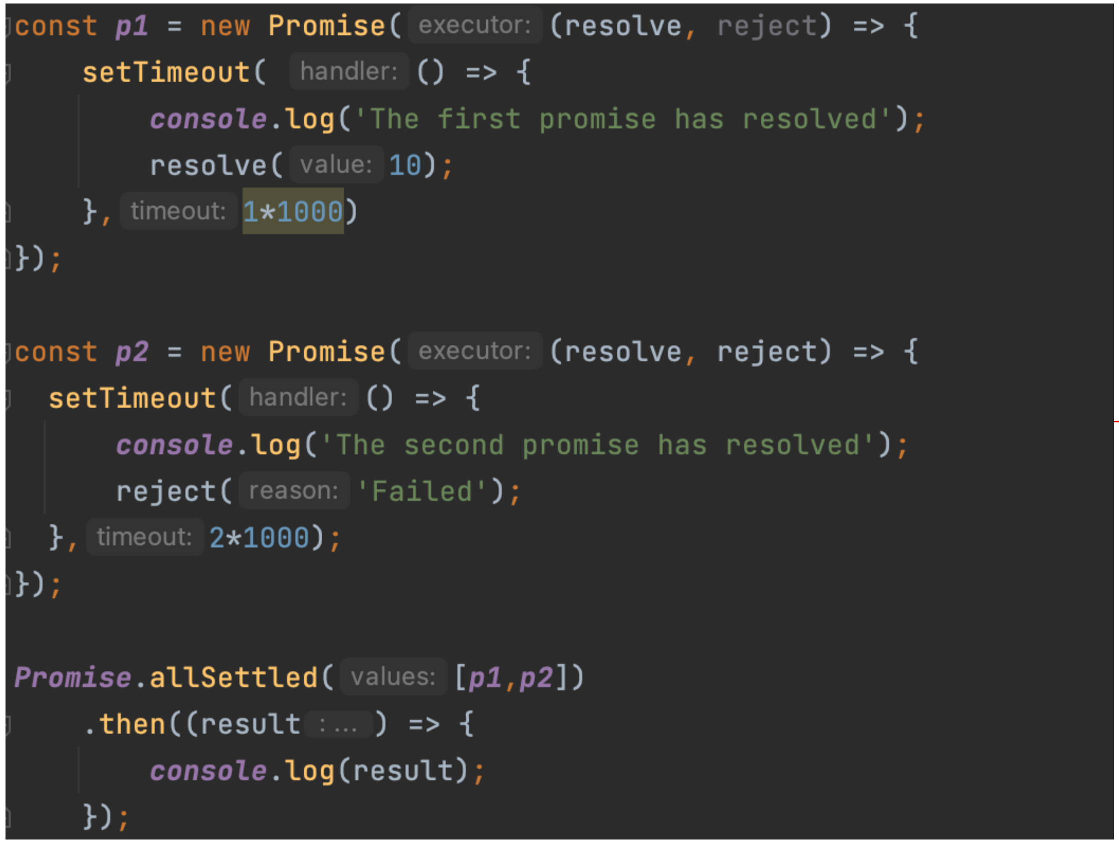
```
Promise.allSettled(iterable);
```

JavaScript Promise.allSettled()

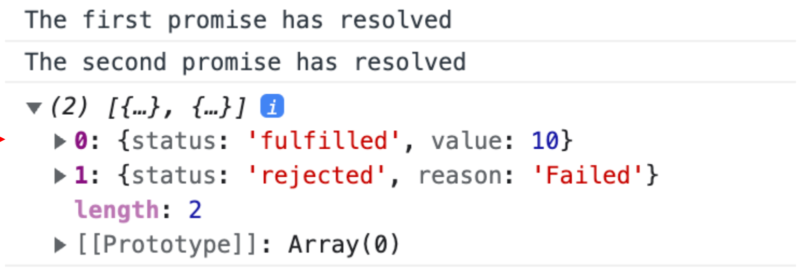

promise1
Pending
Rejected: error
promise2
Pending
Fulfilled: value
Promise.allSettled([promise1, promise2])
Pending
Fulfilled: [{status: 'rejected', value:'error'}, {status:'fulfilled', value: 'value'}]
t1
t2
Time

Promise.allSettled([promise1,promise2]) returns an **array** containing **objects** that describe the statuses and outcomes of the promise1 and promise2

JavaScript Promise.allSettled()



```
const p1 = new Promise( executor: (resolve, reject) => {
    setTimeout( handler: () => {
        console.log('The first promise has resolved');
        resolve( value: 10);
    }, timeout: 1*1000)
});

const p2 = new Promise( executor: (resolve, reject) => {
  setTimeout( handler: () => {
      console.log('The second promise has resolved');
      reject( reason: 'Failed');
  }, timeout: 2*1000);
});

Promise.allSettled( values: [p1,p2])
    .then((result : ... ) => {
        console.log(result);
    });
```



```
The first promise has resolved
The second promise has resolved
▼(2) [{…}, {…}] i
  ▶0: {status: 'fulfilled', value: 10}
  ▶1: {status: 'rejected', reason: 'Failed'}
   length: 2
  ▶[[Prototype]]: Array(0)
```

Promise Error Handling

JavaScript Promise Error Handling - Normal error

When you raise an exception outside the promise, you must catch it with **try/catch**

```
function getUserById(id) {
    if (typeof id !== 'number' || id <= 0) {
        throw new Error('Invalid id argument');
    }

    return new Promise((resolve, reject) => {
        resolve({
            id: id,
            username: 'admin'
        });
    });
}
```

```
try {
    getUserById('a')
        .then(user => console.log(user.username))
        .catch(err => console.log(`Caught by .catch ${error}`));
} catch (error) {
    console.log(`Caught by try/catch ${error}`);
}
```

```
Caught by try/catch Error: Invalid id argument
```

JavaScript Promise Error Handling – Errors inside the Promises

If you throw an error inside the promise, the **catch**() method will catch it, not the **try/catch**.

Note: Throwing an error has the same effect as calling the **reject**().

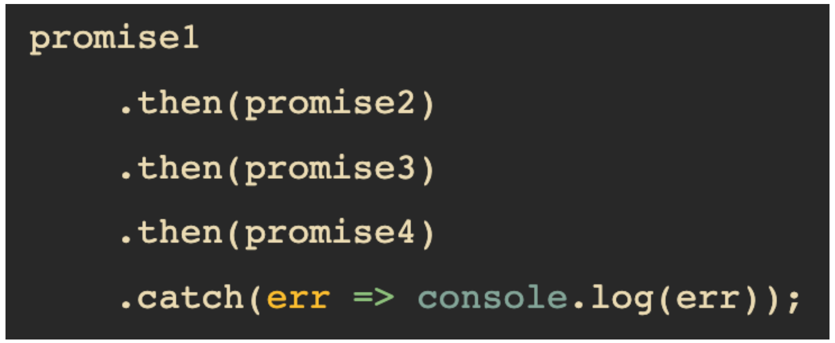
```
let authorized = false;

function getUserById(id) {
    return new Promise((resolve, reject) => {
        if (!authorized) {
            throw new Error('Unauthorized access to the user data');
        }

        resolve({
            id: id,
            username: 'admin'
        });
    });
}
```

```
try {
    getUserById(10)
        .then(user => console.log(user.username))
        .catch(err => console.log(`Caught by .catch ${error}`));
} catch (error) {
    console.log(`Caught by try/catch ${error}`);
}
```

```
Caught by .catch Error: Unauthorized access to the user data
```

JavaScript Promise Error Handling – Errors inside the Promises

If you chain promises, the **catch**() method will catch errors that occurred in any promise.



```
promise1
    .then(promise2)
    .then(promise3)
    .then(promise4)
    .catch(err => console.log(err));
```

Async/Await

JavaScript Async/Await

In the past, to deal with **asynchronous operations**, you often used the **callback functions**. However, when you **nest many callback functions,** the code will be more **difficult to maintain**. And you end up with a notorious issue which is known as the **callback hell.**

→ To avoid this **callback hell** issue, ES6 introduced the **promises** that allow you to write asynchronous code in more manageable ways.

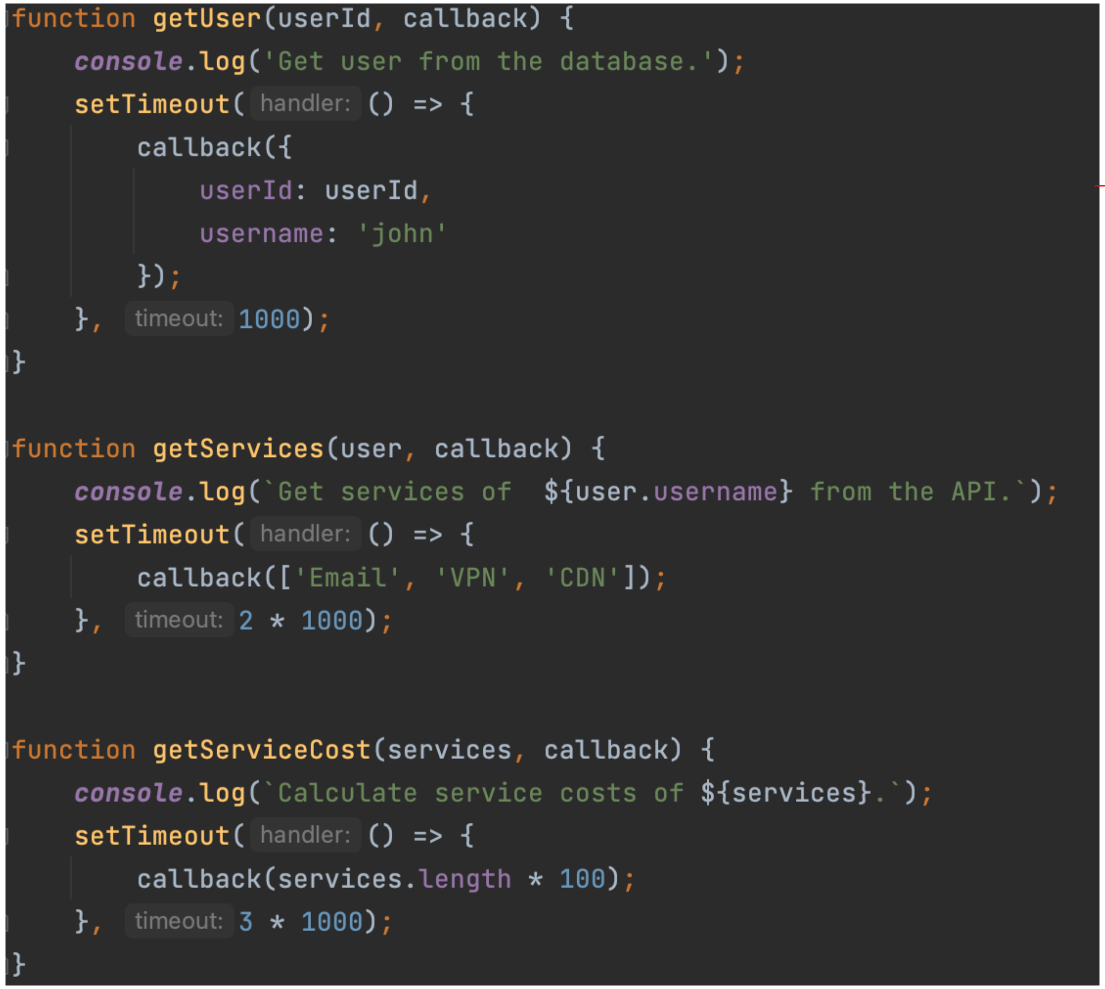
JavaScript Async/Await

Suppose that you need to perform three asynchronous operations in the following sequence:

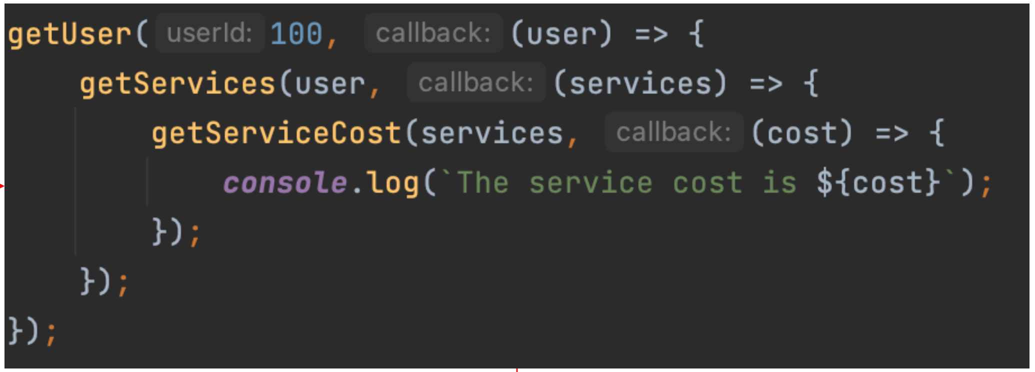
1. Select a user from the database
2. Get services of the user from an API
3. Calculate the service cost based on the services from the server

JavaScript Async/Await

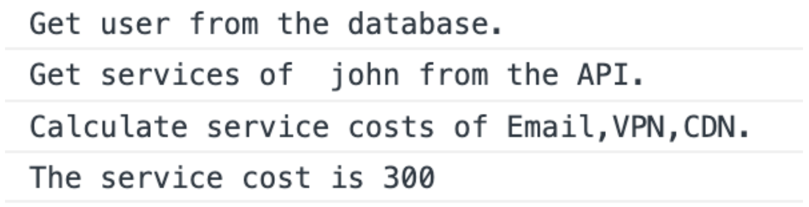
Callback Approach



```
function getUser(userId, callback) {
    console.log('Get user from the database.');
    setTimeout( handler: () => {
        callback({
            userId: userId,
            username: 'john'
        });
    },  timeout: 1000);
}

function getServices(user, callback) {
    console.log(`Get services of  ${user.username} from the API.`);
    setTimeout( handler: () => {
        callback(['Email', 'VPN', 'CDN']);
    },  timeout: 2 * 1000);
}

function getServiceCost(services, callback) {
    console.log(`Calculate service costs of ${services}.`);
    setTimeout( handler: () => {
        callback(services.length * 100);
    },  timeout: 3 * 1000);
}
```



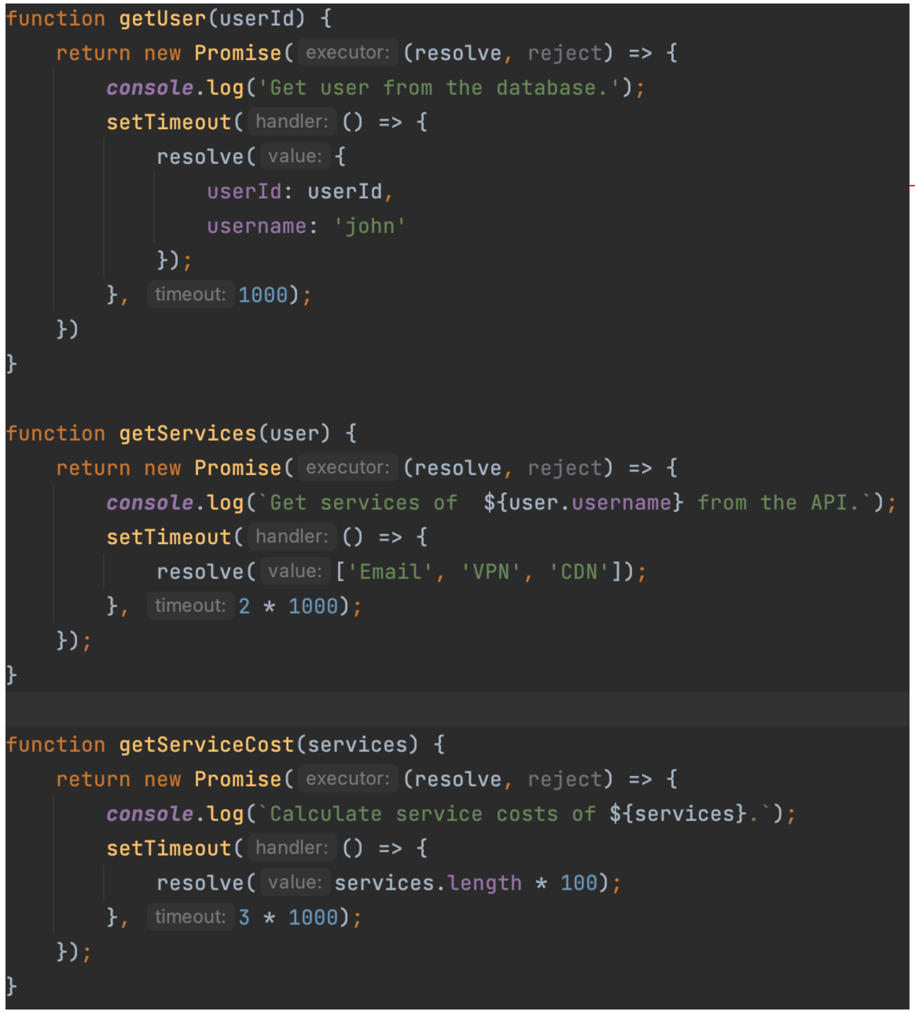
```
getUser( userId: 100,  callback: (user) => {
    getServices(user,  callback: (services) => {
        getServiceCost(services,  callback: (cost) => {
            console.log(`The service cost is ${cost}`);
        });
    });
});
```



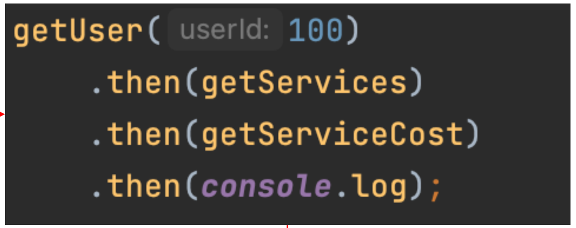
```
Get user from the database.
Get services of  john from the API.
Calculate service costs of Email,VPN,CDN.
The service cost is 300
```

JavaScript Async/Await

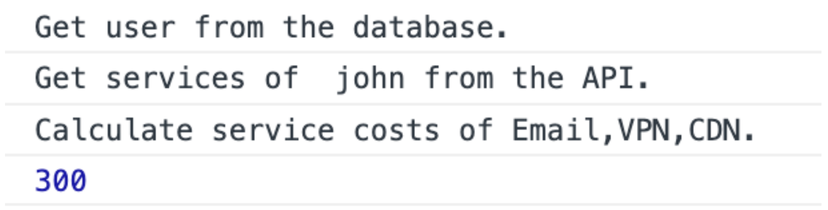
Promise Approach



```
function getUser(userId) {
    return new Promise( executor: (resolve, reject) => {
        console.log('Get user from the database.');
        setTimeout( handler: () => {
            resolve( value: {
                userId: userId,
                username: 'john'
            });
        }, timeout: 1000);
    })
}

function getServices(user) {
    return new Promise( executor: (resolve, reject) => {
        console.log(`Get services of  ${user.username} from the API.`);
        setTimeout( handler: () => {
            resolve( value: ['Email', 'VPN', 'CDN']);
        }, timeout: 2 * 1000);
    });
}

function getServiceCost(services) {
    return new Promise( executor: (resolve, reject) => {
        console.log(`Calculate service costs of ${services}.`);
        setTimeout( handler: () => {
            resolve( value: services.length * 100);
        }, timeout: 3 * 1000);
    });
}
```



```
getUser( userId: 100)
    .then(getServices)
    .then(getServiceCost)
    .then(console.log);
```

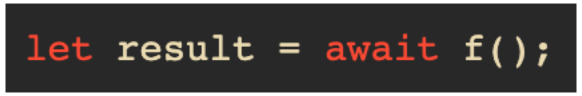


```
Get user from the database.
Get services of  john from the API.
Calculate service costs of Email,VPN,CDN.
300
```

JavaScript Async/Await

ES2017 introduced the **async/await** keywords that build on top of **promises**, allowing you to write **asynchronous code** that looks more like **synchronous code** and is more readable

If a function returns a **Promise**, you can place the **await** keyword in front of the function call

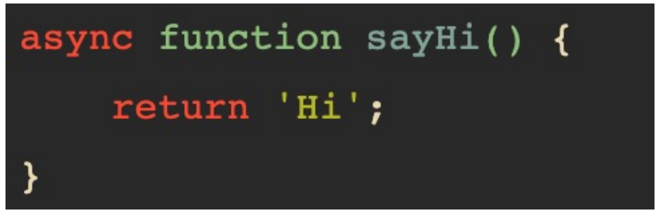


```
let result = await f();
```

The **await** will wait for the **Promise** returned from the f() to settle. The **await** keyword can be used only inside the **async functions**

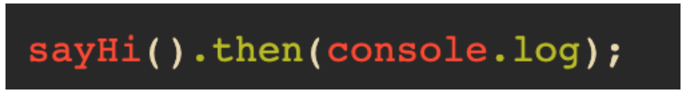
JavaScript Async/Await

To define an **async function**, you place the **async** keyword in front of the **function** keyword



```
async function sayHi() {
    return 'Hi';
}
```

Asynchronous functions always return a **Promise**



```
sayHi().then(console.log);
```

JavaScript Async/Await

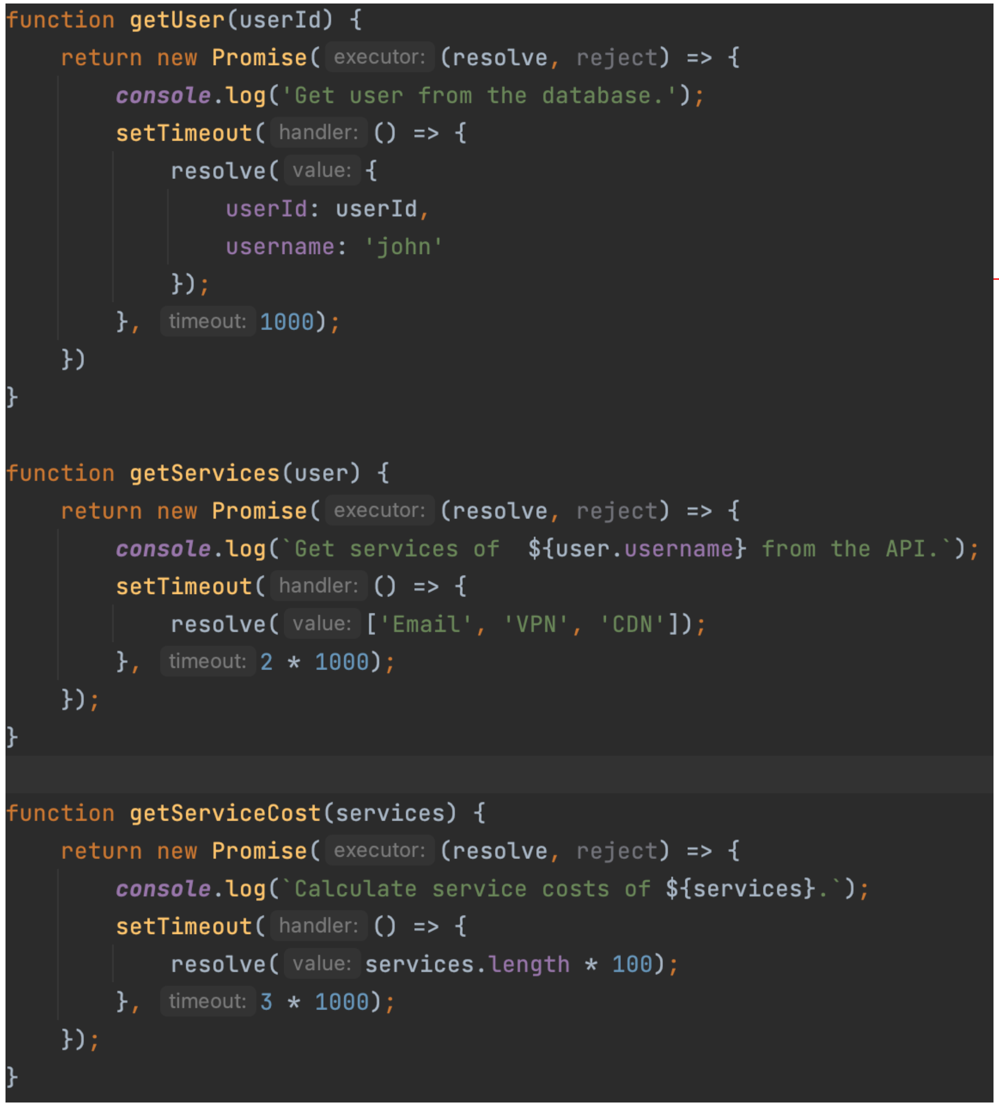
Besides the regular functions, you can use the **async** keyword in the function expressions, arrow functions, and methods of classes

```
let sayHi = async function () {
    return 'Hi';
}
```

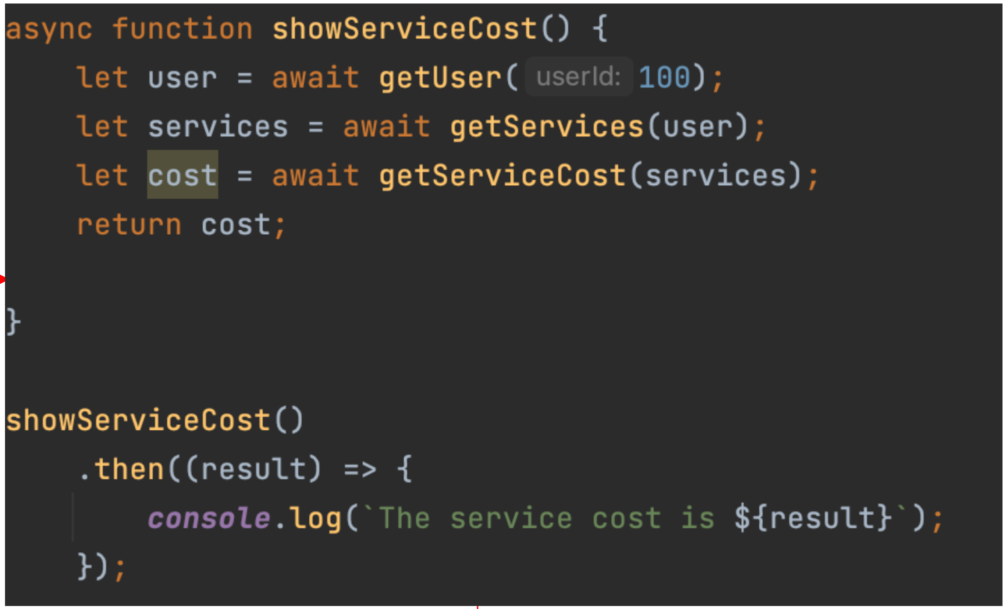
```
let sayHi = async () => 'Hi';
```

```
class Greeter {
    async sayHi() {
        return 'Hi';
    }
}
```

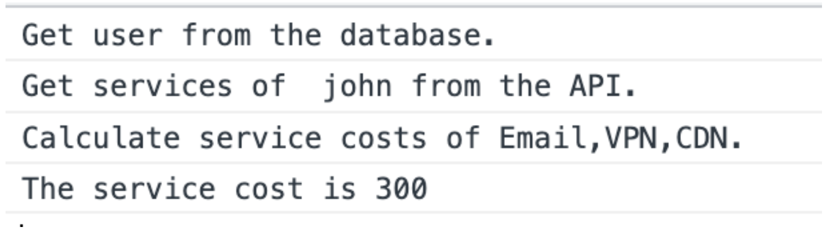
JavaScript Async/Await



```
function getUser(userId) {
    return new Promise( executor: (resolve, reject) => {
        console.log('Get user from the database.');
        setTimeout( handler: () => {
            resolve( value: {
                userId: userId,
                username: 'john'
            });
        }, timeout: 1000);
    })
}

function getServices(user) {
    return new Promise( executor: (resolve, reject) => {
        console.log(`Get services of  ${user.username} from the API.`);
        setTimeout( handler: () => {
            resolve( value: ['Email', 'VPN', 'CDN']);
        }, timeout: 2 * 1000);
    });
}

function getServiceCost(services) {
    return new Promise( executor: (resolve, reject) => {
        console.log(`Calculate service costs of ${services}.`);
        setTimeout( handler: () => {
            resolve( value: services.length * 100);
        }, timeout: 3 * 1000);
    });
}
```



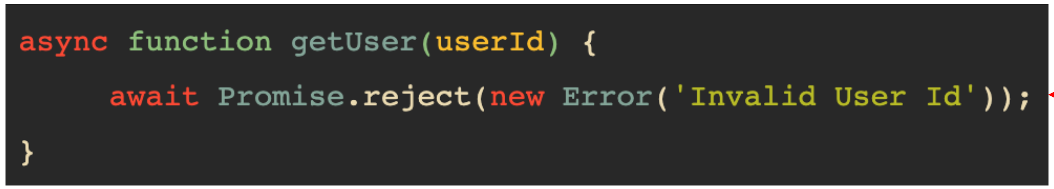
```
async function showServiceCost() {
    let user = await getUser( userId: 100);
    let services = await getServices(user);
    let cost = await getServiceCost(services);
    return cost;

}

showServiceCost()
    .then((result) => {
        console.log(`The service cost is ${result}`);
    });
```



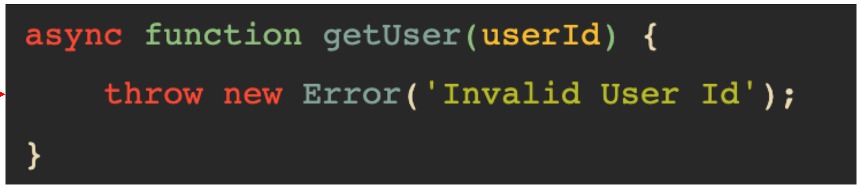
```
Get user from the database.
Get services of  john from the API.
Calculate service costs of Email,VPN,CDN.
The service cost is 300
```

JavaScript Async/Await - Error handling

If a promise **resolves**, the **await promise** returns the result. However, when the **promise rejects**, the await promise will **throw an error** as if there were a throw statement



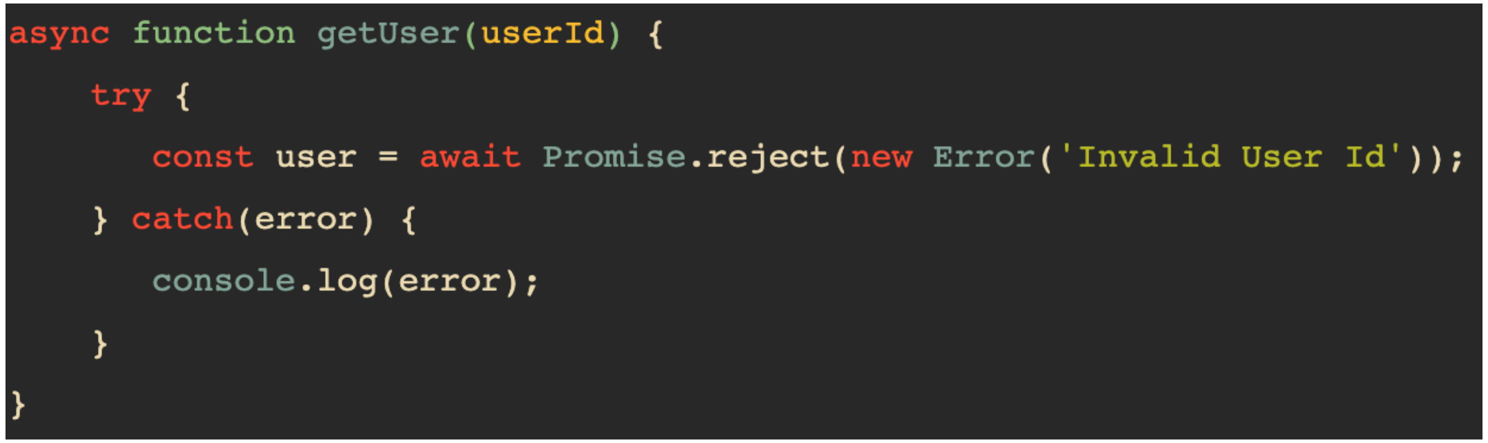
```javascript
async function getUser(userId) {
    await Promise.reject(new Error('Invalid User Id'));
}
```



```javascript
async function getUser(userId) {
    throw new Error('Invalid User Id');
}
```

JavaScript Async/Await - Error handling

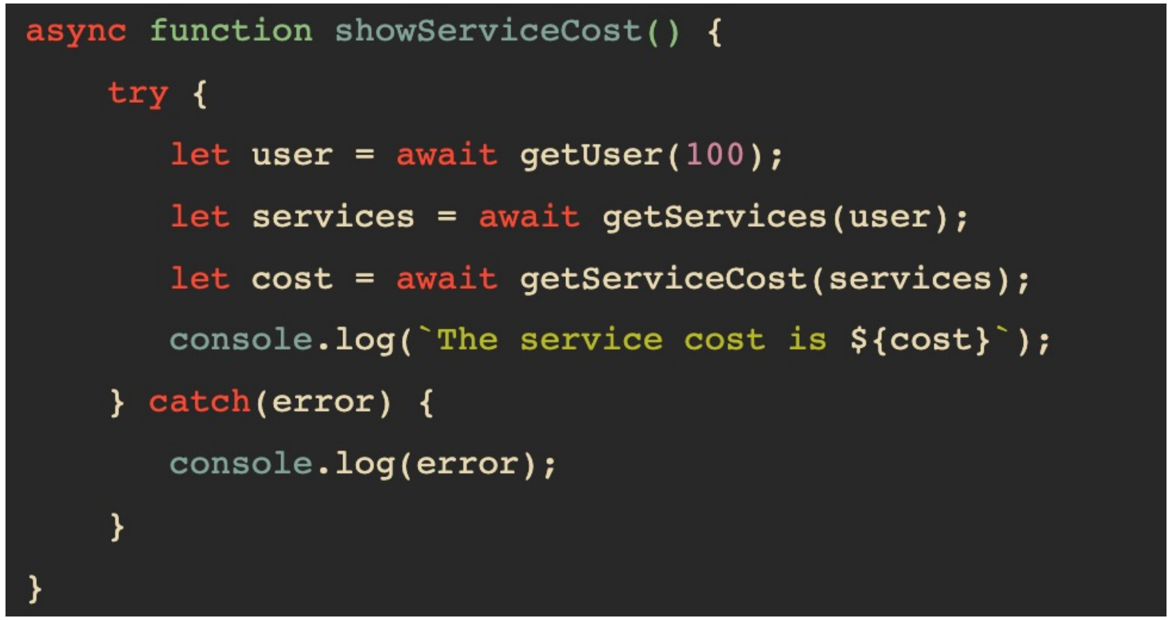
We can catch the error by using the **try...catch** statement, the same way as a regular throw statement:



```
async function getUser(userId) {
    try {
        const user = await Promise.reject(new Error('Invalid User Id'));
    } catch(error) {
        console.log(error);
    }
}
```

JavaScript Async/Await - Error handling

It’s possible to catch errors caused by one or more **await promise**:


```
async function showServiceCost() {
    try {
        let user = await getUser(100);
        let services = await getServices(user);
        let cost = await getServiceCost(services);
        console.log(`The service cost is ${cost}`);
    } catch(error) {
        console.log(error);
    }
}
```

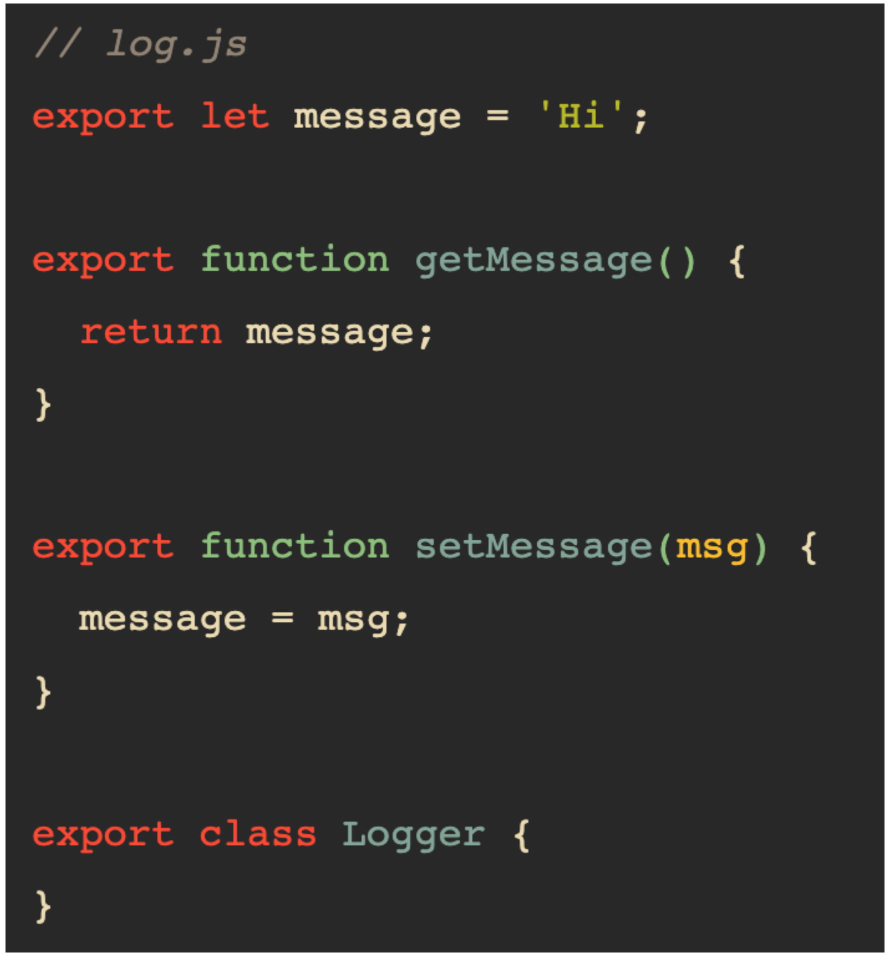
ES6 Module

JavaScript ES6 Module

- A **module** organizes a related set of JavaScript code.
- A **module** can contain variables and functions.
- A **module** is nothing more than a chunk of JavaScript code written in a file.
- By default, variables and functions of a **module** are **not available for use**. Variables and functions within a module should be **exported** so that they can be **accessed** from within other **modules**.
- Variables or functions declared in a **module** will not be accessible globally.

JavaScript ES6 Module - Export

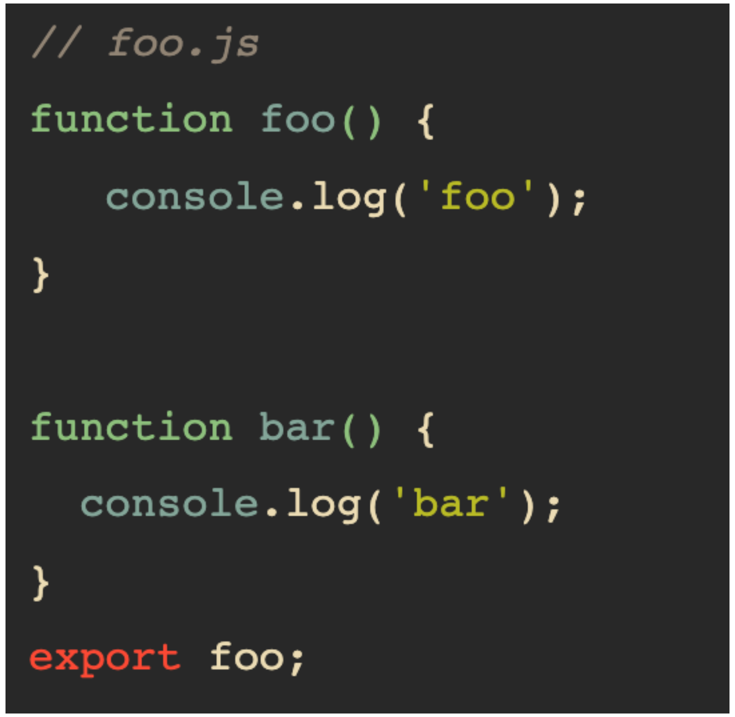
- To **export** a variable, a function, or a class, you place the **export** keyword in front of it
- Note that the **export** keyword requires the function or class **to have a name** to be exported. You **can't export an anonymous** function or class



```
// log.js
export let message = 'Hi';

export function getMessage() {
  return message;
}

export function setMessage(msg) {
  message = msg;
}

export class Logger {
}
```

JavaScript ES6 Module - Export

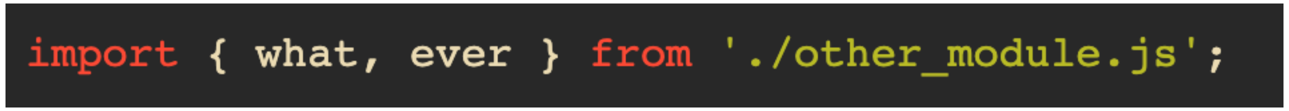
- You can define a variable, a function, or a class first and then **export it later.**



```
// foo.js
function foo() {
   console.log('foo');
}


function bar() {
  console.log('bar');
}
export foo;
```

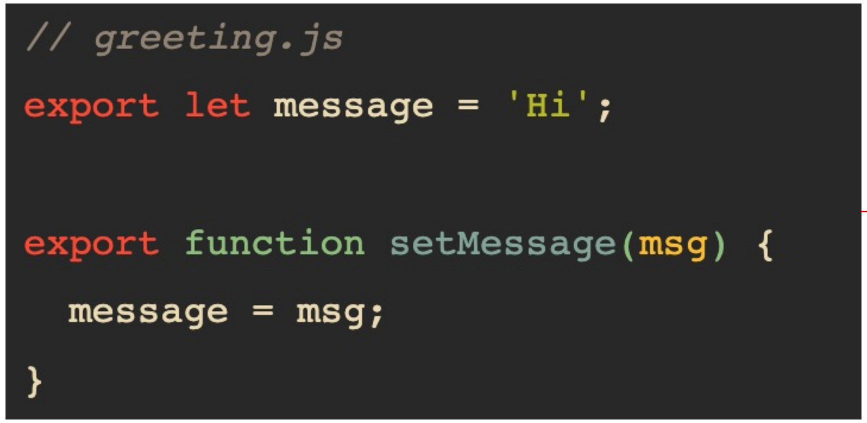
JavaScript ES6 Module - Importing

Once you define a module with **exports**, you can access the exported variables, functions, and classes in another module by using the **import** keyword

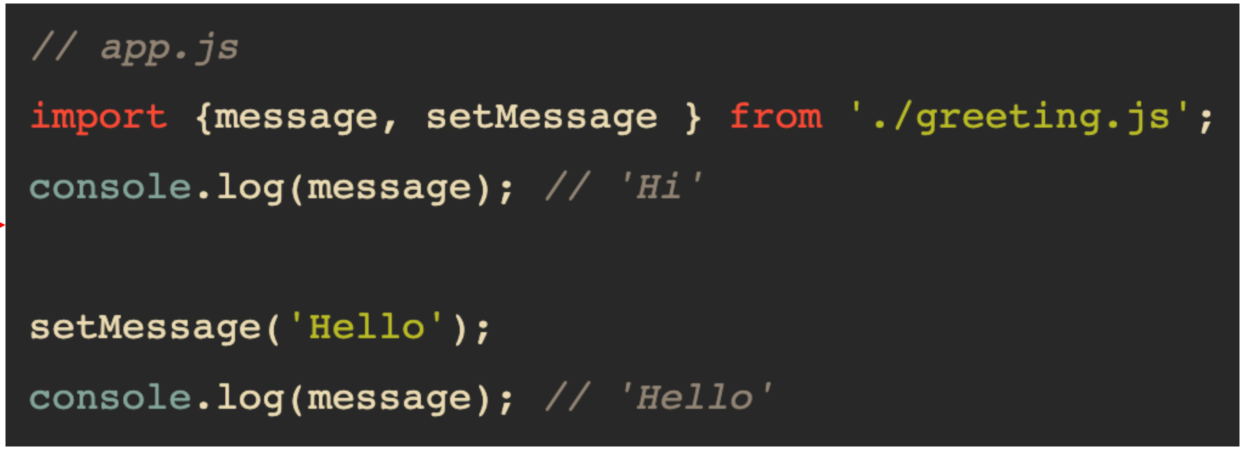


```
import { what, ever } from './other_module.js';
```

JavaScript ES6 Module



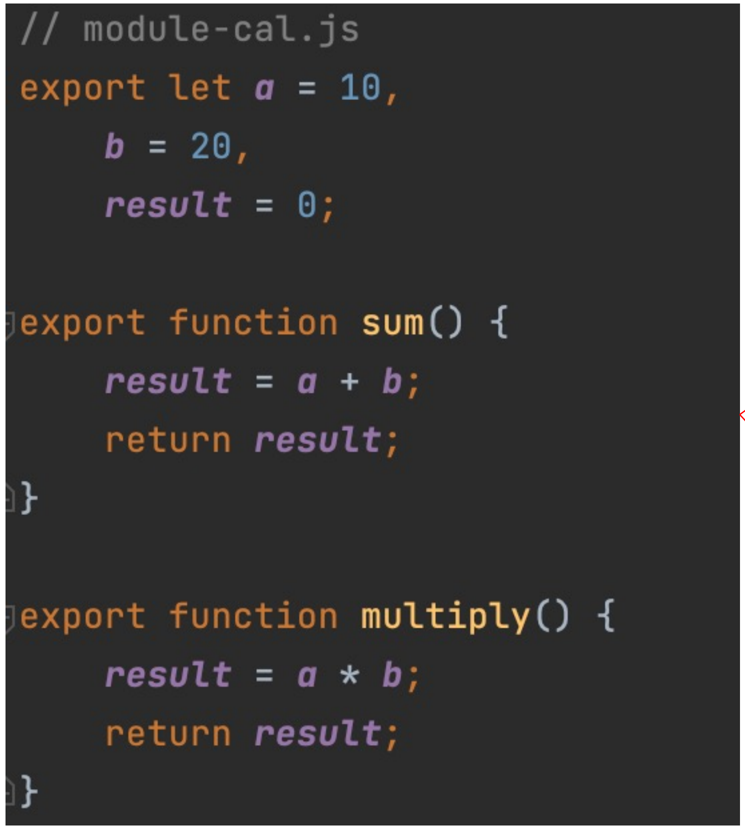
```
// greeting.js
export let message = 'Hi';

export function setMessage(msg) {
  message = msg;
}
```



```
// app.js
import {message, setMessage } from './greeting.js';
console.log(message); // 'Hi'

setMessage('Hello');
console.log(message); // 'Hello'
```

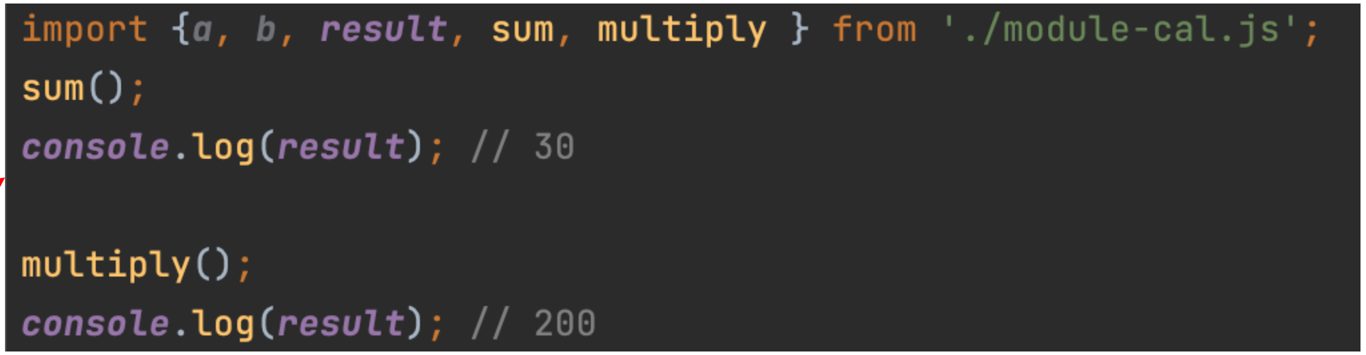
NOTE: You can’t change the value of the **message** variable directly => This will cause an error

JavaScript ES6 Module



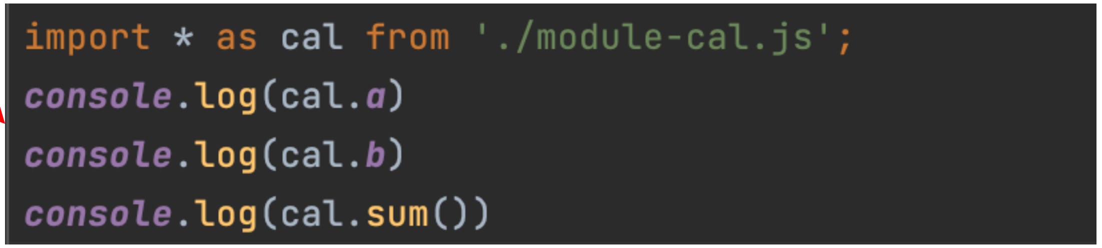
```
// module-cal.js
export let a = 10,
    b = 20,
    result = 0;

export function sum() {
    result = a + b;
    return result;
}

export function multiply() {
    result = a * b;
    return result;
}
```

Import multiple bindings



```
import {a, b, result, sum, multiply } from './module-cal.js';
sum();
console.log(result); // 30

multiply();
console.log(result); // 200
```

Import an entire module as an object



```
import * as cal from './module-cal.js';
console.log(cal.a)
console.log(cal.b)
console.log(cal.sum())
```

JavaScript ES6 Module - Aliasing

```javascript
// math.js
function add( a, b ) {
   return a + b;
}

export { add as sum };
```

```javascript
import { sum } from './math.js';
```

```javascript
import {sum as total} from './math.js';
```