JavaScript / DOM

André Restivo

Index

Introduction

DOM

- The **Document Object Model** (DOM) is a fully object-oriented representation of a web page as a logical tree of *nodes*.
- It allow programmatic access to the tree, allowing programs to **read** and **change** the document **structure**, **style** and **content**.
- Nodes can also have event handlers attached to them. Once an event is triggered, the event handlers get executed.
- It can be manipulated from the browser using **JavaScript**.

Document

The Document object represents an HTML document.

You can access the current document in *JavaScript* using the **global** variable **document**.

Some Document **properties**:

- URL – Read-only location of the document.
- title – The document title.
- location – A Location object that can be assigned in order to navigate to another document.

```
document.location.assign('https://www.google.com/
```

Another **global** variable represents the browser called Window.

Location

A Location allows us to separate the URL into its many components.

If the current URL is:

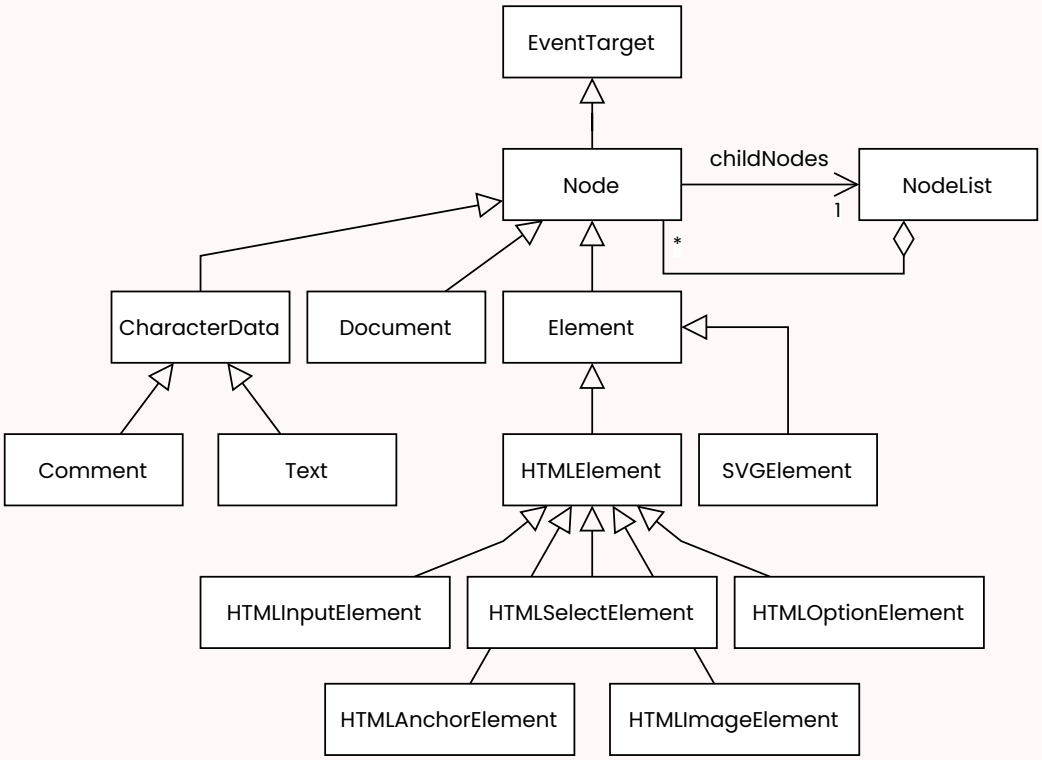
```
https://www.example.com:8080/path?key=value#somew
```

Then:

```
console.log(document.location.protocol)    // htt
console.log(document.location.host)        // www
console.log(document.location.hostname)    // www
console.log(document.location.port || 80)  // 808
console.log(document.location.pathname)    // /pa
console.log(document.location.search)      // ?ke
console.log(document.location.hash)        // #so
```

DOM UML Diagram

A **partial** representation of the DOM:



Script Element

The HTML <script> element is used to associate *JavaScript* code with an *HTML* page (either **embedded** or using a **reference** to a separate document containing the code):

```
<html>
  <head>
    <script src="...url of javascript script...">
    <script>...javascript code goes here...</scri
  </head>
</html>
```

Either way, the <script> tag must be closed.

Defer and Async

By default, when the browser encounters a <script> element, it **pauses** rendering HTML, **fetches** the JavaScript script (if not embedded), and **runs** the corresponding code.

However, as most JavaScript code interacts with the page's HTML, normally we need to have all HTML code parsed and rendered before running it. A common way to solve this problem was to only use the <script> in the end of the HTML document.

A more modern way is to use one of two special attributes of the <script> tag: **defer** and **async**.

```
<script src="...url of javascript script..." defe
<script src="...url of javascript script..." asyn
```

Both don't pause the HTML loading and rendering process, but **defer** only executes the code when the page finishes loading.

Resources

- Reference:
  - WHATWG DOM Specification
  - W3C DOM Specification
  - MDN DOM Reference
- Tutorials:
  - The Modern JavaScript Tutorial
  - JavaScript Style Guide

Elements

Selecting Elements

The following Document and Element **methods** can be used to access specific HTML elements:

- getElementById(id) that returns an Element.
  returns the element with the specified id.
- getElementsByClassName(class) that returns a NodeList.
  returns all elements with the specified class.
- getElementsByTagName(name) that returns a NodeList.
  returns all elements with the specified tag name.
- querySelector(selector) that returns an Element.
  returns the **first** element selected by the specified CSS selector.
- querySelectorAll(selector) that returns a NodeList.
  returns all elements selected by the specified CSS selector.

```
const menu = document.getElementById('menu')
const paragraphs = document.getElementsByTagName(
const intros = document.querySelectorAll('article
const links = menu.querySelectorAll('a')
```

Element

An Element object represents an element in a Document.

More specific elements, like the HTMLElement, will have more specific properties.

Some common Element **properties**:

- id – The element's identifier.
- tagName – The tag name.
- innerHTML – The markup code of the element's content.
- outerHTML – The markup code describing the element, including its descendants.
- textContent – The text content of an Element.
  Inherited from Node.

```
const article = document.querySelector('#posts ar
article.innerHTML = '<p>Changed the content of th
```

In most cases, using innerHTML to add new elements is not a good idea.

Element Attributes

Element attributes can be accessed and modified using the following methods:

- getAttribute(name) – **returns** the value for the attribute with the given name (or null).
- setAttribute(name, value) – **modifies** the attribute with the given name to value.
- removeAttribute(name) – **removes** the attribute with the given name from the element.
- hasAttribute(name) – returns true if the element **has** an attribute with the given name.

```
const link = document.querySelector('#posts a:fir
console.log(link.getAttribute('href'))
link.setAttribute('href', 'http://www.example.com
```

Element Class List

To modify the class attribute of an Element, we should use the classList property that returns a live collection of classes.

This property can then be used to manipulate the class list:

- add(class, ...) – **adds** one or more classes to the class list.
- remove(class, ...) – **removes** one or more classes from the class list.
- replace(oldClass, newClass) – **replaces** *oldClass* with *newClass*.
  But only if the *oldClass* is present. Returns true if the class was replaced.
- toggle(class) – **toggles** class from the class list.
  Returns true if the class was added, false if it was removed.

```
const article = document.querySelector('#posts ar
article.classList.add('selected')
```

Creating Elements

The createElement method of the Document object is the preferred method for creating new elements:

```
const text = 'The quick brown fox jumps over the
const paragraph = document.createElement('p')

paragraph.textContent = text

console.log(paragraph.outerHTML)
```

The **paragraph** variable is a subclass of the HTMLParagraphElement class.

```
<p>The quick brown fox jumps over the lazy dog</p
```

The paragraph still **has not been inserted** anywhere in the *document*.

HTMLElement

The HTMLElement inherits from the Element object.

For each HTML tag, a different class implements (directly or indirectly) this interface. These are some of their **properties**:

- style – The CSS style of the element.
- hidden – Is the element hidden.

And **methods**:

- focus() – Sets keyboard focus on the element.
- blur() – Removes keyboard focus on the element.
- click() – Simulates a mouse click on the element.

HTML*Element

For each *HTML tag*, there is a class implementing the HTMLElement interface.

These are some of them and some of their *attributes* and *methods*:

- HTMLInputElement – name, type, value, checked, autocomplete, autofocus, defaultChecked, defaultValue, disabled, min, max, readOnly, required, select().
- HTMLSelectElement – name, multiple, required, size, length, add(item, before), item(index), remove(index).
- HTMLOptionElement – disabled, selected, defaultSelected, text, value.
- HTMLAnchorElement – href, host, hostname, port, hash, pathname, protocol, text, username, password.
- HTMLImageElement – alt, src, width, height.

HTMLInputElement

Notice the difference between using the **value** attribute, and the **getAttribute** method to get the value of an HTMLInputElement:

```
<form id="register">
  <input type="text" name="username" value="johnd
</form>
```

If the user changes the value to '*johndoe123*':

```
const input = document.querySelector('#register i
console.log(input.getAttribute('value'))  // stil
console.log(input.value)                  // chan
```

Node

The Node object represents a node in the document tree.

The *Element* and *HTMLElement* objects inherit these methods from the *Node* object:

- appendChild(child) – **adds** a node to the end of a parent's node list of children.
- replaceChild(newChild, oldChild) – **replaces** a child of this node with another one.
- removeChild(child) – **removes a child** from this node.
- insertBefore(newNode, referenceNode) – **inserts** a new child **before** the reference child.
- remove() – **removes the element** from its parent.
  From the Element interface.

When adding nodes, if the node already has a parent, it is **first removed** from its current location.

HTMLElement Style

To change the inline style of an HTMLElement, we can use the style object.

Either changing the **whole inline style** at once:

```
const article = document.querySelector('#posts ar
article.style = 'color: red'
```

Or just **one property**:

```
article.style.color = 'red'
```

To **reset** all inline styles we can set the style object to *null* or to an *empty string*:

```
article.style = ''
article.style = null
```

Element and Node

A simple example:

```
// gets the first article
const article = document.querySelector('#posts ar

article.style.color = 'blue'  // changes the text
article.style.padding = '2em' // and the padding

// creates a new paragraph
const paragraph = document.createElement("p")
// inserts text in the paragraph
paragraph.textContent = 'Some text'

article.appendChild(paragraph) // adds the paragr
```

See this example in action.

NodeList

- A NodeList is an object that behaves like an array of elements.
- Functions like querySelectorAll and getElementsByTagName() return a *NodeList*.
- In some cases, the NodeList is live. DOM changes automatically update it.

Items in a *NodeList* can be accessed **by index** like in an array:

```
const paragraphs = document.querySelectorAll('p')
for (let i = 0; i < paragraphs.length; i++) {
  const paragraph = paragraphs[i]
  // do something with the paragraph
}
```

Or using a **for..of** loop:

```
const paragraphs = document.querySelectorAll('p')
for (const paragraph of paragraphs) {
  // do something with the paragraph
}
```

Traversing

Traversing the DOM tree (Node)

The *Node* object has the following properties that allow traversing the DOM tree:

- firstChild and lastChild – first and last node children of this node.
- childNodes – all children nodes as a **live** NodeList.
- previousSibling and nextSibling – previous and next siblings to this node.
- parentNode – parent of this node.
- nodeType – the type of this node.

Be careful, as all these functions return nodes that might not be HTMLElements (*e.g.*, text and comment nodes).

See the complete node type list.

Traversing Example

Consider the following HTML:

```
<section id="posts">
  <h1>Title</h1>
  <p>Some text</p>
</section>
```

And the following *JavaScript*:

```
const posts = document.querySelector('#posts')
console.log(posts.firstChild)
console.log(posts.firstChild.textContent)
console.log(posts.firstChild.nextSibling)
console.log(posts.firstChild.nextSibling.textCont
```

Traversing the DOM tree (Element)

To simplify traversing HTML documents, the following properties have been added to the Element interface:

- firstElementChild and lastElementChild – first and last element children of this node.
- children – all children elements as a NodeList.
- previousElementSibling and nextElementSibling – previous and next element siblings of this node.

```
<section id="posts">
  <h1>Title</h1>
  <p>Some text</p>
</section>
```

```
const posts = document.querySelector('#posts')
console.log(posts.firstElementChild)
console.log(posts.firstElementChild.textContent)
```

Events

Event-driven Architecture

The DOM follows an event-driven architecture (an architecture built on top of the publish-subscribe or observer pattern):

- Events are occurrences that happen in the system.
  e.g., the user clicks on a button.
- Specific events in specific objects can have event handlers attached to them.
- When the event happens, the attached handler is called.

Some possible events:

- Mouse (MouseEvent) – click, dblclick, mouseup, mousenter, mouseleave, mouseover.
- Forms (InputEvent, FocusEvent, FormDataEvent and SubmitEvent) – input, change, focus, blur, formdata, submit.
- Keyboard (KeyboardEvent) – keydown, keyup, keypress.

Events in HTML

A possible way to get notified of events of a particular type (such as click) for a given object is to specify an event handler using an HTML attribute named on{eventtype} on an element.

For example:

```
<button onclick="console.log('User clicked button
  Click me
</button>
```

Or:

```
<button onclick="return handleClick(event)">
  Click me
</button>
```

But, you should not use this!

Events on Element Properties

Another way of attaching an event handler would be by setting the corresponding property.

For example:

```
document.querySelector("button").onclick = functi
  console.log('User clicked button')
}
```

Or:

```
function handleEvent(event) {
  console.log('User clicked button')
}

document.querySelector("button").onclick = handle
```

Add Event Handler

On modern browsers, the addEventListener function is the most common way to attach event handlers.

For example:

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

button.addEventListener('click', function(event){
  console.log('User clicked button')
})
```

Or:

```
function handleEvent() {
  console.log('User clicked button')
}

const button = document.querySelector("button")
button.addEventListener('click', handleEvent)
```

The Event Object

A function that handles an event can receive a parameter representing the event:

- Depending on its type, the event can have different properties and methods.
- We can use the preventDefault() method to ensure that the default behavior is suppressed (*e.g.*, a link isn't followed or a form isn't submitted).

```
function handleEvent(event) {
  console.log('You shall not pass!')
  event.preventDefault()
}

const button = document.querySelector("button")
button.addEventListener('click', handleEvent)
```

See this example in action.

Bubbling

When an event happens on an element, it first runs any handlers attached to it, then on its parent, then up to the root.

In each step, the handler can know the current target (event.currentTarget or *this*) and also the initial target (event.target).

Example where we add some events on all elements and print **this** and **event.target** tag names:

```
<section> <article> <p>Text</p> </article> </section>
```

```
document.querySelector('section').addEventListener('click', function(e
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName
document.querySelector('article').addEventListener('click', function(e
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName
document.querySelector('p').addEventListener('click', function(event)
  console.log('Bubble: ' + this.tagName + " - " + event.target.tagName
```

Clicking on the paragraph:

```
Bubble: P - P
Bubble: ARTICLE - P
Bubble: SECTION - P
```

To stop bubbling, we can use the event.stopPropagation() method.

Capturing

Event processing has two phases:

- Capturing: goes down to the element.
- Bubbling: the event bubbles up from the element.

Although rarely used, the **useCapture** parameter of the addEventListener method allows us to set the event handler on the capturing phase.

Adding capture events to the previous example:

```
document.querySelector('section').addEventListener('click', function(e
  console.log('Capture: ' + this.tagName + " - " + event.target.tagNar
document.querySelector('article').addEventListener('click', function(e
  console.log('Capture: ' + this.tagName + " - " + event.target.tagNar
document.querySelector('p').addEventListener('click', function(event)
  console.log('Capture: ' + this.tagName + " - " + event.target.tagNar
```

```
Capture: SECTION - P
Capture: ARTICLE - P
Capture: P - P
Bubble: P - P
Bubble: ARTICLE - P
Bubble: SECTION - P
```

On Load Event

Besides placing the <script> at the end of the HTML code, another common way of waiting for the DOM to be completely loaded before adding events to any elements is to add any initialization code to the *load* event of the *window* element:

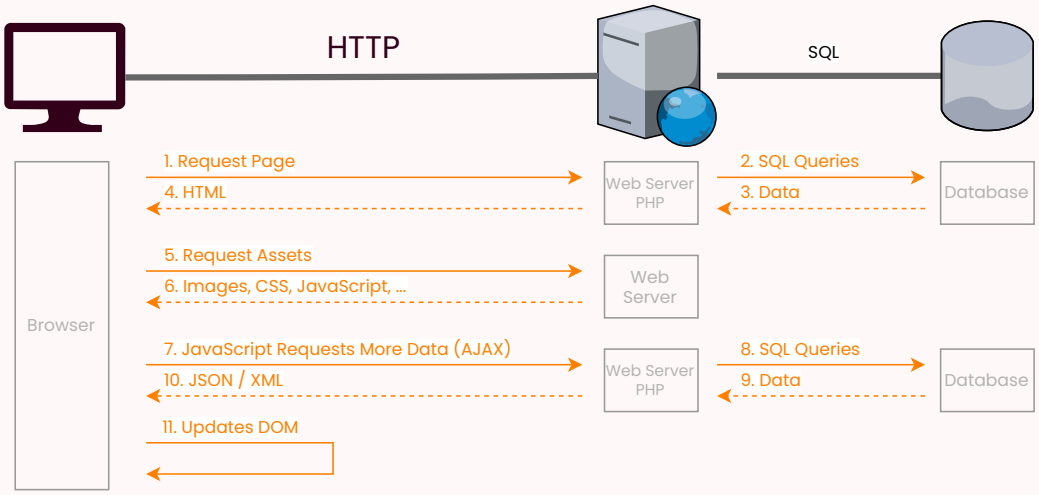
```
window.addEventListener('load', function() {
  // initialization code goes here.
})
```

This is no longer needed as we can now use the defer attribute.

AJAX

AJAX

- Asynchronous JavaScript + XML (JSON is more common now).
- Not a technology in itself, but a term coined in 2005 by **Jesse James Garrett** that describes an approach to using several existing technologies: namely the **XMLHttpRequest** object.



XMLHttpRequest

XMLHttpRequest makes sending HTTP requests from JavaScript very easy.

```
void open(method, url, async)
```

- method: **get** or **post** (or others to be seen later).
- url: The URL to fetch.
- async: if false, execution will stop while waiting for response.
  Default is true.

Example:

```
function requestListener () {
  console.log(this.responseText)
}

const request = new XMLHttpRequest()
request.addEventListener('load', requestListener)
// or request.onload = requestListener
request.open("get", "getdata.php", true)
request.send()
```

Monitoring Progress

Different events let us monitor progress of the request:

```
const request = new XMLHttpRequest()

request.addEventListener("progress", updateProgress)
request.addEventListener("load", transferComplete)
request.addEventListener("error", transferFailed)
request.addEventListener("abort", transferCanceled)

request.open("get", "getdata.php", true)
request.send()

function updateProgress (event) {
  if (event.lengthComputable)
    const percentComplete = event.loaded / event.total
}

function transferComplete(event) {
  console.log("The transfer is complete.")
}

function transferFailed(event) {
  console.log("An error occurred while transferring the file.")
}

function transferCanceled(event) {
  console.log("The transfer has been canceled by the user.")
}
```

Sending data

To send data to the server, we first must encode it properly:

We are simplifying, there are other ways of doing this.

```
function encodeForAjax(data) {
  return Object.keys(data).map(function(k){
    return encodeURIComponent(k) + '=' + encodeUR
  }).join('&')
}
```

Sending it using **get**:

```
request.open("get",
  "getdata.php?" + encodeForAjax({id: 1, name: 'J
request.send()
```

Sending it using **post**:

```
request.open("post", "getdata.php", true)
request.setRequestHeader('Content-Type',
  'application/x-www-form-urlencoded')
request.send(encodeForAjax({id: 1, name: 'John'})
```

Analyzing the Response

If the server responds in **XML** format, the responseXML property will be a DOM Object containing a parsed XML document, which can be hard to manipulate and analyze.

If the server responds in **JSON**, it is straightforward to parse the responseText property:

```
const request = new XMLHttpRequest()
request.addEventListener("load", transferComplete
request.open("get", "getdata.php", true)
request.send()

function transferComplete() {
  const response = JSON.parse(this.responseText)
}
```

AJAX with Promises

The Fetch API is a more modern interface for fetching remote resources:

- The global fetch() can be used to fetch a remote resource.
- It returns a *Promise* that will eventually be **fulfilled** as a Response.
- It will only get **rejected** if there was a network or permission error (*i.e.*, any response from the server is fulfilled).

```
async function getData() {
    return fetch('https://example.com/')
}

getData().then(response => {
  console.log(response)
}).catch(() => {
  console.error('Network Error')
})
```

Response

Because only failed requests get rejected, the response must be checked:

- ok – Boolean indicating if the response was successful (*i.e.*, the status is in the 200–299 interval).
- status – The status code of the response (*e.g.*, 200, 404).
- redirected – Indicates if the request was redirected to another URL.
- url – Final URL after redirects.
- body – The body of the response (*i.e.*, the data).

```
getData().then(response => {
  if (response.ok)
    console.log(response.body)
  else
    console.error(`Error: code ${request.status}`
}).catch(() => {
  console.error('Network Error')
})
```

JSON Response

To parse a JSON Response we use the json() method that also returns a promise:

```
getData()
  .catch(() => console.error('Network Error'))
  .then(response => response.json())
  .catch(() => console.error('Error parsing JSON'
  .then(json => console.log(json))
```

If the Response is text, then we use the text() method that also returns a promise:

```
getData()
  .catch(() => console.error('Network Error'))
  .then(response => response.text())
  .then(text => console.log(text))
```

Using Await

This can all be simplified by using the async/await mechanism:

```
async function getJsonData() {
  const response = await getData()
  const content = await response.json()
}
```

Or for text:

```
async function getTextData() {
  const response = await getData()
  const content = await response.text()
}
```

Request

For more complicated requests, the fetch() method can receive an object with extra parameters:

```
async function postData(data) {
  return fetch('https://example.com/', {
    method: 'post',
    headers: {
      'Content-Type': 'application/x-www-form-url
    },
    body: encodeForAjax(data)
  })
}

postData({id: 100, name: 'John'})
  .catch(() => console.error('Network Error'))
  .then(response => response.json())
  .catch(() => console.error('Error parsing JSON'
  .then(json => console.log(json))
```

More on this will be discussed when we study the **HTTP** protocol in depth.

Timers

Set Timeout

The window.setTimeout(funtion, delay) fuction sets a timer which executes a function once after a certain delay:

```
const id = window.setTimeout(function() {
  console.log('5 seconds later!')
}, 5000)
```

The return value is an *id* that can be used to cancel the timer:

```
window.clearTimeout(id)
```

Set Interval

The window.setInterval(function, interval) is similar but executes the function until it is stopped with a fixed time delay between calls.

```
let counter = 1
const id = window.setInterval(function() {
  console.log(`${counter++}s later!`)
}, 1000)
```

The return value is an *id* that can be used to cancel the timer:

```
window.clearInterval(id)
```

Advanced DOM

Closures and Events

Let's start by thinking if this code should work...

```
const paragraphs = document.querySelectorAll('p')
for (let i = 0; i < paragraphs.length; i++)
  paragraphs[i].addEventListener('click', functio
      console.log('I am paragraph #' + i)
  })
```

Closures and Events

Let's start by thinking if this code should work...

```
const paragraphs = document.querySelectorAll('p')
for (let i = 0; i < paragraphs.length; i++)
  paragraphs[i].addEventListener('click', functio
      console.log('I am paragraph #' + i)
  })
```

When we click a paragraph, what will be the value of the *i* variable? Let's test it.

Closures and Events

Let's start by thinking if this code should work...

```
const paragraphs = document.querySelectorAll('p')
for (let i = 0; i < paragraphs.length; i++)
  paragraphs[i].addEventListener('click', functio
      console.log('I am paragraph #' + i)
  })
```

When we click a paragraph, what will be the value of the *i* variable? Let's test it.

The only reason why this code works as intended is that each time an event handler is added, a new function is created with a **different closure** (and a different *i* variable in that closure with a **different value**).

> *"When a function is created, it **retains** the lexical environment in which it was **created**."*
> — *Closures, JavaScript Slides.*

Bind and Events

Sometimes we **lose** our *this*:

```
class Foo {
  setup() {
    document.querySelector('h1').addEventListener
  }

  bar(event) {
    // we want to get the Foo object, but:
    console.log(this)         // the h1 element
    console.log(event.target) // the h1 element
  }
}

new Foo().setup()
```

We can **fix** it using *bind*:

```
setup() {
  document.querySelector('h1')
    .addEventListener('click', this.bar.bind(this
}
```

Partial Functions

We might want to call a function with parameters that depend on the element:

```
document.querySelector('p.blue').addEventListener('click', changeColo
document.querySelector('p.red').addEventListener('click', changeColor(

function changeColor(color) {
  this.style.color = color
}
```

But it obviously doesn't work.

A solution would be to create **anonymous functions**:

```
document.querySelector('p.blue').addEventListener('click', function(ev
  changeColor('blue', event)}
)
document.querySelector('p.red').addEventListener('click', function(eve
  changeColor('red', event)}
)

function changeColor(color, event) {
  event.target.style.color = color
}
```

Partial Functions

Another, more elegant solution, would be to create **partial functions** using *bind*:

```
const blue = document.querySelector('p.blue')
blue.addEventListener('click', changeColor.bind(b

const red = document.querySelector('p.red')
red.addEventListener('click', changeColor.bind(re

function changeColor(color) {
  this.style.color = color
}
```

Mapping Selectors

We already saw how we could use the map() function with *non-array* iterables (like a NodeList).

One way we can use this feature:

```
const inputs = document.querySelectorAll('input[t
const values = [].map.call(inputs, input => input
console.log(values) // an array with all the numb
```

See this example in action.

Selectors to Arrays

Other times we just want to **convert** a NodeList to an **array**, so we can use functions like map(), reduce(), and filter():

```
const paragraphs = document.querySelectorAll('p')
```

There are several ways to achieve this:

```
const array1 = Array.apply(null, paragraphs)
const array2 = Array.prototype.slice.call(paragra
const array3 = [].slice.call(paragraphs)
const array4 = [...paragraphs]
```

HTML5 Data Attributes

This is not really JavaScript!

HTML5 data-* attributes allow us to store extra information on standard, semantic HTML elements without using hacks.

This can be useful, for example, to store the id of a certain database tuple to be used in an Ajax call.

```
<ul>
  <li data-id="1">Apple</li>
  <li data-id="2">Banana</li>
  <li data-id="3">Pear</li>
</ul>
```