DOM

IIKG1002/IDG1011 – Front-end web development

Johanna Johansen
johanna.johansen@ntnu.no

Recap

- We have seen how we can create models of things from the real world
- Web browsers create similar models of
  - the browser window that the page is shown in and
  - of the web page they are showing
- The *Browser Object Model* creates a model of the browser tab or window
- The top most object is the `window` object → represents the current browser window/tab

Recap

- The `window` objects has as child objects `document, history, location, screen,navigator`
- These are built-in objects
  - a toolkit
  - already implemented objects that come with functionality that are needed by many scripts
  - *properties*, e.g., `window.innerHeight / window.innerWidth, window.location, window.history, window.history.length`
  - *methods*, e.g., `window.print()`

Recap

- The Document Object Model (DOM) creates a model of the current web page
- The top most object is the `document` object
  - represents the web page as a whole
  - represents the web page that loaded in the current browser window/tab
  - props and methods, e.g.:
    - `document.title`
      - `title` is one of the properties of the document object
      - tells us what is between the opening `<title>` and closing `</title>` tag for the web page
    - `document.getElementById()`
      - returns the element if the value of the id attribute matches

DOM

- The Document Object Model (DOM)
  - it is not part of the HTML or JS
  - it is implemented by the major browser makers
  - specifies
    - how a browser should create a model of a HTML page
    - how JavaScript can access and update the contents of a web page
  - while the page is displayed in the browser window

DOM tree

- a model of a web page created by the browser
- is created when the browser loads a web page
  - following the specifications of the DOM
- the model is stored in the memory of the browser
- the tree is made of objects that are structured hierarchically, as in a family tree
  - we use the same terms to refer to these objects as a family tree: *parents*, *children*, *siblings*, *ancestors*, and *descendants*
- each object represents a part of the page
  - the page that is loaded in the browser

DOM node

- Every object in the DOM tree is referred to as a **DOM node**
- one node is created for every HTML element, attribute, and text
- each node is an object with methods and properties
- Types of nodes: *document*, *element*, *attribute*, and *text*

DOM nodes

- the `document` node
  - found at the top of the tree
  - represents the entire page
  - you use it as starting point to reach the other child nodes, by using the **dot notation**
    - e.g., `document.getElementById();`

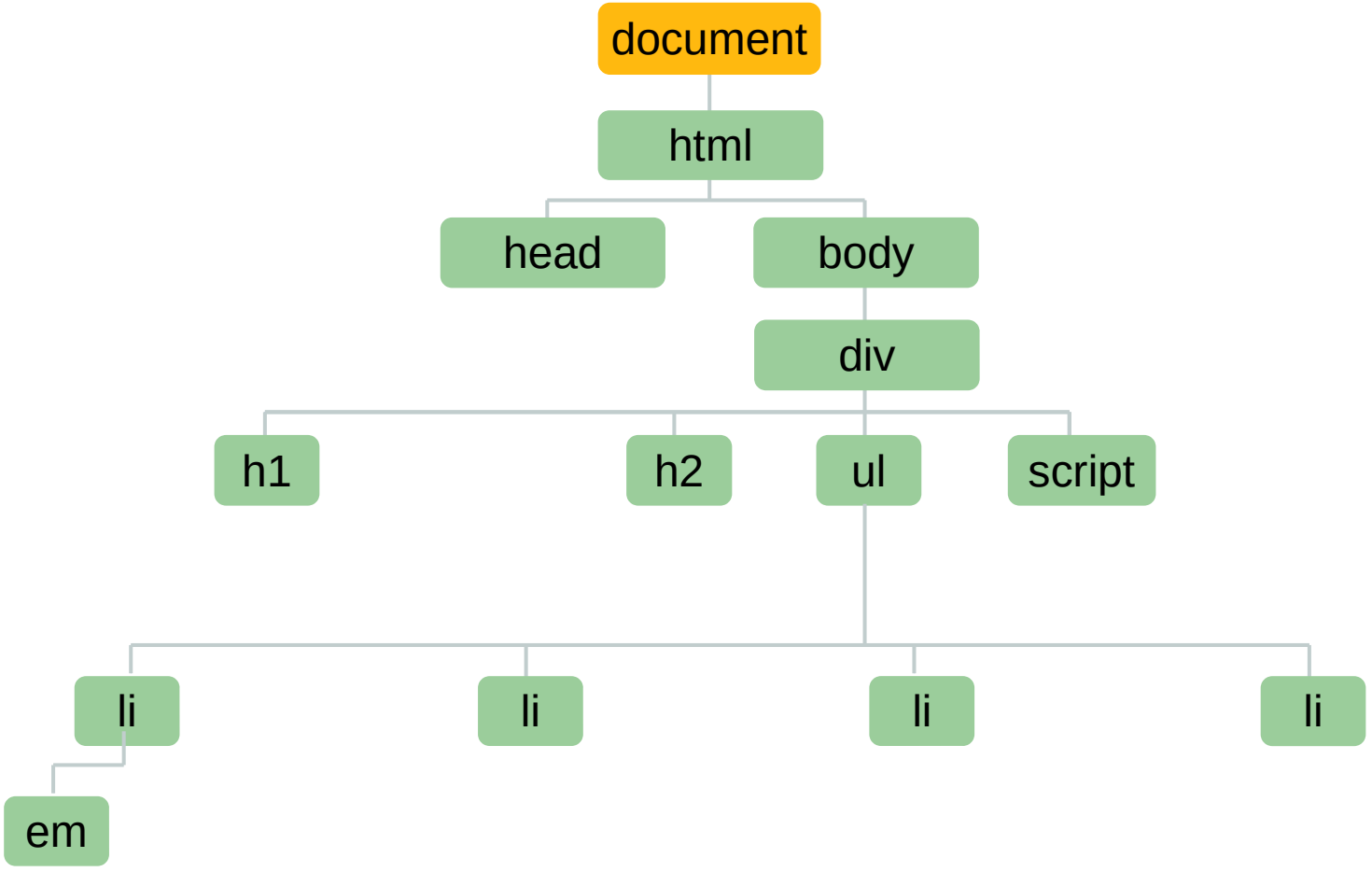
```
<html>
  <head>
   …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em> figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```


document

DOM nodes

- the **element** nodes
  - represent each HTML element on our page
  - when we want to access the DOM tree we start **first** by accessing these element nodes
  - **then** we can access the text and attribute nodes of these elements if we need to
  - we **first** learn methods that allow us to access element nodes, before we learn to access and change text or attributes

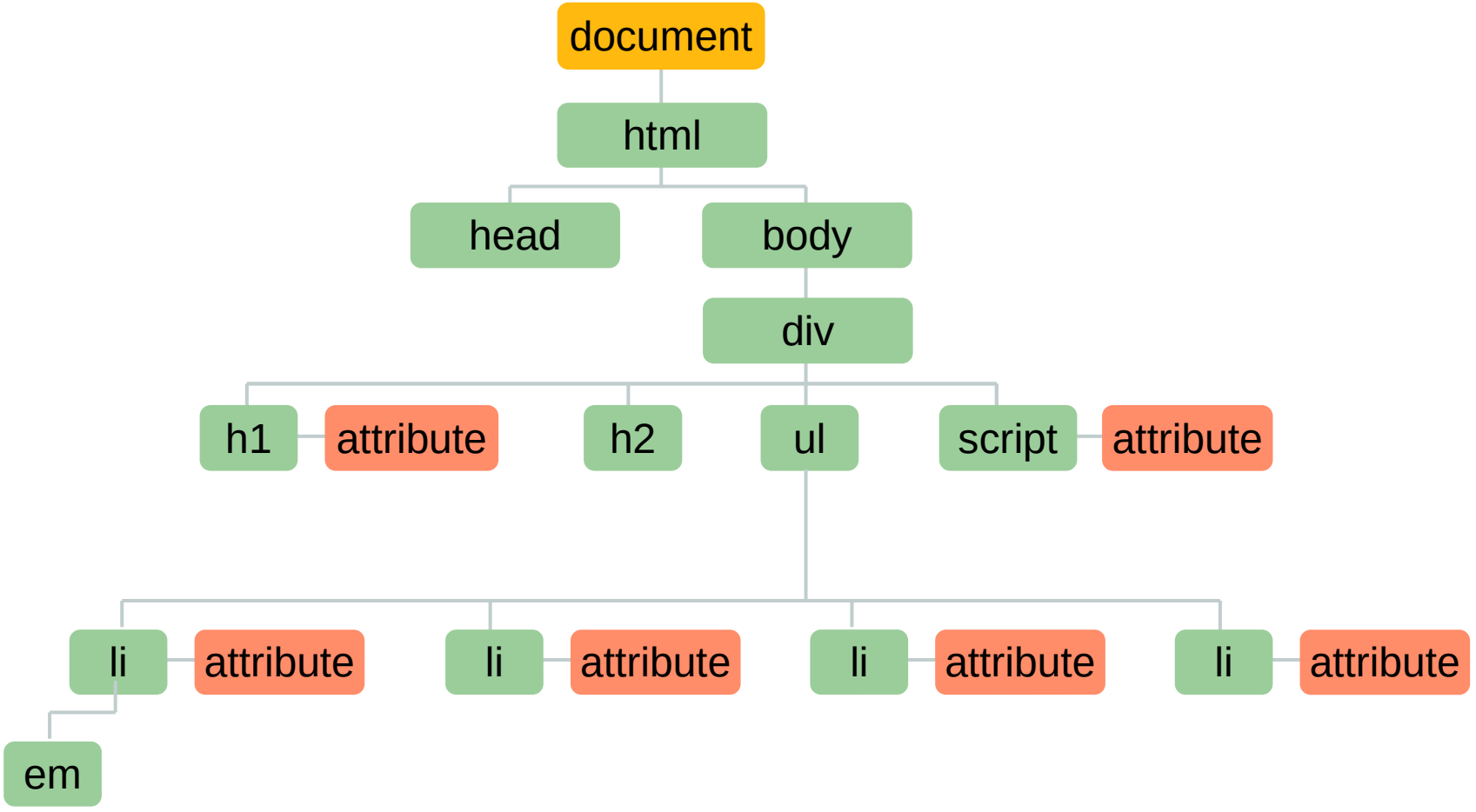
```
<html>
  <head>
    …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em>figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```


document
html
head
body
div
h1
h2
ul
script
li
li
li
li
em

DOM nodes

- the **attribute** nodes
  - are the attributes that we find in the opening tags of the HTML elements
  - they are part of the element that they carry them, and not children of that element
  - to make changes or read these attributes we need to access first the elements that carry them
  - e.g., the value of the *class* attribute is often changed to trigger new CSS rule

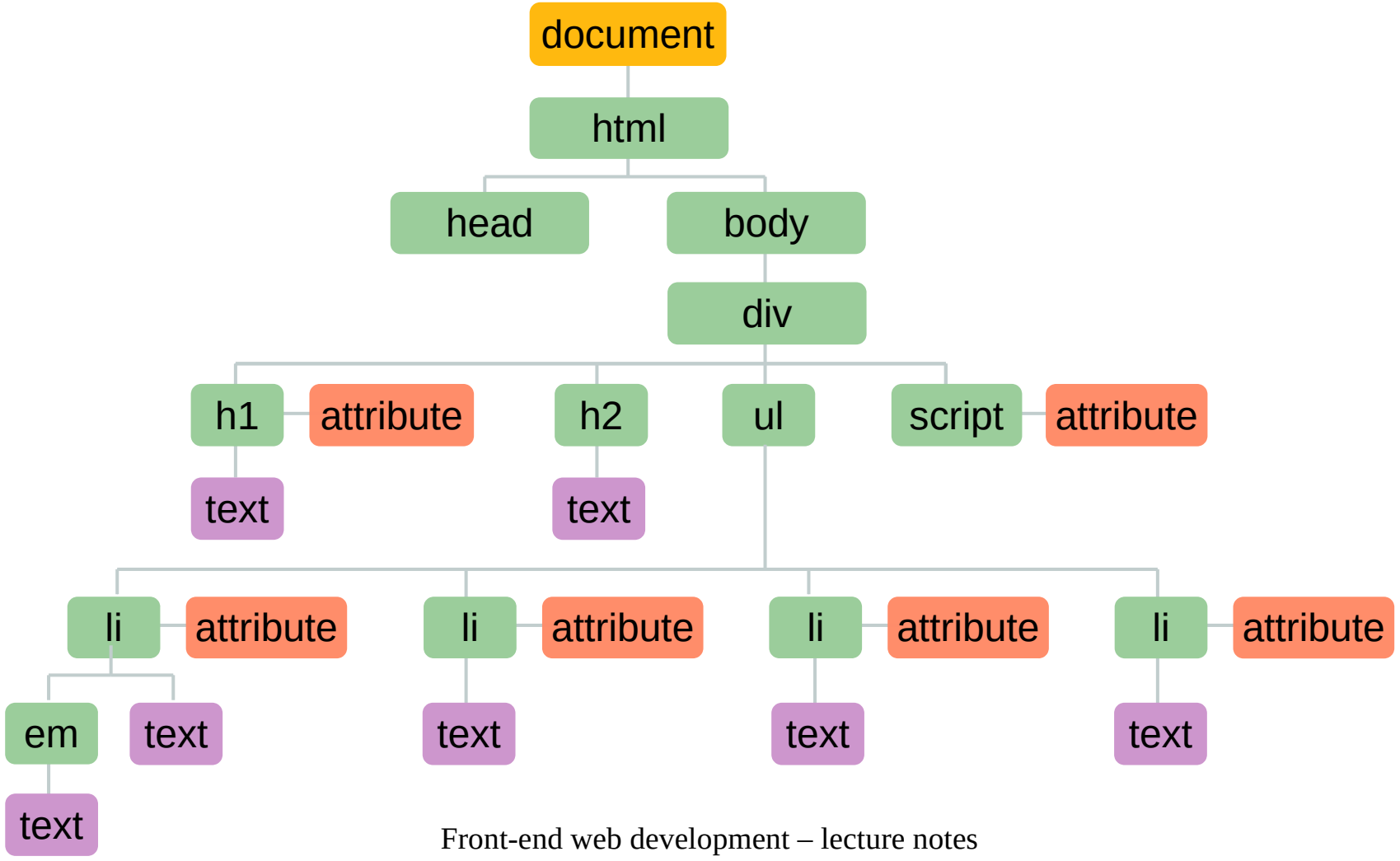
```
<html>
  <head>
   …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em>figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```


document
html
head
body
div
h1
attribute
h2
ul
script
attribute
li
attribute
li
attribute
li
attribute
li
attribute
em

DOM nodes

- the **text** nodes
  - can be accessed only after you first accessed the element node they belong to
  - they cannot have children
  - only the element nodes can have children

```
<html>
  <head>
   …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em>figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```


document
html
head
body
div
h1
attribute
h2
ul
script
attribute
text
text
li
attribute
li
attribute
li
attribute
li
attribute
em
text
text
text
text
text

- We have seen how the browser models the web page
- Now we look at how to access and change the HTML page
- The DOM specifies how JavaScript can access and update the contents of a web page
  - DOM is an **Application Programming Interface (API)**
    - specifies how the JS can communicate with the browser
  - APIs let programs (scripts) talk to each other
  - User Interfaces → let humans interact with programs

Accessing and updating the DOM tree involves

1) Locate the node that represents the element we want to work with
2) Use its text content, child elements, and attributes

Locating nodes

- The element nodes (or just nodes) are DOM representations of the HTML elements
- Methods that find elements in the DOM tree are called **DOM queries**
- Nodes can be located also by traversing the DOM

```
<html>
  <head>
   …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em> figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```

DOM queries

- methods for selecting individual elements:
  - `getElementById(value of the id attribute)`
    - uses the value of an element's *id attribute*
    - since the id is unique, you are getting only one element
    - e.g., `getElementById('one')` → selects the

```
<li id='one'
class='hot'><em>fresh</em>figs</li>
```

DOM queries

- methods for selecting individual elements:
  - …
  - `querySelector(CSS selector)`
    - uses a CSS selector
    - returns the first matching element
    - is a newer addition to JS than *getElementById()*
    - it is more flexible than *getElementById()* because its parameter is a selector, thus being able to accurately target many more elements
    - e.g., `querySelector('li.hot')` → selects the
      `<li id='one' class='hot'><em>fresh</em>figs</li>`

DOM queries

- return also a **collection** of nodes / multiple element nodes
  - multiple element nodes are called a **NodeList**
- methods for selecting multiple elements:
  - `getElementsByClassName(value of the class attribute)`
    - selects one or more elements given the value of their *class* attribute
    - the value of the class attribute can contain several class names, each separated by space
    - e.g., `getElementsByClassName('hot')` → selects the

```
<li id='one' class='hot'><em>fresh</em>figs</li>
<li id='two' class="hot">pine nuts</li>
<li id='three' class="hot">honey</li>
```

DOM queries

- methods for selecting multiple elements:
  - …
  - `getElementsByTagName(tag name)`
    - selects all the elements on the page with the specified tag name
    - we do not include the angle brackets that surround the tag name in the HTML
      - just the letters inside the brackets
    - e.g., `getElementsByTagName('li')` → selects the

```
<li id='one' class='hot'><em>fresh</em>figs</li>
<li id='two' class='hot'>pine nuts</li>
<li id='three' class='hot'>honey</li>
<li id='four'>balsamic vinegar</li>
```

DOM queries

- methods for selecting multiple elements:
  - …
  - `querySelectorAll(css selector)`
    - for selecting **one** element with the help of a CSS selector we had the `querySelector()` method
    - returns all the elements that match the respective CSS selector
    - e.g., `querySelectorAll('li.hot')` → selects the

```
<li id='one' class='hot'><em>fresh</em>figs</li>
<li id='two' class='hot'>pine nuts</li>
<li id='three' class='hot'>honey</li>
```

Which selection methods to use

- Older element selection methods; they are more or less obsoleted now
  - `getElementById(),`
  - `getElementsByClassName(), getElementsByTagName()`
- Use
  - `querySelector(css selector)`
  - `querySelectorAll(css selector)`

Which selection methods to use

- The CSS selectors allows us to reference elements within a document by
  - type (tag name, e.g., `div`)
  - ID (e.g., `#nav`)
  - class (e.g., `.warning`)
  - attributes (e.g., `p[lang="fr"]`)
  - position within the document (e.g., `body>h1:first-child`)

Which selection methods to use

Examples of how you can use these methods instead of the older ones:

```
// Look up an element by id
document.getElementById('section1');
document.querySelector('#section1');

//Look up all elements that have a name='color' attribute
document.getElementsByName('color');
document.querySelectorAll('*[name=“color”]');
```

```
// Look up all <h1> elements in the document
document.getElementsByTagName('h1');
document.querySelectorAll('h1');

// Look up all elements that have class 'tooltip'.
document.getElementsByClassName('tooltip');
document.querySelectorAll('.tooltip');
```

Good practice

- Find the quickest way to access an element within your web page
  - makes the page seem faster and/or more responsive
  - implies evaluating the minimum number of nodes on the way to the element you want to work with
    - e.g., `querySelector('#one')` will always be the faster if you are looking for only one element, because the id is unique

Syntax

- `document.querySelector('#one');`
  - we always access the nodes via the *document* object
  - we use the *dot notation* to access the method
  - the *parameter* is a string
- `let itemOne = document.querySelector('#one');`
  - **catching the selection** → we store the result of the query in a variable
  - we store a **reference** to where that node is in the DOM tree
    - for the case we want to use it more than once
    - the browser does not have to search through the DOM tree to find the same element again
    - we are storing the **location** of the element(s) within the DOM tree

Exercise

- Open the “Initial page” in your browser

  https://javascriptbook.com/code/c05/

- Open the Developer Tools and navigate to the **Console** tab
- Test all the methods that we have just learned by writing them in the Console and Running them to see the output
  - **Note**: do not forget to add `document.` in front of the methods.

NodeList

- is a **collection** of element nodes
- When a method *can* return more than one node
  - even if it only finds one matching element
  - it returns a **NodeList**
- Each node in the list is given an index number
- The element nodes are stored in the NodeList in the order they appear in the HTML page
- Look like arrays and are numbered like arrays
  - they are not arrays, but a collection

NodeList

```
<html>
  <head>
   …
  </head>
  <body>
    <div id="page">
      <h1 id="header">List King</h1>
      <h2>Buy groceries</h2>
      <ul>
        <li id="one" class="hot"><em>fresh</em> figs</li>
        <li id="two" class="hot">pine nuts</li>
        <li id="three" class="hot">honey</li>
        <li id="four">balsamic vinegar</li>
      </ul>
    </div>
  </body>
</html>
```

- Methods that can return more than one element:
  - `getElementsByTagName()`
  - `getElementsByClassName()`
  - `querySelectorAll()`
- `querySelectorAll('li')` → selects a list of elements

stored at index nr. 0	`<li id='one'class='hot'><em>fresh</em>figs</li>`
stored at index nr. 1	`<li id='two' class='hot'>pine nuts</li>`
stored at index nr. 2	`<li id='three' class='hot'>honey</li>`
stored at index nr. 3	`<li id='four'>balsamic vinegar</li>`

NodeList

- What can we do with a list
  - select one element from it
  - loop through each item and perform the same actions on each element in the list
- Like any other objects, the NodeList has properties and methods
  - `length` property → how many items are in the NodeList
  - `item(index number)` method
    - returns a specific node from the NodeList based on the index number
    - it is more common to use the array syntax, with the square brackets

NodeList

1) Selecting one element

- using the `item()` method or the array syntax
- both require to provide the index nr. of the element you want to select
- the `item()` method takes the index nr. as a parameter

NodeList

- Good practice
  - check first that the list contains at least one element
    - you do not want to execute code if the list is empty – is a loss of resources
    - we use the *length* property of the NodeList for this purpose
    - if the list is empty, the check with the *length* prop will return the number `0`
  - as with the DOM queries store the NodeList in a variable if we are going to use the NodeList several times

NodeList

The *item()* method

```
let nodeElements = document.querySelectorAll('li');
if (nodeElements.length >= 1) {
    let firstNode = nodeElements.item(0);  // returns <li
    id='one'class='hot'><em>fresh</em>figs</li>
}
```

Array syntax

```
let nodeElements = document.querySelectorAll('li');
if (nodeElements.length >= 1) {
    let firstNode = nodeElements[0];   // returns <li
    id='one'class='hot'><em>fresh</em>figs</li>
}
```

NodeList

2) Repeating actions for each element in a NodeList

- we use a for *loop* to go through each element in the NodeList
- all the statements inside the curly braces are applied to each element in the NodeList one-by-one

```
let hotItems = document.querySelectorAll('li.hot');

for (let i = 0; i < hotItems.length; i++) {

    hotItems[i].className = 'cool';   // changes the value of
    the class attribute of each item from hot to cool

}
```

Exercise

- Copy the HTML code from within the <body></body> tags from here

  https://javascriptbook.com/code/c05/initial-page.html

  in the **HTML editor** of codepen
- Copy the CSS of this page

  https://javascriptbook.com/code/c05/css/c05.css

  in the **CSS editor** of codepen

Exercise

- CSS
  - in the CSS editor add the following rule

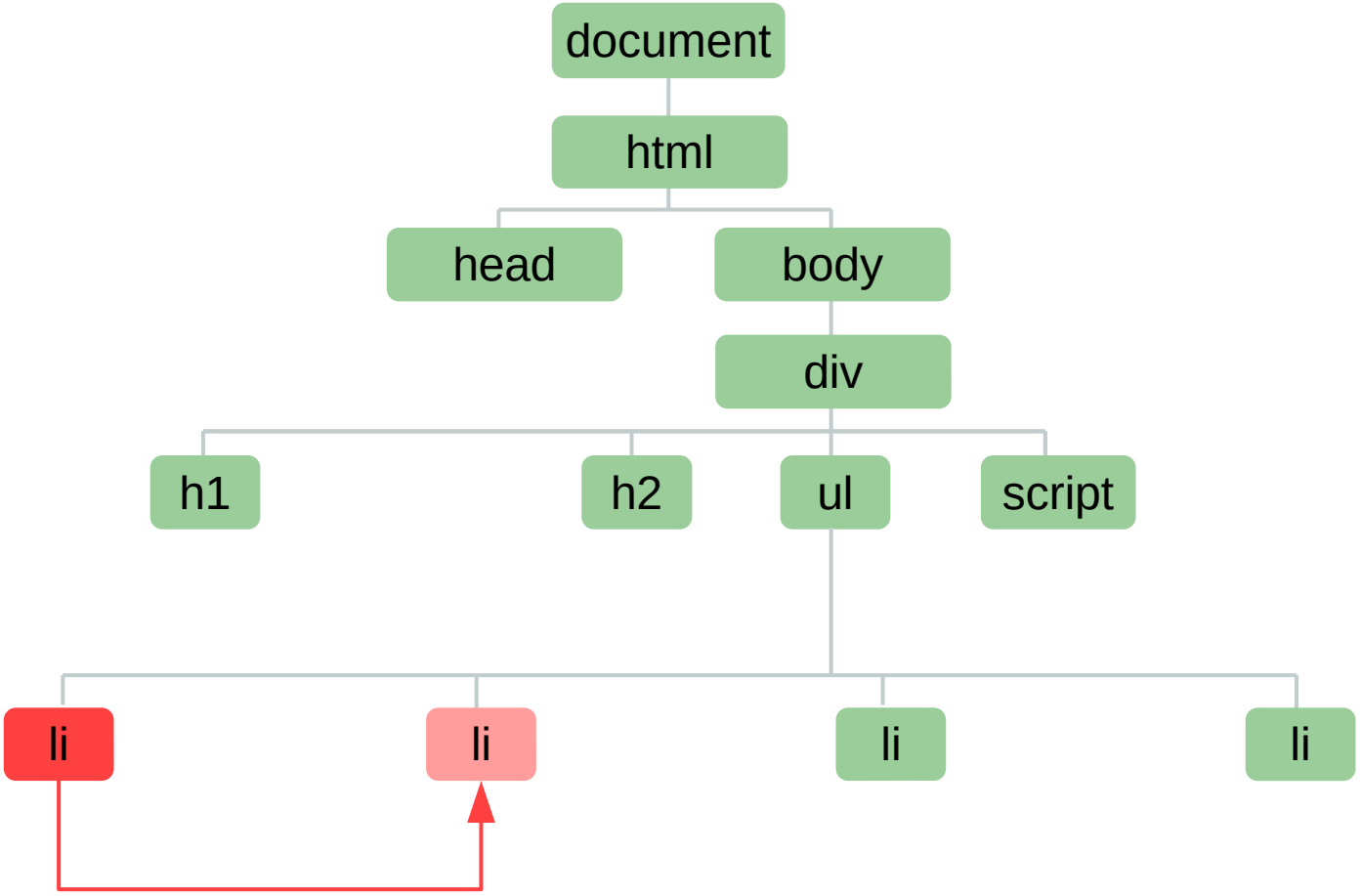
```
/** Add a heart symbol before the favorite items in the list using the pseudo-element ::before
   * For a reminder of what pseudo-elements are see p. 289 of the “HTML & CSS” syllabus book by Duckett or
   * https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements
   */

.favorite::before {
  content: "♥";
}
```

Exercise

- JavaScript implementation
  - following the example on p. 204 of the Duckett book (“JavaScript & jQuery”) add the class name “favorite” to the first two elements in the list by using a *for loop*.
    - a heart symbol should be added to the first two items in the list

Traversing the DOM

- Traversing between element nodes is another method to locate nodes, besides DOM queries
- We move from one initial element node to a “related” element node
  - “related” like in family relationships
  - we use methods that are named based on family tree terminology: *parent*, *sibling*, *child*
- We need to first locate one node


document
html
head
body
div
h1
h2
ul
script
li
li
li
li

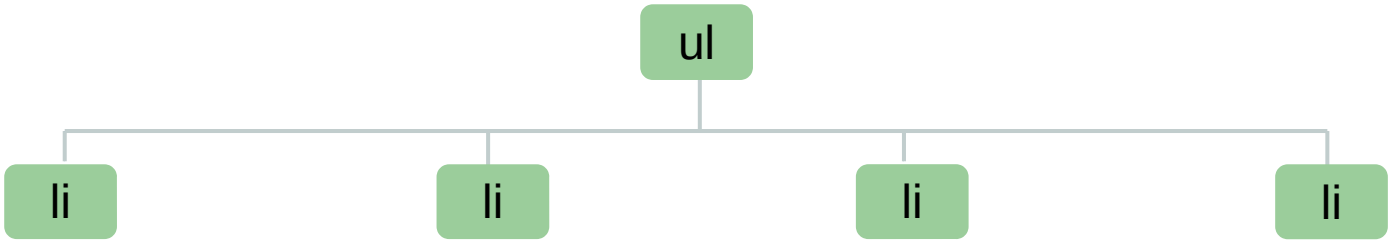
Traversing the DOM

- We use **properties** on the **current node** to find structural related portions
  - if the current node does not have a parent / sibling / child the result will be `null`
  - the properties are read only
  - only used for selection, as with the DOM queries, not for updating parent/sibling/children

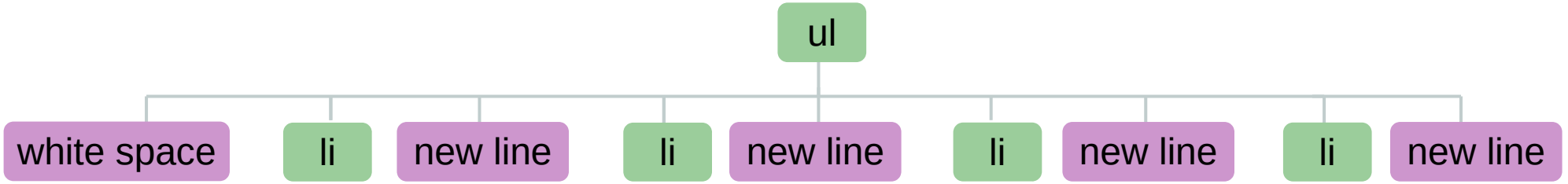
Traversing the DOM

- Properties that can be used for getting the **element nodes only**
  - `parentNode, children, firstElementChild / lastElementChild, nextElementSibling / previousElementSibling`
- Properties that in addition to the **elements** also return **text** and **comment nodes**
  - `childNodes, firstChild / lastChild, nextSibling / previousSibling`
  - the **white space** between elements is also treated as a text node
  - these properties are very sensitive to variations in the document text
    - e.g, inserting/deleting the new lines after the `<li>` elements in the HTML code, the number of the children of the `ul` node will increase/decrease

```
<ul>
  <li id="one" class="hot"><em>fresh</em> figs</li>
  <li id="two" class="hot">pine nuts</li>
  <li id="three" class="hot">honey</li>
  <li id="four">balsamic vinegar</li>
</ul>
```




```
document.querySelectorAll('ul')[0].children.length;
// returns 4 children
```

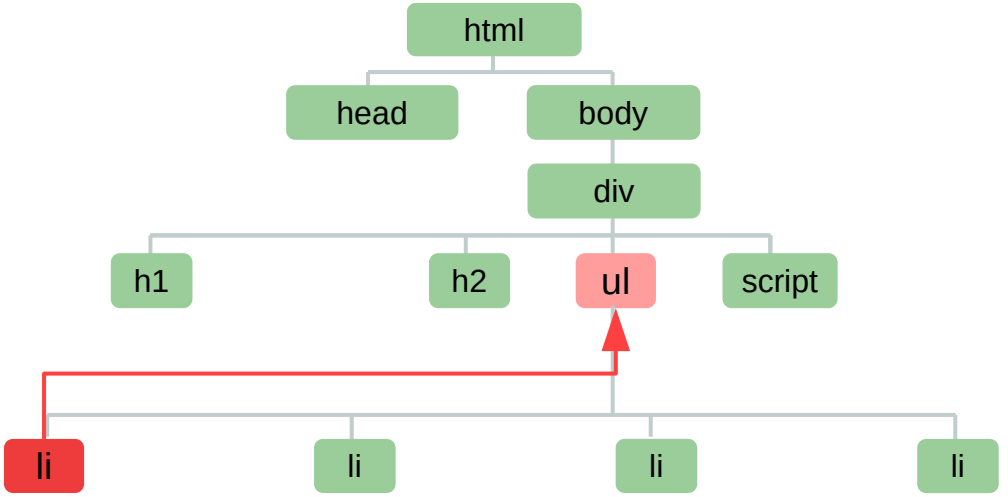



```
document.querySelectorAll('ul')[0]
.childNodes.length; // returns 9 children
```

Traversing the DOM

`parentNode`

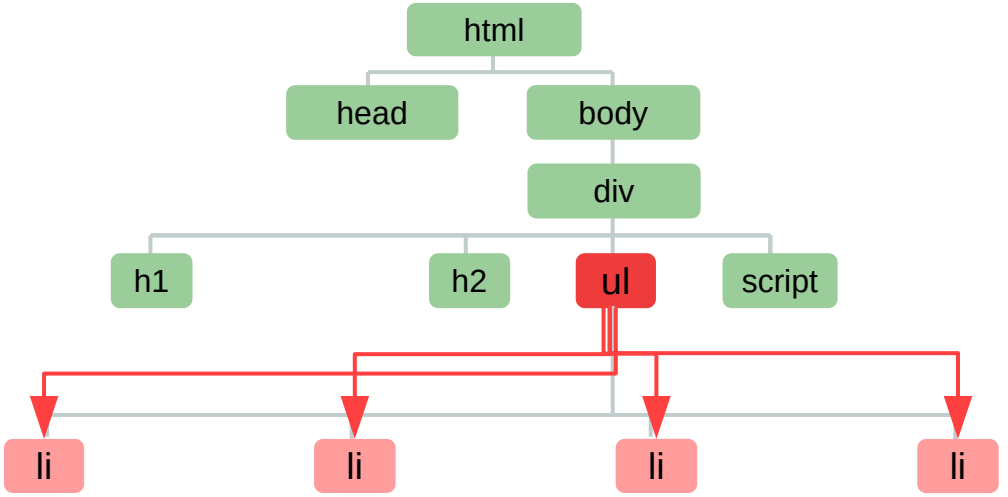
- selects the parent of the current node
- returns just one element
- for the first list `(li)` element in the list the parent element is the `ul`



Traversing the DOM

`children`

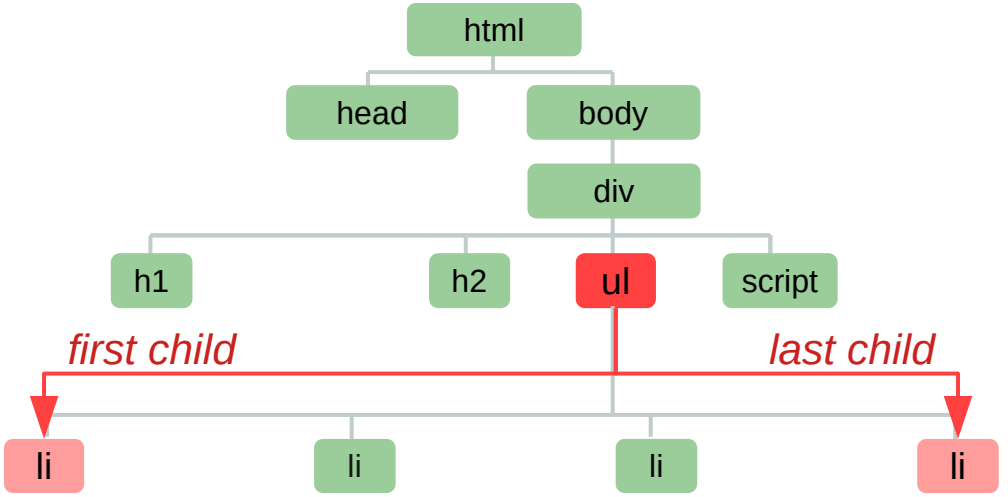
- returns a NodeList containing the element children of the current element (`ul`)
- `childElementCount`
  - returns the number of element children
  - returns the same value as `children.length`



Traversing the DOM

`firstElementChild / lastElementChild`

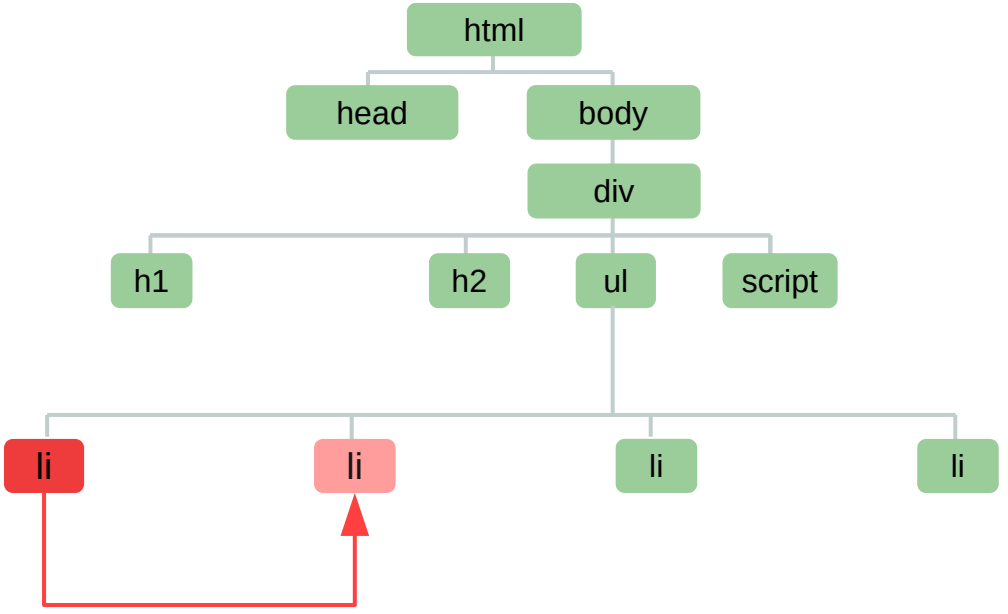
- find the first / last child of the current element
- the current element being the `ul` in our DOM tree
  - the first child is the first `li` in the list
  - the last child is the last `li` in the list



Traversing the DOM

`previousElementSibling` / `nextElementSibling`

- selects the previous / next sibling of a node
- for the first list (`li`) element in the list
  - there is no previous sibling → we will get a *null* back
  - there is a *next sibling* → the node representing the second element in the list, the second `li`



Live & static NodeList

- live NodeList
  - when the script updates the page, the NodeList is updated as well
  - the methods that begin with `getElementsBy...` return live NodeLists
- static NodeList
  - when the script updates the page, the NodeList is **NOT** updated
  - the methods that begin with `querySelector...` return static NodeLists

Example for live vs. static NodeLists

- Modify the example here

  https://javascriptbook.com/code/c05/initial-page.html

  - HTML

    add the following paragraph after the list
    ```
    <p id="nodeListDisplay"</p>
    ```

  - CSS

    ```
    .favorite::before {
      content: "♥";
    }
    ```

  - JavaScript → see on the next slide

```
let para = document.querySelector('#nodeListDisplay');

let listElStatic = document.querySelectorAll('li.hot');
let listElLive = document.getElementsByClassName('hot');

para.innerHTML = 'Static: ' + listElStatic.length + ' Live: ' +
listElLive.length + '<br />';

for (let el of listElStatic) {
  el.className = 'cool';
}

para.innerHTML += 'CHANGED > Static: ' + listElStatic.length + ' Live: ' +
listElLive.length;


for (let i = 0; i < listElStatic.length; i++) { // change listElStatic with
listElLive to see what happens
  if (i === 0) {
    listElStatic[i].className = 'favorite';
  }
}
```

Next lecture:
Get/update the content of the nodes