MENTORMATE™
DevCamp

DOM & Events

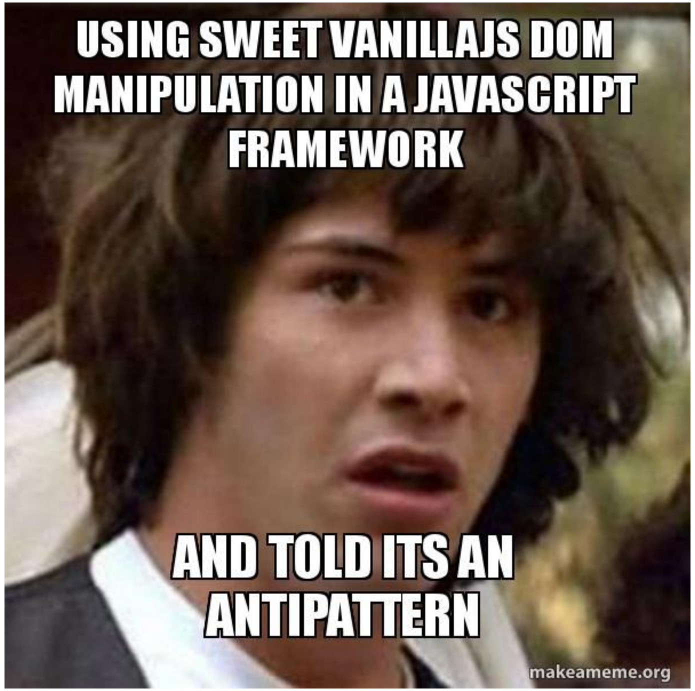
Yulia Tenincheva

Senior Cloud Engineer, MentorMate

& Special thanks to Krasimir Nedev for the slides

Agenda for today

- DOM Introduction & Traversal
- JavaScript Event Model, Event Handler Registration & the “event” object
- Capturing and bubbling events, event chaining
- Creating custom events
- AJAX, Fetch API


USING SWEET VANILLAJS DOM MANIPULATION IN A JAVASCRIPT FRAMEWORK
AND TOLD ITS AN ANTIPATTERN
makeameme.org

What is the DOM?

- DOM = Document Object Model
- Programming interface, through which we can interact with the page
- All of the **properties**, **methods**, and **events** available for manipulating and creating web pages are organized into **objects**

The only way to really learn about the DOM is to read the documentation

The very simplified version:

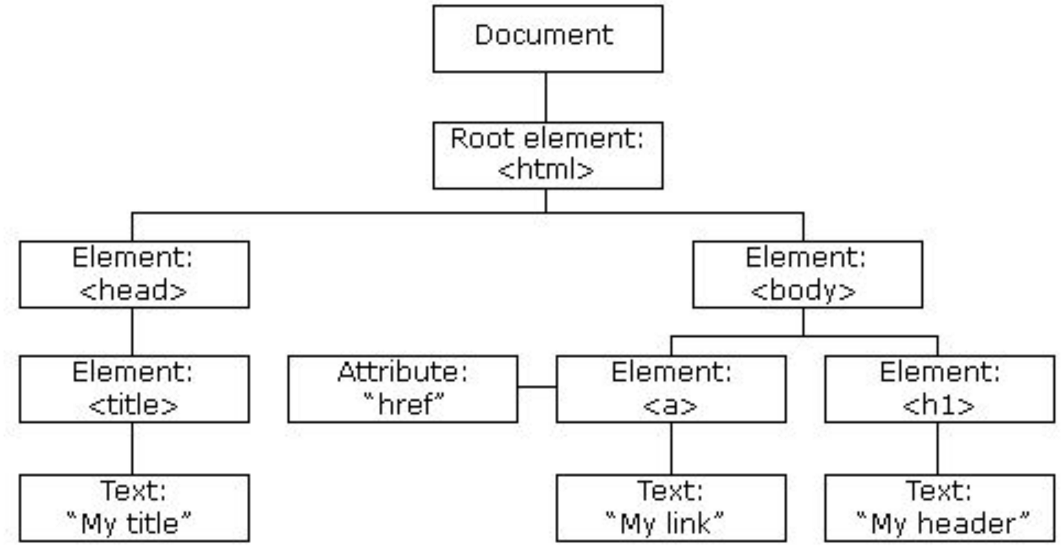
The DOM is a bridge between the HTML world of elements and the JavaScript world of variables, functions, objects, and methods.

What happens when a web page is loaded?

1. The browser makes a request and gets some HTML in response

2. The HTML is parsed and a Document Object Model (**DOM**) is created

3. It's represented as a **tree of objects (nodes)**



Relationships between nodes

The nodes in the node tree have a hierarchical relationship to each other.

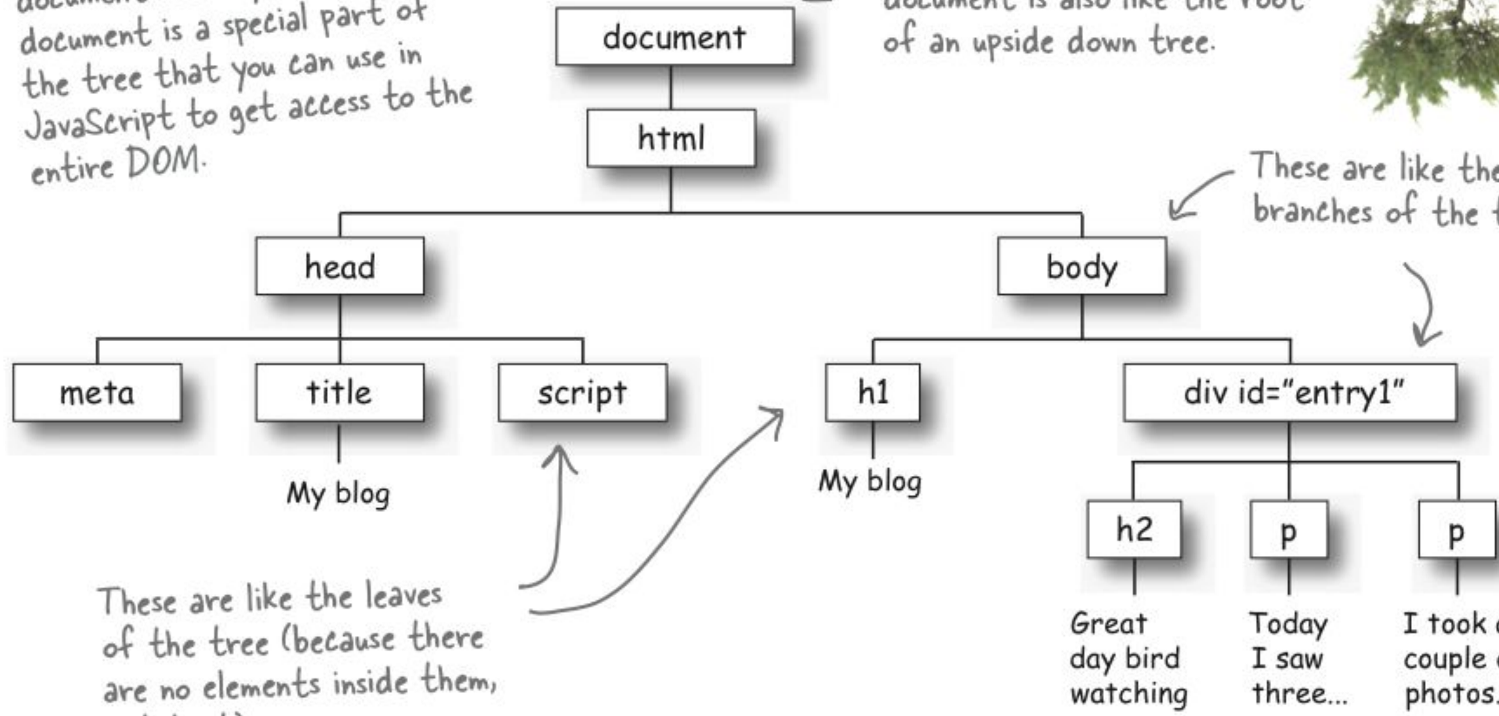
The terms **parent**, **child**, and **sibling** are used to describe the relationships.

- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

document is always at the top. document is a special part of the tree that you can use in JavaScript to get access to the entire DOM.

document is also like the root of an upside down tree.

These are like the branches of the tree.



These are like the leaves of the tree (because there are no elements inside them, just text).

The DOM includes the content of the page as well as the elements. (We don't always show all the text content when we draw the DOM, but it's there).

What can be achieved with JS

With the object model, JavaScript gets all the power it needs to interact with the page:

- All HTML elements and their attributes can be changed or removed
- New HTML elements can be added to the page
- Event listeners can be attached to the entire document or individual elements

Through the document interface we get complete control over the entire page

The “document” object

- The “document” object represents your web page
- If you want to access any element, you always start with accessing the “document” object itself
- The “document” object gives us an API to traverse or interact with elements

Selecting Elements

- By id:

```
const someElement = document.getElementById("someId");
```

- By tag name:

```
const paragraphs = document.getElementsByTagName("p");
```

- By class name:

```
const elements = document.getElementsByClassName("someClass");
```

- By css (query) selector:

```
const paragraphs = document.querySelectorAll("p.someClass");
```

Changing elements

- Changing the inner HTML of elements

```
element.innerHTML = new html content
```

- Changing an attribute of an element

```
element.attribute = new value
element.setAttribute(attribute, value)
```

- Changing the inline style of an element

```
element.style.property = new style
```

Adding and removing elements

- Creating elements

```
document.createElement(element)
```

- Removing elements

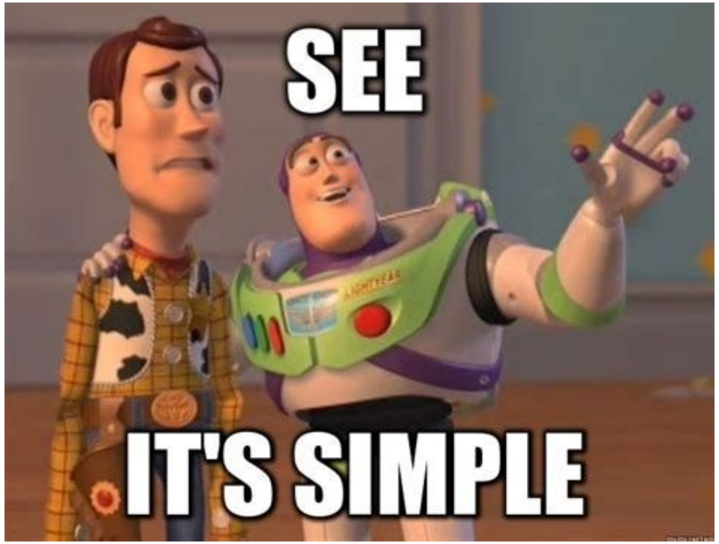
```
element.removeChild(element)
element.remove()
```

- Adding elements to the body

```
element.appendChild(element)
```

- Replacing existing elements

```
element.replaceChild(new, old)
```



Example

```
<div id="parentElement">
     <span id="childElement">foo bar</span>
</div>

<script>

let newNode = document.createElement("span");
let parentDiv = document.getElementById("childElement").parentNode;
let referenceNode = document.getElementById("childElement");
parentDiv.insertBefore(newNode,referenceNode);

// if referenceNode is null, then newNode is inserted at the end of parentNode’s child
nodes.
```

MENTORMATE™ DevCamp

The Node Interface

Node is an interface from which various types of DOM API objects inherit, allowing those types to be treated similarly; for example, inheriting the same set of methods, or being testable in the same way.

This includes the **Document** and **Element** interfaces**.**

This means that every Element in the DOM has some of the same properties and methods.

Navigating the DOM

Every node has a reference to it’s

- `.parentNode`
- `.childNodes[index]`
- `.children[index]`
- `.firstChild`
- `.lastChild`
- `.nextSibling`
- `.previousSibling`

Relevant attributes, properties and methods

- `.textContent` - reads and/or writes text
- `.innerHTML` - returns and/or writes the HTML of an element
- `.value` - gets and sets value of inputs
- `.getAttribute()` - returns the value of attributes of an element
- `.setAttribute()` - sets the value of an attribute of an element
- `.removeAttribute()` - removes an attribute from an element
- `.hasAttribute()` - returns true if the attribute exists, otherwise it returns false
- `.classList` - read-only prop, a collection of the class attributes of an element
  - `add() - adds a class`
  - `remove() - removes a class`

Events

Events

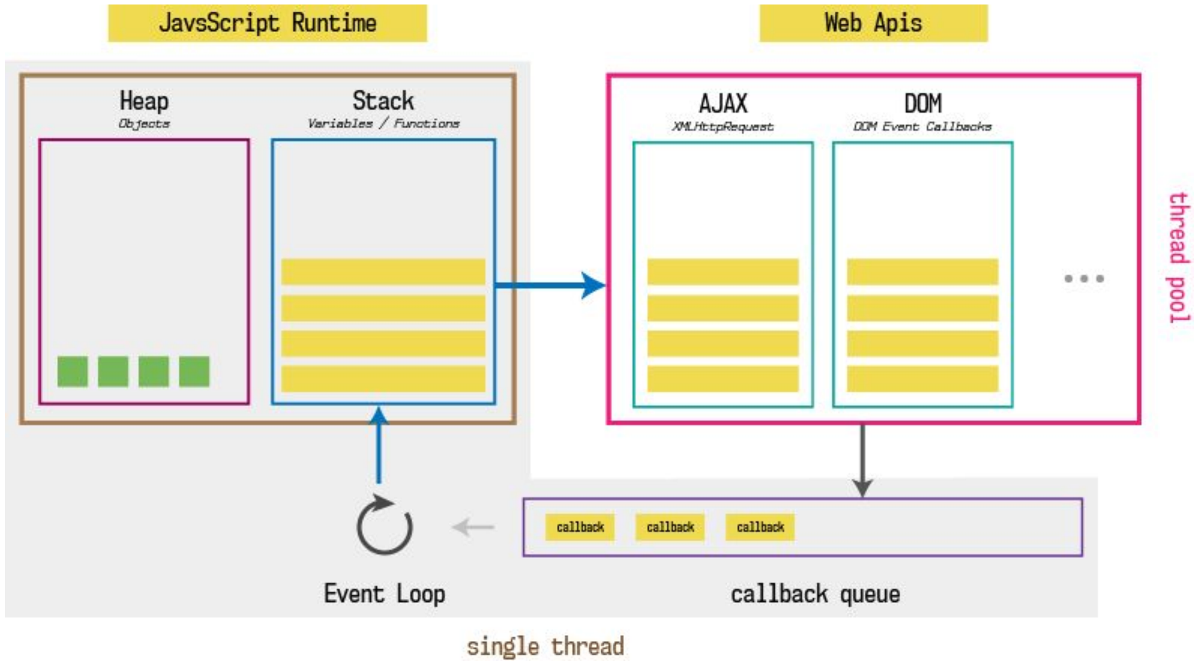
The DOM allows us to “react” to events happening on HTML elements or the page itself. This is achieved through event listeners that we can attach to any element, the document or the window. Some examples of such events:

- Clicking on a button - Example
- Typing in an input
- Selecting an option from a dropdown
- Resizing the browser window
- Dragging an element around

The above are all user initiated events, e.g triggered by a user interaction.

For a full list of available events refer to this documentation

Event Loop Explained



MENTORMATE™ DevCamp

Events - Full List HERE

Mouse Events

```
click
mouseover
mouseout
mousedown
mouseup
```

Touch Events

```
touchstart
touchend
touchmove
touchcancel
```

Form Events

```
input
change
submit
reset
```

Focus Events

```
focus
blur
```

Keyboard Events

```
keydown
keypress
keyup
```

Other Events

```
resize
drag events
Scroll events
load / unload
```

Events

The “Event” interface represents an event which takes place in the DOM. Events are fired when something happens in the DOM and we can attach “listeners” to execute some piece of code when a specific event occurs.

An example of attaching a “click” listener to a button:

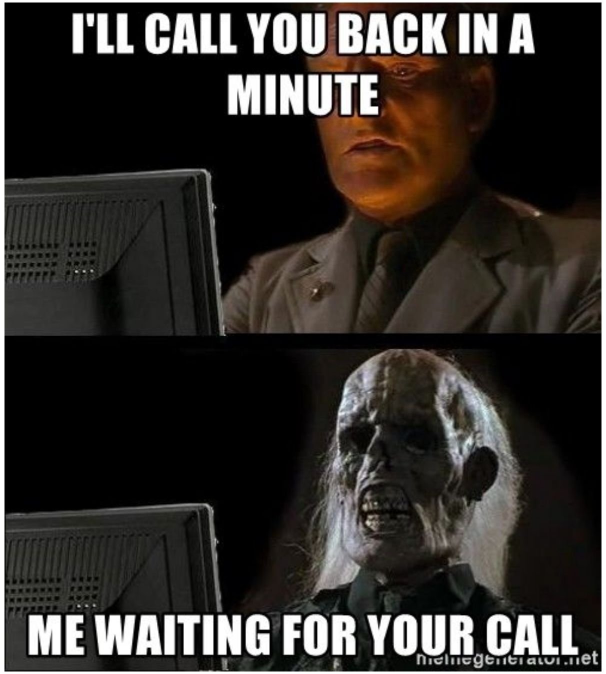
```
const button = document.getElementById('someId');

button.addEventListener('click', event => {
  console.log(`The ${event.target} button was clicked`);
});
```

Callbacks

- What are **callbacks** ?
  Usually callbacks are only used when doing I/O, e.g. downloading things, reading files, talking to databases, etc.

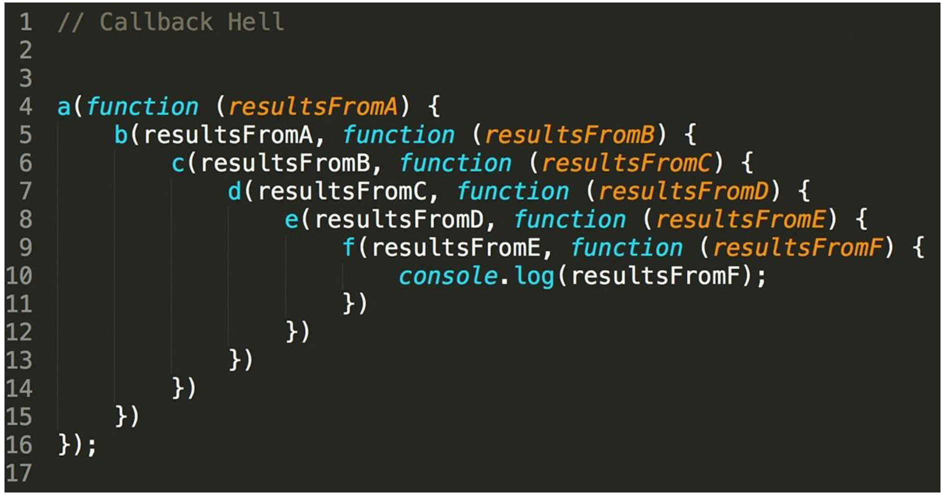
```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
  // photo is 'undefined'!
```

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```



Callback Hell

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this



```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});
```

How do I fix it?

- Keep your code shallow

```
document.querySelector('form').onsubmit = formSubmit;

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value

  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err;
  statusMessage.value = body;
}
```

How do I fix it?

- **Modularize**
- Comment your code
- Handle every single error
- Use advanced methods

```
module.exports.submit = formSubmit;

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value

  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err;
  statusMessage.value = body;
}
```

```
var formUploader = require('formuploader');
document.querySelector('form').onsubmit =
formUploader.submit;
```

```
const makeBurger = nextStep => {
  getBeef(function(beef) {
    cookBeef(beef, function(cookedBeef) {
      getBuns(function(buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger);
        });
      });
    });
  });
};
```

MENTORMATE™ DevCamp

Promises

```
const promise = new Promise((resolve, reject) => {
  /* Do something here */
});

const promise = new Promise((resolve, reject) => {
  // Note: only 1 param allowed
  return resolve(27);
  // return reject('💩💩💩');
})

// Parameter passed to resolve would be the arguments
passed into then.
promise.then(number => console.log(number)) // 27


// Parameter passed into reject would be the
arguments passed into catch.
promise.catch(err => console.log(err)) // 💩💩💩
```

- If `resolve` is called, the promise succeeds and continues into the `then` chain. The parameter you pass into `resolve` would be the argument in the next `then` call.
- If `reject` is called, the promise fails and continues into the `catch` chain. Likewise, the parameter you pass into `reject` would be the argument in the `catch` call.

Promises

```
const jeffBuysCake = cakeType => {
  return new Promise((resolve, reject) => {
    setTimeout(()=> {
      if (cakeType === 'black forest') {
        resolve('black forest cake!')
      } else {
        reject('No cake 😢')
      }
    }, 1000)
  })
}

const promise = jeffBuysCake('black forest');
console.log(promise);

const promise = jeffBuysCake('black forest')
  .then(cake => console.log(cake))
  .catch(nocake => console.log(nocake));
```

- Promises reduces the amount of nested code
- Promises allow you to visualize the execution flow easily
- Promises let you handle all errors at once at the end of the chain.

Promises

```
const friesPromise = getFries();
const burgerPromise = getBurger();
const drinkPromise = getDrink();

const eatMeal = Promise.all([
  friesPromise,
  burgerPromise,
  drinkPromise
])
  .then([fries, burger, drinks] => {
    console.log(`Chomp. Awesome ${burger}! 🍔`);
    console.log(`Chomp. Delicious ${fries}! 🍟`);
    console.log(`Slurp. Ugh, shitty drink ${drink} 🤢
`);
})
```

- Promise methods:
  - **`Promise.all(iterable)`**
  - **`Promise.allSettled()`**
  - **`Promise.race(iterable)`**
  - **`Promise.reject()`**
  - **`Promise.resolve()`**

Async/Await

- New keywords:
  - **`async`** – So the `async` keyword is added to functions to tell them to return a promise rather than directly returning the value
  - **`await`** – You can use `await` when calling any function that returns a Promise, including web API functions. `await` only works inside async functions!

You can use a synchronous `try...catch` structure with `async/await to handle errors and exceptions.`

MENTORMATE™
DevCamp

```
async function hungryExample(){

  const friesPromise = getFries();
  const burgerPromise = getBurger();
  const drinkPromise = getDrink();

  const eatMeal = Promise.all([
    friesPromise,
    burgerPromise,
    drinkPromise
  ]);

  [fries, burger, drink] = await eatMeal;

  console.log(`Chomp. Awesome ${burger}! 🍔`);
  console.log(`Chomp. Delicious ${fries}! 🍟`);
  console.log(`Slurp. Ugh, shitty drink ${drink} 🤢 `);

  async function getFries(){return ’fries’}
  async function getBurger(){return ’burger’}
  async function getDrink(){return ’kompot’}
}

hungryExample().then(console.log).catch(console.log);
```

Downsides of async/await

- `await` keyword blocks execution of all the code that follows until the promise fulfills
- your code could be slowed down by a significant number of awaited promises happening straight after one another

MENTORMATE™ DevCamp

Back to Events

Events

The “Event” interface represents an event which takes place in the DOM. Events are fired when something happens in the DOM and we can attach “listeners” to execute some piece of code when a specific event occurs.

An example of attaching a “click” listener to a button:

```
const button = document.getElementById('someId');

button.addEventListener('click', event => {
  console.log(`The ${event.target} button was clicked`);
});
```

Executing JS after the page has loaded

The **load** event is fired when the whole page has loaded, including all dependent resources such as stylesheets and images.

```
window.addEventListener('load', event => {
  console.log('the page is fully loaded');
});
```

If you need to execute a piece of code only after the page is fully loaded, use this approach

Anti-Pattern!

- Event handling JavaScript code can be specified in the HTML attributes onclick, onload, onmouseover, onresize, … but DON'T do that.

```
<button onclick="buttonClickFunction()">Click Me!</button>
<button onclick="alert('OK clicked')">OK</button>
```

```
<script>
 function buttonClickFunction() {
   console.log("You clicked the [Click Me!] button");
 }
</script>
```

The “event” Object

- The "event" object holds information about the event
  - Passed as parameter to the event handling function

```
domElement.addEventListener(eventType, eventHandler, isCaptureEvent)
```

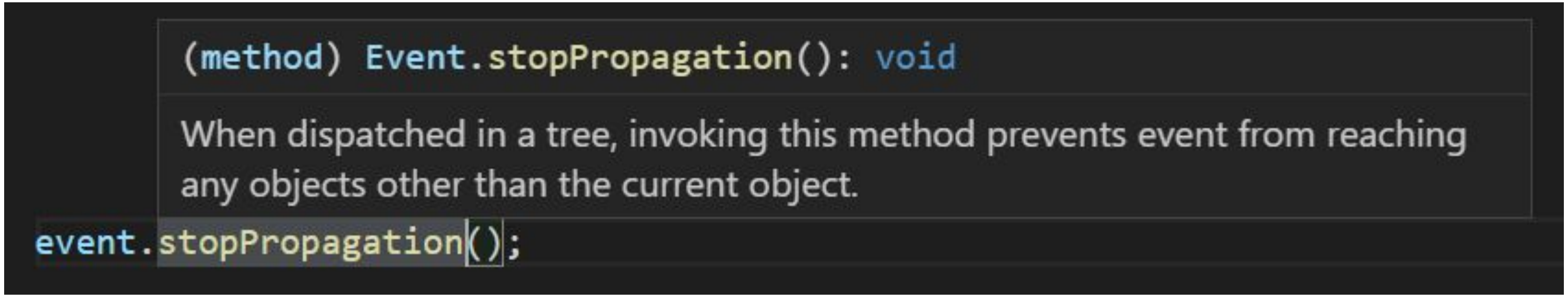
- The event object contains information about:
  - The type of the event (e.g. 'click', 'resize', ...)
  - The target of the event (e.g. the button clicked)
  - The key pressed for keyboard events
  - The mouse button / cursor position for mouse events

Event Bubbling

The bubbling principle in a nutshell:

When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

To stop an event from bubbling you can use the **stopPropagation()** API.



```
(method) Event.stopPropagation(): void
```

When dispatched in a tree, invoking this method prevents event from reaching any objects other than the current object.

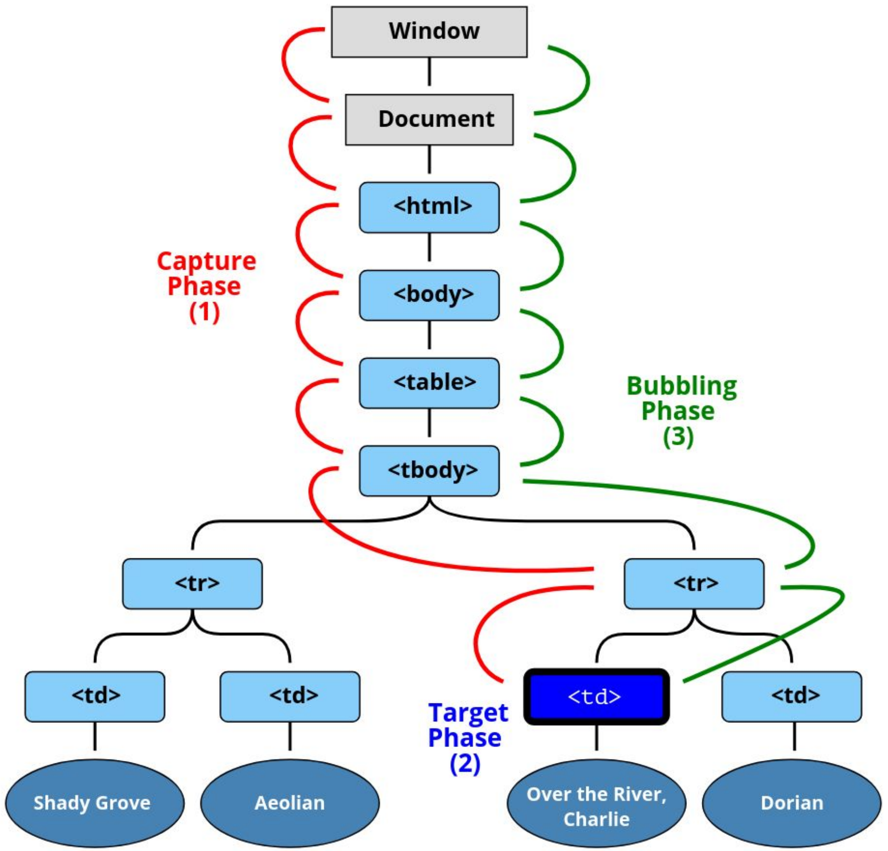
```
event.stopPropagation();
```

MENTORMATE™ DevCamp

Event Capturing

Event capturing, although rarely used directly is still important to be aware of. It’s the opposite of event bubbling:

The event is first captured by the outermost element and propagated to the inner elements.



MENTORMATE™ DevCamp

The Event Interface

- `Event.bubbles`
  A boolean indicating whether or not the event bubbles up through the DOM.
- `Event.cancelable`
  A boolean indicating whether the event is cancelable.
- `Event.currentTarget`
  A reference to the currently registered target for the event. This is the object to which the event is currently slated to be sent.
- `Event.target`
  A reference to the target to which the event was originally dispatched.

For a full reference to the Event API visit the documentation

Custom Events

- Using the JavaScript API we can create our own events, via the **CustomEvent** class.
- To create a custom event you need to call the constructor as follows:

```
new CustomEvent(name, [customEventInitParams]);
```

- After we create the event we need to add a listener that listens for that event.
- Finally we dispatch/trigger the event when needed.

```
var customEv = new CustomEvent('yell');

elem.addEventListener('yell', event => { … });
elem.dispatchEvent(customEv);
```

AJAX

MENTORMATE™
DevCamp

AJAX Introduction

- AJAX is a developer's dream, because you can:
  - `Read data from a web server - after the page has loaded`
  - `Update a web page without reloading the page`
  - `Send data to a web server - in the background`
- What is AJAX?
  - **AJAX = Asynchronous JavaScript And XML** (and JSON too!)
  - AJAX is not a programming language.
  - AJAX just uses a combination of:
    - A browser built-in `XMLHttpRequest` object
    - JavaScript and HTML DOM (to display or use the data)

AJAX Introduction

```
function loadDoc() {
  var xhttp = new XMLHttpRequest();

  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
    }
  };

  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
```

Read about CORS

XMLHttpRequest Object Methods documentation

MENTORMATE™ DevCamp

Fetch API

Introduction to Fetch API

- The Fetch API is a **promise-based JavaScript API** for making **asynchronous HTTP requests** in the browser similar to XMLHttpRequest (XHR).

- Unlike XHR, it is a simple and clean API that uses promises to provides a more powerful and flexible feature set to fetch resources from the server.

- Fetch is pretty much **standardized now** and is supported by all modern browsers except IE. If you need to support all browsers including IE, just add a polyfill released by GitHub to your project.

Introduction to Fetch API

- Just pass the URL, the path to the resource you want to fetch, to fetch() method

```
fetch('https://reqres.in/api/users')
  .then(res => res.json())
  .then(res => {
     res.data.map(user => {
       console.log(`${user.id}: ${user.first_name} ${user.last_name}`);
     });
  });

// 1: George Bluth
// 2: Janet Weaver
// 3: Emma Wong
```

The Window Object

- The window object is supported by all browsers. It represents the browser's window.
- All global JavaScript objects, functions, and variables automatically become members of the window object.
- Global variables are properties of the window object.
- Global functions are methods of the window object.
- Even the document object (of the HTML DOM) is a property of the window object

Useful APIs

- **window.location** can be used to get the current page address (URL) and to redirect the browser to a new page.
- **window.history** contains the browsers history.
  - `history.back()`
  - `history.forward()`
- **window.navigator** contains information about the visitor's browser
- **timers** are also attached to the **window** object
  - `setTimeout`
  - `setInterval`

Persisting Data

- window.localStorage
  Allows us to save key/value pairs in a web browser. Stores the data with no expiration date.

- window.sessionStorage
  Allows us to save key/value pairs in a web browser. Stores the data for one session.

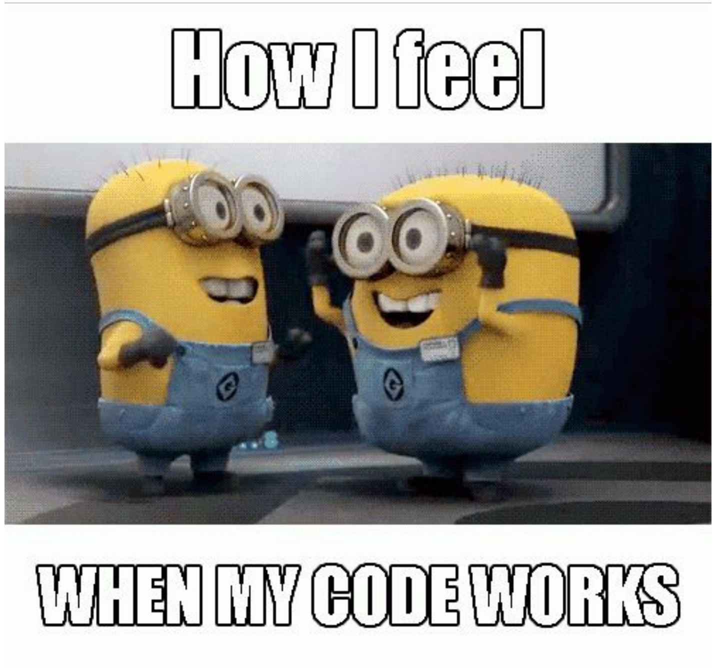
MENTORMATE™
DevCamp

Resources

- MDN Docs
- W3C - Simple Tutorials

More tutorials

1. https://javascript.info/document
2. https://javascript.info/events
3. https://javascript.info/event-details
4. https://javascript.info/forms-controls
5. https://javascript.info/loading
6. https://javascript.info/event-loop



Homework

Final words

- Be curious! Experiment! Do more. **Keep practicing!**
  - Challenge yourself on HackerRank, Exercism, TopCoder, etc…
- **Documentation** is the best source for learning. Read it!
- **Read** You Don't Know JS! book.
- Draw **inspiration** from Awesome JavaScript.
- **Stay up-to-date!** Subscribe to JavaScript Newsletters (JavaScript Weekly).
- **Learn other languages** with other paradigms.
- **Visit IT conferences, Volunteer & Mentor, Contribute to OpenSource**.