CLIENT-SIDE

WEB APPS

ASYNCHRONOUS JAVASCRIPT & XML (AJAX)

THE DOCUMENT OBJECT MODEL (DOM)

DOM EVENTS

VERSION

v1.3	23 November 2021
v1.2	17 November 2021
v1.1	16 November 2021
v1.0	9 November 2021

ACKNOWLEDGMENTS

THANKS TO:

- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

PRINTABLE VERSION OF THE TALK

Download PDF

THE THREE AMIGOS

HTML + JS + CSS

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css" type="text/css">
    <script src="script-1.js"></script>
  </head>
  <body>
    ...
    <script src="script-2.js"></script>
  </body>
</html>
```

AJAX

ASYNCHRONOUS JAVASCRIPT & XML

ABOUT AJAX

AJAX (Asynchronous JavaScript and XML) is a set of web development techniques that uses various web technologies on the client-side to create asynchronous web applications. With AJAX, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behaviour of the existing page. By decoupling the data interchange layer from the presentation layer, AJAX allows web pages and, by extension, web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly utilize JSON instead of XML.

XMLHTTPREQUEST

`XMLHttpRequest` (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing. `XMLHttpRequest` is used heavily in AJAX programming. Despite its name, `XMLHttpRequest` can be used to retrieve any type of data, not just XML.

PROPERTIES

- `onreadystatechange`
- `readyState`
- `response`
- `responseText`
- `responseType`
- `responseURL`
- `responseXML`
- `status`
- `statusText`

METHODS

- `abort()`
- `getAllResponseHeaders()`
- `getResponseHeader()`
- `open()`
- `overrideMimeType()`
- `send()`
- `setRequestHeader()`

EVENTS

- `abort`
- `error`
- `load`
- `loadend`
- `loadstart`
- `progress`
- `timeout`

XMLHTTPREQUEST

`XMLHttpRequest` (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing. `XMLHttpRequest` is used heavily in AJAX programming. Despite its name, `XMLHttpRequest` can be used to retrieve any type of data, not just XML.

`XMLHttpRequest.readyState`

The `XMLHttpRequest.readyState` property returns the state an `XMLHttpRequest` client is in. An XHR client exists in one of the following states:

Value	State	Description
0	UNSENT	Client has been created. `open()` not called yet.
1	OPENED	`open()` has been called.
2	HEADERS_RECEIVED	`send()` has been called, and headers and status are available.
3	LOADING	Downloading; responseText holds partial data.
4	DONE	The operation is complete.

USING XHR

```
function doAjax(url, method) {
  const request = new XMLHttpRequest();                      // Create the XHR request
  return new Promise(function (resolve, reject) {            // Return it as a Promise
    request.onreadystatechange = function () {               // Setup our listener to process compeleted requests
      if (request.readyState !== 4) return;                  // Only run if the request is complete
      if (request.status >= 200 && request.status < 300) {  // Process the response
        resolve(JSON.parse(request.responseText));           // When successful
      } else {                                               // When failed
        reject({
          status: request.status,
          statusText: request.statusText
        });
      }
    };
    request.open(method || 'GET', url, true);               // Setup our HTTP request
    request.send();                                         // Send the request
  });
}
```

SAME ORIGIN POLICY

The same-origin policy is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin.

It helps isolate potentially malicious documents, reducing possible attack vectors. For example, it prevents a malicious website on the Internet from running JS in a browser to read data from a third-party webmail service (which the user is signed into) or a company intranet (which is protected from direct access by the attacker by not having a public IP address) and relaying that data to the attacker.

DEFINITION OF AN ORIGIN

Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. You may see this referenced as the “scheme/host/port tuple”, or just “tuple”. The following table gives examples of origin comparisons with the URL `http://store.company.com/dir/page.html`:

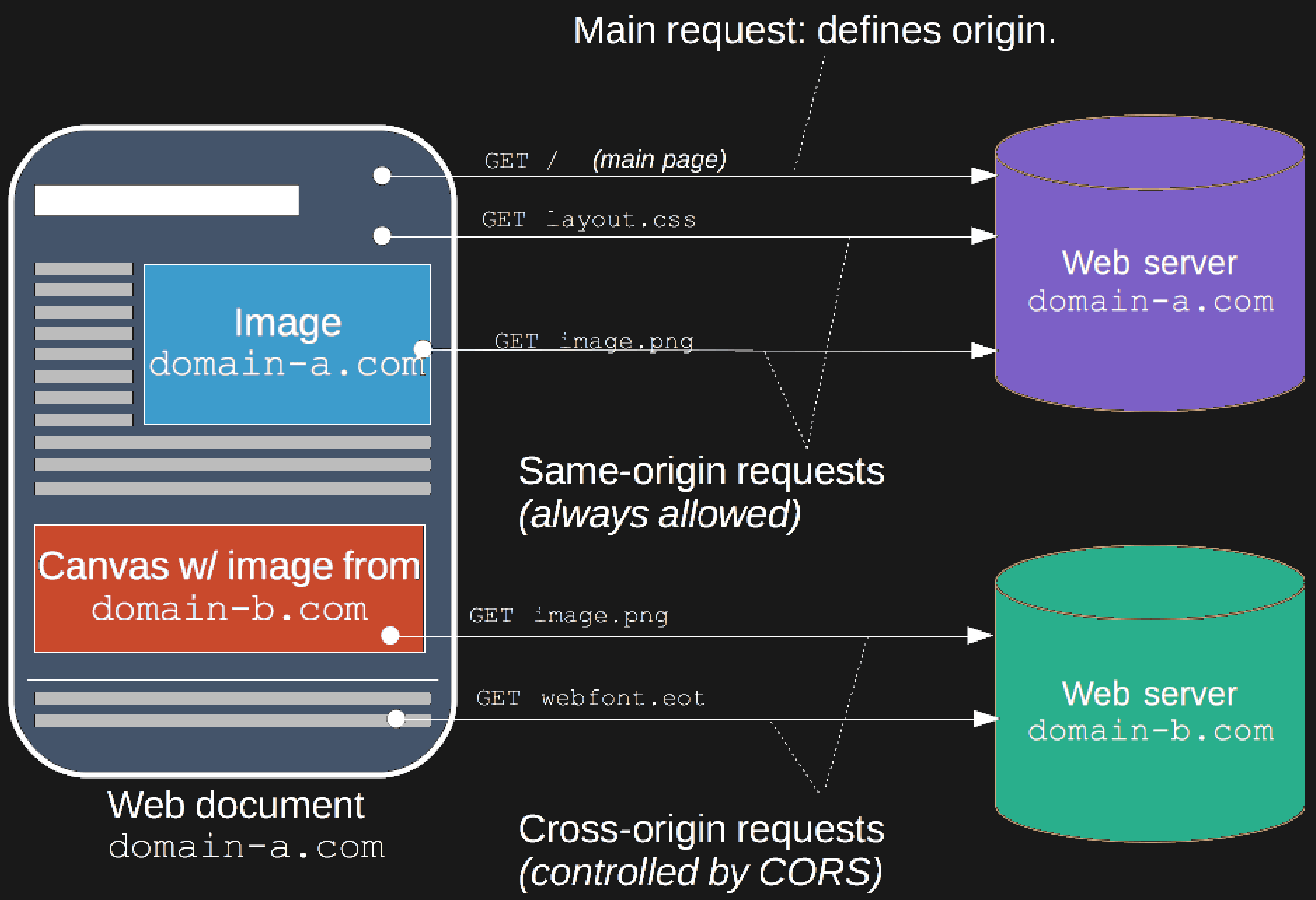
URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Same origin	Only the path differs
`http://store.company.com/dir/inner/another.html`	Same origin	Only the path differs
`https://store.company.com/page.html`	Failure	Different protocol
`http://store.company.com:81/dir/page.html`	Failure	Different port (`http://` is port 80 by default)
`http://news.company.com/dir/page.html`	Failure	Different host

CROSS-ORIGIN RESOURCE SHARING (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from `https://domain-a.com` uses `XMLHttpRequest` to make a request for `https://domain-b.com/data.json`.

For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. For example, `XMLHttpRequest` and the Fetch API follow the same-origin policy. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers.



CROSS-ORIGIN RESOURCE SHARING (CORS)

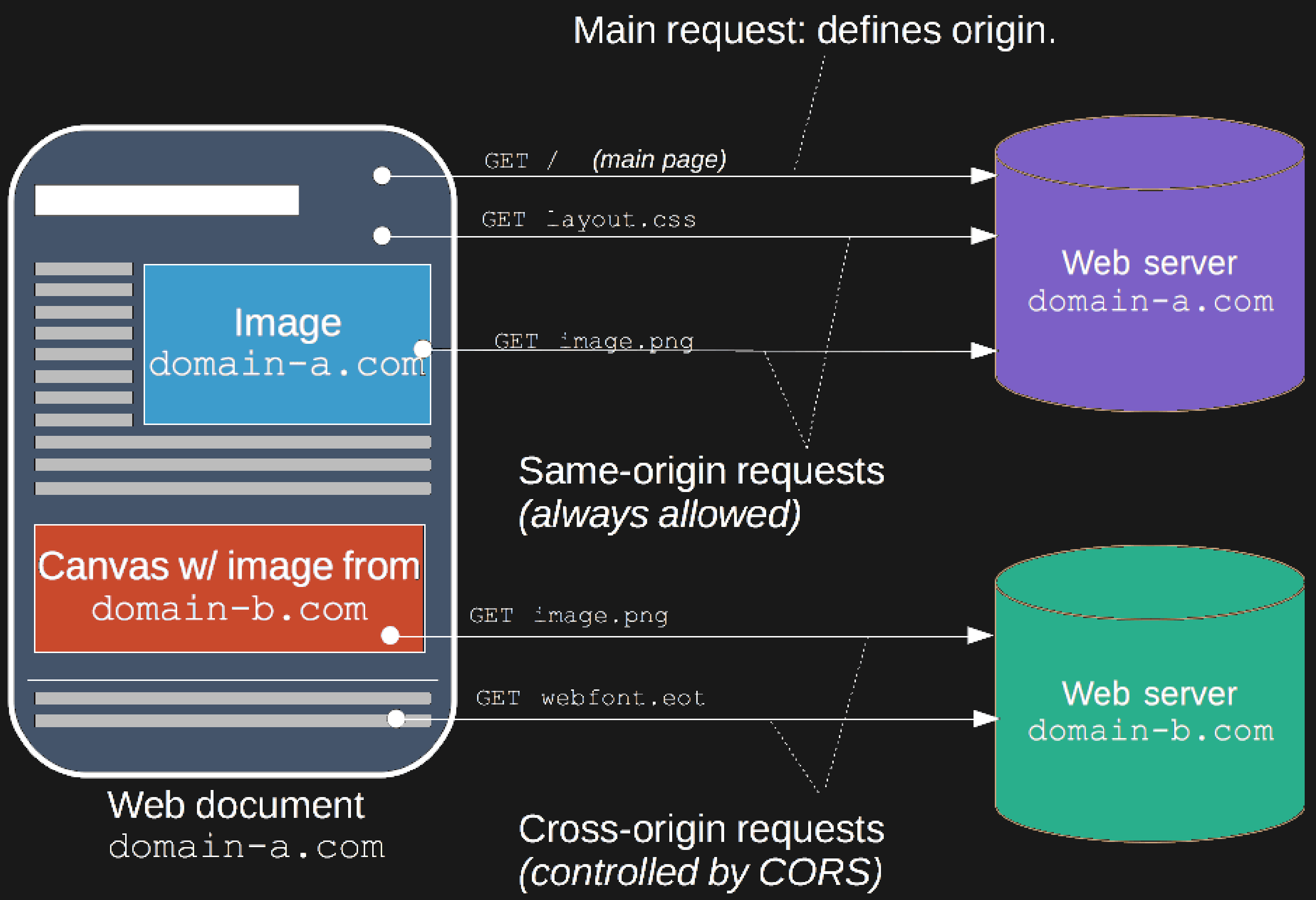
The CORS mechanism supports secure cross-origin requests and data transfers between browsers and servers. Modern browsers use CORS in APIs such as XMLHttpRequest or Fetch to mitigate the risks of cross-origin HTTP requests.

```
app.use((req, res, next) => {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "*");
  next();
});
```

Alternatively,

```
const cors = require('cors');

app.use(cors());
```



THE DOM

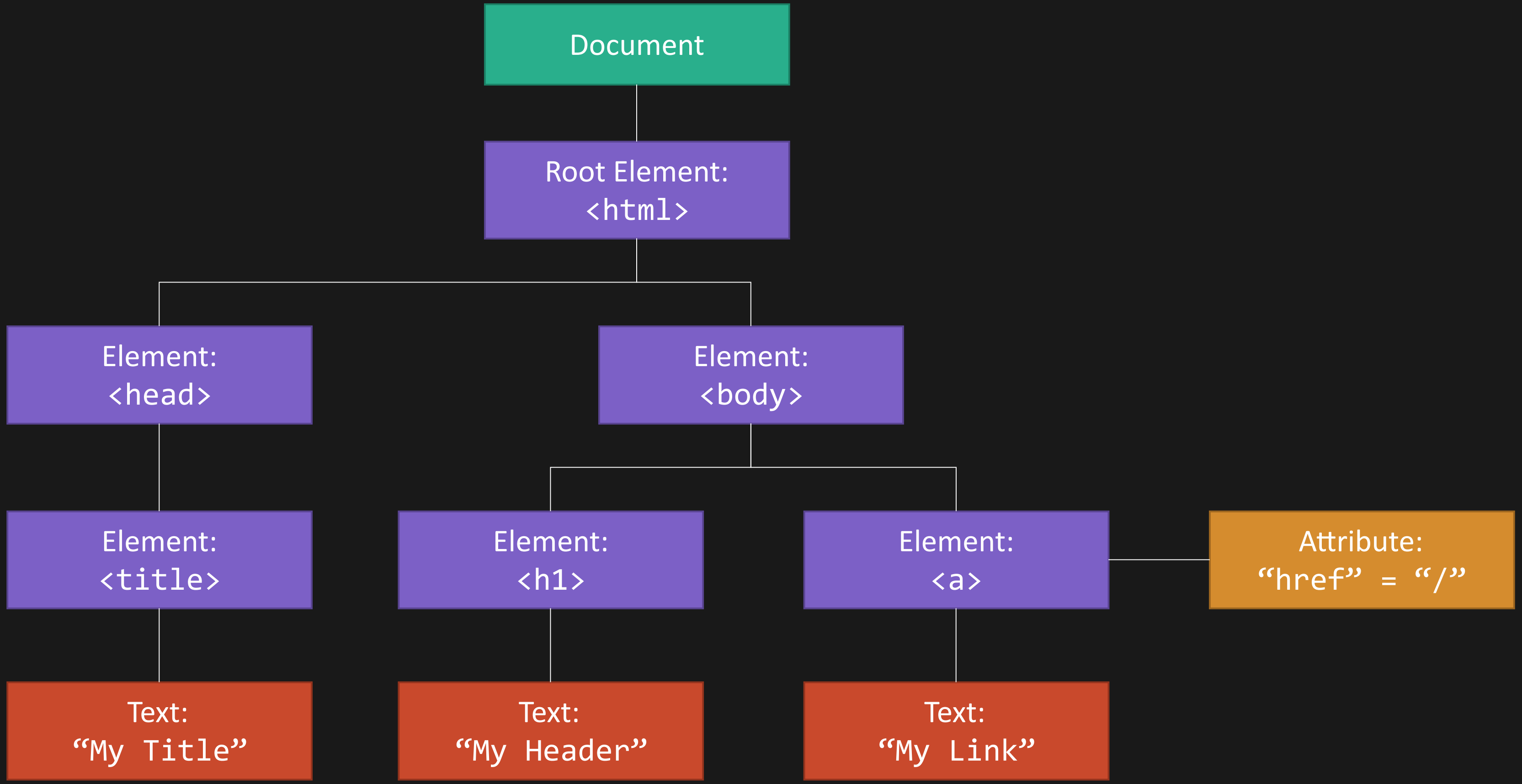
THE DOCUMENT OBJECT MODEL

ABOUT THE DOM

The **Document Object Model (DOM)** is the data representation of the objects that comprise the structure and content of a document on the web. The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

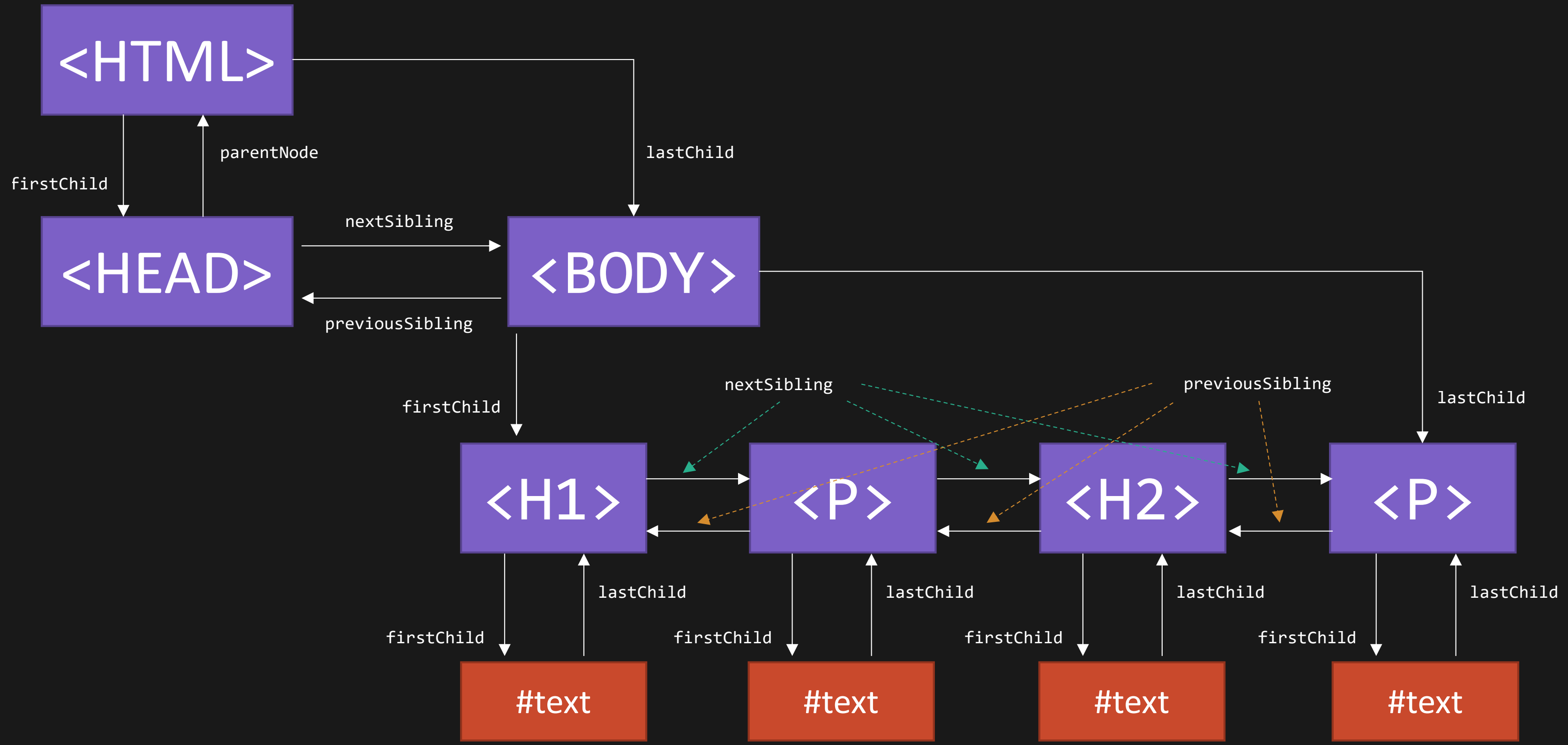
A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

THE DOM TREE



```html
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>My Header</h1>
    <a href="/">My Link</a>
  </body>
</html>
```

WALK THE DOM



```
function walkTheDOM(node, func) {
  func(node);
  node = node.firstChild;
  while (node) {
    walkTheDOM(node, func);
    node = node.nextSibling;
  }
}

walkTheDOM(document, console.log);
```

THE DOM API

DOCUMENT

- `document.head`
- `document.body`
- `document.querySelector(selector)`
- `document.querySelectorAll(name)`
- `document.getElementById(id)`
- `document.getElementsByTagName(tagName)`
- `document.getElementsByClassName(className)`
- `document.createElement(name)`

ELEMENT

- `element.outerHTML`
- `element.innerHTML`
- `element.style`
- `element.className`
- `element.attributes`
- `element.children`
- `element.innerText`
- `element.dataset`

- `element.appendChild(node)`
- `element.setAttribute()`
- `element.getAttribute()`
- `element.hasAttribute()`
- `element.removeAttribute()`
- `element.addEventListener()`
- `element.removeEventListener()`

USING THE DOM API

```html
<div id="page">
  <h1>My Page</h1>
  <p>Lorem ipsum <a href="/link">dolor</a> sit amet, ...</p>
  <div id="my-region" class="region container">
    <button id="my-button"
            name="myButton"
            data-value="my-value"
            class="btn btn-primary"
            onclick="doAction(event);">Action</button>
  </div>
</div>
```

```javascript
const page = document.getElementById('page');

const header = document.createElement('h1');
  header.innerText = 'My Page';

const paragraph = document.createElement('p');
  paragraph.innerText = 'Lorem ipsum ';
  const anchor = document.createElement('a');
    anchor.href = '/link';
    anchor.innerText = 'dolor';
  paragraph.appendChild(anchor);
  const text = document.createTextNode(' sit amet, ...');
  paragraph.appendChild(text);

const myRegionDiv = document.createElement('div');
  myRegionDiv.id = 'my-region';
  myRegionDiv.className = 'region container';
  const button = document.createElement('button');
    button.id = 'my-button';
    button.setAttribute('name', 'myButton');
    button.setAttribute('data-value', 'my-value');
    button.classList.add('btn');
    button.classList.add('btn-primary');
    button.addEventListener('click', function (event) { /* doAction */ });
    button.innerText = 'Action';
  myRegionDiv.appendChild(button);

page.appendChild(header);
page.appendChild(paragraph);
page.appendChild(myRegionDiv);
```

DOM EVENTS

DOM EVENTS

Events are fired to notify code of “interesting changes” that may affect code execution. These can arise from user interactions such as using a mouse or resizing a window, changes in the state of the underlying environment (e.g. low battery or media events from the operating system), and other causes.

Documentation: Event Reference - MDN

EVENT TYPES:

- Mouse-related: mouse movement, enter/leave element, button click
- Keyboard-related: down, up, press
- Focus-related: focus in, focus out (blur)
- Timer events
- Miscellaneous:
    - Content of an element has changed.
    - Page loaded/unloaded.
    - Uncaught error.

EVENT TYPES

- Animation
- Asynchronous data fetching
- Clipboard
- Composition
- CSS transition
- Database
- DOM mutation
- Drag’n’drop, Wheel
- Focus
- Form
- Fullscreen
- Gamepad
- Gestures
- History
- HTML element content display management
- Inputs
- Keyboard
- Loading/unloading documents
- Manifests
- Media
- Messaging
- Mouse
- Network/Connection
- Payments
- Performance
- Pointer
- Print
- Promise rejection
- Sockets
- SVG
- Text selection
- Touch
- Virtual reality
- RTC (real time communication)
- Server-sent events
- Speech
- Workers

EVENT HANDLERS

Creating an event handler: must specify 3 things:

- The event of interest.
- An element of interest.
- JavaScript to invoke when the event occurs on the element.

OPTION 1: IN THE HTML

```
<div onclick="mouseClick('id42');">...</div>
```

OPTION 2: FROM JAVASCRIPT USING THE DOM

```
element.onclick = mouseClick; // old way
element.addEventListener('click', mouseClick); // W3C
```

Event handlers typically need information about the event, such as mouse position or the specific key/button that was pressed. Interesting information in the event:

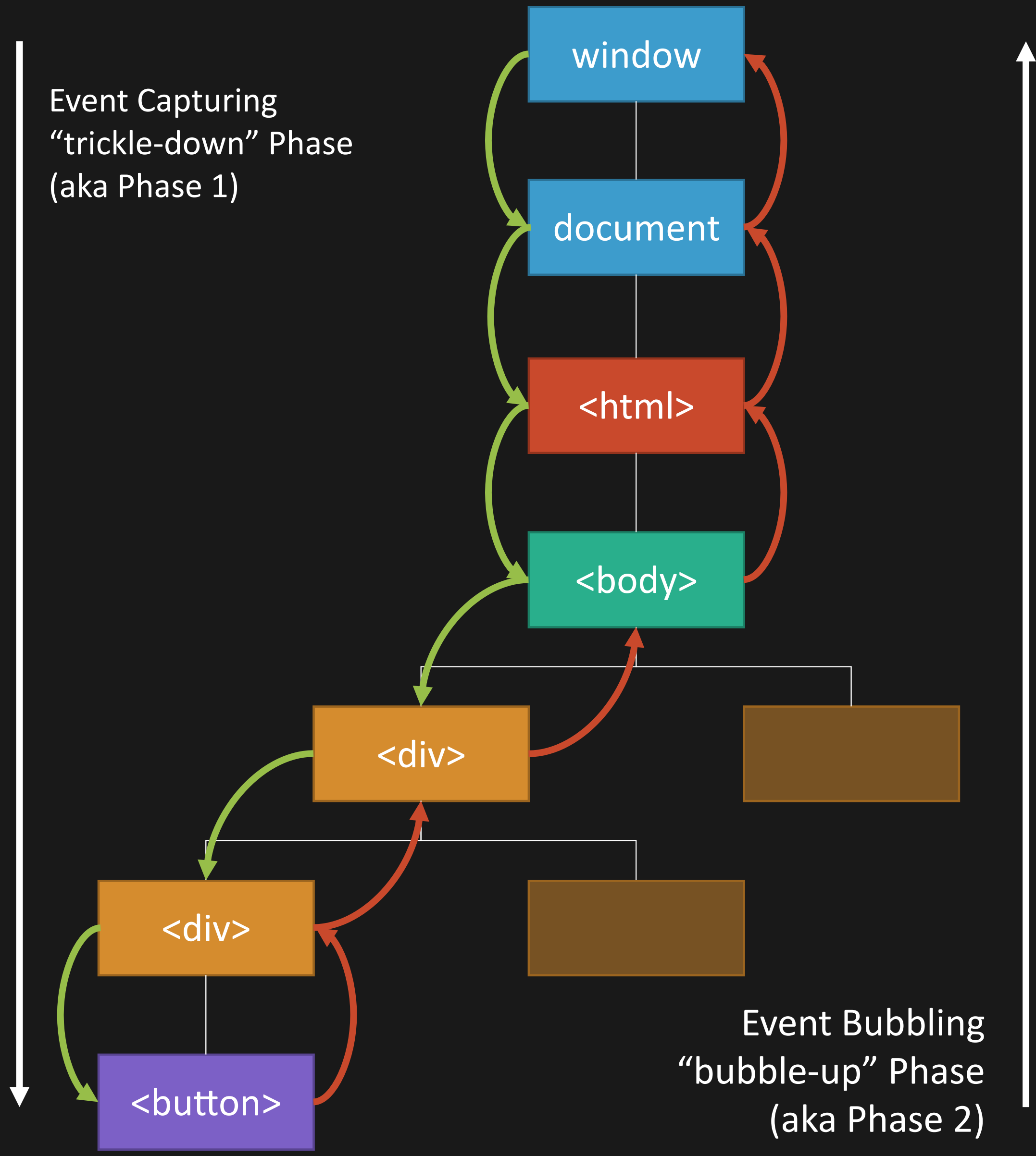
button	Mouse button that was pressed
key	Identifier for the keyboard key that was pressed (not necessarily an ASCII character!)
charCode	Integer Unicode value corresponding to key, if there is one.
clientX, clientY	Mouse position relative to the document in the browser window
screenX, screenY	Mouse position in screen coordinates

This information is available in an event object passed to the handler: in Firefox the event is normally passed as the first argument to the handler:

```
<div onclick="mouseClick(event);">
```

```
function mouseClick(event) { /* ... */ }

element.onclick = mouseClick;
element.addEventListener('click', mouseClick);
```



EVENT PROPAGATION

The `preventDefault()` method of the Event interface tells the user agent that if the event does not get explicitly handled, its default action should not be taken as it normally would be. The event continues to propagate as usual, unless one of its event listeners calls `stopPropagation()` or `stopImmediatePropagation()`, either of which terminates propagation at once.

The `stopPropagation()` method of the Event interface prevents further propagation of the current event in the capturing and bubbling phases. It does not, however, prevent any default behaviors from occurring; for instance, clicks on links are still processed. If you want to stop those behaviors, see the `preventDefault()` method.

```
element.addEventListener("click", function (event) {
  // ...
  event.preventDefault();
  event.stopPropagation();
}, false);
```

EVENT EXAMPLE

DRAG 'N DROP

```html
<div id="elem"
     onmousedown="mouseDown(event);"
     onmousemove="mouseMove(event);"
     onmouseup="mouseUp(event);">Drag Me!</div>
```

```javascript
let isMouseDown = false;
let prevX;
let prevY;

function mouseDown(event) {
  prevX = event.clientX;
  prevY = event.clientY;
  isMouseDown = true;
}
function mouseMove(event) {
  if (!isMouseDown) {
      return;
  }
  let element = document.getElementById("elem");
  element.style.left = (element.offsetLeft + (event.clientX - prevX)) + "
  element.style.top = (element.offsetTop + (event.clientY - prevY)) + "px
  prevX = event.clientX;
  prevY = event.clientY;
}
function mouseUp(event) {
  isMouseDown = false;
}
```

This slide is intentionally left blank.

Return to Course Page or Part II.