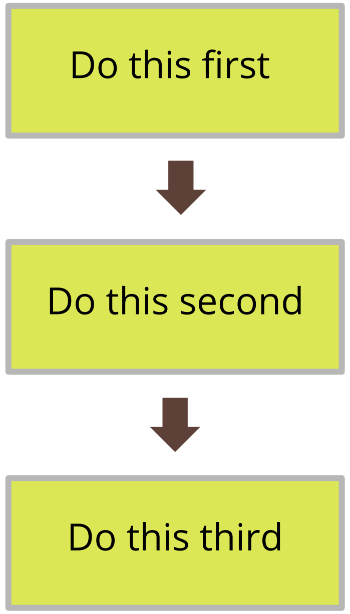
JavaScript Event Loop

The Event Loop In JavaScript

The concurrency model and event loop in JavaScript is different from what you would see in other languages like C or Java.



Consider This Script

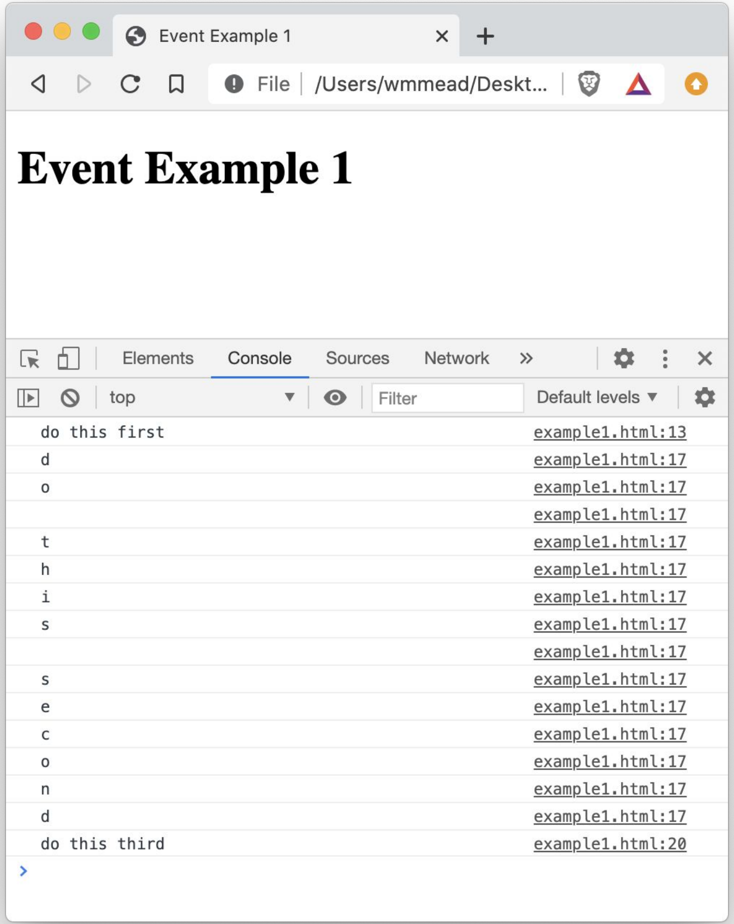
Consider the script on the right.

If you add this to a web page, the results should be fairly obvious.

```
console.log("do this first");

var second = "do this second";
for (var eachLetter of second) {
    console.log(eachLetter);
}

console.log("do this third");
```

Example 1 Results

The first console log statement runs, then the loop starts and runs until it is 100% complete, then the third statement runs.



Event Example 1

```
do this first                example1.html:13
d                            example1.html:17
o                            example1.html:17
                             example1.html:17
t                            example1.html:17
h                            example1.html:17
i                            example1.html:17
s                            example1.html:17
                             example1.html:17
s                            example1.html:17
e                            example1.html:17
c                            example1.html:17
o                            example1.html:17
n                            example1.html:17
d                            example1.html:17
do this third                example1.html:20
>
```

Example 2

Let’s examine how this code works through the stack.

The difference here is that these functions interact with each other a little bit.

```
function mkUpperCase(message) {
    var upper = message.toUpperCase();
    return upper;
}

function mkLowerCase(message) {
    var lower = message.toLowerCase();
    return lower;
}

function outputMessages(message) {
    var output = `Uppercase version: ${mkUpperCase(message)}\n`;
    output += `Lowercase version: ${mkLowerCase(message)}`;
    console.log(output);
}

outputMessages('Llamas are THE best!');
```

Example 2 Stack

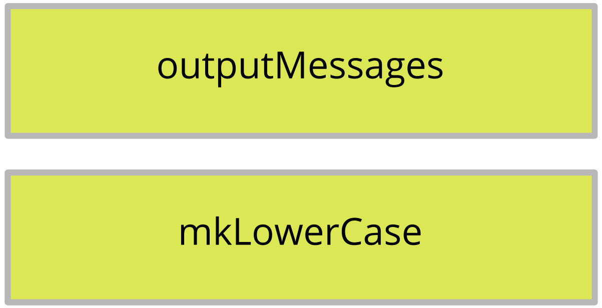
Some functions are defined, then the first thing that enters the event loop stack is this call to run the function:

```
outputMessages('Llamas are THE best!');
```

This function calls mkUpperCase, which enters the stack, it returns, and pops off the stack.

Then mkLowerCase is called, so it enters the stack. It returns and pops off the stack.

And finally outputMessages runs send the messages to the console.log and pops off the stack.



Example 3

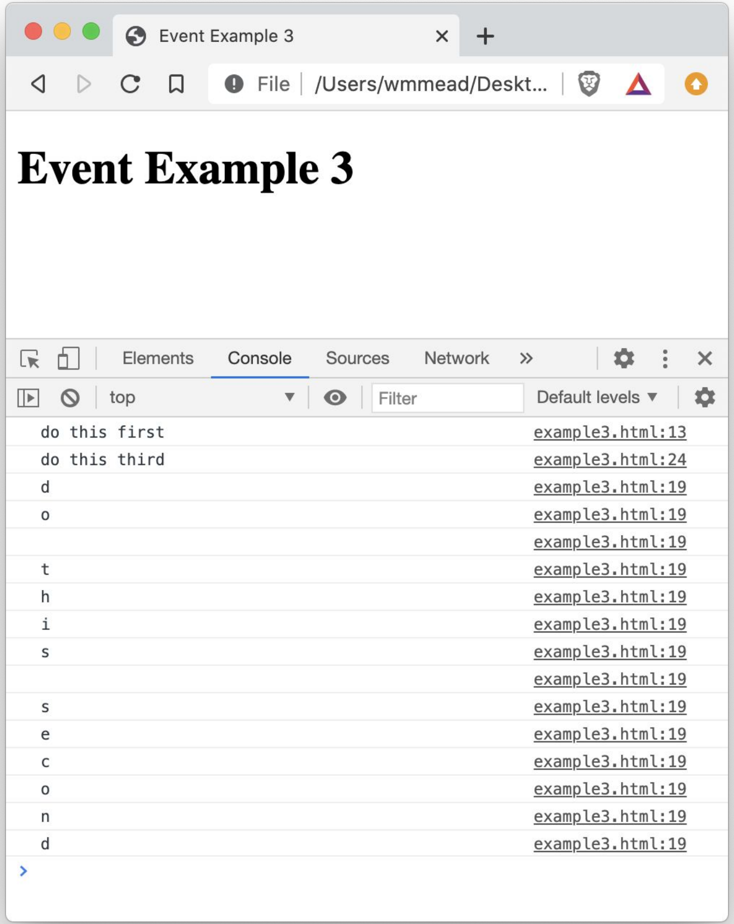
Consider this code. Run it in the browser to see what you get in the console.

```
console.log("do this first");

setTimeout(function () {

    var second = "do this second";
    for (var eachLetter of second) {
        console.log(eachLetter);
    }

}, 2000);

console.log("do this third");
```

Example 3 Results

Here the first item is firing, and then the second is set for a time out so it is waiting for two seconds, but while that is waiting the third item in the list enters the event loop stack and is processed.

Finally, when the two seconds are up, the second item, in the loop, runs.



Event Example 3

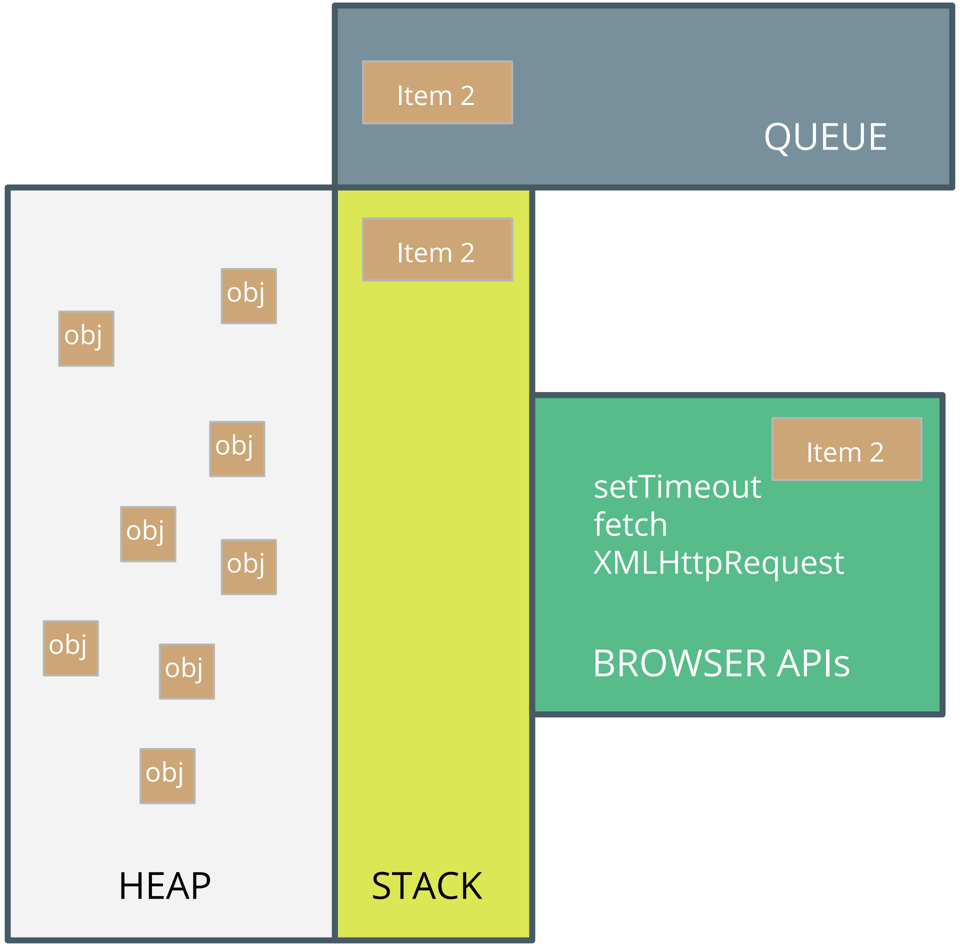
```
do this first          example3.html:13
do this third          example3.html:24
d                      example3.html:19
o                      example3.html:19
                       example3.html:19
t                      example3.html:19
h                      example3.html:19
i                      example3.html:19
s                      example3.html:19
                       example3.html:19
s                      example3.html:19
e                      example3.html:19
c                      example3.html:19
o                      example3.html:19
n                      example3.html:19
d                      example3.html:19
>
```

Updated Event Loop

The heap is mostly unstructured memory.

The stack is where the events get handled in order by the JS engine.

The queue is where browser API calls wait to be completed, where they can then enter back into the stack, once the stack is empty.



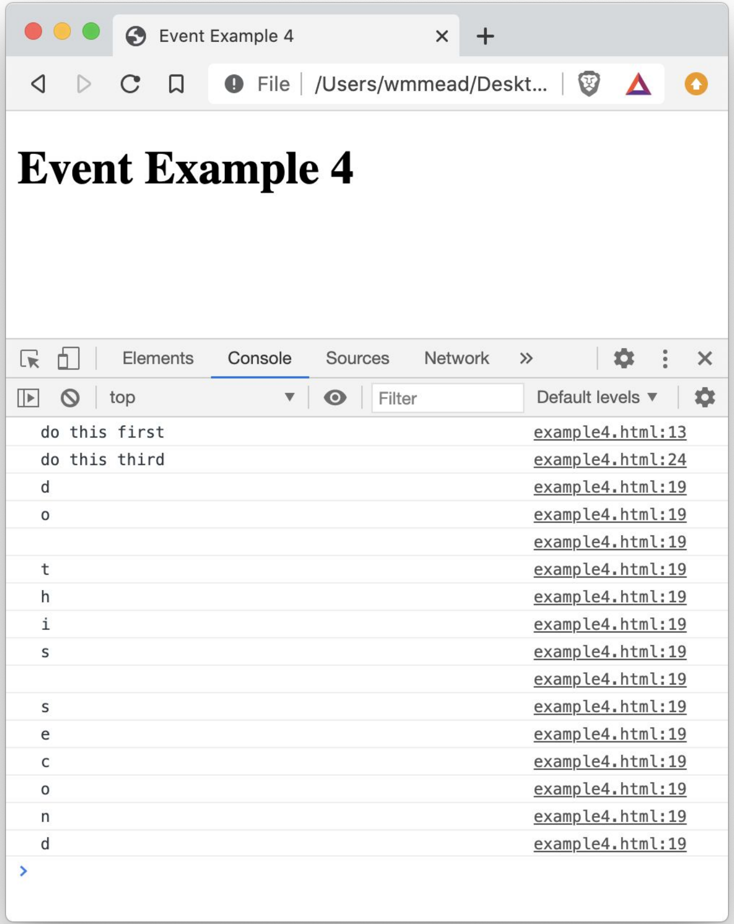
Example 4

Try this code instead. It is the same as example 3, but instead of waiting two seconds, it is set to run at zero seconds.

```
console.log("do this first");

setTimeout(function () {

    var second = "do this second";
    for (var eachLetter of second) {
        console.log(eachLetter);
    }

}, 0);

console.log("do this third");
```

Example 4 Result

Even though we set the setTimeout to wait zero seconds, the JS engine is still sending that call to the browser API and it is still going into the queue, so it will still run after the "do this third" console.log call.



Event Example 4

```
do this first                                    example4.html:13
do this third                                    example4.html:24
d                                                example4.html:19
o                                                example4.html:19
                                                 example4.html:19
t                                                example4.html:19
h                                                example4.html:19
i                                                example4.html:19
s                                                example4.html:19
                                                 example4.html:19
s                                                example4.html:19
e                                                example4.html:19
c                                                example4.html:19
o                                                example4.html:19
n                                                example4.html:19
d                                                example4.html:19
>
```

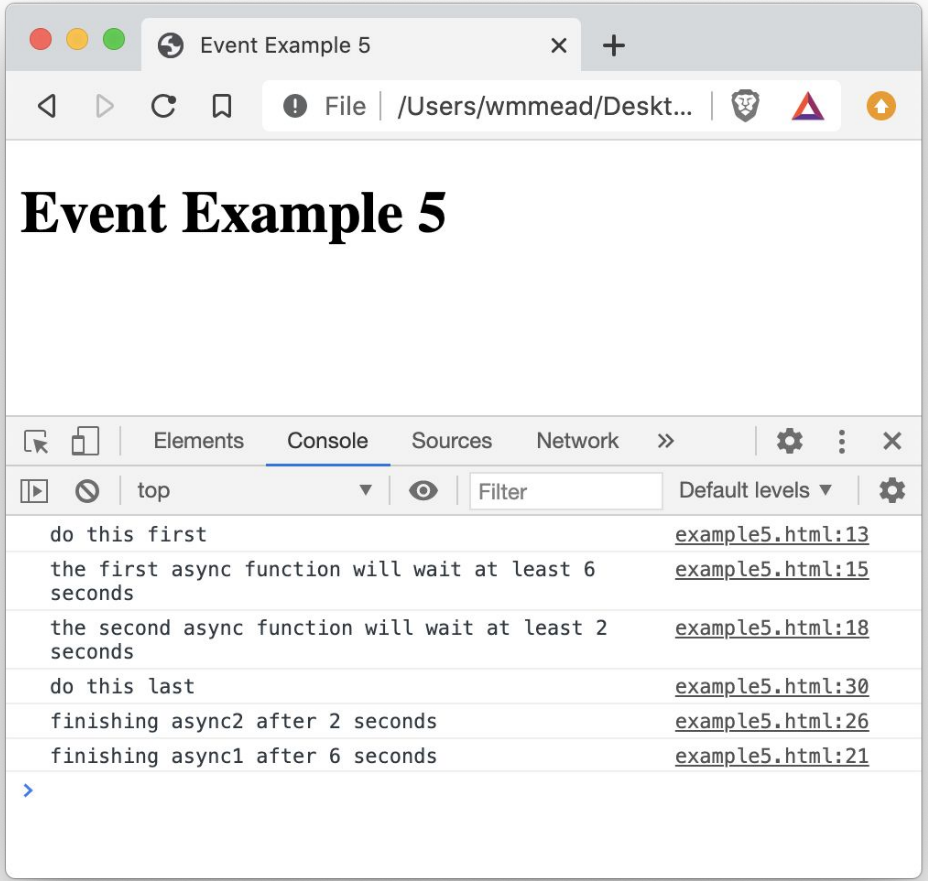
Example 5

Check out the example on the right. This example has two asynchronous calls, which run for a random amount of time.

```
console.log("do this first");
var rand1 = Math.ceil(Math.random() * 10);
console.log(`the first async function will wait at least
${rand1} seconds`);

var rand2 = Math.ceil(Math.random() * 10);
console.log(`the second async function will wait at least
${rand2} seconds`);

setTimeout(function () {
    console.log(`finishing async1 after ${rand1} seconds`);

}, rand1 * 1000);

setTimeout(function () {
    console.log(`finishing async2 after ${rand2} seconds`);

}, rand2 * 1000);

console.log("do this last");
```

Example 5 Result

Notice, depending on the random numbers generated, that the two asynchronous functions could complete in either order.

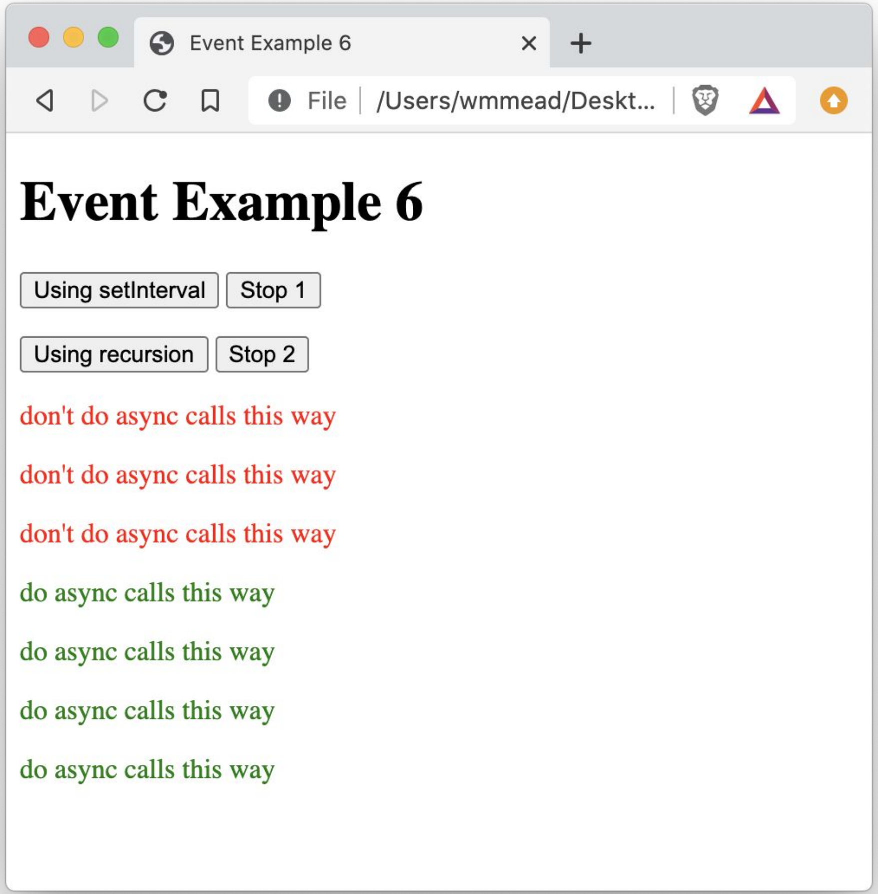


Event Example 5

```
do this first                                       example5.html:13
the first async function will wait at least 6       example5.html:15
seconds
the second async function will wait at least 2      example5.html:18
seconds
do this last                                        example5.html:30
finishing async2 after 2 seconds                    example5.html:26
finishing async1 after 6 seconds                    example5.html:21
```

Example 6

One last thing to talk about, regarding asynchronous functions: If you want to do something asynchronously on an interval, it is better to use setTimeout recursively, rather than use setInterval.



Event Example 6

```
don't do async calls this way
don't do async calls this way
don't do async calls this way
do async calls this way
do async calls this way
do async calls this way
do async calls this way
```

Don’t Use setInterval for Async Functions

Imagine if this line were actually doing something asynchronous.

```
bttnEx1.addEventListener('click', function () {
    let badIdea = setInterval(function () {
        container1.innerHTML += "<p>don't do async calls this way</p>";
    }, 1500);
});
```

Using Recursion is Better

This recursive function is better because if this line were asynchronous, it would wait until that process completes before going on to check the condition of the stopRecursion variable, and then potentially running the doRecursion function again.

```
function doRecursion() {
    setTimeout(function () {
        container2.innerHTML += "<p>do async calls this way</p>";
        if (stopRecursion != 'stop') {
            doRecursion();
        }
    }, 1500);
}
```

From the MDN:

When your code has the potential to take longer to run than the time interval you've assigned, it's better to use recursive setTimeout() — this will keep the time interval constant between executions regardless of how long the code takes to execute, and you won't get errors.

Summary

The event loop and queue can be confusing topics, but it is important to understand how JavaScript works with these.