Internet Technologies

Event-driven programming in JavaScript

University of Cyprus
Department of Computer Science

Event-driven programming

- Most JavaScript written in the browser is **event-driven**: The code doesn't run right away, but it executes after some event fires.

Example:
Here is a UI element that the user can interact with.

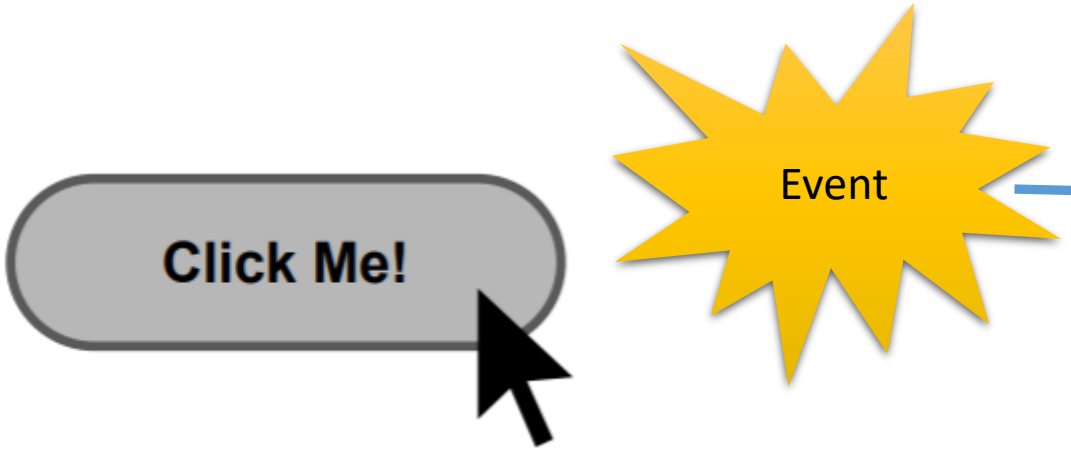
Event-driven programming

- Most JavaScript written in the browser is **event-driven**: The code doesn't run right away, but it executes after some event fires.

When the user clicks the button the button emits an "event," which is like an announcement that some interesting thing has occurred!

Event-driven programming

- Most JavaScript written in the browser is **event-driven**: The code doesn't run right away, but it executes after some event fires.




```
function onClick() {
    ...
}
```
JS

Any function listening to that event now executes. This function is called an "event handler."

Using event listeners

- Let's print "Clicked" to the Web Console when the user clicks the given button:

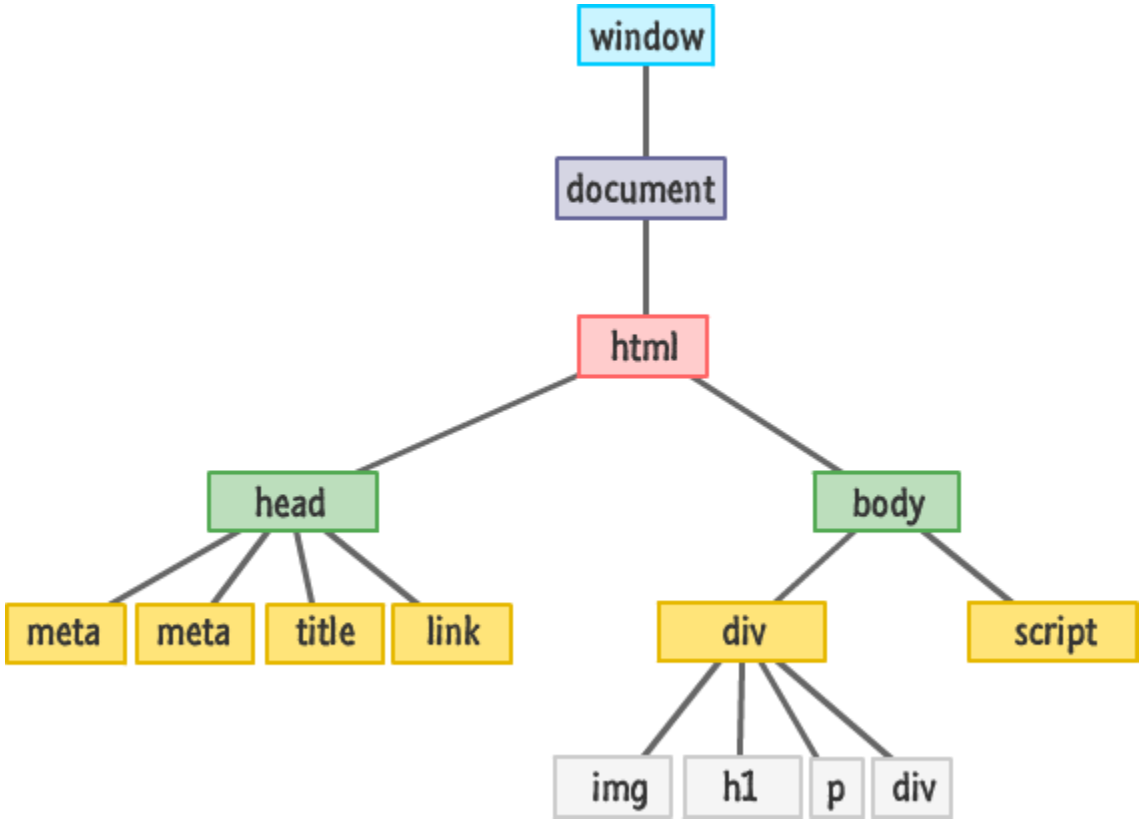
```html
<button>Click me</button>
```
HTML

Click me

- We need to add an event listener to the button...
- How do we talk to an element in HTML from JavaScript?

The DOM

- The DOM (**Document Object Model**) is a tree of node objects corresponding to the HTML elements on a page.



The DOM

- The DOM (**Document Object Model**) is a tree of node objects corresponding to the HTML elements on a page.
  - JS code can **examine** these nodes to see the state of an element
    - (e.g. to get what the user typed in a text box)
  - JS code can **edit** the attributes of these nodes to change the attributes of an element
    - (e.g. to toggle a style or to change the contents of a tag)
  - JS code can **add** elements to and remove elements from a web page by adding and removing nodes from the DOM

Getting DOM objects

- We can access an HTML element's corresponding DOM object in JavaScript via the querySelector() function:

```JS
document.querySelector('css selector');
```

  - Returns the **first** element that matches the given CSS selector.

- And via the querySelectorAll() function:

```JS
document.querySelectorAll('css selector');
```

  - Returns **all** elements that match the given CSS selector.

- .getElementById() and .getElementsByClassName() are also available but are very specific to ids and classes respectively

Getting DOM objects

JS

```
// Returns the DOM object for the HTML element
// with id="button", or null if none exists. **
let element = document.querySelector('#button');

// Returns a list of DOM objects containing all
// elements that have a "quote" class AND all
// elements that have a "comment" class.
let elementList = document.querySelectorAll('.quote, .comment');
```

** `let element = document.getElementById('button');` does the same job but works only for ids.

Adding event listeners

- Each DOM object has the following method defined:

Called using: object.addEventListener()

```
.addEventListener(event name, function name);
```

  - **event name** is the string name of the JavaScript event you want to listen to -- common ones: `click`, `focus`, `blur`, `change`, `mouseover`, `keyup`, etc
  - **function name** is the name of the JavaScript function (handler) you want to execute when the event fires
    - you can have multiple handlers for a single event

Removing event listeners

- To stop listening to an event, use removeEventListener on object:

Called using: object.removeEventListener()

```
.removeEventListener(event name, function name);
```

  - event name is the string name of the JavaScript event to stop listening to
  - function name is the name of the JavaScript function you no longer want to execute when the event fires

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js"></script>
  </head>
  <body>
      <button>Click Me!</button>
  </body>
</html>
```

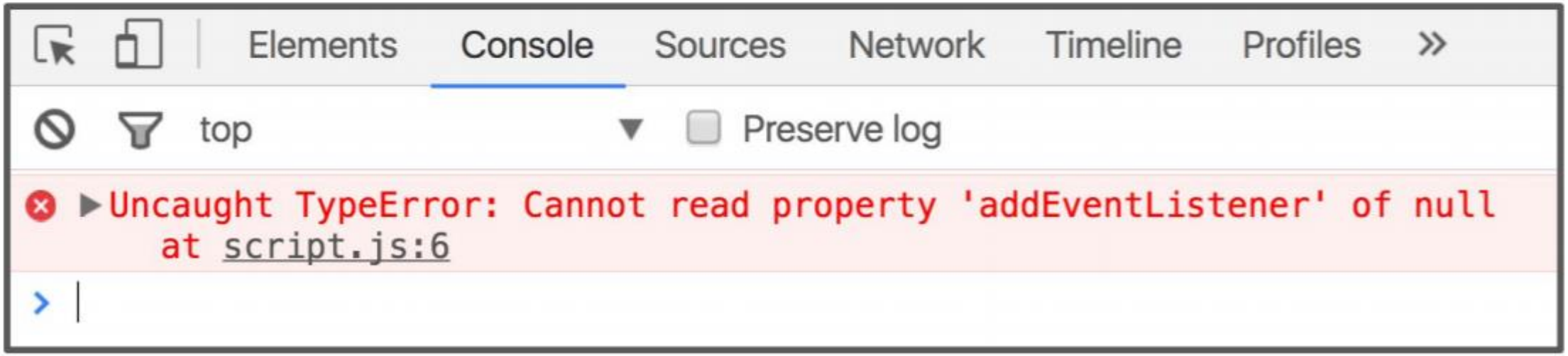
script.js

```
function onClick() {
    console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

Why error??


script.js
1 function onClick() {
2 console.log('clicked');
3 }
4
5 const button = document.querySelector('button');
6 button.addEventListener('click', onClick);
7


Elements
Console
Sources
Network
Timeline
Profiles
top
Preserve log
Uncaught TypeError: Cannot read property 'addEventListener' of null
at script.js:6

Why error??

- Whenever the HTML parser finds the `<script>` tag, a request is made to fetch the script, then it is executed
- The `<script>` tag is at the top of the document so the rest of the HTML document is not loaded yet
- So the `<button>` element is not available when the script is executed
- Therefore querySelector returns null, and `addEventListener` can't be called on `null`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
   <meta charset="utf-8">
   <title>First JS Example</title>
   <script src="script.js"></script>
  </head>
  <body>
     <button>Click Me!</button>
  </body>
</html>
```

script.js

```
1  function onClick() {
2    console.log('clicked');
3  }
4
5  const button = document.querySelector('button');
6  button.addEventListener('click', onClick);
7
```

Use `defer`

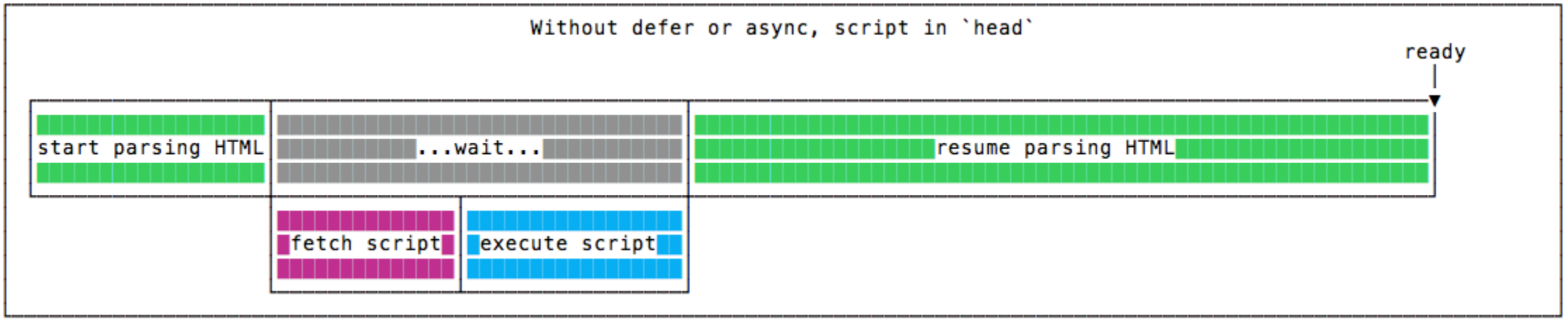
- You can add the defer attribute onto the script tag so that the JavaScript doesn't execute until after the DOM is loaded (mdn):

```
<script src="script.js" defer></script>
```

- Other old ways of doing this (not recommended though):
  - Put the `<script>` tag at the bottom of the page – good for old browsers, which do not support defer
  - Listen for the "load" event on the window or document object
- You will see tons of examples on the internet that do this.
- They are out of date. `defer` is widely supported and better

Performance comparison
page loading time

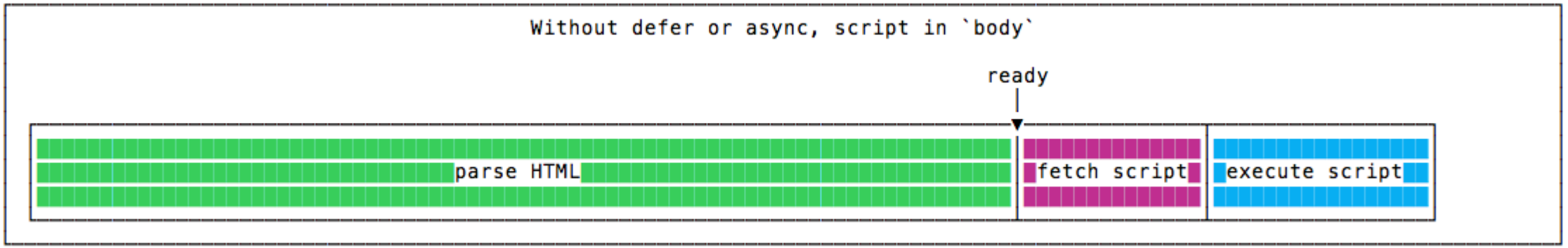
- No defer, `<script>` in head



The parsing is paused until the script is fetched, and executed. Once this is done, parsing resumes.

Performance comparison
page loading time

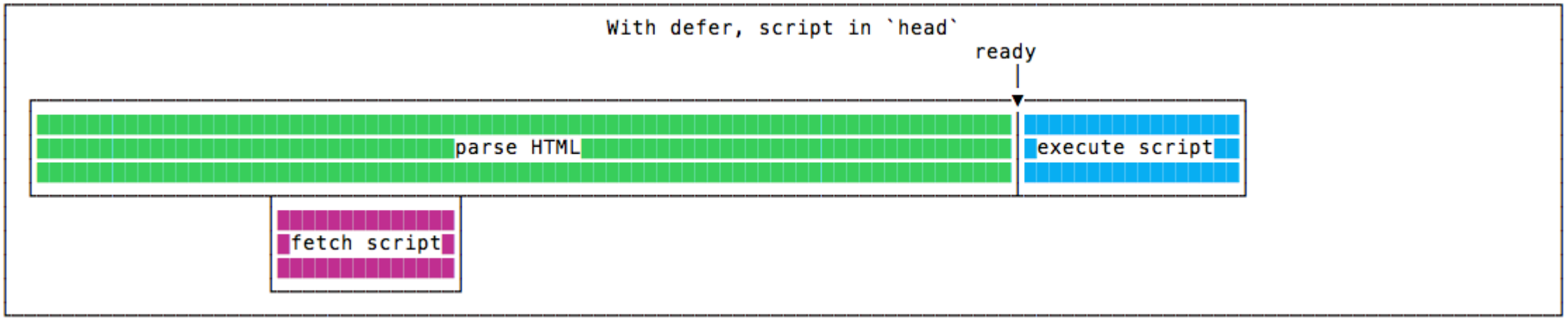
- No defer, `<script>` in the bottom of the page before </body>



The parsing is done without any pauses, and when it finishes, the script is fetched, and executed. Parsing is done before the script is even downloaded, so the page appears to the user way before the previous example.

Performance comparison
page loading time

- With defer, `<script>` in head

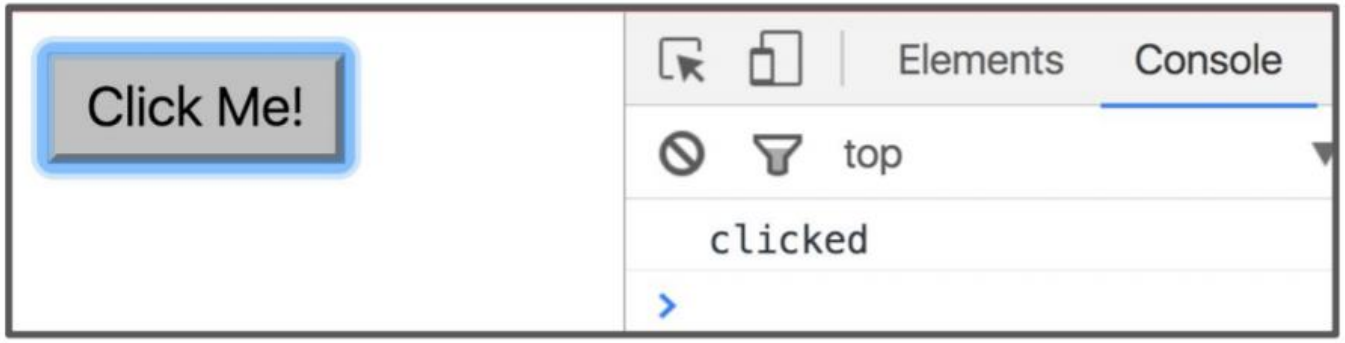


The script is fetched **asynchronously**, and it's executed only after the HTML parsing is done.
Parsing finishes just like previous example, but overall the script execution finishes well before, because the script has been downloaded in parallel with the HTML parsing.
So this is the winning solution in terms of speed

```html
<html>
▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer></script>
  </head>
▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```js
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```



How to pass input args to handler function?

- It is not possible to pass the argument to the handler function directly because this would be the same as calling the function in place

script.js

```
let newContent = ... ; // user provided data
button.addEventListener('click', onClick(newContent));
function onClick(param) {
   console.log(param);
}
```

- Handler function will be executed immediately when the line gets parsed
  - Handler function is executed without the event having fired
  - When the event fires, the handler function will not be executed

How to pass input args to handler function?

- We can use an **anonymous** function instead

script.js

```
let newContent = ... ; // user provided data
button.addEventListener('click', function() {
      onClick(newContent);
});
function onClick(param) {
   console.log(param);
}
```

- This works, because we are not calling the handler function directly
- Anonymous function is used as a handler function that calls our actual handler function with the input argument
  - outer (anonymous) function calls another function that does the actual job

HTML Element Attributes and DOM Object Properties

- Roughly every **attribute** on an HTML element is a **property** on its respective DOM object...

HTML

```
<img src="puppy.jpg" />
```

JS

```
const element = document.querySelector('img');
element.id = 'hello';
element.src = 'kitten.jpg'; // change image
```

DOM Object Properties for all HTML elements

Property	Description
`id`	Gets/sets the value of the id attribute of the element, as a string
`innerHTML`	Gets/sets the raw HTML between the starting and ending tags of an element, as a string (parses content as HTML source code)
`textContent`	Gets/sets the text content of a node and its descendants. (This property is inherited from Node) (parses content as text only)
`classList`	An object containing the classes applied to the element
`setAttribute`	Sets the value of attribute on the specified element

Other DOM Object Properties Examples

HTML
```
<div class="myclass"></div>
```

JS
```
const element = document.querySelector('.myclass');
element.id = 'myid';
```

HTML
```
<div class="myclass" id="myid"></div>
```

JS
```
element.innerHTML = '<em>Hello World</em>';
```

HTML
```
<div class="myclass" id="myid"><em>Hello World</em></div>
```

Hello World

Other DOM Object Properties Examples

HTML

```
<div class="myclass"></div>
```

JS

```
const element = document.querySelector('.myclass');
element.id = 'myid';
```

HTML

```
<div class="myclass" id="myid"></div>
```

JS

```
element.textContent = '<em>Hello World</em>';
```

HTML

```
<div class="myclass" id="myid">&lt;em&gt;Hello World&lt;/em&gt;</div>
```

Hello World

Adding and removing classes

- You can control classes applied to an HTML element via `classList.add` and `classList.remove`:

HTML
```
<img class="hidden" src="puppy.jpg"></div>
```

JS
```
const image = document.querySelector('img');
// Adds a CSS class called "active".
image.classList.add('active');
// Removes a CSS class called "hidden".
image.classList.remove('hidden');
```

HTML
```
<img class="active" src="puppy.jpg"></div>
```

Important:
The `classList.toggle` function is a part of the ClassList API and is a convenient way to add or remove a class from an element's list of classes. If the class is present, it gets removed; if not, it gets added.

Add elements via DOM

- We can create elements dynamically and add them to the web page via `createElement` and `appendChild`:

```JS
new_elem = document.createElement(tag string);
other_elem.appendChild(new_elem);
```

- Technically you can also add elements to the webpage via `innerHTML`, but it poses a security risk.
- Try not to use innerHTML like this:
  `element.innerHTML = '<h1>Hooray!</h1>'`

Example: Add elements via DOM

HTML

```
<div class="row1"></div>
<div class="row2"></div>
```

JS

```
const newHeader = document.createElement('h1');
newHeader.textContent = 'Hooray!';
const element = document.querySelector('div.row1');
element.appendChild(newHeader);
```

HTML

```
<div class="row1"><h1>Hooray!</h1></div>
<div class="row2"></div>
```

Remove elements via DOM

- We can also call remove elements from the DOM by calling the `remove()` method on the DOM object:

```JS
const element = document.querySelector('img');
element.remove();
```

- And actually setting the innerHTML of an element to an empty string is a fine way of removing all children from a parent node:
- This is fine and poses no security risk:
  `element.innerHTML = '';`

Adding and removing attributes

HTML

```
<button id="ok">OK</button>
```

JS

```
const button= document.querySelector("#ok");
button.setAttribute("type", "submit");
button.setAttribute("disabled", "");
```

HTML

```
<button "type"="submit" id="ok" disabled>OK</button>
```

Adding handler functions via attributes

using the onclick attribute

HTML

```
<button id="ok">OK</button>
```

JS

```
function myFunction() {
  console.log("Button pressed!")
}
const button= document.querySelector("#ok");
button.setAttribute("onclick", "myFunction()");
```

HTML

```
<button "onclick"="myFunction()" id="ok">OK</button>
```

Adding handler functions via attributes

using the onclick attribute (can provide fixed input parameters)

HTML

```
<button id="ok">OK</button>
```

JS

```
function myFunction(param) {
  console.log("Button pressed with input: " + param)
}
const button= document.querySelector("#ok");
button.setAttribute("onclick", "myFunction(10)");
```

HTML

```
<button "onclick"="myFunction()" id="ok">OK</button>
```

Hide elements via DOM

- There is yet another super helpful value for `display`:

```
display: block;
display: inline;
display: inline-block;
display: flex;
display: none;
```

- `display: none;` turns off rendering for the element and all its children. It's treated as if the element were not in the document at all but the content (such as the images) is still loaded.

Hide elements via DOM

- Hide element with `id="name"`

JS
```
const element = document.querySelector('#name');
element.classList.add('hidden');
```

- You have to add hidden class in your CSS file:

CSS
```
.hidden {
   display: none;
}
```

- Reveal element:

JS
```
const element = document.querySelector('#name');
element.classList.remove('hidden');
```

Hide elements via DOM (in Bootstrap)

- Hide element with `id="name"`

```JS
const element = document.querySelector('#name');
element.classList.add('d-none');
```

- where d-none class is already defined in Bootstrap 5 CSS.
- Reveal element:

```JS
const element = document.querySelector('#name');
element.classList.remove('d-none');
```

Recap

- Several strategies for updating HTML elements in JS:

1. Change content of existing HTML elements in page (e.g. innerHTML):
   - Good for simple text updates
2. Add elements via `createElement` and `appendChild`
   - Needed if you're adding a variable number of elements
3. Put all "views" in the HTML but set inactive ones to hidden, then update display state as necessary.
   - Good when you know ahead of time what element(s) you want to display
   - Can be used in conjunction with (1) and/or (2)

HTML input elements

- Single-line text input:

```
<input type="text" />
```
HTML

- Multi-line text input:

```
<textarea></textarea>
```
HTML

- In JavaScript, you can read and set the input text via `inputElement.value`

```
// change event is fired when element loses focus (user leaves element)
// keyup event is fired after every key press
const input = document.querySelector('input');
input.addEventListener('change', myFunction);
```
JS

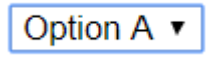
HTML input elements

Codepen

- Checkbox:

```
<label><input type="checkbox" name="size" value="done"/>Option</label>
```
HTML


```
const checkbox = document.querySelector('input');
checkbox.addEventListener('change', function() {
  console.log(checkbox.checked);
  console.log(checkbox.value);
})
```
JS

- Select: 


```
<select>
  <option value="1" selected>Option A
  </option>
  <option value="2">Option B</option>
  <option value="3">Option C</option>
</select>
```
HTML

```
const selectElem = document.querySelector('select');
selectElem.addEventListener('change', function() {
  const index = selectElem.selectedIndex;
  console.log('index selected: ' + index);
  console.log('option value selected: ' +
selectElem.options[index].value);
  console.log('option text selected: ' +
selectElem.options[index].text);
})
```
JS

Form submit

- What if you want to have a form with input elements that can be submitted after you click "enter"?

Form submit

1. Wrap your input elements in a `<form>` :

HTML

```
<form>
  First name:<br/>
  <input type="text" id="fname"/><br/>
  Last name:<br/>
  <input type="text" id="lname"/><br/><br/>
  <input type="submit" value="Submit"/>
</form>
```

First name:

Last name:

Submit

- You need to use **`<input type="submit">`** or **`<button type="submit">`** instead of `type="button"` to capture "submit" event

Form submit

2. Listen for the `"submit"` event on the form element:

```
const form = document.querySelector('form');
form.addEventListener('submit', onFormSubmit);
```
JS

- When you use **`type="submit"`** on `<input>` or on `<button>` the `"submit"` event will fire on form element. When `type="button"` the `"submit"` event is not fired on form element.

Form submit

3. Capture input data:

```JS
function onFormSubmit(event) {
  event.preventDefault();
  const name = document.querySelector('#fname');
  const surname = document.querySelector('#lname');
  console.log(name.value+" "+surname.value);
}
```

- The page will refresh (and data on form will be automatically reset) on submit event unless you explicitly prevent it
  - You may prevent the default action before handling the event through `event.preventDefault()`:

Form submit (in Boostrap)

```
<form>
  <div class="mb-3 row">
    <label for="firstname" class="form-
label">First name</label>
    <input type="text" class="form-control"
id="firstname" placeholder="Enter your first
name">
  </div>
  <div class="mb-3 row">
    <label for="lastname" class="form-label">Last
name</label>
    <input type="text" class="form-control"
id="lastname" placeholder="Enter your last name">
  </div>
  <button type="submit" class="btn btn-
primary">Submit</button>
</form>
```

First name

Enter your first name

Last name

Enter your last name

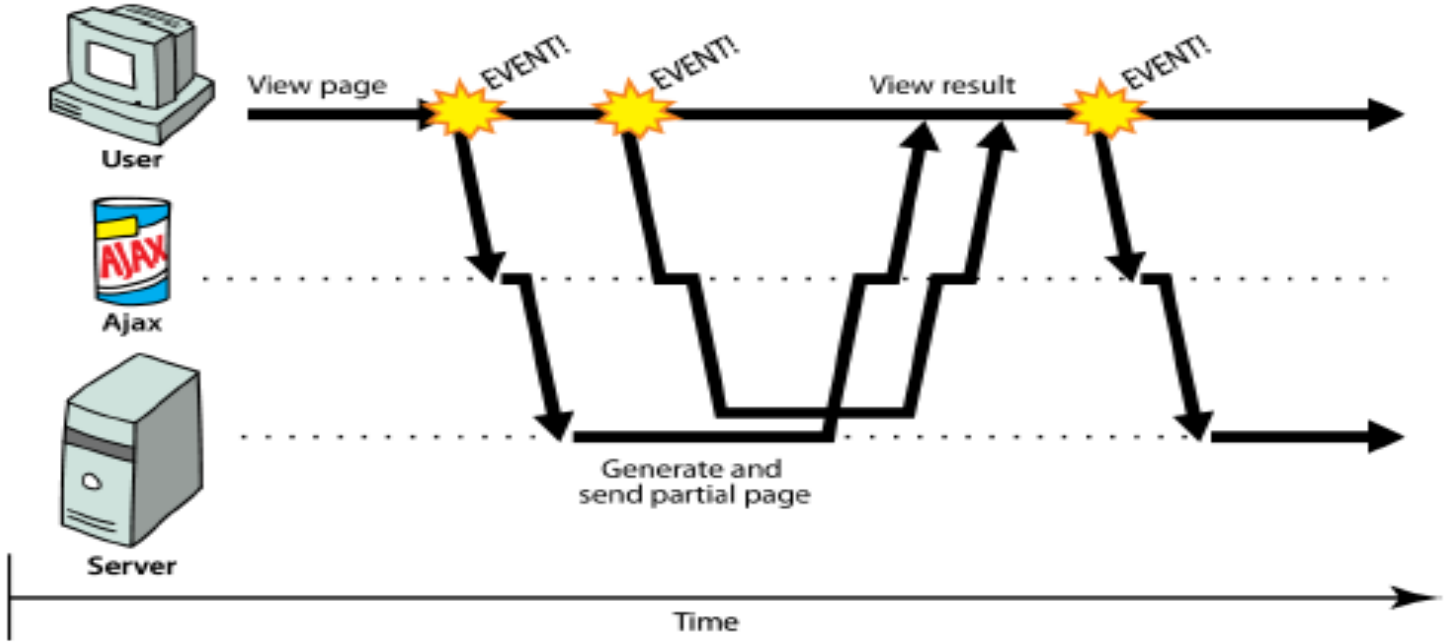
Submit

AJAX

- Asynchronous JavaScript and XML
- Not a programming language; a particular way of using JavaScript
- Can be used to download/upload data from/to a server in the background (Asynchronous)
- Allows dynamically updating a page without making the user wait and without refreshing the page
- Avoids the "click-wait-refresh" pattern
- Implemented in vanilla JavaScript using **XMLHttpRequest** or **Fetch API**

AJAX

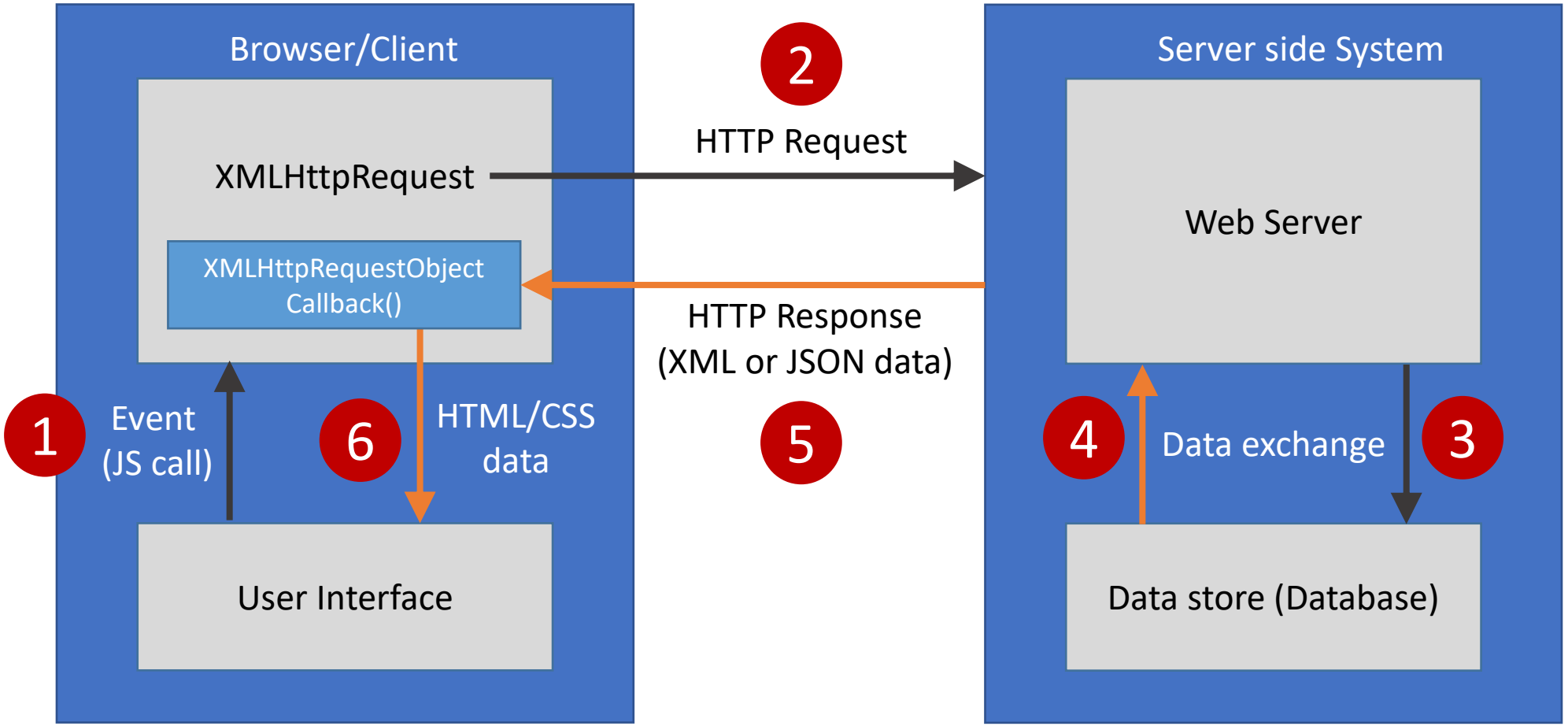
- **asynchronous**: user can keep interacting with page while data loads
  - communication pattern made possible by Ajax



How XMLHttpRequest works?

- The `XMLHttpRequest` object makes an asynchronous call to a web server.
- The web server processes the request and returns an XML or JSON document that contains the result.
- The `XMLHttpRequest` object calls the callback function and exposes the response from the web server so that the request can be processed.
- The client updates the DOM representing the page with the new data.

How XMLHttpRequest works?



XMLHttpRequest

- `XMLHttpRequest` object is the key to Ajax programming.
- It’s main purpose is to put an asynchronous HTTP request to the web server.
- Because of this asynchronous call to the web server, the user is allowed to continue using the webpage without the interruption of a browser refresh and the loading of a new or revised page.
- This object has few properties.

Properties of XMLHttpRequest

- Property 1: `objXMLHttp.onreadystatechange`

  This property holds the reference of callback function which is going to process the response from the server.

```
objXMLHttp.onreadystatechange = procRequest;
```

** "procRequest" is the function which will process the response*

Properties of XMLHttpRequest

- Property 2 : `objXMLHttp.readyState`

  This property holds the status of server response.

```
objXMLHttp.readyState = [state];
```

State	Description
0	The request is not initialized
1	The request has been set up
2	The request has been sent
3	The request is in process
4	The request is complete

Properties of XMLHttpRequest

- Property 3: `objXMLHttp.responseText`

  This property retrieves the data sent back from server.

```
var objVal = objXMLHttp.responseText;
```

  While the `responseText` is used to return text (including JSON), `responseXML` can be used to return an XML document object.

  var xmlDoc = objXMLHttp.responseXML.documentElement

Download data

Example using XMLHttpRequest (GET)

JS

```
// Set up our HTTP request
const xhr = new XMLHttpRequest();
// Setup our listener to process completed requests
xhr.onreadystatechange = function () {
        // Only run if the request is complete (xhr.readyState = 4)
        if (xhr.readyState !== 4) return;
        // Process our return data
        if (xhr.status >= 200 && xhr.status < 300) {
                // What to do when the request is successful
                console.log(JSON.parse(xhr.responseText));
        } else {
                // What to do when the request has failed
                console.log('Error: ', xhr);
        }
};
// Create and send a GET request
// The first argument is the post type (GET, POST, PUT, DELETE, etc.)
// The second argument is the endpoint URL
xhr.open("GET", "https://api.openaq.org/v1/countries");
xhr.setRequestHeader("Accept", "application/json");
xhr.send();
```

To test in VSCode you need to install xhr2 package via terminal since XMLHttpRequest is a built-in object in web browsers:

```
node install xhr2
```

and then include this line in the first line of your script:

```
const
XMLHttpRequest =
require('xhr2');
```

Example using Fetch API (GET)

JS

```
fetch("https://api.openaq.org/v1/countries", {
  method: "GET",
  headers: {
    "Accept": "application/json",
  }
})
.then(
  response => { // handle the response
    if (response.status !== 200) {
      console.log('Status Code: ' + response.status);
      return;
    }
    // Parse response as JSON (no need to call JSON.parse())
    response.json().then(
        data => {
            console.log(data);
        }
    );
  } // end of response
) // end of then
.catch( error => { // handle the error
    console.log('Error: ', error);
});
```

Upload data

Example using XMLHttpRequest (POST)

JS

```
// Set up our HTTP request
const xhr = new XMLHttpRequest();
// Setup our listener to process completed requests
xhr.onreadystatechange = function () {
        // Only run if the request is complete
        if (xhr.readyState !== 4) return;
        // Process our return data
        if (xhr.status >= 200 && xhr.status < 300) {
                console.log(JSON.parse(xhr.responseText));
        } else {
                console.log('error', xhr);
        }
};
// Create and send a POST request. The second argument is the endpoint
URL which will receive the POST message (e.g. a PHP file on the same
server and folder)
xhr.open('POST', 'https://cs.ucy.ac.cy/~csp5pa1/test.php');
xhr.setRequestHeader("Content-type", "application/json");
data = {};
data.name = "John";
data.surname = "Smith";
xhr.send(JSON.stringify(data));
```

- Create a JavaScript object with two properties (name, surname).
- Convert JS object to JSON string.
- Send JSON string to the predefined endpoint.

Upload data

Example using Fetch API (POST)

JS

```
fetch('https://cs.ucy.ac.cy/~csp5pa1/test.php', {
  method: "POST",
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(
  response => { // handle the response
    if (response.status !== 200) {
      console.log('Status Code: ' + response.status);
      return;
    }
    // Parse response as JSON (no need to call JSON.parse())
    response.json().then(
        data => {
            console.log(data);
         }
    );
  } // end of response
)
.catch( error => { // handle the error
    console.log('Error: ', error);
});
```

```
data = {};
data.name = "John";
data.surname = "Smith";
```