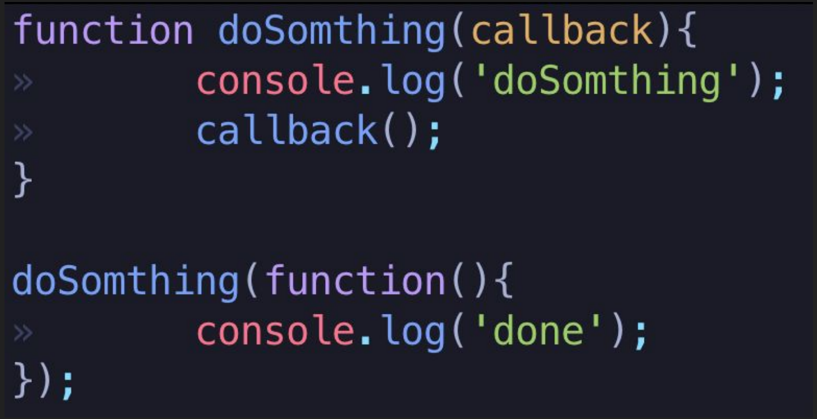
# CallBack Function

# What is CallBack function?

A callback function is a function passed into another function as an argument

# Sample



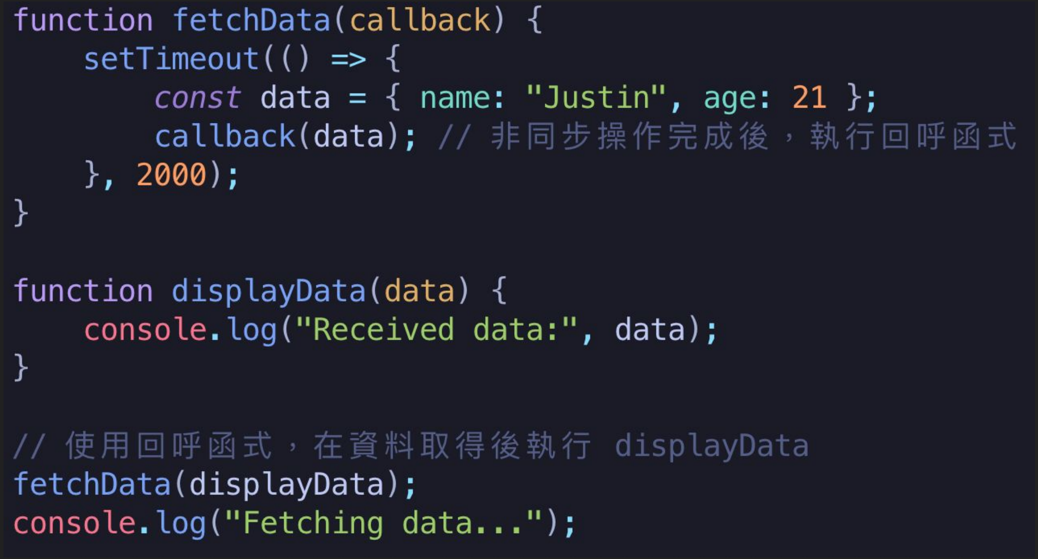
```
function doSomthing(callback){
        console.log('doSomthing');
        callback();
}

doSomthing(function(){
        console.log('done');
});
```

# Why we need CallBack function?

In JavaScript, many operations
(such as reading files or sending api requests) are asynchronous operation.

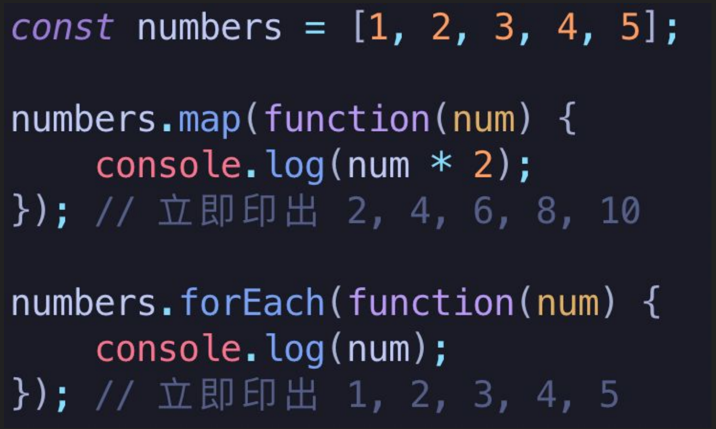
Callback functions allow us to “ handle the results ” of asynchronous operations
without blocking the program.

# Sample

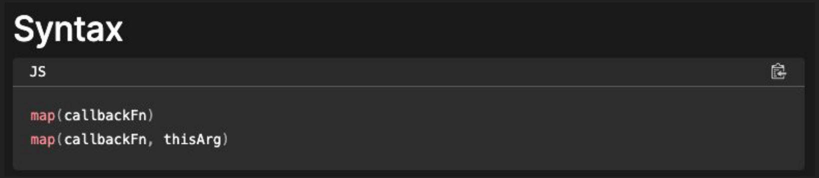


```
function fetchData(callback) {
    setTimeout(() => {
        const data = { name: "Justin", age: 21 };
        callback(data); // 非同步操作完成後，執行回呼函式
    }, 2000);
}

function displayData(data) {
    console.log("Received data:", data);
}

// 使用回呼函式，在資料取得後執行 displayData
fetchData(displayData);
console.log("Fetching data...");
```

# Synchronous Callback v.s. Asynchronous Callback



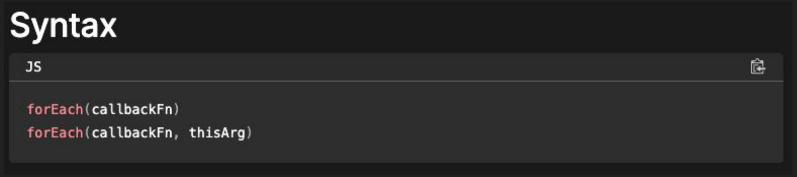
```
const numbers = [1, 2, 3, 4, 5];

numbers.map(function(num) {
    console.log(num * 2);
}); // 立即印出 2, 4, 6, 8, 10

numbers.forEach(function(num) {
    console.log(num);
}); // 立即印出 1, 2, 3, 4, 5
```



## Syntax

JS

```
map(callbackFn)
map(callbackFn, thisArg)
```



## Syntax

JS

```
forEach(callbackFn)
forEach(callbackFn, thisArg)
```

# When to Use Callback Functions?

1. I/O Operations
2. Server Request
3. setTimeOut
4. Event handling
5. Error Handling

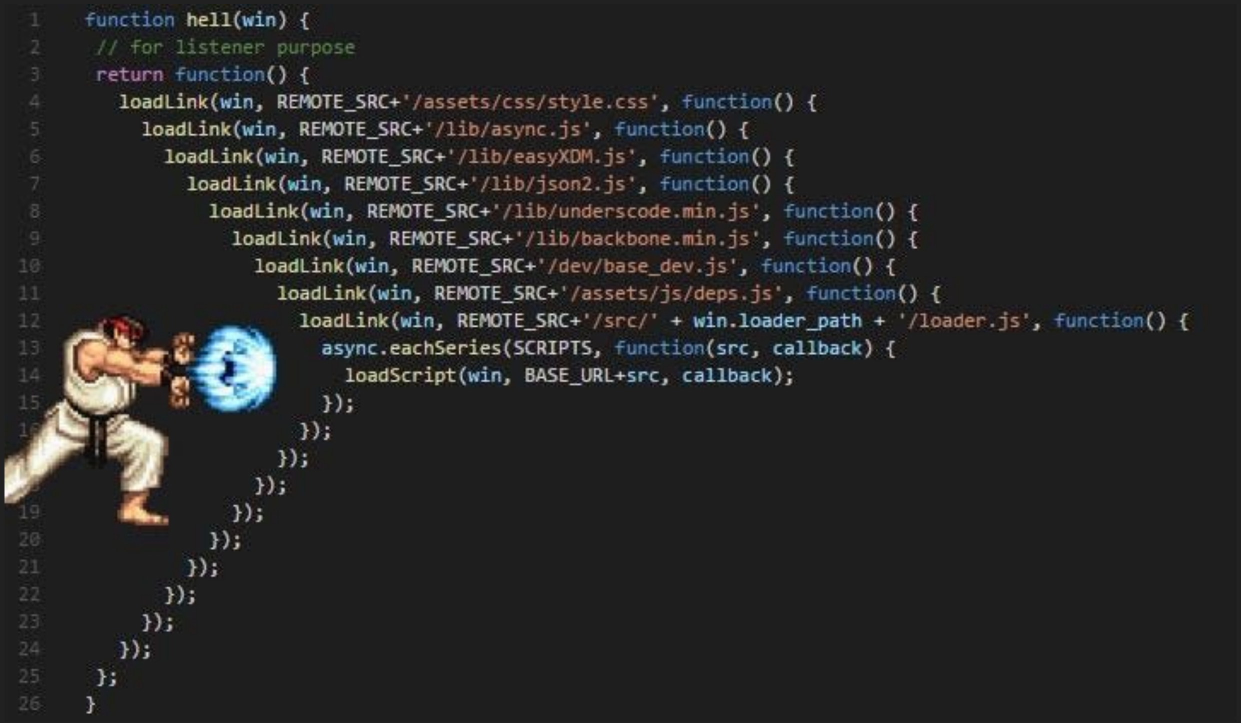
## Pros

- Non-blocking Behavior
- Handling Asynchronous Operations
- Event Handling

## Cons

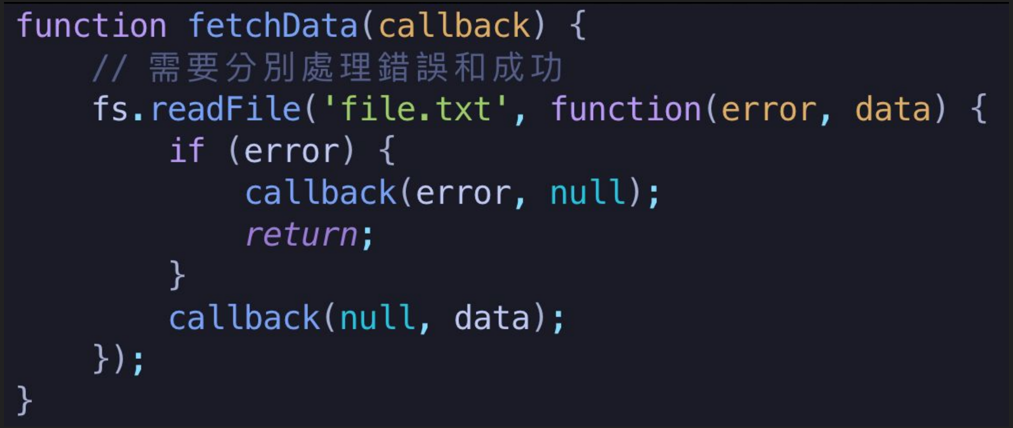
- CallBack Hell
- Error Handling Complexity
- Inconsistent Execution Order

# Callback Hells



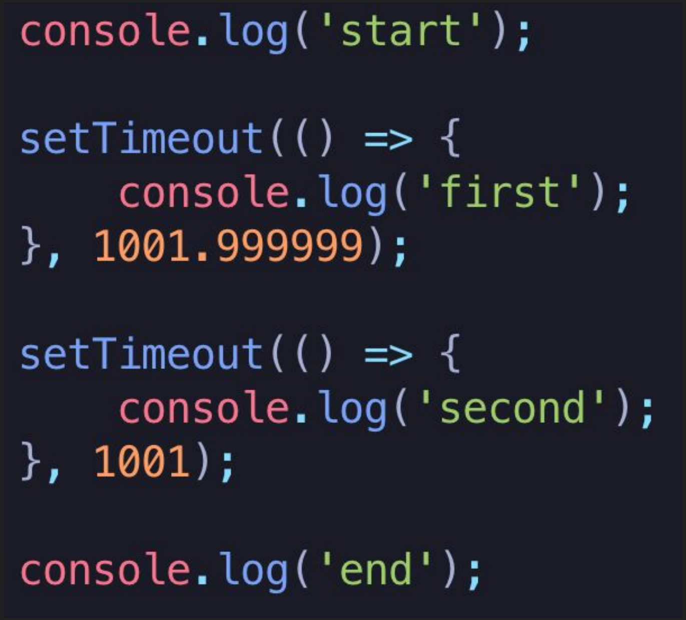
```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

# Error Handling Complexity



```javascript
function fetchData(callback) {
    // 需要分別處理錯誤和成功
    fs.readFile('file.txt', function(error, data) {
        if (error) {
            callback(error, null);
            return;
        }
        callback(null, data);
    });
}
```

# Inconsistent Execution Order



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

setTimeout(() => {
    console.log('first');
}, 1001.999999);

setTimeout(() => {
    console.log('second');
}, 1001);

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

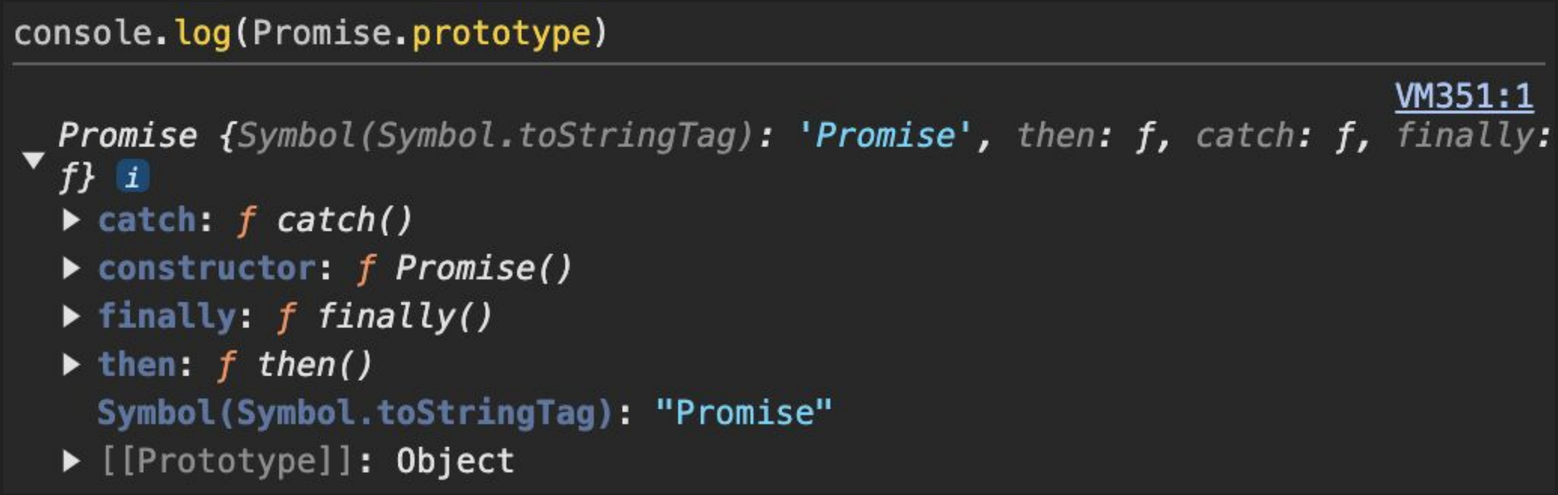
How can we address the drawbacks of using callback functions?

# Promise

# What is Promise?

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

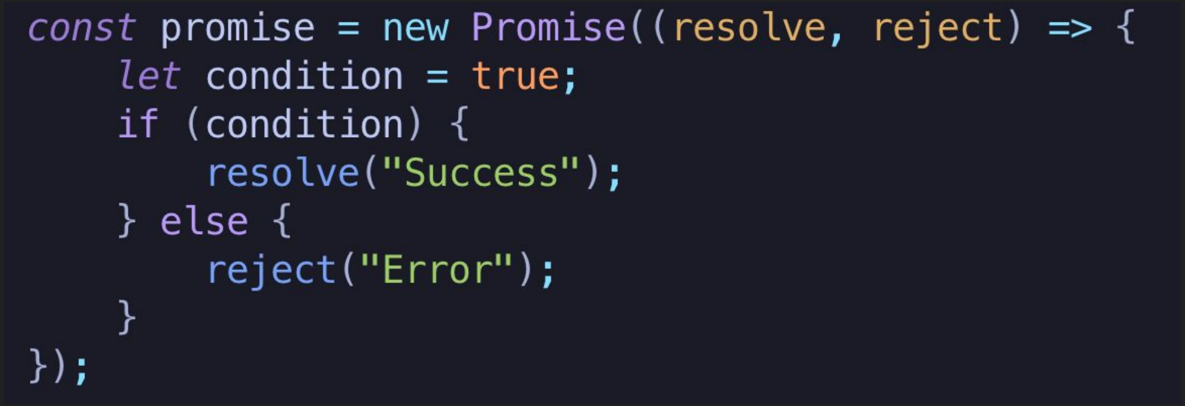
# Let’s see Promise Prototype



```
console.log(Promise.prototype)
```

```
                                                                                  VM351:1
▼ Promise {Symbol(Symbol.toStringTag): 'Promise', then: ƒ, catch: ƒ, finally:
  ƒ} i
  ▶ catch: ƒ catch()
  ▶ constructor: ƒ Promise()
  ▶ finally: ƒ finally()
  ▶ then: ƒ then()
    Symbol(Symbol.toStringTag): "Promise"
  ▶ [[Prototype]]: Object
```

# Sample



```javascript
const promise = new Promise((resolve, reject) => {
    let condition = true;
    if (condition) {
        resolve("Success");
    } else {
        reject("Error");
    }
});
```

# How to use Promise?

**new Promise**: We create a new Promise instance with the new keyword.

**Executor Function**: The argument for new Promise() is a function, which executes as soon as the Promise is created. It receives two parameters:

- **resolve**: Called when the asynchronous operation is successful, passing the result as a parameter.
- **reject**: Called if the asynchronous operation fails, passing the error reason as a parameter.

A Promise is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation was completed successfully.
- *rejected*: meaning that the operation failed.

Once resolve or reject is called, the Promise changes from Pending to either Fulfilled or Rejected, and this state cannot be changed afterward.

# Handling Promise Result

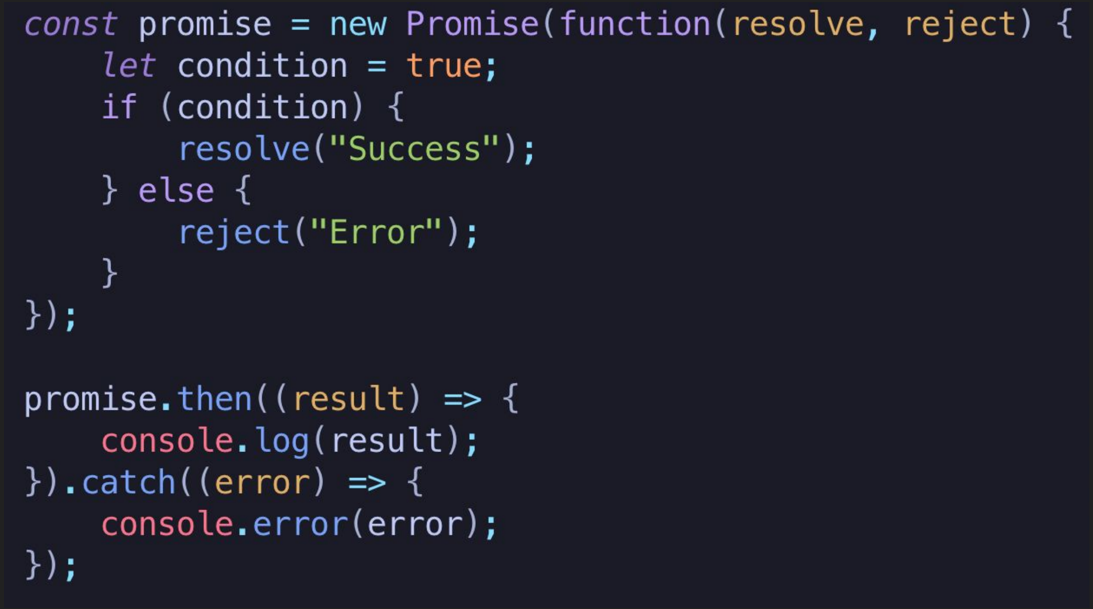
.then():

Used to handle the result when the Promise is fulfilled. It takes a callback function as an argument.

.catch():

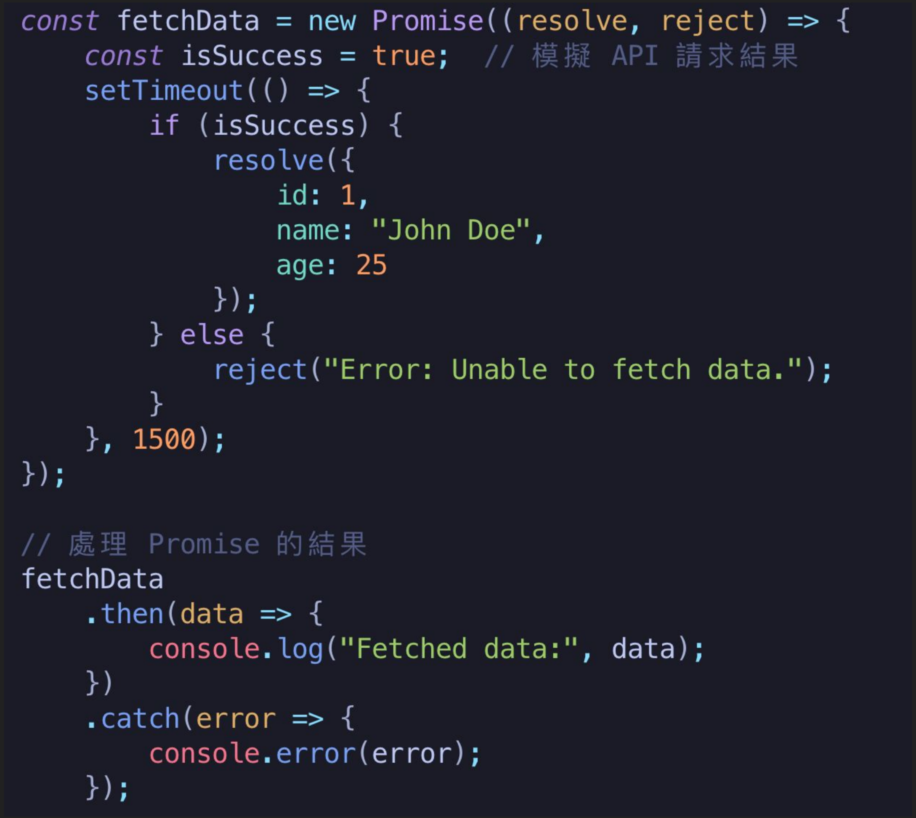
Used to handle cases where the Promise is rejected.

# Sample-1



```javascript
const promise = new Promise(function(resolve, reject) {
    let condition = true;
    if (condition) {
        resolve("Success");
    } else {
        reject("Error");
    }
});

promise.then((result) => {
    console.log(result);
}).catch((error) => {
    console.error(error);
});
```

# Sample-2



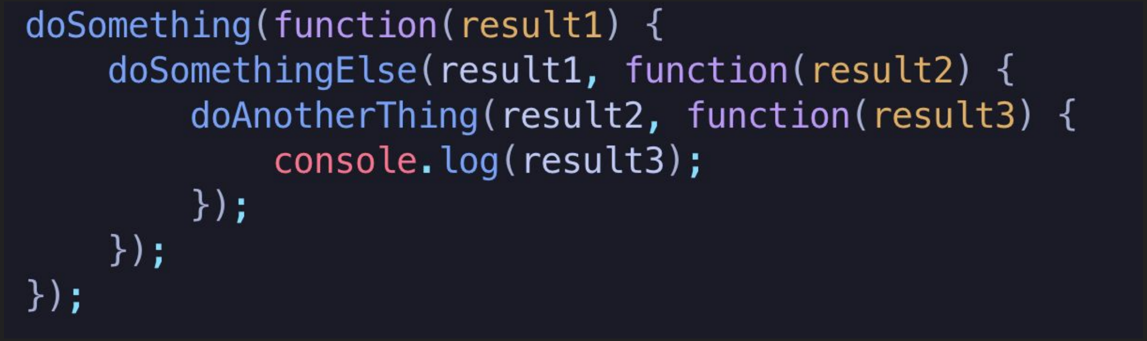
```javascript
const fetchData = new Promise((resolve, reject) => {
    const isSuccess = true;  // 模擬 API 請求結果
    setTimeout(() => {
        if (isSuccess) {
            resolve({
                id: 1,
                name: "John Doe",
                age: 25
            });
        } else {
            reject("Error: Unable to fetch data.");
        }
    }, 1500);
});

// 處理 Promise 的結果
fetchData
    .then(data => {
        console.log("Fetched data:", data);
    })
    .catch(error => {
        console.error(error);
    });
```

## Pros

- Better Readability (fix callback hell)
- Error Handling
- Flow Control

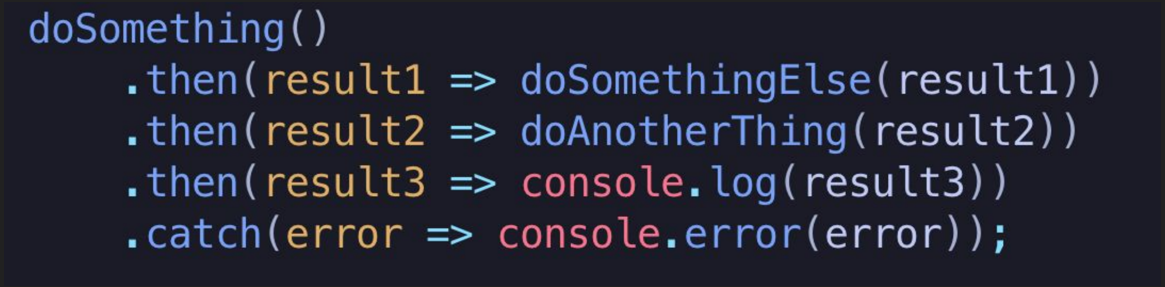
# Callbacks hell

- Using callback function



```
doSomething(function(result1) {
    doSomethingElse(result1, function(result2) {
        doAnotherThing(result2, function(result3) {
            console.log(result3);
        });
    });
});
```

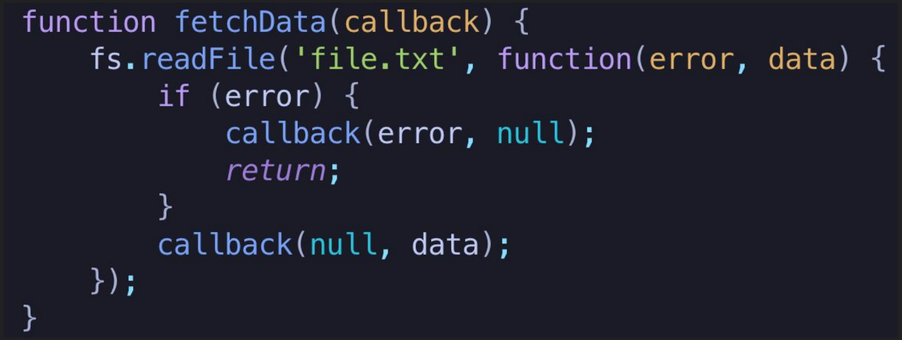
- Using Promise



```
doSomething()
    .then(result1 => doSomethingElse(result1))
    .then(result2 => doAnotherThing(result2))
    .then(result3 => console.log(result3))
    .catch(error => console.error(error));
```

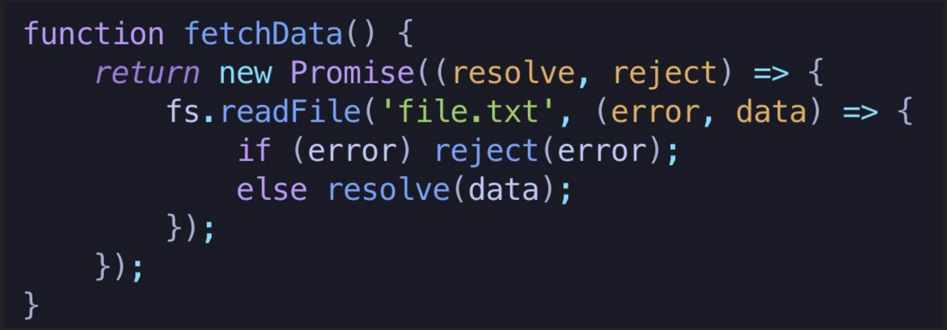
# Error Handling Complexity

- Using callback function



```
function fetchData(callback) {
    fs.readFile('file.txt', function(error, data) {
        if (error) {
            callback(error, null);
            return;
        }
        callback(null, data);
    });
}
```

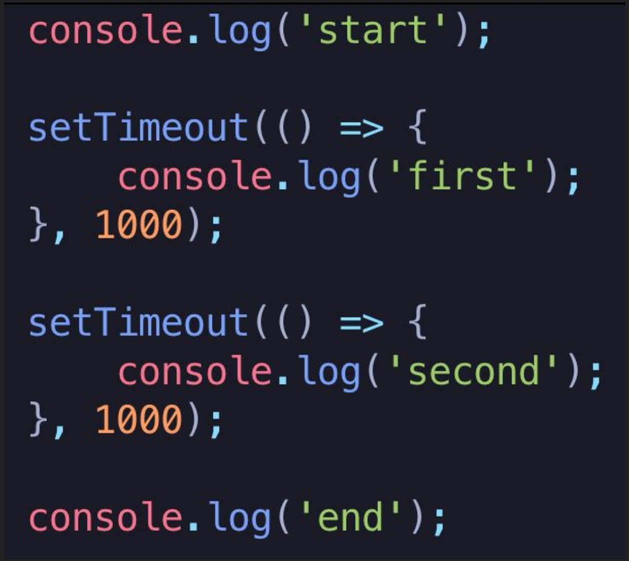
- Using Promise



```
function fetchData() {
    return new Promise((resolve, reject) => {
        fs.readFile('file.txt', (error, data) => {
            if (error) reject(error);
            else resolve(data);
        });
    });
}
```

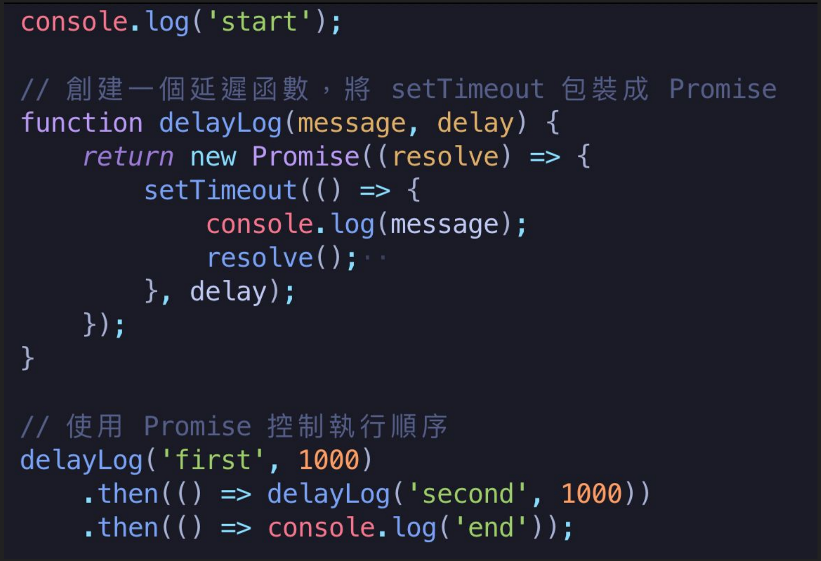
# Flow Control

- Using callback function



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

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

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

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

- Using Promise



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

// 創建一個延遲函數，將 setTimeout 包裝成 Promise
function delayLog(message, delay) {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log(message);
            resolve();
        }, delay);
    });
}

// 使用 Promise 控制執行順序
delayLog('first', 1000)
    .then(() => delayLog('second', 1000))
    .then(() => console.log('end'));
```

## Cons

- Poor Readibility
- Error Handling

# Async / Await

# What is async function?

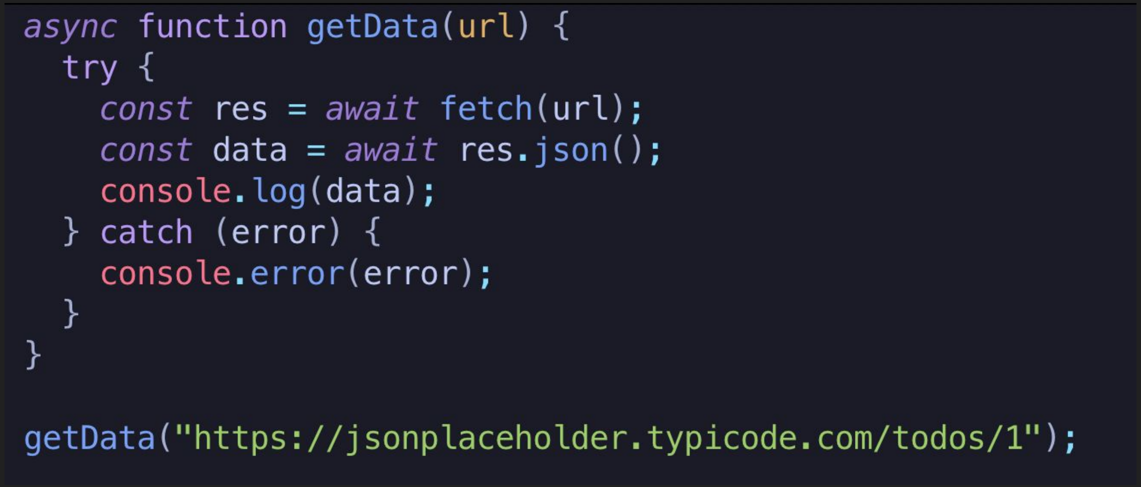
async/await is syntactic sugar for Promises, primarily improving the readability of Promise chaining syntax

# What is async function?

- **async**: Declaring a function as async allows it to return a Promise, enabling the use of await inside the function.

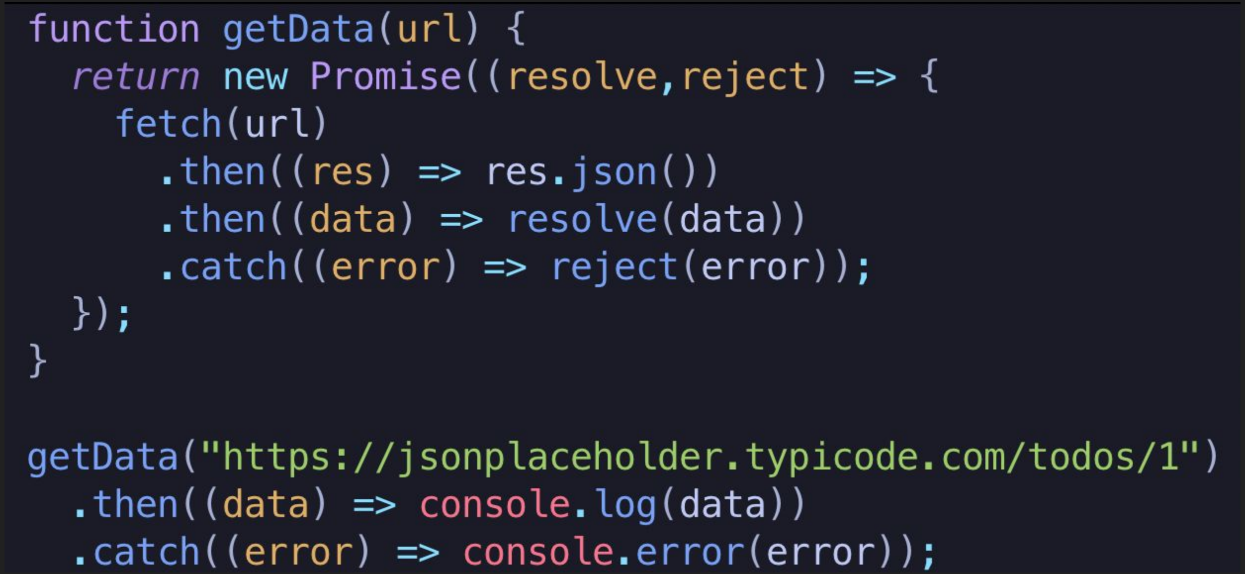
- **await**: Pauses the execution of an async function until the Promise is resolved, giving the appearance of synchronous behavior.

# Sample



```javascript
async function getData(url) {
  try {
    const res = await fetch(url);
    const data = await res.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}

getData("https://jsonplaceholder.typicode.com/todos/1");
```

# Promise v.s. Async / Await



```
function getData(url) {
  return new Promise((resolve,reject) => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => resolve(data))
      .catch((error) => reject(error));
  });
}

getData("https://jsonplaceholder.typicode.com/todos/1")
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

# Summary

**Async/await** offers a more concise and intuitive syntax, making asynchronous code easier to read and maintain.

**Error Handling**: In async/await, you can directly use try...catch blocks to capture errors. With Promises, error handling is done using the .catch method.

**Code Flow**: async/await makes asynchronous code appear more like synchronous code, making it easier to read and understand.

When await is used in an async function, JavaScript waits for the Promise to resolve before proceeding to the next line of code. This makes asynchronous code look synchronous and easier to understand.