FREE eBook

LEARNING DOM

Free unaffiliated eBook created from
Stack Overflow contributors.

#dom

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dom

It is an unofficial and free DOM ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official DOM.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with DOM

Remarks

The DOM, or Document Object Model, is the API used by web browsers and other applications to access the contents of an HTML document.

The DOM represents the structure as a tree, nodes can contain child-nodes, nodes with no children are said leaf nodes.

With it, one can manipulate the structure and properties of the document and its constituent parts.

Major topics include finding elements, accessing style information, and animation.

Most work with the DOM is done using the JavaScript language, but the API is open to any language.

Versions

W3C DOM

Version	Release Date
1	1998-10-01
2 (Core)	2000-11-13
3 (Core)	2004-04-07
4	2013-11-07

Selectors API Level

Version	Release Date
1	2013-02-21

Examples

Retrieving existing html elements

One of the most common tasks is retrieving an existing element from the DOM to manipulate. Most commonly these methods are executed on `document`, because it is the root node, but all these

methods work on any HTML element in the tree. They will only return children from the node it is executed on.

Retrieve by id

```
var element = document.getElementById("logo");
```

`element` will contain the (only) element that has its `id` attribute set to "logo", or contains `null` if no such element exists. If multiple elements with this id exist, the document is invalid, and anything can happen.

Retrieve by tag name

```
var elements = document.getElementsByTagName("a");
```

`elements` will contain a *live* `HTMLCollection` (an array-like object) of all link tags in the document. This collection is in sync with the DOM, so any changes made to the DOM are reflected in this collection. The collection provides random access and has a length.

```
var element = elements[0];
//Alternative
element = elements.item(0);
```

> `element` contains the first encountered HTML link element, or `null` if the index is out of bounds

```
var length = elements.length;
```

> `length` is equal to the number of HTML link elements currently in the list. This number can change when the DOM is changed.

Retrieve by class

```
var elements = document.getElementsByClassName("recipe");
```

`elements` will contain a *live* `HTMLCollection` (an array-like object) of all elements where their `class` attribute includes "recipe". This collection is in sync with the DOM, so any changes made to the DOM are reflected in this collection. The collection provides random access and has a length.

```
var element = elements[0];
//Alternative
element = elements.item(0);
```

> `element` contains the first encountered HTML element with this class. If there are no such elements, `element` has the value `undefined` in the first example and `null` in the

second example.

```
var length = elements.length;
```

`length` is equal to the number of HTML elements that currently have the class "recipe". This number can change when the DOM is changed.

Retrieve by name

```
var elements = document.getElementsByName("zipcode");
```

`elements` will contain a *live* `NodeList` (an array-like object) of all elements with their `name` attribute set to "zipcode". This collection is in sync with the DOM, so any changes made to the DOM are reflected in this collection. The collection provides random access and has a length.

```
var element = elements[0];
//Alternative
element = elements.item(0);
```

`element` contains the first encountered HTML element with this name.

```
var length = elements.length;
```

`length` is equal to the number of HTML elements that currently have "zipcode" as their name attribute. This number can change when the DOM is changed.

Getting started

The DOM (Document Object Model) is the programming interface for HTML and XML documents, it defines the logical structure of documents and the way a document is accessed and manipulated.

The main implementers of the DOM API are web browsers. Specifications are standardized by the W3C and the WHATWG groups, and the object model specifies the logical model for the programming interface.

The representation of DOM structure resembles a tree-like view, where each node is an object representing a part of the markup, depending on the type each element also inherits specific and shared functionalities.

The name "Document Object Model" was chosen because it is an "object model" in the traditional object oriented design sense: documents are modeled using objects, and the model encompasses not only the structure of a document, but also the behavior of a document and the objects of which it is composed. In other words, taking the example HTML diagram, the nodes do not represent a data structure, they represent objects, which have functions and identity. As an object model, the Document Object Model identifies:

- the interfaces and objects used to represent and manipulate a document
- semantics of these interfaces and objects - including both behavior and attributes
- the relationships and collaborations among these interfaces and objects

Wait for DOM to be loaded

Use `DOMContentLoaded` when the `<script>` code interacting with DOM is included in the `<head>` section. If not wrapped inside the `DOMContentLoaded` callback, the code will throw errors like

> Cannot read something of `null`

```
document.addEventListener('DOMContentLoaded', function(event) {
    // Code that interacts with DOM
});
```

https://html.spec.whatwg.org/multipage/syntax.html#the-end

Alternative to `DOMContentLoaded`

An alternative (suitable for **IE8**)

```
// Alternative to DOMContentLoaded
document.onreadystatechange = function() {
    if (document.readyState === "interactive") {
        // initialize your DOM manipulation code here
    }
}
```

https://developer.mozilla.org/en/docs/Web/API/Document/readyState

Use innerHTML

HTML

```
<div id="app"></div>
```

JS

```
document.getElementById('app').innerHTML = '<p>Some text</p>'
```

and now HTML looks like this

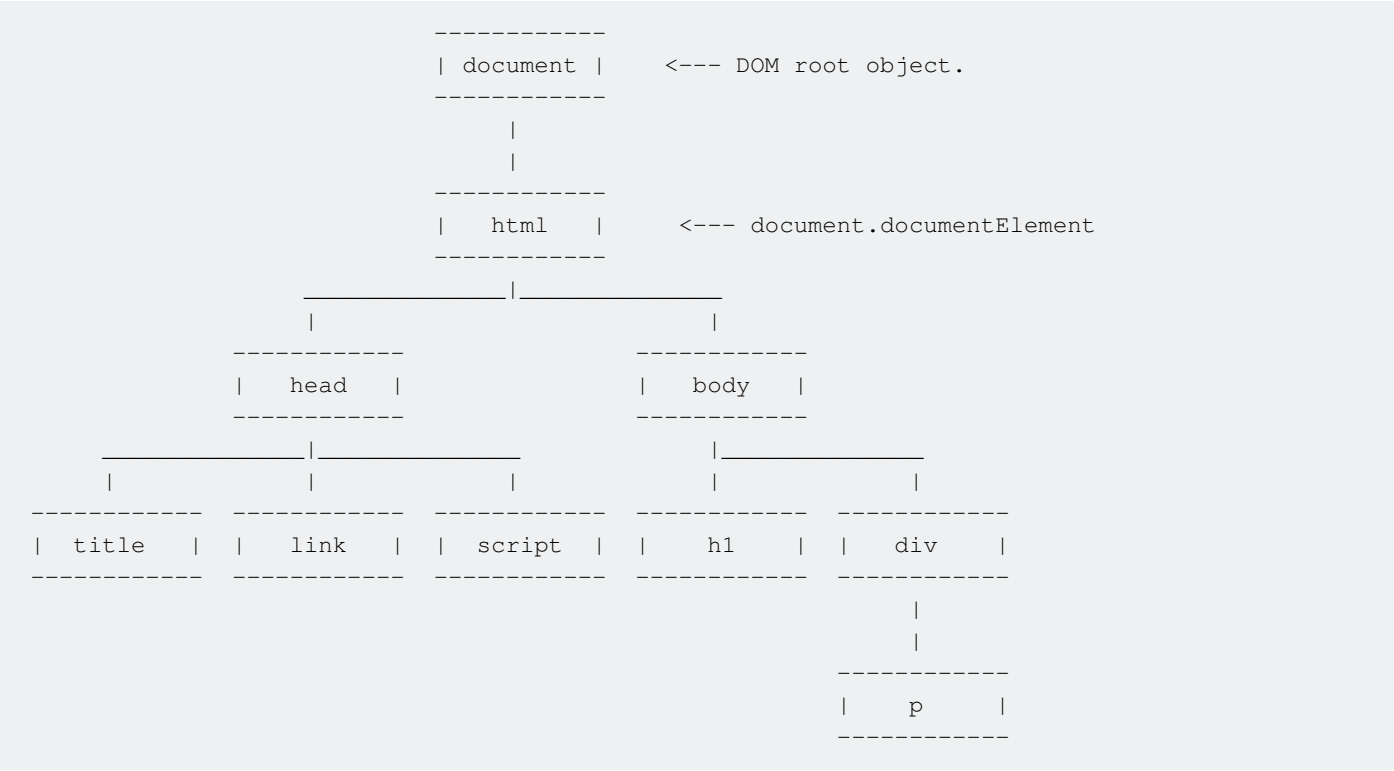
```
<div id="app">
    <p>Some text</p>
</div>
```

HTML markup

example input:

```
<html>
  <head>
    <title>the title</title>
    <link href='css/app.css' type='text/css' rel='stylesheet'>
    <script src='js/app.js'></script>
  </head>
  <body>
    <h1>header</h1>
    <div>
      <p>hello!</p>
    </div>
  </body>
</html>
```

DOM element output:



All the above elements inherit from HTMLElement interface and get customized depending on specific tag

Read Getting started with DOM online: https://riptutorial.com/dom/topic/2584/getting-started-with-dom

Chapter 2: Events

Parameters

Parameter	Description
type	`String` defines the name of the event to listen to.
listener	`Function` triggers when the event occurs.
options	`Boolean` to set capture, if `Object` you can set the following properties on it, notice that the object option is weakly supported.
1. *capture*	A Boolean that indicates that events of this type will be dispatched to the registered listener before being dispatched to any EventTarget beneath it in the DOM tree.
2. *once*	A Boolean indicating that the listener should be invoked at most once after being added. If it is true, the listener would be removed automatically when it is invoked.
3. *passive*	A Boolean indicating that the listener will never call preventDefault(). If it does, the user agent should ignore it and generate a console warning.

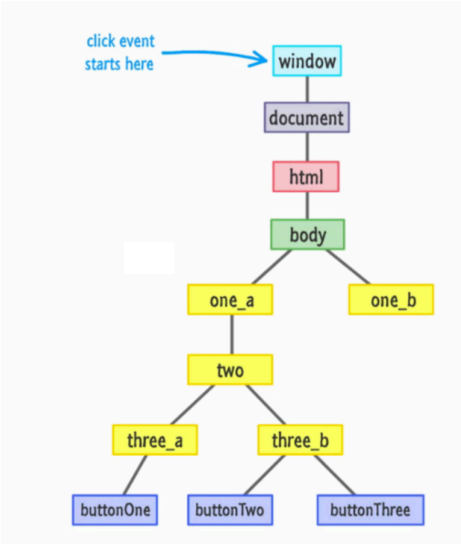
Remarks

Origin of events

Events dont start at the thing you trigger the event on (a button for example).

Instead

It touches every element in its path and it inform every element that an event is happening. Events also go back up after they reach their destination, informing the elements again of its occurrence.



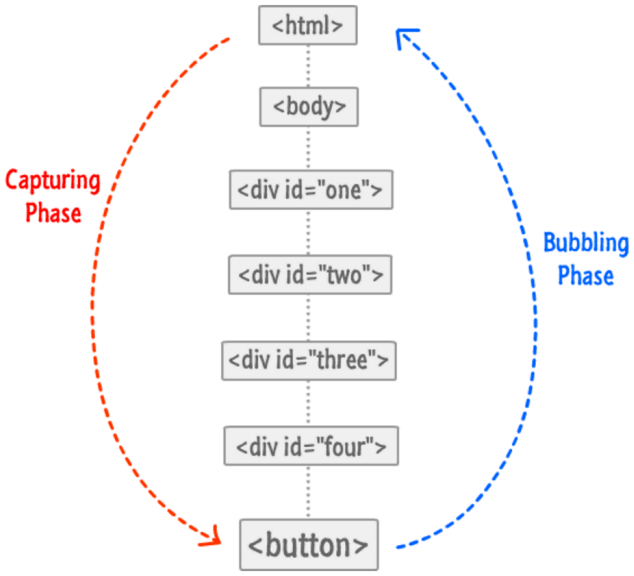
Capturing & Bubbling

As we learned, events start from the top of DOM tree, informs every node in its path down to its destination, then goes back up when it reaches its destination, also informing every element it touches on its way up about its occurrence.

Events going down the DOM tree are in the **capturing phase**, events going up the

DOM tree are in the **bubbling phase**.

By default events are listened to in the bubbling phase. To change this you can specify which phase the event gets listened to by specifying the third parameter in the addEventListener function. (code example in the *capture* section)



Examples

Introduction

Definition:

> In computing, an event is an action or occurrence recognized by software that may be handled by the software. Computer events can be generated or triggered by the system, by the user or in other ways. *Definition Source*

HTML events are "things" that happen to HTML elements. JavaScript can "react" on these events. via `Event Listeners`. Additionally, custom events can be triggered using `dispatchEvent`. But this is only an introduction, so lets get started!

Basic Event Listener

To listen to events, you call `target.addEventListener(type, listener);`

```
function loadImage() {
  console.log('image code here!');
}
var myButton = document.querySelector('#my-button');
myButton.addEventListener('click', loadImage);
```

This will trigger loadImage every time `my-button` is clicked.

Event listeners can be attached to any node in the DOM tree. to see a full list of all the events natively triggered in the browser: go here MDN link for full event list

Removing event listeners

The removeEventListener() method removes event handlers that have been attached with the

addEventListener() method:

```
element.removeEventListener("mousemove", myFunction);
```

Everything (eventname, function, and options) in the `removeEventListener` must match the one set when adding the event listener to the element.

.bind with removeListener

using `.bind` on the function when adding an event listener will prevent the function from being removed, to actually remove the eventListener you can write:

```
function onEvent() {
   console.log(this.name);
}

var bindingOnEvent = onEvent.bind(this);

document.addEventListener('click', bindingOnEvent);

...

document.removeEventListener('click', bindingOnEvent);
```

listen to an event only once

Until `once` option is widely supported, we have to manually remove the even listener once the event is triggered for the first time.

This small helper will help us achieve this:

```
Object.prototype.listenOnce = Object.prototype.listenOnce ||
  function listenOnce(eventName, eventHandler, options) {
      var target = this;
      target.addEventListener(eventName, function(e) {
          eventHandler(e);
          target.removeEventListener(eventName, eventHandler, options);
      }, options);
  }

var target = document.querySelector('#parent');
target.listenOnce("click", clickFunction, false);
```

**It is not a best practice to attach functions to the Object prototype, hence you can remove the first line of this code and add a target to it as a first param.*

Waiting for the document to load

One of the most commonly used events is waiting for the document to have loaded, including both

script files and images. The `load` event on `document` is used for this.

```
document.addEventListener('load', function() {
   console.log("Everything has now loaded!");
});
```

Sometimes you try to access a DOM object before it is loaded, causing null pointers. These are really tough to debug. To avoid this use `document`'s `DOMContentLoaded` event instead. `DOMContentLoaded` ensures that the HTML content has been loaded and initialized without waiting for other external resources.

```
document.addEventListener('DOMContentLoaded', function() {
   console.log("The document contents are now available!");
});
```

Event Object

To access the event object, include an `event` parameter in the event listener callback function:

```
var foo = document.getElementById("foo");
foo.addEventListener("click", onClick);

function onClick(event) {
  // the `event` parameter is the event object
  // e.g. `event.type` would be "click" in this case
};
```

e.stopPropagation();

HTML:

```
<div id="parent">
   <div id="child"></div>
</div>
```

Javascript:

```
var parent = document.querySelector('#parent');
var child = document.querySelector('#child');

child.addEventListener('click', function(e) {
   e.stopPropagation();
   alert('child clicked!');
});

parent.addEventListener('click', function(e) {
   alert('parent clicked!');
});
```

since the child stops the event propagation, and the events are listened to during bubbling phase,

clicking on the child will only trigger the child. without stopping the propagation both events will be triggered.

e.preventDefault();

The `event.preventDefault()` method stops the default action of an element from happening.

For example:

- Prevent a submit button from submitting a form
- Prevent a link from following the URL

```
var allAnchorTags = document.querySelector('a');

allAnchorTags.addEventListener('click', function(e){
    e.preventDefault();
    console.log('anchor tags are useless now! *evil laugh*');
});
```

e.target vs e.currentTarget

> e.currentTarget Identifies the current target for the event, as the event traverses the DOM. It always refers to the element the event handler has been attached to as opposed to event.target which identifies the element on which the event occurred.

in other words

`e.target` will return what triggers the event dispatcher to trigger

`e.currentTarget` will return what you assigned your listener to.

HTML:

```
<body>
   <button id="my-button"></button>
</body>
```

Javascript:

```
var body = document.body;
body.addEventListener( 'click', function(e) {
    console.log('e.target', e.target);
    console.log('e.currentTarget', e.currentTarget);
});
```

if you click `my-button`,

- **e.target** will be `my-button`
- **e.currentTarget** will be `body`

Event Bubbling and Capturing

Events fired on DOM elements don't just affect the element they're targeting. Any of the target's ancestors in the DOM may also have a chance to react to the event. Consider the following document:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
</head>
<body>
  <p id="paragraph">
    <span id="text">Hello World</span>
  </p>
</body>
</html>
```

If we just add listeners to each element without any options, then trigger a click on the span...

```
document.body.addEventListener('click', function(event) {
  console.log("Body clicked!");
});
window.paragraph.addEventListener('click', function(event) {
  console.log("Paragraph clicked!");
});
window.text.addEventListener('click', function(event) {
  console.log("Text clicked!");
});

window.text.click();
```

...then the event will **bubble** up through each ancestor, triggering each click handler on the way:

```
Text clicked!
Paragraph clicked!
Body clicked!
```

If you want one of your handlers to stop the event from triggering any more handlers, it can call the `event.stopPropagation()` method. For example, if we replace our second event handler with this:

```
window.paragraph.addEventListener('click', function(event) {
  console.log("Paragraph clicked, and that's it!");
  event.stopPropagation();
});
```

We would see the following output, with `body`'s `click` handler never triggered:

```
Text clicked!
Paragraph clicked, and that's it!
```

Finally, we have the option to add event listeners that trigger during "**capture**" instead of bubbling. Before an event bubbles up from an element through its ancestors, it's first "captured" down to the element through its ancestors. A capturing listener is added by specifying `true` or `{capture: true}` as the optional third argument to `addEventListener`. If we add the following listeners to our first example above:

```
document.body.addEventListener('click', function(event) {
  console.log("Body click captured!");
}, true);
window.paragraph.addEventListener('click', function(event) {
  console.log("Paragraph click captured!");
}, true);
window.text.addEventListener('click', function(event) {
  console.log("Text click captured!");
}, true);
```

We'll get the following output:

```
Body click captured!
Paragraph click captured!
Text click captured!
Text clicked!
Paragraph clicked!
Body clicked!
```

By default events are listened to in the bubbling phase. To change this you can specify which phase the event gets listened to by specifying the third parameter in the addEventListener function. (To learn about capturing and bubbling, check *remarks*)

```
element.addEventListener(eventName, eventHandler, useCapture)
```

useCapture: `true` means listen to event when its going down the DOM tree. `false` means listen to the event while its going up the DOM tree.

```
window.addEventListener("click", function(){alert('1: on bubble')}, false);
window.addEventListener("click", function(){alert('2: on capture')}, true);
```

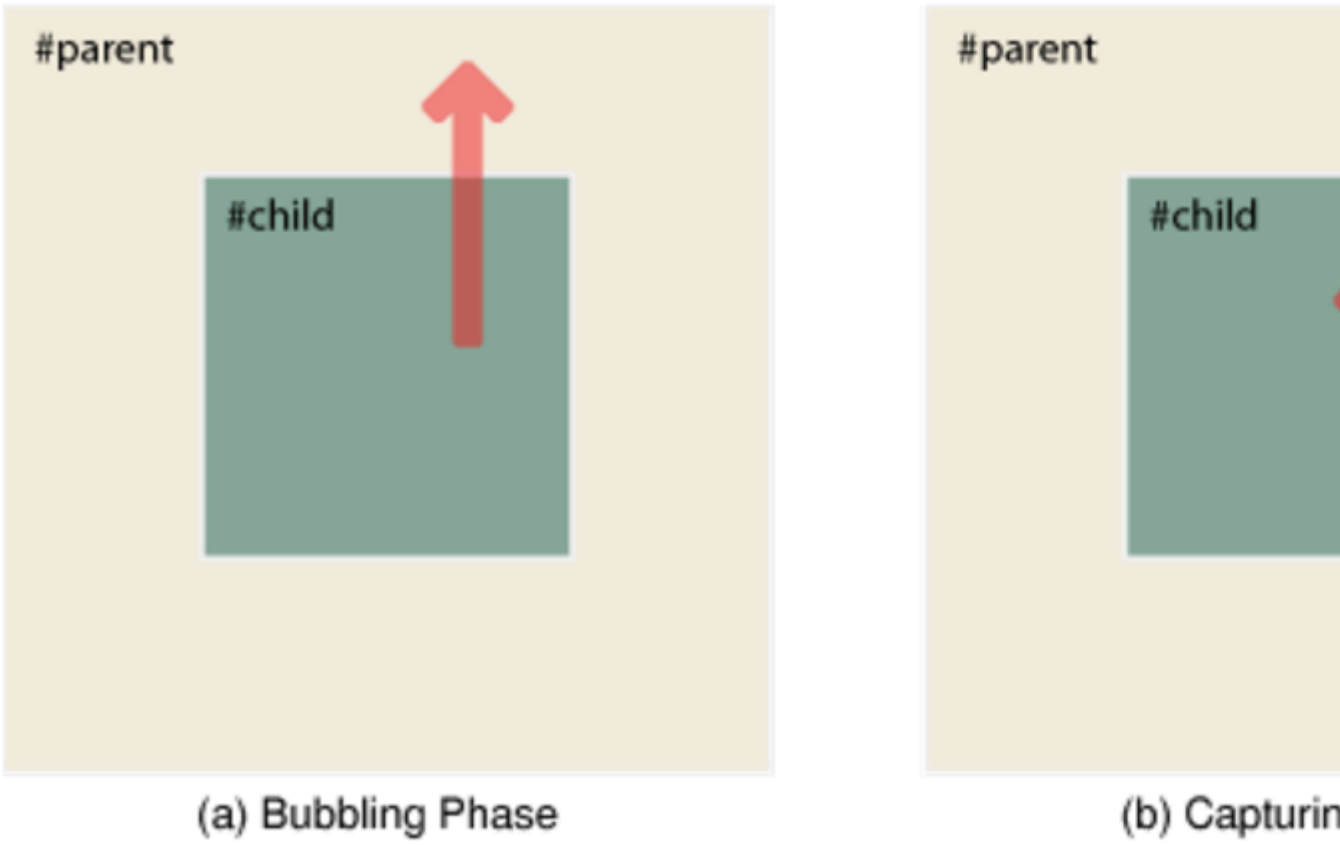
The alert boxes will pop up in this order:

- 2: on capture
- 1: on bubble

Real-world use cases

Capture Event will be dispatch before Bubble Event, hence you can ensure than an event is listened to first if you listen to it in its capture phase.

if you are listening to a click event on a parent element, and another on its child, you can listen to the child first or the parent first, depending on how you change the useCapture parameter.



in bubbling, child event gets called first, in capture, parent first

HTML:

```
<div id="parent">
   <div id="child"></div>
</div>
```

Javascript:

```
child.addEventListener('click', function(e) {
   alert('child clicked!');
});

parent.addEventListener('click', function(e) {
   alert('parent clicked!');
}, true);
```

Setting true to the parent eventListener will trigger the parent listener first.

Combined with e.stopPropagation() you can prevent the event from triggering the child event listener / or the parent. (more about that in the next example)

Event Delegation

Event delegation is a process which allow us to avoid adding event listeners to specific nodes; instead, the event listener is added to parent node. This mechanism utilizes the event propagation/bubbling to handle an event at a higher level element/node in the DOM instead of using the element on which the event was originated. For example, think we need to add events for the following list elements:

```
<ul id="container">
    <li id="item-1" class="new">Item 1</li>
    <li id="item-2">Item 2</li>
    <li id="item-3">Item 3</li>
</ul>
```

We need to add `click` handlers and basically, we can add listeners to each element using a loop but imagine that, we want to add elements dynamically. So, we register all the event handlers when the DOM is loaded and after the DOM initializes and registers all the event handlers for each element, the newly inserted element into the above `UL` will not respond on click because that element was not present in the DOM when we've registered the click event listeners.

So, to overcome this problem, we may leverage the event delegation. Which means, instead of registering the listeners to each `li` elements themselves, we can bind the event listener to it's parent `UL` element for example:

```
document.getElementById("container").addEventListener("click", function(e) {
    console.log("List item " e.target.id, " was clicked!");
});
```

Since, the event propagates (bubbles upwards) by default, then clicking on any `LI` element will make the `UL` element to fire the same event as well. In this case, we can use the `e` parameter in the function, which is actually the event object and it carries helpful information about the event including the original element, which initiated the event. So, for example, we can use something like the following:

```
document.getElementById("container").addEventListener("click", function(e) {

    // If UL itself then no action is require
    if(e.target.nodeName == 'UL') return false;

    if(e.target.classList.contains('new')) {
        console.log("List item " e.target.id, " was clicked and it's new!");
    }
});
```

So, it's obvious that, `e` (Event Object) allow us to examine the source element (e.target) and we can easily inject new elements to the `UL` after DOM is loaded and the only one delegated event handler will handle all the click events within the parent `UL` which is also less memory consuming because we declared only one function for all the elements.

Triggering custom events

The CustomEvent API allows developers to create custom events and trigger them on DOM nodes, passing data along the way.

```
event = new CustomEvent(typeArg, customEventInit);
```

typeArg - DOMString representing the name of the event.

customEventInit - is optional parameters (that will be passed as `e` in following example).

You can attach `eventListeners` to `document` or *any* HTML element.

Once custom event has been added and bound to element (or document) one might want to manually fire it from javascript.

```
document.addEventListener("event-name", function(e) {
  console.log(e.detail); // logs custom object passed from the event.
});

var event = new CustomEvent("event-name", { "param-name": "param-value" });
document.dispatchEvent(event);
```

Read Events online: https://riptutorial.com/dom/topic/5388/events

Chapter 3: Manipulating a list of CSS classes

Examples

Adding a class

Modern browsers provide a `classList` object to ease manipulation of the element's class attribute. Older browsers require direct manipulation of the element's `className` property.

W3C DOM4

A simple method to add a class to an element is to append it to the end of the `className` property. This will not prevent duplicate class names, and spaces **must** be included between class names.

```
document.getElementById("link1").className += " foo";
document.getElementById("link2").className += " foo bar";
```

For multiple elements, you'll need to add the class names inside of a loop

```
var els = document.getElementsByClassName("foo"),
  indx = els.length;
while (indx--) {
  els[indx].className += " bar baz";
}
```

W3C DOM4

A single class name may be added as a string. To add multiple class names, use ES6's spread operator:

```
document.querySelector("#link1").classList.add("foo");
document.querySelector("#link2").classList.add(...['foo', 'bar']);
```

For multiple elements, you'll need to add the class names inside of a loop

```
document.querySelectorAll(".foo").forEach(el => {
  el.classList.add(...['bar', 'baz']);
});
```

Removing a class

Modern browsers provide a `classList` object to ease manipulation of the element's class attribute. Older browsers require direct manipulation of the element's `className` property.

* Note class names are not stored in the element's property in any particular order

W3C DOM4

Removing one class from an element requires a bit of manipulation of the `className` property.

```
var toRemove = "bar",
  el = document.getElementById("link1");
el.className = el.className.replace(new RegExp("\\b" + toRemove + "\\b", "g"), "").trim();
```

Removing multiple class names would require a loop. The remaining examples will use a function to isolate the work

```
function removeClass(el, name) {
  name = name.split(/\s+/);
  var index = name.length,
    classes = el.className;
  while (index--) {
    classes = classes.replace(new RegExp("\\b" + name[index] + "\\b", "g"), "").trim();
  }
  el.className = classes;
}
var el = document.getElementById("link1");
removeClass(el, "bar baz");
```

Multiple elements with multiple class names to remove would require two loops

```
function removeClass(els, name) {
  name = name.split(/\s+/);
  var regex, len,
    index = name.length;
  while (index--) {
    regex = new RegExp("\\b" + name[index] + "\\b", "g");
    len = els.length;
    while (len--) {
      els[len].className = els[len].className.replace(regex, "").trim();
    }
  }
}
var els = document.getElementsByTagName("a");
removeClass(els, "bar baz");
```

W3C DOM4

A single class name may be removed as a string. To remove multiple class names, use ES6's spread operator:

```
document.querySelector("#link1").classList.remove("foo");
document.querySelector("#link2").classList.remove(...['foo', 'bar']);
```

For multiple elements, you'll need to remove the class names inside of a loop

```
document.querySelectorAll(".foo").forEach(el => {
  el.classList.remove(...['bar', 'baz']);
});
```

Testing for a class

Modern browsers provide a `classList` object to ease manipulation of the element's class attribute. Older browsers require direct manipulation of the element's `className` property.

* Note class names are not stored in the element's property in any particular order

W3C DOM4

Testing if an element contains a class requires a bit of manipulation of the `className` property. This example is using an array method to test for the class.

```
function hasClass(el, name) {
  var classes = (el && el.className || "").split(/\s+/);
  return classes.indexOf(name) > -1;
}
var el = document.getElementById("link1");
console.log(hasClass(el, "foo"));
```

Testing for multiple class names would require a loop.

```
function hasClass(el, name) {
  name = name.split(/[\s.]+/);
  var hasClass = true,
    classes = (el && el.className || "").split(/\s+/),
    index = name.length;
  while (index--) {
    hasClass = hasClass && classes.indexOf(name[index]) > -1;
  }
  return hasClass;
}
var el = document.getElementById("link1");
console.log(hasClass(el, "foo"));
```

Instead of using `.indexOf()`, you may also consider using a regular expression.

```
function hasClass(el, name) {
  return new RegExp("\\b" + name+ "\\b").test(el.className);
}
var el = document.getElementById("link1");
console.log(hasClass(el, "foo"));
```

W3C DOM4

Testing for a single class name is done as follows:

```
var hasClass = document.querySelector("#link1").classList.contains("foo");
```

For multiple class names, it is easier to use `matches`. Note the use of the class selector; The selector can be any valid string selector (id, attribute, pseudo-classes, etc).

```
var hasClass = document.querySelector("#link1").matches('.foo.bar');
var hasClass = document.querySelector("#link2").matches('a.bar[href]');
```

Read Manipulating a list of CSS classes online:

https://riptutorial.com/dom/topic/5865/manipulating-a-list-of-css-classes

Chapter 4: Manipulating Attributes

Remarks

Attributes are a specific type of object in the DOM API. In earlier versions of the DOM API, they inherited from the `Node` type, but this was changed in version 4.

In the examples referring to `dataset`, "modern browsers" specifically excludes versions of Internet Explorer less than 11. See caniuse.com for more up to date information.

Examples

Getting an attribute

Some attributes are directly accessible as properties of the element (e.g. `alt`, `href`, `id`, `title` and `value`).

```
var a = document.querySelector("a"),
   url = a.href;
```

Other attributes, including data-attributes can be accessed as follows:

```
var a = document.querySelector("a"),
   tooltip = a.getAttribute("aria-label");
```

Data attributes can also be accessed using `dataset` (modern browsers)

```
// <a href="#" data-tracking-number="ABC-123">Widget</a>
var a = document.querySelector("a"),
  tracker = a.dataset.trackingNumber;
```

Setting an attribute

Some attributes are directly accessible as properties of the element (e.g. alt, href, id, title and value).

```
document.querySelector("a").href = "#top";
```

Other attributes, including data-attributes can be set as follows:

```
document.querySelector("a").setAttribute("aria-label", "I like turtles");
```

Data attributes can also be set using dataset (modern browsers)

```
var a = document.querySelector("a");
```

```
a.dataset.test = "123";
a.dataset['test-2'] = "456";
```

results in

```
<a href="#" data-test="123" data-test-2="456">Widget</a>
```

Removing an attribute

To remove an attribute, including directly accessible properties

```
document.querySelector("a").removeAttribute("title");
```

Data attributes can also be removed as follows (modern browsers):

```
// remove "data-foo" attribute
delete document.querySelector("a").dataset.foo;
```

Read Manipulating Attributes online: https://riptutorial.com/dom/topic/5236/manipulating-attributes

Chapter 5: Manipulating Elements

Examples

Cloning elements

An element can be cloned by invoking the `cloneNode` method on it. If the first parameter passed to `cloneNode` is `true`, the children of the original will also be cloned.

```
var original = document.getElementsByTagName("li")[0];
var clone = original.cloneNode(true);
```

Adding an element

In this example we create a new list element with the text "new text", and select the first unordered list, and its first list element.

```
let newElement = document.createElement("li");
newElement.innerHTML = "new text";

let parentElement = document.querySelector("ul");
let nextSibling = parentElement.querySelector("li");
```

When inserting an element, we do it *under* the parent element, and just before a particular child element of that parent element.

```
parentElement.insertBefore(newElement, nextSibling);
```

> The new element is inserted under `parentElement` and just before `nextSibling`.

When one wants to insert an element as the last child element of `parentElement`, the second argument can be `null`.

```
parentElement.insertBefore(newElement, null);
```

> The new element is inserted under `parentElement` as the last child.

Instead, `appendChild()` may be used to simply append the child to the children of the parent node.

```
parentElement.appendChild(newElement);
```

> The new element is inserted under `parentElement` as the last child.

Replacing an element

In this example we create a new list element with the text "new text", and select the first unordered

list, and its first list element.

```
let newElement = document.createElement("li");
newElement.innerHTML = "new text";

let parentElement = document.querySelector("ul");
let nextSibling = parentElement.querySelector("li");
```

To replace an element, we use `replaceChild`:

```
parentElement.replaceChild(newElement, nextSibling);
```

`nextSibling` is removed from the DOM. In its place is now `newElement`.

Removing an element

An element can be removed by calling `remove()` on it. Alternatively, one can call `removeChild()` on its parent. `removeChild()` has better browser support than `remove()`.

```
element.remove();
```

`element`, and all its childnodes, are removed from the DOM.

```
parentElement.removeChild(element);
```

`element`, and all its childnodes, are removed from the DOM.

In any case, one can insert this node in the DOM at a later point in time as long as there are still references to this node.

Append and Prepend methods

JavaScript now have the Append and Prepend methods which was present in jQuery

The main advantage of `append` and `prepend` is unlike `appendChild` and `insertBefore`, it can take any number of arguments either HTML element or plain text(which will be converted to text nodes).

To append say 1 div, 1 text node and 1 span

```
document.body.append(document.createElement('div'),"Hello
world",document.createElement('span'))
```

This will change the page to the following structure

```
<body>
      .....(other elements)
      <div></div>
      "Hello World"
      <span></span>
</body>
```

To prepend the same in body

Use

```
document.body.prepend(document.createElement('div'),"Hello
world",document.createElement('span'))
```

This will change the page to the following structure

```
<body>
      <div></div>
      "Hello World"
      <span></span>
      .....(other elements)
</body>
```

Note that browser supports are

Chrome 54+
Firefox 49+
Opera 39+

Read more at MDN

Append

Prepend

Read Manipulating Elements online: https://riptutorial.com/dom/topic/5200/manipulating-elements

Chapter 6: Retrieving Elements

Examples

By ID

```
document.getElementById('uniqueID')
```

will retrieve

```
<div id="uniqueID"></div>
```

As long as an element with the given ID exists, `document.getElementById` will return only that element. Otherwise, it will return `null`.

Note: IDs must be unique. Multiple elements cannot have the same ID.

By Class Name

```
document.getElementsByClassName('class-name')
```

will retrieve

```
<a class="class-name">Any</a>
<b class="class-name">tag</b>
<div class="class-name an-extra-class">with that class.</div>
```

If no existing elements contain the given class, an empty collection will be returned.

Example:

```
<p class="my-class">I will be matched</p>
<p class="my-class another-class">So will I</p>
<p class="something-else">I won't</p>
```

```
var myClassElements = document.getElementByClassName('my-class');
console.log(myClassElements.length); // 2
var nonExistentClassElements = document.getElementByClassName('nope');
console.log(nonExistentClassElements.length); // 0
```

By Tag Name

```
document.getElementsByTagName('b')
```

will retrieve

```
<b>All</b>
<b>of</b>
<b>the b elements.</b>
```

If no elements with the given tag name exist, an empty collection will be returned.

By CSS Selector

Consider following html code

```
<ul>
  <li id="one" class="main">Item 1</li>
  <li id="two" class="main">Item 2</li>
  <li id="three" class="main">Item 3</li>
  <li id="four">Item 4</li>
</ul>
```

Following dom tree will be constructed based on above html code

```
                      ul

                      |

     |          |          |          |

    li         li         li         li
     |          |          |          |
Item 1        Item 2    Item 3    Item 4
```

We can select elements from DOM tree with the help of CSS selectors. This is possible by means of two javascript methods viz `querySelector()` and `querySelectorAll()`.

querySelector() method returns the first element that matches the given css selector from the DOM.

```
document.querySelector('li.main')
```

returns the first `li` element who's class is `main`

```
document.querySelector('#two')
```

returns the element with id `two`

NOTE: If no element is found `null` is returned. If the selector string contains a CSS pseudo-element, the return will be `null`.

---

querySelectorAll() method returns all the elements that matches the given css selector from the DOM.

```
document.querySelectorAll('li.main')
```

returns a node list containing all the `li` elements who's class is `main`.

NOTE: If no element is found an empty node list is returned. If the selectors string contains a CSS pseudo-element, the returned elementList will be empty

Query Selectors

In modern browsers [1], it is possible to use CSS-like selector to query for elements in a document -- the same way as sizzle.js (used by jQuery).

querySelector

Returns the first `Element` in the document that matches the query. If there is no match, returns `null`.

```
// gets the element whose id="some-id"
var el1 = document.querySelector('#some-id');

// gets the first element in the document containing "class-name" in attribute class
var el2 = document.querySelector('.class-name');

// gets the first anchor element in the document
var el2 = document.querySelector('a');

// gets the first anchor element inside a section element in the document
var el2 = document.querySelector('section a');
```

querySelectorAll

Returns a `NodeList` containing all the elements in the document that match the query. If none match, returns an empty `NodeList`.

```
// gets all elements in the document containing "class-name" in attribute class
var el2 = document.querySelectorAll('.class-name');

// gets all anchor elements in the document
var el2 = document.querySelectorAll('a');

// gets all anchor elements inside any section element in the document
var el2 = document.querySelectorAll('section a');
```

Read Retrieving Elements online: https://riptutorial.com/dom/topic/2658/retrieving-elements

Chapter 7: Traversal

Examples

Tree walking

`TreeWalker` is a generator-like interface that makes recursively filtering nodes in a DOM tree easy and efficient.

The following code concatenates the value of all `Text` nodes in the page, and prints the result.

```
let parentNode = document.body;
let treeWalker = document.createTreeWalker(parentNode, NodeFilter.SHOW_TEXT);

let text = "";
while (treeWalker.nextNode())
    text += treeWalker.currentNode.nodeValue;

console.log(text); // all text in the page, concatenated
```

The `.createTreeWalker` function has a signature of

```
createTreeWalker(root, whatToShow, filter, entityReferenceExpansion)
```

Parameter	Details
root	The 'root' node who's subtree is to be traveresed
whatToShow	Optional, unsigned long designating what types of nodes to show. See NodeFilter for more information.
filter	Optional, An object with an acceptNode method to determine whether a node, after passing the whatToShow check should be considered
entityReferenceExpansion	Obsolete and optional, Is a Boolean flag indicating if when discarding an EntityReference its whole sub-tree must be discarded at the same time.

Iterating over nodes

The NodeIterator interface provides methods for iterating over nodes in a DOM tree.

Given a document like this one:

```
<html>
<body>
```

```
  <section class="main">
    <ul>
      <li>List Item</li>
      <li>List Item</li>
      <li>List Item</li>
      <li>List Item</li>
    </ul>
  </section>
</body>
</html>
```

One could imagine an iterator to get the `<li>` elements:

```
let root = document.body;
let whatToShow = NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT;
let filter = (node) => node.nodeName.toLowerCase() === 'li' ?
  NodeFilter.FILTER_ACCEPT :
  NodeFilter.FILTER_REJECT;
let iterator = document.createNodeIterator(root, whatToShow, filter);
var node;
while (node = iterator.nextNode()) {
  console.log(node);
}
```

Example adapted from the example provided by the Mozilla Contributors from the `document.createNodeIterator()` documentation on the Mozilla Developer Network, licensed under CC-by-SA 2.5.

This will log something like:

```
<li>List Item</li>
<li>List Item</li>
<li>List Item</li>
<li>List Item</li>
```

Note that this is similar to the TreeWalker iterface, but provides only `nextNode()` and `previousNode()` functionality.

Read Traversal online: https://riptutorial.com/dom/topic/5261/traversal

Chapter 8: Using CSS styles

Remarks

The interfaces detailed herein were introduced in DOM Level 2 Style, which came out at approximately the same time as DOM Level 2 Core and is thus considered "part of DOM version 2".

Examples

Reading and changing inline styles

Inline style

You can manipulate the inline CSS style of an HTML element by simply reading or editing its `style` property.

Assume the following element:

```
<div id="element_id" style="color:blue;width:200px;">abc</div>
```

With this JavaScript applied:

```
var element = document.getElementById('element_id');

// read the color
console.log(element.style.color); // blue

//Set the color to red
element.style.color = 'red';

//To remove a property, set it to null
element.style.width = null;
element.style.height = null;
```

However, if `width: 200px;` were set in an external CSS stylesheet, `element.style.width = null` would have no effect. In this case, to reset the style, you would have to set it to `initial`: `element.style.width = 'initial'`.

Reading and changing styles from a stylesheet

`element.style` only reads CSS properties set inline, as an element attribute. However, styles are often set in an external stylesheet. The actual style of an element can be accessed with `window.getComputedStyle(element)`. This function returns an object containing the actual computed value of all the styles.

Similar to the Reading and changing inline styles example, but now the styles are in a stylesheet:

```
<div id="element_id">abc</div>
<style type="text/css">
    #element_id {
        color:blue;
        width:200px;
    }
</style>
```

JavaScript:

```
var element = document.getElementById('element_id');

// read the color
console.log(element.style.color); // '' -- empty string
console.log(window.getComputedStyle(element).color); // rgb(0, 0, 255)

// read the width, reset it, then read it again
console.log(window.getComputedStyle(element).width); // 200px
element.style.width = 'initial';
console.log(window.getComputedStyle(element).width); // 885px (for example)
```

Read Using CSS styles online: https://riptutorial.com/dom/topic/5595/using-css-styles

Credits

S. No	Chapters	Contributors
1	Getting started with DOM	Blackus, Community, D.J., Dr. J. Testington, Henrique Barcelos, Jonas S, Leon Byford, maioman, Mike McCaughan, Mikhail, mnoronha, Mottie, Noushad PP, Roko C. Buljan, rvighne, Scimonster, Shog9, Sumurai8, Tushar
2	Events	Bamieh, Ian, Jeremy Banks, kamoroso94, Matas Vaitkevicius, Mike McCaughan, Mottie, Rap, The Alpha, Thriggle, zer00ne
3	Manipulating a list of CSS classes	Mike McCaughan, Mottie, Shog9
4	Manipulating Attributes	Mike McCaughan, Mottie
5	Manipulating Elements	Mike McCaughan, mnoronha, Sagar V, Sumurai8
6	Retrieving Elements	geeksal, Henrique Barcelos, maioman, Mike C, Mike McCaughan
7	Traversal	Jonas S, Mike McCaughan, rvighne
8	Using CSS styles	Blackus, Mike McCaughan, Scimonster