Metodologie per la Programmazione per il Web - MF0437
Programmazione client-side con Javascript

Docente

Giancarlo **Ruffo**[giancarlo.ruffo@uniupo.it]

Informazioni, materiale e risorse su:

moodle [https://www.dir.uniupo.it/course/view.php?id=16455]

Slide adattate da una versione precedente a cura del Prof. Alessio Bottrighi

Goal

- Understand how to load and use JS in the browser
- Learn what is the DOM
  - how to manipulate it
  - how to style it
- Study how to effectively handling events

Loading JS in the Browser

Mozilla Developer Network: The Script element
https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script

Loading JavaScript in the Browser

- JS must be loaded from an HTML document
- `<script>` tag
  - Inline

```
...
<script>
alert('Hello');
</script>
...
```

  - External

```
...
<script src="file.js"></
script>
...
```

https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script

Inline JavaScript

- **Immediately** executed when encountered
- Output is substituted to the tag content, and interpreted as HTML code
  - Avoid this behavior as much as possible
    - Difficult to maintain, slows down parsing and display, …

```
...
<script>
document.write('<p>Hello</
p>');
</script>
...
```

```
...
<p>Hello</p>
...
```

JavaScript External Resources

- JS code is loaded from one or more external resources (files)
- Loaded with `src=` attribute in `<script>` tag
- The JS file is loaded, and immediately executed
  - Then, HTML processing continues

```
<script src="file.js"></script>
<!-- type="text/javascript" is the
default: not needed -->
```

Where to Insert the <script> Tag?

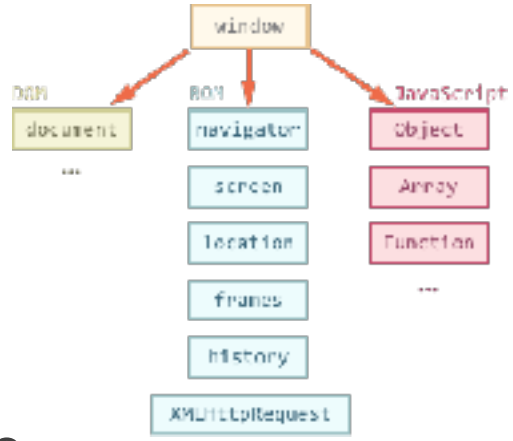
- In the `<head>` section
  - "clean" / "textbook" solution
  - Very **inefficient**: HTML processing is stopped until the script is loaded and executed
  - Quite **inconvenient**: the script executes when the document's DOM doesn't exist, yet
- Just before the end of the document
  - Much more efficient
- But … see later "Performance tips"

```
<!DOCTYPE html>
<html>
    <head>
        <title>Loading a script</title>
        <script src="script.js"></script>
    </head>
    <body>
        ...
    </body>
</html>
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Loading a script</title>
    </head>
    <body>
        ...
        <script src="script.js"></script>
    </body>
</html>
```

Where Does the Code Go?

- Loaded and run in the browser ***sandbox***
- Attached to a *global context*: the `window` object
- May access only a limited set of APIs
  - JS Standard Library
  - Browser objects (BOM)
  - Document objects (DOM)



- Multiple `<script>`s are independent
  - They all access the same global scope
  - To have structured collaboration, ***modules*** are needed

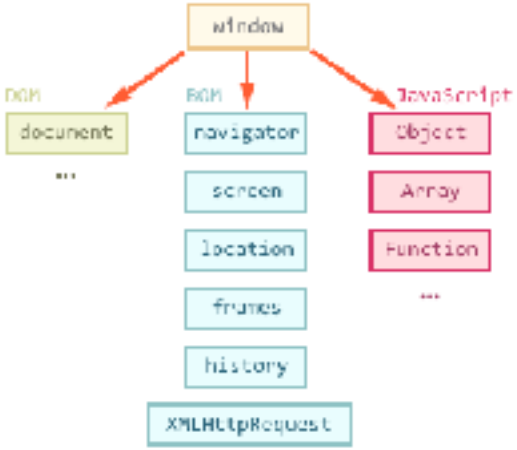
Remind: Events and Event Loop

- Most phases of processing and interaction with a web document will generate Asynchronous ***Events*** (100's of different types)
- Generated events may be handled by:
  - Pre-defined behaviors (by the browser)
  - User-defined ***event handlers*** (in your JS)
  - Or just ignored, if no event handler is defined
- But JavaScript is single-threaded
  - Event handling is ***synchronous*** and is based on an ***event loop***
  - Event handlers are queued on a ***Message Queue***
  - The Message Queue is polled when the main thread is idle

Browser Object Model

Browser's Main Objects

- `window` represents the window that contains the DOM document
  - allows to interact with the browser via the BOM: browser object model (not standardized)
  - global object, contains all JS global variables
    - can be omitted when writing JS code in the page
- `document`
  - represents the DOM tree loaded in a window
  - accessible via a `window` property: `window.document`



source: https://medium.com/@fknussel/dom-bom-revisited-cf6124e2a816

The *Global* Scope

- `window` represents the global scope of the JS program
- Attributes may be added to `window`
  - Explicitly: `window.myprogram = "nice";`
  - Implicitly: `var myprogram = "nice";`
  - Beware name clashes with other scripts or predefined properties
- `window` attributes are automatically visible
  - `window.document` and `document` are equivalent

Browser Object Model

- `window` properties
  - `console`: browser debug console (visible via developer tools)
  - `document`: the document object
  - `history`: allows access to History API (history of URLs)
  - `location`: allows access to Location API (current URL, protocol, etc.). Read/write property, i.e. can be set to load a new page
  - `localStorage` and `sessionStorage`: allows access to the two objects via the Web Storage API, to store (small) info locally in the browser

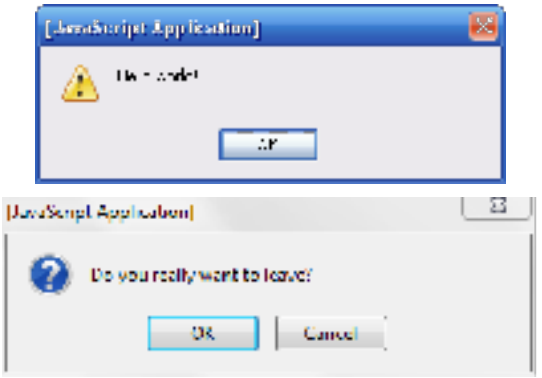
source: https://developer.mozilla.org/en-US/docs/Web/API/Window

Frequently Seen Properties and Methods

Object	Property anc Methods
window	Other g oba objects, open(), c ose(), moveTo(), resizeTo()
screen	width, height, colorDepth, pixelDepth, ...
location	hostname, pathname, port, protocol, assign(), ...
history	back(), forward()
navigator	userAgent, platform, systemLanguage, ...
document	body, forms, write(), close(), getElementById(), ...
Popup Boxes	alert(), confirm(), prompt()
Timing	setInterval(func,time,p1,...), setTimeout(func,time)

Window Object: Main Methods

- Methods
  - `alert()`, `prompt()`, `confirm()`: handle browser-native dialog boxes *Never use them - just for debug*



  - `setInterval()`, `clearInterval()`, `setTimeout()`, `setImmediate()`: allows to execute code via the event loop of the browser
  - `addEventListener()`, `removeEventListener()`: allows to execute code when specific events happen to the document

source: https://developer.mozilla.org/en-US/docs/Web/API/Window

Window Object: Main Methods

- `open()`: allows to open a **new** browser window
- `moveTo()`, `resizeTo()`, `minimize()`, `focus()`: allows to manipulate the browser window
- ...

source: https://developer.mozilla.org/en-US/docs/Web/API/Window

Storing Data

Cookies

- String/value pairs, Semicolon separated
- Cookies are transferred on to every request

```
document.cookie = "name=Jane Doe; nr=1234567;
expires="+date.toGMTString()
```

Web Storage (Local and Session Storage)

- Store data as key/value pairs on user side
- Browser defines storage quota

Local Storage (`window.localStorage`)

- Store data in users browser
- Comparison to Cookies: more secure, larger data capacity, not transferred
- No expiration date

Session Storage (`window.sessionStorage`)

- Store data in session
- Data is destroyed when tab/browser is closed

```
let storage = permanent ? window.localStorage :
                          window.sessionStorage;
if(!storage["name"]) {
   storage["name"] = "A simple storage"
}
alert("Your name is " + storage["name"]);
```

Document Object Model

DOM History

- DOM Level "0": legacy DOM
  - Partly specified in HTML4. Mainly to access interactive elements (forms, links, …)
- DOM Level 1 (1998): W3C recommendation
  - DOM Core: a model for easy manipulation of an XML-based document
  - Extended with HTML-specific objects and methods that can change portions of the doc
  - Note: DOM is not JavaScript-specific. However, in the browser context, has been implemented using ECMAScript

DOM History

- DOM Level 2 (2000)
  - Introduces new interfaces to manage events, styles (CSS support), possibility to more easily access elements (e.g., getElementById)
- DOM Level 3 (2004)
  - Includes full support for XML 1.0, e.g., Xpath to access elements, and keyboard event handling
- DOM Level 4 (2015)
  - Snapshot of the WHATWG living standard
  - Several significant non-backward compatible changes (e.g., the attributes are not nodes)

DOM Living Standard

- Standardized by WHATWG in the DOM Living Standard Specification
- https://dom.spec.whatwg.org

DOM

Living Standard — Last Updated 14 March 2020

Participate:

GitHub whatwg/dom (new issue, open issues)

IRC: #whatwg on Freenode

Commits:

GitHub whatwg/dom/commits

Snapshot as of this commit

@thedomstandard

Tests:

web-platform-tests dom/ (ongoing work)

Translations (non-normative):

日本語

Abstract

DOM defines a platform-neutral model for events, aborting activities, and node trees.

Table of Contents

Goals

1 Infrastructure

1.1 Trees

1.2 Ordered sets

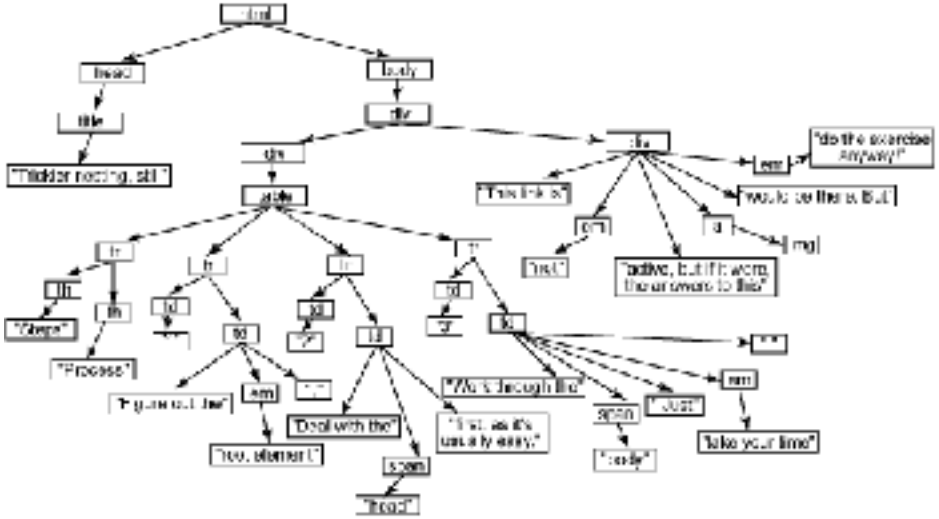
1.3 Selectors

1.4 Namespaces

2 Events

DOM

- Browser's internal representation of a web page
- Obtained through parsing HTML
  - Example of parsed HTML tree structure
- Browsers expose an API that you can use to interact with the DOM



https://flaviocopes.com/dom/

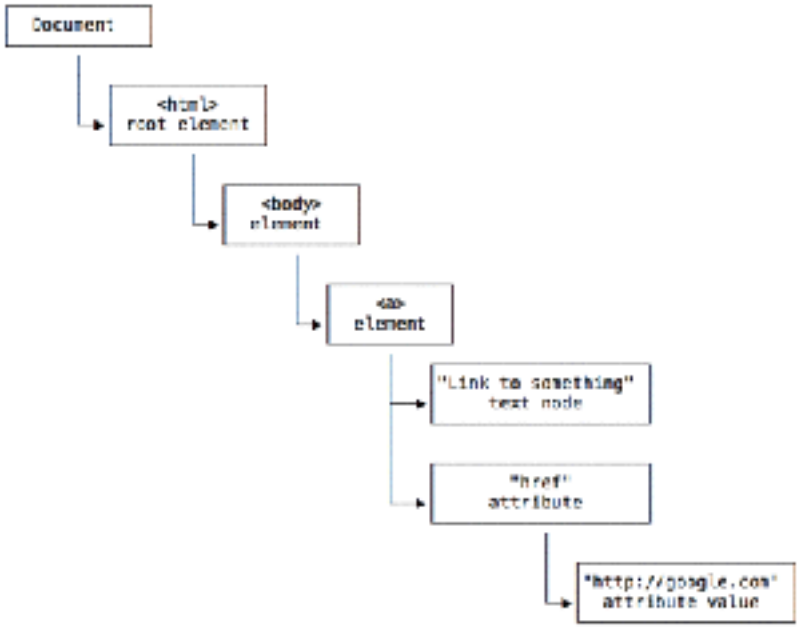
Interaction with the DOM

- Via JavaScript it is possible to
  - Access the page metadata and headers
  - Inspect the page structure
  - Edit any node in the page
  - Change any node attribute
  - Create/delete nodes in the page
  - Edit the CSS styling and classes
  - Attach or remove ***event listeners***

source: https://flaviocopes.com/dom/

Types of Nodes

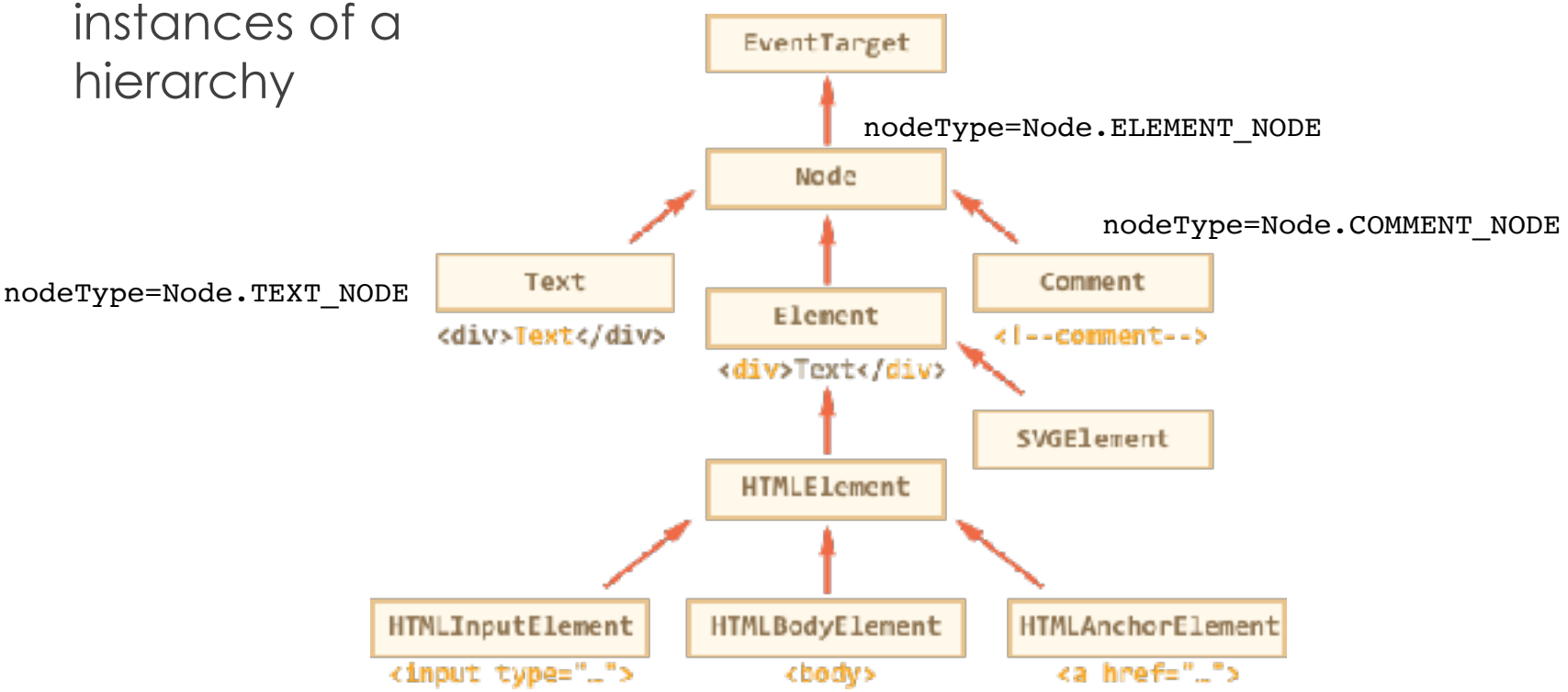
- **Document:** the document Node, the root of the tree
- **Element:** an HTML tag
- **Attr:** an attribute of a tag
- **Text:** the text content of an Element or Attr Node
- **Comment:** an HTML comment
- **DocumentType:** the Doctype declaration



source: https://flaviocopes.com/dom/

DOM Classes Hierarchy

- Objects in DOM are instances of a hierarchy



Node Lists

- The DOM API may manipulate sets/lists of nodes
- The NodeList type is an array-like sequence of Nodes
- May be accessed as a JS array
  - .length property
  - .item(i) , equivalent to list[i]
  - .entries(), .keys(), .values() iterators
  - .forEach() functional iteration primitive
  - for…of for classical iteration

DOM Manipulation

Finding DOM Elements

- `document.getElementById(value)`
  - Node with the attribute id=value
- `document.getElementsByTagName(value)`
  - NodeList of all elements with the specified tag name (e.g., 'div')
- `document.getElementsByClassName(value)`
  - NodeList of all elements with attribute class=value (e.g., 'col-8')
- `document.querySelector(css)`
  - First Node element that matches the CSS selector syntax
- `document.querySelectorAll(css)`
  - NodeList of all elements that match the CSS selector syntax

source: https://flaviocopes.com/dom/

Note

- Node-finding methods also work on any Element node
- In that case, they only search trough ***descendant*** elements
  - May be used to refine the search

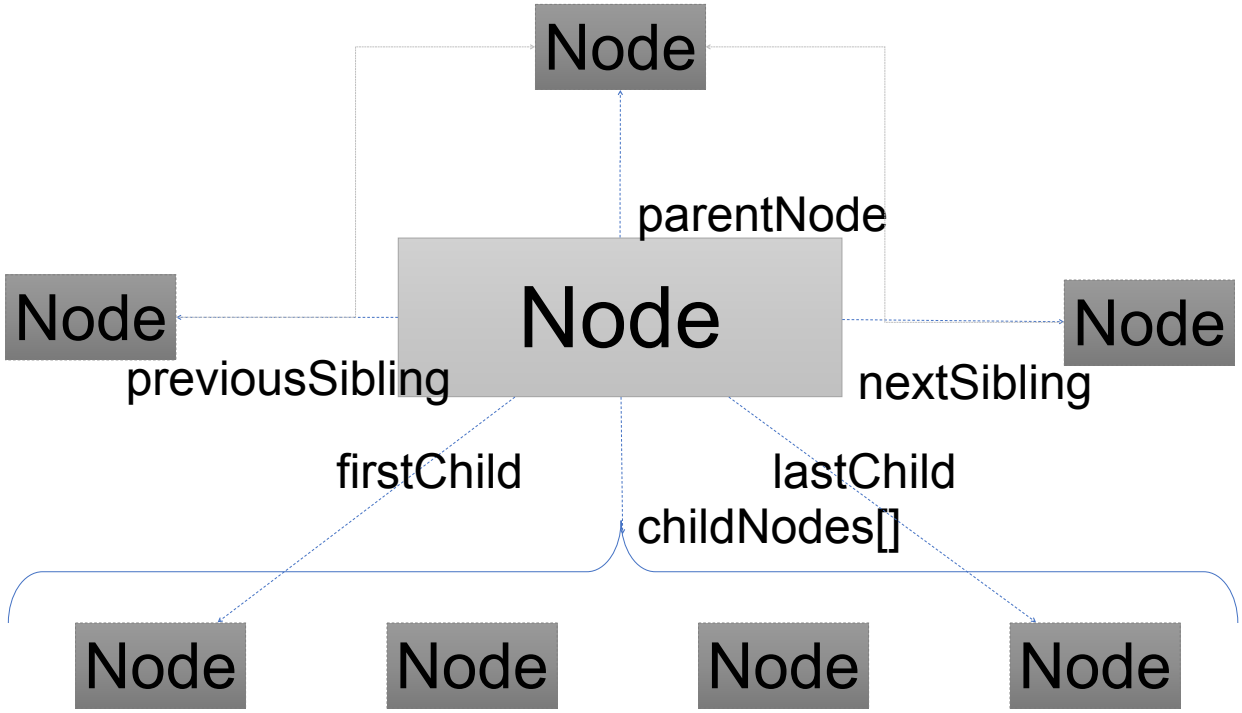
Accessing DOM Elements

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<div id="foo"></div>
<div class="bold"></div>
<div class="bold color"></div>
<script>
 document.getElementById('foo');
 document.querySelector('#foo');
 document.querySelectorAll('.bold');
 document.querySelectorAll('.color');
 document.querySelectorAll('.bold, .color');
</script>
</body>
</html>
```

```
<div id="foo"></div>
<div id="foo"></div>
▶ NodeList(2) [div.bold, div.bold.color]
▶ NodeList [div.bold.color]
▶ NodeList(2) [div.bold, div.bold.color]
>
```

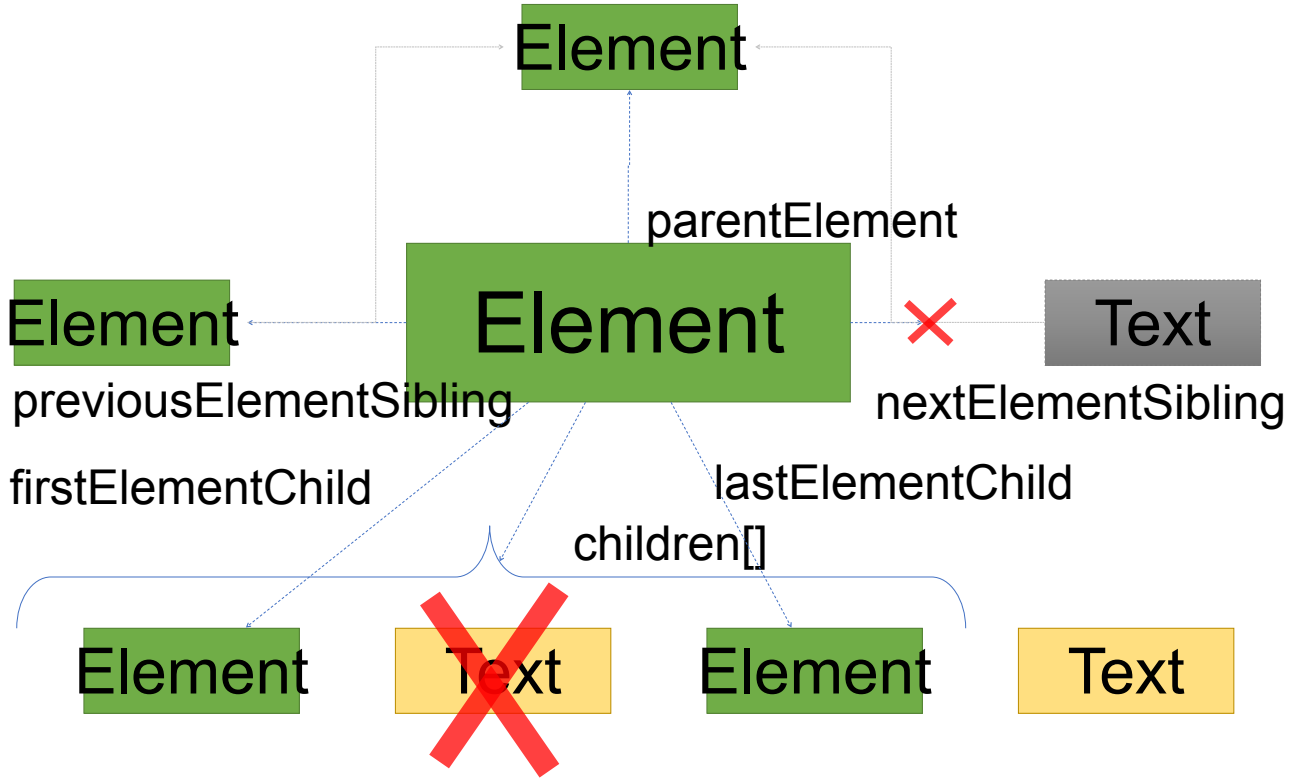
Navigating the Tree

- Properties to navigate the tree



Navigating the Tree

- "Elements" do not include text



Tag Attributes Exposed as Properties

- Attributes of the HTML elements become properties of the DOM objects
- Example
  - `<body id="page">`
  - DOM object: `document.body.id="page"`
  - `<input id="input" type="checkbox" checked />`
  - DOM object: `input.checked  // boolean`
- Can read attributes, but to modify content of visualized objects, use `setAttribute()`

Handling Tag Attributes

- `elem.hasAttribute(name)`
  - check the existence of the attribute
- `elem.getAttribute(name)`
  - check the value
- `elem.setAttribute(name, value)`
  - set the value of the attribute
- `elem.removeAttribute(name)`
  - delete the attribute
- `elem.attributes`
  - collection of all attributes
- `elem.matches(css)`
  - Check whether the element matches the css selector

Creating Elements

- Use document methods:
  - `document.createElement(tag)` to create an element with a tag
  - `document.createTextNode(text)` to create a text node with the text
- Example: div with class and content

```
let div = document.createElement('div');
div.className = "alert alert-success";
div.innerText = "Hi there! You've read an important message.";
```

```
<div class="alert alert-success">
Hi there! You've read an important message.
</div>
```

Inserting Elements in the DOM Tree

- If not inserted, they will not appear

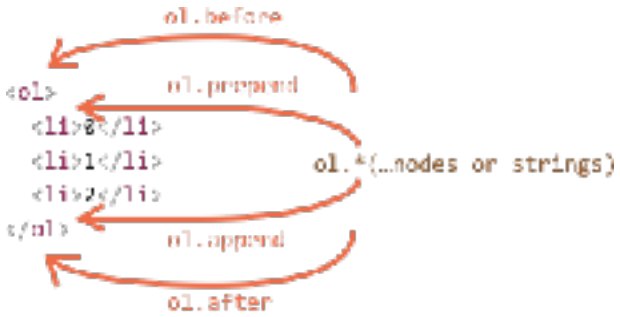
  `document.body.appendChild(div)`

```
...
<body>
 <div class="alert alert-success">
 <strong>Hi there!</strong> You've read an important message.
 </div>
<body>
```

Inserting Children

- `parentElem.appendChild(node)`
- `parentElem.insertBefore(node, nextSibling)`
- `parentElem.replaceChild(node, oldChild)`

- `node.append(…nodes or strings)`
- `node.prepend(…nodes or strings)`
- `node.before(…nodes or strings)`
- `node.after(…nodes or strings)`
- `node.replaceWith(…nodes or strings)`



Handling Tag Content

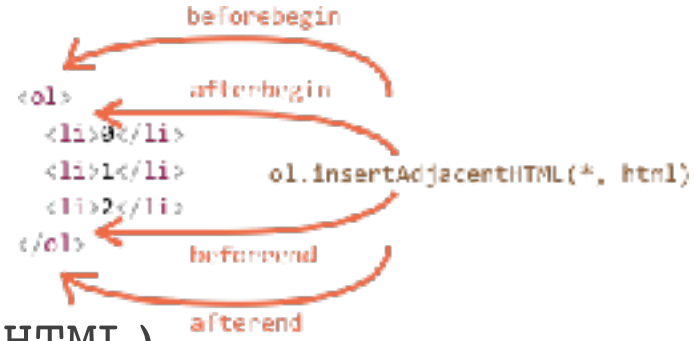
- `.innerHTML` to get/set element content in textual form
- The browser will parse the content and convert it into DOM Nodes and Attributes

```
<div class="alert alert-success">
<strong>Hi there!</strong> You've read an important message.
</div>
```

```
div.innerHTML  // "<strong>Hi there!</strong> You've read
an important message."
```

Inserting New Content

- `elem.innerHTML = "html fragment"`



- `elem.insertAdjacentHTML(where, HTML)`
  - where = beforebegin | afterbegin | beforeend | afterend
  - HTML = nodes to insert

- `elem.insertAdjacentText(where, text)`
- `elem.insertAdjacentElement(where, elem)`

Cloning Nodes

- `elem.cloneNode(true)`
  - Recursive (deep) copy of the element, including its attributes, sub-elements, …
- `elem.cloneNode(false)`
  - Shallow copy (will not contain the children)
- Useful to "replicate" some parts of the document

DOM Styling

Styling Elements

- Via values of **class** attribute defined in CSS
- Change class using the property **`className`**
  - Replaces the whole string of classes
  - *Note*: className, not class (JS reserved word)
- To add/remove a single class use `classList`
  - `elem.classList.add("col-3")` add a class
  - `elem.classList.remove("col-3")` remove a class
  - `elem.classList.toggle("col-3")` if the class exists, it removes it, otherwise it adds it
  - `elem.classList.contains("col-3")` returns true/false checking if the element contains the class

Styling Elements

- `elem.style` contains all CSS properties
  - Example: hide element

    `elem.style.display="none"`

    (equivalent to CSS declaration `display:none`)

- `getComputedStyle(element[,pseudo])`
  - `element`: selects the element of which we want to read the value
  - `pseudo`: a pseudo element, if necessary
- For properties that use more words the camelCase is used (`backgroundColor, zIndex...` instead of `background-color ...`)

Event Handling

Mozilla Developer Network: Event Reference
https://developer.mozilla.org/en-US/docs/Web/Events

Event Listeners

- JavaScript in the browser uses an ***event-driven*** programming model
  - Everything is triggered by the firing of an event
- Events are determined by
  - The Element generating the event (event ~~source~~ target)
  - The type of generated event
- JavaScript supports three ways of defining event handlers
  - Inline event handlers
  - DOM on-event handlers
  - Using `addEventListener()` (modern way)

source: https://flaviocopes.com/javascript-events/

Inline Event Handlers

- Rarely used nowadays
- Inline JavaScript code as value of a special attribute

```
<a href="site.com" onclick="doSomething();">A link</a>
```

source: https://flaviocopes.com/javascript-events/

DOM On-Event Handlers

- Assign a callback to a special property
- Only one callback can be assigned

```
window.onload = () => {
  //window loaded
}
```

source: https://flaviocopes.com/javas

addEventListener

- Can add as many listeners as desired, even to the same node
- Callback receives as first parameter an **`Event`** object

```
window.addEventListener('load', () => {
  //window loaded
})
```

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // mouse button pressed
  console.log(event.button) //0=left, 2=right
})
```

source: https://flaviocopes.com/javascript-events/

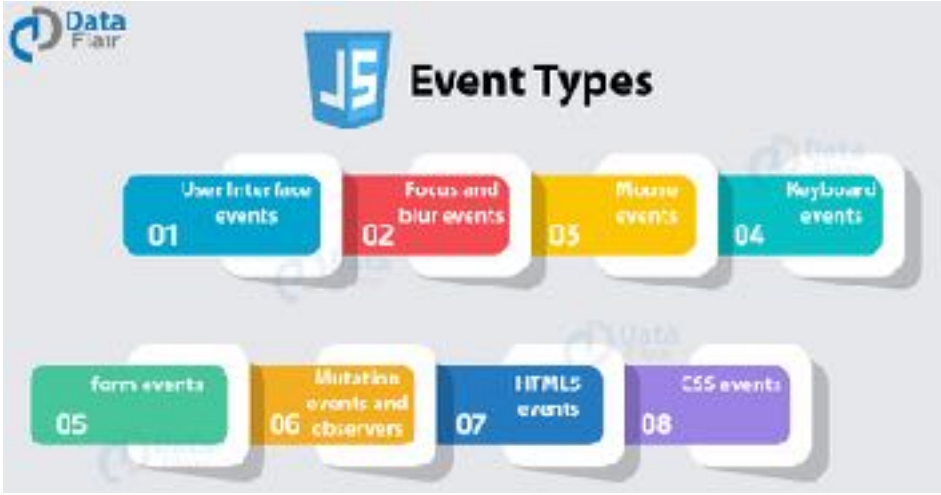
Event Object

- Main properties:
  - `target`, the DOM element that originated the event
  - `type`, the type of event
  - `stopPropagation()` called to stop propagating the event in the DOM

source: https://developer.mozilla.org/en-US/docs/Web/API/Event/type

Event Categories

- User Interface events (load, resize, scroll, etc.)
- Focus/blur events
- Mouse events (click, dblclick, mouseover, drag, etc.)
- Keyboard events (keyup, etc.)
- Form events (submit, change, input)
- Mutation events (DOMContentLoaded, etc.)
- HTML5 events (invalid, loadeddata, etc.)
- CSS events (animations etc.)

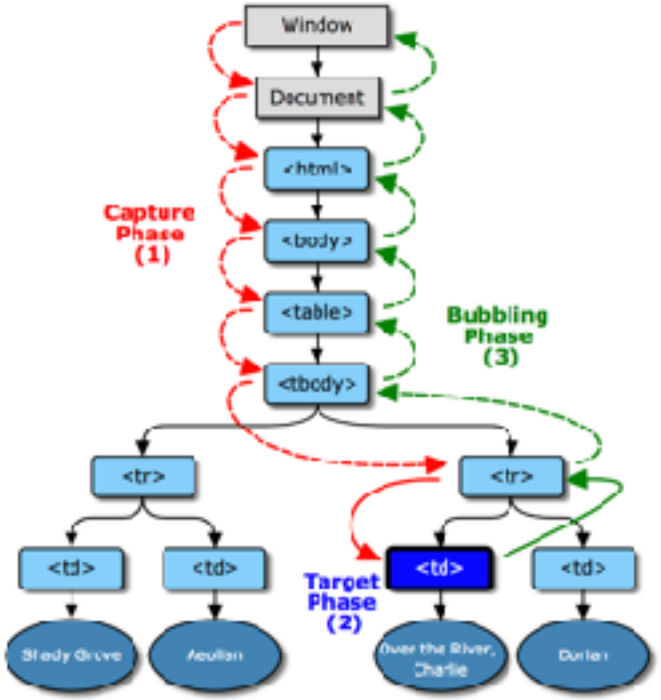


Category	Type	Attribute	Description	Bubbles	Cancelable
Mouse	click	onclick	Fires when the primary device button is clicked over an element. It is defined as a mousedown and mouseup over the same screen location. The sequence of these events is: • mousedown • mouseup • click	Yes	Yes
	dblclick	ondblclick	Fires when the primary device button is double-clicked over an element	Yes	Yes
	mousedown	onmousedown	Fires when the primary device button is pressed over an element	Yes	Yes
	mouseup	onmouseup	Fires when the primary device button is released over an element	Yes	Yes
	mouseover	onmouseover	Fires when the pointing device is moved onto an element	Yes	Yes
	mousemove[8]	onmousemove	Fires when the pointing device is moved while it is over an element	Yes	No
	mouseout	onmouseout	Fires when the pointing device is moved away from an element	Yes	Yes
	mouseenter	onmouseenter	Fires on an element when a drag is initiated	Yes	No
	drag	ondrag	This event is fired at the source of the drag, that is, the element which originally was fired, during the drag operation	Yes	Yes
	mouseleave	onmouseleave	Fires when the mouse is first moved over an element while a drag is occurring	Yes	No
	dragstart	ondragstart	This event is fired when the user starts dragging an element or text selection. A drag is occurring.	Yes	Yes
	dragover	ondragover	This event is fired as the user is dragging the mouse over a valid drop target. A drag is occurring.	Yes	No
	drop	ondrop	This event is fired when an element or text selection is dropped on a valid drop target.	Yes	No
	dragend	ondragend	This event is fired when the user has released the mouse button to end the drag operation. The source of the drag will receive a dragend event when the drag operation is complete, whether it was successful or not.	Yes	No
Keyboard	keydown	onkeydown	Fires before keypress, when a key on the keyboard is pressed.	Yes	Yes
	keypress	onkeypress	Fires after keydown, when a key on the keyboard is pressed.	Yes	Yes
	keyup	onkeyup	Fires when a key on the keyboard is released	Yes	Yes
HTML / Object	load	onload	Fires when the user agent finishes loading all content within a document, including windows, frames, objects and images For elements, it fires when the target element and all of its content has finished loading	No	No
	unload	onunload	Fires when the user agent removes all content from a window or frame For elements, it fires when the target element or any of its content has been removed	No	No
	abort	onabort	Fires when an object/image is stopped from loading before completely loaded	Yes	No
	error	onerror	Fires when an object/image/frame cannot be loaded properly	Yes	No
	select	onselect	Fires when a document view is selected	Yes	No
	resize	onresize	Fires when an element or document view is resized	No, except Bubbles when event on document must bubble to the window[9]	No
	scroll	onscroll	Fires when a user selects some text in a textfield, including input and textarea	Yes	No
HTML Form	change	onchange	Fires when a control loses the input focus and its value has been modified since gaining focus	Yes	No
	submit	onsubmit	Fires when a form is submitted	Yes	Yes
	reset	onreset	Fires when a form is reset	Yes	No
	focus	onfocus	Fires when an element receives focus either via the pointing device or by tab navigation	No	No
	blur	onblur	Fires when an element loses focus either via the pointing device or by tab navigation	No	No
User Interface	DOMActivate	onactivate	Similar to HTML onclick event, but can be applied to any focusable element	Yes	No
	DOMFocusIn	onfocusin	Similar to HTML onfocus event, but can be applied to any focusable element	Yes	No
	DOMFocusOut	onfocusout	Similar to HTML onblur event, but can be applied to any focusable element	Yes	No
Mutation	DOMSubtreeModified	onsubtreemodified	Fires when the subtree is modified	Yes	No
	DOMNodeInserted	onnodeinserted	Fires when a node has been added as a child of another node	Yes	No
	DOMNodeRemoved	onnoderemoved	Fires when a node has been removed from its parent node	Yes	No
	DOMNodeRemovedFromDocument	onnoderemovedfromdocument	Fires when a node is being removed from a document	No	No
	DOMNodeInsertedIntoDocument	onnodeinsertedintodocument	Fires when a node is being inserted into a document	No	No
	DOMAttrModified	onattrmodified	Fires when an attribute has been modified	Yes	No
	DOMCharacterDataModified	oncharacterdatamodified	Fires when the character data has been modified	Yes	No
Progress	loadstart	onloadstart	Progress has begun	No	No
	progress	onprogress	In progress. After loadstart has been dispatched.	No	No
	error	onerror	Progression failed. After the last progress has been dispatched, or after loadstart has been dispatched if progress has not been dispatched.	No	No
	abort	onabort	Progression is terminated. After the last progress has been dispatched, or after loadstart has been dispatched if progress has not been dispatched.	No	No
	load	onload	Progression is successful. After the last progress has been dispatched, or after loadstart has been dispatched if progress has not been dispatched.	No	No
	loadend	onloadend	Progress has stopped. After one of error, abort, or load has been dispatched.	No	No

https://en.wikipedia.org/wiki/DOM_events

Event Handling on the DOM Tree

- Something occurs (e.g., a mouse click, a button press)
- Capture phase
  - The event is passed to all DOM elements on the path from the Document to the parent of the target element
  - No event handlers are fired
    - Except if registered with useCapture=true
- Target phase
  - The event reaches the target
  - Event handlers are triggered
- Bubbling phase
  - Trace back the path towards the document root
  - Event handlers are triggered on any encountered node
  - Allows us to handle an event on any element by its parent elements
  - event.stopPropagation() interrupts the bubbling phase



source: https://medium.com/prod-io/javascript-understanding-dom-event-life-cycle-49e1cf62b2ea

Event Bubbling

- Events propagate along the DOM tree
- Bubbling: the event propagates from the item that was affected (target) up to all its parent tree, starting from the nearest one
  - Every time it fires the handler of the element, if present
- Useful to create default handlers (on the outer elements)

```
<div id="container">              // 2nd
  <button>Click me</button>       // 1st
</div>
```

Preventing Default Behavior

- Many events cause a default behavior
  - Click on link: go to URL
  - Click on submit button: form is sent
- Can be prevented by

```
event.preventDefault()
```

Stopping Event Propagation

- Can be done with `event.stopPropagation()`
  - Typically in the event handler

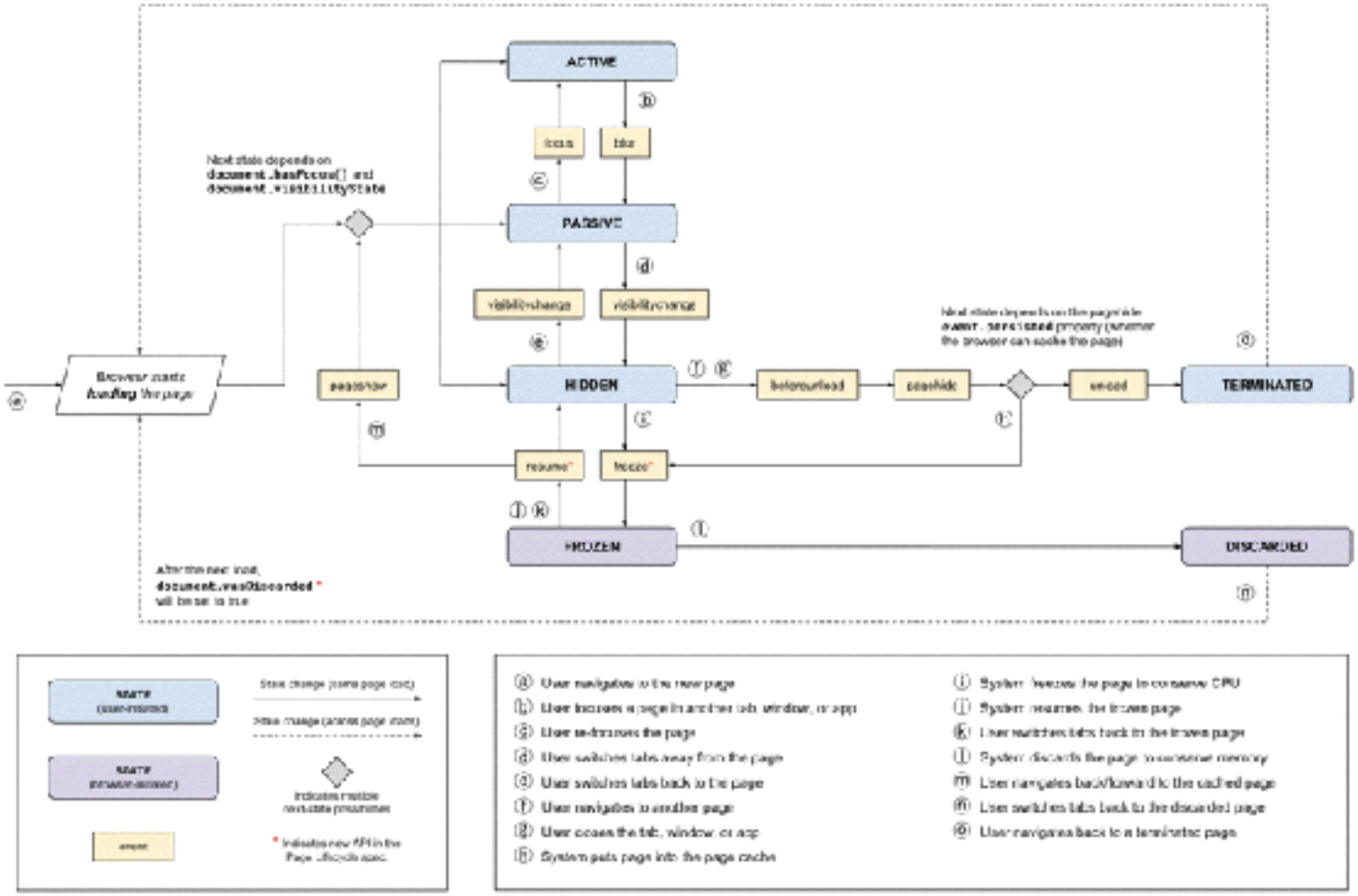
```
const link = document.getElementById('my-
link')
link.addEventListener('mousedown', event => {
  // process the event
  // ...

  event.stopPropagation()
})
```

HTML Page Lifecycle: Events

- `DOMContentLoaded` (defined on `document`)
  - The browser loaded all HTML and the DOM tree is ready
  - External resources are not loaded, yet
- `load` (defined on `window`)
  - The browser finished loading all external resources
- `beforeunload`/`unload`
  - The user is about to leave the page / has just left the page
  - Not recommended (non totally reliable)

```
document.addEventListener("DOMContentLoaded", ready);
```

More Lifecycle Events



https://developers.google.com/web/updates/2018/07/page-lifecycle-api

Throttling

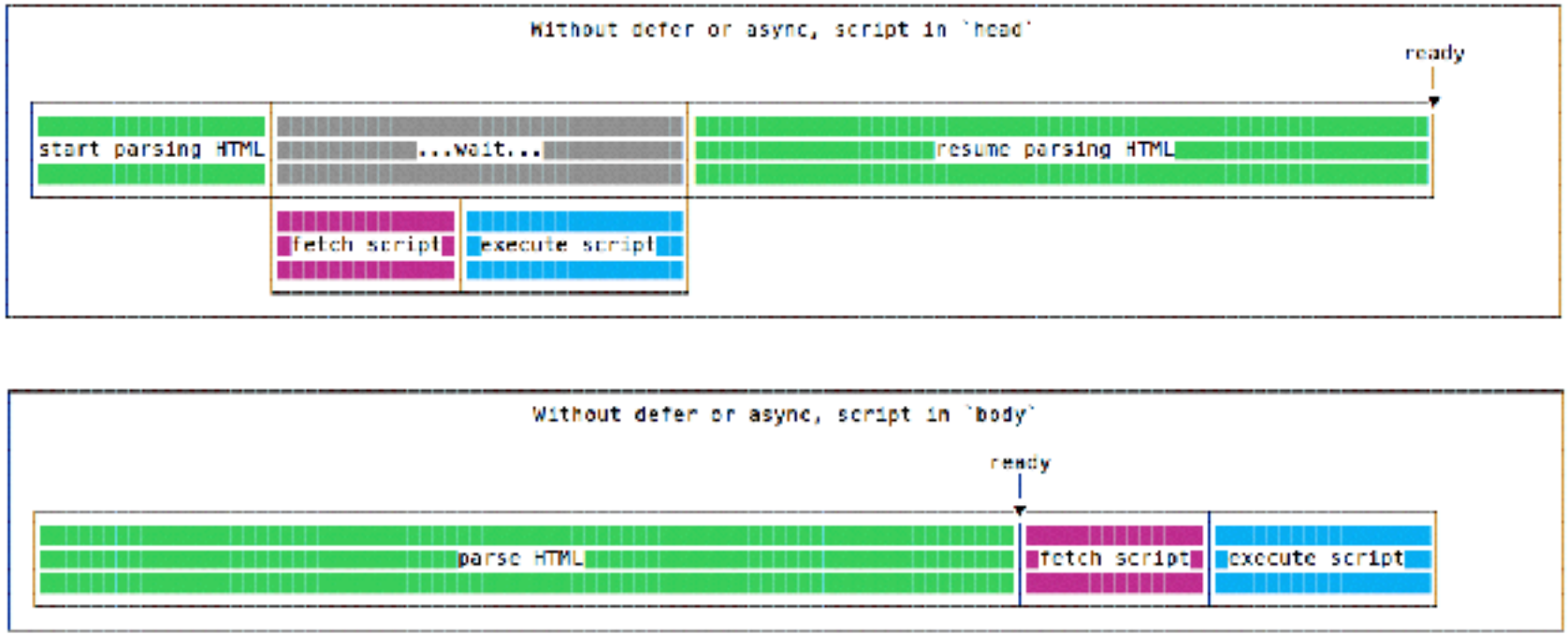
- Some events fire continuously (mousemove, scroll, etc.) providing coordinates, so that user behavior can be tracked
- Complex operations in the event handler result in sluggish user experience
- Use external libraries or set timers to process them only periodically

```
let cached = null ;
window.addEventListener('scroll', event => {
  if (!cached) {
    setTimeout(() => {
      // process event -- you can access the original event at `cached`
      cached = null ;
    }, 100) }
  cached = event ;
}) ;
```

source: https://flaviocopes.com/javascript-events/

Performance Tips

Performance Comparison in Loading JS

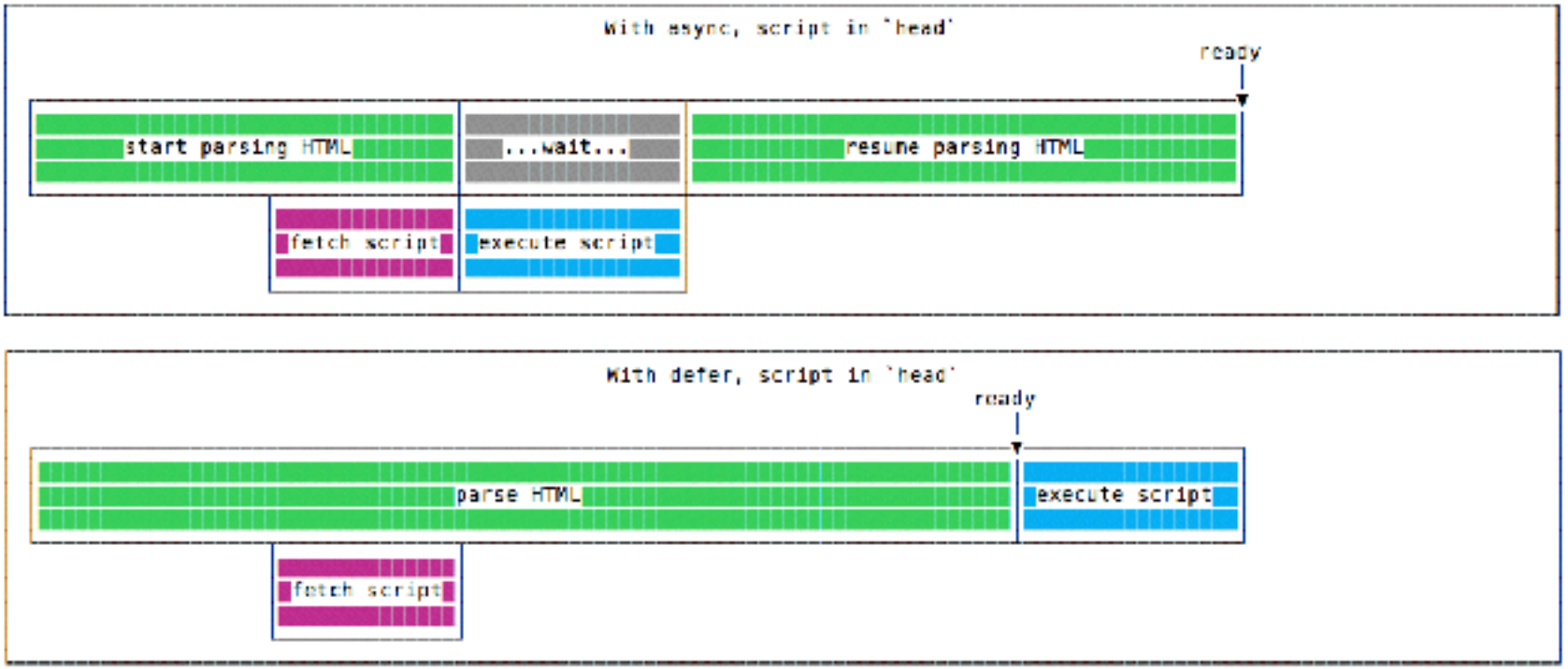


source: https://flaviocopes.com/javascript-async-defer

New Loading Attributes

- `<script async src="script.js"></script>`
  - Script will be fetched in parallel to parsing and evaluated as soon as it is available
  - Not immediately executed, not blocking
- `<script defer src="script.js"></script>` (*preferred*)
  - Indicate to a browser that the script is meant to be executed after the document has been parsed, but before firing `DOMContentLoaded` (that will wait until the script is finished)
  - Guaranteed to execute in the order they are loaded
- Both should be placed in the `<head>` of the document

Defer vs. Async



source: https://flaviocopes.com/javascript-async-defer

Sources

- Useful and source links are reported in the slides
- These slides are adapted from the "Structure of a Web Page: CSS" slides of the ***Web Applications I*** course at Politecnico di Torino
  - http://bit.ly/polito-wa1
- Web Engineering SS20 - TU Wien, prof. Jürgen Cito
  - https://web-engineering-tuwien.github.io/
- Async and defer
  - https://hacks.mozilla.org/2017/09/building-the-dom-faster-speculative-parsing-async-defer-and-preload/