Async JS

Carlos Alberto Lopez Montealegre

Event Handlers

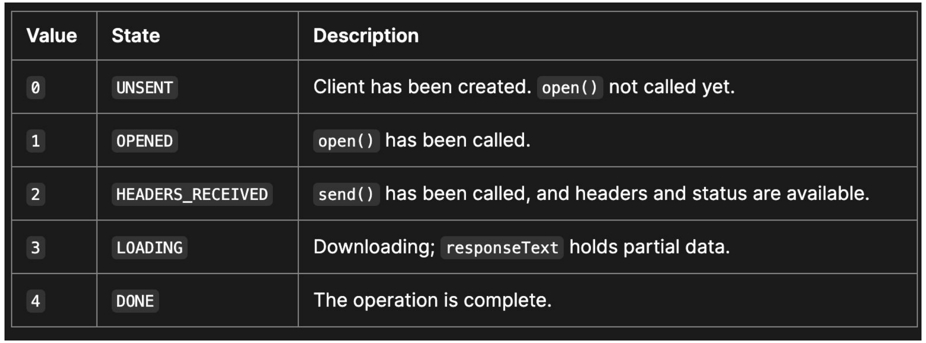
- Event handlers are a form of asynchronous programming
  - You provide a function that will not be called right away, but it will actually get called when the event the eventListener is listening for occurs
- Now we can listen to events such as an asynchronous function was completed. Hence, the callback of the event listener could notify the user that the async function was successfully finished
- There are JS apis such as XMLhttpRequest and Fetch
  - These two API’s enable you to make http requests to servers

A shallow overview of XMLhttpRequest

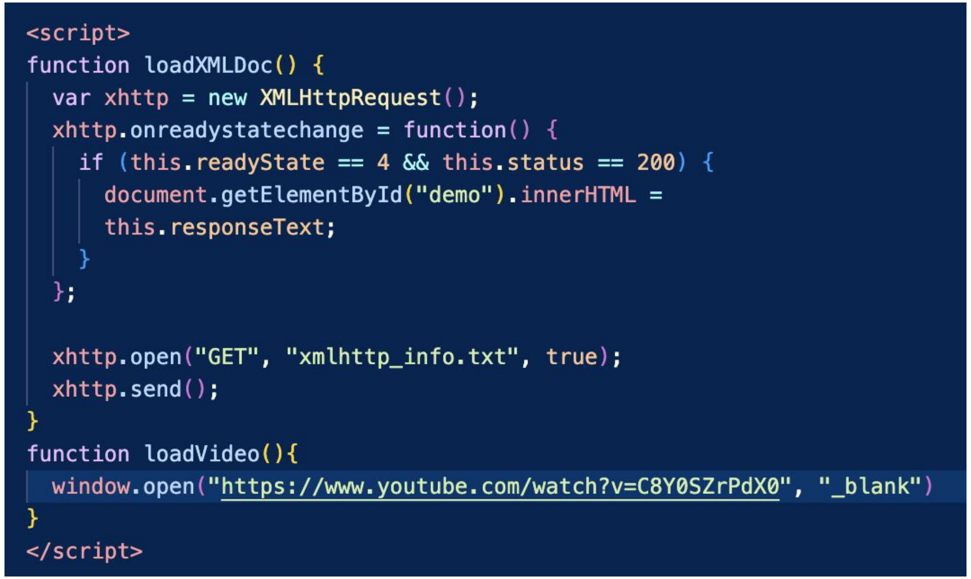
- Update a web page without reloading it
- Request data from a server after the page has loaded
- Receive data from a server after the page has loaded
- Send data to a server in the background

What difference do we see compared to traditional event listeners

- Here they are listening for a change in state for some object rather than a user action.
- 200 means ok



Value	State	Description
`0`	`UNSENT`	Client has been created. `open()` not called yet.
`1`	`OPENED`	`open()` has been called.
`2`	`HEADERS_RECEIVED`	`send()` has been called, and headers and status are available.
`3`	`LOADING`	Downloading; `responseText` holds partial data.
`4`	`DONE`	The operation is complete.



```
<script>
function loadXMLDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML =
      this.responseText;
    }
  };

  xhttp.open("GET", "xmlhttp_info.txt", true);
  xhttp.send();
}
function loadVideo(){
  window.open("https://www.youtube.com/watch?v=C8Y0SZrPdX0", "_blank")
}
</script>
```

Callbacks

- An event handler is a type of callback
- A callback is simply a function that is passed into another function
- However we want to avoid nested callback they are hard to read;
    - The solution is to make your callbacks named function that reduce the clutter of words in the call back

Let’s check some code that exemplifies this

- Lesson1/nestedCallback.html and Lesson1/nestedCallback.js

Promises

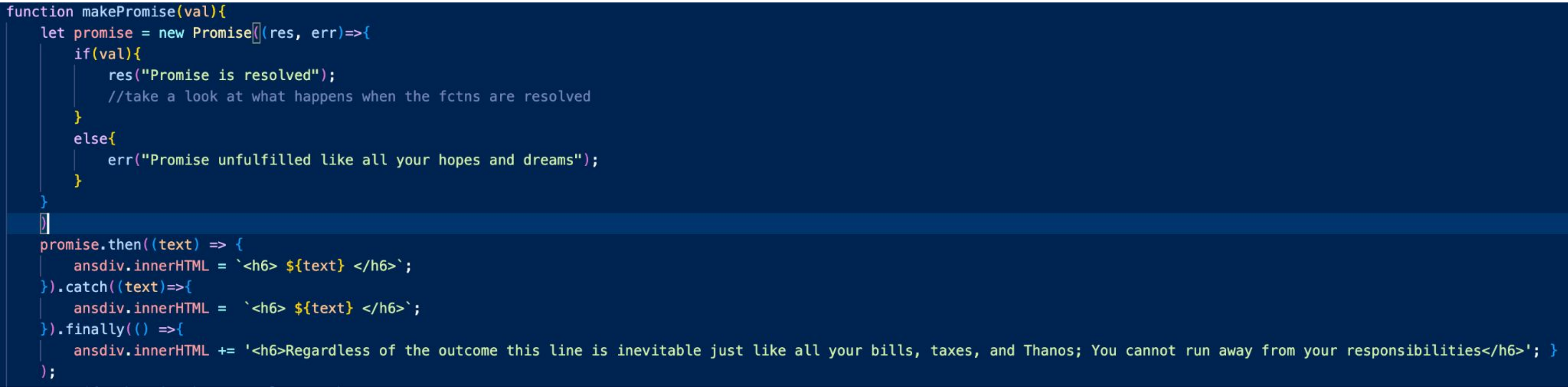
- A promise is an object that represents the eventual completion (or failure) of a procedure and its values
- A promise has 3 states
    - **Pending**: Neither fulfilled, nor rejected
    - **Fulfilled**: The operation completed successfully
    - **Rejected**: The operation failed

Promises have methods to handle their completion or failure

- Most commonly used promise methods
  - **then()**
    - Used to specify the programmed actions that follow if the promise fulfilled, meaning the request is successful
  - **catch()**
    - Used to specify what should happen when the promise is rejected, which is when the request is a failure
  - **finally()**
    - Used to perform actions after the promise has been settled
    - A promise is settled if it is either rejected or fulfilled
- Fun fact you can avoid using catch in particular scenarios!
- You can make a new promise using the following format: `new Promise((res, err) => {...})`

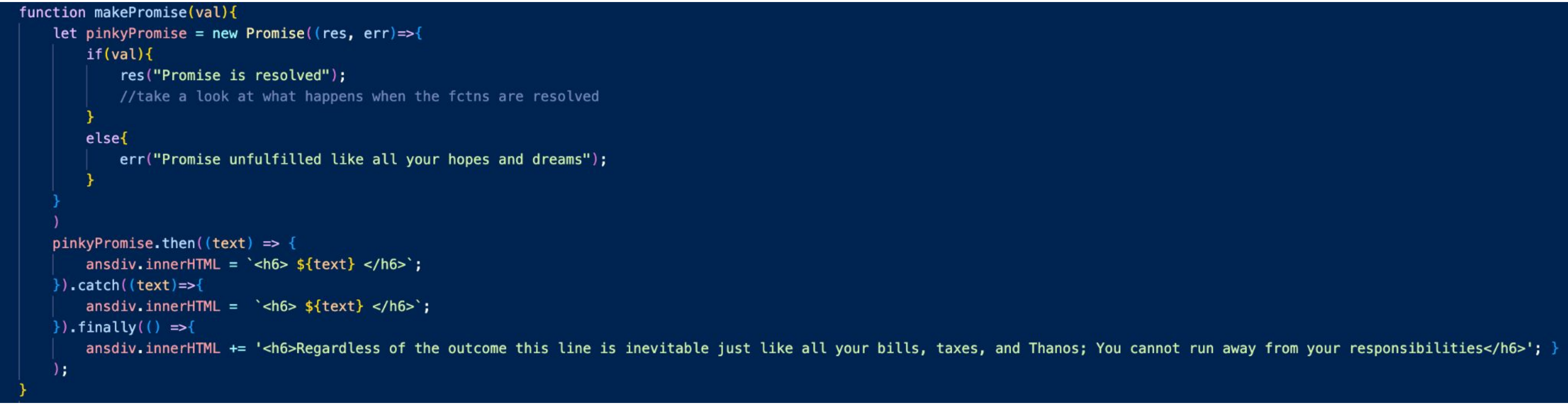
Promise Example

- I know what you are thinking thi is not a pinky promise, but it works for js purposes



```javascript
function makePromise(val){
    let promise = new Promise((res, err)=>{
        if(val){
            res("Promise is resolved");
            //take a look at what happens when the fctns are resolved
        }
        else{
            err("Promise unfulfilled like all your hopes and dreams");
        }
    }
    )
    promise.then((text) => {
        ansdiv.innerHTML = `<h6> ${text} </h6>`;
    }).catch((text)=>{
        ansdiv.innerHTML =  `<h6> ${text} </h6>`;
    }).finally(() =>{
        ansdiv.innerHTML += '<h6>Regardless of the outcome this line is inevitable just like all your bills, taxes, and Thanos; You cannot run away from your responsibilities</h6>'; }
    );
```

Ok maybe you can call your variable pinkyPromise =/



```
function makePromise(val){
    let pinkyPromise = new Promise((res, err)=>{
        if(val){
            res("Promise is resolved");
            //take a look at what happens when the fctns are resolved
        }
        else{
            err("Promise unfulfilled like all your hopes and dreams");
        }
    }
    )
    pinkyPromise.then((text) => {
        ansdiv.innerHTML = `<h6> ${text} </h6>`;
    }).catch((text)=>{
        ansdiv.innerHTML =  `<h6> ${text} </h6>`;
    }).finally(() =>{
        ansdiv.innerHTML += '<h6>Regardless of the outcome this line is inevitable just like all your bills, taxes, and Thanos; You cannot run away from your responsibilities</h6>'; }
    );
}
```

Combining Multiple Promises

- If you want your promises in order use Promise.all([p1, p2, …, pn]), where pi stands for the i-th promise
- If you do not care about the order of the promise fulfillment, use Promise.any([p1, p2, …, pn])

Async and Await

- async: adding it at the beginning of a function (fctn) makes the fctn async
- await: makes the code wait at the point where a promise was created such that it moves on only if the promise is settled (resolved or rejected)

Links

- https://medium.com/@uci.lasmana/what-is-ajax-xmlhttprequest-fetch-and-promise-ca8e4df9f8bd
- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing
- https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/number
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Links

- https://www.geeksforgeeks.org/how-to-append-html-code-to-a-div-using-javascript/