Introduction to the Document Object Model

Before we DOM it up

- Rest / Spread Operators
- Array.isArray and why we need it

Rest / Spread Operator

- The Rest / Spread operator (…) was introduced in ES6 and provides some great functionality.

- Rest: used in functions to gather any nth number of arguments that may be passed after the defined parameter into an array.

- Spread: opposite of Rest, and is used to restructure an array (or array-like object) into individual parts. Simplest use case is to concatenate arrays, but has many other helpful use cases.

Array.isArray

- As we noticed yesterday with our typeof demo, Arrays [ ] are actually Objects. This has to do with the fact that Arrays are descendants of the Object prototype chain. We will attempt to dive into this more in the future, but here is a good medium article discussing the prototype chain: https://codeburst.io/master-javascript-prototypes-inheritance-d0a9a5a75c4e

- Because we cannot use typeof to determine if something is an Array, the Array.isArray(value) method was created to serve this purpose.

You’re about to learn

- What is the DOM?
- Why should we care?
- DOM Manipulation
    - Searching the DOM
    - How to traverse the DOM
    - How to change the DOM

What is the DOM?

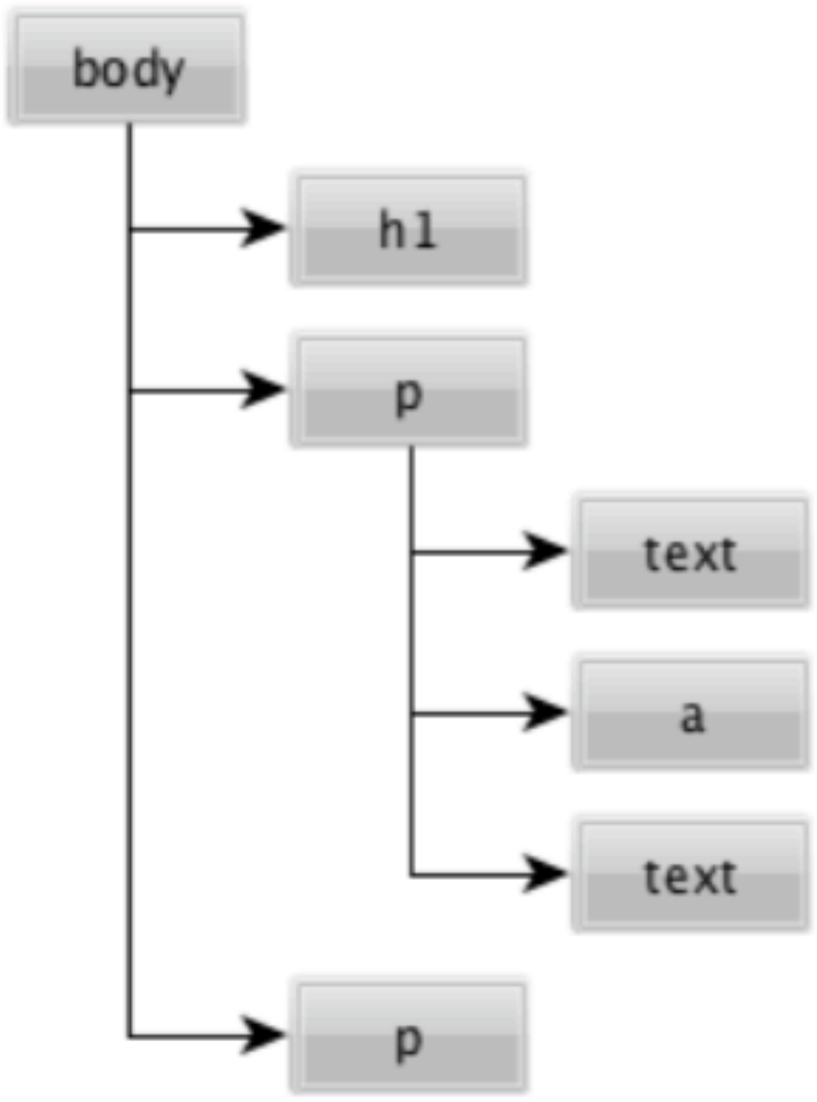
What is the DOM?

The Document Object Model is what allows web pages to render, respond to user events, and change

HTML vs DOM

```
<body>
   <h1>Hello</h1>
   <p>
      Check out my
      <a href="/page">Page!</a>
      It's the best page out there
   </p>

   <p>Come back soon!</p>
</body>
```



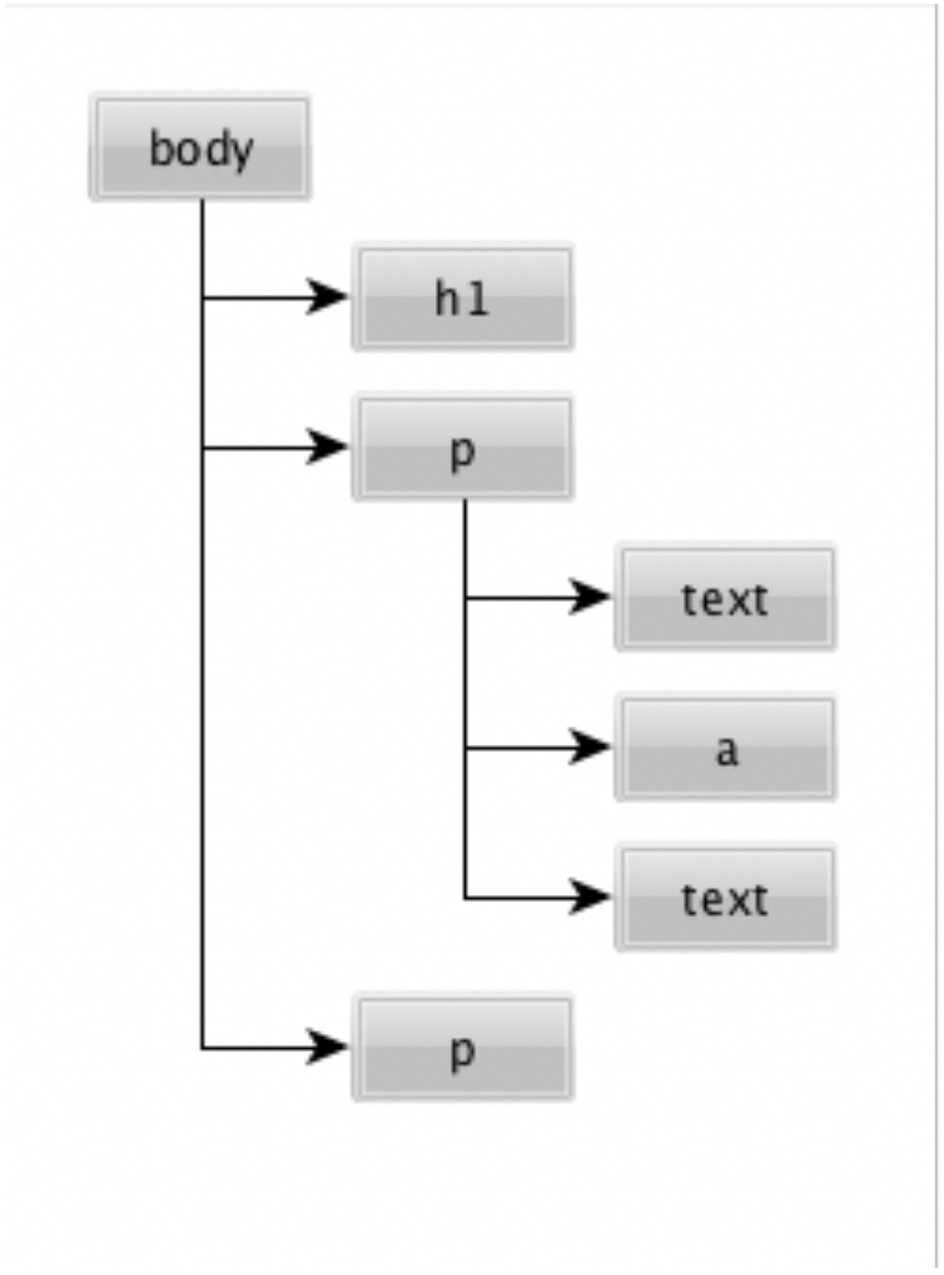
The DOM is a tree

- The main idea here: There is a root Node that branches into other Nodes, known as its children Nodes

- Each Node can have none or many children Nodes

- Nodes can have 0 or 1 parent

- Nodes can have 0 to many Sibling Nodes



Browser Dev Tools (Chrome, Firefox)

Why do we care?

The DOM makes it possible to use JavaScript to manipulate the document content and structure

Nodes have lots of Attributes

- Nodes are JavaScript Objects
- Nodes have Attributes that are JavaScript properties
- Attributes define how the Node looks and responds to User activity

The *document* Object

- ‘document’ is the Global reference to the DOM entry point
- Provides methods for navigating and manipulating the DOM

- The *document* object is the important connection between the DOM and JavaScript code˜

Searching the DOM

- **getElementById - finds node with a certain ID attribute**
  - `document.getElementById("will");`
- **getElementsByClassName - finds nodes with a certain CLASS attribute**
  - `document.getElementsByClassName("will")`
- **getElementsByTagName - finds nodes with a certain HTML tag**
  - document.getElementsByTagName("div");
- **querySelector, querySelectorAll - searches using CSS selectors**
  - `document.querySelector("#will .will:first-child");`

Array-Like Objects?

- When you use any DOM selector methods that will return a collection of Nodes, what is returned is an object called *HTMLCollection* (or *NodeList* if you use querySelectorAll).

- This NodeList looks suspiciously like an array, but it is not.

```
const divList = document.getElementsByTagName(“div”)

Array.isArray(divList) // false
```

- The NodeList is still zero-indexed, and values are accessible by index look-up like arrays, e.g. `divList[0]` gets you the first element.

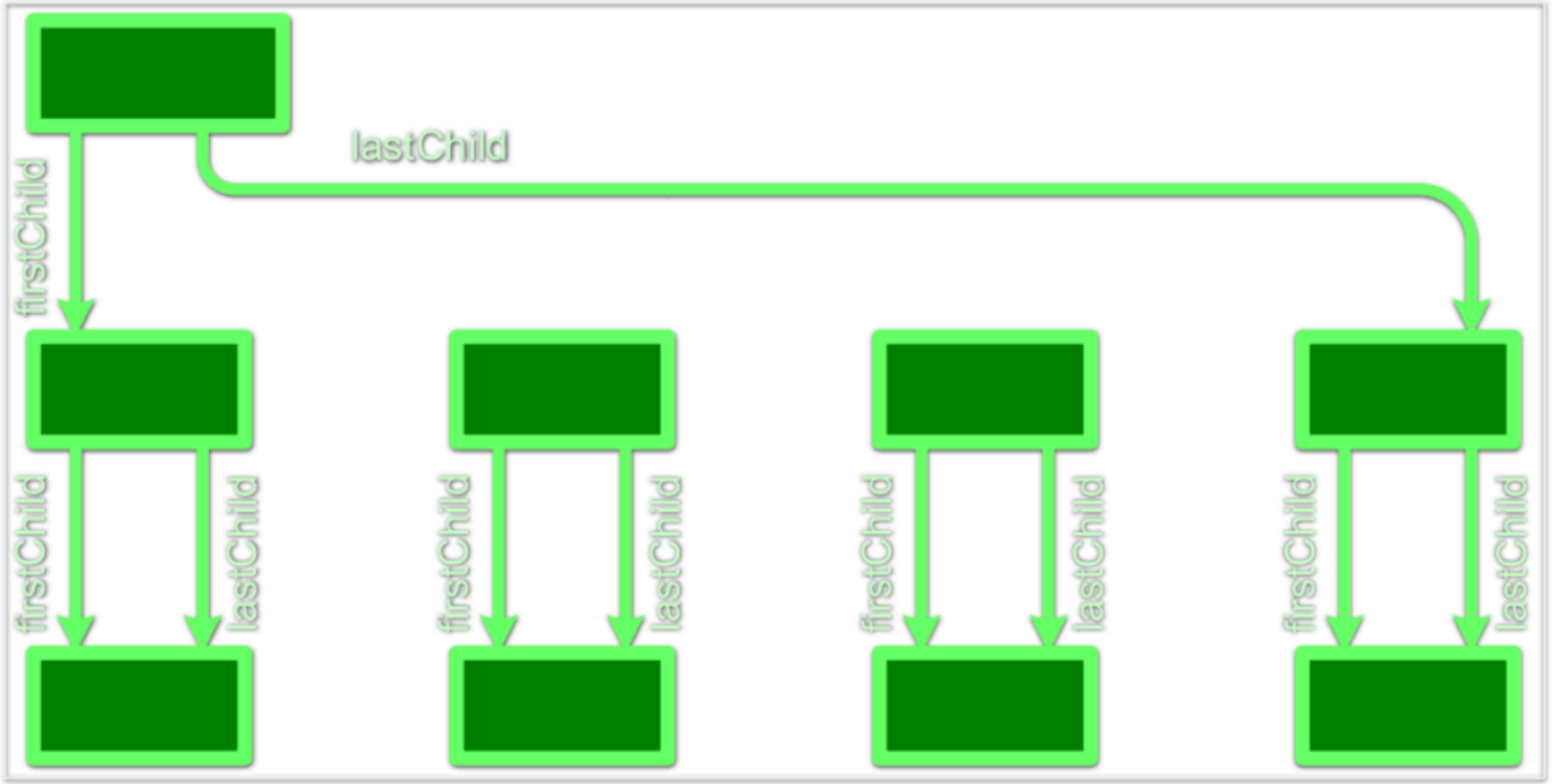
- However, you won’t have access to any of the array methods available to true arrays, and therefore are somewhat limited in how you could programmatically operate over the NodeList

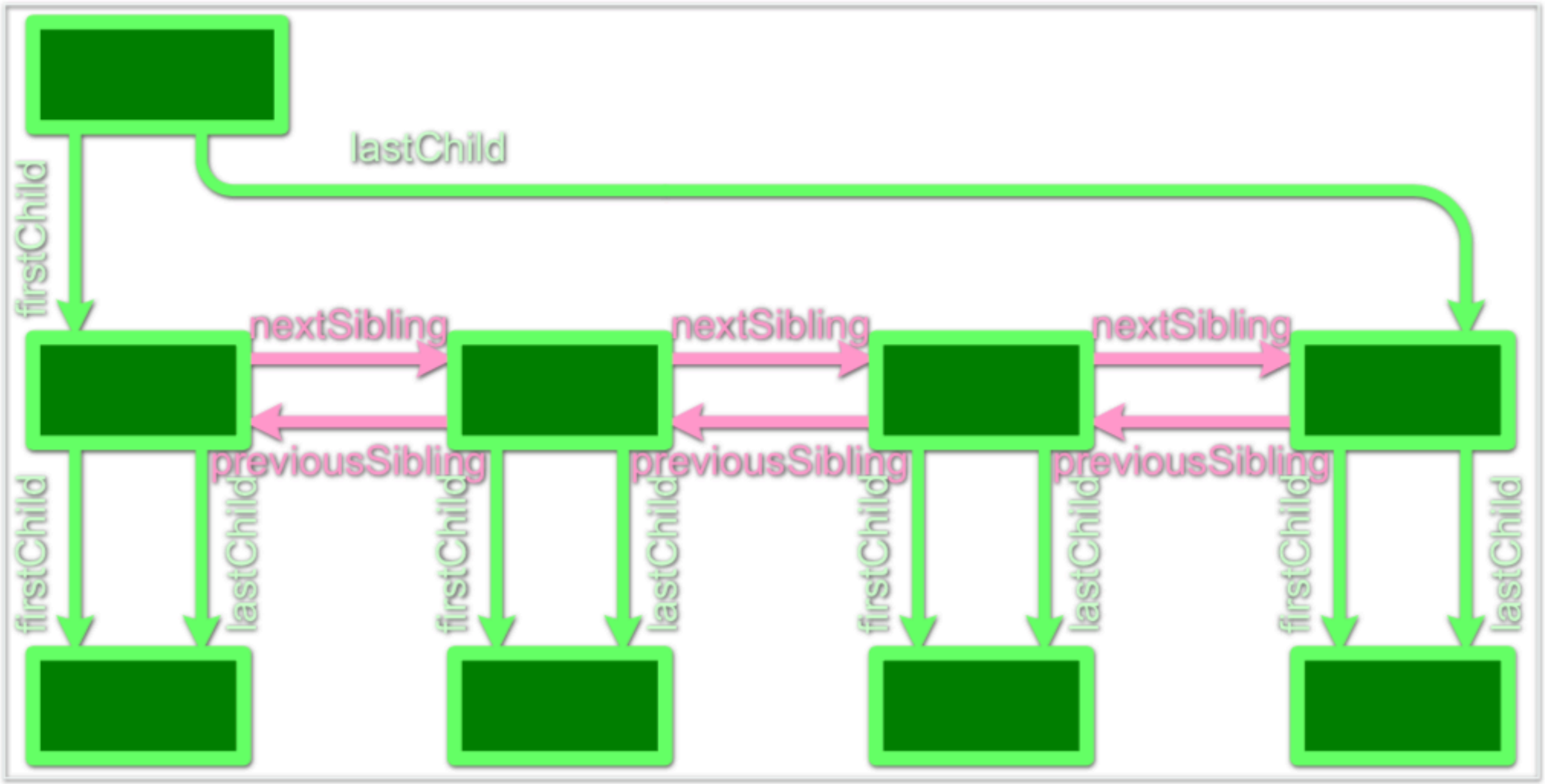
Array-Like Objects?

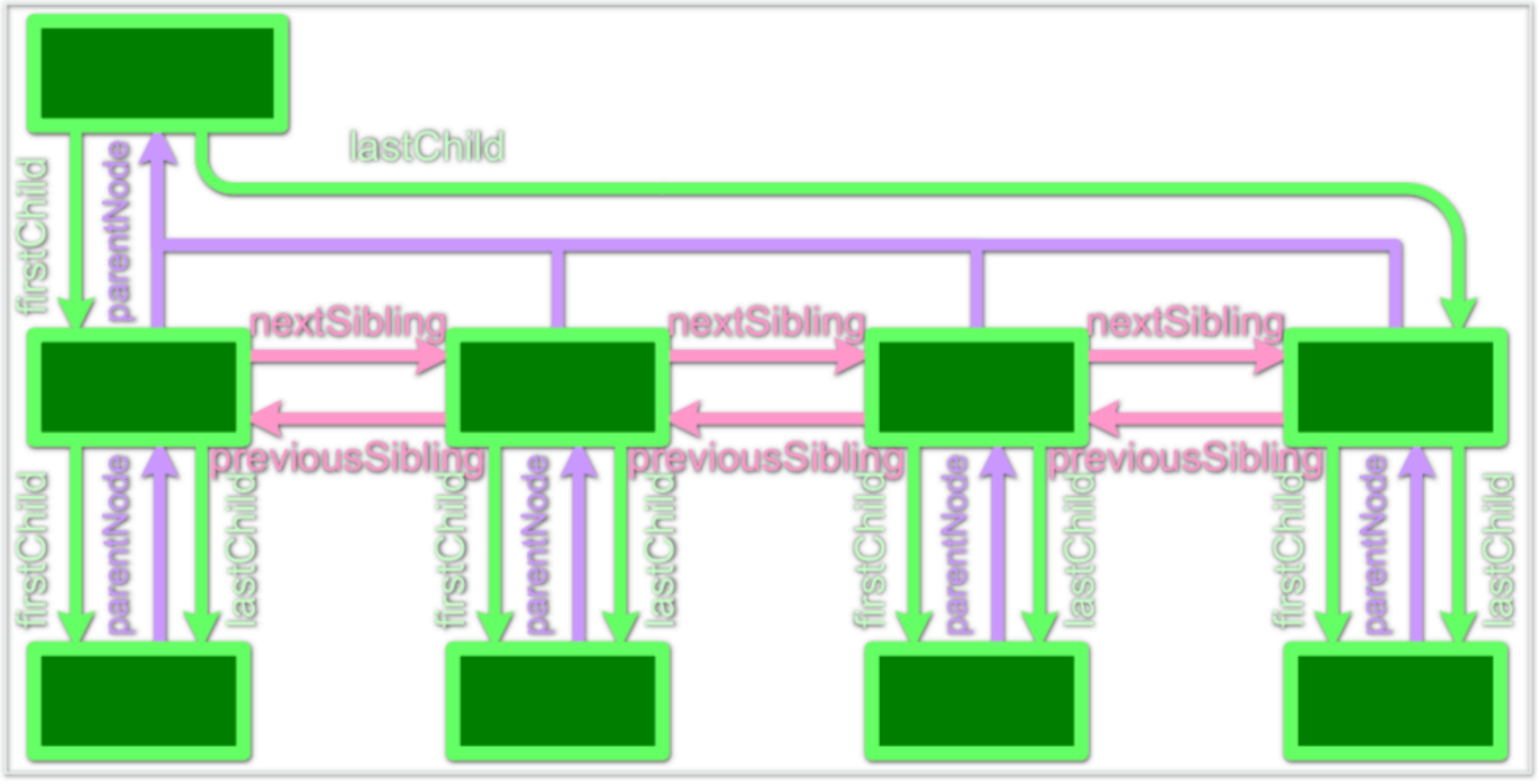
- There are three ways to get around this:
  - `const divArr = [].prototype.slice.call(divList)`
  - `const divArr = Array.from(divList)`
  - `const divArr = […divList]`

Traversing the DOM

- As the DOM is a Tree Structure, it is relatively easy to navigate because:
    - At any point in the DOM you are at a Node
    - No matter where you go, you’re still at a Node
        - Child
        - Parent
        - Sibling
    - All Nodes share similar DOM navigation methods


lastChild
firstChild
firstChild
lastChild
firstChild
lastChild
firstChild
lastChild
firstChild
lastChild


lastChild
firstChild
nextSibling
nextSibling
nextSibling
previousSibling
previousSibling
previousSibling
firstChild
lastChild
firstChild
lastChild
firstChild
lastChild
firstChild
lastChild


lastChild
firstChild
parentNode
nextSibling
nextSibling
nextSibling
previousSibling
previousSibling
previousSibling
firstChild
parentNode
lastChild
firstChild
parentNode
lastChild
firstChild
parentNode
lastChild
firstChild
parentNode
lastChild

Traversing the DOM

- Access children Nodes

```
element.children, element.lastChild, element.firstChild
```

- Access sibling Nodes

```
element.nextElementSibling, element.previousElementSibling
```

- Access parent Node (if any)

```
element.parentElement
```

Changing the DOM: Changing style attributes

```
element.style.fontWeight = “bold”;
```

- CSS
  - background-color
  - border-radius
  - font-weight
  - list-style-type
  - word-spacing
  - z-index
- JavaScript
  - backgroundColor
  - borderRadius
  - fontWeight
  - listStyleType
  - wordSpacing
  - zIndex

Changing the DOM: Changing CSS Classes

- *className* attribute is a string of all of a Node's classes
- *classList* is HTML5 way to modify which classes are on a Node

```
document.getElementById("MyElement").classList.add('class');

document.getElementById("MyElement").classList.remove('class');

if(document.getElementById("MyElement").classList.contains('class')){
    document.getElementById("MyElement").classList.toggle('class');
}
```

Changing the DOM: Creating Elements

- Create an element
  - `document.createElement(tagName)`
- Duplicate an existing Node
  - `node.cloneNode()`
- Nodes are just free floating, and not connected to the document itself until you link them to the DOM.

Changing the DOM: Adding Elements to the DOM

- Insert newNode at end of current Node
  - `node.appendChild(newNode);`
- Insert newNode at beginning of current Node
  - `node.prependChild(newNode);`
- Insert newNode before a certain childNode
  - `node.insertBefore(newNode, sibling);`

Changing the DOM: Removing Elements

- Remove the oldNode child
  - `node.removeChild(oldNode)`
- Quick hack:
  - `oldNode.parentNode.removeChild(oldNode)`

JS Event Handling

What is an event?

- A JavaScript event is a callback that gets fired when something happens related to the DOM on your website.

- For instance, when an element is clicked, or perhaps hovered over

- An event handler can be attached to an element so that when a specific event happens, a specific “callback” function gets fired

Listening for events - native JS

```
document.getElementById(“myId”).addEventListener(“click”, function(event){
    alert(‘you clicked me’)
}
```

- The key bit to the snippet above is the .addEventListener, which attaches an event handler (an anonymous function to execute) when the element is clicked

- There are many other events to listen for, too, such as:
    - hover
    - keyup / keydown
    - mouseover
    - scroll

The HTML `<form>` element

- The login, signup, and address forms you see online all share a common tag: `<form>`

- Inside of `<form>` are several elements that make up forms:
  - Text input boxes
  - Dropdown
  - Radio buttons,
  - Checkboxes, etc

<form> example

```
<form action="/process" method="POST">
  <label for="username">Username</label>
  <input type="text" name="username" id="username">
  <label for="password">Password</label>
  <input type="password" name="password">
  <input type="submit" value="Submit">
</form>
```

Don’t worry about action and method for now - also don’t worry about submitting your form just yet.

Retrieve input from a form element

You can see what's inside of a form element fairly easily, using the .value attribute:

```
<!-- Sample form input element -->
<input type="text" name="username" id="username">

document.getElementById('username').value
// returns the value of the field
```

Get the title of the form

Imagine a <form> with an <h1> tag above it that has the form title. We can use the attribute .innerText to retrieve the title inside the <h1> tag, or even change it.

Get the title of the form

```
<h1 id="title">Enter your information</h1>


heading = document.getElementById('title')
heading.innerText
>> "Enter your information"
```

```
// set name to Zach
var name = "Zach"

// Changes the text in the DOM
heading.innerText = "Enter " + name + "'s Information"

// inside of <h1> to say this instead
heading.innerText
>> "Enter Zach's Information"
```

Change the content of a `<div>`

Let’s now say that our <h1> lives inside of a . Using the .innerHTML attribute, we can change the innerHTML of the entirely.

Change the content of a `<div>`

Before:

```
<div id="main-section">
  <h1>Hello World</h1>
</div>
```

JS:

```
document.getElementById('main-section').innerHTML = "<h3>Hello World Smaller</h3>"
```

After:

```
<div class="main-section">
  <h3>Hello World Smaller</h3>
</div>
```

Assignment