Promises

A JavaScript Promise is created with the new Promise constructor function - `new Promise()` . A promise will let you start some work that will be done **asynchronously** and let you get back to your regular work. When you create the promise, you must give it the code that will be run asynchronously. You provide this code as the argument of the constructor function:

```
new Promise(function () {
    window.setTimeout(function createSundae(flavor = 'chocolate') {
        const sundae = {};
        // request ice cream
        // get cone
        // warm up ice cream scoop
        // scoop generous portion into cone!
    }, Math.random() * 2000);
});
```

This code creates a promise that will start in a few seconds after I make the request. Then there are a number of steps that need to be made in the `createSundae` function.

Indicated a Successful Request or a Failed Request

But once that's all done, how does JavaScript notify us that it's finished and ready for us to pick back up? It does that by passing two functions into our initial function. Typically we call these `resolve` and `reject` .

The function gets passed to the function we provide the Promise constructor - typically the word "resolve" is used to indicate that this function should be called when the request completes successfully. Notice the `resolve` on the first line:

```
new Promise(function (resolve, reject) {
    window.setTimeout(function createSundae(flavor = 'chocolate') {
        const sundae = {};
        // request ice cream
        // get cone
        // warm up ice cream scoop
        // scoop generous portion into cone!
        resolve(sundae);
    }, Math.random() * 2000);
});
```

Now when the sundae has been successfully created, it calls the `resolve` method and passes it the data we want to return - in this case the data that's being returned is the completed sundae. So the `resolve` method is used to indicate that the request is complete and that it completed *successfully*.

If there is a problem with the request and it couldn't be completed, then we could use the second function that's passed to the function. Typically, this function is stored in an identifier called "reject" to indicate that this function should be used if the request fails for some reason. Check out the `reject` on the first line:

```
new Promise(function (resolve, reject) {
    window.setTimeout(function createSundae(flavor = 'chocolate') {
        const sundae = {};
        // request ice cream
        // get cone
        // warm up ice cream scoop
        // scoop generous portion into cone!
        if ( /* iceCreamConeIsEmpty(flavor) */ ) {
            reject(`Sorry, we're out of that flavor :-(`);
        }
        resolve(sundae);
    }, Math.random() * 2000);
});
```

A Promise constructor takes a function that will run and then, after some amount of time, will either complete successfully (using the `resolve` method) or unsuccessfully (using the `reject` method). When the outcome has been finalized (the request has either completed successfully or unsuccessfully), the promise is now *fulfilled* and will notify us so we can decide what to do with the response.

Promises Return Immediately

The first thing to understand is that a Promise will immediately return an object.

```
const myPromiseObj = new Promise(function (resolve, reject) {
    // sundae creation code
});
```

That object has a `.then()` method on it that we can use to have it notify us if the request we made in the promise was either successful or failed. The `.then()` method takes two functions:

1. the function to run if the request completed successfully
2. the function to run if the request failed to complete

```
mySundae.then(function(sundae) {
    console.log(`Time to eat my delicious ${sundae}`);
}, function(msg) {
    console.log(msg);
    self.goCry(); // not a real method
});
```

As you can see, the first function that's passed to `.then()` will be called and passed the data that the Promise's `resolve` function used. In this case, the function would receive the `sundae` object. The second function will be called and passed the data that the Promise's `reject` function was called with. In this case, the function receives the error message "Sorry, we're out of that flavor :-(" that the `reject` function was called with in the Promise code above.