# Lecture 3

JavaScript (2)

DOM Manipulation & JS Timers

## Today’s Contents

- More about event handling
- JS File Skeleton
- DOM Manipulation
    - Classes
    - Nodes
- Anonymous functions, callbacks, and `this`
- Debugging JS
- JS Timers

## JS File Skeleton

- JavaScript “strict” mode
- The “module pattern”
- Visualization of how a DOM tree is parsed and built by the browser
- The window “load” event

# JavaScript “strict” mode

```
"use strict";
// your code
```

Writing `"use strict";` at the very top of your JS file turns on strict syntax checking:

- Shows an error if you try to assign to an undeclared variable
- Stops you from overwriting key JS system libraries
- Forbids some unsafe or error-prone language features

You should **always** turn on strict mode for your code!

# The “module pattern”

```
(function() {
   // statements;
})();
```

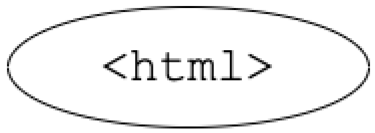
Wraps all of your file's code in an anonymous function that is declared and immediately called.

`0` global symbols will be introduced!

The variables and functions defined by your code cannot be accessed/modified externally (i.e. by other JS scripts).

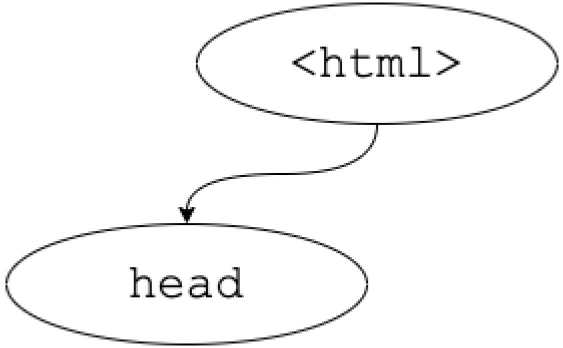
You should use this pattern for all of your JS files.

# How the browser builds a DOM Tree



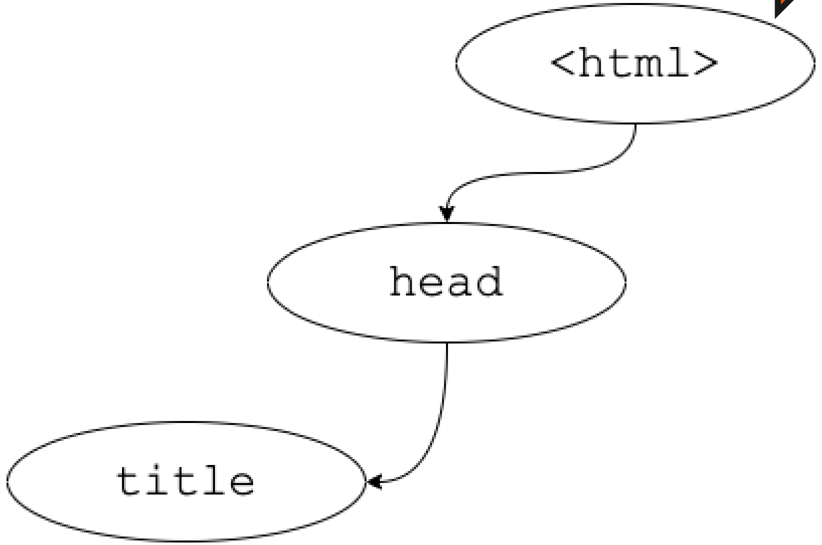
```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

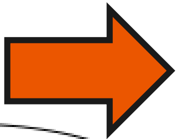
## How the browser builds a DOM Tree



```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

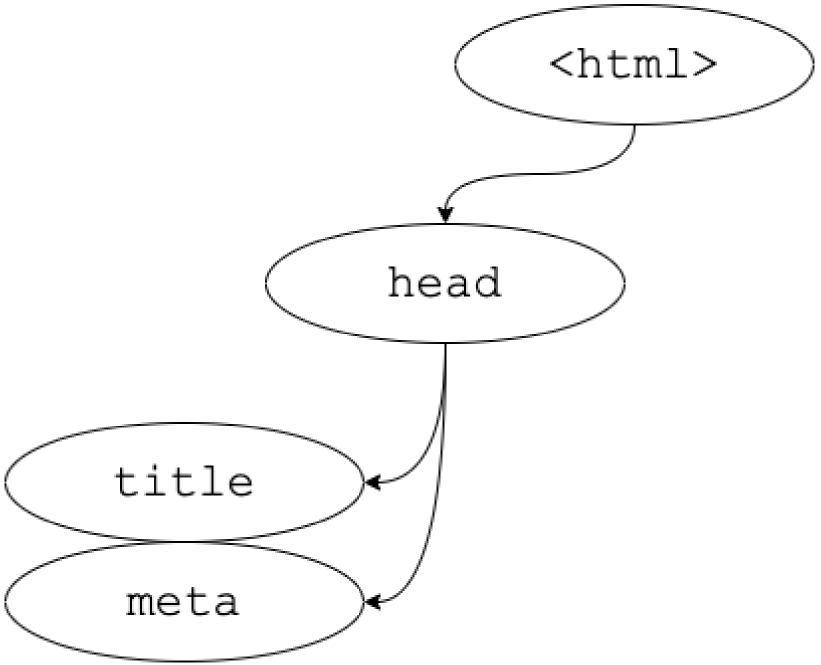
# How the browser builds a DOM Tree





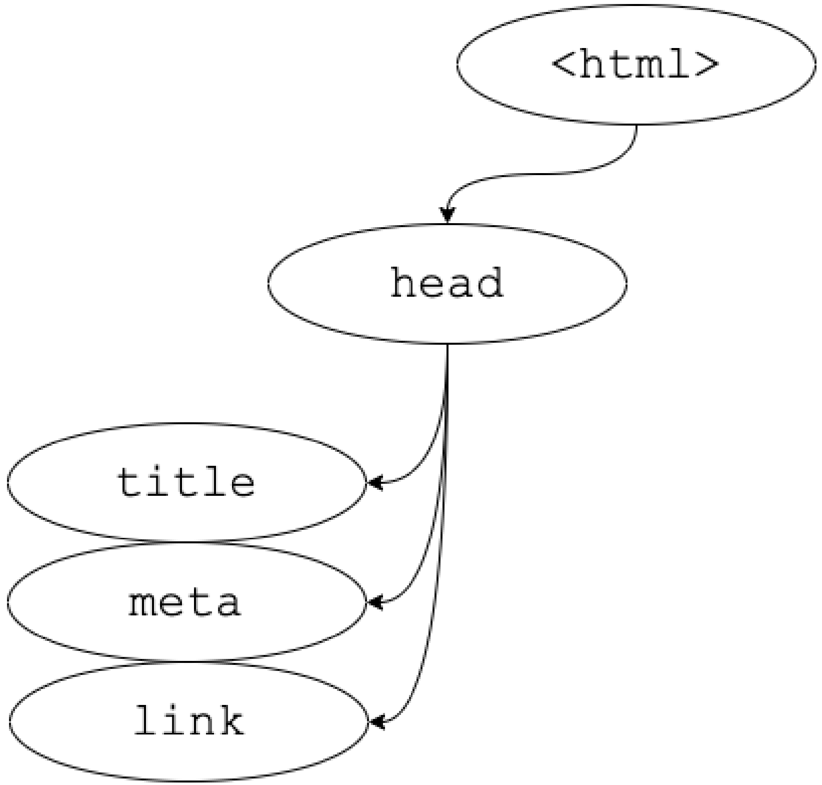
```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

# How the browser builds a DOM Tree



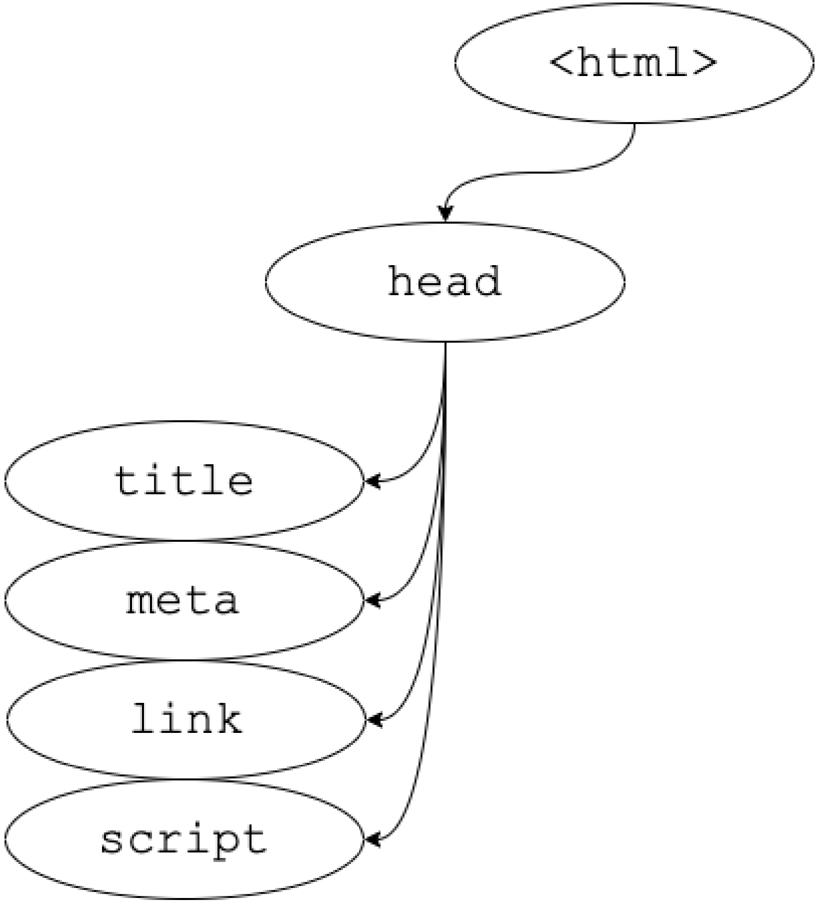
```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

# How the browser builds a DOM Tree



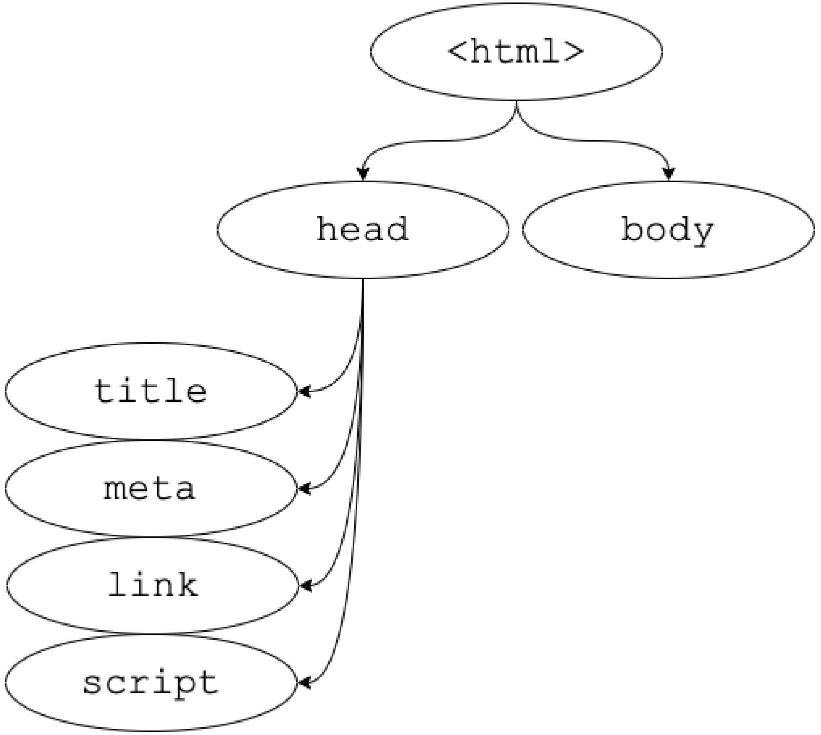
```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

# How the browser builds a DOM Tree



```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

## How the browser builds a DOM Tree

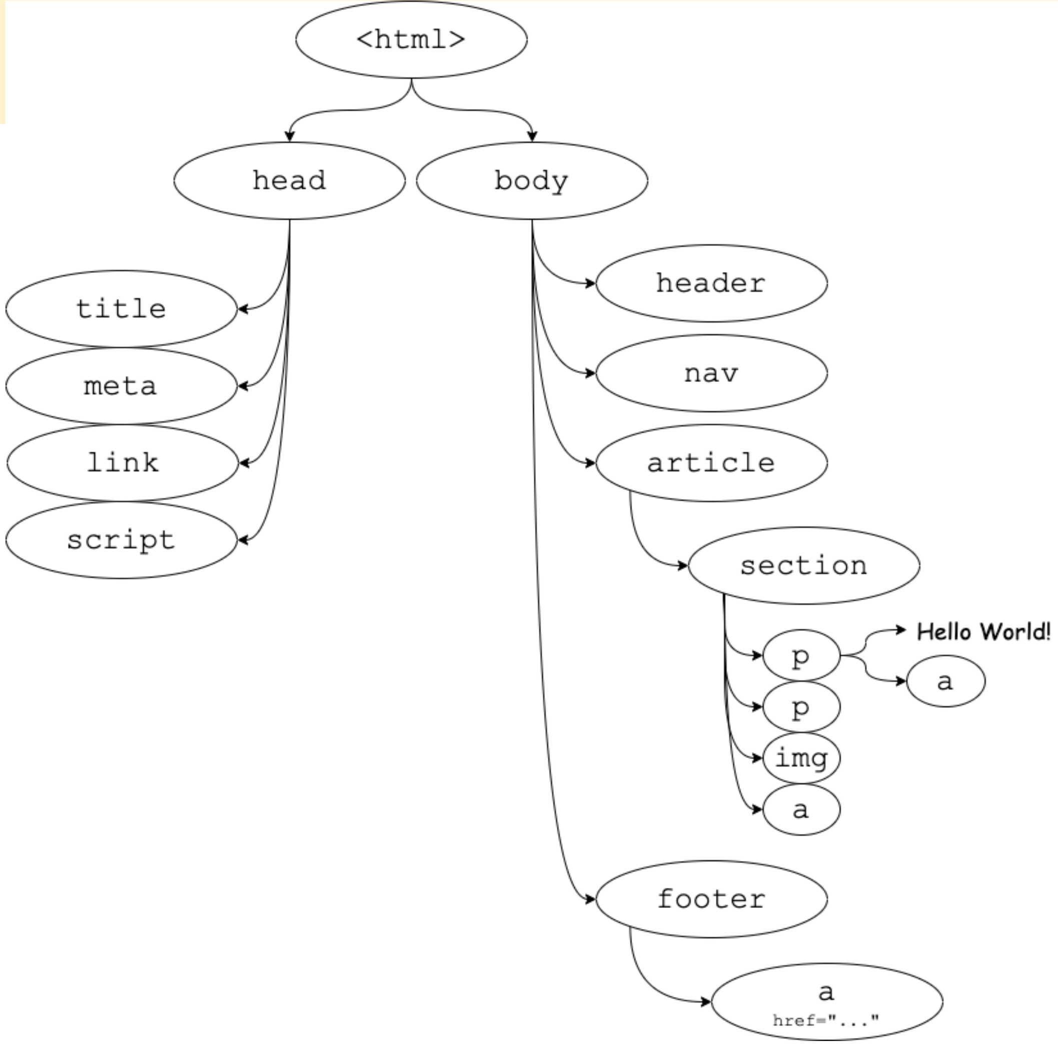

$$\begin{tikzcd}
 & \text{<html>} \arrow[d] \arrow[dr] & \\
 & \text{head} \arrow[dl] \arrow[dl] \arrow[dl] \arrow[dl] & \text{body} \\
\text{title} & & \\
\text{meta} & & \\
\text{link} & & \\
\text{script} & &
\end{tikzcd}$$

```
<html>
<head>
  <title>My Fancy Title</title>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <header>...</header>
  <nav>...</nav>
  <article>
    <section>
      <p>Hello world: <a href="...">here</a></p>
      <p>Welcome!</p>
      <img src="...">
      <a href="...">citation</a>
    </section>
  </article>
  <footer>
    <a href="..."></a>
  </footer>
</body>
</html>
```

## Full DOM

Each node in the tree has:

- Name of element
- Attributes, and their values
- Content (if any)



## Listening to the window “load” event

You can only access HTML elements after the “load” event has fired

```
"use strict";
(function() {
  window.addEventListener("load", init);
  // no access to the document here

  function init() {
    // we now have access to the DOM tree!
    // set up your initial document event handlers here.
  }
})();
```

## DOM Manipulation: Classes

What if I want to change the styles of an element on the page?

# Hiding/Showing Elements

How can we hide an HTML element?

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

In JS, it’s possible to modify the style properties of an element directly

```
id("my-img").style.display = "none";
```

- What’s wrong with the method above?

## Modifying the `classList`

You can manipulate the DOM element's `classList` with the following methods:

| Name | Description |
|---|---|
| `add(classname)` | Adds the specified class(es) to the list of classes on this element. Any that are already in the classList are ignored. |
| `remove(classname)` | Removes the specified class(es) to the list of classes from this element. Any that are already not in the classList are ignored without an error |
| `toggle(classname)` | Removes a class that is in the list, adds a class that is not in the list. |
| `contains(classname)` | Returns true if the class is in the DOM element's classList, false if not. |
| `replace(oldclass, newclass)` | Replaces the old class with the new class. |

## Example: adding a class to `classList`

Add the `hidden` class to the element when you want to hide it:

```
id("my-img").classList.add("hidden");
```

Remove the class to restore the element's styles to the state before:

```
id("my-img").classList.remove("hidden");
```

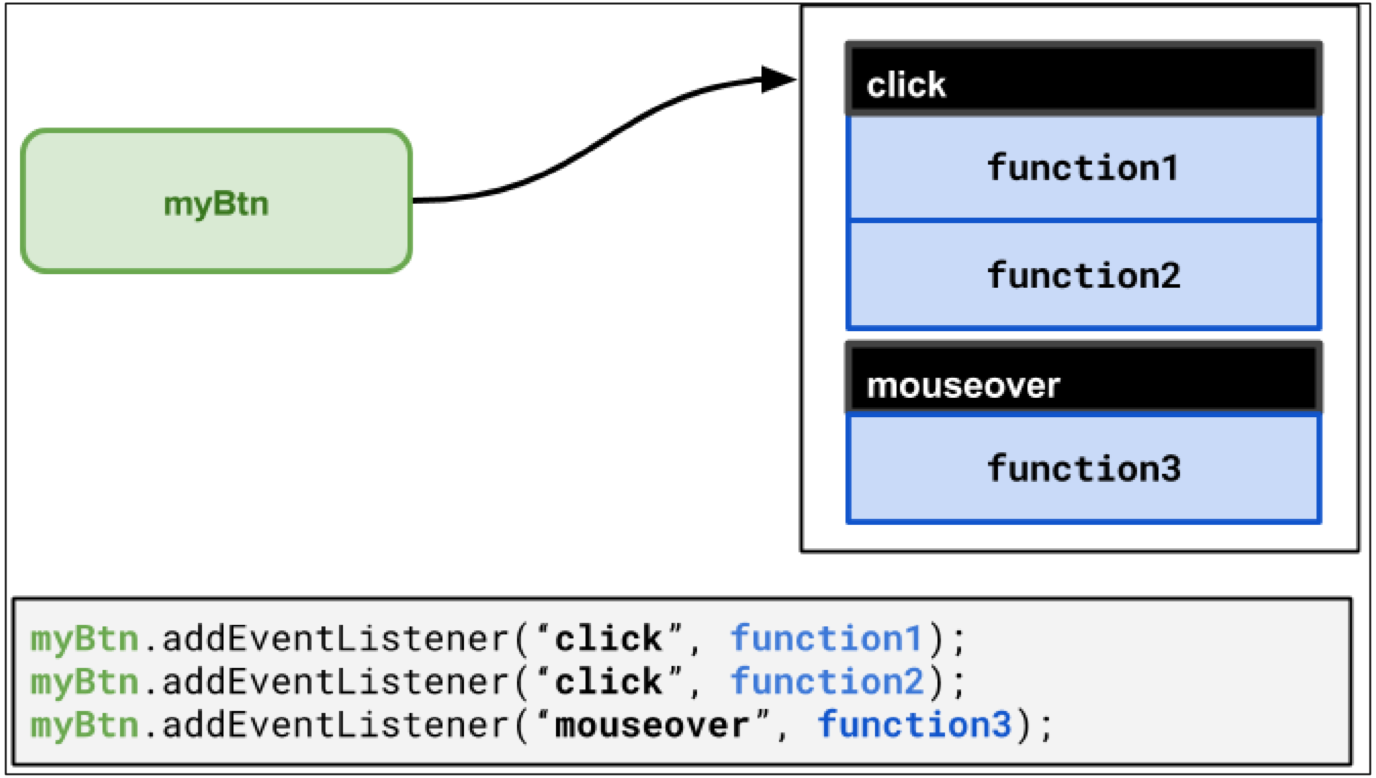
A more convenient option is to toggle the class in both cases:

```
id("my-img").classList.toggle("hidden");
```

## More about event handling

- Multiple event handlers
- Removing event handlers
- Event objects

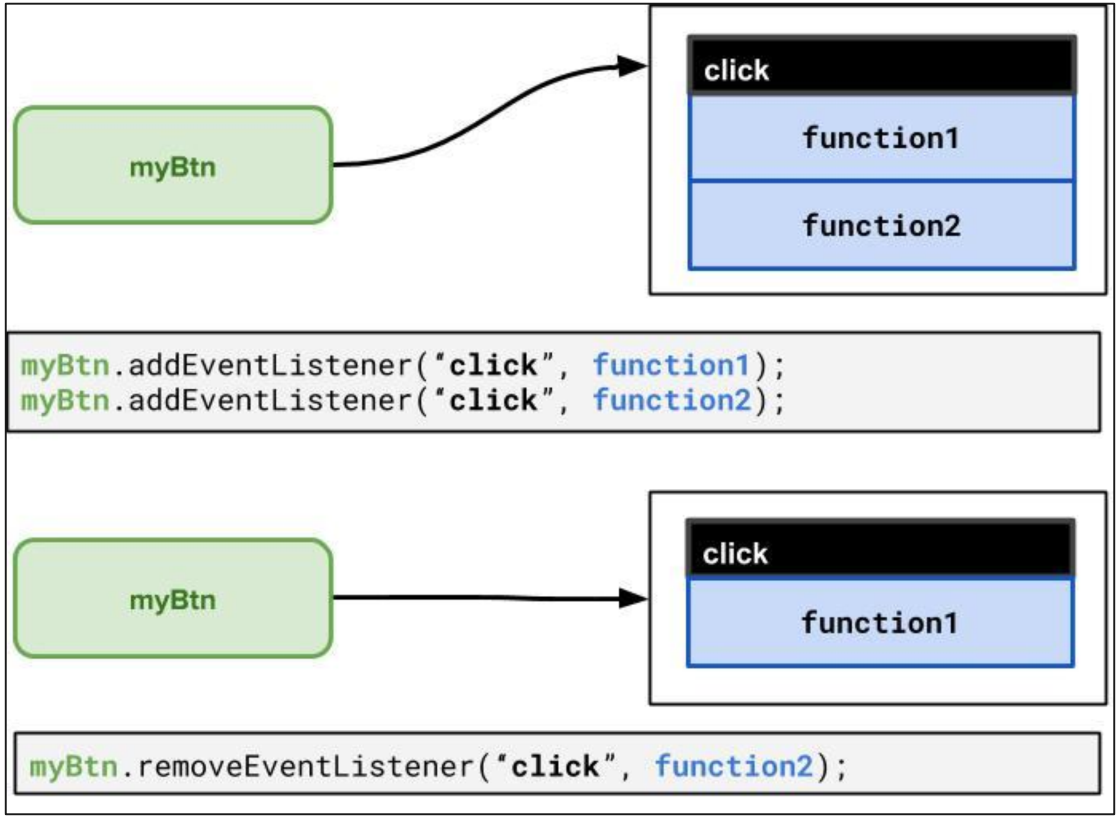
# `addEventListener` with multiple events



```
myBtn.addEventListener("click", function1);
myBtn.addEventListener("click", function2);
myBtn.addEventListener("mouseover", function3);
```

# removeEventListener

As opposed to adding event listeners to an element, you can also remove them:



```
myBtn.addEventListener("click", function1);
myBtn.addEventListener("click", function2);
```

```
myBtn.removeEventListener("click", function2);
```

## Event Objects!

Recall that the event handler function can be attached to objects (window, DOM elements, etc.)

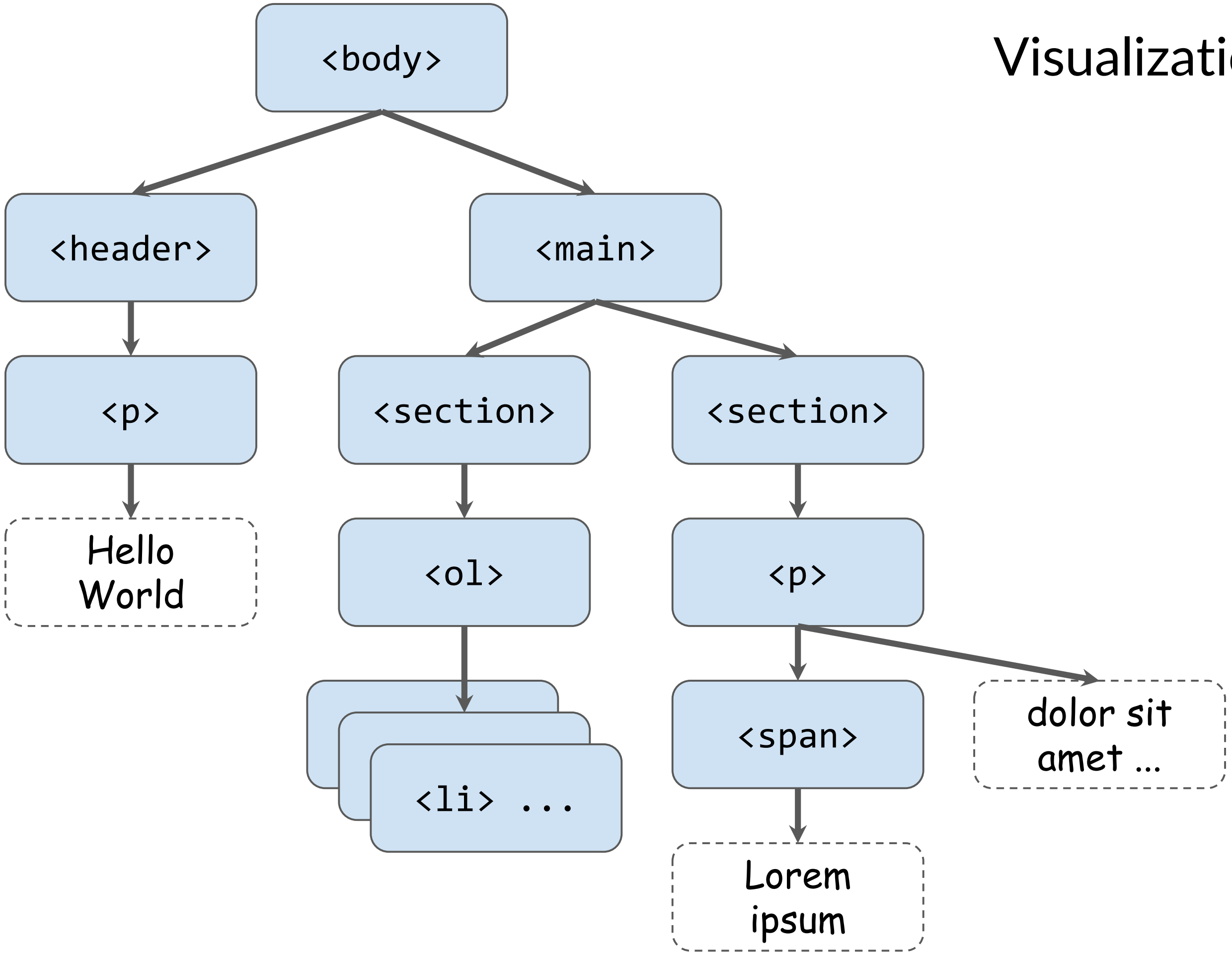
```
source.addEventListener("click", responseFunction);

function responseFunction(e) {
  // we can access the click Event object here!
}
```

When the event occurs, an **Event object is created** and passed to the event listener. You can "catch" this event object as an optional first parameter to get more information about the event.

## DOM Manipulation: Nodes

What if I want to add or remove elements on my page?

# Visualization of a DOM Tree



# Creating New Elements

| Name | Description |
|---|---|
| `document.createElement("tag")` | creates and returns a new empty DOM node representing an element of that type |

```
// create a new <h2> node
let newHeading = document.createElement("h2");
newHeading.textContent = "This is a new heading!";
```

**Note:** Merely creating an element does not add it to the page
You must add the new element as a child of an existing element on the page...

## An alias function

When creating new DOM elements using JS, you may use `document.createElement` often.

You are allowed to copy this *shortcut function* to your JavaScript code. Just be sure to use comments to describe it so that you and others understand what this function does.

```
function gen(tagName) {
    return document.createElement(tagName);
}
```

## Other handy alias functions

It's handy to create shortcut functions to code faster as we use these methods a lot!

```
function id(id) {
  return document.getElementById(id);
}

function qs(selector) {
  return document.querySelector(selector);
}

function qsa(selector) {
  return document.querySelectorAll(selector);
}
```

# Adding and Moving Nodes on the DOM

When you have a parent DOM node, you can add or remove a child DOM node using the following functions:

| Name | Description |
|---|---|
| `parent.appendChild(node)` | Places the given node at the end of this node's child list |
| `parent.insertBefore(new, old)` | Places the given node in this node's child list just before old child |
| `parent.replaceChild(new, old)` | Replaces given child with new nodes |
| `parent.insertAdjacentElement(location, newElement)` | Inserts new element at given location relative to parent |

```
let li = document.createElement("li");
li.textContent = "A list item!";
id("my-list").appendChild(li);
```

# Removing Nodes from the DOM

When you have a parent DOM node, you can remove a child DOM node using the following functions:

| Name | Description |
|---|---|
| `parent.removeChild(node)` | Removes the given node from this node's child list |
| `node.remove()` | Removes the node from the page |

```
qs("#my-list li:last-child").remove();
/* or */
let li = qs("#my-list li:last-child");
li.parentElement.removeChild(li);
```

## Removing *all* Nodes using `innerHTML`

```
// before js
<section id="fyi">
       <p>hi</p>
       <p>bye</p>
<section>

// after js
<section id="fyi">
</section>
```

```
let el = document.getElementById("fyi");
el.innerHTML = "";
```

## DOM Traversal Methods

We can use the DOM tree to traverse parent/children/sibling relationships (e.g. to remove an element from its parent node). Every node's DOM object has the following (read-only) properties to access other DOM nodes in the tree:

| Name | Description |
|---|---|
| `firstElementChild`, `lastElementChild` | start/end of this node's list of children elements |
| `children` | Array of all of this node's children (not the same as childNodes, which includes text) |
| `nextElementSibling`, `previousElementSibling` | Neighboring element nodes with the same parent, skipping whitespace nodes |
| `parentNode` `parentElement` | The element that contains the node<br>(These properties are mostly the same, see differences) |

These are the common traversal properties we'll see, but you can find a complete list here

## DOM Tree Traversal Example

Write JS code to get these elements using DOM Tree Traversal methods:

```html
<div id="container">
  <div id="column1">
    <div>1</div>
    <div id="box2">2</div>
    <div>3</div>
  </div>
  <div id="column2">
    <div>4</div>
    <div>5</div>
    <div>6</div>
  </div>
</div>
```

```js
// [#column1, #column2]

// all three ways to get <div>1</div>




// <div>3</div>


// <div>6</div>


// #container
```

# DOM Tree Traversal Example

Write JS code to get these elements using DOM Tree Traversal methods:

```html
<div id="container">
  <div id="column1">
    <div>1</div>
    <div id="box2">2</div>
    <div>3</div>
  </div>
  <div id="column2">
    <div>4</div>
    <div>5</div>
    <div>6</div>
  </div>
</div>
```

```javascript
// [#column1, #column2]
id("container").children;
// all three ways to get <div>1</div>
id("column1").firstElementChild;
id("container").firstElementChild.firstElementChild;
id("box2").previousElementSibling;
// <div>3</div>
id("box2").nextElementSibling;
// <div>6</div>
id("column2").lastElementChild;
// #container
id("box2").parentNode.parentNode;
```

## Recall: Event handler syntax

This doesn't, work, right?

```
addEventListener("click", openBox());
```

What if I wanted to pass a parameter into this function?

```
addEventListener("click", openBox(param));
```

# Anonymous Functions

```
/* named function with one parameter that logs to the console */
function sayHello(name) {
  console.log("Hello " + name);
}
/* Nameless functions which are assigned to variables */
let sayHello = function(name) {
  console.log("Hello " + name);
}
let sayHello = (name) => { // arrow function
  console.log("Hello " + name);
}
/* Equivalent function with no parens because there is only 1 parameter */
let sayHello = name => { console.log("Hello " + name); }
/* This arrow function has 0 parameter */
let sayHello = () => { console.log("Hello!"); };
```

## Parameter passing to Event Handlers

```
let defaultReply = "Hello World";
button.addEventListener("click", function() {
  draftReply(defaultReply);
});
```

How else could we do this?

```
let defaultReply = "Hello World";
// with an arrow function
button.addEventListener("click", () => {
  draftReply(defaultReply);
});
```

## Named Functions vs Anonymous Functions

```
function addElement() {
  // assume ‘element’ has been successfully defined.

  // example 1: named callback function
  element.addEventListener("dblclick", removeElement);

  // example 2: anonymous function
  element.addEventListener("dblclick", function() {
    this.parentNode.removeChild(this);
  });
}

// Removes an element when dblclicked
function removeElement() {
  this.parentNode.removeChild(this); // or this.remove();
}
```

- Both 1. and 2. work equivalently
- Do not overuse annonymous functions! Breaking down your code into named functions can be useful to reduce redundancy and keep code understandable.
- If you have more than 3 lines of code, it should be a named function.

# The keyword `this`

```
function init() {
  // this === window
  id("btn1").addEventListener("click", namedFunction);

  id("btn2").addEventListener("click", function() {
    console.log("this === " + this); // this === #btn2
  });
}

function namedFunction() {
  console.log("this === " + this); // this === #btn1
}
```

- All JavaScript code actually runs inside of "an object" that we can access with the keyword `this`
- By default, code runs in the global `window` object (so `this === window`)
- Event handlers attached in an event listener are **bound to the element**
- Inside the handler, that element becomes `this`

## `this` in other languages

```
class Test {
public:
  void hello() {
    cout << this << endl;
  }
};
```

**C++**

```
class Test {
  public void hello() {
    System.out.println(this);
  }
}
```

**Java**

```
class This {
  void hello() {
    print(this);
  }
}
```

**Dart**

```
class Test:
  def hello(self):
    print(self)
```

**Python**

**What do these all have in common?**

- They all refer to the object that owns the method.

## What happens to `this` in JavaScript?

```
let defaultReply = "Hello World";
button.addEventListener("click", function() {
  draftReply(defaultReply);
});
```

vs.

```
let defaultReply = "Hello World";
button.addEventListener("click", () => {
  draftReply(defaultReply);
});
```

```
function draftReply(startingText) {
  this.parentNode.appendNode( /*....*/ );
  // ...
}
```

## Arrow functions DO NOT bind `this`

We've seen how `this` refers to the bound element in an event handler. However, arrow functions do not bind `this` the same way.

```
element.addEventListener("dblclick", function() {
  // All good! this === element that was clicked
  this.parentNode.removeChild(this);
  id("result").textContent = "the element has been removed";
});
```

```
element.addEventListener("dblclick", () => {
  // error! this === window
  this.parentNode.removeChild(this);
  id("result").textContent = "the element has been removed";
});
```

## Comparing `this` in different callback functions

```
function init() {
  id("btn1").addEventListener("click", namedFunction);

  id("btn2").addEventListener("click", function() {
    console.log("this === " + this); // this === #btn2
  });

  id("btn3").addEventListener("click", () => {
    console.log("this === " + this); // this === window
  });
}

function namedFunction() {
  console.log("this === " + this); // this === #btn1
}
```

## Debugging JS - Tips & Tricks

- Strategies
- JS Debugger

## Strategies

- Check if your JS file has been loaded
- Use `console.log`
- Read the error messages in the console
- Use the JS debugger in Chrome Dev Tools

## Did Your JS File Load?

- Use the "Network" tab in Chrome Dev Tools
- Did you spell your JS filename correctly?
- Are you listening for the "load" event
- Sometimes, your browser caches the old script file
    - Press Ctrl + F5 to force refresh (download latest version from server)

## `console.log` everything

After adding an event listener

```
myBtn.addEventListener("click", launchRocket);

function launchRocket() {
    console.log('Launching!!!');
}
```

After accessing an element

```
let myBtn = qs('button');
console.log(myBtn);
```

After a complex calculation

```
const COLORS = ['red', 'blue', 'green'];
let randIndex = Math.floor(Math.random() * COLORS.length);
console.log(randIndex);
```

## Read the Messages in the Console

Errors in your code are printed to the Chrome Dev Tools console

- Error message
- File the error originated from
- Line number

The console error messages even link you to the line in your code where the error occurred!

## Use the JS Debugger

Under the "Sources" tab in Chrome Dev Tools

- Set breakpoints to pause execution of your code at a desired point
- View variable values at that point
- Execute code step-by-step
- Step in and out of functions

## JS Timers

Delaying and/or repeating functions with `setTimeout` and `setInterval`

## Counting Down - A Classic Loop Problem

```
function startCountDown() {
  let count = 10;
  for (let i = count; i > 0; i--) {
    console.log(i + "...");
  }
  console.log("0!");
}
```

This prints a countdown to the console as soon as it's called. But what if we want to delay each line printed by 1 second?

# Setting a Timer

| Function | Description |
|---|---|
| `setTimeout(responseFn, delayMS)` | Arranges to call given function after given `delayMS`, returns timer id |
| `setInterval(responseFn, delayMS)` | Arranges to call function repeatedly every `delayMS` milliseconds, returns timer id |
| `clearTimeout(timerID)` `clearInterval(timerID)` | Stop the given timer |

- Both `setTimeout` and `setInterval` return an ID representing the timer. A unique identifier the `window` has access to in order to manage the page timers.
- If you have access to the id, you can tell the window to stop that particular timer by passing it to `clearTimeout/Interval` later

## `setTimeout` Example

```html
<button id="demo-btn">Click me!</button>
<p id="output-text"></p>
```

```javascript
function init() {
  id("demo-btn").addEventListener("click", delayedMessage);
}

function delayedMessage() {
  id("output-text").textContent = "Wait for it...";
  setTimeout(sayHello, 3000);
}

function sayHello() { // called when the timer goes off
  id("output-text").textContent = "Hello!";
}
```

## `setInterval` Example

```html
<button id="demo-btn">Click me!</button>
<p id="output-text"></p>
```

```javascript
let timerId = null; // stores ID of interval timer
function repeatedMessage() {
  timerId = setInterval(sayHello, 1000);
}

function sayHello() {
  id("output-text").textContent += "Hello!";
}
```

## Motivating the `timerId` variable

- We sometimes need to keep track of our timer(s) when managing them between functions so we can use `clearInterval/clearTimeout` or know if we have a timer already running on our page.
- When we can't keep track of them as local variables, it is good practice to store them as module-global variables (within the scope of the module pattern, but accessible to all functions in your program).
- These examples will assume we are writing inside a module pattern for brevity, but you can refer to the full examples (linked on slides).

## “Toggling” animation with `clearInterval`

```
<button id="toggle-btn">Start/Stop<button>
```

```
let timerId = null; // stores ID of interval timer
function init() {
  id("toggle-btn").addEventListener("click", toggleMessageInterval);
}
function toggleMessageInterval() {
  if (timerId === null) {
    timerId = setInterval(sayHello, 1000);
  } else {
    clearInterval(timerId);
    timerId = null; // 2. Why is this line important?
    // 3. What happens if you swap the two lines above?
  }
}
function sayHello() {
  id("output-text").textContent += "Hello!";
}
```

## Passing Additional Parameters to `setTimeout/setInterval`

```
function delayedMultiply() {
  // 6 and 7 are passed to multiply when timer goes off
  setTimeout(multiply, 2000, 6, 7);
}

function multiply(a, b) {
  console.log(a * b);
}
```

Any parameters after the delay are eventually passed to the timer function

- Doesn't work in IE; must create an intermediate (anonymous) function to pass the parameters

**Why not just write this?**

```
setTimeout(multiply(6, 7), 2000);
```

## Common Timer Errors

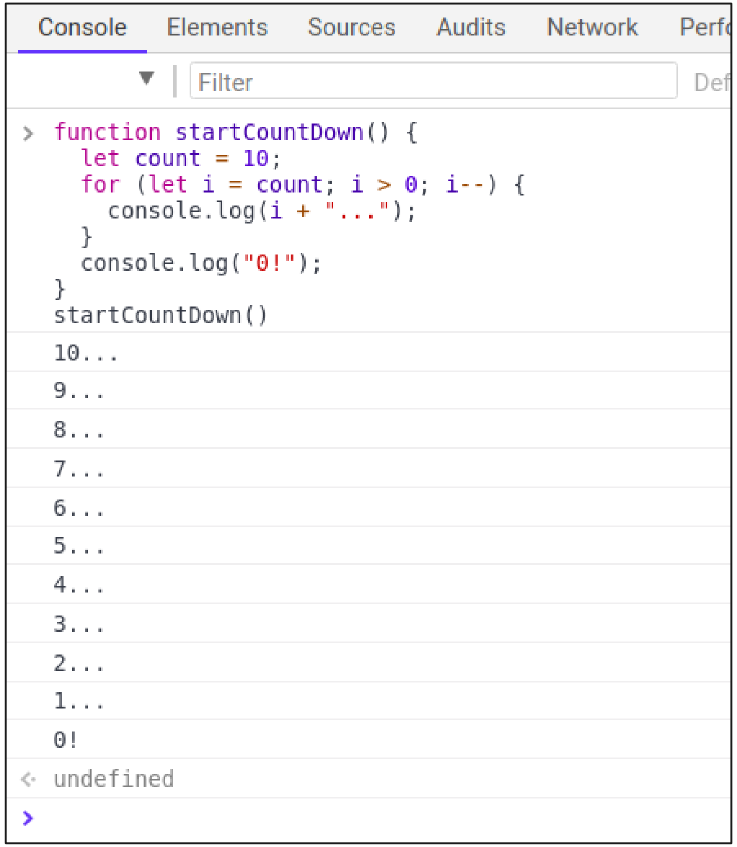
Many programmers mistakenly write `()` when passing the function

```
~~setTimeout(sayHello(), 2000);~~
setTimeout(sayHello, 2000);

~~setTimeout(multiply(num1, num2), 2000);~~
setTimeout(multiply, 2000, num1, num2);
```

What does it actually do if you have the `()`?

➢ It calls the function immediately, rather than waiting the 2000 ms

## Back to our Countdown Example



```
> function startCountDown() {
    let count = 10;
    for (let i = count; i > 0; i--) {
      console.log(i + "...");
    }
    console.log("0!");
  }
  startCountDown()
  10...
  9...
  8...
  7...
  6...
  5...
  4...
  3...
  2...
  1...
  0!
<- undefined
>
```

Recall that this function prints each line immediately (in order). If we want to output each line every 1 second (1000ms), what kind of timer should we use?

## Timed Countdown: An Initial Attempt

```
function startCountDown() {
  let i = 10;
  setInterval(function() {
    console.log(i + "...");
    i--;
  }, 1000);
  console.log("0!");
}
```

- What's wrong here? (remember we want a 10 second countdown printed to the console)
- Note that we could also replace `function() { ... }` with `() => { ... }`

## A Better Attempt

```
function startCountDown() {
  let i = 10;
  setInterval(function() {
    if (i === 0) {
      console.log("0!");
    } else {
      console.log(i + "...");
      i--;
    }
  }, 1000);
}
```

- This is closer! But there's still something wrong...
  - ➢ Our timer won't stop when we reach 0!

# A working solution...

```
function startCountDown() {
  let i = 10;
  let timerId = setInterval(function() {
    if (i === 0) {
      clearInterval(timerId);
      console.log("0!");
    } else {
      console.log(i + "...");
      i--;
    }
  }, 1000);
}
```

When `startCountDown` is called, we assign a new interval to our timer and start a one second countdown at 10.

When we reach 0, we need to clear the interval from the window's tasks

## Timers Summary

- When you want to call a function after a specified delay in time, use `setTimeout`.
- When you want to call a function repeatedly every `X` seconds, use `setInterval` (though you can also use `setTimeout` recursively!)
- For both types of timers, if you want to stop the delay/interval you'll need a variable to keep track of the timer id (returned by both functions) to pass to `clearTimeout`/`clearInterval`