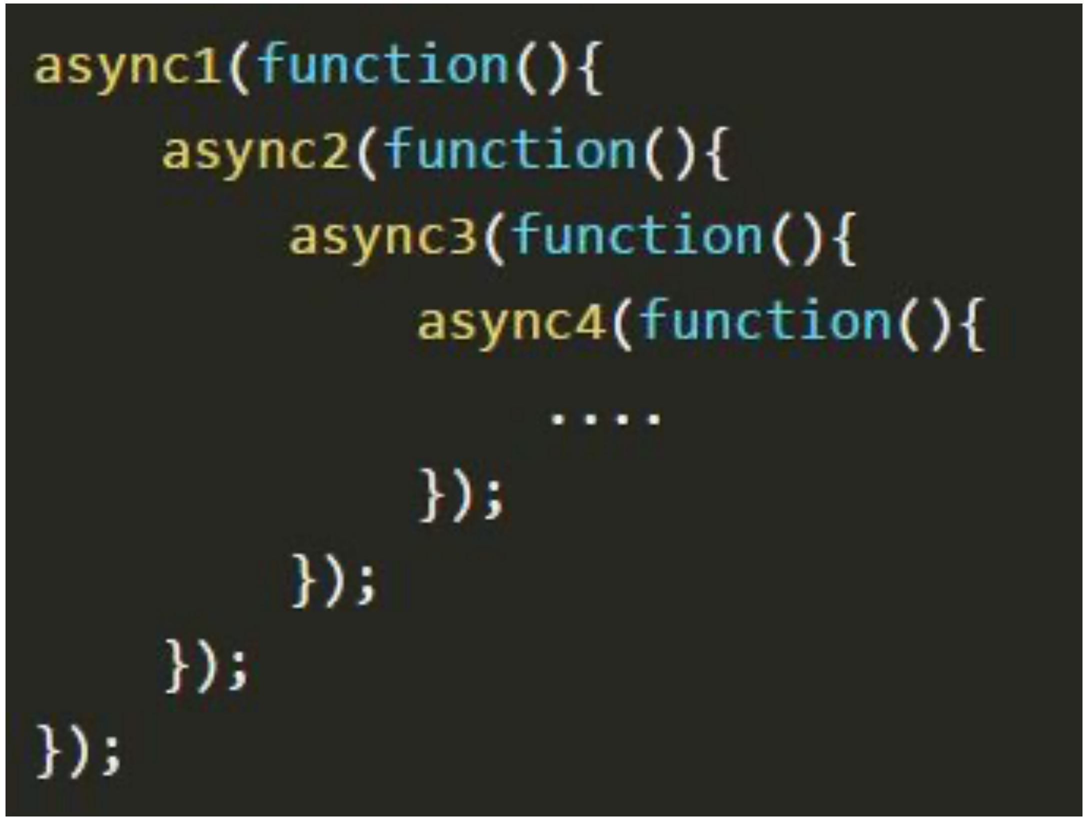
Promise, Async & Await

The Boomerang Effect (Callback Hell)

As you may have noticed, asynchronous programming relies on callback functions that are usually passed as arguments.
This can turn your code into "**callback spaghetti**", making it visually hard to track which context you are in. This style also makes debugging your application difficult, reducing even more the maintainability of your code.



```
async1(function(){
    async2(function(){
        async3(function(){
            async4(function(){
                ....
            });
        });
    });
});
```

Asynchronous Code

To write a nice asynchronous code away from the callback hell we can use one of these code structures:

- Promises (ES6)
- Function Generators (ES6)
- Async & Await (ES7, Node 8)
- util.promisify (Node 8)
- Observables (ES7)

The Promise Object

The **`Promise`** object is used for asynchronous computations. A **`Promise`** represents a value which may be available now, or in the future, or never.

```
new Promise( function(resolve, reject) { ... } );
```

A Promise has one of these states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: meaning that the operation completed successfully.
- **rejected**: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error).

Creating a promise

```
var giveMePizza = function(){
        return new Promise(function(resolve,reject){
                if(everythingWorks){
                        resolve("This is true"); // then() will be called
                } else {
                        reject("This is false"); // catch() will be called
                }
        })
}
giveMePizza()
        .then(data => console.log(data))
        .catch(err => console.error(err));
console.log('I will run immediately after calling giveMePizza() and before any
result arrives');
```

The callback from the `Promise` constructor gives us two parameters, `resolve` and `reject` functions, that will affect the state of the `Promise` object. If everything works, call `resolve`, otherwise call `reject`. Note that you can pass in values to `resolve` and `reject` which will be further passed on to the respective handlers, `then` and `catch`.

How Promises can make our code easy to read

The **`Promise`** object has two methods, **`then`** and **`catch`**. The methods will later be called depending on the state (fulfilled or rejected) of the Promise Object.

```
const postsPromise = fetch('http://mywebsite.com/API'); // return Promise

postsPromise.then(data => data.json()) // After data is being received
            .then(data => { console.log(data) })
            .catch((err) => { console.error(err); }) // in case rejected
```

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.

How Do Promises work?

The biggest misconception about Promises in JavaScript is that they are asynchronous, but not everything of Promises is asynchronous.

Only the parts of **`resolve`** and **`reject`** are going to be asynchronous.

```
const promise = new Promise((resolve, reject) => {
   console.log(`Promise starts`)
   resolve(`Promise result`)
   console.log(`Promise ends`)
})

console.log(`Code starts`)
promise.then(console.log)
console.log(`Code ends`)
```

Example

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => { resolve('Promise results')}, 1000); // resolve after 1 second
});

console.log('Code starts');

promise.then(console.log)

console.log('I love JS');
```

What happens when we change the timer to 0

Async Await (ES7)

Take advantage of the synchronous-looking code style.

`await` may only be used in functions marked with the `async` keyword. It works similarly to generators, suspending execution in your context until the promise settles. If the awaited expression isn't a promise, its casted into a promise.

An async Function always returns a Promise. That promise is rejected in the case of uncaught exceptions, and it's otherwise resolved to the return value of the **async** function.

Just like with generators, keep in mind that you should wrap await in try / catch so that you can capture and handle errors in awaited promises from within the `async` function.

Example - Promise

```
var Studied = true;
var willPassTheExam = function(){
       return new Promise(function(resolve, reject) {
       if (Studied) resolve('Pass');
       else reject(new Error('Fail'));
       })
};

var askMe = function () {
       willPassTheExam()
              .then(function(results){ console.log(results); })
              .catch(function (error) { console.log(error); });
};

askMe();
console.log('Finish')

// Finish
// Pass
```

Example – Async & Await

1 2 3

```
async function askMe() {
        try {
                console.log('Before taking the exam');
                let results = await willPassTheExam();
                console.log(results);
                console.log('After taking the exam');
        } catch (error) {
                console.log(error);
        }
}

askMe();
console.log('Finish')

// Before taking the exam
// Finish
// Pass
// After taking the exam
```