MDN - Link (Async & Await for React)

Async JS

- What does Asynchronous mean?
  - Doesn't need to wait for prior functions to "finish" executing, before the next function starts.
  - Features for multi-threading (making multiple threads)
  - Some Asynchronous features you have already worked with...
    > Events & event listeners
    > Click, or input something on your keyboard.

- Why do we need it?
  - We use it when communicating w/ APIs & Servers.
    > Time delay between the communication & response w/ another server over the internet.
    > We don't want to have to wait for the first https request to finish before we start the 2nd, 3rd, & 4th.
    > Talking to dozens of APIS or Servers.

~~How to use Promises~~

- Trouble w/ long-running synchronous functions
  - When programs take too long to respond, we find ourselves unable to do anything else on the website. →

To fix this, we need to do one of these options:

1. Start a long-running operation using a function.
2. Have a function that starts the operation &, return immediately, so that our program can still be responsive to other events.
3. Notify us w/ the results of the operation. when it eventually completes!

Event Handlers

Are a form of asynchronous programming where you provide a function ~~&~~ that will be called, not right away, but whenever the event happens.

Ex. XMLHttpRequest API

Callbacks

A callback is just a function that's passed into another function, where that callback will be called at an appropriate time.

They can get tricky & difficult to debug (callback hell)
We use Promises for a better foundation for asynchronous programming.

How to use Promises

A promise is an object returned by an asynchronous function, w/ch represents, the current state of an operation.

- Once the promise is returned to the caller, the operation isn't finished, but the Promise object provides methods to handle the eventual success / failure.

Using the fetch API

```
const fetchPromise = fetch("url");
console.log(fetchPromise);

fetchPromise.then((response) => { console.log(
                                        `Received Response: ${respons
                                                            Status}
                                  });

console.log("Started request...");
```

```
Output: Promise {<state>: "pending"}
        Starting request...
        Received ~~(request)~~ response: 200
```

→

Chaining Promises

Fetch API

Once you get a response object, you need to call another function to get the response data.
We usually want it as a JSON format.

Ex.
```
const fetchPromise = fetch("url");
fetchPromise.then((response) => {
    const jsonPromise = response.json();
    jsonPromise.then((data) => {
        console.log(data[0].name);
    });
});
```

Rewrite Code:
```
const fetchPromise = fetch('url');
fetchPromise
  .then((response) => {
    if(!response.ok) {
      throw new Error('HTTP error: ${response.~~error~~status}');
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  });
```

Catching Errors

Promise objects provide a catch() method for error handling.

- .then() - pass
- .catch() - fail

Ex.
```
const fetchPromise = fetch("url");
fetchPromise
    .then((response) => {
        if (!response.ok) {
            throw new Error(`HTTP error: ${response.status}`);
        }
        return response.json();
    })
    .then((data) => {
        console.log(data[0].name);
    })
    .catch((error) => {
        ~~console.log~~ console.error(`Could not get products:
                               ${error}`);
    });
```

Promises Terminology

Promises can come in three states:

① pending: the promise is created, & the asynchronous function it's associated w/ has not succeeded or failed yet. →

② fulfilled: the async function has succeeded.
③ rejected: the async function failed.

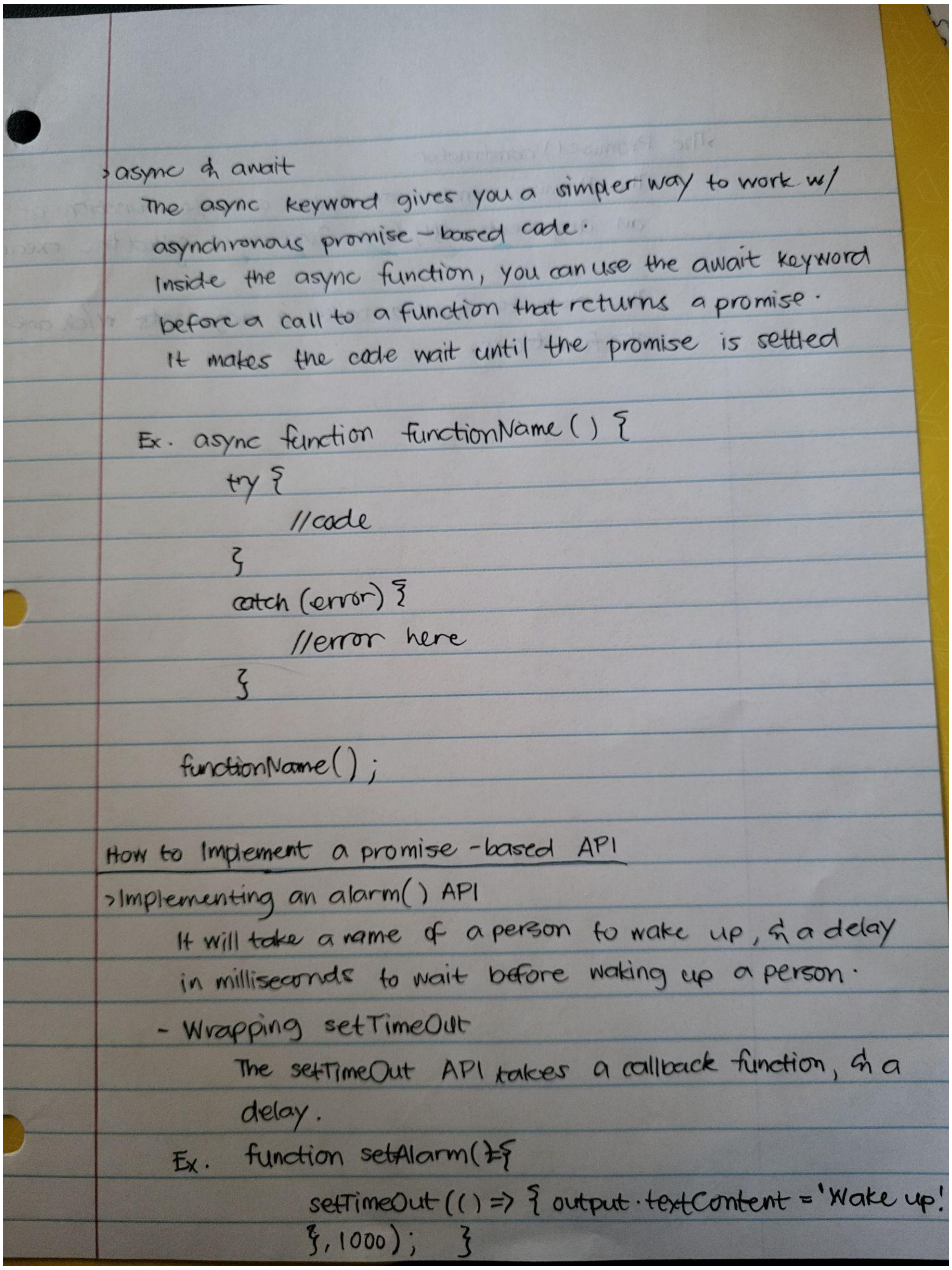
Note: we can use the term settled to cover both fulfilled &
rejected.
A promise is resolved if it is settled, or if has been
"locked in" to follow the state of another promise

Combining Multiple Promises

If you need all Promises to be fulfilled, but they don't depend on each other, you can start them all off together. You can use the Promise.all() method.

The Promise.all() method takes an array of promises & returns a single promise.
The Promis returned by Promise.all():
① fulfilled: when & if all the promises in the array are fulfilled.
② rejected: when ~~any)~~ & if any of the promises in the array are rejected.

Promise.any() is used when you need any one set of promises to be fulfilled, & don't care w/ch one.



>async & await

The async keyword gives you a simpler way to work w/ asynchronous promise-based code.

Inside the async function, you can use the await keyword before a call to a function that returns a promise. It makes the code wait until the promise is settled

Ex.
```
async function functionName() {
    try {
        //code
    }
    catch (error) {
        //error here
    }

functionName();
```

How to Implement a promise-based API

>Implementing an alarm() API

It will take a name of a person to wake up, & a delay in milliseconds to wait before waking up a person.

- Wrapping setTimeOut

The setTimeOut API takes a callback function, & a delay.

Ex.
```
function setAlarm(){
    setTimeOut(() => { output.textContent = 'Wake up!
    }, 1000);  }
```

The Promise Constructor

- The Promise constructor takes a single function as an argument. (The function is called the executor).
- The executor function takes 2 arguments w/ch are also functions. They are called
    - Resolve - success

    - Reject - fails

Introducing Workers