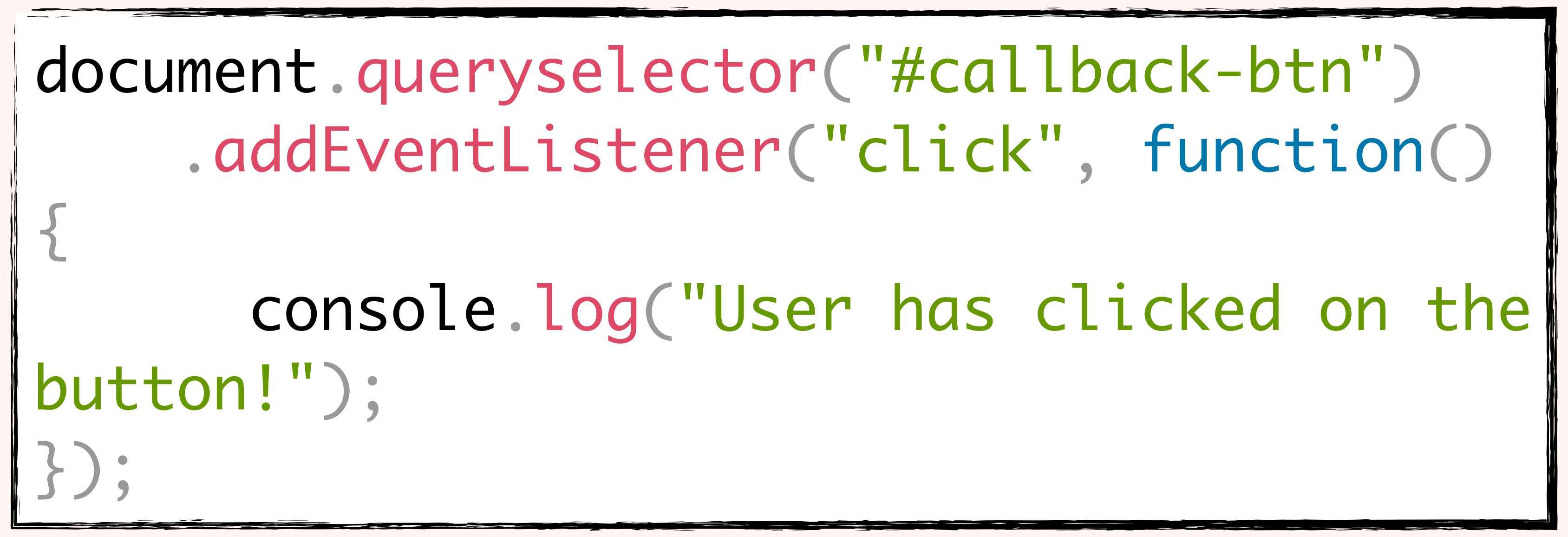
JAVASCRIPT

CALLBACK

- In JavaScript, functions are objects. Can we pass objects to functions as parameters?
- Callback is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.



```
document.queryselector("#callback-btn")
    .addEventListener("click", function()
{
      console.log("User has clicked on the
button!");
});
```

Here we select the button first with its id, and then we add an event listener with the addEventListener method. It takes 2 parameters. The first one is its type, “click”, and the second parameter is a callback function, which logs the message when the button is clicked.

setTimeout

```
const message = function() {
    console.log("This message is shown after 3 seconds");
}

setTimeout(message, 3000);
```

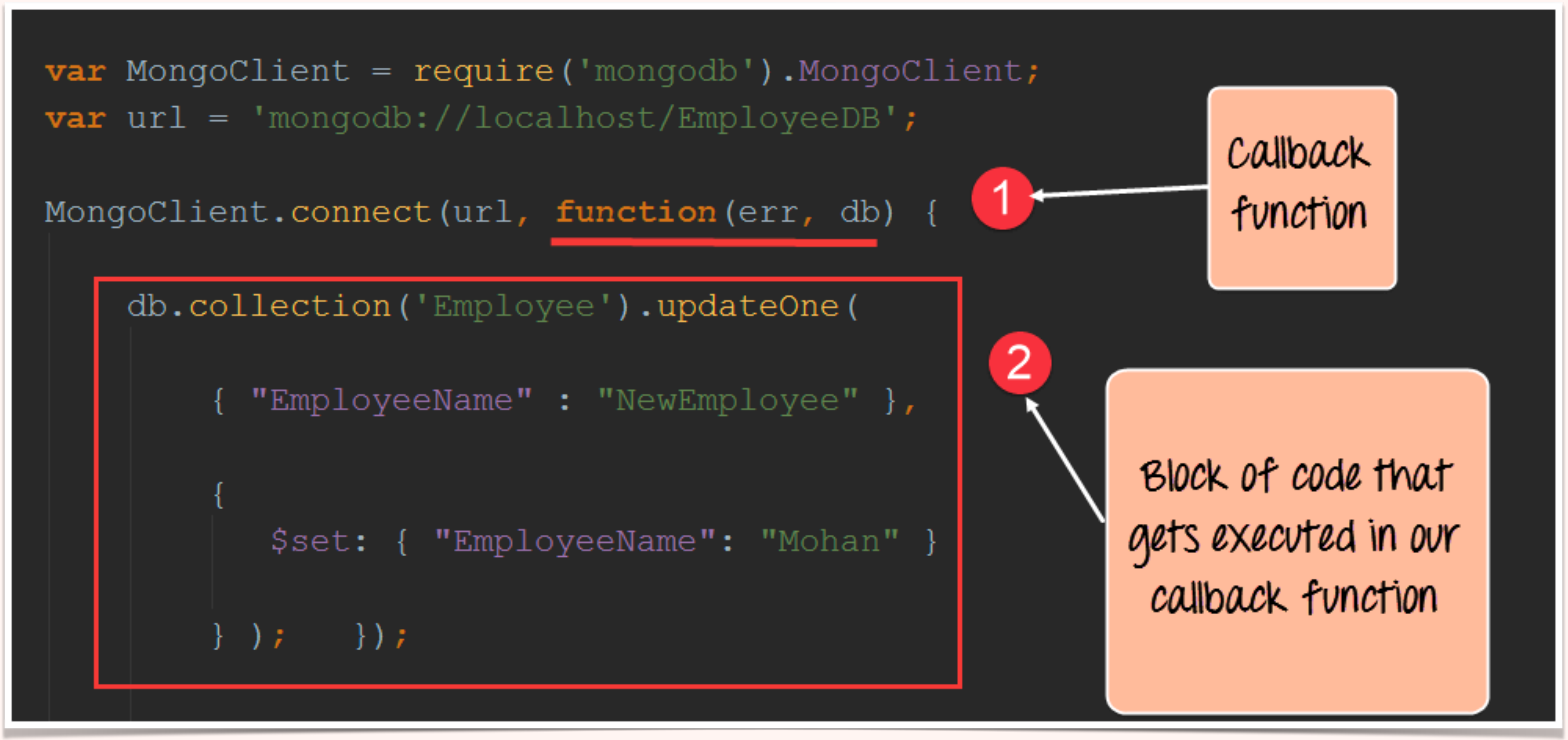
There is a built-in method in JavaScript called “setTimeout”, which calls a function or evaluates an expression after a given period of time (in milliseconds). So here, the “message” function is being called after 3 seconds have passed. (1 second = 1000 milliseconds)

ANONYMOUS FUNCTION

- Alternatively, we can define a function directly inside another function, instead of calling it. It will look like this:

```
setTimeout(function() {
    console.log("This message is shown after 3 seconds");
}, 3000);
```

EXAMPLE



```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

    db.collection('Employee').updateOne(

        { "EmployeeName" : "NewEmployee" },

        {
            $set: { "EmployeeName": "Mohan" }

        } );    });
```

HOW SYNCHRONOUS JS WORKS?

```
const second = () => {
  console.log('Hello there!');
}
const first = () => {
  console.log('Hi there!');
  second();
  console.log('The End');
}
first();
```

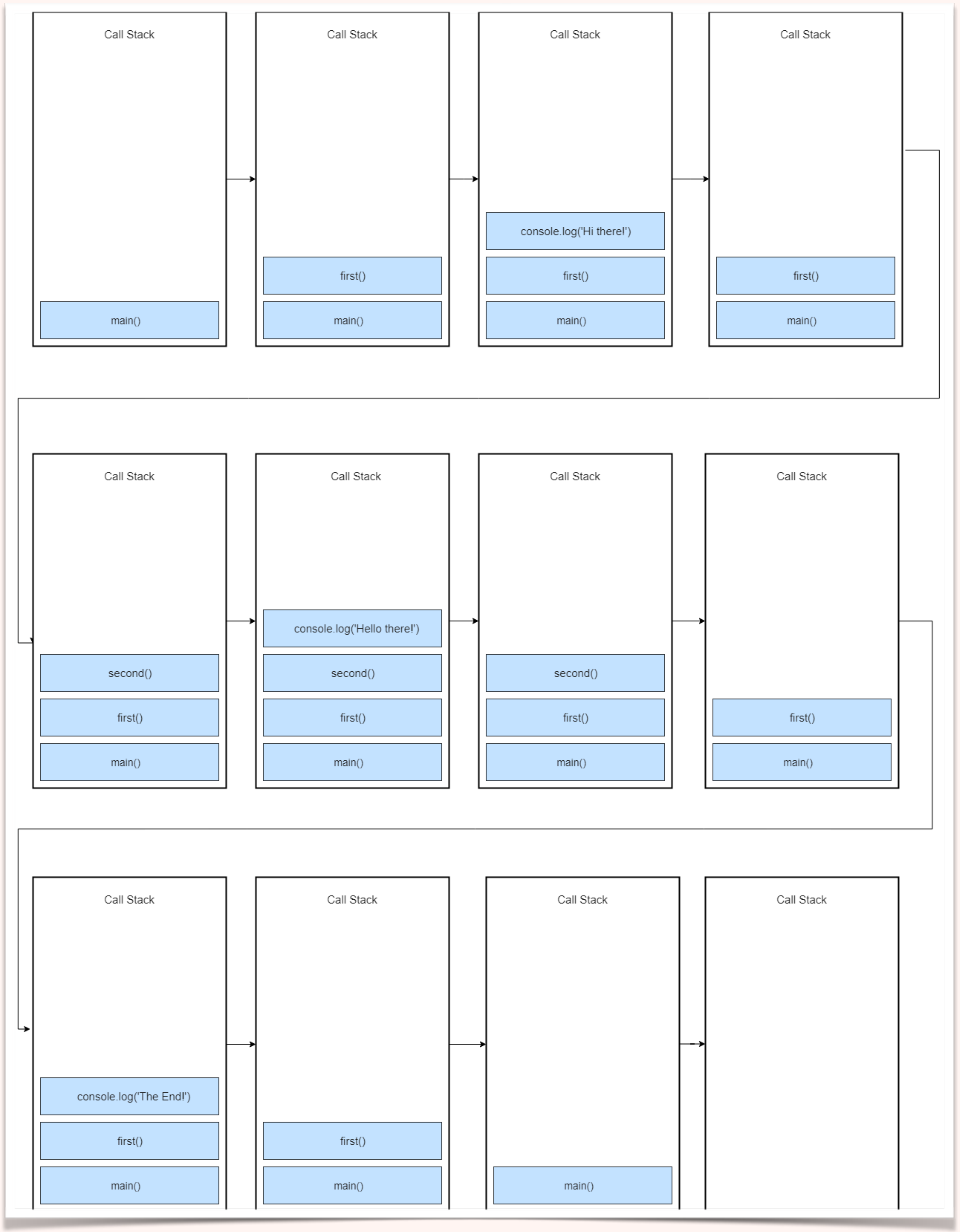
EXECUTION CONTEXT

- Execution Context is an abstract concept of an environment where the JavaScript code is evaluated and executed. Whenever any code is run in JavaScript, it's run inside an execution context.
- What about Call Stack?

CALL STACK

- Call stack as its name implies is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.
- JavaScript has a single call stack because it's a single-threaded programming language. The call stack has a LIFO structure which means that the items can be added or removed from the top of the stack only.

- When this code is executed, a global execution context is created (represented by main()) and pushed to the top of the call stack. When a call to first() is encountered, it's pushed to the top of the stack.
- Next, console.log('Hi there!') is pushed to the top of the stack, when it finishes, it's popped off from the stack. After it, we call second(), so the second() function is pushed to the top of the stack.
- console.log('Hello there!') is pushed to the top of the stack and popped off the stack when it finishes. The second() function finishes, so it's popped off the stack.
- console.log('The End') is pushed to the top of the stack and removed when it finishes. After it, the first() function completes, so it's removed from the stack.
- The program completes its execution at this point, so the global execution context(main()) is popped off from the stack.



PROMISE

- Promise is an object
- Similar to a promise in real life. When we make a promise in real life, it is a guarantee that we are going to do something in the future. Because promises can only be made for the future.
- .then() always returns a new promise

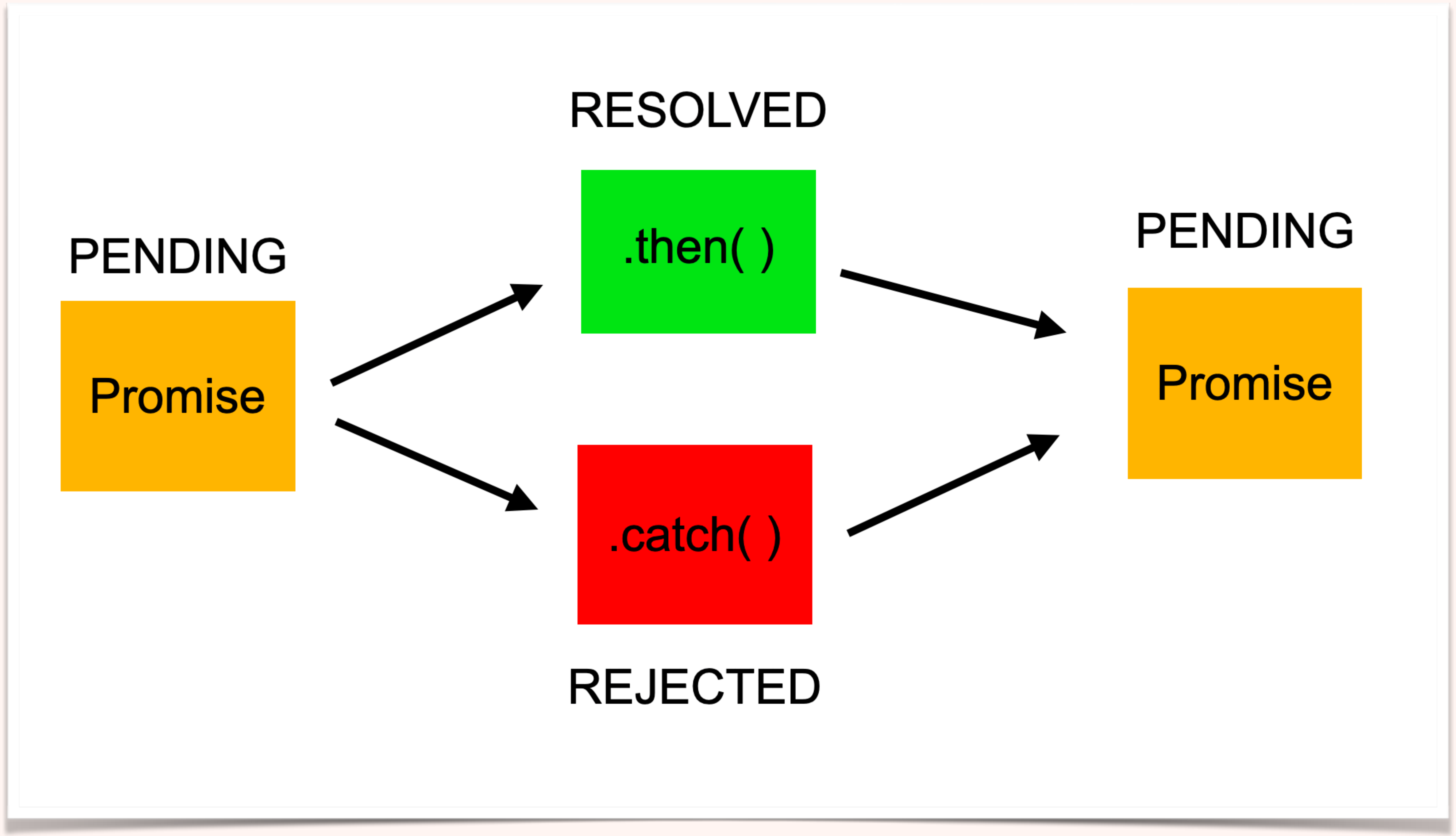
```
const wait = time => new Promise((resolve) =>
setTimeout(resolve, time));

wait(3000).then(() => console.log('Hello!')); // 'Hello!'
```

ASYNC/AWAIT

- Special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.

PROMISE STATES

- Pending: Initial State, before the Promise succeeds or fails
- Resolved: Completed Promise
- Rejected: Failed Promise



CREATING AND USING PROMISE

```
const myPromise = new Promise((resolve, reject) => {
    const condition = true;

    if(condition) {
        resolve('Promise is resolved successfully.');
    } else {
        reject('Promise is rejected');
    }
});

myPromise.then((message) => {
    console.log(message);
}).catch((message) => {
    console.log(message);
});
```