Introduction to JavaScript

Lecture 15

DOM And Dom Manipulation

What is the DOM

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

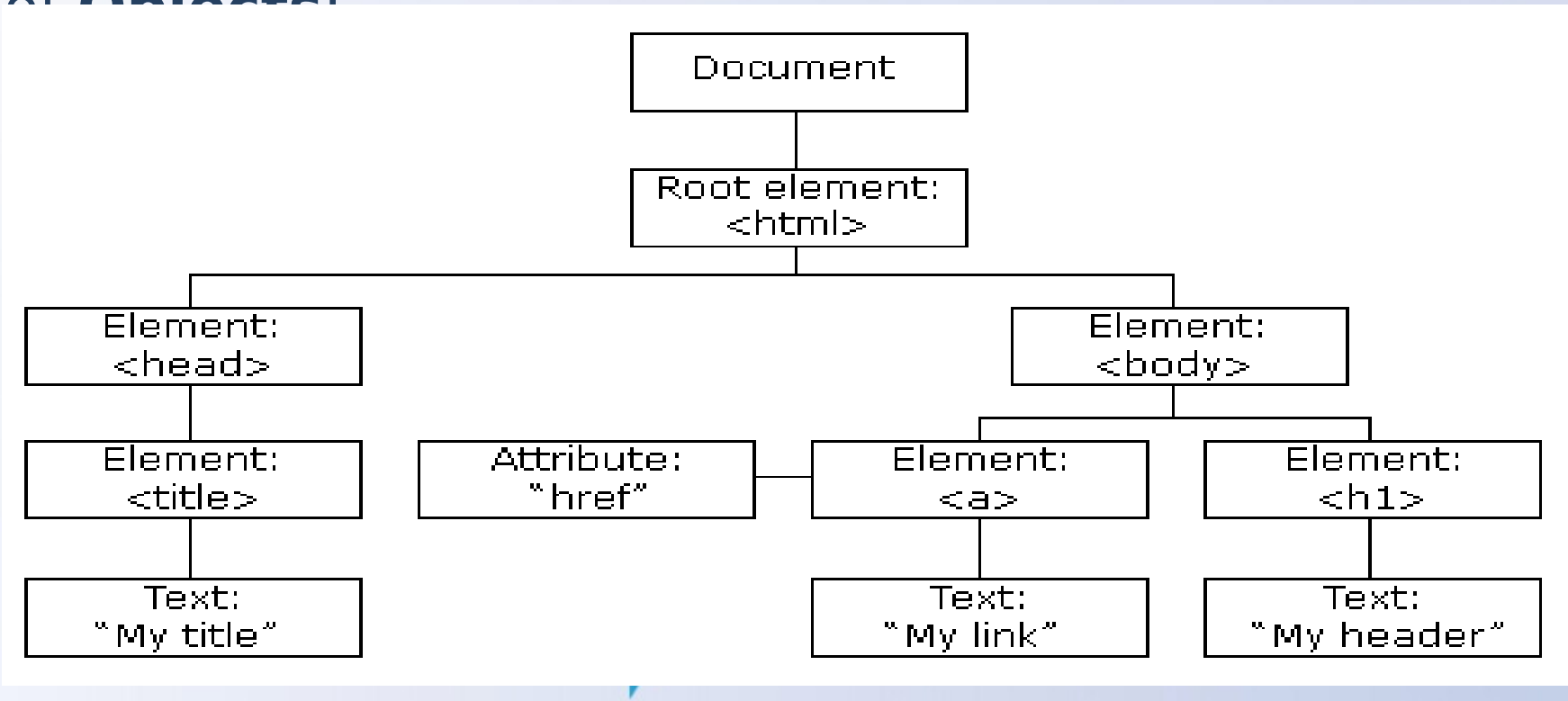
The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The **HTML DOM** model is constructed as a tree of **Objects**:



What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements
- In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

How can we manipulate the DOM with JavaScript

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

Traversing the DOM

- The HTML DOM can be accessed with JavaScript (and with other programming languages).
- In the DOM, all HTML elements are defined as **objects**.
- The programming interface is the properties and methods of each object.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element).

Methods for traversing the DOM

Everything starts with the document – this is the root node.

You can find elements in the DOM with following methods:

- document.getEmementById('x') – returns element with id 'x' or null;
- document.getElementsByTagName('tagname') – returns collection of all elements with a tag name 'tagname';
- document.getElementsByClassName('class') – returns collection of all elements with a class name 'class';
- document.forms[], document.images[] etc. - Finding elements by HTML element objects

These methods return collections containing DOM nodes or a DOM Node, represented by JavaScript object/s/.

Selectors API

One of the most popular capabilities of JavaScript libraries is the ability to retrieve a number of DOM elements matching a pattern specified using CSS selectors. Indeed, the library jQuery (www.jquery.com) is built completely around the CSS selector queries of a DOM document in order to retrieve references to elements instead of using getElementById() and getElementsByTagName().

At the core of Selectors API Level 1 are two methods: querySelector() and querySelectorAll().

On a conforming browser, these methods are available on the Document type and on the Element type.

Selectors API Level 1 was fully implemented in Internet Explorer 8+, Firefox 3.5+, Safari 3.1+, Chrome, and Opera 10+

The querySelector() Method

The querySelector() method accepts a CSS query and returns the fi rst descendant element that matches the pattern or null if there is no matching element. Here is an example:

```
//get the body element
var body = document.querySelector("body");

//get the element with the ID "myDiv"
var myDiv = document.querySelector("#myDiv");

//get first element with a class of "selected"
var selected = document.querySelector(".selected");
```

When the querySelector() method is used on the Document type, it starts trying to match the pattern from the document element; when used on an Element type, the query attempts to make a match from the descendants of the element only.

The CSS query may be as complex or as simple as necessary. If there's a syntax error or an
unsupported selector in the query, then querySelector() throws an error.

The querySelectorAll() Method

The querySelectorAll() method accepts the same single argument as querySelector() — the CSS query — but returns all matching nodes instead of just one. This method returns a static instance of NodeList.

Any call to querySelectorAll() with a valid CSS query will return a NodeList object regardless of the number of matching elements; if there are no matches, the NodeList is empty.

As with querySelector(), the querySelectorAll() method is available on the Document, DocumentFragment, and Element types. Here are some examples:

```
//get all <em> elements in a <div> (similar to getElementsByTagName("em"))
var ems = document.getElementById("myDiv").querySelectorAll("em");

//get all elements with class of "selected"
var selecteds = document.querySelectorAll(".selected");

//get all <strong> elements inside of <p> elements
var strongs = document.querySelectorAll("p strong");
```

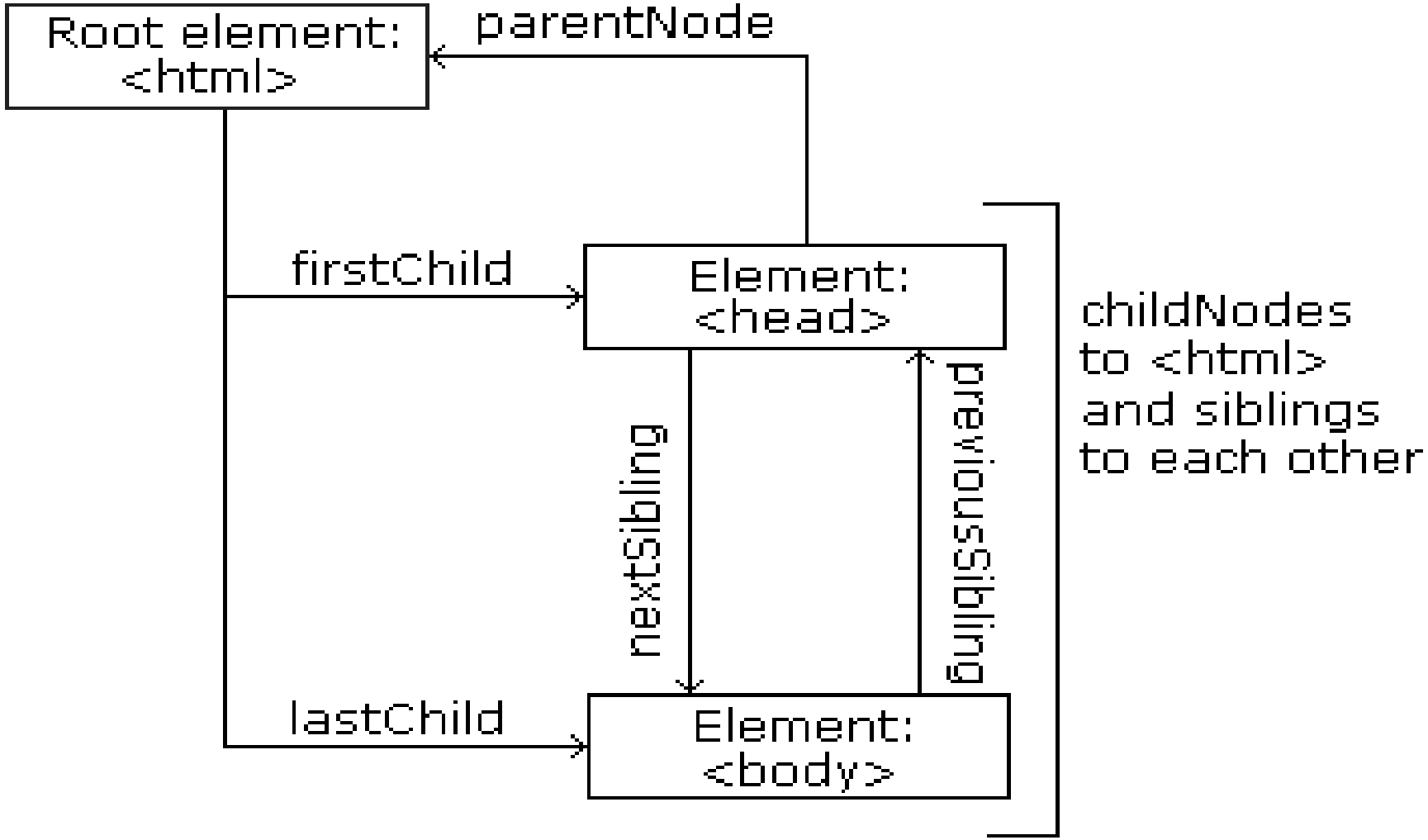
Tasks

Write a function that returns how many paragraphs are contained in a element given as a argument using the DOM API

Write a function that does the same using the Selectors API

Write a function that does the same as the above accepting a argument which defines which API to be used for obtaining the elements.

Relationships between the nodes in the DOM



Manipulating the DOM

Since the DOM is a tree you can do with it everything you can do with a tree :

- Creating new elements;
- Changing new or existing elements;
- Removing elements;
- Inserting new content;
- Replacing content.

Creating new elements

You have two options if you want to create new DOM element :

1. Using document.createElement method.

```
var p = document.createElement('p');
```

2. Creating the element with an HTML string

```
var p = '<p>Some text</p>';
```

Changing DOM elements

There are two things you would like to change in a DOM element – it's visual appearance (style) or it's content.

1. Changing the style of an element happens that way: you have to get a reference of the node and change its style property, or add/remove/change it's class property with predefined CSS class.

2. You can change the content of an HTML element, like changing it's text or it's HTML content

Changing appearance of a element

Modifying element by changing it's style property

```
var p = document.getElementById('myParagraph');
p.style.backgroundColor = 'red';
p.style.color = 'white';
```

Modifying element by changing it's className property

```
<style>
    .my-class {backgroud-color: 'red'; color: 'white'}
</style>
<script type="text/javascript">
    var p = document.getElementById('myParagraph');
    p.className = 'my-class';
</script>
```

Changing the content of a DOM element

You can change the text of a node

```
var p = document.getElementById('myParagraph');
p.innerText = 'My paragraph inner text';
```

Or you can change the HTML content of the node

```
var p = document.getElementById('myParagraph');
p.innerHTML= 'My <strong>paragraph</strong> inner <em>html</em>';
```

Removing elements from DOM

That's how you can remove elements from DOM

```
var p = document.getElementById('myParagraph');
p.parentNode.removeChild(p);
```

If p is a single child of its parent you can do the following

```
var p = document.getElementById('myParagraph');
p.parentNode.innerHTML = '';
```

Inserting element in the DOM Tree

You have two ways of doing this, first one , more complicated , but more flexible is to create a new node and insert it with JavaScript somewhere in the DOM. With this method you have the ability to decide where exactly to put the node.

The second is much more easy, but you don't have much choice and flexibility.

```
var p = document.getElementById('myParagraph');
var newParagraph = document.createElement('p');
newParagraph.innerText = 'I am the new pragraph';

//insert the new node after our paragraph
p.parentNode.appendChild(newParagraph);

//insert the new node before p
p.parentNode.insertBefore(p, newParagraph);
```

Inserting element in the DOM Tree with innerHTML

The secont way to add new content to a DOM node is to manipulate its innerHTML property.

```
var p = document.getElementById('myParagraph');

//appending content
p.parentNode.innerHTML += '<p>I am new pragraph</p>';

// prepending content
p.parentNode.innerHTML = '<p>I am new pragraph</p>'  +
p.parentNode.innerHTML;
```

Replacing DOM content

Again, you have two options – with DOM API methods or using the innerHTML property of the element.

```
var p = document.getElementById('myParagraph');

//using innerHTML
p.parentNode.innerHTML = '<p>I am new pragraph</p>';

// using DOM API
var myP = document.createElement('p');
myP.innerText = 'I am new pragraph';
p.parentNode.replaceChild(p. myP);
```

Tasks

With the given HTML structure:

```
<div></div>
<div></div>
<div></div>
```

In the middle div create unordered list containing 5 elements with content ‘Item 1’, 2 and so on

Create a function which should be able to add a new item on specified position in the unordered list

Create a function which removes item from specified position in the list

Create a function which changes the background color to green and the font color to red for an item in the unordered list with a specified position.