Asynchronous JS

Sync vs Async

- Synchronous code: each statement executed to completion before the next is executed
- Asynchronous code: a statement or statements is scheduled to be executed in the future
- The **stack** portion of memory stores functions in execution and their local variables, while the **heap** stores objects
- The **queue** portion of memory stores messages waiting to be processed, e.g. an event handler; code added to the queue is executed asynchronously
- JS in the browser is effectively single-threaded and non-blocking

setTimeout

- `setTimeout` adds a handler to the queue that will be processed after a specified delay and only when the stack is empty, e.g.:

  ```
  setTimeout(() => {
    console.log('After 1 sec');
  }, 1000);
  ```

- The handler is executed asynchronously, i.e. it will be executed at some point in the future

Callbacks

- A callback is a function that is invoked after something else happens

- Can you spot the problem below?

```
const longRunningTask = () => {
  let result = null;
  setTimeout(() => result = 42);
  return result;
}
```

AJAX

- AJAX is short for Asynchronous JS and XML; it is the mechanism by which HTTP requests are executed programmatically

```
const req = new XMLHttpRequest();
req.onload = () => {
  console.log(req.response);
}
req.open('GET', url);
req.send();
```

- When the request's load event is fired the handler is added to the queue and will be executed in its turn

Common Problems

- The principle problem with traditional AJAX code is how one deals with chaining asynchronous tasks together
- Other problems arise from a misunderstanding of the nature of JS in the browser - what's wrong with this code?

```
const now = () => new Date().getTime();
console.log('Pausing...');
let start = now();
while (now() - start < 3000);
console.log('Resuming...');
```

Promises

- A `Promise` is an object that wraps some asynchronous code; it facilitates the chaining of asynchronous tasks and error handling

```
const promise = new Promise(
  (successCb, failureCb) => {

    // Do async work
    if (success) {
      successCb(result);
    } else {
      failureCb(error);
    }
});
promise.then(console.log).catch(console.err);
```

fetch

- `fetch` simplifies the making of HTTP requests programmatically; it creates and returns a Promise that encapsulates AJAX code

```
fetch(url)
   .then(response => response.json())
   .then(console.log)
   .catch(console.err);
```

- fetch is OK but requires explicit parsing of the response body and non-GET requests require a configuration object argument; axios is a nice alternative

async await

- The `async` and `await` keywords facilitate the writing of asynchronous code in a synchronous manner
- An async function is one that returns a promise, regardless of whether or not it is actually asynchronous
- The await keyword is used to conceptually force the interpreter to wait for an async task to complete (it compiles into a Promise) and can only be used inside a async function

```
const getApiData = async url => {
  await const response = fetch(url);
  await const json = response.json();
  return json;
}
```

Summary

- Asynchronous code is code that is scheduled to be executed in the future
- setTimeout (and setInterval) are asynchronous
- Callbacks are the mechanism commonly used to execute code on completion of some asynchronous task
- AJAX is the mechanism for making HTTP requests programmatically
- A Promise is an object is an object that wraps some asynchronous task
- fetch returns a Promise that encapsulates AJAX code
- The async and await keywords are used to execute asynchronous code in a synchronous manner