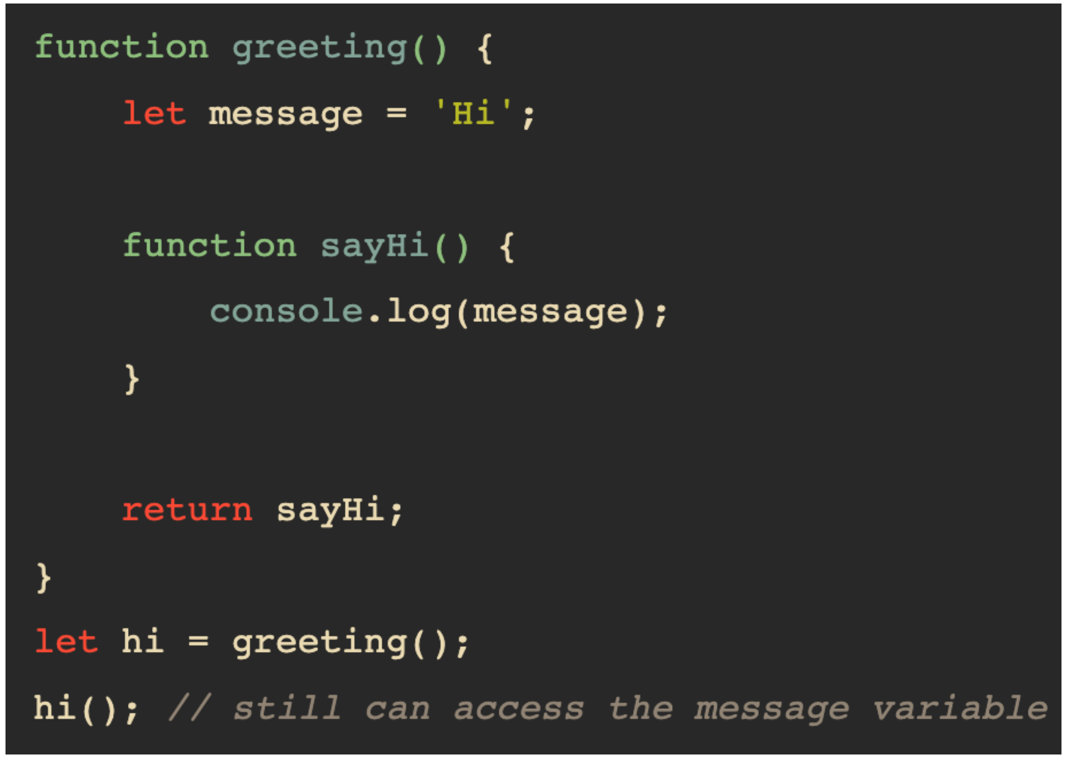
JavaScript Fundamentals

Instructor: Bui Binh Giang
buibinhgiang@vanlanguni.vn

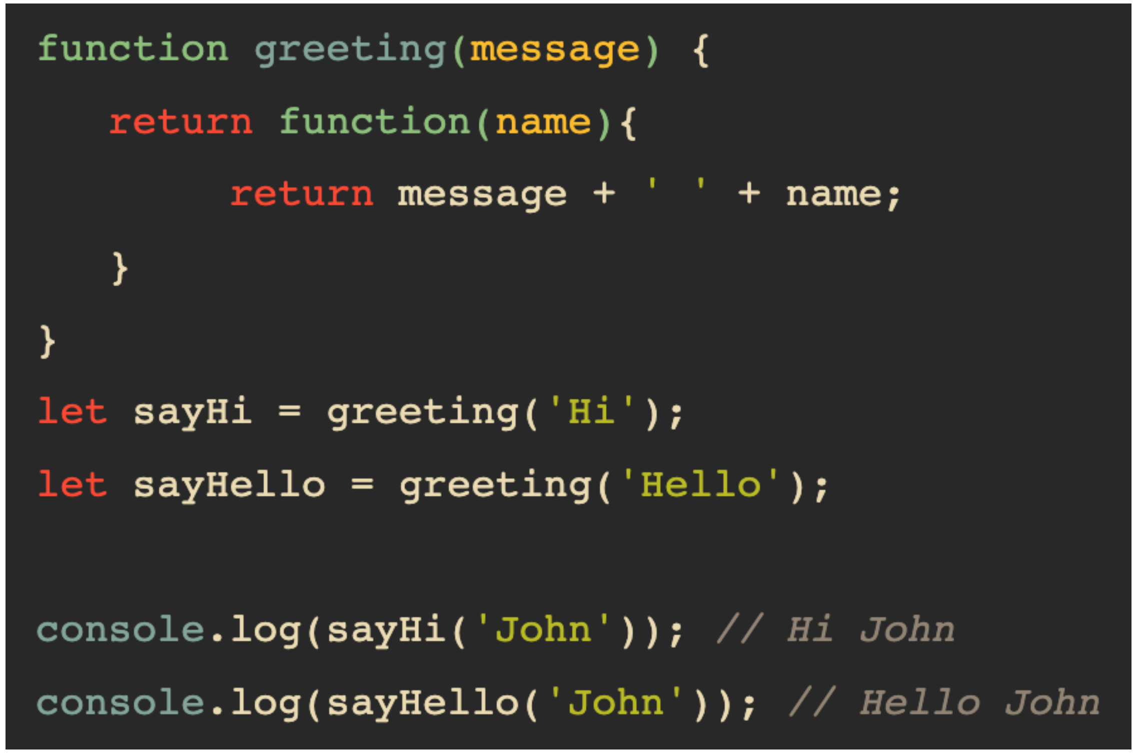
ADVANCED FUNCTIONS

JavaScript Closures

A **closure** is a **function** that preserves the **outer scope** in its **inner scope**.



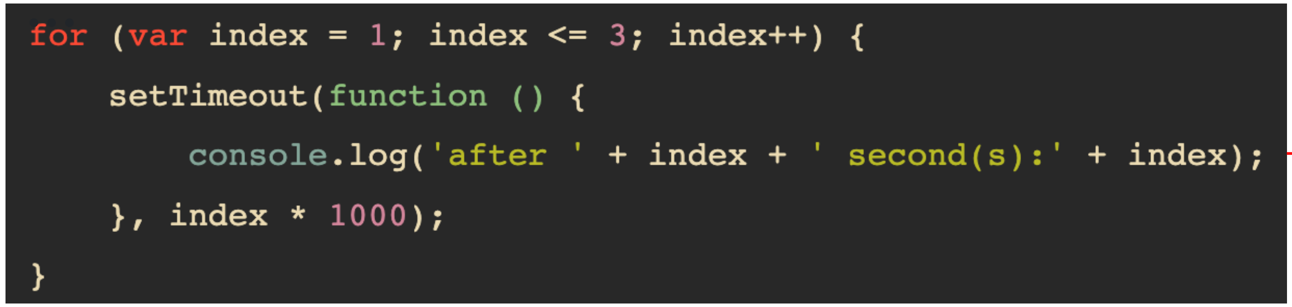
```javascript
function greeting() {
    let message = 'Hi';

    function sayHi() {
        console.log(message);
    }

    return sayHi;
}
let hi = greeting();
hi(); // still can access the message variable
```



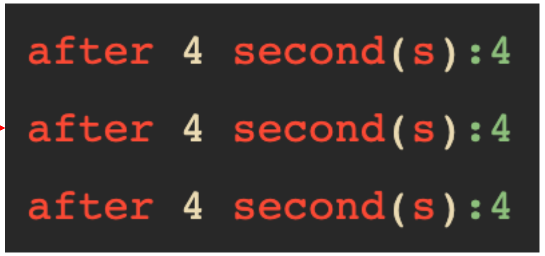
```javascript
function greeting(message) {
   return function(name){
        return message + ' ' + name;
   }
}
let sayHi = greeting('Hi');
let sayHello = greeting('Hello');

console.log(sayHi('John')); // Hi John
console.log(sayHello('John')); // Hello John
```

JavaScript closures in a loop

What we wanted to do in the **loop** is to copy the value of **i** in each iteration at the time of iteration to display messages: **"after 1 second(s):1", "after 2 second(s):2",**



```
for (var index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```



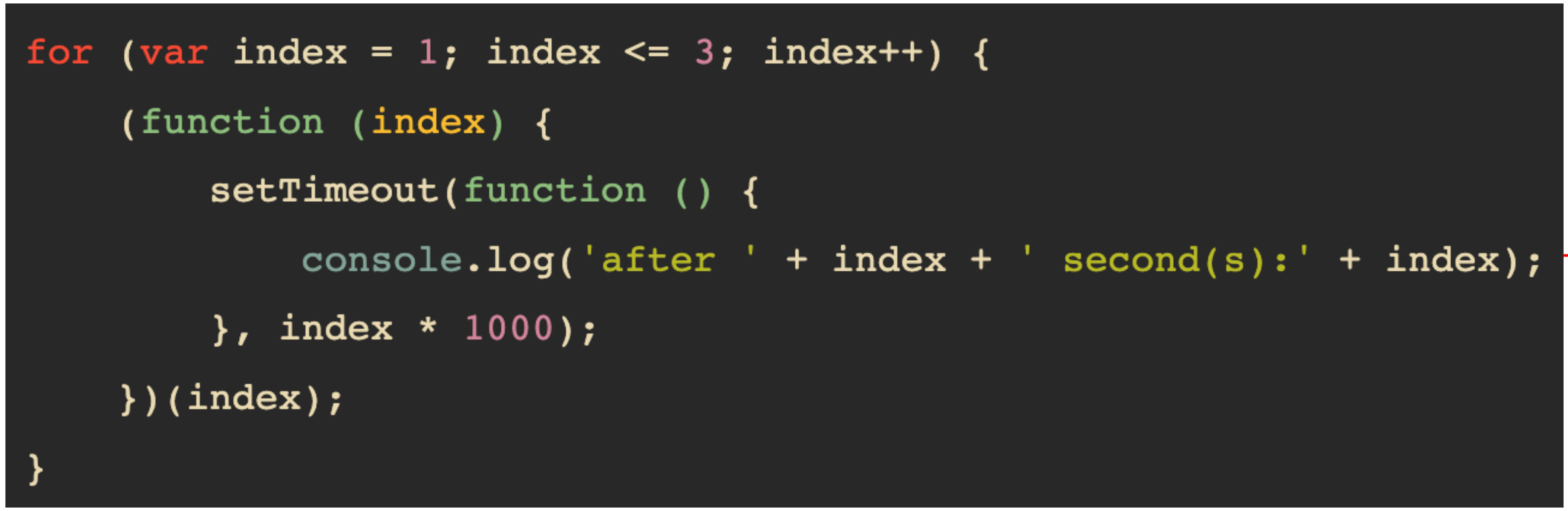
```
after 4 second(s):4
after 4 second(s):4
after 4 second(s):4
```

The reason you see the same message after 4 seconds is that the callback passed to the **setTimeout**() a **closure**. All three **closures** created by the **for-loop** share the same **global scope** access the same value of **i**

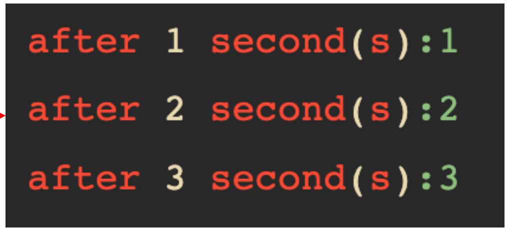
2 popular solutions: **IIFE** & **let** keyword

JavaScript closures in a loop

Using the IIFE solution: **IIFE** creates a **new scope** by declaring a **function** and immediately execute it



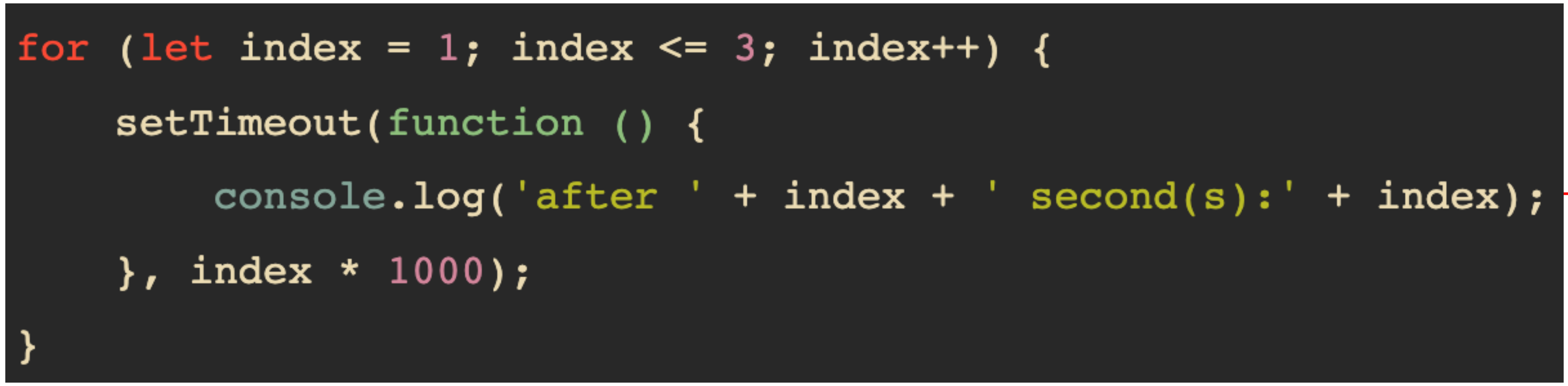
```
for (var index = 1; index <= 3; index++) {
    (function (index) {
        setTimeout(function () {
            console.log('after ' + index + ' second(s):' + index);
        }, index * 1000);
    })(index);
}
```



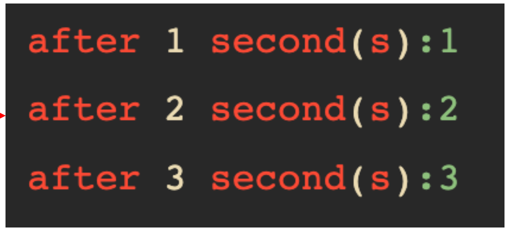
```
after 1 second(s):1
after 2 second(s):2
after 3 second(s):3
```

JavaScript closures in a loop

Using let keyword in ES6: If you use the **let** keyword in the **for-loop**, it will create a **new scope** in each iteration → In other words, you will have a new index variable in each iteration



```
for (let index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```



```
after 1 second(s):1
after 2 second(s):2
after 3 second(s):3
```

JavaScript Arrow functions

ES6 **arrow functions** provide you with an alternative way to write a shorter syntax compared to the function expression.

```
let show = function () {
    console.log('Anonymous function');
};
```

```
let show = () => console.log('Anonymous function');
```

```
let add = function (x, y) {
        return x + y;
};
```

```
let add = (x, y) => x + y;
```

JavaScript Arrow function vs. regular function

An **arrow function** doesn’t have its binding to **this** or **super**, it inherits **this** from the parent scope

An **arrow function** doesn’t have **arguments** object, **new.target** keyword, and **prototype** property.

An **arrow function** cannot be used as a **function constructor**. If you use the **new** keyword to create a **new object** from an **arrow function**, you will get an error.

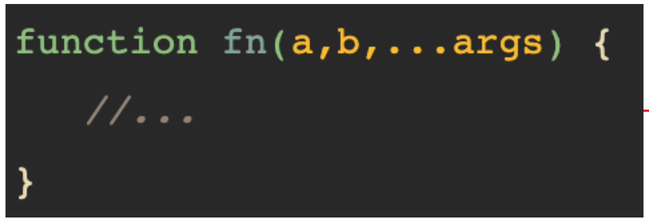
Summary: An **arrow function** doesn’t have its own **this** value and the **arguments** object. Therefore, you should not use it as an **event handler**, a **method of an object**, a **prototype method**, or when you have a function that uses the **arguments** object.

JavaScript Rest Parameters

ES6 provides a new kind of parameter so-called **rest parameter** that has a prefix of three dots **(...)**.

A **rest parameter** allows you to represent **an indefinite number of arguments as an array**

Notice: the **rest parameters** must appear **at the end** of the argument list



```
function fn(a,b,...args) {
   //...
}
```

fn(1, 2, 3, "A", "B", "C"); → a = 1; b = 2; args = [3, “A”, “B”, “C”]

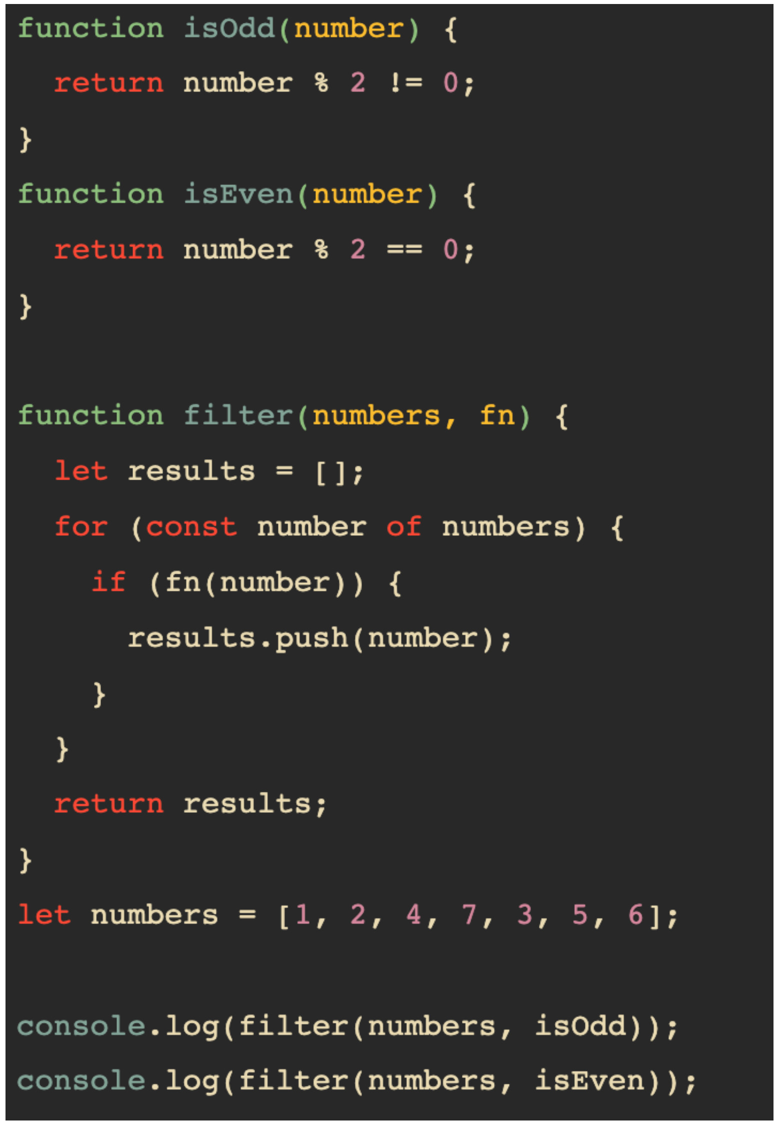
Exercise - 401

Requirement:

- Create a function that accepts indefinite number of arguments type Number.
- Print all odd and even numbers from input arguments.

JavaScript Callbacks

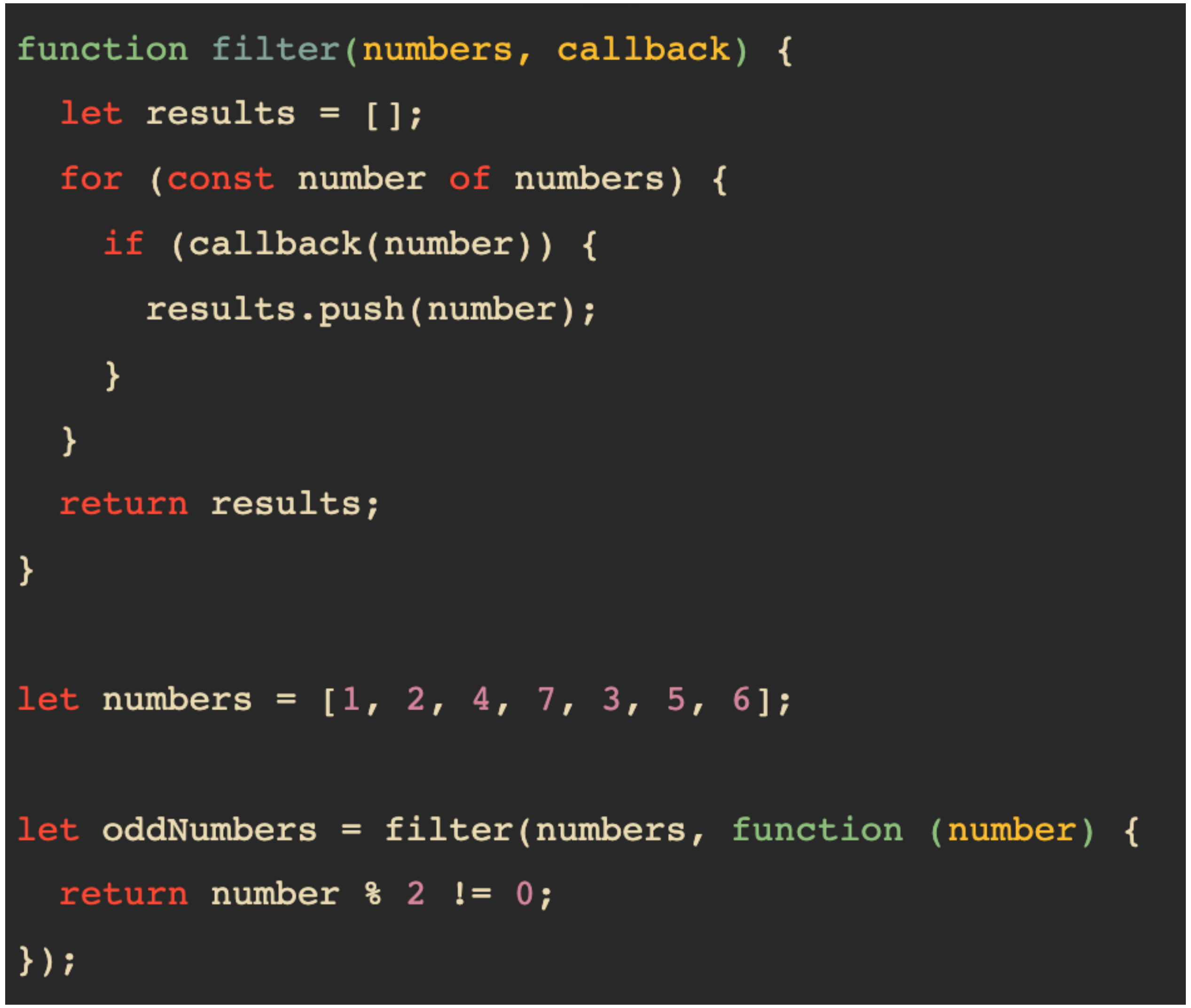
Callback is a function that you pass into another function as an argument for **executing later**



```javascript
function isOdd(number) {
  return number % 2 != 0;
}
function isEven(number) {
  return number % 2 == 0;
}

function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));
console.log(filter(numbers, isEven));
```

JavaScript Callbacks

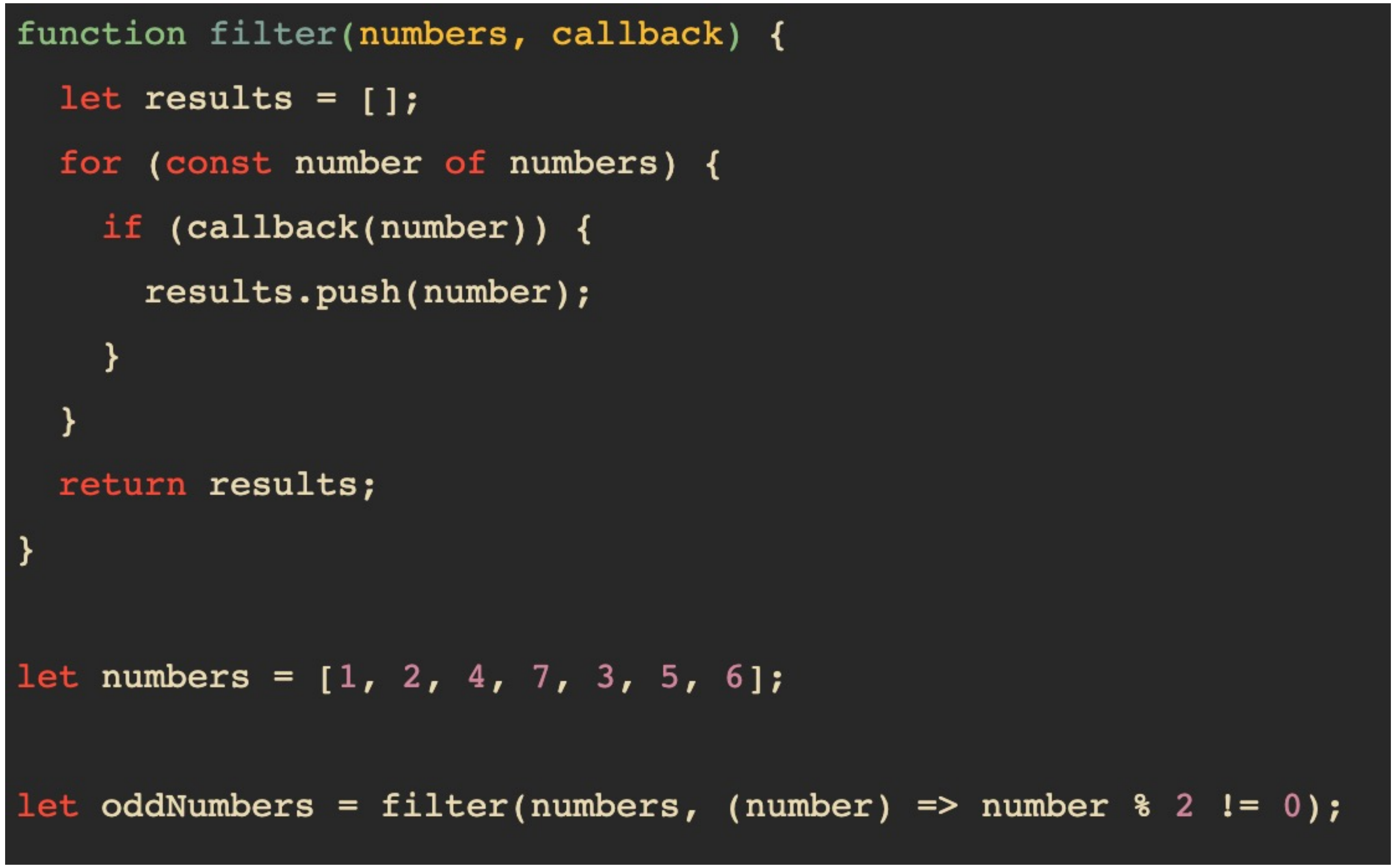
A **callback** can be an **anonymous function**, which is a function without a name



```javascript
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {
  return number % 2 != 0;
});
```

JavaScript Callbacks

A **callback** can be an **arrow function**

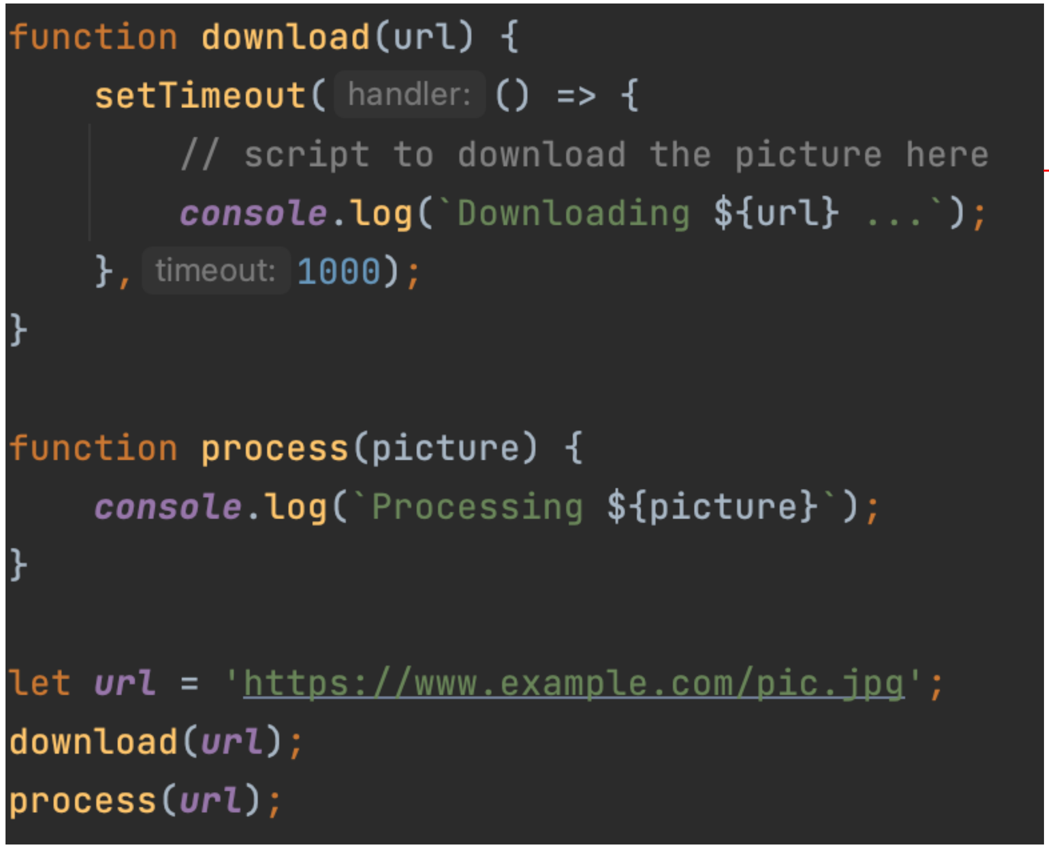


```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, (number) => number % 2 != 0);
```

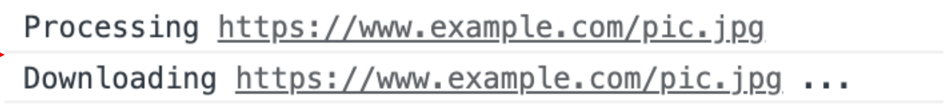
Synchronous Callbacks - Asynchronous Callbacks

- A **synchronous callback** is executed during the execution of the high-order function that uses the **callback**
- An **asynchronous callback** is executed after the execution of the high-order function that uses the **callback**. **Asynchronicity** means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

JavaScript Asynchronous Callbacks



```
function download(url) {
    setTimeout( handler: () => {
        // script to download the picture here
        console.log(`Downloading ${url} ...`);
    }, timeout: 1000);
}

function process(picture) {
    console.log(`Processing ${picture}`);
}

let url = 'https://www.example.com/pic.jpg';
download(url);
process(url);
```

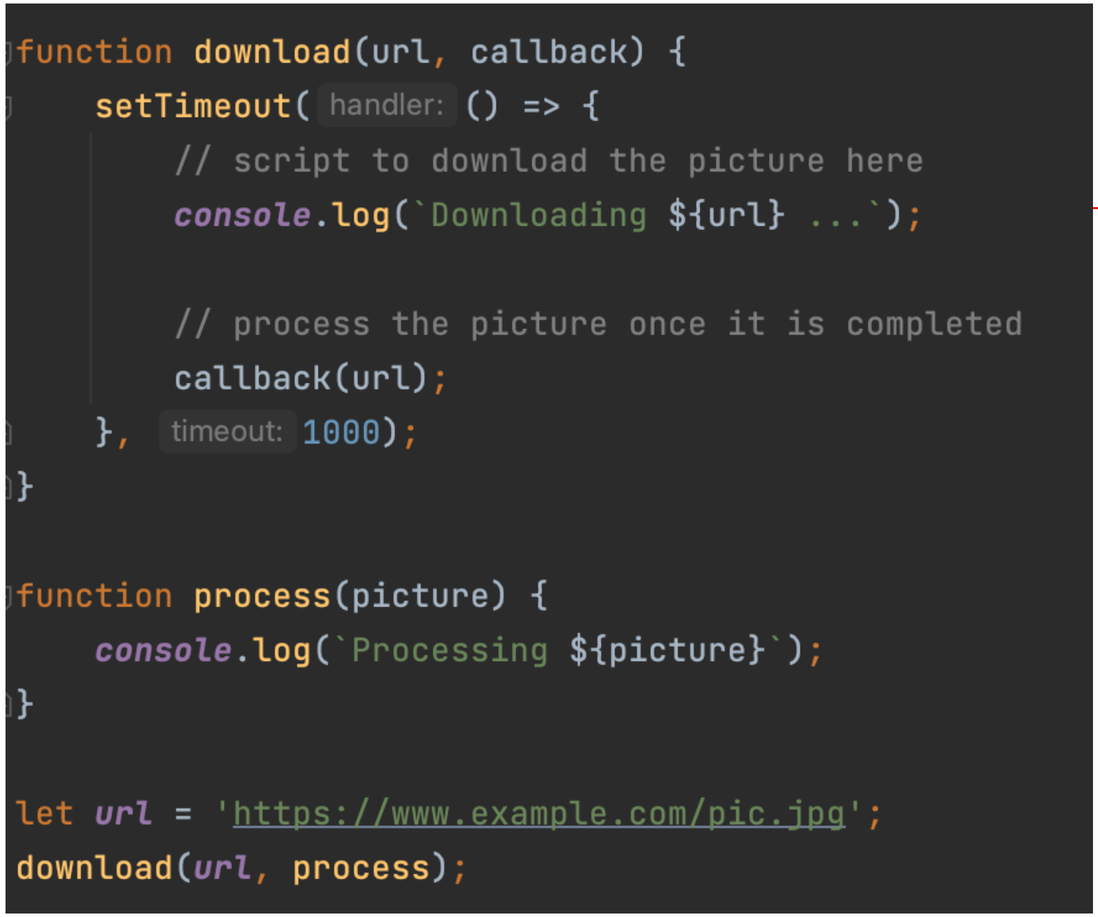


```
Processing https://www.example.com/pic.jpg
Downloading https://www.example.com/pic.jpg ...
```

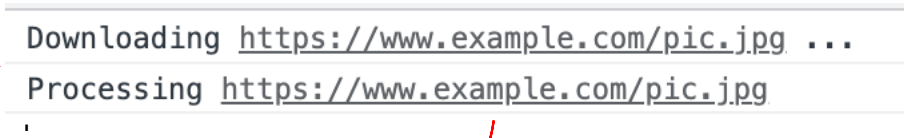
Incorrect sequence!!! The correct sequence should be:

- Download the picture and wait for the download completes.
- Process the picture.

JavaScript Asynchronous Callbacks



```
function download(url, callback) {
    setTimeout( handler: () => {
        // script to download the picture here
        console.log(`Downloading ${url} ...`);

        // process the picture once it is completed
        callback(url);
    }, timeout: 1000);
}

function process(picture) {
    console.log(`Processing ${picture}`);
}

let url = 'https://www.example.com/pic.jpg';
download(url, process);
```

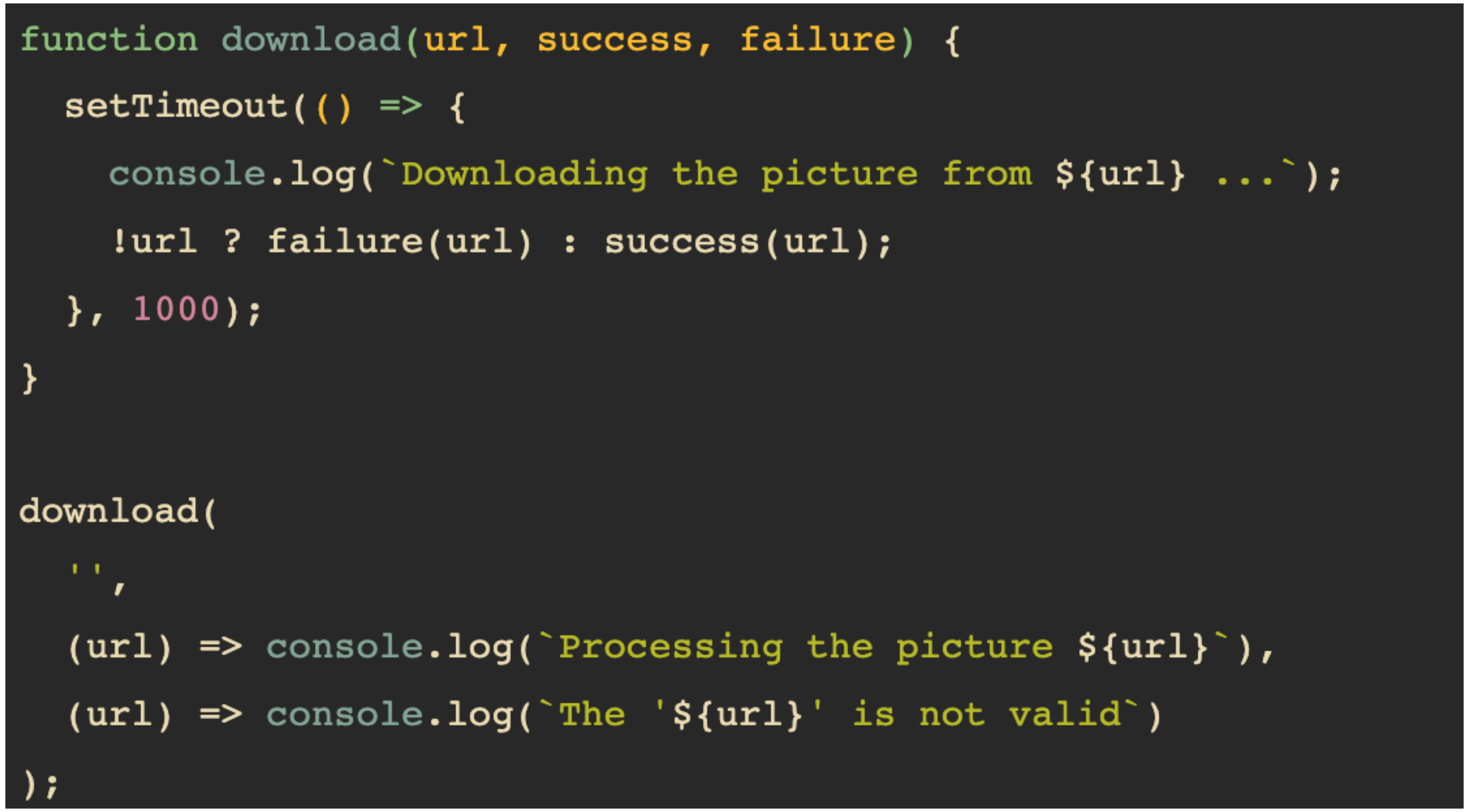


```
Downloading https://www.example.com/pic.jpg ...
Processing https://www.example.com/pic.jpg
```

It works as expected!!! → the **process**() is a callback passed into an asynchronous function

JavaScript Callbacks

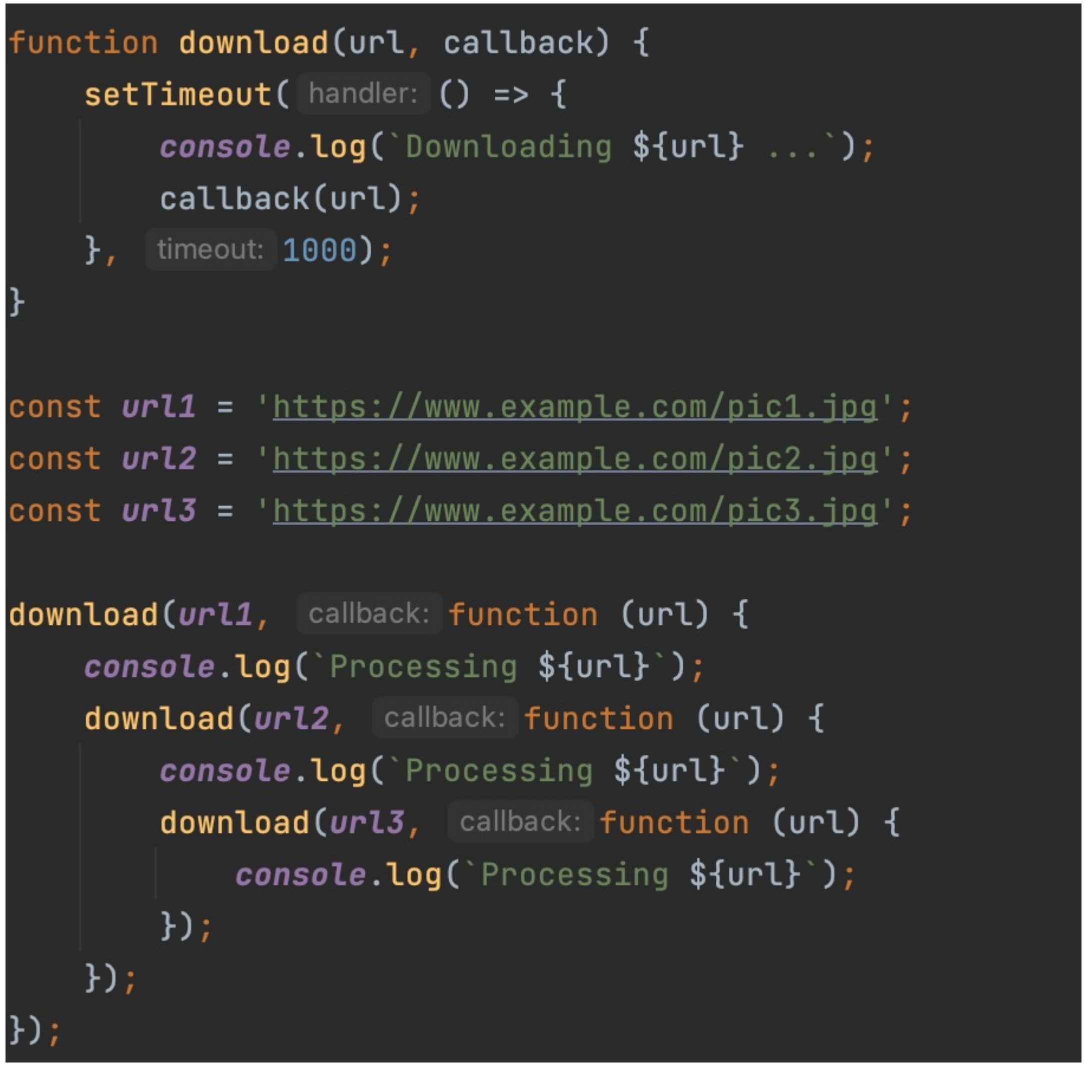
Handling errors



```
function download(url, success, failure) {
  setTimeout(() => {
    console.log(`Downloading the picture from ${url} ...`);
    !url ? failure(url) : success(url);
  }, 1000);
}

download(
  '',
  (url) => console.log(`Processing the picture ${url}`),
  (url) => console.log(`The '${url}' is not valid`)
);
```

JavaScript Callbacks

Nesting callbacks



```
function download(url, callback) {
    setTimeout( handler: () => {
        console.log(`Downloading ${url} ...`);
        callback(url);
    }, timeout: 1000);
}

const url1 = 'https://www.example.com/pic1.jpg';
const url2 = 'https://www.example.com/pic2.jpg';
const url3 = 'https://www.example.com/pic3.jpg';

download(url1, callback: function (url) {
    console.log(`Processing ${url}`);
    download(url2, callback: function (url) {
        console.log(`Processing ${url}`);
        download(url3, callback: function (url) {
            console.log(`Processing ${url}`);
        });
    });
});
```