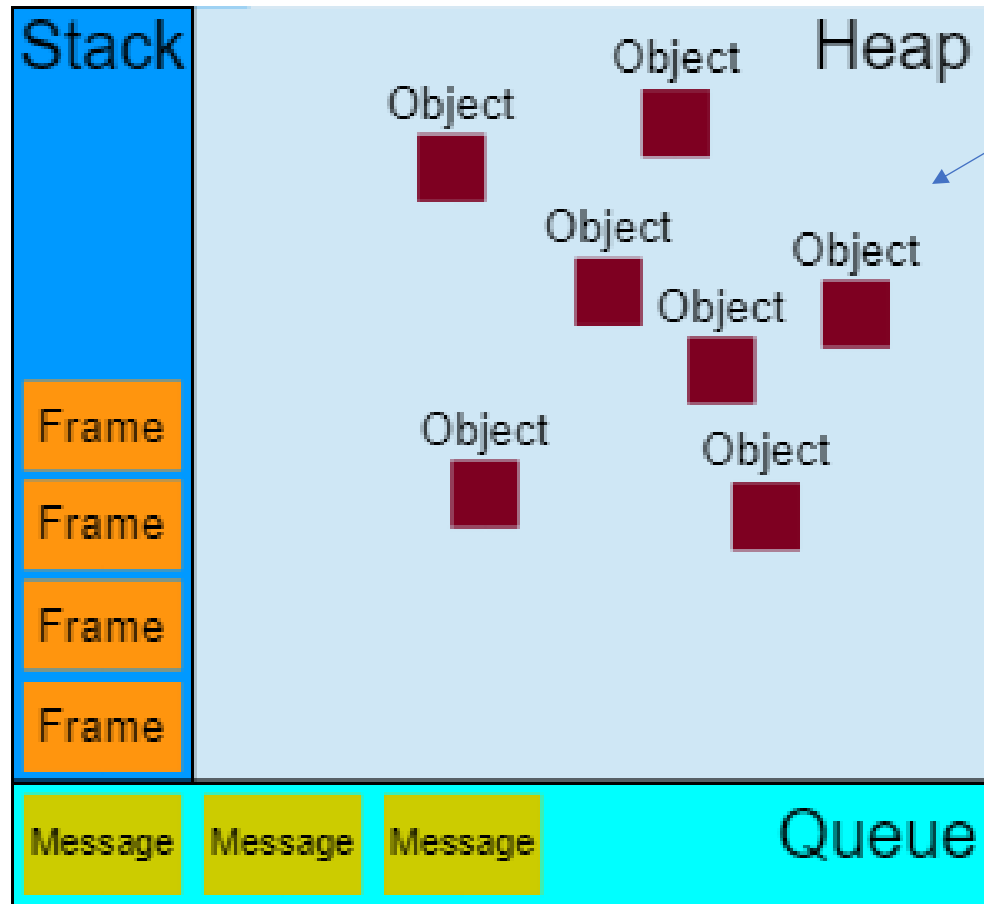




# Basic Event Handling Model (Synchronous)

JavaScript has a runtime model based on an event loop, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks



Heap: Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

```
function foo(b) {  
  const a = 10;  
  return a + b + 11;  
}
```

```
function bar(x) {  
  const y = 3;  
  return foo(x * y);  
}
```

```
const baz = bar(7); // assigns 42  
to baz
```

1. When calling bar, a first frame is created containing references to bar's arguments and local variables.
2. When bar calls foo, a second frame is created and pushed on top of the first one, containing references to foo's arguments and local variables.
3. When foo returns, the top frame element is popped out of the stack (leaving only bar's call frame).
4. When bar returns, the stack is empty.

## Message (event) queue

Each event/message has an associated function that gets called to handle the message.

Message is removed from the queue and its corresponding function is called with the message as an input parameter.

Calling a function creates a new stack frame for that function's use. The processing of functions continues until the stack is once again empty.

Then, the event loop will process the next message in the queue (if there is one).

## Event Loop:

```
while (queue.waitForMessage()) {  
    queue.processNextMessage();  
}
```

## **The difference between event handlers and event listeners**

The terms event handler and event listener are often used interchangeably. However, there is a slight difference between the two. These are two ways of handling events.

When a code runs after an event takes place, this is known as registering an event handler. On the other hand, the event listener listens to the event and then triggers the code for handling the event.

Also, Event handler for particular event

Event listener tied to object for different events



- Process Blocking
  - e.g. request for some service and waiting for a reply
- Synchronous programming
  - Waits until not blocking anymore (until the service comes back with results)
  - Do things one at a time
- Asynchronous programming
  - Let the wait go on / Do some other unrelated stuff
  - Alternative: Create another thread and go parallel or concurrent
    - Javascript has only one thread – need some added mechanism

//initialization code

const posts = loadCollection(postsUrl);

const post = getRecord(posts, "001");

const comments = loadCollection(commentsUrl);

const postComments = getCommentsByPost(comments, post.id);

console.log(post);

console.log(postComments);

Loading a file from  
internet

Wait until this file opening  
is finished even it takes a  
long long time





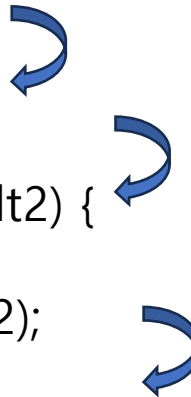
# Callback function: First class object (passable)

```
function first(cb) {  
  setTimeout(function() {  
    return cb('first');  
  }, 0);  
}
```

```
function second(cb) {  
  return cb('second');  
}
```

```
function third(cb) {  
  return cb('third');  
}
```

```
first(function(result1) {  
  console.log(result1);  
  second(function(result2) {  
    console.log(result2);  
    third(function(result3) {  
      console.log(result3);  
    }); // third  
  }); // second  
}); // first
```



First schedules passed CB1 execution  
(using timer)

Second just executes passed CB2

Third just executes passed CB3

Schedules first to execute ...

when first executes

Print result1

Execute second

Prints result 2

Execute third

Prints result3

So this will print "first", "second", and then "third".

Callbacks let us force tasks to execute  
Sequentially.

--> But strange and difficult to understand

Callbacks are like nested !

```
function task1() {  
    setTimeout(function() {  
        console.log('first');  
    }, 0);  
}
```

```
function task2() {  
    console.log('second');  
}
```

```
function task3() {  
    console.log('third');  
}
```

```
task1();  
task2();  
task3();
```

This example will print “second”, “third”, and then “first”. It doesn’t matter that the `setTimeout` function has a 0 delay. It is an asynchronous task in JavaScript, so it will always be deferred to execute later.

SetTimeout finish ... then print 'first'

```
function loadCollection(url) {  
  try {  
    const response = fs.readFileSync(url, 'utf8');  
    return JSON.parse(response);  
  } catch (error) {  
    console.log(error);  
  }  
}
```

The `readFileSync` method opens the file synchronously. But this is not the best way to open the file because this is a potentially blocking task.

Opening the file should be done asynchronously so that the flow of execution can be continuous (assuming rest of code is not dependent)

```
function loadCollection(url, callback) {  
  fs.readFile(url, 'utf8', function(error, data) {  
    if (error) {  
      console.log(error);  
    } else {  
      return callback(JSON.parse(data));  
    }  
  });  
}
```

We could include a callback to do some post processing once the file is open

- Asynchronous programming is a method used in our code to defer events for later execution. When you are dealing with an asynchronous task, callbacks are one solution for timing our tasks so they execute sequentially.
- If we have multiple tasks that depend on the result of previous tasks, one solution is to use multiple nested callbacks. However, this could lead to a problem known as "callback hell." Promises solve the problem with callback hell, and async functions let us write our code in a synchronous way.



# Javascript event model

- (Single) Call stack --> for procedure call handling
  - Monolithic call stack structure alone cannot support asynchronous programming
  - Owing to the basic Javascript being single threaded
    - One thing at a time
- But Javascript also has all those things: asynchoronous call support ???
  - Promise, Async, Await, SetTimeout, ...
- It turns out there is something added on to Javascript basic engine to make this happen
  - Event loop and Callback queue
  - Part of the Browser ... but who cares ...

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>

# Call Stack

```
function multiply(a, b) {  
    return a * b;  
}
```

```
function square(n) {  
    return multiply(n, n);  
}
```

```
function printSquare(n) {  
    var squared = square(n);  
    console.log(squared);  
}
```

```
printSquare(4);
```

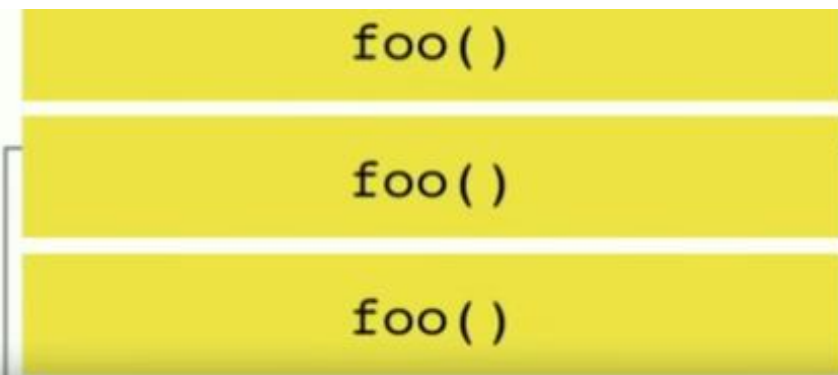
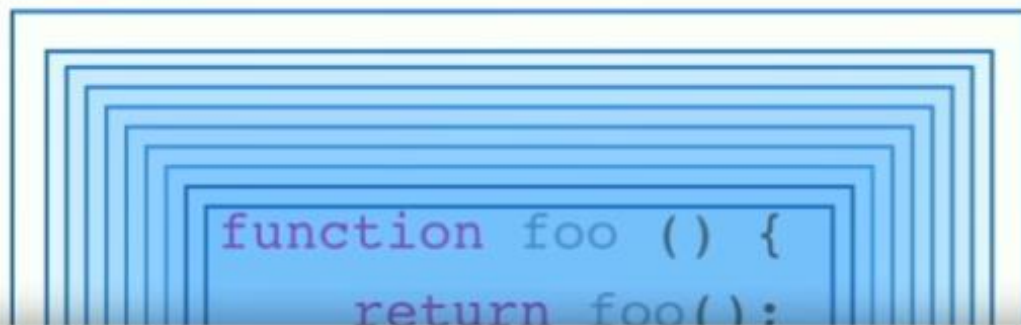
stack

multiply(n, n)

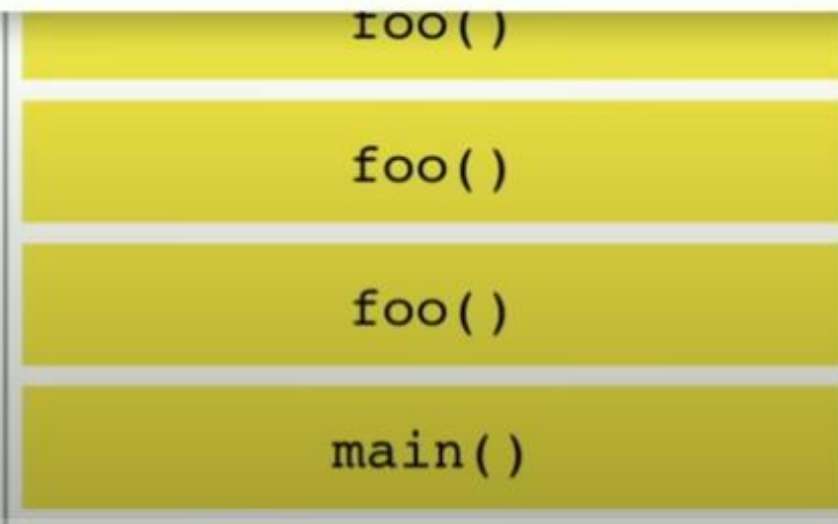
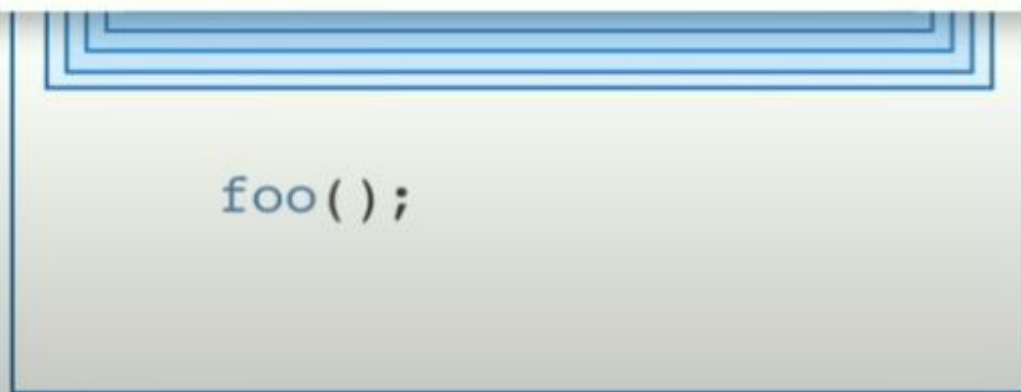
square(n)

printSquare(4)

main()



**X** **RangeError: Maximum call stack size exceeded**





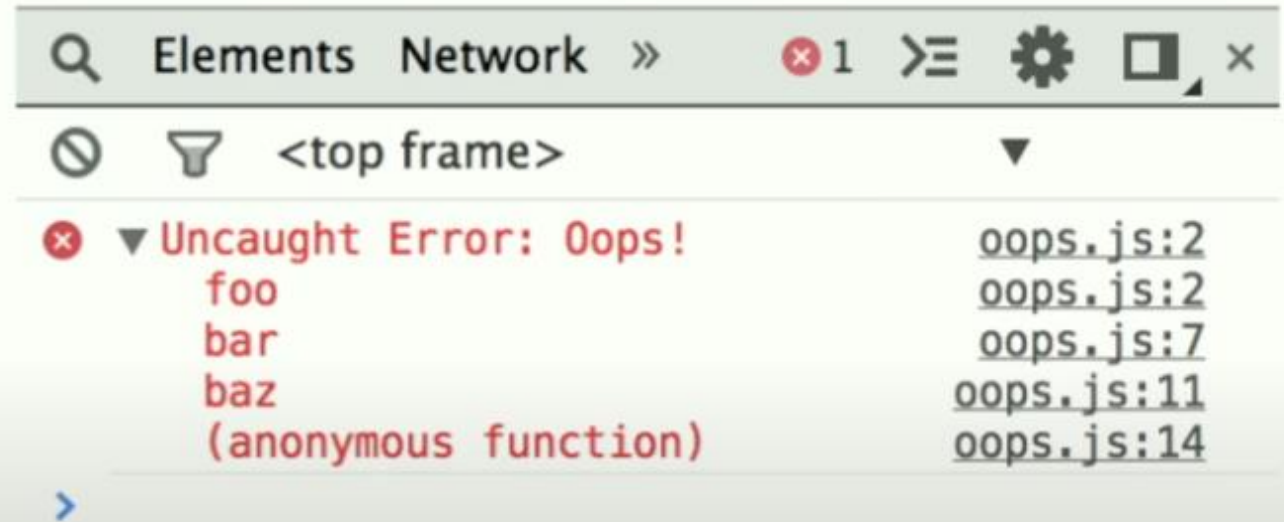
```
function foo() {  
  throw new Error('Oops!');  
}
```

```
function bar() {  
  foo();  
}
```

```
function baz() {  
  bar();  
}
```

```
baz();
```

Blocks ... 개발자 도구 prints the call stack ...



This is a problem for web programming and more generally interfacing to external service which can take some time to finish and return ~

--> Need some mechanism to support Asynchronous programming and execution

**JS** console.log('Hi');

```
setTimeout(function cb() {  
  console.log('there');  
}, 5000);
```

```
console.log('JSConfEU');
```

stack

webapis

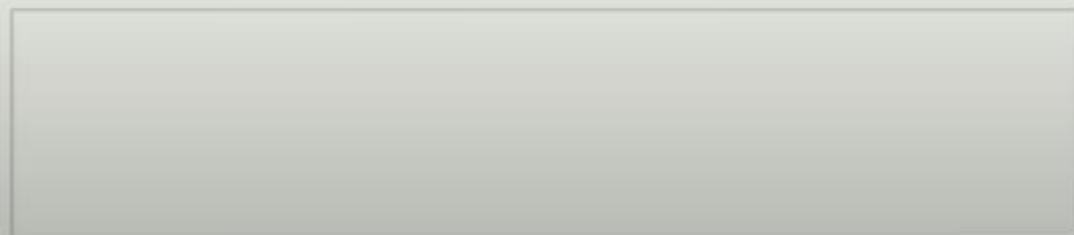
main()

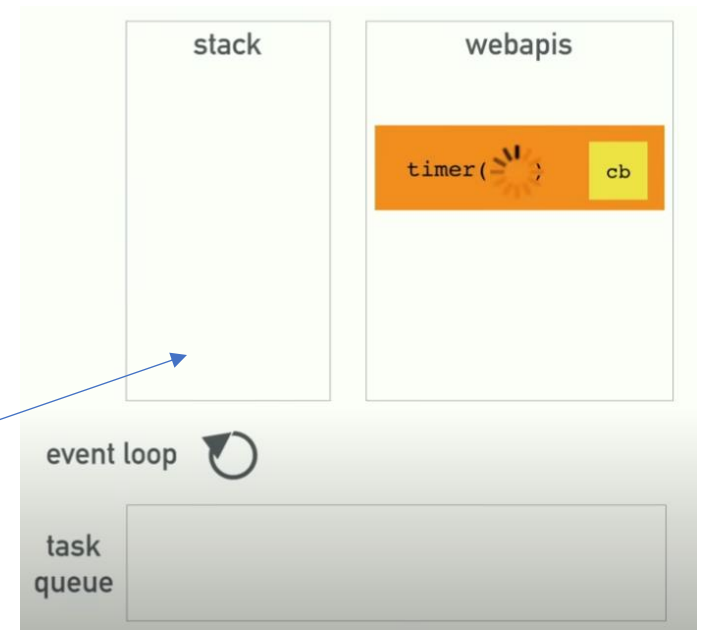
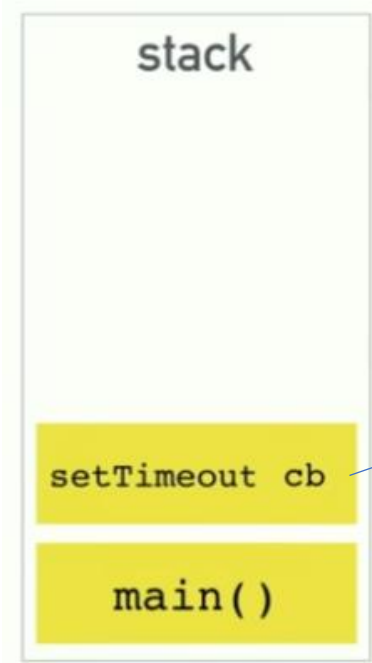
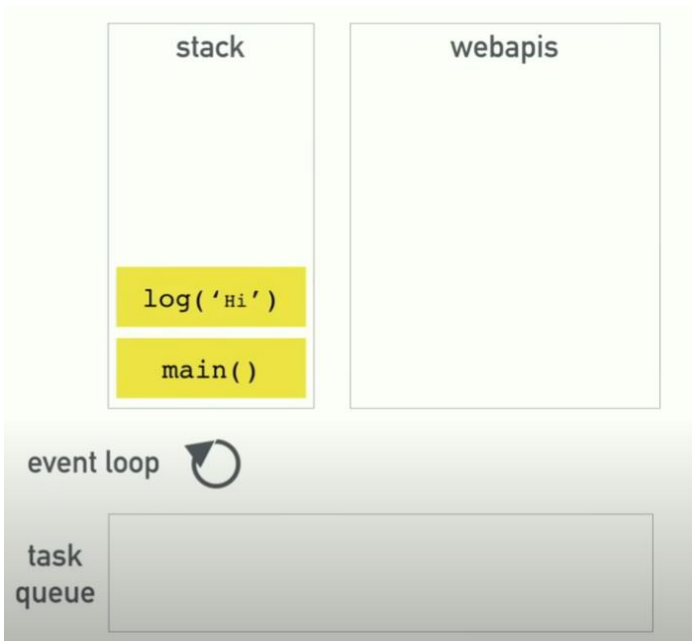
Console

event loop

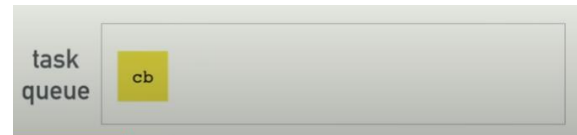
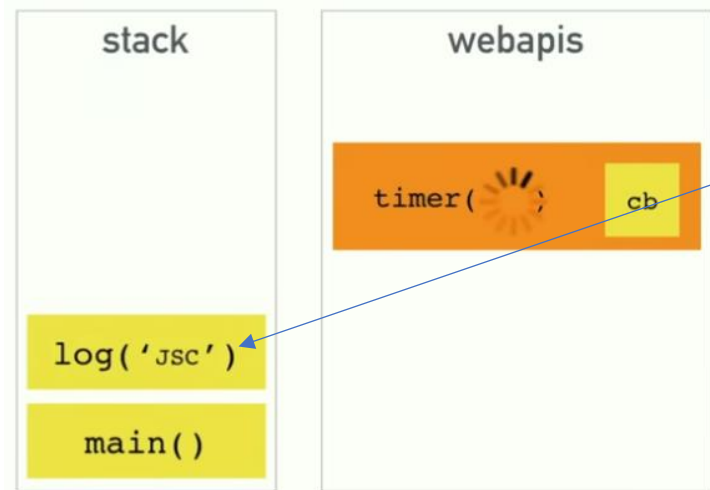


task  
queue



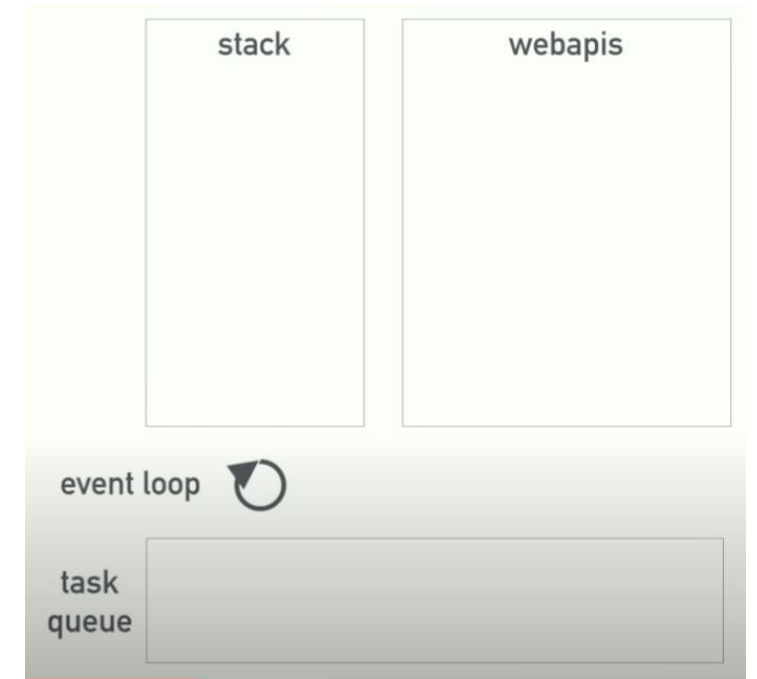
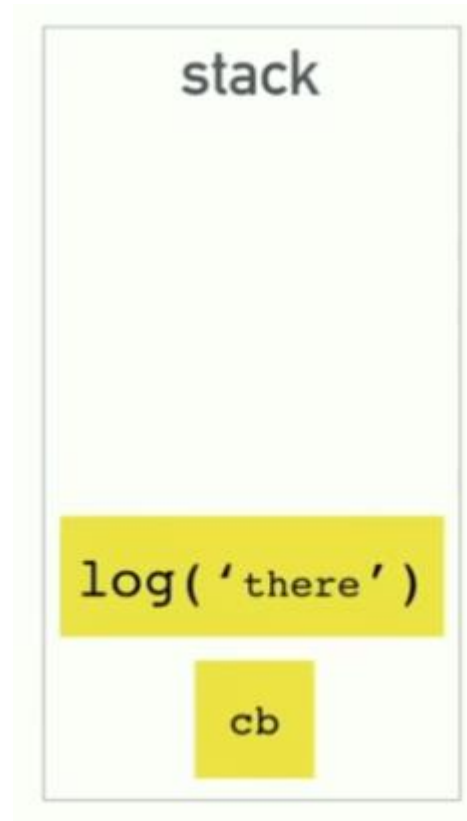
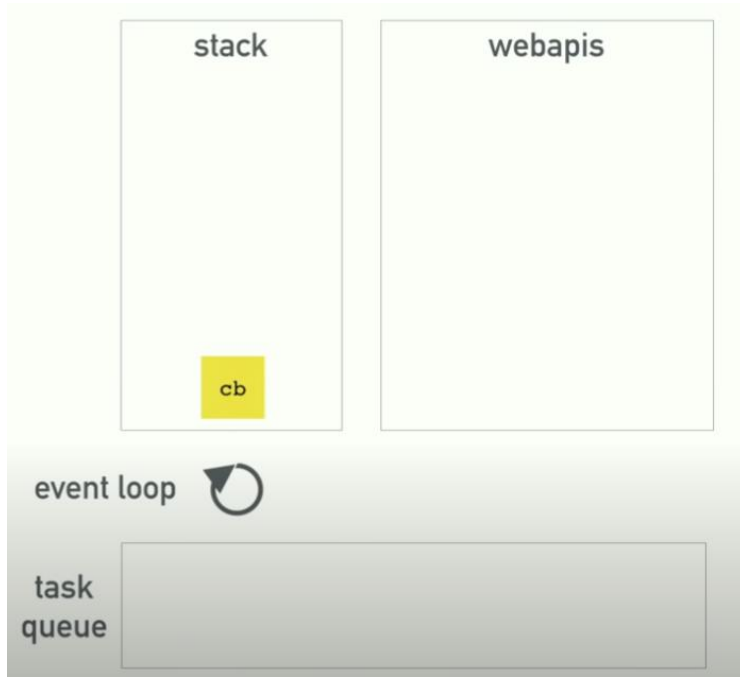


SetTimeout goes on stack but taken off ... and delegated to webapi handler  
And log ("JSC ...") goes on to call stack to get executed



When timer is done (as monitored separately by the webapi stuff, it puts the cb onto the task queue

Event loop looks at the stack and TQ: if stack is empty it will put the first thing on the TQ and push into stack



Note that TQ job will not go into call stack unless stack is empty

Asynch programming was done by hacking with SetTimeout + Call back

Other similar webapi async calls (possibly with call back) will behave the same way

**JS** console.log( 'Hi' );

Behaves like SetTimeout

```
$.get( 'url', function cb(data) {  
    console.log(data);  
});  
  
console.log( 'JSConfEU' );
```

Simulate the same way for above code

16:40 in the video

A pink square containing the white text 'JS'.

```
le.log('Started'); Edit Rerun Pause Resume  
3 $.on('button', 'click', function onClick () {  
4     console.log('Clicked');  
5 });  
6  
7 setTimeout(function onTimeout () {  
8     console.log('Timeout Finished');  
9 }, 5000);  
10  
11 console.log('Done');
```

Assume there is html that creates a button .. this sets up the onClick Callback

Call to SetTimeout  
Execute console.log after 5 seconds


JS

```
console.log('Started');
$.on('button', 'click', function onClick () {
  console.log('Clicked');
});
setTimeout(function onTimeout () {
  console.log('Timeout Finished');
}, 5000);
console.log('Done');
```

Save + Run

Call Stack

Web Apis



Callback Queue

Click Me!

Edit

First console.log onto Call stack

JS


```
console.log('Started');
$.on('button', 'click', function onClick () {
  console.log('Clicked');
});
setTimeout(function onTimeout () {
  console.log('Timeout Finished');
}, 5000);
console.log('Done');
```

Edit Rerun Pause Resume

Call Stack

Web Apis

console.log('Started')



Callback Queue

Click Me!

Edit

JS

console.log('Started');  
\$.on('button', 'click', function onClick () {  
 console.log('Clicked');  
});  
  
setTimeout(function onTimeout () {  
 console.log('Timeout Finished');  
}, 5000);  
  
console.log('Done');

EditRerunPauseResume

Call Stack

\$.on('button', 'click',  
function onClick () {  
 console.log('Clicked');  
})

Web Apis

Click Me!

Edit

Callback Queue

JS

console.log('Started');  
\$.on('button', 'click', function onClick () {  
 console.log('Clicked');  
});  
  
setTimeout(function onTimeout () {  
 console.log('Timeout Finished');  
}, 5000);  
  
console.log('Done');

EditRerunPauseResume

Call Stack

\$.on('button', 'click', ...)

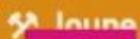
Web Apis

Click Me!

Edit

Callback Queue





```
1 console.log('Started');  
2  
3 $.on('button', 'click', function onClick () {  
4   console.log('Clicked');  
5 });  
6  
7 setTimeout(function onTimeout () {  
8   console.log('Timeout Finished');  
9 }, 5000);  
10  
11 console.log('Done');
```

Edit

Rerun

Pause

Resume

Call Stack

setTimeout(function  
onTimeout () {  
console.log('Timeout  
Finished'); }, 5000)

Web Apis

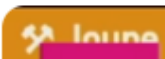
\$.on('button', 'click', ...)



Callback Queue

Click Me!

Edit



```
1 console.log('Started');  
2  
3 $.on('button', 'click', function onClick () {  
4   console.log('Clicked');  
5 });  
6  
7 setTimeout(function onTimeout () {  
8   console.log('Timeout Finished');  
9 }, 5000);  
10  
11 console.log('Done');
```

Edit

Rerun

Pause

Resume

Click Me!

Edit

Call Stack


Web Apis

\$.on('button', 'click', ...)

onTimeout()



Callback Queue




```
1 console.log('Started');
2
3 $.on('button', 'click', function onClick () {
4   console.log('Clicked');
5 });
6
7 setTimeout(function onTimeout () {
8   console.log('Timeout Finished');
9 }, 5000);
10
11 console.log('Done');
```

Click Me! Edit

Call Stack

Web Apis

console.log('Done')



Callback Queue

Not clicked  
Time out not reached yet

JS

```
1 console.log('Started');
2 $.on('button', 'click', function onClick () {
3   console.log('Clicked');
4 });
5
6
7 setTimeout(function onTimeout () {
8   console.log('Timeout Finished');
9 }, 5000);
10
11 console.log('Done');
```

Click Me!

Edit Rerun Pause Resume

Call Stack


Web Apis

\$.on('button', 'click', ...)

onTimeout ( )

Time out event -->  
Place the CB on CBQ





```
1 console.log('Started');
2
3 $.on('button', 'click', function onClick () {
4   console.log('Clicked');
5 });
6
7 setTimeout(function onTimeout () {
8   console.log('Timeout Finished');
9 }, 5000);
10
11 console.log('Done');
```


Click Me!

Edit

Call Stack


Web Apis

onTimeout()



Callback Queue

Call stack is empty  
Event loop puts CB on it



le.log('Started');

EditRerunPauseResume


5\$.on('button', 'click', function onClick () {  
6 console.log('Clicked');  
7 });  
8  
9 setTimeout(function onTimeout () {  
10 console.log('Timeout Finished');  
11 }, 5000);  
12 console.log('Done');

Click Me!Edit

Call Stack

console.log('Timeout Finished')

onTimeout()



Callback Queue

Web Apis

\$.on('button', 'click', ...)

JS

```
1 console.log('Started');
2
3 $.on('button', 'click', function onClick () {
4   console.log('Clicked');
5 });
6
7 setTimeout(function onTimeout () {
8   console.log('Timeout Finished');
9 }, 5000);
10
11 console.log('Done');
```

Call Stack

Web Apis

\$.on('button', 'click', ...)

Click event ...

Click call back put into queue

ClickMe!

Edit

Callback Queue

[click] onClick()



JS

```
1 console.log('Started');  
2  
3 $.on('button', 'click', function onClick () {  
4   console.log('Clicked');  
5 });  
6  
7 setTimeout(function onTimeout () {  
8   console.log('Timeout Finished');  
9 }, 5000);  
10  
11 console.log('Done');
```

Edit

Rerun

Pause

Resume

### Call Stack

console.log('Clicked')

[click] onClick()

### Web Apis

\$.on('button', 'click', ...)



### Callback Queue

ClickMe!

Edit



JS

console.log('Started');

Edit

Rerun

Pause

Resume

3 \$.on('button', 'click', function onClick () {

4     console.log('Clicked');

5 });

6

7 setTimeout(function onTimeout () {

8     console.log('Timeout Finished');

9 }, 5000);

10

11 console.log('Done');

Click Me!

Edit

Call Stack

Web Apis

\$.on('button', 'click', ...)

Stays for further clicks  
Same procedure repeats ...

Callback Queue

JS

```
1 console.log('Started');
2
3 $.on('button', 'click', function onClick () {
4   console.log('Clicked');
5 });
6
7 setTimeout(function onTimeout () {
8   console.log('Timeout Finished');
9 }, 5000);
10
11 console.log('Done');
```

Click Me!

Call Stack

Web Apis

\$.on('button', 'click', ...)

[click] onClick()

Callback Queue

[click] onClick()

[click] onClick()

[click] onClick()

[click] onClick()

[click] onClick()

[click] onClick()

Each click will be taken care of one by one by the same process

- Call backs that gets passed and pushed into the CBQ is a form of asynchronous function realization
- CB itself is passable function ... it is used as above
- Imagine 광 클릭 ...
  - Lots of CB pushed on the CQ
  - Each one of them gets processed one by one until call stack is clear
  - Can cause slow down anyway ... even though asynch and non blocking



More details ...

Keyboard input --> Early unrefined systems would handle it the traditional way ...

- interrupt comes in, handler reads buffer, handler does this and that including finding what key came in ...

So many different devices now ... Computers are so fast too

Keyboard is now a USB device ...

Nowadays e.g. no "keyboard" specific interrupt / Instead there's a "USB" interrupt (but in an odd way)

(and) keyboard adds the key press to its internal queue and waits for the computer to poll it

- USB devices aren't allowed to talk on the bus except in response to a request by the host
- As USB transactions are scheduled into one millisecond frames, operating system will only poll a USB keyboard once a millisecond, asking it to report any events that have occurred
- Only once the keyboard responds to this request --> USB controller generate an interrupt

The response from the keyboard will be in the form a HID (Human Interface Device) report.

The HID stack will decode it to see what keyboard events have been reported, converting them into a format common to all keyboard types. This will be further processed by some sort of user interface layer in the operating system (eg. The "Win32" API layer on Windows, or an X Server on Linux) and then put on the UI event queue for the browser.

--> Handler takes the input and relays to higher level processing

The browser will not be "interrupted" as a result of the key being pressed.

Instead the browser will have a main UI interface event loop and events will only be processed one a time at a single defined point in the program.

All this loop does is pull events from UI event queue and dispatches them to the appropriate code in the browser. When the queue is empty, the loop simply waits for an event.

Once the UI event loop gets a keyboard event, it's passed onto the browser's keyboard event handler which will then pass it to the JavaScript engine. The engine will then execute the function assigned as the key down event listener. The engine may have its own event queue.

