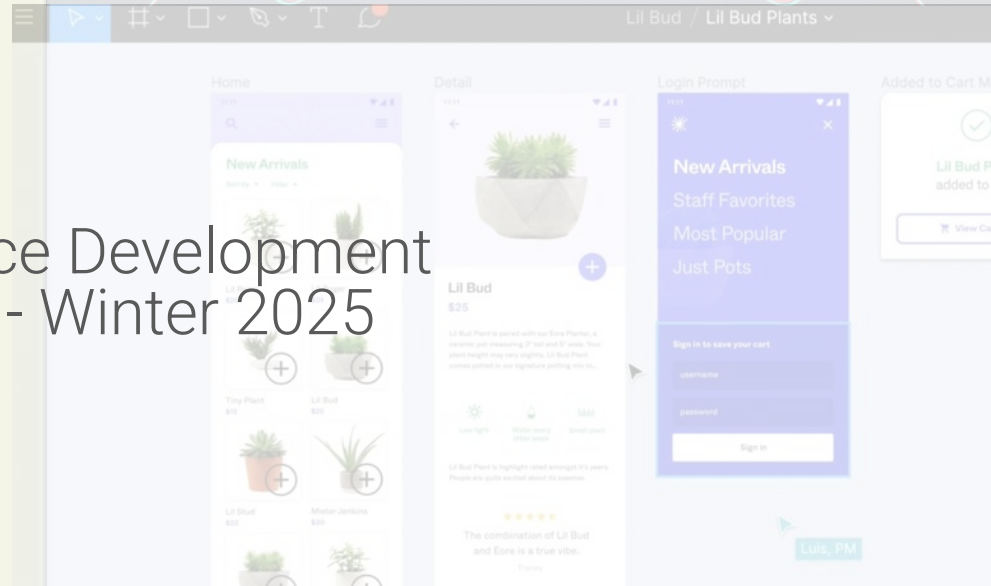


The DESIGN
of EVERY DAY
THINGS

Asynchronous JavaScript

User Interface Development
EECS 493 - Winter 2025



Questions?

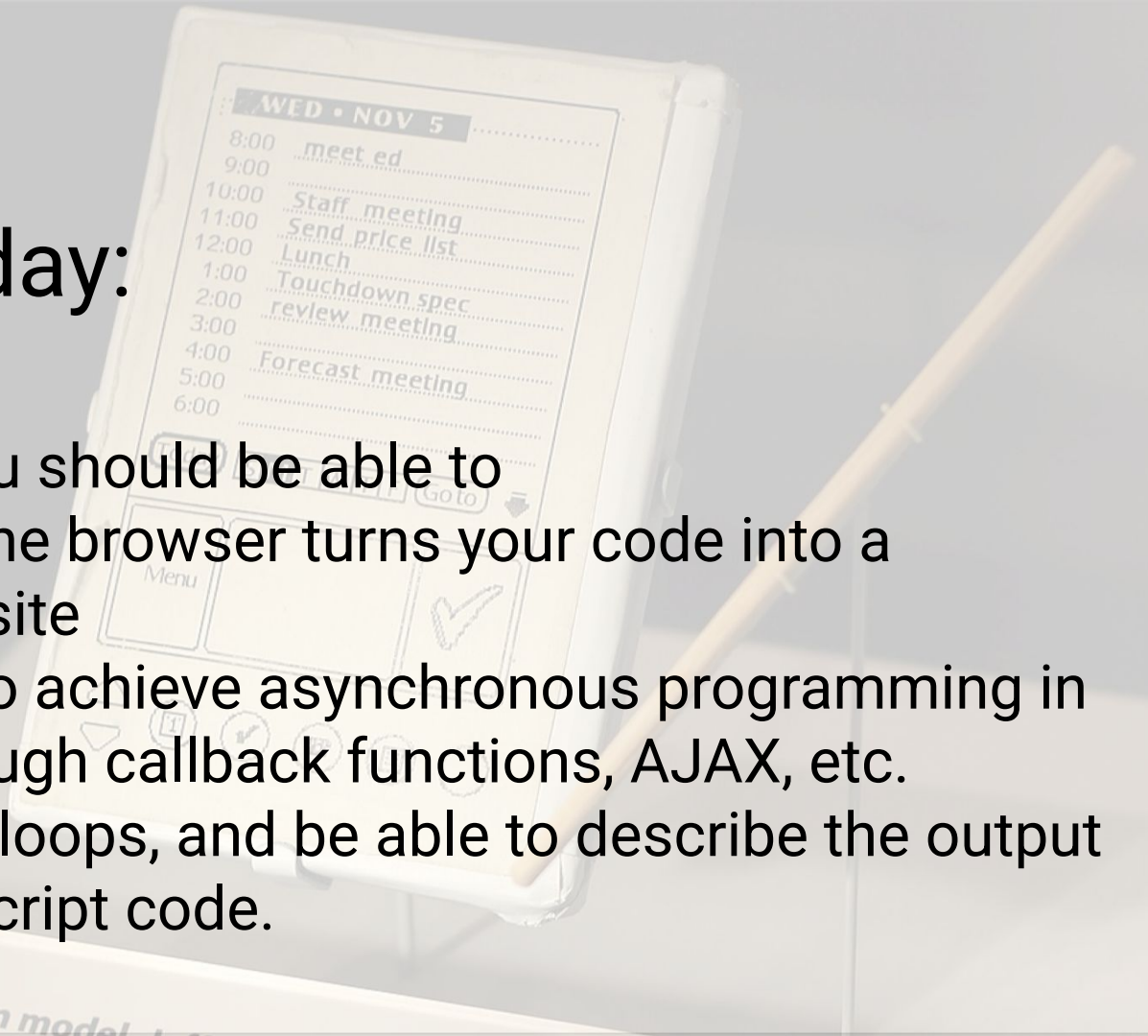


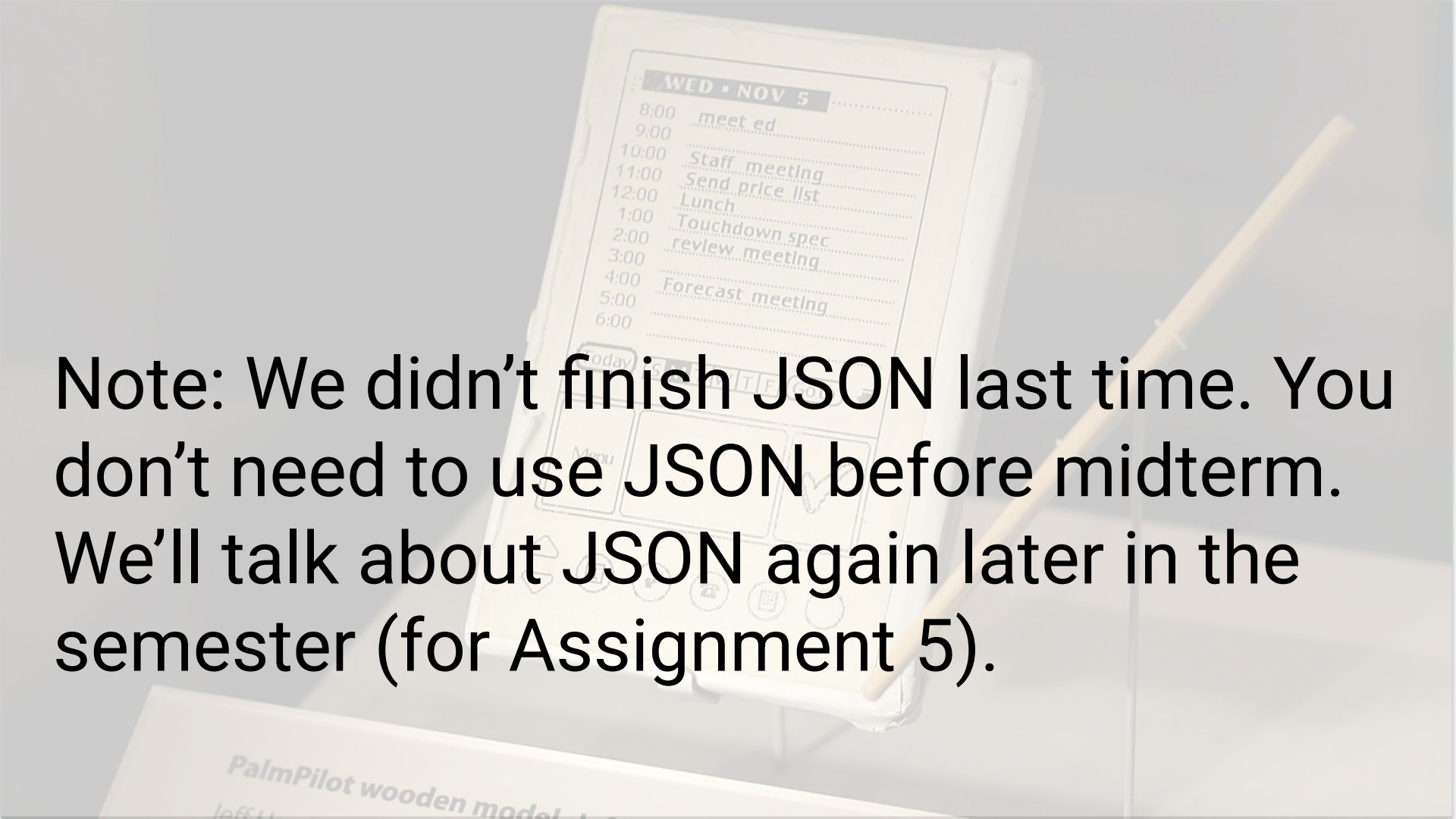
PalmPilot wooden model

Goals for today:

After this class, you should be able to

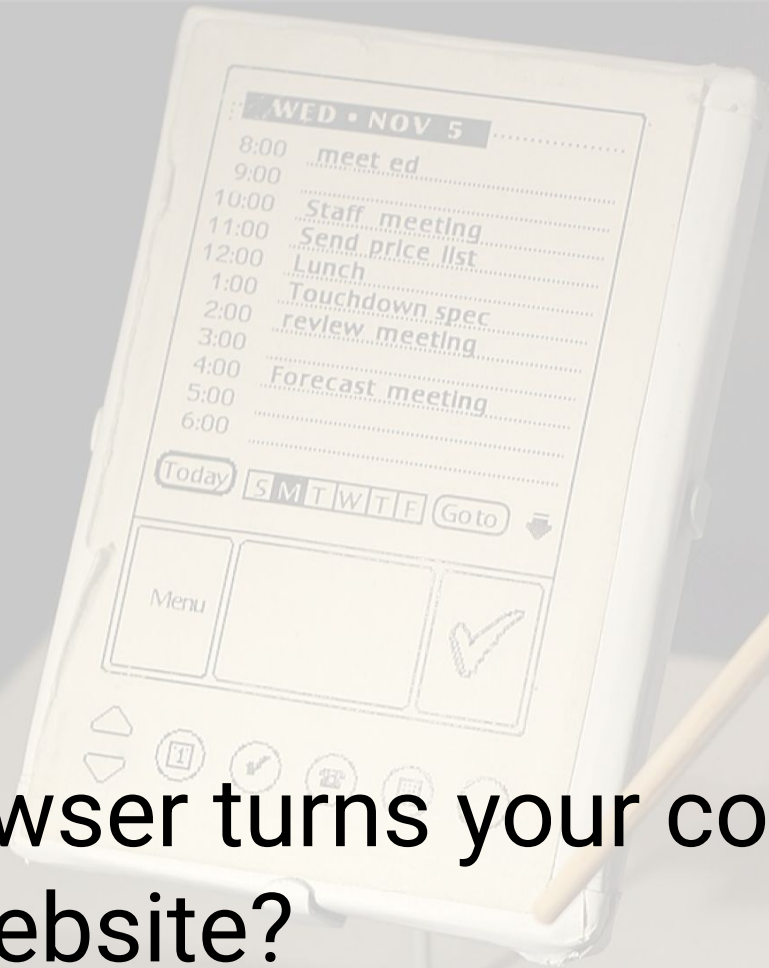
1. Describe how the browser turns your code into a functional website
2. Describe how to achieve asynchronous programming in Javascript through callback functions, AJAX, etc.
3. Describe event loops, and be able to describe the output of async Javascript code.



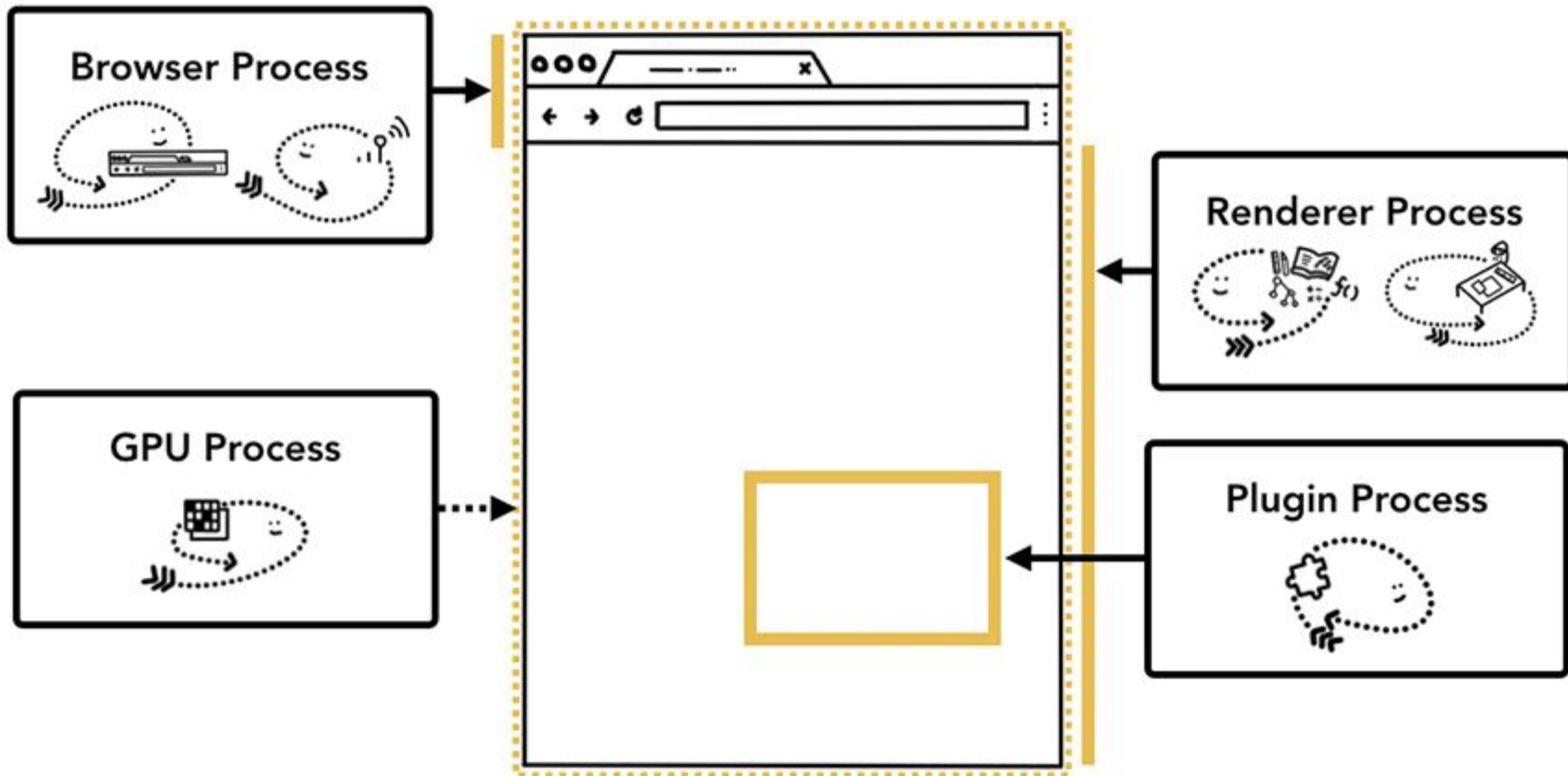
A wooden model of a PalmPilot PDA is shown, displaying a calendar for Wednesday, November 5. The calendar lists several events: 8:00 meet ed, 9:00, 10:00 Staff meeting, 11:00 Send price list, 12:00 Lunch, 1:00 Touchdown spec, 2:00 review meeting, 3:00, 4:00 Forecast meeting, 5:00, and 6:00. A wooden pencil is resting on the device. The background is a light gray.

Note: We didn't finish JSON last time. You don't need to use JSON before midterm. We'll talk about JSON again later in the semester (for Assignment 5).

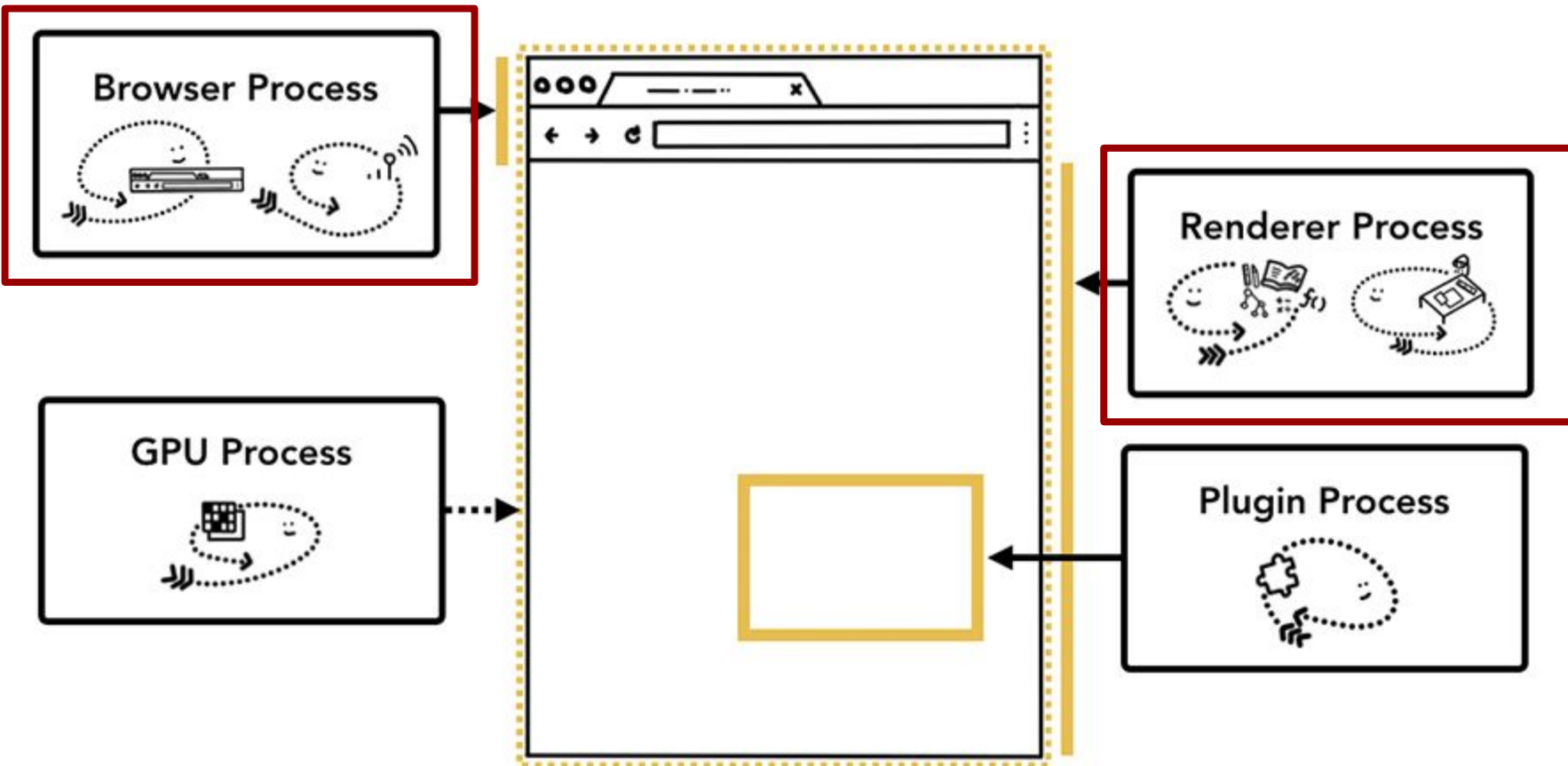
How the browser turns your code into a functional website?



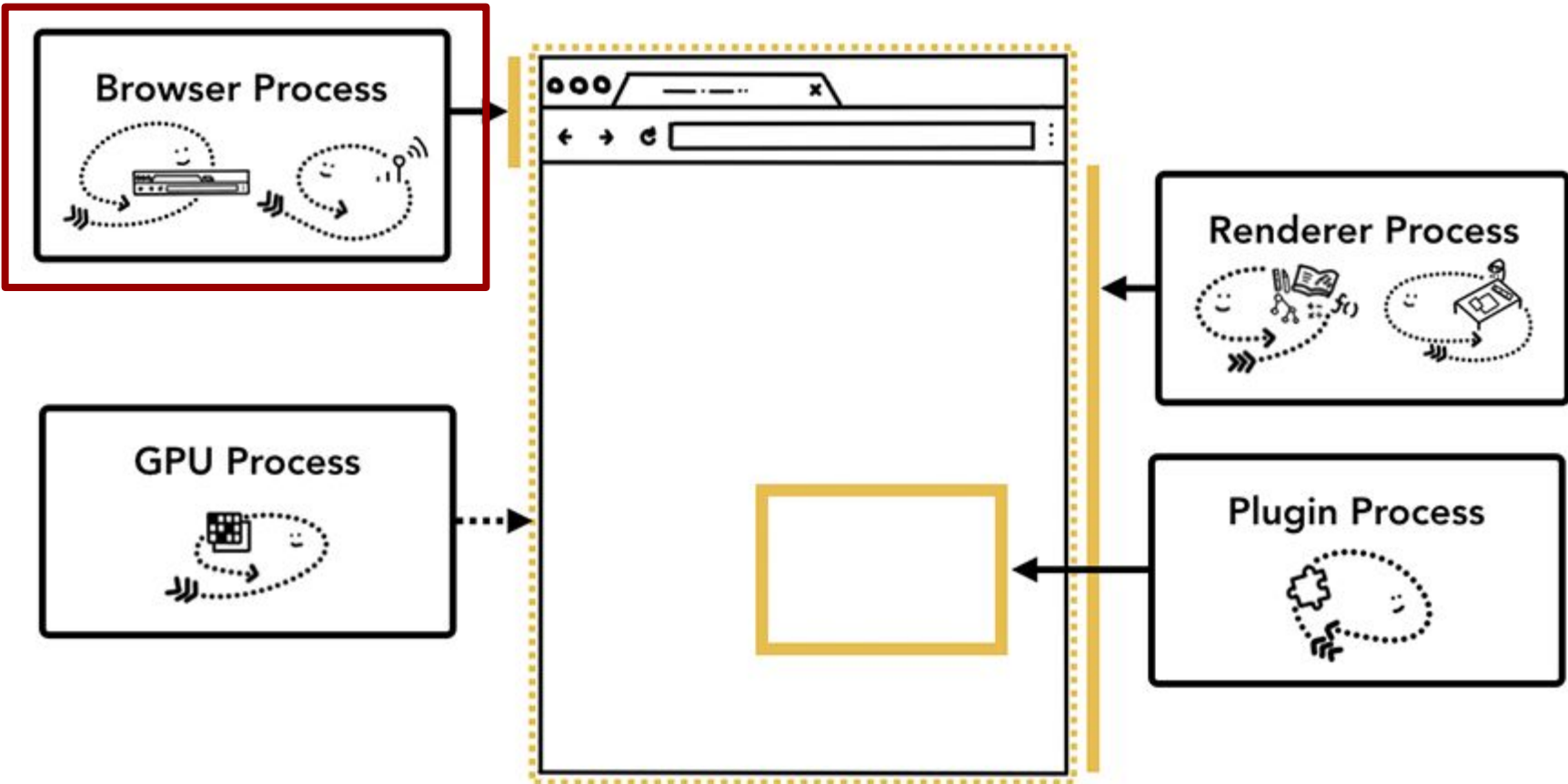
Web Browser - Chrome as an example



Web Browser - Chrome as an example



Browser Process



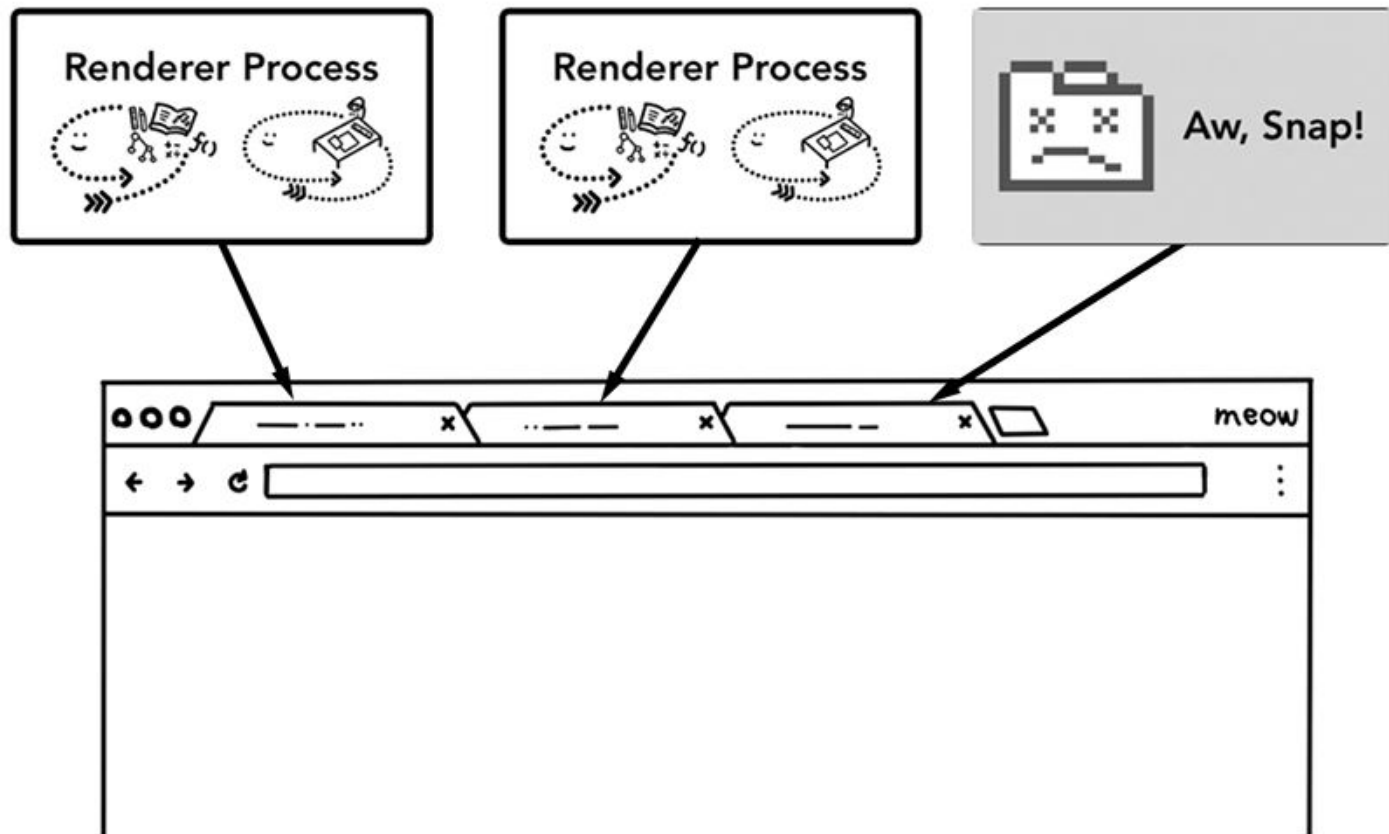
Browser Process

- Main user interface, e.g., the address bar, bookmarks, forward and backward buttons
- Creation, switching and closing of tabs
- ...

Renderer Process

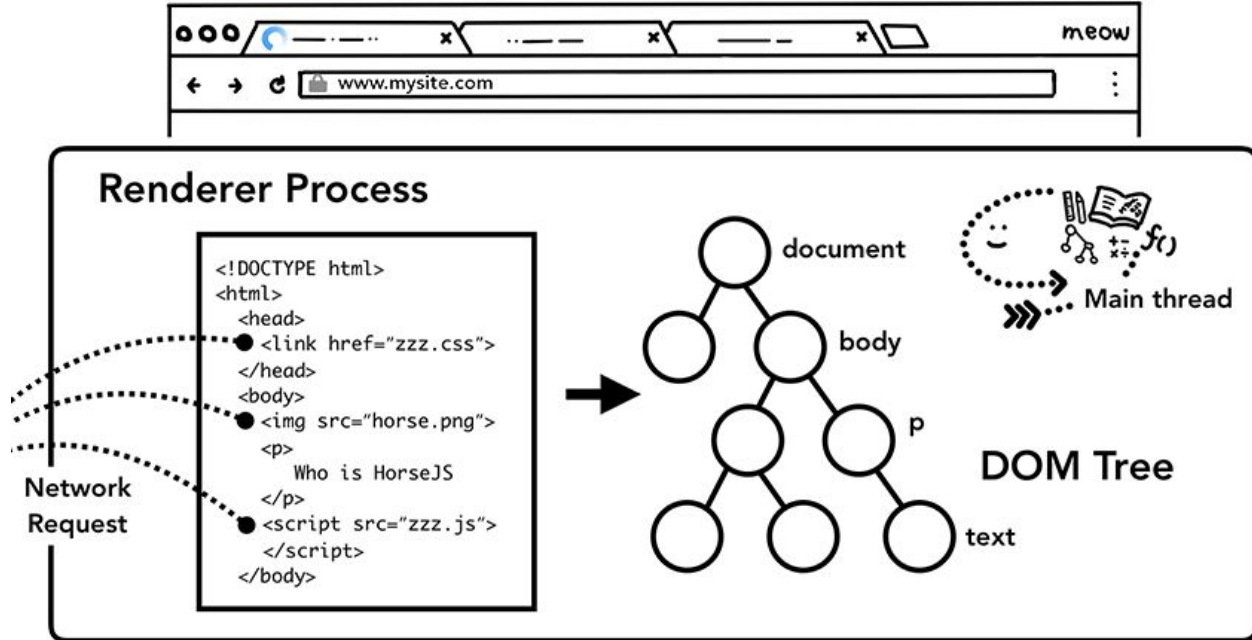
The render process is responsible for everything inside of a tab.

Renderer Process



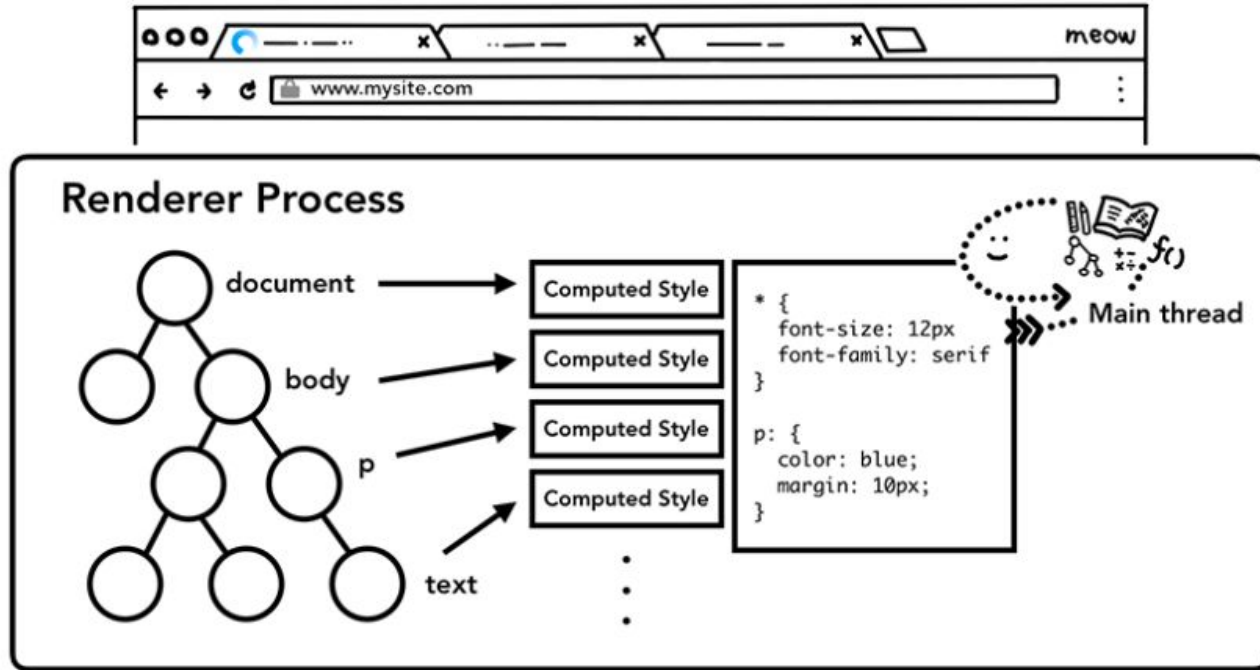
Renderer Process - Creates DOM

The renderer process parses the HTML code. When it finds a `<script>` tag, it pauses the parsing of HTML and load, parse, execute the JavaScript code first.



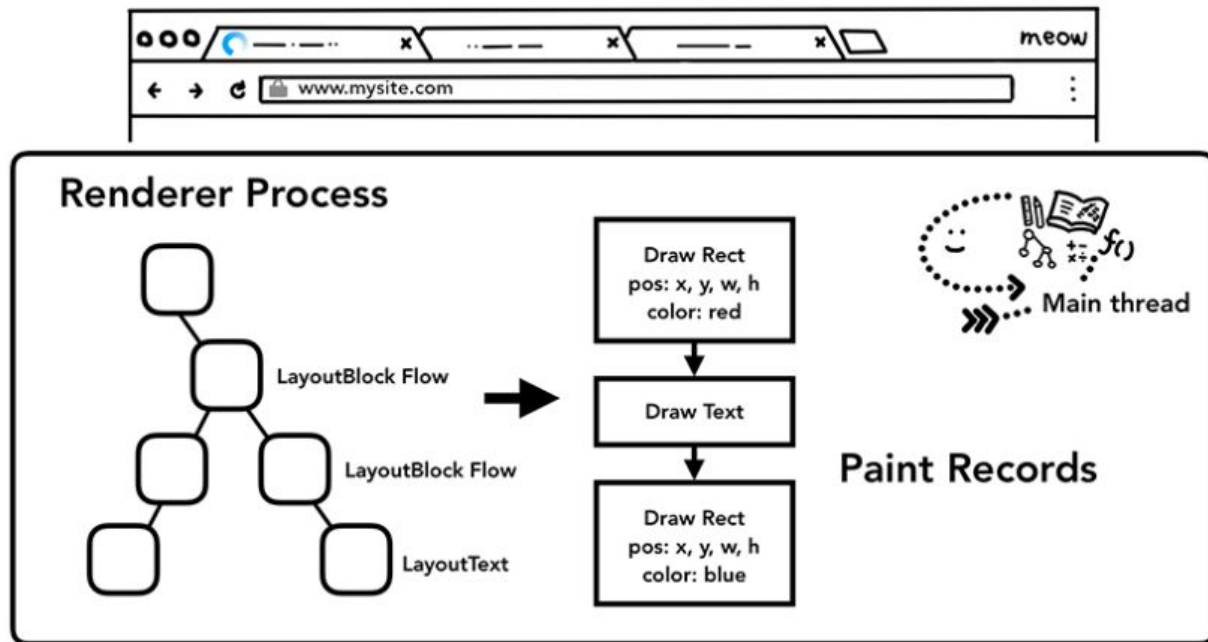
Renderer Process - Style and Layout

The render process parses the CSS code and determines the computed style and layout for each DOM node



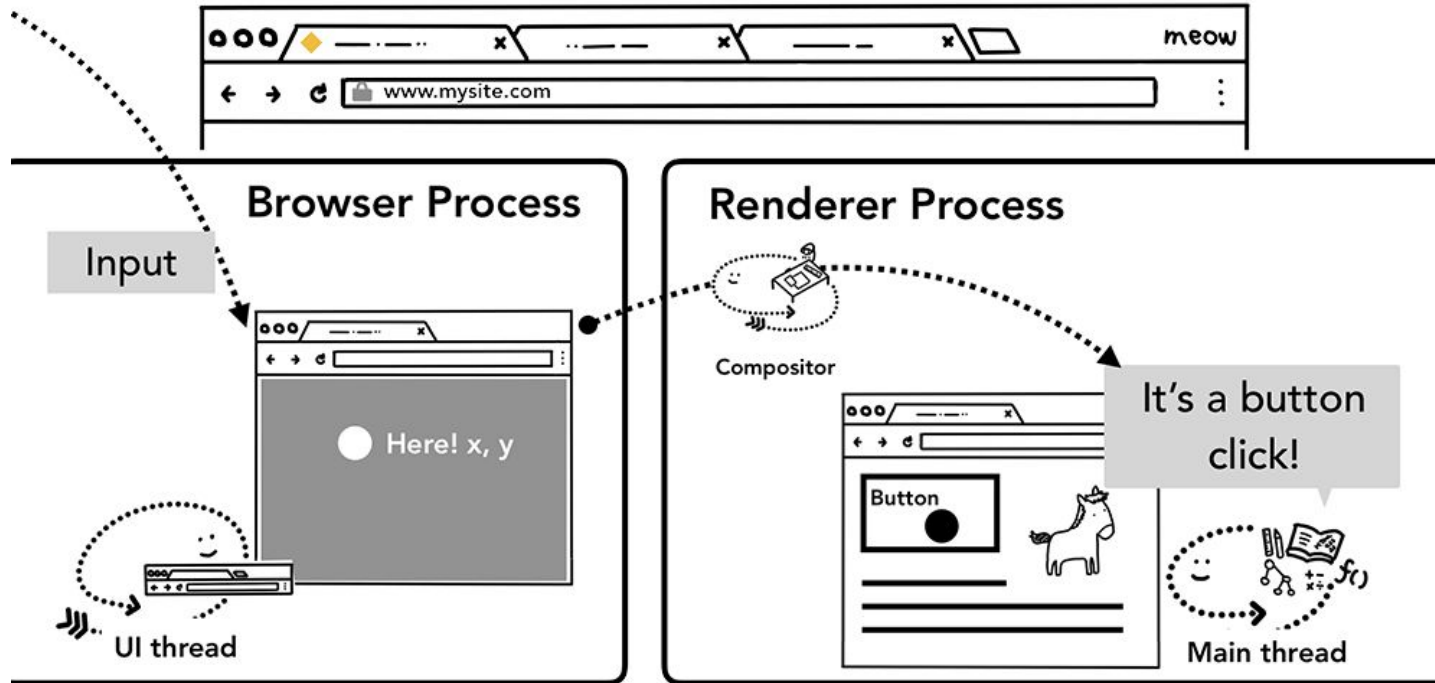
Renderer Process - Paint

The render process paints the records.



Browser Process Communicates with Renderer Process

The browser process takes user input (by position). Renderer process handles the event (through JavaScript)

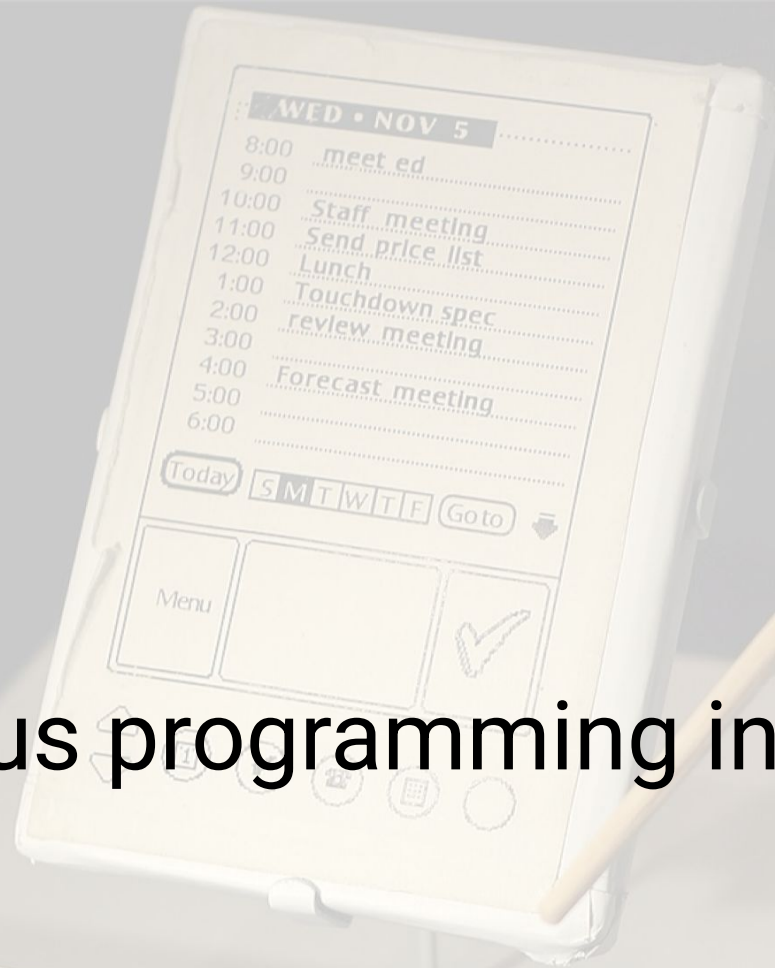


Lecture 6 - Survey 1

Which of the following are TRUE about the Renderer Process? *

- ☐ The renderer process handles JavaScript events.
- ☐ Different tabs in a browser have different renderer processes
- ☐ The renderer process captures the location of a user touch or click in the browser
- ☐ The renderer process has a single thread that parses HTML, and calculates style and layout.

Asynchronous programming in JavaScript



PalmPilot wooden model
Jeff U.

JavaScript Execution

- JavaScript is single threaded
 - Run by the JavaScript runtime engine (V8) in the browser
- Ways to run JavaScript asynchronously:
 - **setTimeout()**
 - **Click events (event listeners)**
 - ajax()
 - Promise

setTimeout()

- A function in JavaScript to execute a piece of code after a specified delay.
- `setTimeout(function(){}, delay)`
- Returns an ID, which can be used to cancel the timeout.
- `timeoutID = setTimeout(function(){}, delay)`
- `clearTimeout(timeoutID)`

Live coding example

example.html

Callback functions

| | | | |
|----------------------|---|---|---|
| <input type="text"/> | | | C |
| 1 | 2 | 3 | + |
| 4 | 5 | 6 | - |
| 7 | 8 | 9 | / |
| * | 0 | . | = |

Code:

```
<button onclick="handleClick()">
```

or

```
document.querySelector("myBtn").onclick  
= function(event) { ... }
```

or

```
myBtn.addEventListener("click",  
function(event));
```

AJAX (Asynchronous JavaScript and XML)

- Allows a webpage to communicate with a server and updates parts of a web page without reloading the whole page.
- An event is triggered (e.g., a button is clicked)
- JavaScript creates an XMLHttpRequest object and sends a request to the web server.
- The server processes the request and sends a response back to the web page.
- The JavaScript on the web page reads the response and updates the page.
- **When the server is processing the request, the browser can do other stuff async.**

Live coding example

`ajaxexample.html`

```
<div id="demo">
  <h2>Let AJAX change this text</h2>
  <button type="button" onclick="loadDoc()">Change Content</button>
</div>
```

```
<script>
  // Your JavaScript code here
  function loadDoc(){
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange=function(){
      if (this.readyState == 4 && this.status == 200){
        document.getElementById("demo").innerHTML = this.responseText;
      }
    };
    // var dataUri = "data:text/plain;base64," + btoa("This is text displayed");
    // xhttp.open("GET", dataUri, true);
    xhttp.open("GET", "ajax_info.txt", true);
    xhttp.send();
  }
}
```

Promise

- A powerful way to complete asynchronous tasks in JavaScript
- There are three states for a Promise
 - Pending: initial state
 - Fulfilled: when the operation was completed successfully
 - Rejected: when the operation failed

Live coding example

promise.html

```
19     let promise = new Promise(function(  
20         resolve, reject) {  
21             // Do an async task and then...  
22             setTimeout(()=>  
23                 {if(true){  
24                     resolve("Stuff worked!");  
25                 } |  
26                 else{  
27                     reject(Error("It broke"));  
28                 }  
29             }, 2000)  
30         });  
31  
32     // Use the promise  
33     promise.then(function(result) {  
34         console.log(result); // "Stuff  
35         worked!"  
36     }, function(err) {  
37         console.log(err); // Error: "It  
38         broke"  
39     });
```

Promise Chaining

- Promise.then returns a promise

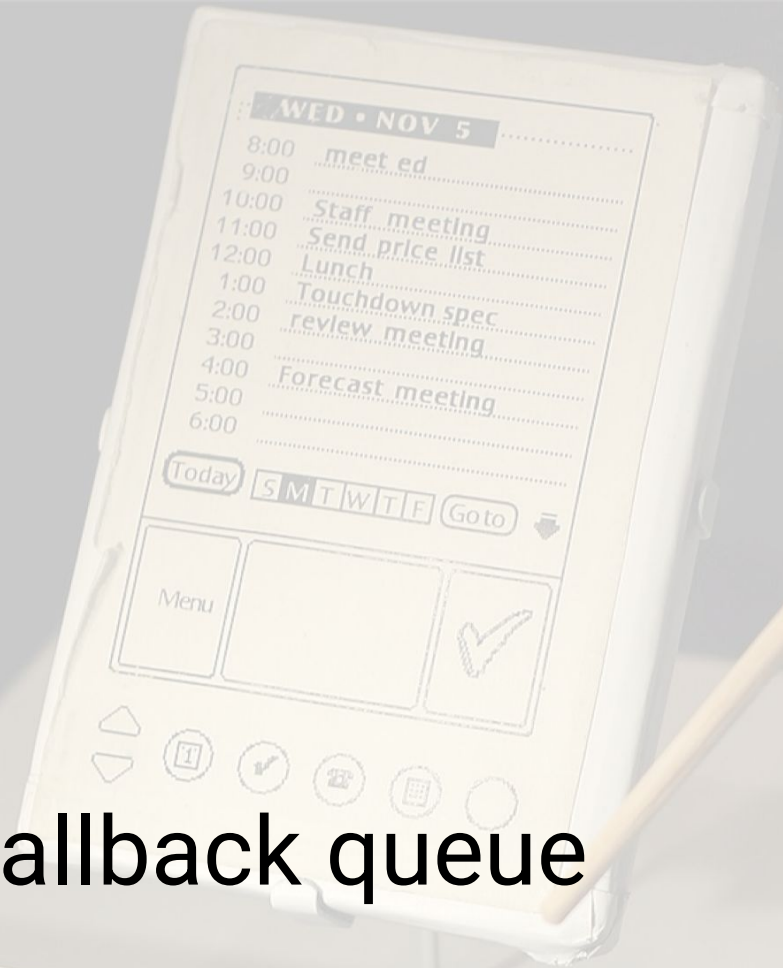
promisechaining.html

```
<script>
  new Promise(function(resolve, reject){
    setTimeout(()=> resolve(1), 1000);
  }).then(function(result){
    alert(result);
    return result*2;
  }).then(function(result){
    alert(result);
    return result*2;
  }).then(function(result){
    alert(result);
    return result*2;
  })
})
```

Promise Chaining - recap of arrow functions

```
<script>
  new Promise(function(resolve, reject){
    setTimeout(()=> resolve(1), 1000);
  }).then(function(result){
    alert(result);
    return result*2;
  }).then(function(result){
    alert(result);
    return result*2;
  }).then(function(result){
    alert(result);
    return result*2;
  })
}
```

```
new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve(1);
  }, 1000);
})
```

JavaScript callback queue

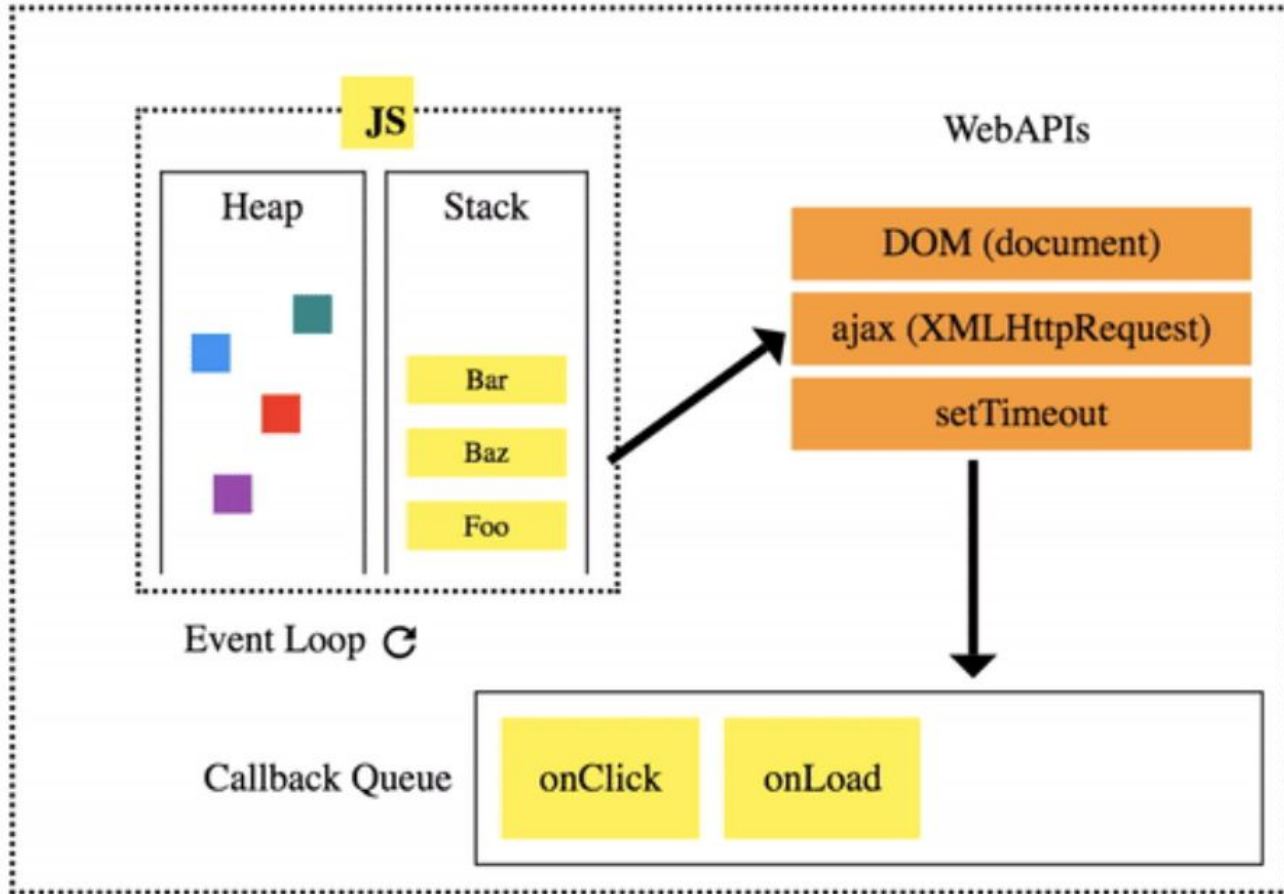
PalmPilot wooden model
Jeff U.

Predict the output

```
console.log("statement1");  
setTimeout(function timeout(){  
    console.log("statement2");  
}, 0);  
console.log("statement3");
```

Show it in console

How JavaScript does async programming?



Visualize in Loupe

```
console.log("statement1");  
setTimeout(function timeout(){  
    console.log("statement2");  
}, 0);  
console.log("statement3");
```

<http://latentflip.com/loupe/>

```
11 $("#clickme").click(function onClick() {  
12     setTimeout(function timer() {  
13         console.log('You clicked the button!');  
14     }, 0);  
15 });  
16  
17 console.log("Hi!");  
18  
19 setTimeout(function timeout() {  
20     console.log("Click the button!");  
21 }, 10000);  
22  
23 console.log("Welcome to loupe.");
```

Console

- Hi
- Welcome to Loupe

Callback Queue

WebAPIs

- timer of 10 seconds for timeout to enter the callback queue.

```
11 $("#clickme").click(function onClick() {  
12     setTimeout(function timer() {  
13         console.log('You clicked the button!');  
14     }, 0);  
15 });  
16  
17 console.log("Hi!");  
18  
19 setTimeout(function timeout() {  
20     console.log("Click the button!");  
21 }, 10000);  
22  
23 console.log("Welcome to loupe.");
```

Console

- Hi
- Welcome to Loupe

Callback Queue

- onClick()

WebAPIs

- timer of 10 seconds for timeout to enter the callback queue.
- timer of 0 seconds for timer() to enter the callback queue.

```
11 $("#clickme").click(function onClick() {  
12     setTimeout(function timer() {  
13         console.log('You clicked the button!');  
14     }, 0);  
15 });  
16  
17 console.log("Hi!");  
18  
19 setTimeout(function timeout() {  
20     console.log("Click the button!");  
21 }, 10000);  
22  
23 console.log("Welcome to loupe.");
```

Console

- Hi
- Welcome to Loupe
- You clicked the button!

Callback Queue

- `onClick()`
- `timer()`

WebAPIs

- timer of 10 seconds for timeout to enter the callback queue.

```
11 $("#clickme").click(function onClick() {  
12     setTimeout(function timer() {  
13         console.log('You clicked the button!');  
14     }, 0);  
15 });  
16  
17 console.log("Hi!");  
18  
19 setTimeout(function timeout() {  
20     console.log("Click the button!");  
21 }, 10000);  
22  
23 console.log("Welcome to loupe.");
```

Console

- Hi
- Welcome to Loupe
- You clicked the button!

Callback Queue

- `onClick()`
- `timer()`
- `timeout()`

WebAPIs


```
11 $("#clickme").click(function onClick() {  
12     setTimeout(function timer() {  
13         console.log('You clicked the button!');  
14     }, 0);  
15 });  
16  
17 console.log("Hi!");  
18  
19 setTimeout(function timeout() {  
20     console.log("Click the button!");  
21 }, 10000);  
22  
23 console.log("Welcome to loupe.");
```

Console

- Hi
- Welcome to Loupe
- You clicked the button!
- Click the button!

Callback Queue

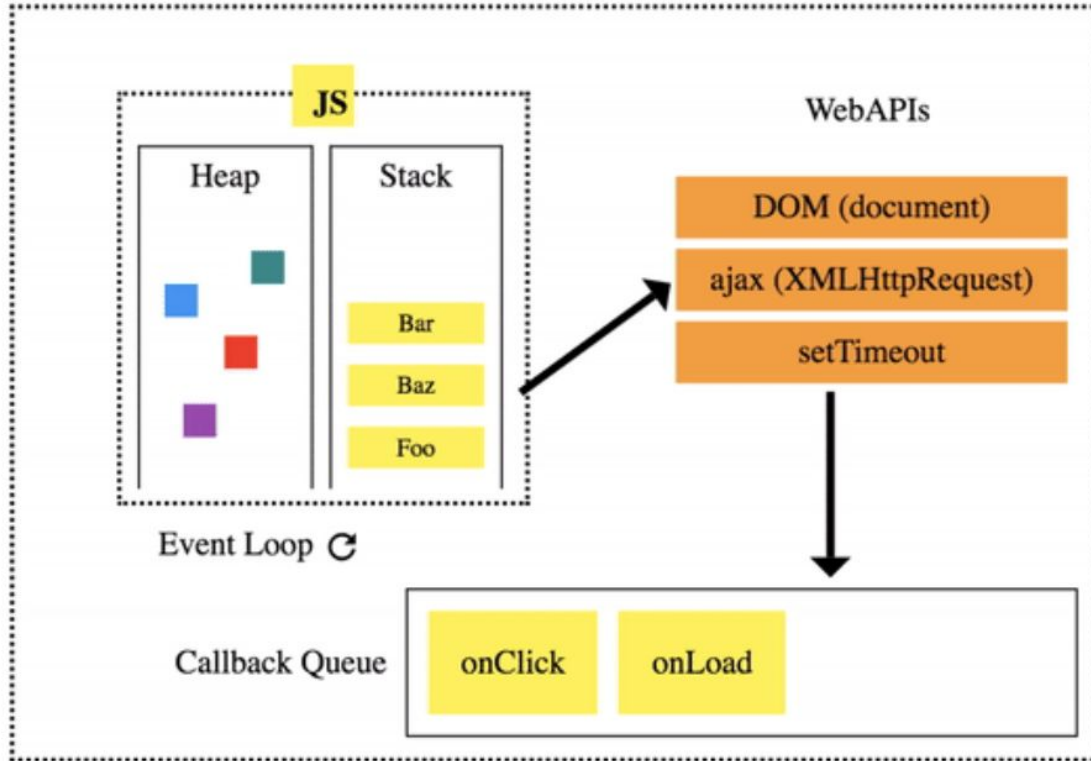
- `onClick()`
- `timer()`
- `timeout()`

WebAPIs

Visualize in Loupe

<http://latentflip.com/loupe/>

Promises -> additional job queue



Additional job Queue
for Promises: Event
Loop prioritize
Promises over
Callback Queues

What is the order of output?

```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
  .then(res => console.log(res))

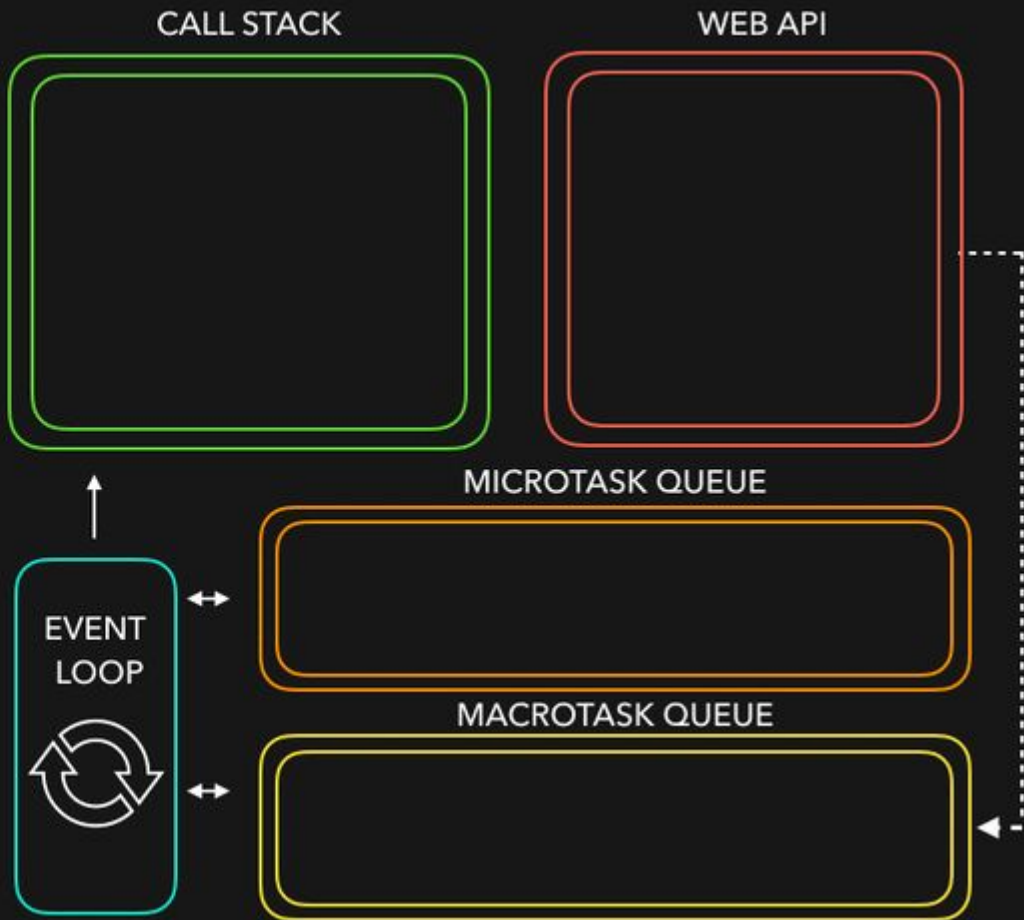
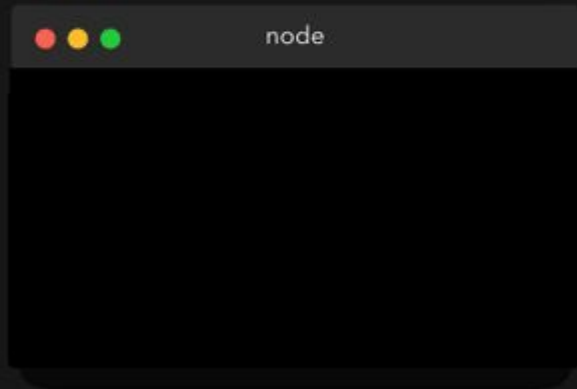
console.log('End!')
```

```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
  .then(res => console.log(res))

console.log('End!')
```



```
console.log('Start!')
```

```
setTimeout(() => {  
  console.log('Timeout!')  
}, 0)
```

```
Promise.resolve('Promise!')  
  .then(res => console.log(res))
```

```
console.log('End!')
```



node

Start

CALL STACK

WEB API

MICROTASK QUEUE

MACROTASK QUEUE

EVENT
LOOP



```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
  .then(res => console.log(res))

console.log('End!')
```



node

Start

CALL STACK

WEB API



```
() => {
  console.log('Timeout')
}
```

MICROTASK QUEUE

EVENT
LOOP



MACROTASK QUEUE

```
console.log('Start!')
```

```
setTimeout(() => {  
  console.log('Timeout!')  
}, 0)
```

```
Promise.resolve('Promise!')  
  .then(res => console.log(res))
```

```
console.log('End!')
```



node

Start!

CALL STACK

WEB API

MICROTASK QUEUE

MACROTASK QUEUE

EVENT
LOOP



```
res => console.log(res)
```

```
() => {  
  console.log('Timeout')  
}
```




```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
  .then(res => console.log(res))

console.log('End!')
```



node

Start!

End!

CALL STACK

WEB API

MICROTASK QUEUE

MACROTASK QUEUE

EVENT
LOOP



`res => console.log(res)`

```
() => {
  console.log('Timeout')
}
```

```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
  .then(res => console.log(res))

console.log('End!')
```



node

```
Start!
End!
Promise!
```

CALL STACK

WEB API

MICROTASK QUEUE

MACROTASK QUEUE

EVENT
LOOP

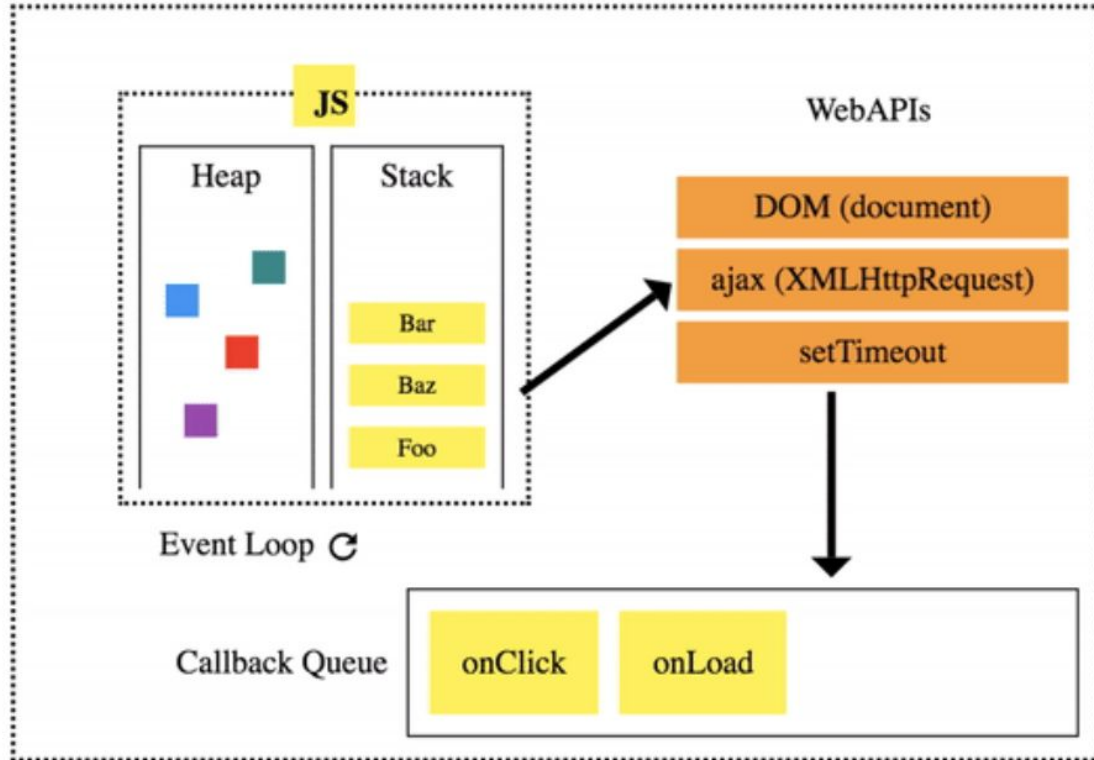


```
() => {
  console.log('Timeout')
}
```



```
1 console.log('Message no. 1: Sync');
2 setTimeout(function() {
3     console.log('Message no. 2: setTimeout');
4 }, 0);
5 var promise = new Promise(function(resolve, reject) {
6     console.log("creating promise")
7     resolve();
8 });
9 promise.then(function(resolve) {
10     console.log('Message no. 3: 1st Promise');
11 })
12 .then(function(resolve) {
13     console.log('Message no. 4: 2nd Promise');
14 });
15
16 var promise1 = new Promise(function(resolve, reject){
17     resolve();
18 });
19 promise1.then(function(resolve){
20     console.log("Message no. 6")
21 })
22 console.log('Message no. 5: Sync');
```

JavaScript Event Loop



JavaScript Event Loop

- Event Loop is constantly running to check the stack and the callback queues. When the stack is empty, it pushes functions from callback queues to stack.
- `setTimeout()` - 10 seconds
 - At least there's a wait of 10 seconds, possibly it'll be executed in more than 10 seconds

Don't block the event loop

- The browser can't do anything when there's stuff in the stack.

```
1 //sync
2 [1,2,3,4].forEach(function(i)){
3     console.log(i);
4 }
5
6
7 //async
8 function asyncForEach(array,cb){
9     array.forEach(function(){
10         setTimeout(cb, 0);
11     })
12 }
13
14 asyncForEach([1,2,3,4], function(i){
15     console.log(i);
16 })
```

Assignment 3 - setInterval()

- It is similar to setTimeout(), but it repeatedly adds a function to the callback queue
- Think about having a spawn function inside each asteroid that's moving it.
- With the callback functions that are moving the asteroids, you can have asteroids that move “at the same time”
- For checking collision, think about how you could have a function inside each asteroid to check its collision with others. Almost like each asteroid is asking themselves this question “am I colliding with another asteroid”?

Goals for today:

After this class, you should be able to

1. Describe how the browser turns your code into a functional website
2. Describe how to achieve asynchronous programming in Javascript through callback functions, AJAX, etc.
3. Describe event loops, and be able to describe the output of async Javascript code.

