Callbacks & Promises

https://bitsofco.de/javascript-promises-101/

Promises

- A JavaScript Promise represents the result of an operation that hasn't been completed yet, but will at some undetermined point in the future.

- An example of such an operation is a network request.

- When we fetch data from some source, for example an API, there is no way for us to absolutely determine when the response will be received.

Callback Hell

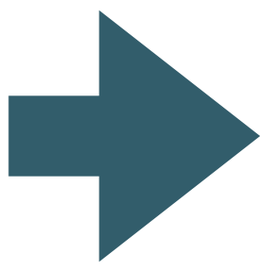
- This can be problematic if we have other operations dependent on the completion of this network request.
- Without Promises, we would have to use callbacks to deal with actions that need to happen in sequence.
- This isn't necessarily a problem if we only have one asynchronous action.
- But if we need to complete multiple asynchronous steps in sequence, callbacks become unmanageable and result in the infamous callback hell.

```
doSomething(function(responseOne) {
    doSomethingElse(responseOne, function(responseTwo, err) {
        if (err) { handleError(err); }
        doMoreStuff(responseTwo, function(responseThree, err) {
            if (err) { handleAnotherError(err); }
            doFinalThing(responseThree, function(err) {
                if (err) { handleAnotherError(err); }
                // Complete
            }); // end doFinalThing
        }); // end doMoreStuff
    }); // end doSomethingElse
}); // end doSomething
```

Promise Invocation Pattern

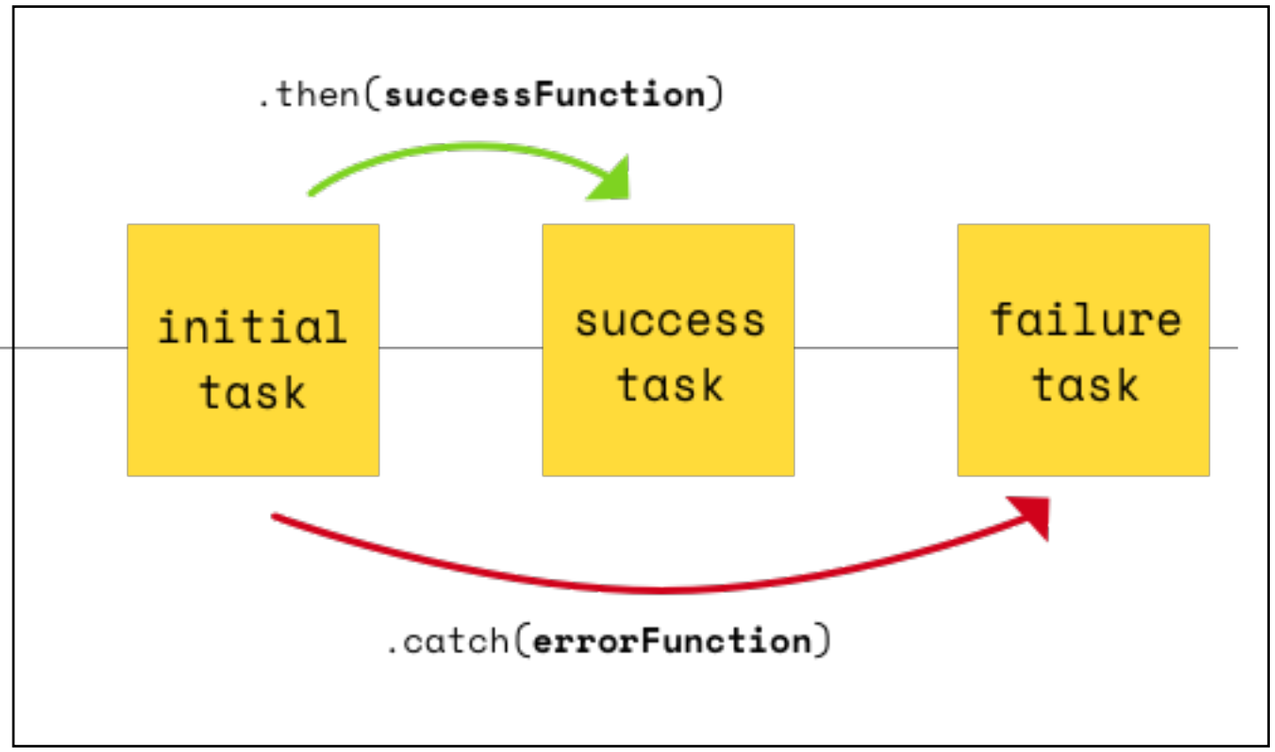
- Promises provide a standardised and cleaner method of dealing with tasks that need to happen in sequence.

```
doSomething(function(responseOne) {
    doSomethingElse(responseOne, function(responseTwo, err) {
        if (err) { handleError(err); }
        doMoreStuff(responseTwo, function(responseThree, err) {
            if (err) { handleAnotherError(err); }
            doFinalThing(responseThree, function(err) {
                if (err) { handleAnotherError(err); }
                // Complete
            }); // end doFinalThing
        }); // end doMoreStuff
    }); // end doSomethingElse
}); // end doSomething
```



```
doSomething()
.then(doSomethingElse)
.catch(handleError)
.then(doMoreStuff)
.then(doFinalThing)
.catch(handleAnotherError)
```

Using Promises (implicit)



- To execute a promisified function, we can call it like any regular function.

- Because it is returns a promise, we now have access to the .then() and .catch() methods, which indicate success and error results.

```
get(url)
.then(function(response) {
    /* successFunction */
})
.catch(function(err) {
    /* errorFunction */
})
```

Using Promises (explicit)

- Can also capture the promise, and explicitly trigger **then()** and **catch()** subsequently

```
get(url)
.then(function(response) {
    /* successFunction */
})
.catch(function(err) {
    /* errorFunction */
})
```

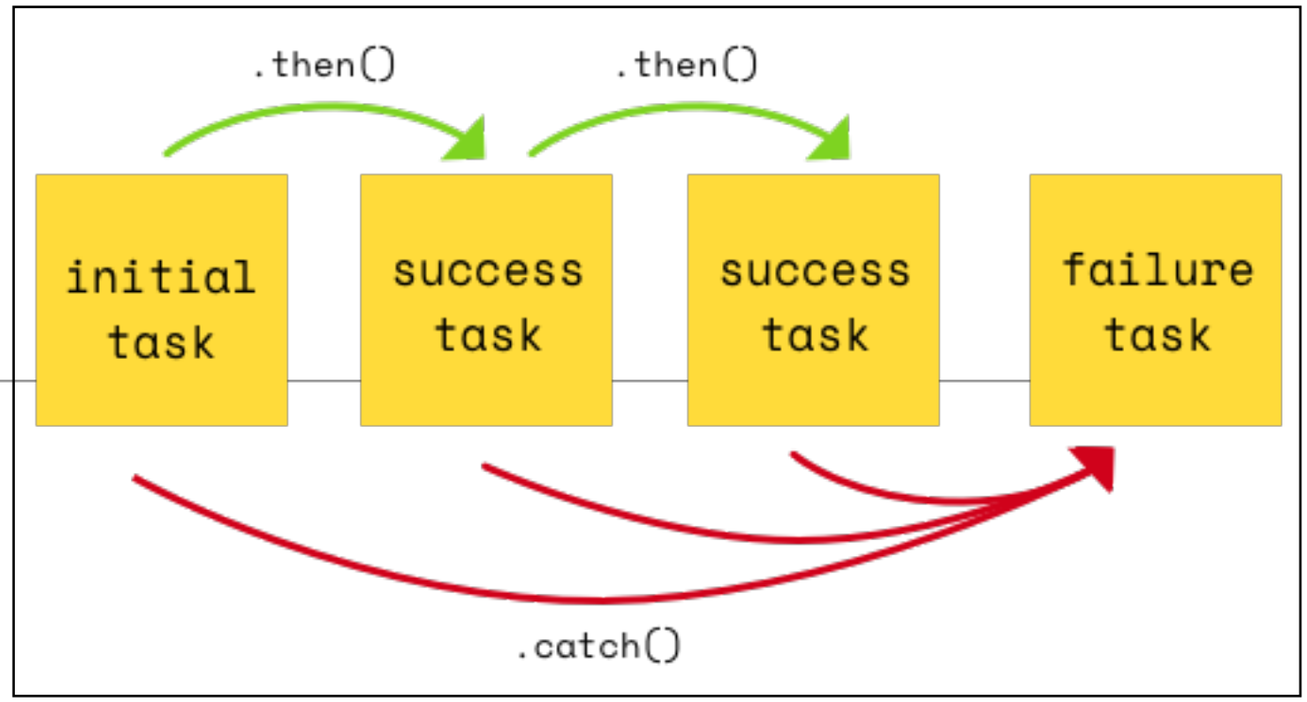
```
const promise = get(url);

promise.then(function (response) {
  /* successFunction */

});

promise.catch(function (err) {
  /* errorFunction */

});
```

Promise Chaining

- The real value in promises is when we have multiple asynchronous functions we need to execute in order.

- We can chain .then() and .catch() together to create a sequence of asynchronous functions.

- We do this by returning another promise within a success or error function




```
get(firstUrl).then(function (response) {

  response = JSON.parse(response);
  var secondURL = response.data.url;
  return get(secondURL);
  /* Return another Promise */

}).then(function (response) {

  response = JSON.parse(response);
  var thirdURL = response.data.url;
  return get(thirdURL);
  /* Return another Promise */

}).catch(function (err) {

  handleError(err);

});
```

Chaining with Arrow Functions

```
get(firstUrl).then(function (response) {

  response = JSON.parse(response);
  var secondURL = response.data.url;
  return get(secondURL);
  /* Return another Promise */

}).then(function (response) {

  response = JSON.parse(response);
  var thirdURL = response.data.url;
  return get(thirdURL);
  /* Return another Promise */

}).catch(function (err) {

  handleError(err);

});
```

```
get(url).then(response => {

  response = JSON.parse(response);
  var secondURL = response.data.url;
  return get(secondURL);
  /* Return another Promise */

}).then(response => {

  response = JSON.parse(response);
  var thirdURL = response.data.url;
  return get(thirdURL);
  /* Return another Promise */

}).catch(err => {

  handleError(err);

});
```

arrow functions

```
doSomething(function(responseOne) {
    doSomethingElse(responseOne, function(responseTwo, err) {
        if (err) { handleError(err); }
        doMoreStuff(responseTwo, function(responseThree, err) {
            if (err) { handleAnotherError(err); }
            doFinalThing(responseThree, function(err) {
                if (err) { handleAnotherError(err); }
                // Complete
            }); // end doFinalThing
        }); // end doMoreStuff
    }); // end doSomethingElse
}); // end doSomething
```

nested callbacks

```
get(url).then(response => {

  response = JSON.parse(response);
  var secondURL = response.data.url;
  return get(secondURL);
  /* Return another Promise */

}).then(response => {

  response = JSON.parse(response);
  var thirdURL = response.data.url;
  return get(thirdURL);
  /* Return another Promise */

}).catch(err => {

  handleError(err);

});
```

chained promises