Welcome!

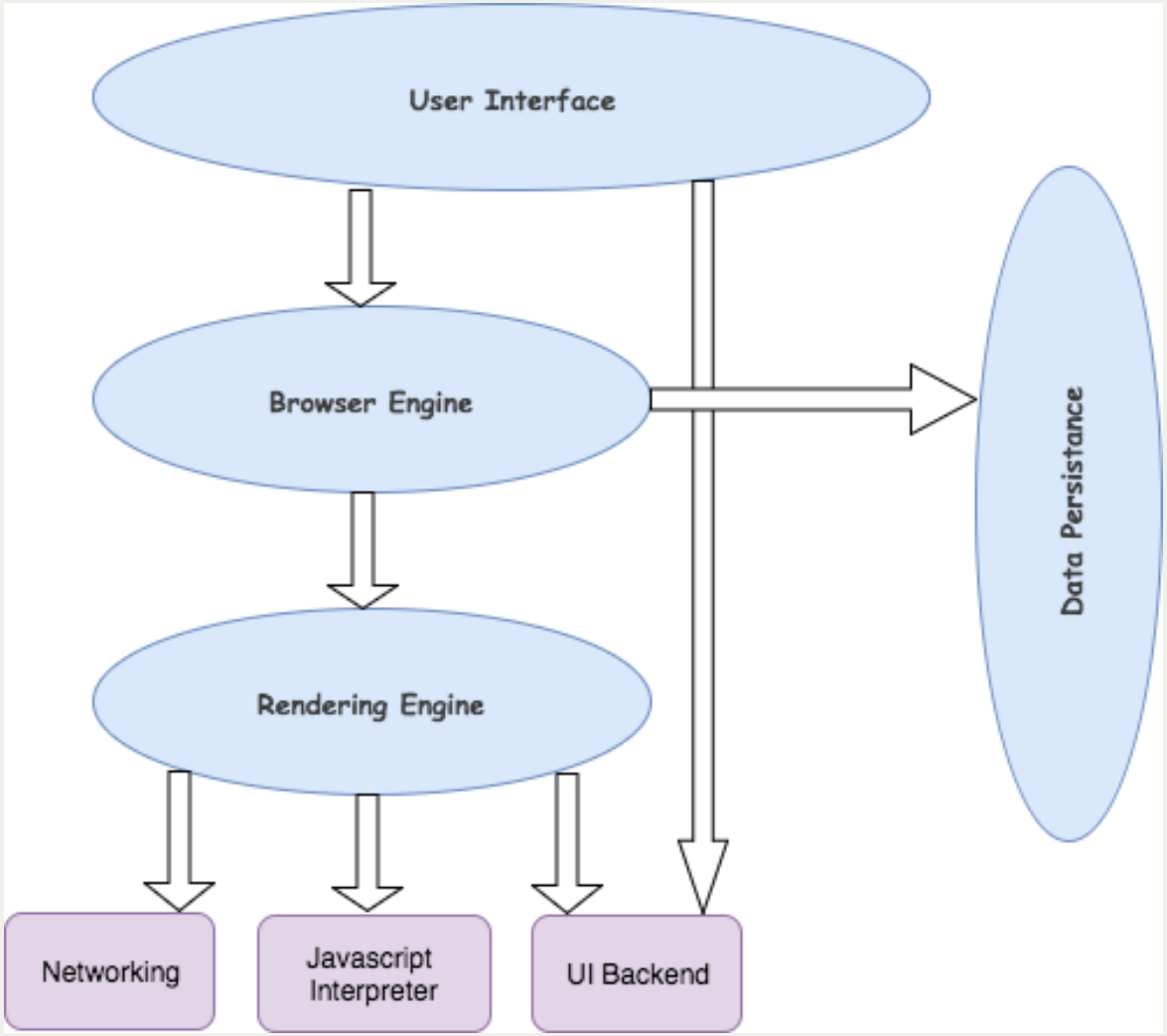
Agenda

- Review and Homework Recap
- Browser Internals
- JavaScript and the Browser
  - The Document Object Model
  - DOM Selectors
  - DOM Traversal
  - Creating DOM Nodes
  - Events
  - Animations

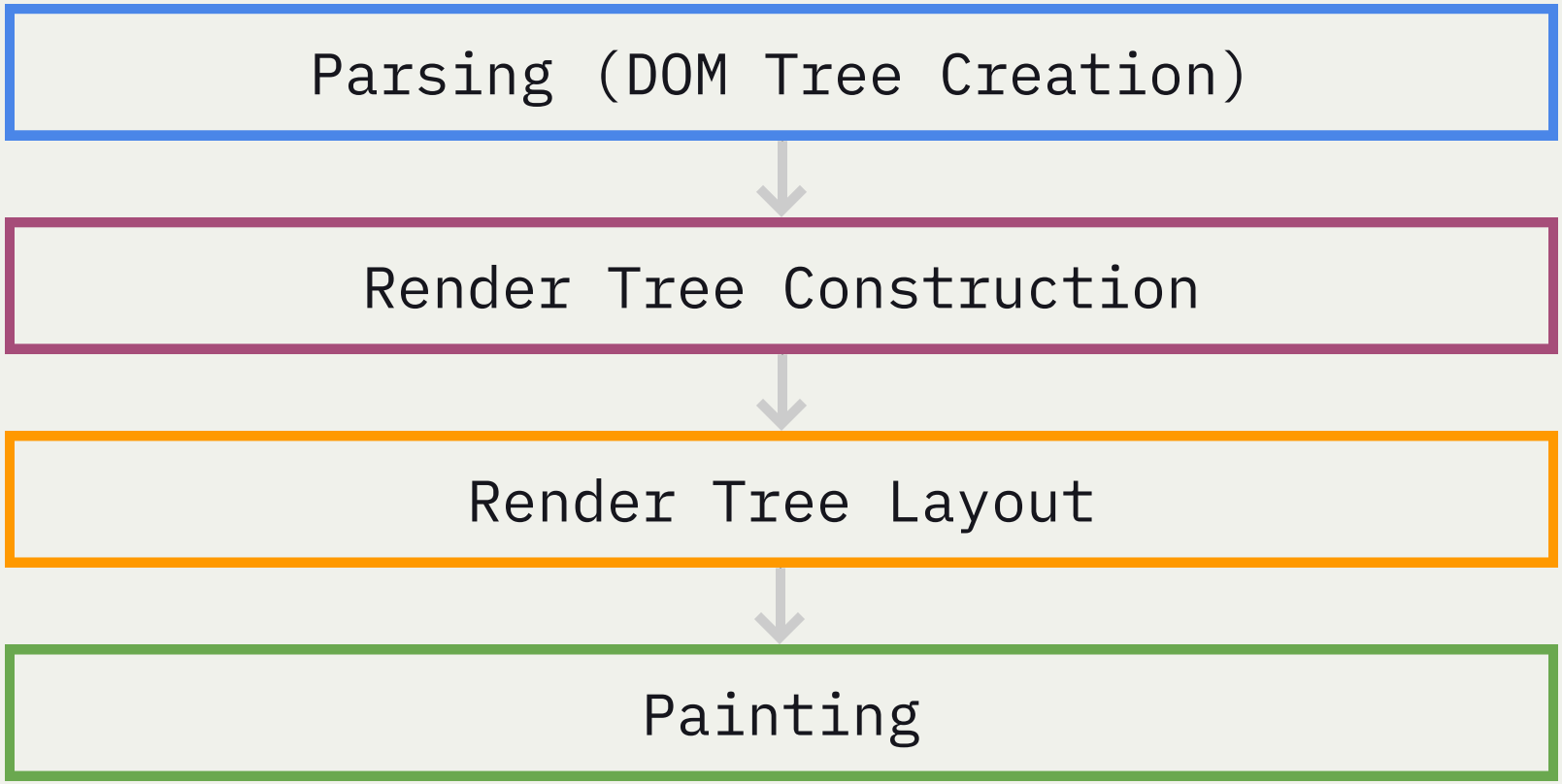
Browsers

Browser Parts

- User Interface (search bar, menu etc.)
- Browser Engine (manipulates rendering engine)
- Rendering Engine (renders the page)
- Networking (retrieves URLs)
- UI Backend (draws basic widgets - not just for the browser)
- JavaScript Interpreter (interprets and executes JS)
- Data Storage (persistence layer)


User Interface
Browser Engine
Rendering Engine
Networking
Javascript Interpreter
UI Backend
Data Persistance

Rendering Engine



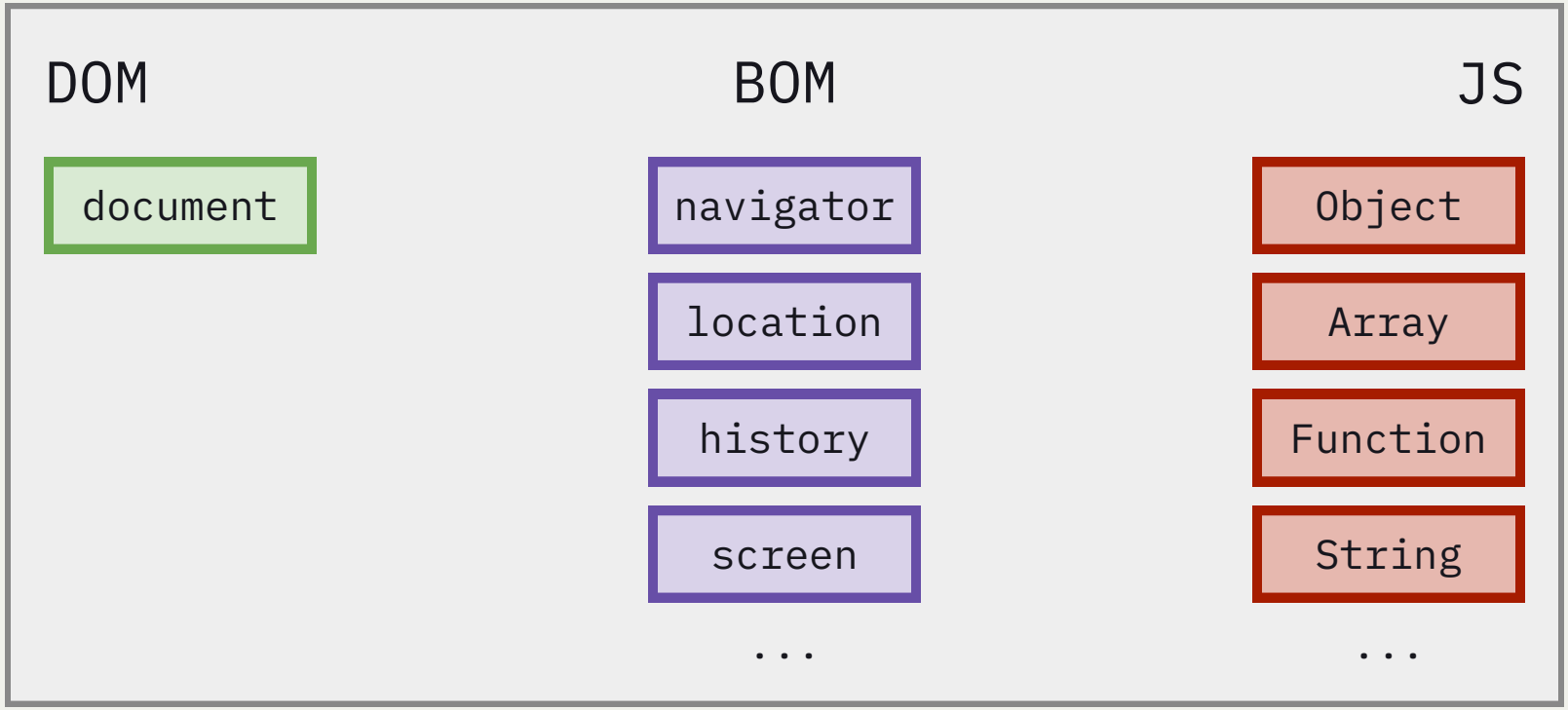
Resources

- Lin Clark: How do browsers work
  - Podcast by CodeNewbie
- HTML5 Rocks: How Browsers Work
- Moz://a Hacks: Building the DOM Faster
- Umar Hansa: An Introduction to Browser Rendering

Browser Environment

What do we have when JS runs?

`window`



What do we have when JS runs?

- `window` is the root object. It is considered the global object in a browser context, and it represents the "browser window". It contains everything
- The **Document Object Model** (**DOM**) represents all the content of a page and allows them to be modified (it presents them as objects (there is also a CSS Object Model, known as CSSOM, that allows us to change styles)
- The **Browser Object Model** provides ways of interacting with the browser itself (e.g. the URL, history, browser details etc.)

Document Object Model

What is the DOM?

- It stands for the Document Object Model
- It is a very large object, represented by the globally available `document` variable that has properties and methods
- It represents all page content as objects that can be modified. We can access, change, create or delete anything on the page using it

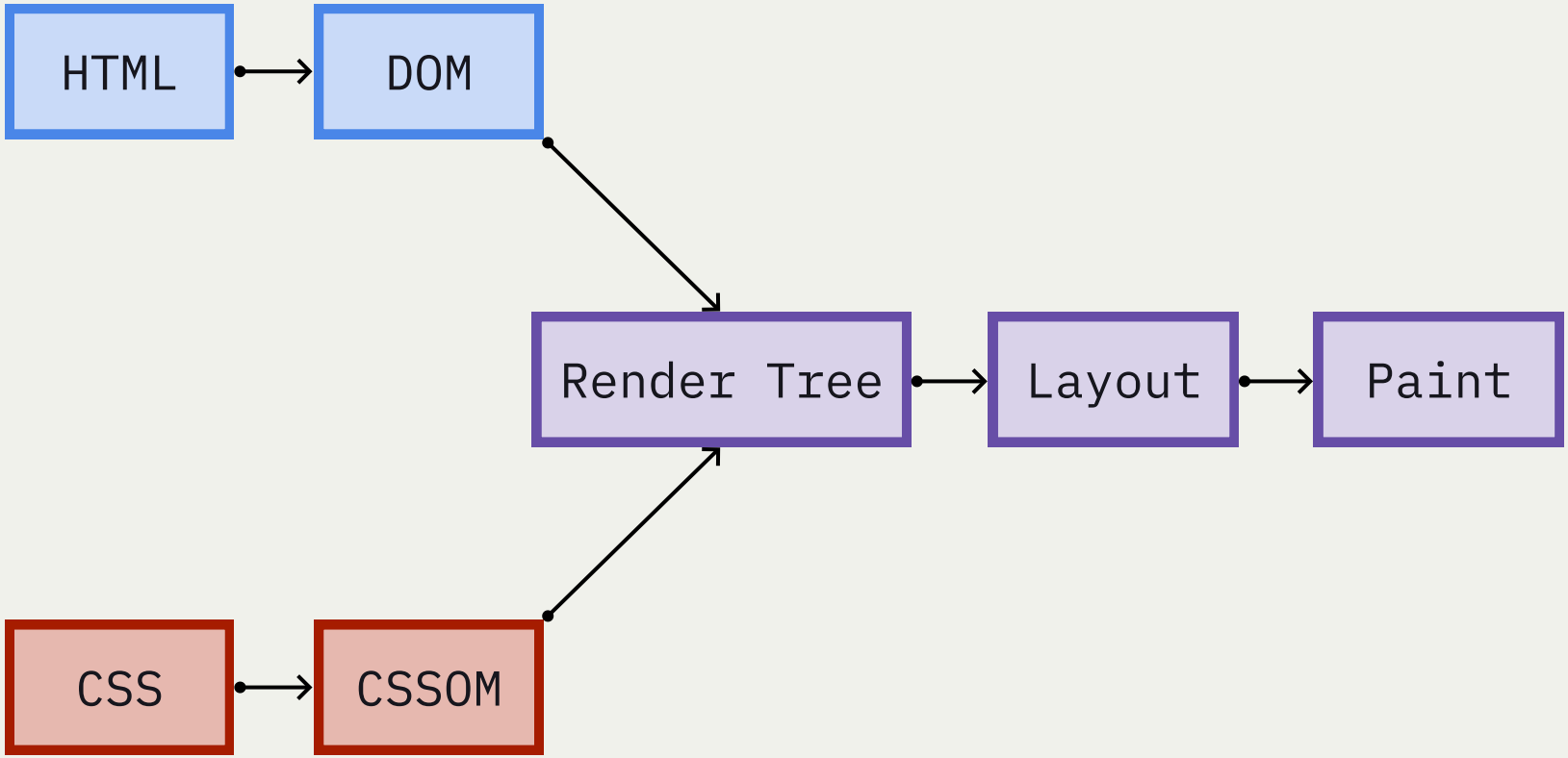
It helps define

- HTML elements as Objects
- Events for HTML elements
- Properties for HTML elements
- Methods for HTML elements

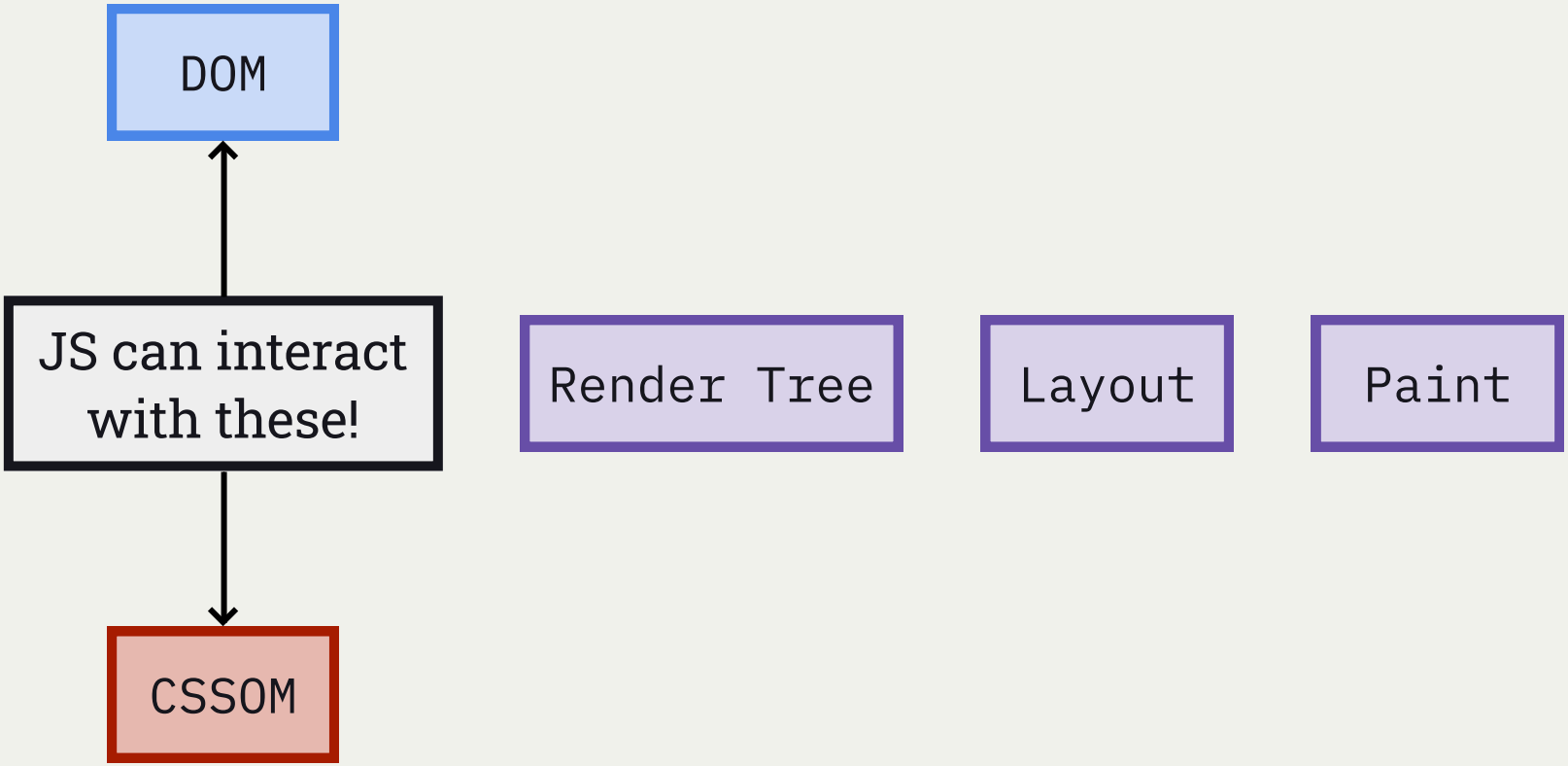
You can:

- Add, change or remove HTML elements
- Add, change or remove HTML attributes
- Add, change or remove CSS styles
- Add, change or remove Event Listeners
  - Which allows us to react to events taking place (like clicks, scrolls etc)

Where does it come from?



Where does it come from?

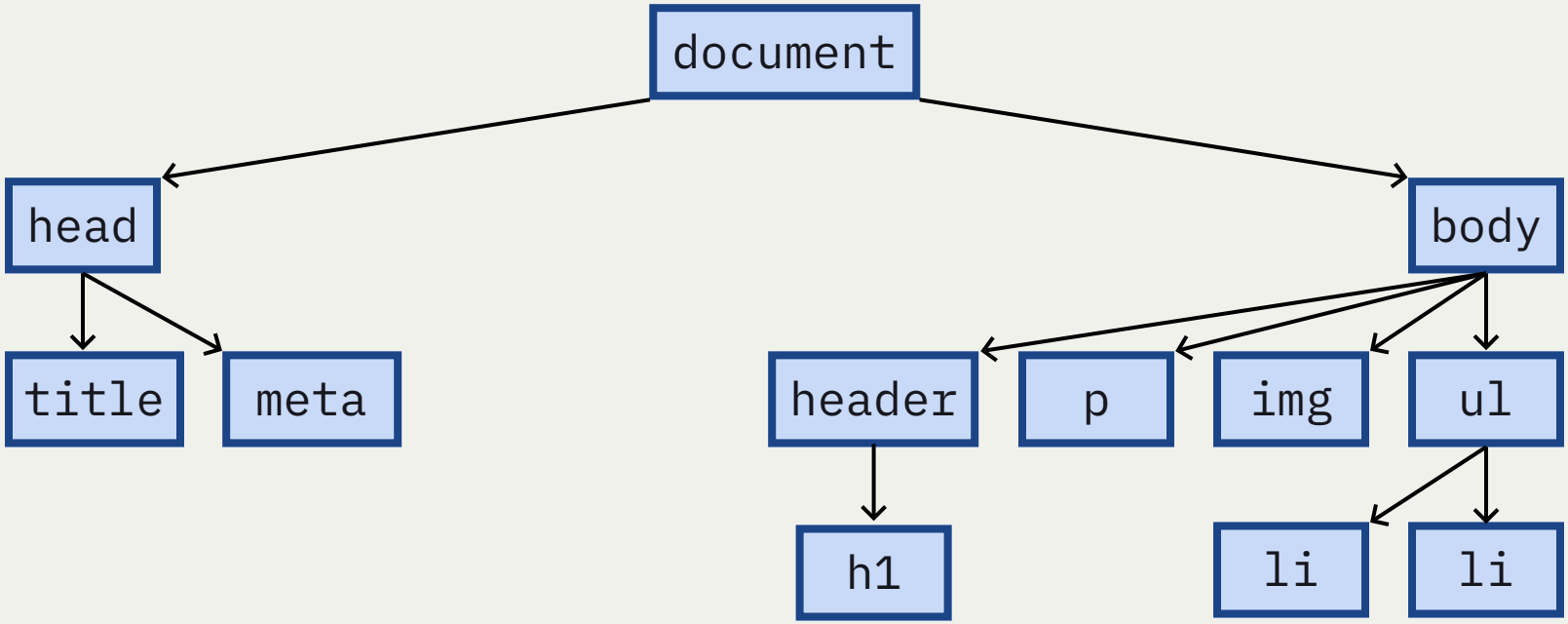


When the DOM changes

When the DOM changes, the page gets updated.

1. You make a change to the DOM with JS (through the `document` variable)
2. The browser creates a render tree
3. The browser figures out the layout tree
4. The browser re-paints the page

What does it look like?



Key Terms

- Each point of data is called a **node**
- Each node can have ***parents***, ***children*** and ***siblings***
- The DOM is accessed through a global variable called `document`
- We can call methods, and access, manipulate and delete properties (just like regular objects)
- It's called the **DOM Tree**

SOLO

Draw a DOM tree!

```
<!DOCTYPE html>
<html>
<head>
  <title>Some website</title>
</head>
<body>
  <div class="container">
    <h1>Some heading</h1>
    <a href="http://www.google.com">Some <span>link</span></a>
  </div>
  <ul>
    <li>A list item</li>
    <li>Another list item</li>
  </ul>
</body>
</html>
```

DOM Access

The `document` object gives us ways of accessing the DOM, finding elements, changing styles, etc.

The general strategy for DOM manipulation:

- Find the DOM node by using an access method and store it in a variable
- Manipulate the DOM node by changing its attributes, style, inner HTML, or by appending nodes to it

document.querySelector

```
document.querySelector("VALID_CSS_SELECTOR");
```

```
<h1>Our App</h1>

<p>Welcome</p>

<ul>
  <li>Item</li>
</ul>
```

```
const heading = document.querySelector("h1");

const para = document.querySelector("p");


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

Returns the ***first*** DOM node that matches a given CSS selector (or `null`)

document.querySelectorAll

```
document.querySelectorAll("VALID_CSS_SELECTOR");
```

```
<p>First para</p>
<p>Second para</p>

<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

```
const paras = document.querySelectorAll("p");


const items = document.querySelectorAll("li");
```

Returns ***all*** DOM nodes that match a given CSS selector, as a **`NodeList`** (very similar to an Array), or **`null`**

SOLO!

Do the exercises ~~here,~~ please!

See you in 10 minutes!

DOM Traversal

```html
<div>
  <h1>Hi</h1>
  <p>P tag</p>
  <h3>Heading</h3>
</div>
```

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

console.log(div.children);

console.log(div.childNodes);

console.log(div.parentNode);
```

node.getAttribute

```html
<img src="https://picsum.photos/400/300" alt="A nice image">

<a href="https://ga.co" id="general-assembly">
    A link to GA
</a>
```

```javascript
const image = document.querySelector("img");
const srcText = image.getAttribute("src");
const altText = image.getAttribute("alt");

const aTag = document.querySelector("a");
const href = aTag.getAttribute("href");
const id = aTag.getAttribute("id");
```

node.setAttribute

```html
<img src="https://picsum.photos/400/300" alt="A nice image">

<a href="https://ga.co" id="general-assembly">
    A link to GA
</a>
```

```js
const image = document.querySelector("img");
const srcText = image.setAttribute("src", "http://picsum.photos/300");
const altText = image.setAttribute("alt", "Another image");

const aTag = document.querySelector("a");
const href = aTag.setAttribute("href", "/home");
const id = aTag.setAttribute("id", "home");
```

Working with HTML

```
<h1>Hello World</h1>
```

```
const heading = document.querySelector("h1");
const currentText = heading.innerText;
const currentHTML = heading.innerHTML;

heading.innerText = "This is the text";
heading.innerHTML = "<u>Hi there</u>";
heading.innerHTML += "!!!";
```

Can anyone think of a reason as to why you need to be careful when changing the text using `.innerHTML`?

Getting Values

```html
<input type="text" value="User types here">
```

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

const currentValue = input.value;

input.value = "Something else";

const newValue = input.value;
```

Working with Styles

```html
<h1>Hello World</h1>
```

```javascript
const heading = document.querySelector("h1");

// Getting Styles
const currentStyles = getComputedStyle(heading);
const fontSize = currentStyles.fontSize;

// Setting Styles
heading.style.width = "400px";
heading.style.fontSize = "24px";
```

Working with Styles

- CSS properties that normally have a hyphen in it, you must camelCase it
- Number properties must have a unit - they won't default to pixels

SOLO!

Do the exercises here, please!

See you in 10 minutes!

Creating DOM Nodes

We can make our own HTML elements as well!

```
const myParagraph = document.createElement("p");
myParagraph.innerText = "Created with JS";
myParagraph.style.fontSize = "24px";
myParagraph.style.color = "hotpink";

// Put it on the page

document.body.appendChild(myParagraph);
// Or...
document.body.insertBefore(myParagraph, document.body.firstChild);
// Or...
document.body.innerHTML += newPara;
```

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.)

Callbacks

What are callbacks?

A callback function is really just a regular function passed into another function as an argument.

They are very useful because they allow us to schedule asynchronous actions - they are functions that serve as a response (could be an event, or an interaction with an API - or anything, really)

Callbacks

```
function runCallback(cb) {
  // Wait a second...
  cb();
}

function delayedFunction() {
  console.log("I was delayed");
}

runCallback(delayedFunction);
```

Callbacks

```
function sayHi(name) {
  alert("Hello " + name);
}

function processInput(cb) {
  const name = prompt("Please enter your name.");
  cb(name);
}

processInput(greeting);
```

Let's see some examples!

Scheduling

Scheduling

Occasionally, we don't need to run a function straight away - we want to run it after some time has elapsed, or at some regular interval.

`setTimeout`

Delays a function's execution by some amount of milliseconds

`setInterval`

Repeats the execution of a function continuously with an interval in between each call

setTimeout

setTimeout

Occasionally, we don't need to run a function straight away - we want to run it after some time has elapsed.

setTimeout

```
function delayedFunction() {
  console.log("I was delayed!");
}

setTimeout(delayedFunction, 1000);

setTimeout(function() {
  console.log("I was also delayed - but I am anonymous");
}, 2000);
```

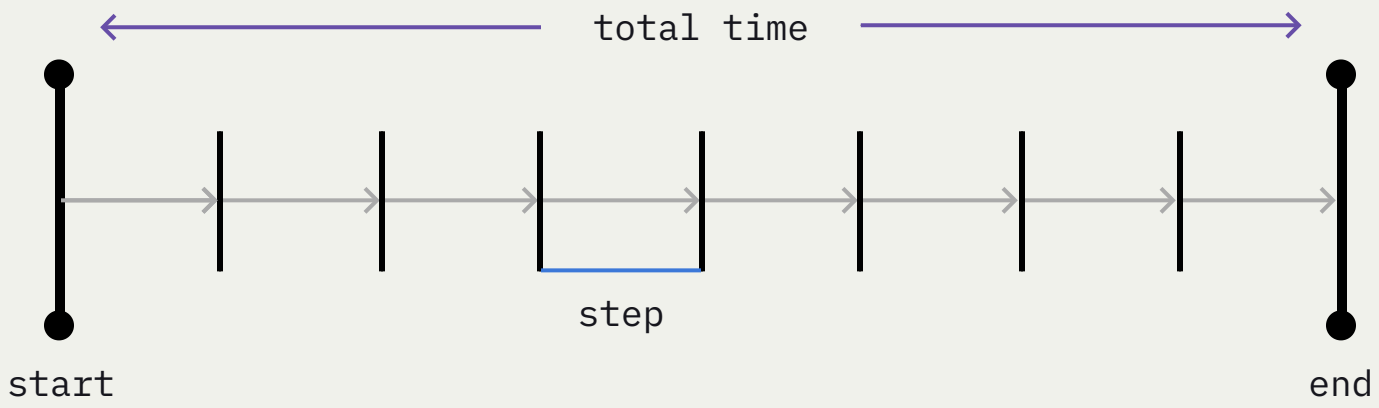
setInterval

setInterval

```
function regularlyCalledFunction() {
  console.log("I am called regularly");
}

const timer = setInterval(regularlyCalledFunction, 1200);

clearInterval(timer); // At some point, you can cancel the inteval too!

setInterval(function() {
  console.log("I am also called regularly - but I am anonymous");
}, 2000);
```

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 >= 0
    CALL fadeImgAway in 10ms

CALL fadeImgAway to start the animation
```

Fade Away

```
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);
```

Review

That's all!

Homework

- Go through DOM Events and watch this course
- Finish off in-class exercises
  - The DOM Detective
  - Replace The Logo
  - More DOM Manipulation
- Work on your CSS Selectors using Flukeout
- Any previous homework
- Extra: Begin reviewing the next lesson's content

What's next?

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

Thank you!