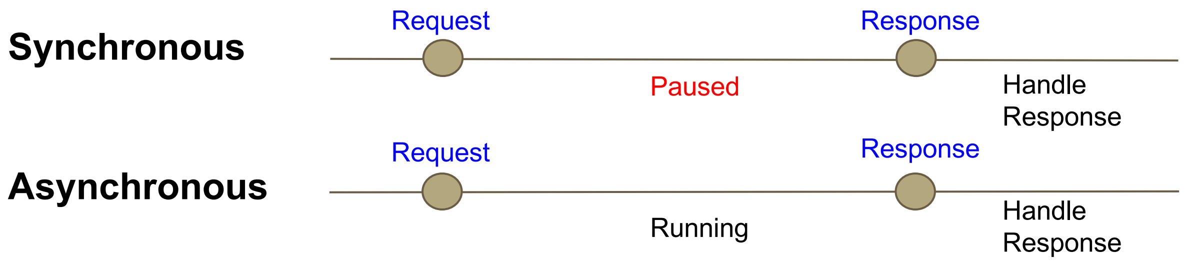
Asynchronous Programming

Rich Web Application Technologies

Synchronous vs Asynchronous

- Whenever we do input/output programming we invariably encounter some delay in fetching results from the external data source
- That delay is relatively long with respect to program execution
- A *synchronous* approach pauses the current context's execution to wait for the result to be retrieved. If a function depends on the result of another function, it needs to wait until this function finishes.
- An *asynchronous* approach allows execution to continue and fetches the result some time later on



Interactive User Interfaces

- User interfaces must be responsive to user input so completely pausing execution while waiting for a slow request to complete is not an option
- However a synchronous approach in UI programming is possible in a multithreaded environment
- However, Javascript and the browser do not support multithreaded programming so the asynchronous approach is what is used
- A basic way that asynchronous programming can work is through the use of callback functions or callbacks

Synchronous programming

- When a web app executes a complex block of code, it doesn't return the control to the browser, making it look like it's frozen. This is called **blocking**, and can be a frustrating situation to the user
- Check the following link for an example. This application runs a time-consuming operation every time the user clicks the button: it calculates 10 million dates and displays the result to the console).

https://mdn.github.io/learning-area/javascript/asynchronous/introducing/simple-sync.html

Asynchronous x Multithreaded

- Asynchronous environment: a single process thread runs all the time, but it may, for event-driven reasons, switch from one function to another.

- Multi-threaded environment: many individual threads of programming are running at the same time. These are difficult and there are issues of threads locking each other's memory to prevent them from overrunning one another. Multiple threads can use multiple cores to complete multiple tasks:

  Thread 1: Task A --> Task B

  Thread 2: Task C --> Task D

JavaScript is single-threaded

- By default, JavaScript follows a single-threaded approach
- Web workers API can help JS to be multi-threaded, but cannot modify the DOM
- There are, however, functions that allow you to “schedule a call”, by setting up an interval before executing another function (which we call a **callback function**)

setTimeOut( )

- This function executes a particular block of code after a specified time
- Time is given in milliseconds (1000 milliseconds = 1 second)

```
let myGreeting = setTimeout(() => {
alert('Hello, Mr. Universe!');
}, 2000);
```

setTimeInterval( )

- This function executes a particular block of code repeatedly considering a specified number of milliseconds
- This can be useful if you need to use JS to animate an image, for example

```
setInterval(myFunction, 1000);

function myFunction() {
let d = new Date();
document.getElementById("demo").innerHTML=
d.getHours() + ":" +
d.getMinutes() + ":" +
d.getSeconds();
}
```

Callback Functions

- A callback function is a function that you provide as an argument to the I/O request function which implements the logic to be executed at the point when the request has been fulfilled

```
const url = "https://s3.amazon.com/23123fd123/95fdae";
const handler = (data) => {
  // do something with data ...
};
getSomeData(url, handler);  // named callback

// inlined callback (anonymous function)
getSomeData(url, data => {
  // do something with data ...
});
```

Callback Functions

- When we pass a callback function as an argument to another function, we are only passing the function's reference as an argument - the callback function is **not** executed immediately. It is “called back” asynchronously somewhere inside the containing function’s body. The containing function is responsible for executing the callback function when the time comes.

Using Callbacks

- The callback style can be used whenever the code needs to make a request which will take a relatively long time
- Examples include reading data from a network or from disk storage
- As soon as the request is made, it is queued by the browser and the callback function is stored for subsequent execution
- Meanwhile control is returned back to the executing context and the program proceeds
- During this time, the app may handle user input events or make futher async requests
- When the original request is fulfilled, the callback is called

Problems With Callbacks

- If a further asynchronous request needs to be made on foot of a response from a previous one, then the second asynchronous request must be made with the first request’s callback logic
- This leads to a pattern of nested callbacks
- Nesting callbacks is difficult to reason about, difficult to debug and difficult to handle error cases

```
getSomeData(url1, data => {
  getSomeMoreData(url2, data => {
    getEvenMoreData(url3, data => {
      // do something here ...
    });
  });
});
```

Summary

- In the single-threaded context of JS in the browser, asynchronous programming is how we allow execution to proceed, even when waiting for responses from relatively long-running requests
- The simplest approach is to use the function-as-an-argument facility in the language and provide a callback or response handler
- This works quite well for simple cases of depth one or two but becomes a problem when response handlers need to be nested deeply
- Later on we’ll see alternative approaches to async handling