Learning Objectives

- **Define the purpose of promises**
- **Identify the number of functions needed to implement a promise**
- **Explain the steps of how a promise is either resolved or rejected**
- **Invoke a promise and the resolved and rejected callbacks**
- **Identify the syntax for the or operator in JavaScript**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Promises

Promise Intro & Purpose

Promises are one of the more difficult parts of JavaScript to understand, but you should know all the fundamentals now, and we'll take it slow to understand everything that's happening.

The purpose of promises is to provide a method of performing asynchronous code, or running code in the background. Since JavaScript doesn't have a threading model, this is accomplished with callbacks. They do this with a consistent interface that allows a promise to be *resolved* (complete successfully) or (optionally) *rejected* (fail). In the last part of this module, we're going to use promises when we fetch data from our API. The data fetching will happen in the background and our callback function will be executed when the data is received.

Before using the `Promise` class, let's talk about all the callbacks that are needed.

Promise Construction

Promises work because functions can be passed around in JavaScript. We need at least two, and sometimes three functions to implement a Promise. The first is the function that actually does the work. Rather than return a result, the worker function will call a function with the result, to "resolve" the promise. If there's a failure in the worker function, then it might also be able to "reject" the promise. This will depend on how the `Promise` class has actually been configured.

Most of the time, you won't actually be instantiating and using `Promise` classes yourself, you'll be working with `Promises` that other functions are returning. But in order to demonstrate what's happening and to give a better understanding, we'll make use of a `Promise` class.

In this example, we'll create a `lazyAdder` function. It will add two numbers together. It will not return the result, but resolve it. If there is a problem with the arguments, like they are not numbers, it will reject the promise. Note that this code is not asynchronous but does have the same interface as what would be used in an asynchronous promise.

Let's look at this function, then discuss how it works in more detail.

```
const lazyAdd = function (a, b) {
  const doAdd = (resolve, reject) => {
    if (typeof a !== "number" || typeof b !== "number") {
      reject("a and b must both be numbers")
    } else {
      const sum = a + b
      resolve(sum)
    }
  }

  return new Promise(doAdd)
}
```

The function that other code will call is `lazyAdd`, but we define another function inside it called `doAdd` which contains the actual code to do the addition. We need to do this because the function we pass to the `Promise` class must only take resolve and reject functions as parameters. We could not pass a function to `Promise` that takes the numbers to add and the resolve and reject functions.

By wrapping `doAdd()`, it has access to the parent function's variables, and so it can access the `a` and `b` parameters. `doAdd()` is redefined with access to the new `a` and `b` values whenever `lazyAdd()` is called, and so when `lazyAdd()` is eventually called with the resolve and reject callback functions, it will use the right parameters.

How is `doAdd()` called if it is defined only inside the `lazyAdd()` function? It's used to instantiate a `Promise`, which is returned from the `lazyAdd()` function. When the `Promise` is executed it then calls the `doAdd()` function.

To briefly explain the behavior of `doAdd()`: it uses the special `typeof` function, which returns a string, to check the types of `a` and `b`. If either of these is not `number` (`||` is the equivalent of `or` in Python) then the promise is rejected by calling the `reject()` function that was passed in. If the numbers can be added, their sum is passed to the `resolve()` function.

So `lazyAdd()` is the first callback, what about the other two? What we've defined so far could be considered the "producer" side of the `Promise` system. As we mentioned before, you will not be defining `Promises`, instead you'll be consuming them. Let's take a look at this "other side".

Executing Promises

We can pretend that we've started here with a blank slate: someone has provided us with a `lazyAdd()` function that we know returns a promise that can resolve or reject. We don't need to know about its implementation. How are we as the consumers to use this function?

We need to define two callback functions. We would know from documentation, or inspecting source code, that the resolve function accepts a single argument, the result of the addition. We'd also know the reject function also accepts a single argument, a string that describes why the promise was rejected. It is up to us as the consumer to decide how to deal with the results.

Think of this like a normal function provided by a third party. You know what type of variable a function returns, but it's up to you to decide how you use that variable after calling the function.

With that introduction, how do we actually get the `lazyAdd()` function to give us a result? We use the `then()` method on a `Promise`. The `then()` method has the same signature as whatever function was passed to the `Promise` constructor. In our case, this was the `doAdd()` internal function, and we know it accepts `resolve` and `reject` callback functions. Once the `then()` method is called, whether it resolves or rejects, the `Promise` is then called *settled*.

We could write callback functions like this:

```
function resolvedCallback(data) {
  console.log('Resolved with data: ' +  data)
}

function rejectedCallback(message) {
  console.log('Rejected with message: ' + message)
}
```

And use them with the promise like this:

```
const p = lazyAdd(3, 4)
// p is a Promise instance that has not yet been settled
// There will be no console output at this point.


// This next line will settle the doAdd function
p.then(resolvedCallback, rejectedCallback)
// There will be some console output now
```

More commonly, the function that returns the promise and the `then()` call will be chained in a single line, like this:

```
lazyAdd(3, 4).then(resolvedCallback, rejectedCallback)
```

You could read this line as “perform a lazy add and then call my callbacks resolvedCallback or rejectedCallback”.

To make the rejectedCallback get called, we could pass in some non-numbers:

```
lazyAdd("nan", "alsonan").then(resolvedCallback,
        rejectedCallback)
```

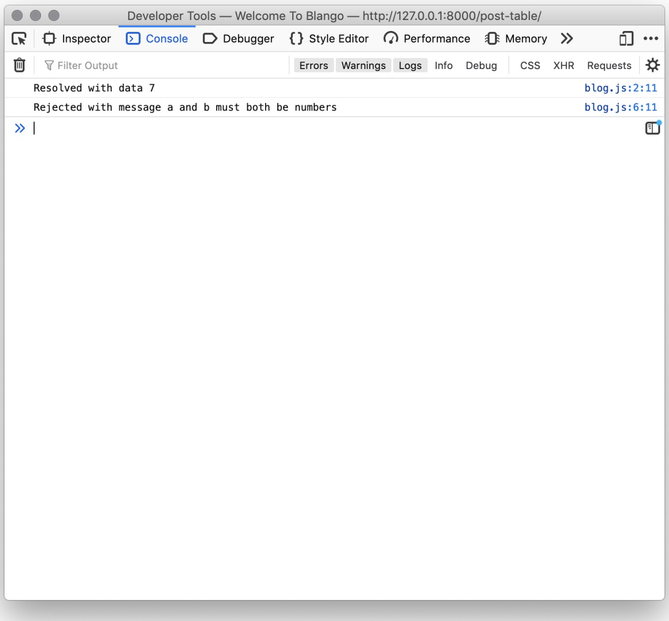
Try It Out

Now you’ll add some Promises to your Blango test script. Open `blog.js` and remove all its content, then paste in this code:

```
function resolvedCallback(data) {
  console.log('Resolved with data ' +  data)
}

function rejectedCallback(message) {
  console.log('Rejected with message ' + message)
}

const lazyAdd = function (a, b) {
  const doAdd = (resolve, reject) => {
    if (typeof a !== "number" || typeof b !== "number") {
      reject("a and b must both be numbers")
    } else {
      const sum = a + b
      resolve(sum)
    }
  }

  return new Promise(doAdd)
}

const p = lazyAdd(3, 4)
p.then(resolvedCallback, rejectedCallback)

lazyAdd("nan", "alsonan").then(resolvedCallback,
        rejectedCallback)
```

Load `/post-table/` in your browser and check the console. You should see both resolve and reject messages in the browser console.

View Blog



```
Resolved with data 7                                          blog.js:2:11
Rejected with message a and b must both be numbers            blog.js:6:11
```

Figure 1

It’s important to reiterate once again: most of the time you won’t be writing your own promises but will be using promises provided by existing functions or libraries. We’ve needed to create a `Promise` here to give you a better understanding of what’s happening in the background in those cases. You’ll have much better understanding when it comes to using promise-based code in the future now that you have the additional groundwork.

In the next section, we’re going to start using the React framework.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish JavaScript promises"
```

- Push to GitHub:

```
git push
```