

# Before we begin...

Open  <https://bit.ly/jsd-dom-2>

Zoom  Videos On

Welcome!

# Agenda

- Review and Homework Recap
- JavaScript and the Browser
  - More events and animations
  - Bubbling and capturing
  - Event propagation
  - Event delegation
  - Preventing default behaviour
  - Templating

Review

# Events

# Some Terminology

- **Event**: something that happens
- **Callback**: a function that executes after the event has happened
- **Event Listener**: a method that binds an event to a callback

# Events with JavaScript

- Three important things:
  - **The DOM Node** that is going to be interacted with (body, h1, p etc.)
  - **The event type** (click, hover, scroll etc.)
  - **The response** (often called *the callback* - a function!)

# Events Pseudocode



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

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

    IF currentBackground === "hotpink"
        CHANGE the body CSS to have a ghostwhite background

    ELSE
        CHANGE the body CSS to have a hotpink background
```



# node.addEventListener



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

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

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

The basic process: find the DOM Node using a selector method, create a callback function and then create the event listener (using the DOM Node, an Event Type and the callback function)

# node.removeEventListener



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

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

myButton.addEventListener("click", myCallback);

// Later on...

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

# Anonymous Functions



```
const myButton = document.querySelector("button");  
  
myButton.addEventListener("click", function() {  
  console.log("button clicked!");  
});
```

I don't typically suggest following this approach. You can't ever remove this event handler plus it's harder to debug!

Aim for extensibility and ease of debugging every time.

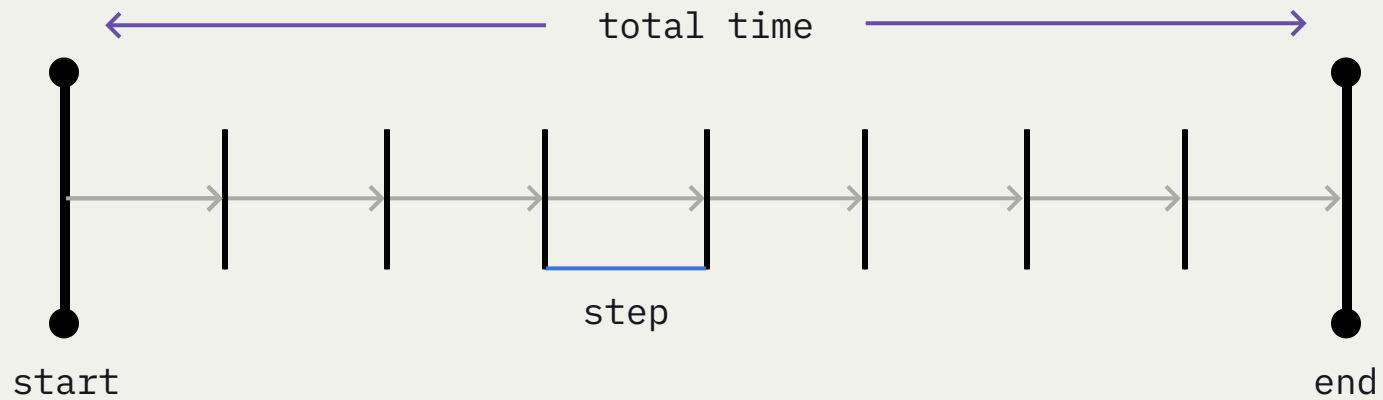
# What events are there?

Given that an event is a signal that something has taken place, there are lots of different events occurring all of the time. We always create event listeners in the same way!

- 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

# Animations



# Animations

Things you need to define:

1. Starting Point
2. Step
3. Time between steps
4. Total time
5. 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  $\geq 0$ 
```

```
    CALL fadeImgAway in 10ms
```

```
CALL fadeImgAway to start the animation
```



# Fade Away



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

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

setTimeout(fadeImgAway, 1000);
```

# Advanced Events

event

# The event parameter

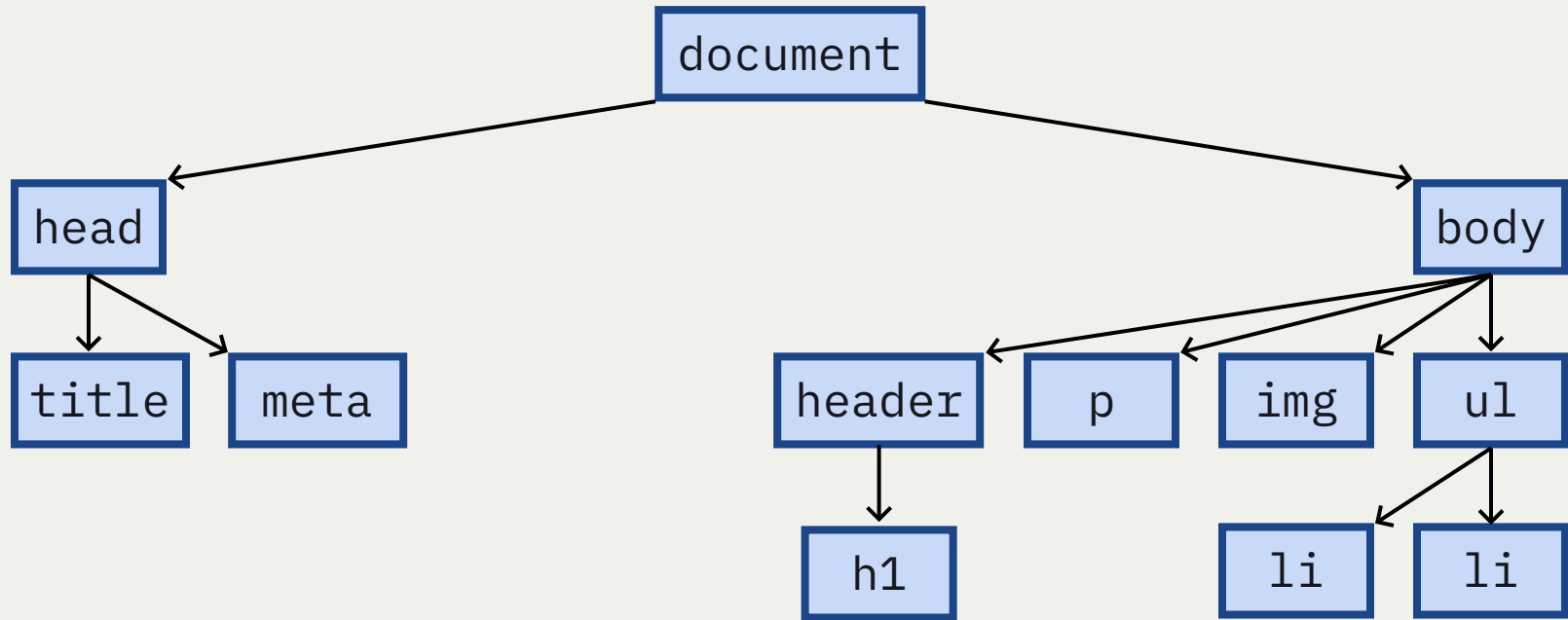
- When an event takes place, and the corresponding listener runs - JavaScript provides us with some information as an object!
  - We can receive this in our callback function (as a parameter - it can be called whatever you want)
- The event parameter contains all sorts of things, what element was interacted with, the type of event, where the mouse was, what keys were pressed etc.
  - Let's take a look!

# The event parameter

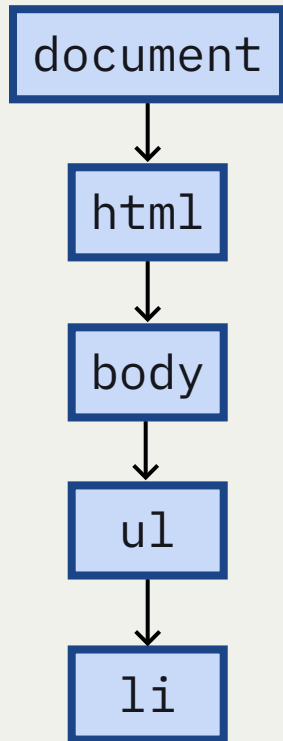
```
● ● ●  
  
const div = document.querySelector("div");  
  
function myCallback(event) {  
  console.log(event);  
  
  // You may be interested in things like:  
  // - .type  
  // - .currentTarget  
  // - .screenX  
  // - .screenY  
}  
  
div.addEventListener("click", myCallback);
```

# Bubbling and Capturing

# A quick recap: The DOM Tree



# How do events work?



When you click on the `li`, a process called propagation, or bubbling, occurs.

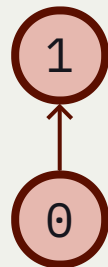
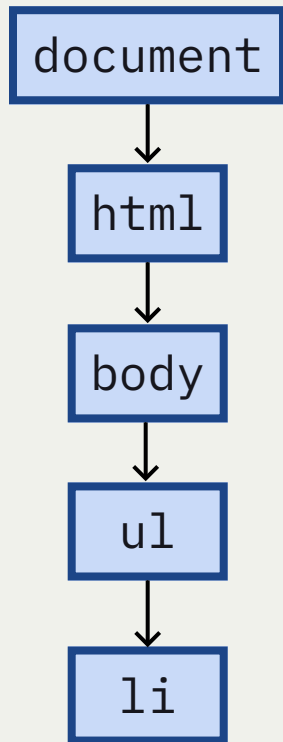
When an event, the listeners are run on the element, then its parent's listeners are run, then its grandparent's listeners are run - all the way to the top of the DOM tree.

So, the `li`'s events would run, then the `ul`'s event would run, then `body` etc.





# How do events work?

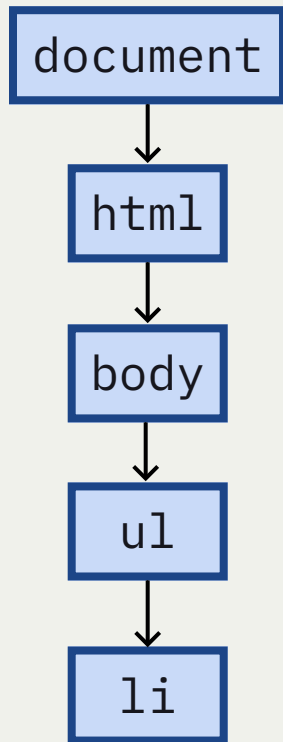


When you click on the `li`, a process called propagation, or bubbling, occurs.

When an event, the listeners are run on the element, then its parent's listeners are run, then its grandparent's listeners are run - all the way to the top of the DOM tree.

So, the `li`'s events would run, then the `ul`'s event would run, then `body` etc.

# How do events work?

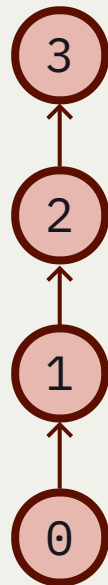
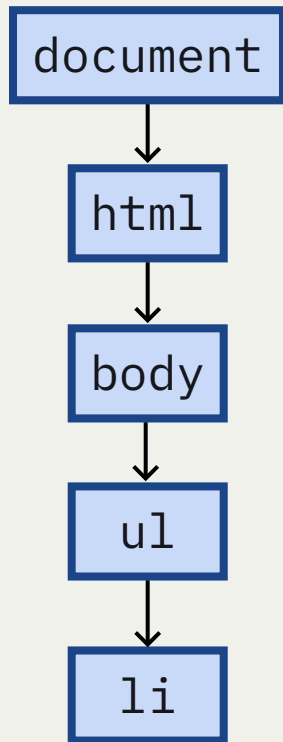


When you click on the `li`, a process called propagation, or bubbling, occurs.

When an event, the listeners are run on the element, then its parent's listeners are run, then its grandparent's listeners are run - all the way to the top of the DOM tree.

So, the `li`'s events would run, then the `ul`'s event would run, then `body` etc.

# How do events work?

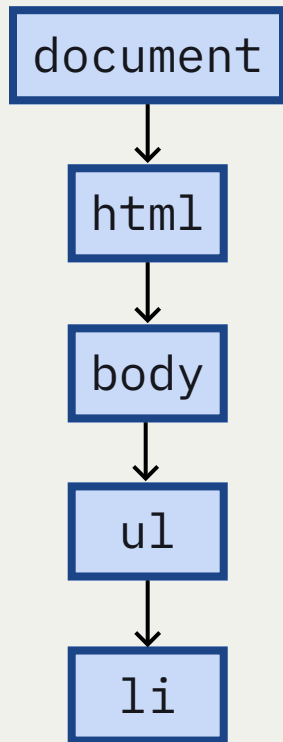


When you click on the li, a process called propagation, or bubbling, occurs.

When an event, the listeners are run on the element, then its parent's listeners are run, then its grandparent's listeners are run - all the way to the top of the DOM tree.

So, the **li**'s events would run, then the **ul**'s event would run, then **body** etc.

# How do events work?

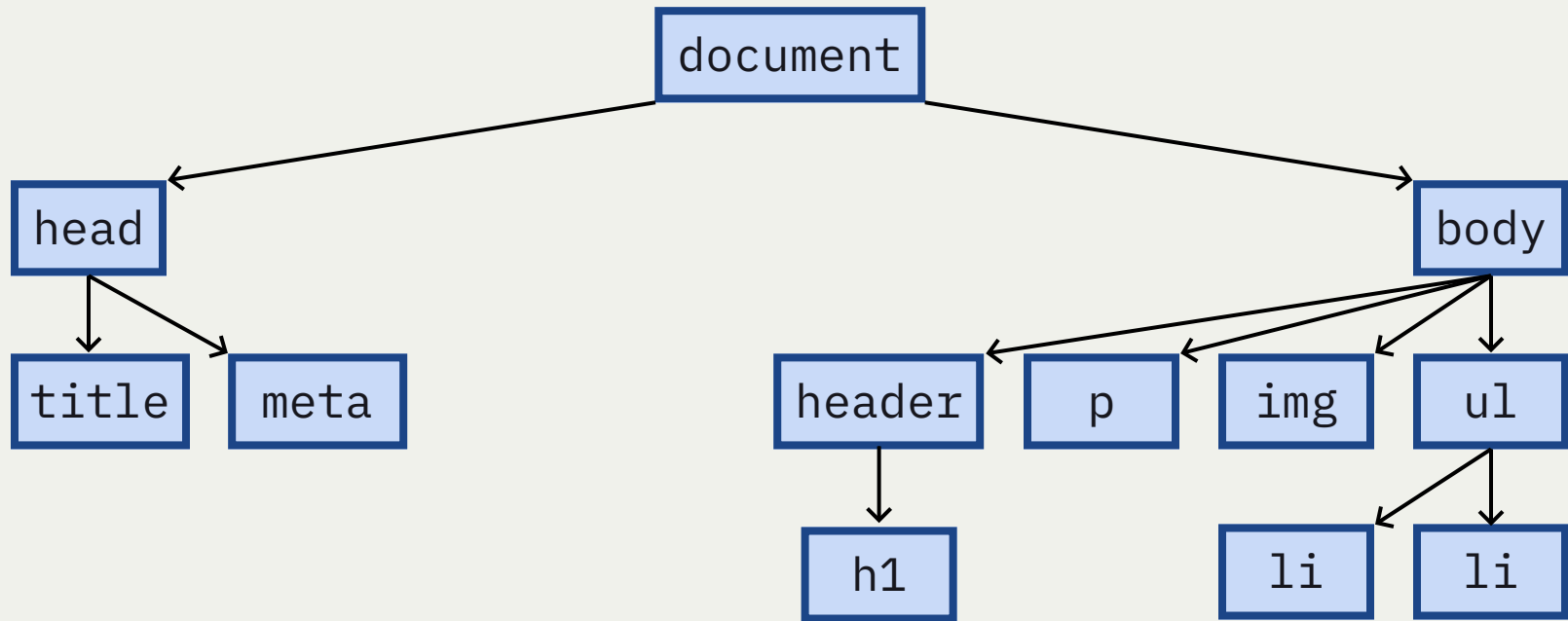


When you click on the `li`, a process called **propagation**, or **bubbling**, occurs.

When an event, the listeners are run on the element, then its parent's listeners are run, then its grandparent's listeners are run - all the way to the top of the DOM tree.


So, the `li`'s events would run, then the `ul`'s event would run, then `body` etc.

# Walk me through it, please!




So the user just clicked the h1, what happens next?

# Show me the code!



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



```
const ul = document.querySelector("ul");
const 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();



```
1 const ul = document.querySelector("ul");
2 const li = document.querySelector("li");
3
4 document.body.addEventListener("click", function() {
5   console.log("This won't run");
6 });
7
8 ul.addEventListener("click", function() {
9   console.log("This won't run");
10 });
11
12 li.addEventListener("click", function(event) {
13   event.stopPropagation(); // This stops the bubbling process!
14   console.log("This will run");
15 });
```

# Watch Out!

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.



# 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.

# Examples of where it could help

- Working with tables
  - Each cell has similar functionality
- Working with a feed of information
  - Each section might have similar functionality (e.g. Instagram, Twitter etc.)
- These slides
  - The way I change slides is consistent across the whole presentation
- Plus many more...

# 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!

```

<!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>

```

```

const div = document.querySelector("div");

div.addEventListener("click", function(event) {
  let target = event.target;
  if (target.tagName === "P") {
    target.style.color = "red";
  }
});

```

```
event.preventDefault();
```

# Defaults

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

- When someone clicks on a link, they can 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

# Show me the code!



```
<a href="http://endless.horse/">  
  Don't follow this link!  
</a>
```



```
1 const a = document.querySelector("ul");  
2  
3 a.addEventListener("click", function(event) {  
4   event.preventDefault();  
5   alert("Not letting you do that!");  
6 });
```



# Page Events

# The whole page has events too

Websites have lifecycles, broken down into 4 main events:

- DOMContentLoaded - the DOM is ready, but things like `imgs` 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);
```

# window.onload



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

# window.onunload



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

# window.onbeforeunload



```
function beforeClose() {  
  console.log("The page is about to be shut down.");  
  return "There are unsaved changes. Do you still want to leave?";  
}  
  
window.addEventListener("beforeunload", beforeClose);
```

# Templating

# Strings

- There are three ways to create strings
  - Single quotes
  - Double quotes
  - Backticks (normally above the TAB key)
- Single quotes and double quotes are mostly interchangeable but backticks are special! They allow:
  - Strings that are multiple lines long
  - And interpolation
- Backtick strings are 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!

# Template Literals



```
const fancyString = `This is a string`;  
  
const favNumber = 42;  
  
const message = `My favourite number is ${favNumber}`;  
  
const maths = `4 * 2 = ${4 * 2}`;  
  
const name = `Douglas Hofstadter`;  
  
const greeting = `Hello ${name.toUpperCase()}`;
```

# Advanced Template Literals

```
const course = {
  name: "JavaScript Development",
  provider: "General Assembly",
  topics: ["JavaScript", "React", "Firebase"],
  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);
```

SOLO!

Do the lab here, please!

See you in 25 minutes!

Review

That's all!

# Homework

- Any previous homework
- The Interactive Glossary Lab
  - Feel free to work on this with a classmate!
- Extra: Begin reviewing the next lesson's content

# Homework

- Watch [Umar Hansa's Browser Rendering Talk](#)
- Watch [Jake Archibald's In The Loop](#)
- Go through [The Modern JavaScript Tutorial](#)
- Read [Eloquent JavaScript](#)
- Read [Speaking JavaScript](#)



# What's next?

- Pseudocode
- Advanced Functions
  - Callbacks
  - Scope and Hoisting
  - Closures
  - Higher Order Functions
  - Rest Parameters
  - Spread Operator

Thank you!