Promises

What are promises? This lesson will cover the basics to get started using them.

WE'LL COVER THE FOLLOWING

`JavaScript` work **synchronously** which means that each block of code will be executed after the previous one.

```
const data = fetch('your-api-url-goes-here');
console.log('Finished');
console.log(data);
```

In the example above, we are using `fetch` to retrieve data from a url (in this example we are only pretending to be doing so).

In case of synchronous code we would expect the subsequent line to be called only after the `fetch` has been completed. But in reality, what's going to happen is that `fetch` will be called, and straight away the subsequent two `console.log` will also be called, resulting in the last one `console.log(data)` to return `undefined`.

This happens because `fetch` performs **asynchronously**, which means that the code **won't stop** running once it hits that line but, instead, will continue executing.

What we need is a way of waiting until `fetch` returns us something before we

What we need is a way of waiting until `fetch` returns as something before we continue executing our code.

To avoid this we would use **callbacks** or **promises**.

Callback hell

You may have heard of something called **callback hell**, which is something that happens when we try to write **asynchronous** code as if it were **synchronous**, meaning that we try to chain each block of code after the other.

In simple words it would something like:

do A, If A do B, if B do C and so on and so forth.

The following is an example just to showcase the meaning of **callback hell**. Imagine each step is asynchronous so that we would need to send a request to our server for each step of preparing the pizza. Then we need to wait for the server to respond.

```
const makePizza = (ingredients, callback) => {
    mixIngredients(ingredients, function(mixedIngredients)){
      bakePizza(mixedIngredients, function(bakedPizza)){
        console.log('finished!')
      }
    }
}
```

We try to write our code in a way where executions happens visually from top to bottom. This causes excessive nesting on functions and results in what you can see above.

To improve your callbacks, you can check out http://callbackhell.com/

Promises will help us escape this “hell” and improve our code.

What is a Promise?

From MDN:

> A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

Have a quick look at the following example:

```
const data = fetch('your-api-url-goes-here');
console.log('Finished');
console.log(data);
```

Create your own promise

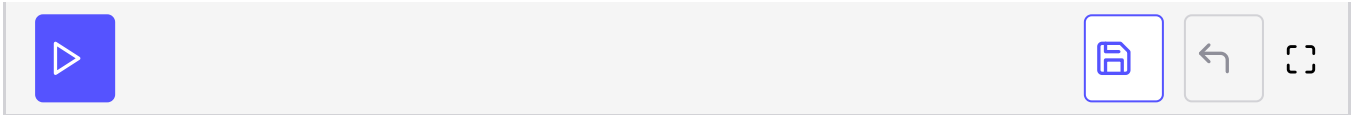
```
const myPromise = new Promise((resolve, reject) => {
  // your code goes here
});
```

This is how you create your own promise, `resolve` and `reject` will be called once the promise is finished.

We can immediately return it to see what we would get:

```
const myPromise = new Promise((resolve, reject) => {
  resolve("The value we get from the promise");
});

myPromise.then(
  data => {
    console.log(data);
  });
// The value we get from the promise
```



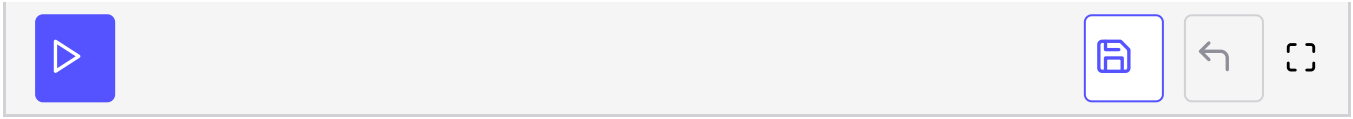
We immediately resolved our promise and can see the result in the console.

We can combine a `setTimeout()` to wait a certain amount of time before resolving.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
      resolve("The value we get from the promise");
    }, 2000);
});
```

```
myPromise.then(
  data => {

    console.log(data);
  });
// after 2 seconds ...
// The value we get from the promise
```

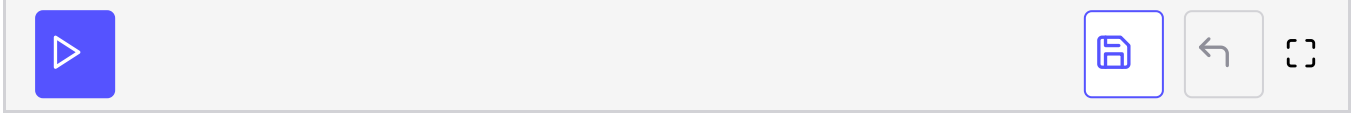


These two examples are very simple but **promises** are very useful when dealing with big requests of data.

In the example above, we kept it simple and only resolved our promise. In reality, you will also encounter errors. So let's see how to deal with them:

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
      reject(Error("this is our error"));
    }, 2000);
});

myPromise
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    console.error(err);
  })
  // Error: this is our error
  // Stack trace:
  // myPromise</<@debugger eval code:3:14
```



We use `.then()` to grab the value when the promise resolves and `.catch()` when the promise rejects.

Looking at our error log, you can see that it tells us where the error occurred. That's because we wrote `reject(Error("this is our error"));` and not simply `reject("this is our error");`.

Chaining promises

We can chain promises one after the other, using what was returned from the

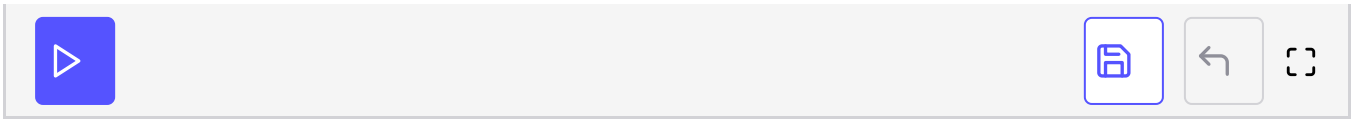
previous one as the base for the subsequent one, whether the promise was resolved or rejected.

You can chain as many promises as you want and the code will still be more readable and shorter than what we have seen above in **callback hell**.

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

myPromise
  .then(data => {
    // return something new
    return 'working...'
  })
  .then(data => {
    // log the data that we got from the previous promise
    console.log(data);

    throw 'failed!';
  })
  .catch(err => {
    // grab the error from the previous promise and log it
    console.error(err);
    // failed!
  })
```



As you can see, the first `then` passed a value down to the second one where we logged it and also threw an error that was caught in the `catch` clause.

We're not limited to chaining in case of success, we can also chain when we get a `reject`.

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

myPromise
  .then(data => {
    throw new Error("ooops");

    console.log("first value");
  })
  .catch(() => {
    console.log("catch an error");
  })
  .then(data => {
  console.log("second value");
  });
  // catch an error
  // second value
```

We did not get “first value” because we threw an error therefore we only got the first `.catch()` and the last `.then()`.

`Promise.resolve()` & `Promise.reject()`

`Promise.resolve()` and `Promise.reject()` will create promises that automatically resolve or reject.

```
//Promise.resolve()
Promise.resolve('Success').then(function(value) {
  console.log(value);
  // "Success"
}, function(value) {
  // not called
});

// Promise.reject()
Promise.reject(new Error('fail')).then(function() {
  // not called
}, function(error) {
  console.log(error);
  // Error: fail
});
```

As you can see in the first example, the Promise created in the `then` clause has two arguments, one function that get’s called when the Promise resolves, and one that gets called when the Promise rejects. Since `Promise.resolve()` immediately resolves the promise, the first functions is being called.

The opposite happens in the second example, where we use `Promise.reject()` to immediately reject the Promise, therefore the second argument of the `then` clause gets called.

`Promise.all()` & `Promise.race()`

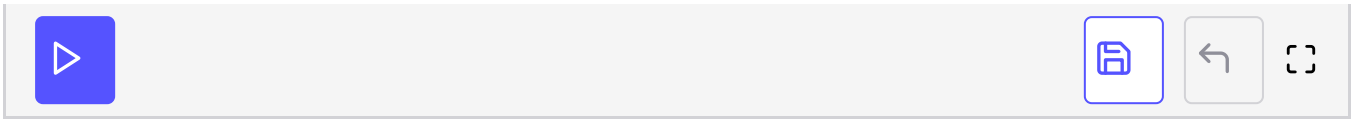
`Promise.all()` returns a single Promise that resolves when all promises have resolved.

Let's look at this example where we have two promises.

> Please notice that due to the way the Educative platform works, you will see the results show up at the same time. If you want to see the timeouts being respected, run the code in the Dev Tools.

```
const promise1 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 1000, 'second value');
});

promise1.then(data => {
  console.log(data);
});
// after 500 ms
// first value
promise2.then(data => {
  console.log(data);
});
// after 1000 ms
// second value
```

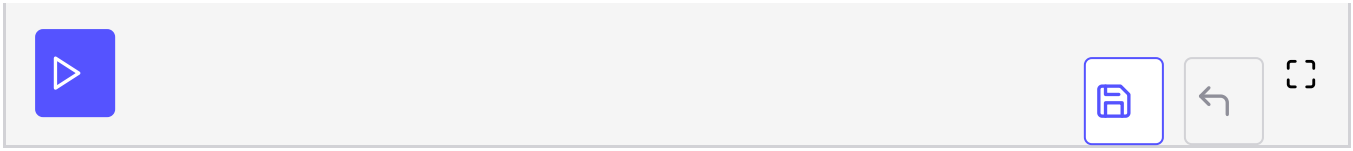


They will resolve independently from one another but look at what happens when we use `Promise.all().`

```
const promise1 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 1000, 'second value');
});

Promise
  .all([promise1, promise2])
  .then(data => {
    const[promise1data, promise2data] = data;
    console.log(promise1data, promise2data);
  });
// after 1000 ms
```

```
// first value second value
```



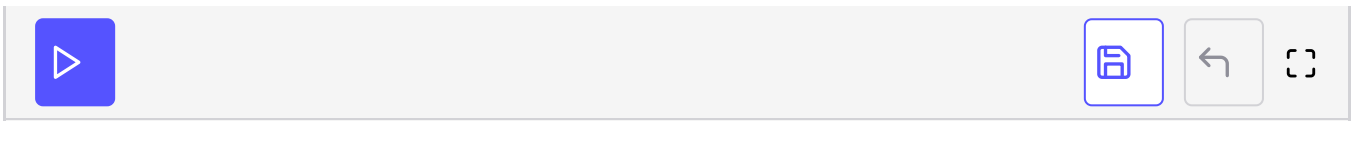
Our values returned together, after 1000ms (the timeout of the *second* promise). This means that the first one had to wait for the completion of the second one.

If we were to pass an empty iterable then it will return an already resolved promise.

If one of the promises was rejected, then all of them would asynchronously reject with the value of that rejection even if they resolved.

```
const promise1 =  new Promise((resolve,reject) => {
  resolve("my first value");
});
const promise2 =  new Promise((resolve,reject) => {
  reject(Error("oooops error"));
});

// one of the two promises will fail, but `.all` will return only a rejection.
Promise
  .all([promise1, promise2])
  .then(data => {
    const[promise1data, promise2data] = data;
    console.log(promise1data, promise2data);
  })
  .catch(err => {
    console.log(err);
  });
  // Error: oooops error
```



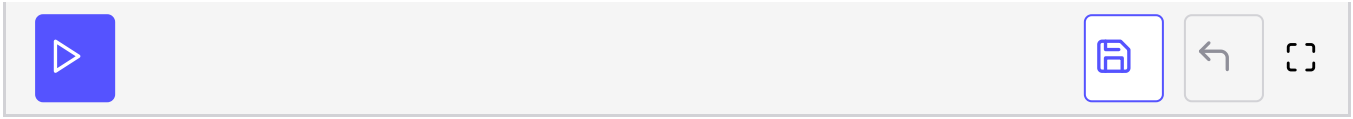
`Promise.race()` on the other hand returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects with the value from that promise.

```
const promise1 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 500, 'first value');
});
const promise2 =  new Promise((resolve,reject) => {
  setTimeout(resolve, 100, 'second value');
});

Promise.race([promise1, promise2]).then(function(value) {
```

```
  console.log(value);
  // Both resolve, but promise2 is faster

});
// expected output: "second value"
```



If we passed an empty iterable, the race would be pending forever!

Let’s see if this all sticks with you in the following quiz.