Before we begin...

- Videos On!

Welcome

Agenda

- JavaScript and the Browser
  - More Events
  - More Animations
  - Bubbling and Capturing
  - Event Propagation
  - Event Delegation
  - Preventing Default Behaviour

Review

- Browser Internals
- JavaScript and the Browser
  - The Document Object Model (D.O.M.)
  - DOM Selectors and Traversal
  - Creating DOM Nodes
  - Events
  - Animations

Events

Some Terminology

- **Event**
  - Something that occurs
- **Callback**
  - A function that executes after the event has happened
- **Event Listener**
  - A method that binds an event to a callback

Events with JavaScript

There are three important things:

- The **event target** (usually a DOM Node) that is going to be interacted with (`body`, `h1` etc.)
- The **event type** (click, hover, scroll etc.)
- The **callback function** (which acts in response to the event)

We need to bind these together to create an **event listener**

Events Pseudocode - Dark/Light Mode

```
WHEN the element with ID of toggleMode is CLICKED
    SELECT the body tag and save as body
    CHANGE the body CSS to have a "black" background

WHEN the element with ID of toggleMode is CLICKED
    SELECT the body tag and save as body
    STORE the currentBackground of body

    IF currentBackground === "black"
        CHANGE the body CSS to have a "white" background

    ELSE
        CHANGE the body CSS to have a "black" background
```

`node.addEventListener`

The Basic Process

- Find the DOM Node using a DOM access method
- Create a callback function
- Create the event listener

node.addEventListener

```
let myButton = document.querySelector("button");

function myCallback() {
  console.log("The button was clicked");
}

myButton.addEventListener("click", myCallback);
```

node.addEventListener

```
let myButton = document.querySelector("button");

// You can also use anonymous functions
myButton.addEventListener("click", function () {
  console.log("button clicked!");
});
```

node.removeEventListener

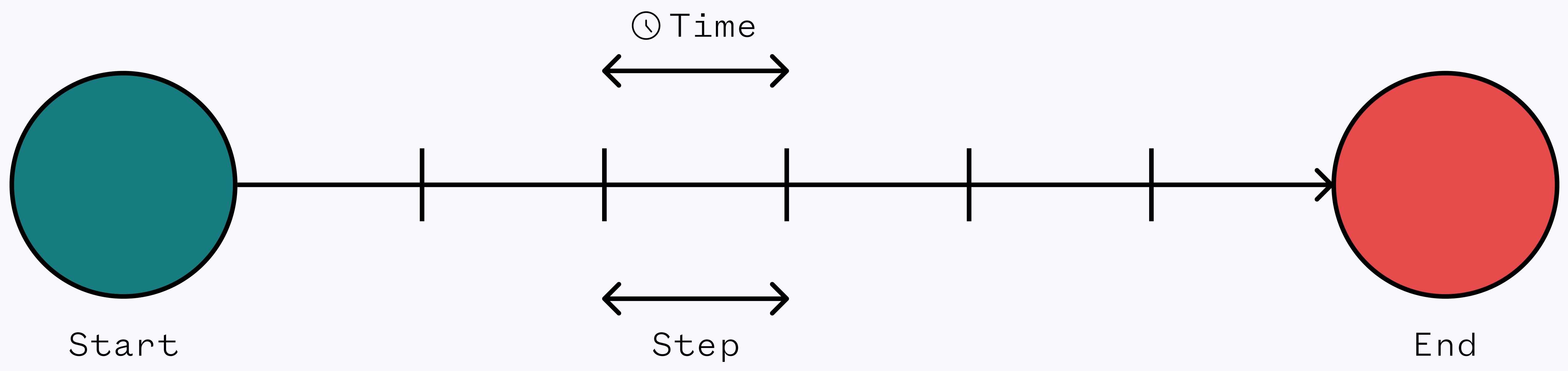
```javascript
let myButton = document.querySelector("button");

function myCallback() {
  console.log("The button was clicked");
}

myButton.addEventListener("click", myCallback);

// Later on...
myButton.removeEventListener("click", myCallback);
```

What events are there?

Given that an event is a signal that something has taken place, there are lots of different events occurring all of the time. Some of them are:

- Mouse Events (click, contextmenu, mouseover/mouseout, mousedown/mouseup, mousemove etc.)
- Keyboard Events (keydown, keyup etc.)
- Browser Events (submit, focus etc.)
- Form Events (DOMContentLoaded etc.)
- Window Events (scroll etc.)

Animations


Time
Start
Step
End

Animations

Things you need to define:

- Starting Point
- Step
- Time Between Steps
- Total Time
- Ending Point

Fade Out: Pseudocode

```
SELECT and STORE the image as myImg

CREATE a function called fadeImgAway
  GET the current opacity and store as currentOpacityAsString
  GET the current opacity as a number and store as currentOpacity

  CREATE newOpacity by subtracting 0.01 from currentOpacity
  UPDATE myImg opacity to be newOpacity

  IF the currentOpacity is >= 0
    CALL fadeImgAway in 10ms

CALL fadeImgAway to start the animation
```

Fade Out: Pseudocode

```
let img = document.querySelector("img");

function fadeImgAway() {
  let currentOpacityAsString = getComputedStyle(img).opacity;
  let currentOpacity = parseFloat(currentOpacityAsString, 10);
  let newOpacity = (currentOpacity -= 0.01);
  img.style.opacity = newOpacity;
  if (currentOpacity >= 0) {
    setTimeout(fadeImgAway, 10);
  }
}

setTimeout(fadeImgAway, 1000);
```

Advanced Events

event

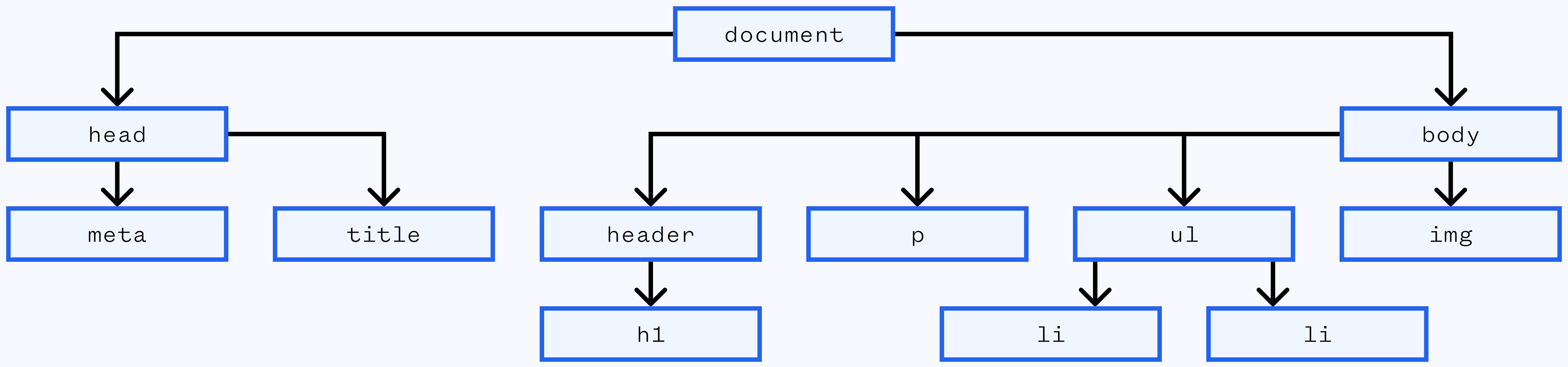
The `event` parameter

- When an event takes place and the callback runs - JavaScript provides us with some information as an object!
- We receive this in our callback function (as a parameter - it can be called whatever you want, but is often called `event` or `e`)
- This parameter contains all sorts of things, what element was interacted with, the type of event, where the mouse was, what keys were pressed etc.

The `event` parameter

```
let div = document.querySelector("div");

function myCallback(event) {
  console.log(event);
}

div.addEventListener("click", myCallback);
```

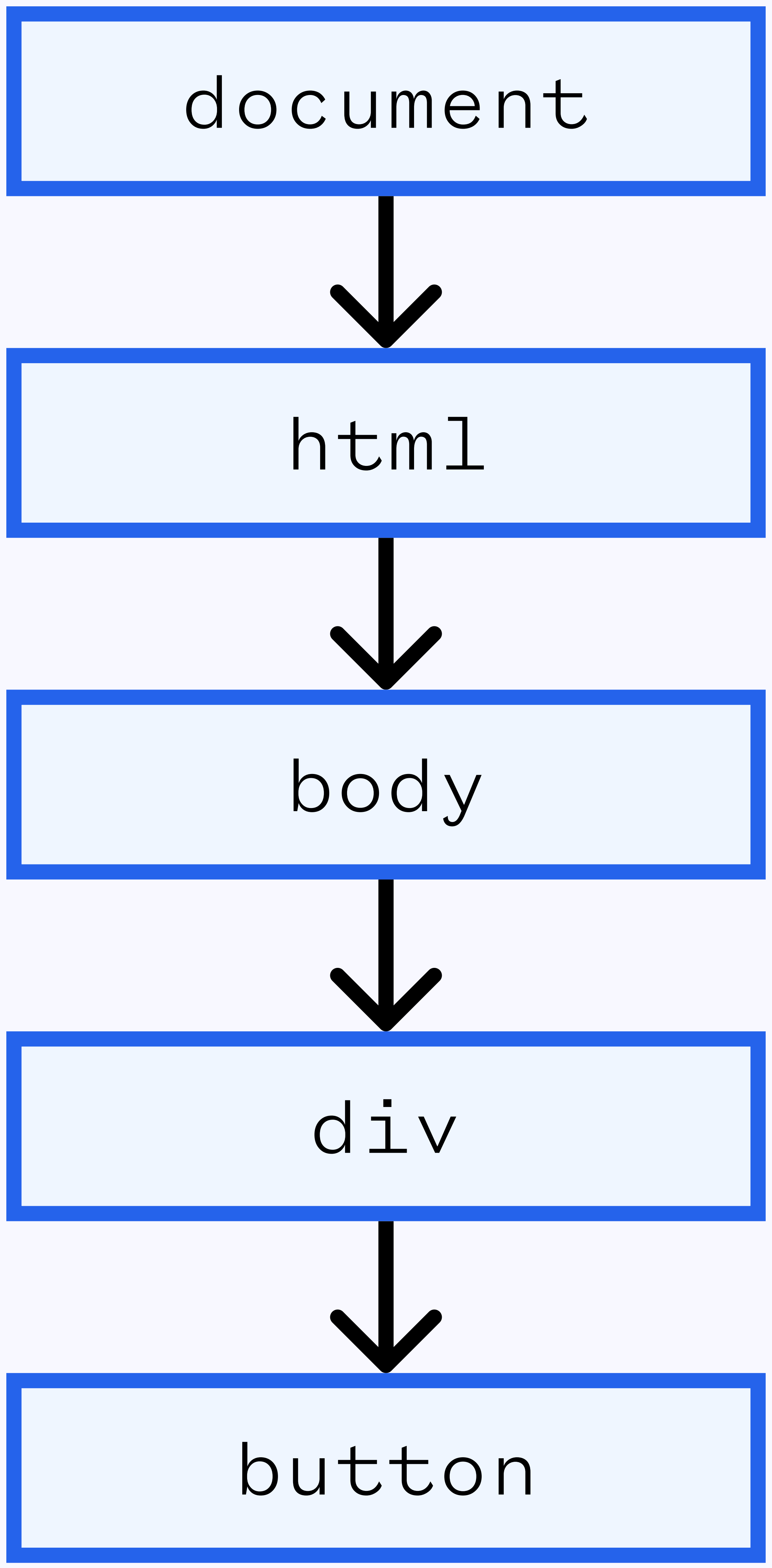
Bubbling and Capturing

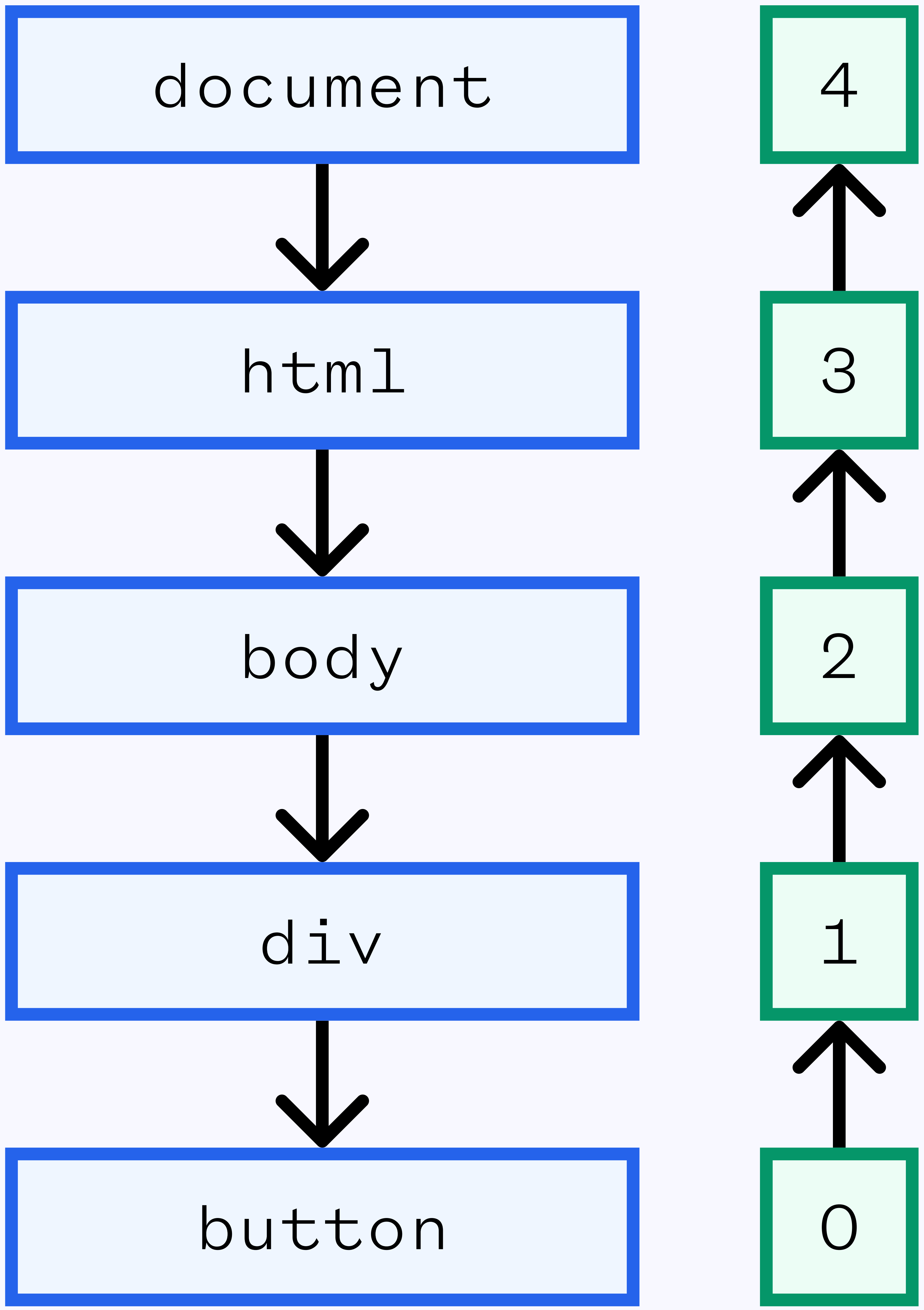

document
head
body
meta
title
header
p
ul
img
h1
li
li

How do `event`s work?

When you click an element, a process called **propagation** or **bubbling** occurs.

- The event listeners are run on the element itself
- Then, the event listeners are run on the element's parents
- Then, the event listeners are run on the element's grandparents
- And so on


document
html
body
div
button


document
html
body
div
button
4
3
2
1
0

```
<!DOCTYPE html>
<html>
  <head>
    <title>Events</title>
  </head>
  <body>
    <ul>
      <li>My List Item</li>
    </ul>
  </body>
</html>
```

```javascript
let ul = document.querySelector("ul");
let li = document.querySelector("li");

document.body.addEventListener("click", function () {
  console.log("The body was clicked");
});

ul.addEventListener("click", function () {
  console.log("The ul was clicked");
});

li.addEventListener("click", function () {
  console.log("The li was clicked");
});
```

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

```javascript
let ul = document.querySelector("ul");
let li = document.querySelector("li");

document.body.addEventListener("click", function () {
  console.log("This won't run");
});

ul.addEventListener("click", function () {
  console.log("This won't run");
});

li.addEventListener("click", function (event) {
  event.stopPropagation(); // This stops the bubbling process!
  console.log("This will run");
});
```

A Warning

Bubbling is a very convenient process, and without good reason - stopping it can cause significant problems.

The classic example of this is with things such as Google Analytics - they often will track clicks by adding events to the document itself. Stopping propagation could potentially stop those clicks being recorded.

Imagine for a second...

We know that events have to be added one element at a time. What happens if we had...

- A massive list, and we needed events for every item in it?
- A table, and we needed events for every cell in that table?
- A social meed feed, and we needed events for every post?

Event Delegation

So, what is event delegation?

Event delegation is a powerful pattern that came from the fact that adding events to lots of elements requires lots of code, and can cause performance issues under certain circumstances.

If lots of element's events should be handled in a similar fashion, we just attach one event listener to their common parent.

The Process

- Put a single event listener on a parent DOM node
- In your callback function, check the DOM node that was interacted with using `event.target`
- If that is a DOM node that interests us
  - Handle the event however necessary

By the way - if you stop bubbling using `event.stopPropagation()`, this won't work!

Show me the code

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Events</title>
  </head>
  <body>
    <div>
      <p>Click Me 1</p>
      <p>Click Me 2</p>
      <p>Click Me 3</p>
      <p>Click Me 4</p>
      <h1>Ignore clicks here!</h1>
    </div>
  </body>
</html>
```

```javascript
let div = document.querySelector("div");

function myCallback(event) {
  let target = event.target;
  if (target.tagName === "P") {
    target.style.color = "red";
  }
}

div.addEventListener("click", myCallback);
```

```
event.preventDefault()
```

Sensible Defaults

The Browser has sensible defaults. When certain events take place, they automatically lead to certain actions. For example:

- When someone clicks on a link, they are navigated to the link's `"href"`
- When someone presses their mouse over a piece of text and then moves it, it'll highlight that text
- When someone right clicks, it will show the context menu

```javascript
let a = document.querySelector("ul");

a.addEventListener("click", function (event) {
  event.preventDefault();
  alert("Not letting you do that!");
});
```

Page Events

The whole page has events too

Websites have life-cycles, broken down into 4 main events:

- **`"DOMContentLoaded"`** - the DOM is ready, but things like `img`s haven't been downloaded yet
- **`"load"`** - Similar to `"DOMContentLoaded"` but external resources have been loaded too
- **`"beforeunload"`** - The user is about to leave the page, we can try to get them to stay or perform some action!
- **`"unload"`** - The user is leaving, we can perform last minute things (like analytics)

DOMContentLoaded

```
function pageReady() {
  console.log("The DOM is ready to go!");
}

document.addEventListener("DOMContentLoaded", pageReady);
```

load

```
function everythingLoaded() {
  console.log("Everything has been loaded and downloaded");
}

window.addEventListener("load", everythingLoaded);
```

unload

```
function aboutToClose() {
  console.log("The page is being shut down. Do last minute things!");
}

window.addEventListener("unload", aboutToClose);
```

beforeunload

```
function beforeClose(event) {
  event.preventDefault();
  return "There are unsaved changes. Do you still want to leave?";
}

window.addEventListener("beforeunload", beforeClose);
```

Templating

Creating Strings

- There are three ways to create strings
  - Single quotes (e.g. `'Hello'`)
  - Double quotes (e.g. `"Hello"`)
  - Backticks (e.g. ``` `Hello` ```)
- Single quotes and double quotes are mostly interchangeable but *backticks are special*!

Backticks

Backticks are special because they allow:

- Strings that are multiple lines long
- Interpolation

Backticks are often called **template literals**

Interpolation

- The process of inserting a value into a string
  - Almost like substitution
- We can run any JavaScript code with interpolation
  - We can call functions or methods
  - Access properties
  - Perform mathematical operations
  - Anything!

```
let fancyString = `This is a string`;

let favNumber = 42;
let message = `My favourite number is ${favNumber}`;

let maths = `4 * 2 = ${4 * 2}`;

let name = `Douglas Hofstadter`;
let greeting = `Hello ${name.toUpperCase()}`;
```

```javascript
const course = {
  name: "JavaScript Development",
  provider: "General Assembly",
  topics: ["JavaScript", "APIs", "React"],
  numberOfHours: 60,
};

const markup = `
  <div>
    <h2>${course.name}</h2>
    <h4>Provided by ${course.provider}</h4>
    <h5>Course time: ${course.numberOfHours}</h5>
    <p>This course covers: ${course.topics.join(", ")}</p>
  </div>
`;

console.log(markup);
```

Start the exercises here, please!

See you in 25 minutes

That's all for tonight!

Homework

- Finish off the following exercises:
  - The Glossary Exercise
- Go through as much of JavaScript.info's Browser: Documents, Events and Interfaces section as you can

Homework: Extra

- Watch this course on DOM Events and go through DOM Events
- Work on your CSS Selectors using FlukeOut

What's Next?

- Review
- Methods
- Functional JavaScript
- `this`
- Classes
- Inheritance
  - Classical vs. Prototypal

Thank You!