Lecture 9

- Javascript in Document Object Model (DOM) manipulations:
  - Selecting elements
    - individual elements
    - collections of elements
      - live vs. static
  - Traversing elements
  - Adding/removing elements
  - Selecting and updating attributes
- Events
  - Event object
  - Event handler
  - Passing parameters
  - Event propagation

Human Potential Unlimited.

How to access DOM tree elements?

```
const x
    x:
```

```
  . . .
<li id="idx" class="hot">
    content to the
    li element is something
</li>
  . . .
```

- Both id and class attributes functions as potential handles

Human Potential Unlimited.

Access individual elements

- By their unique id value

```
const x = document.getElementById("idx");
```

- Using CSS selectors

```
const x = document.querySelector("li.hot");
x.className = "cool";
```

returns a reference to the **first** element with the class 'hot'

index

Accessing several elements with `querySelectorAll()`

- The following provides a NodeList of *all* the elements that satisfy the css selector

```
const x = document.querySelectorAll("li.hot");
```

- NodeList is similar to an array, e.g., has a **'length'** property
- NodeList is accessed by indexing (similar to an array)

```
x[0].className = "newValue"
```

- Can use a **'for'** statement to loop through each item in the collection

index

getElementsByClassName
getElementsByTagName

- A reference to a collection of all the elements that belong to a particular class

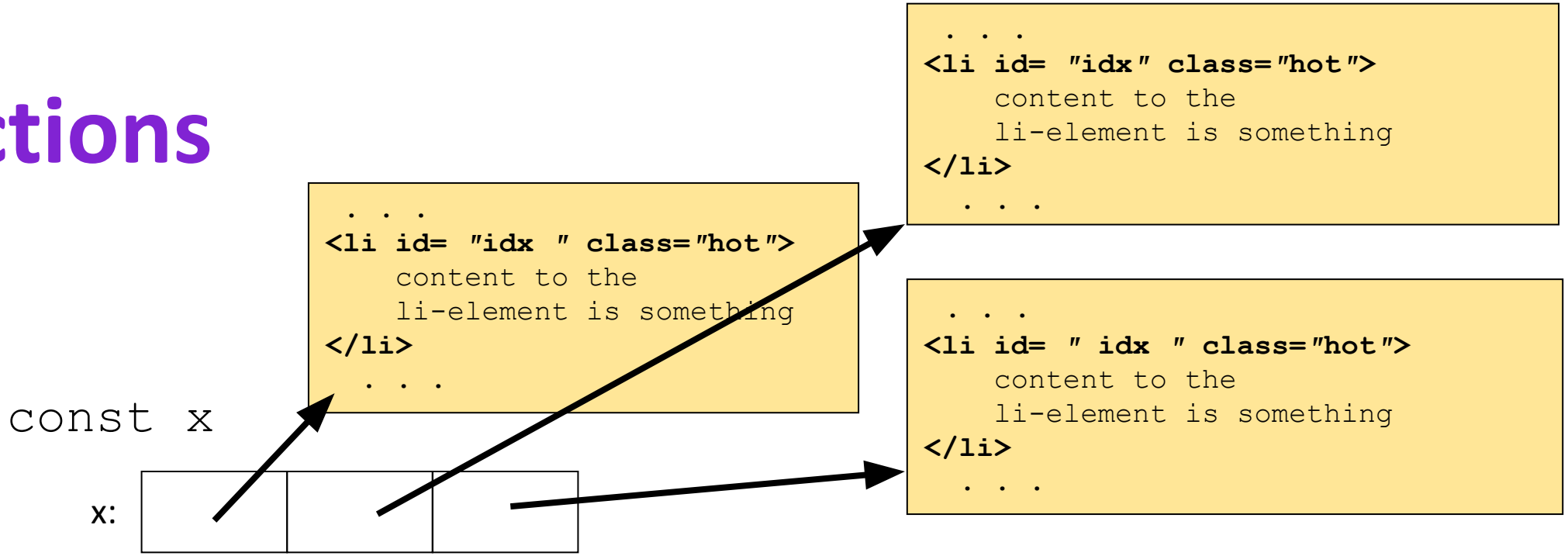
```
const x =
document.getElementsByClassName("hot");
```

- A reference to a collection of all the <li> elements

```
const x = document.getElementsByTagName("li");
```

index

Collections



- A collection of references to the document objects that satisfy a particular condition (e.g., all with the same class name)

```
const x = document.getElementsByClassName("hot");
//x.length==3
x[1].hidden = true;     // access the second item
```

Human Potential Unlimited.

Live and static collections

- HTMLCollections returned by **getElementsByXXX()** are **'live'**.. they are updated when the document is changed

- NodeLists returned by **querySelectorAll()** are **'static'**.. they reflect the state of the document when the query was made.. and aren't updated afterwards

(so what? is this a problem?)

Human Potential Unlimited.

Traversing between elements

- Each element on a page is represented as a node in DOM tree
  - nodes have methods and properties
    - e.g., properties for traversing:

```
nextSibling, previousSibling, parentNode, firstChild, lastChild
```

index

Human Potential Unlimited.

previousSibling, nextSibling

- Properties of a node: previousSibling, nextSibling

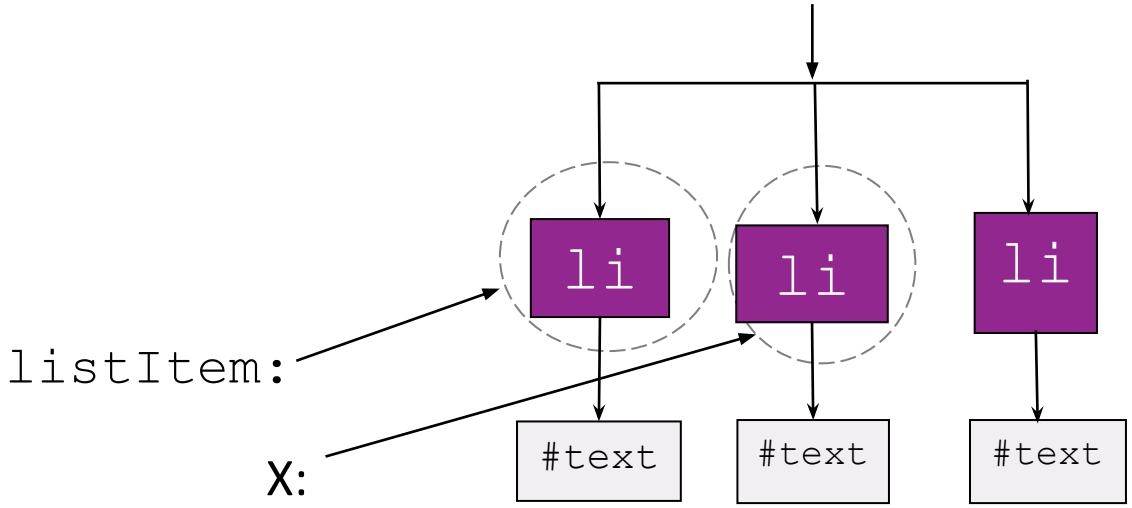
```
const listItem = document.getElementById("idx");
const x = listItem.nextSibling;
```

- these are properties, not methods, thus, no brackets
- if there is no 'nextSibling' then x is **null**
- order of siblings is determined by their order in the DOM tree

index

nextSibling

```
const listItem = document.getElementById("idx");
const x = listItem.nextSibling;
```

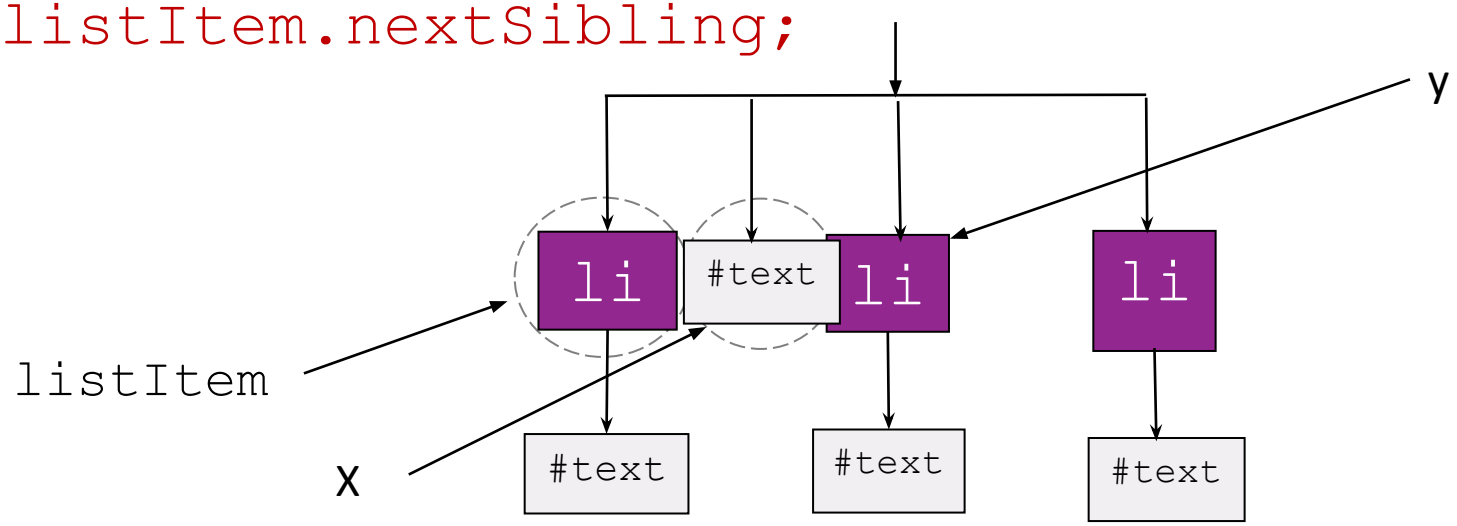


the same applies to previousSibling

Problem: white space between elements

- Most browsers (except IE) treat whitespace in the html document as text nodes

```
const x = listItem.nextSibling;
```

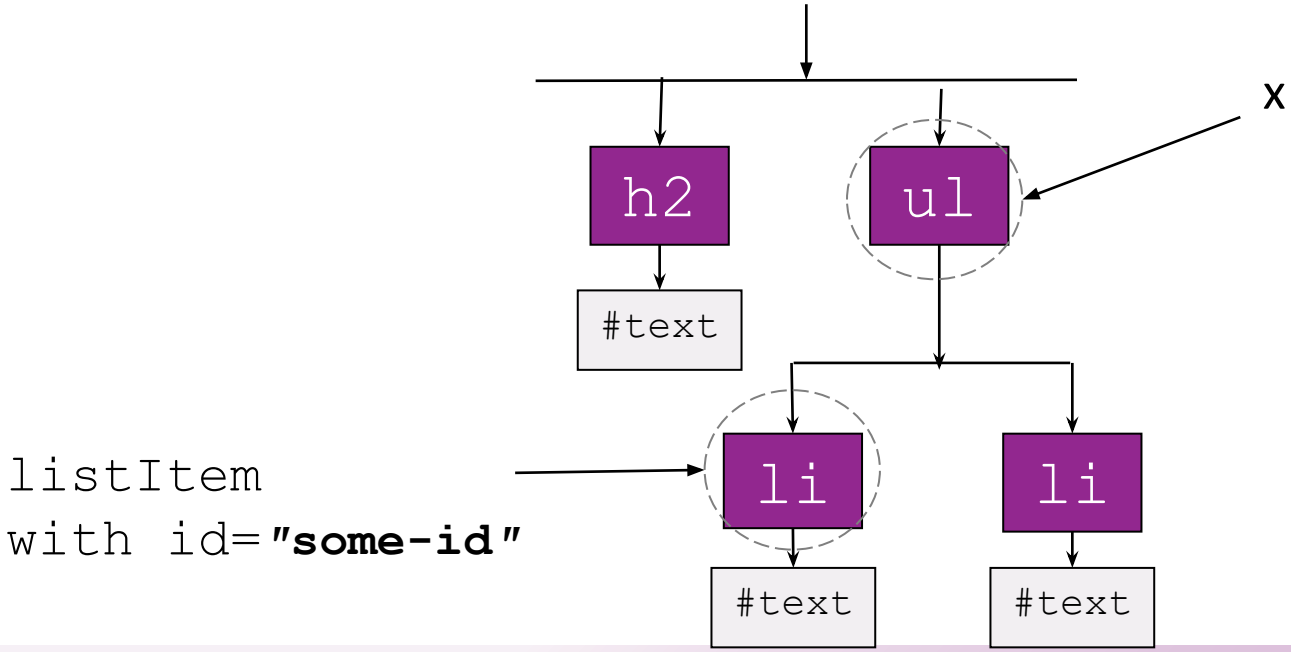



```
const y = listItem.nextElementSibling;
```

Human Potential Unlimited.

parentNode

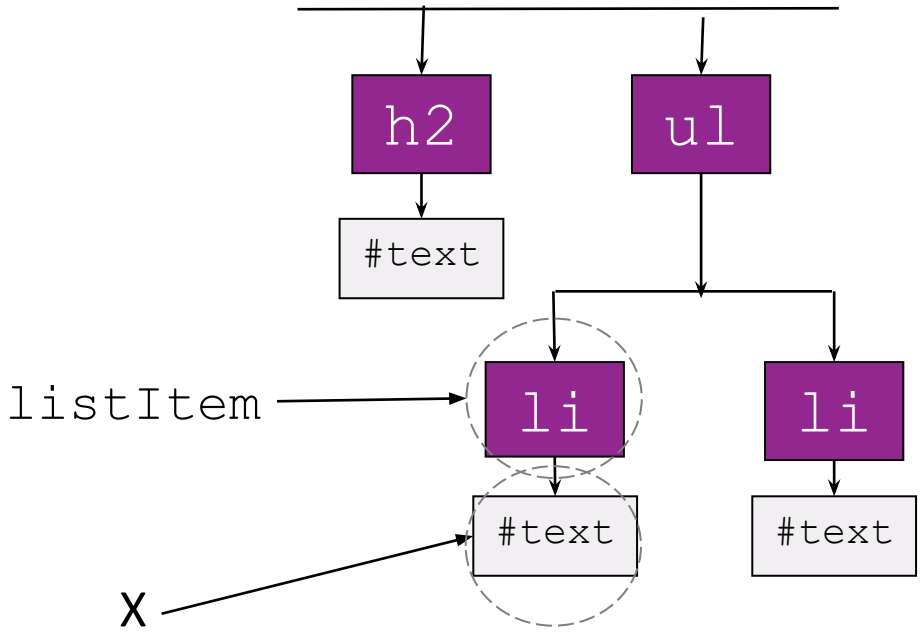
```
const listItem = document.getElementById("some-id");
const x = listItem.parentNode;
```



Human Potential Unlimited.

firstElementChild,lastElementChild

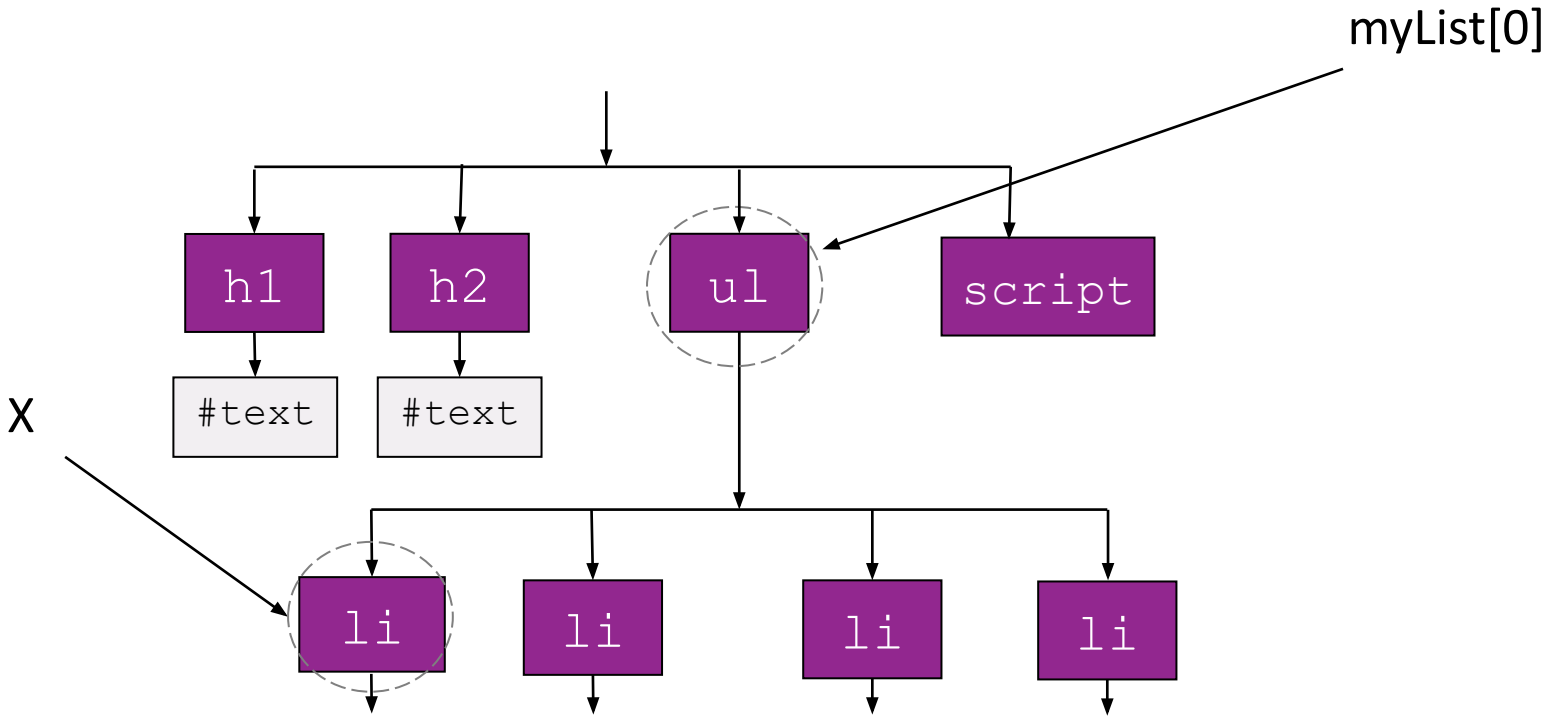
```
const listItem = document.getElementById("some-id");
const x = listItem.firstElementChild;
```



index

firstElementChild, lastElementChild

```
const myList = document.getElementsByTagName("ul");
const x = myList[0].firstElementChild;
```

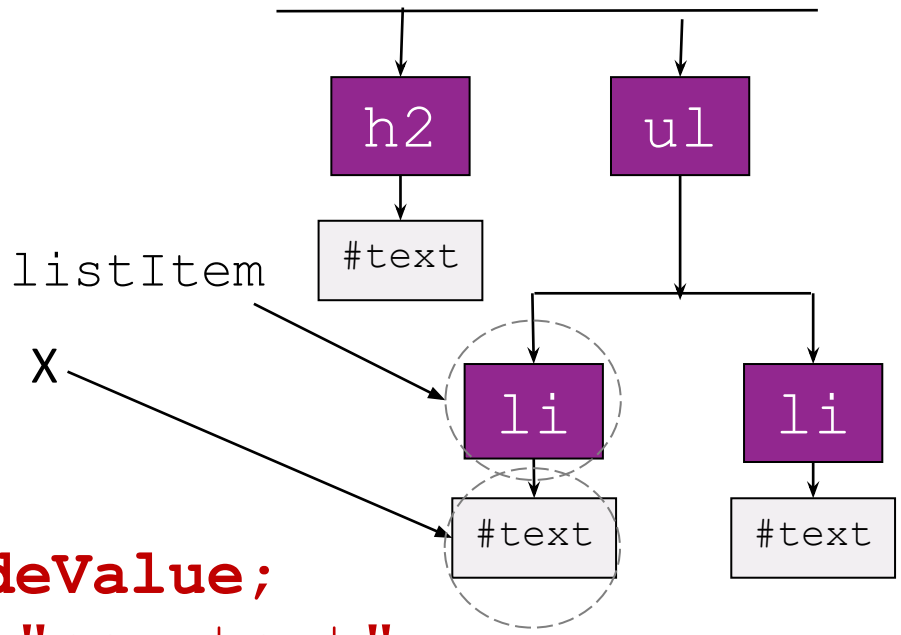


Text nodes: `nodeValue`

- nodeValue accesses text from a node

```
const listItem = document.getElementById("idx");

const x = listItem.firstChild;
const line = x.nodeValue;
```



- missing out x altogether

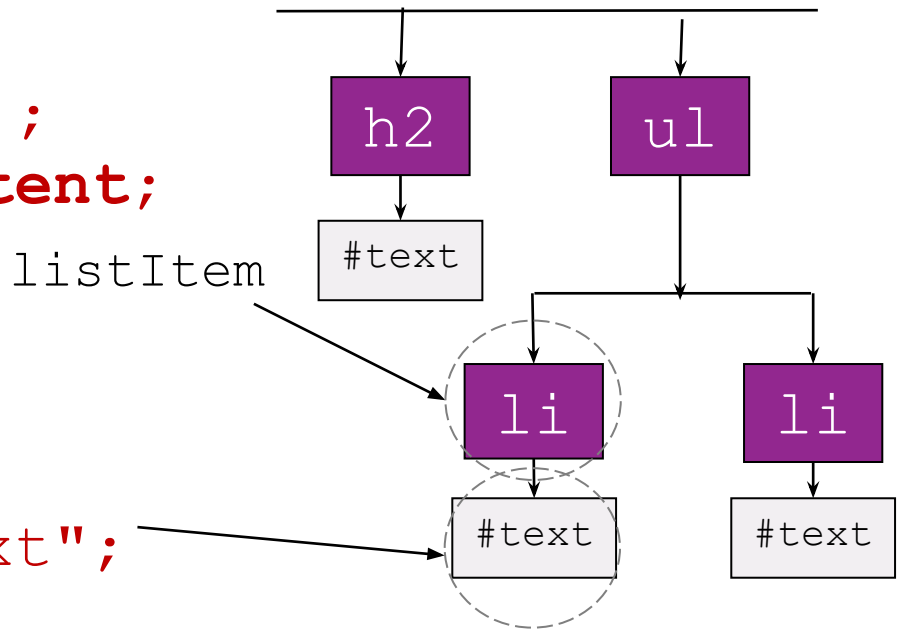
```
line = listItem.firstChild.nodeValue;
listItem.firstChild.nodeValue="new text";
```

index

Text content of a node: `textContent`

- nodeValue accesses text from a text node

```
const listItem =
document.getElementById("idx");
const line = listItem.textContent;
```




```
listItem.textContent ="new text";
```

Human Potential Unlimited.

Adding elements to the DOM tree

- we can create new elements, say an `<h2>` element
  ```
  const newHeader = document.createElement('h2');
  ```
- we can create new text nodes
  ```
  const newTxt = document.createTextNode("Great!");
  ```
- then append the text node to the new h2 node
  ```
  newHeader.appendChild(newTxt);
  ```
- then append the new h2 node to the document somewhere, say to a div
  ```
  const parentDiv = document.getElementById("page");
  parentDiv.appendChild(newHeader);
  ```

index

Adding text element

- We can create new elements, say, `<h2>` element

```
const newHeader = document.createElement('h2');
```

- We can set the text of previously created element

```
newHeader.innerText = "Hello World!";
```

- then append the new h2 node to the document somewhere, to a div, for example

```
const parentDiv = document.getElementById("page");
parentDiv.appendChild(newHeader);
```

index

ADDED

Adding elements using innerHTML

- with `createTextNode()` method or with `textContent` property we cannot include html code in the content of a node, **but with `innerHTML` we can**
- create a new h2 element

```
const newHeader = document.createElement('h2');
```

- write the code inside

```
newHeader.innerHTML = "<strong>Great</strong> " + "job";
```

- and then insert the node into DOM

```
const parentDiv = document.getElementById("page");
parentDiv.appendChild(newHeader);
```

Human Potential Unlimited.

Removing elements from the DOM tree

- store the element to be removed in a variable

```
const removeThis =
document.getElementById('losethis');
```

- store the parent of that element in a variable

```
const parentEl = removeThis.parentNode;
```

- remove the element from its parent

```
parentEl.removeChild(removeThis);
```

- Or simply remove it with command: `removeThis.remove()`

index

Accessing and changing attributes

- We use the methods hasAttribute() and getAttribute()

```
const firstItem=document.getElementById("an-id");

if(firstItem.hasAttribute("class")){
  const attrVal=firstItem.getAttribute("class");
}   // attrVal now has value "hot"
```

- use setAttribute() to give the attribute a value

```
firstItem.setAttribute("class","cool");
```

```
<div id="an-id" class="hot">
   content...
</div>
```

```
<div id="an-id" class="cool">
   content...
</div>
```

index

Accessing and removing attributes

- We use the methods hasAttribute() and removeAttribute()

```
const button = document.getElementById("btn");

if (button.hasAttribute("disabled")){
  button.removeAttribute("disabled");
}
```

- use setAttribute() to give the attribute a value

```
button.setAttribute("disabled", "");
```

index

Handling classes with JavaScript

- Toggling removes the class if it already exists and adds it if it does not exist

```
const firstItem=document.getElementById("an-id");

firstItem.classList.toggle("cool");
```

- Remove a class

```
firstItem.classList.remove("cool");
```

- Check if class exists

```
firstItem.classList.contains("cool");
```

- Add a class

```
firstItem.classList.add("cool");
```

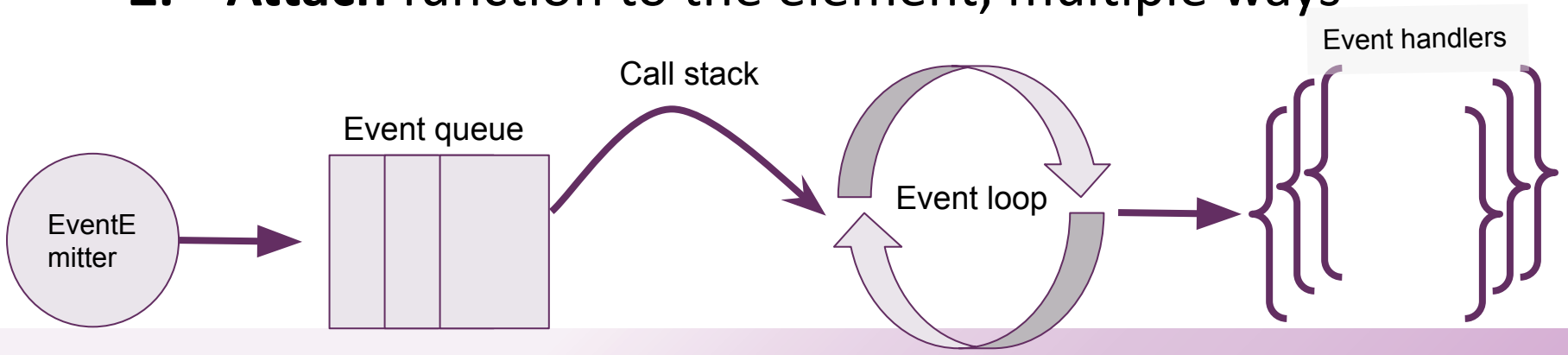
```
<div id="an-id">
   content…
</div>
```

```
<div id="an-id" class="cool">
   content…
</div>
```

index

Events

- enable event-driven nature of JS
- triggered by a user (or browser), e.g., user clicks a button or targets other interactive content
- HTML Element then emits an event
- To handle the event
    1. Write a function that handles the event
    2. **Attach** function to the element, multiple ways



Event object

JWT

JS 14
tapahtuman kohde this

`target` - the target of the event, e.g., the element that was pressed or hovered (the same as `this` inside an eventhandler)
`keyCode` - in case of a keypress, which key was pressed
`type` - which event happened (e.g., click, keypress,...)

Human Potential Unlimited.

Event types

- related to
  - browser: `load,unload,error,resize,scroll`
  - keyboard: `keydown, keyup,keypress`
  - mouse: `click,dblclick,mousedown,mouseup,mousemove,mouseover,mouseout`
  - focus events: `focus,blur`
  - connected to forms (interactive elements) : `input,change,submit, reset,cut,copy,paste,select`
  - DOM modifications: `DOMSubtreeModified,DOMNodeInserted, DOMNodeRemoved,MNodeInsertedIntodocument,DOMNodeRemovedFromdocument`
- more details in MDN

Events

Event as a parameter to event handler

- The first parameter of **every** eventhandler is an **event object,** which
  - carries information of the event, which triggered the eventhandler
  - information accessed via event object's properties and methods

```
function eventHandler() {
      let e = arguments[0];
      e.target.style.color = red;
}
```

  - the information includes, for example
    - the target of the event (`e.target`, the same as `this` inside an eventhandler function)
    - what was the event (`e.type`)
    - if it was a key event, which key was pressed (`e.keyCode`)
    - If it was a mouse event, where in the window the event happened (`e.clientX`, `e.offset`)

index

How an event is attached to an element?

(1) ~~HTML event-attributes, *on*event~~

- ~~`<a onclick="hide()">`~~

  ~~old way – use only to quick testing~~

(2) DOM event handlers

- in JavaScript, for example, `elem.onclick = hide;`
- element can be attached only with one function for each event

(3) DOM event listeners

- event can trigger several functions
- for example, `elem.addEventListener("click", hide);`

index

Adding event listener sets a function to be called asynchronously

element object

```
el.addEventListener('event', functionName [, Boolean]);
```

'bubbling' / 'capturing'; **false** by default, which implies bubbling

event, such as 'click', 'blur',...

the name of the event handler that will be performed, when the element launches the event (note: **no** parenthesis)

- An event can be attached with several event handlers per element

index

Addition and removal of a listener

```
function checkTheLength() {
  ...
  if (this.value.length < 8) {
     // error message
  }
}
// get the reference to an element
var el = document.getElementById('id');
// attach function checkTheLength to a blur event
el.addEventListener('blur', checkTheLength, false)
```

- The removal of a handler:

```
el.removeEventListener = ('event', functionName [, Boolean]);
```

- What if we want to pass parameters for the event handler?

Human Potential Unlimited.

Passing parameters to an event handler

- No parentheses - no parameters
  - Work-arounds:
    - a wrapper function, which calls the event handler

```
function wrapper() {
    checkTheLength(8)
}
```

    - the wrapper can also be **anonymous**:

```
function () {
    checkTheLength(8)
}
```

index

Different handler functions - different scopes

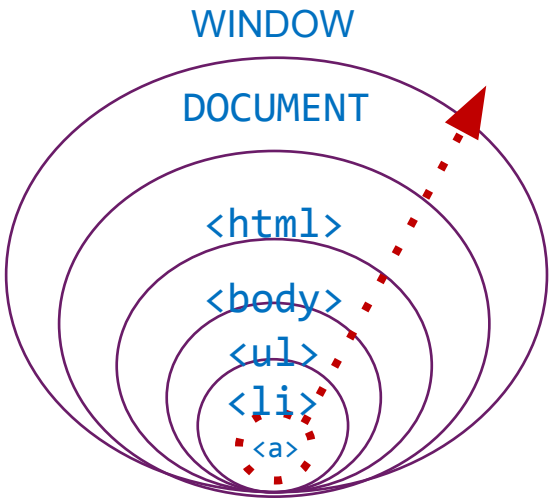
```
function checkTheLength(minLength) {
    let val = (this != window) ? this.value.length : null;
    if (!val && arguments.length > 1) val = arguments[1].value.length;
    if (val && val < minLength) console.log("Hello");
}

// a reference to an element
var el = document.getElementById('id');
// to attach function checkTheLength() to a blur event
el.addEventListener('blur', function () { checkTheLength(8) }, false);    //Not OK
el.addEventListener('blur', () => { checkTheLength(8) }, false);          //Not OK
el.addEventListener('blur', () => { checkTheLength(8, el) }, false);      //OK
```

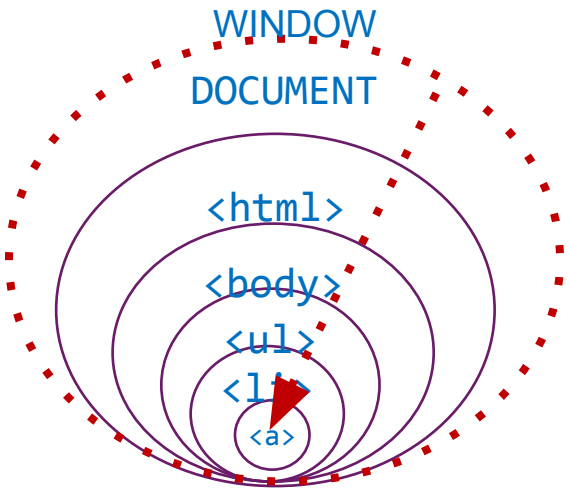
- problem: anonymous functions refer to a global scope (i.e., window object)

Event propagation

- HTML elements form a hierarchy
- When an element receives an event (e.g. mouseover) the same event goes also to the ancestors



event bubbling



event capturing

- this matters only if an event is attached not only to an element but to its ancestors, as well