EECS1012

Net-centric Introduction to Computing

Lecture
JavaScript DOM

Acknowledgements
Contents are adapted from web lectures for “Web Programming Step by Step”, by M. Stepp, J. Miller, and V. Kirst.
Slides have been ported to PPT by Dr. Xenia Mountrouidou.
These slides have been edited for EECS1012, York University.
The contents of these slides may be modified and redistributed, please give appropriate credit.
(Creative Commons) Michael S. Brown, 2017.

A bit more on JavaScript

- Local vs. Global Variables
- Other types of popup boxes
- Null and undefined
- parseInt() and parseFloat()
- Style object
- Accessing elements data

Local scope variables

```
function myFunction {
   var a = 10;                    /* these variables are only valid inside this
   var b = 20;                       function.   we say their scope is limited
   var c = a  * b;                   to this function.  We call this local scope. */
}
function myFunction2 {
    var a = 100;                 /* variable a here is different than the
                                    the variable a function above.  Variable b and c are
                                    undefined in this function.  */
}
```

- Variables can have two types of "scope".
- **Local** scope means a variable is only available inside a function
- **A variable defined inside a function is only valid inside that function (see above).**

Global variables

```
/* Code runs as soon as it is loaded – these variables are created before any events*/
var className = "EECS1012";  /* This is a global variable */
var intCounter = 0;                    /* This is another global variable */

function myFunction {
  var Name = document.getElementbyId("Header1");
  Name.innerHTML += className;     /* Global className can be used here */
  intCounter++;                               /* Global intCounter can be used here */
}
```

- JS runs immediately when the file is loaded
- Variables (and out) defined **outside** functions are consider "Global". These variables can be used by **any** function. Their values will be remembered between function calls. See above.

Special values: null and undefined

```
var num1 = null;
var num2 = 9;
if (num3 === undefined)
{
        alert("num3 is not defined");
}
```

- `null` : is a special value that can be assigned to variables to denote it has "no value".
  We often use these to initialize variables.
- `undefined` : is a keyword that denotes that a variable doesn't exist.

null variable

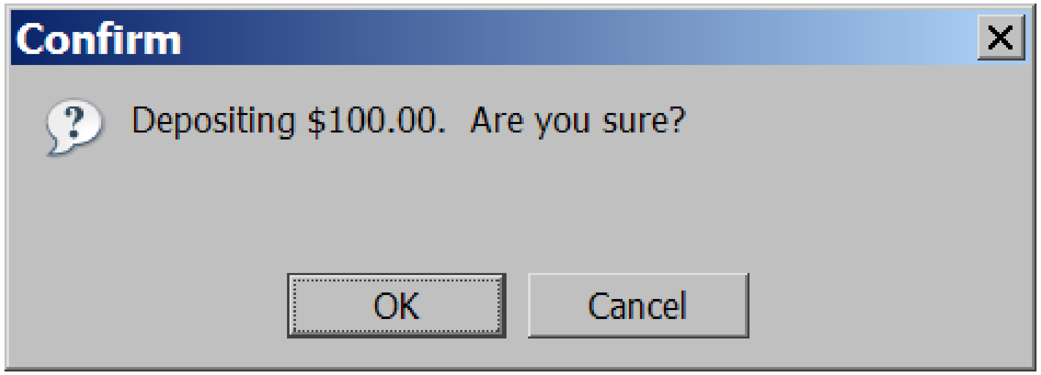
The value **null** can be returned by object methods and functions. For example, if you try to use the document object to get an element that *isn't* in the webpage, the method getElementById() will return a *null*.

```
var p = document.getElementById("myElement");

if (p == null)
{
       alert("There is no element with id=myElement");
}                                                  JS
```

Popup boxes

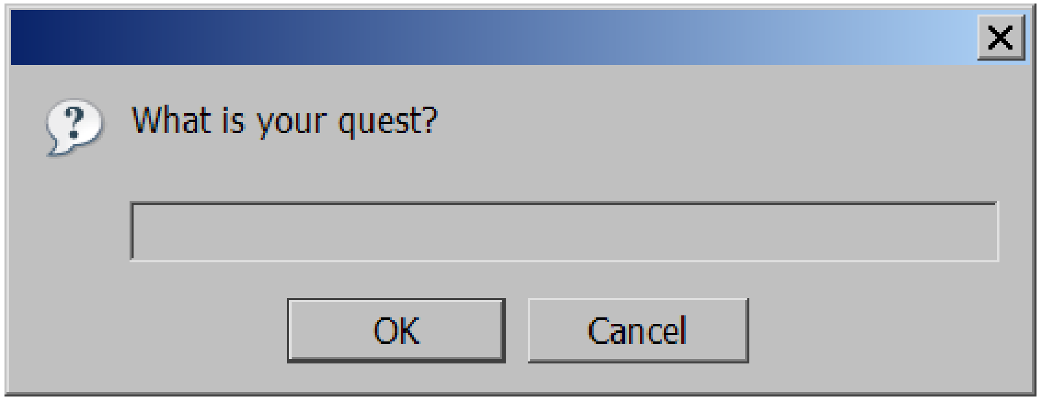
```
var result = confirm("message"); // returns true or false
var inputStr = prompt("message"); // returns an input string
                                                         JS
```

Two more type of popup boxes, in addition to alert("message");



Depositing $100.00. Are you sure?

Confirm has two buttons.
This box returns either a true (ok) or false (cancel).



What is your quest?

Prompt has an input text field.
It returns a string that the user typed in. If "cancel" is pressed, a ***null*** is returned.

Converting strings to numbers

```
var count = 10;
var s1 = "" + count;                        // "10"
var s2 = count + " bananas, ah ah ah!";    // "10 bananas, ah ah ah!"
var n1 = parseInt("42");      // 42 Number
var n2 = parseFloat("3.403");    // 3.403 Number
```

- Last lecture we learned how to convert a number type to a string, but how about a string to a number?
- To convert strings to number we need to use the following:

```
    parseInt()      /* string to Integer*/
    parseFloat()    /* string to Float */
```

parseInt()

- parseInt() is a built-in JavaScript function

```
var num1 = "10";                /* string "10" */
var num2 = "10";                /* string "10" */
var stringAdd = null;
var numAdd = null;

stringAdd = num1 + num2;        /* stringAdd is "1010" */
numAdd = parseInt(num1) + parseInt(num2);   /* numAdd is 20 */

/* parseInt() converts a string to an integer */
```

parseFloat()

- parseFloat() is a built-in JavaScript function

```
var num1 = "10.30";
var num2 = "5.40";
var stringAdd = null;
var numAdd = null;

stringAdd = num1 + num2;      /* stringAdd is "10.30.5.40" */
numAdd = parseFloat(num1) + parseFloat(num2); /* numAdd is 15.70 */

/* parseFloat() converts a string to an integer */
```

Loading multiple JS files

- Like CSS, you can load multiple JS files in your header. Src can point to a URL instead of file.

```
<!DOCTYPE html>
<html>
<head>
  <script src="http://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js"
type="text/javascript"></script>
  <script src="example.js" type="text/javascript"></script>
</head>
<body>
          . . .
</body>
</html>
```

This is a URL to a JS file.
The second is a local JS file.

Style object

Style object

```html
<!DOCTYPE html>
<html>
<head><script src="example.js" type="text/javascript"></script>
</head>
<body>
       <p id="myP"> This paragraph has no style </p>
       <button onclick="myloadFunction();"> Change Style </button>
</body>
</html>
```

```javascript
/* an example of a function named "myFunction" */
function myFunction() {

       var p = document.getElementById("myP");

       p.style.backgroundColor = "blue";/*change BG color to blue*/
       p.style.color = "yellow"; /*change text color to yellow*/

}                                                           JS
```

Accessing style

```
var p = document.getElementById("myP");
```

$$\underbrace{\texttt{p.style.}}_{(1)}\underbrace{\texttt{backgroundColor}}_{(2)}\texttt{ = "blue";}$$

(1) The first . operator access the "style" object
(2) The second . operator, access the backgroundColor property of the style object.

Accessing style alternative (#1)

```
var p = document.getElementById("myP");
var pS = p.style;                 /* 1 */
pS.backgroundColor = "blue"; /* 2 */
```

(1) The var pS is assigned the p variable's style object.
(2) now, pS can directly access the style properties in the style object.

Accessing style alternative (#2)

```
var pS = document.getElementById("myP").style;
```

This directly returns the style object from the document object.

```
pS.backgroundColor = "blue";
```

Variable pS can directly access the style properties in the style object.

Some style object's properties

Style property in JavaScript	Example Values
fontFamily	"monospace", "sans-serif", "serif"
fontSize	"10pt", "125%", "2em"
color	"blue", "black", "red", "yellow"
backgroundColor	"yellow", "blue", "black", "red"

```
/* examples */
var p = document.getElementById("myP");
p.style.backgroundColor = "blue";
p.style.fontFamily = "monospace";
p.style.color = "red";
p.style.fontSize = "125%";
```

Even more style properties you can change --
https://www.w3schools.com/jsref/dom_obj_style.asp

Accessing element's data

- Input field's "value"
- Image's source

Accessing values in an input field

add.html

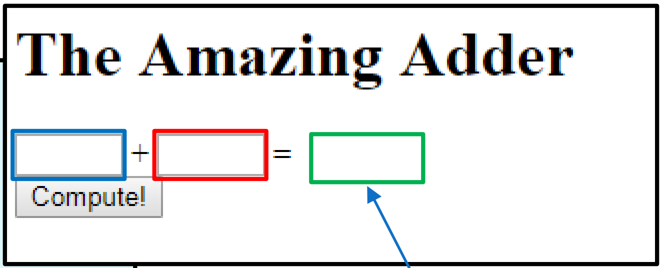
```
<html>
<head>  <script src="add.js" type="text/javascript"></script> </head>
<body>
<h1>The Amazing Adder</h1>
<div>
            <input id="num1" type="text" size="3"> +
            <input id="num2" type="text" size="3"> =
            <span id="answer"></span> <br>
            <button onclick="compute();">Compute!</button>
</div>
</body>
</html>
```

add.js

```
function compute() {
  var input1 = document.getElementById("num1");
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = input1.value + input2.value;
   answer.innerHTML = result;
}
```

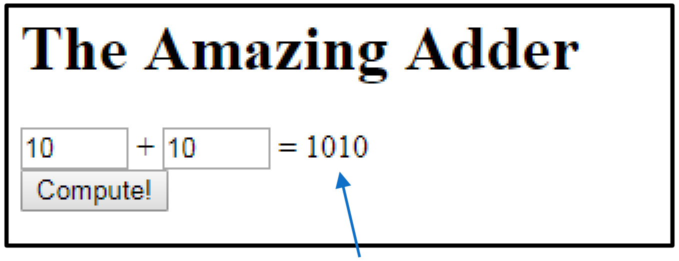
Accessing values in an input field

```
<input id="num1" type="text" size="3"> +
<input id="num2" type="text" size="3"> =
<span id="answer"></span> <br>
<button onclick="compute();">Compute!</button>
```



There is an empty span here with id=answer

```
function compute() {
  var input1 = document.getElementById("num1");
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = input1.value + input2.value;
   answer.innerHTML = result;
}
```

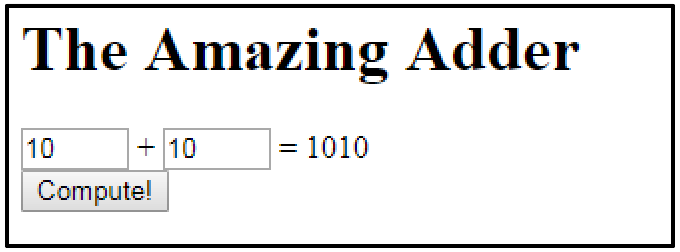


Placed the results in the span elements "innerHTML"

The uses the DOM to get the elements. input1 and input2 are <input> so to access their data we use ".value". answer is a <span> so to change its value we use "innerHTML"

But wait?

```
function compute() {
  var input1 = document.getElementById("num1");
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = input1.value + input2.value;
   answer.innerHTML = result;
}
```



The Amazing Adder

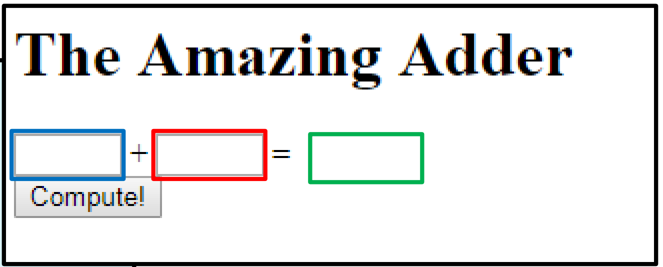
10 + 10 = 1010

Why is the answer "1010"? Because input1.value="10" and input2.value="10", are strings. So, "10" + "10" in JavaScript is "1010", because + is the concatenation operator.

If we want to convert these, we need to use the function parseInt( );

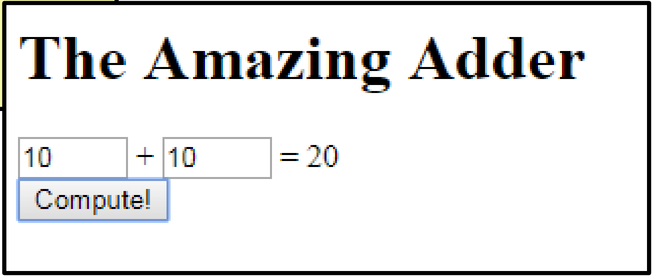
Accessing values (version 2)

```
<input id="num1" type="text" size="3"> +
<input id="num2" type="text" size="3"> =
<span id="answer"></span> <br>
<button onclick="compute();">Compute!</button>
```



```
function compute() {
  var input1 = document.getElementById("num1");
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = parseInt(input1.value) + parseInt(input2.value);
   answer.innerHTML = result;
}
```

We only need to modify the JS code.



Accessing image src

food.html

```
<!DOCTYPE html>
<html>
<head>
  <script src="example3.js" type="text/javascript"></script>
</head>
<body>
  <p>  Click image to see some of my favorite foods:
 <img onclick="changeImage();" src="dosa.jpg" height="100" id="food"></p>
</body>
</html>
```

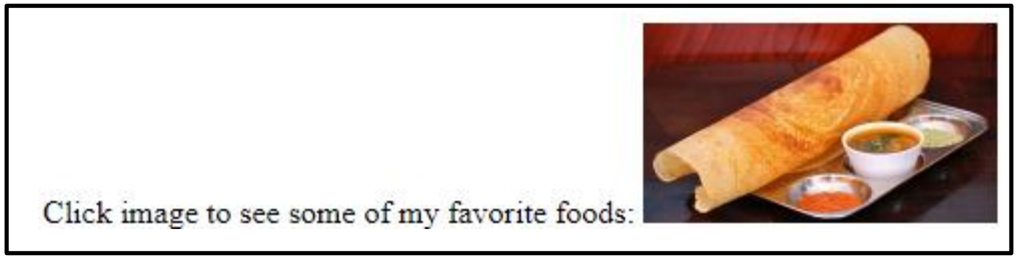
food.js

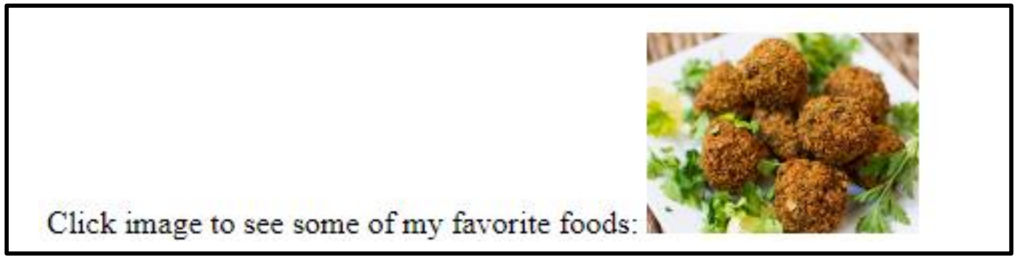
```
var i = 1;
function changeImage() {
        var foodimage = document.getElementById("food");
        var images = ["dosa.jpg", "falafel.jpg", "pide.jpg", "malaxiangguo.jpg"];
        foodimage.src = images[i];
        i++;
        if (i > 3)  { i = 0; }
}
```

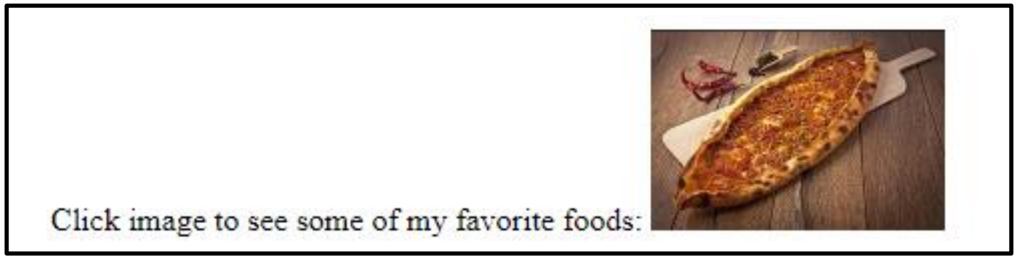
Previous JS example

- Attaches code to an "onclick" event for an image!
  - Events do not have to be only for buttons!
- When the image is clicked, the code gets the image's element source and changes the images source
- An array of four images is used to store the image names
- A "global" variable is declared (outside the function) that keeps its value between function calls

Result of previous code


Click image to see some of my favorite foods:


Click image to see some of my favorite foods:


Click image to see some of my favorite foods:

Each time you click the image, the image changes!

Global DOM objects

Global DOM objects

- Web browser provides six global objects that you can access in your JavaScript code
- These objects can be used to control the browsers and get information about the current webpage (and history)

The six global DOM objects

name	**description**
document	**current HTML page and its content**
history	list of pages the user has visited
location	URL of the current HTML page
navigator	info about the web browser you are using
screen	info about the screen area occupied by the browser
window	the browser window

In our class, we will examine four of these: **document**, **location**, **screen**, and **window**.

The `location` object

- *the URL of the current web page address*
- properties:
  - `location, hostname, href, pathname, port, protocol, search`
- methods:
  - `assign(), reload(), replace()`

Ex: location object

```
<!DOCTYPE html>
<html>
<head>
  <script src="example.js" type="text/javascript"></script>
</head>
<body>
       <button onclick="myloadFunction();"> Refresh Page </button>
</body>
</html>
```

```
/* an example of a function named "myFunction" */
function myFunction() {
       location.reload();// calls location object's reload page
       // this will refresh the page
}                                                           JS
```

The `screen` object

- *information about the client's display screen*
- properties:
  - `availHeight, availWidth, colorDepth, height, pixelDepth, width`

Ex: screen object

```
<!DOCTYPE html>
<html>
<head>
  <script src="example.js" type="text/javascript"></script>
</head>
<body>
       <button onclick="myloadFunction();"> Refresh Page </button>
</body>
</html>
```

```
/* an example of a function named "myFunction" */
function myFunction() {
       var h = screen.availWidth;
       var w = screen.availWidth;
       alert("This device has " + w + " by " + h + " pixels ");
}                                                          JS
```

The `window` object

- *the entire browser window; the top-level object in DOM hierarchy*
- technically, all global code and variables become part of the window object properties:
  - document, history, location, name
- important method
  - `onload()`
    - This method is called when the entire HTML document has completed loading
- We will see examples of this later in this lecture

Unobtrusive JavaScript

Obtrusive event handlers

```html
<button id="ok" onclick="okayClick();">OK</button>
```

HTML

We directly link "onlick" to our JS function "okayClick()". This is considered "obtrusive".

```js
// called when OK button is clicked
function okayClick() {
	alert("booyah");
}
```

JS

- Last lecture example was bad style (HTML is cluttered with JS code)
- This is similar to "inline" CSS style.
- **GOAL**: remove all JavaScript code from the HTML body

Why unobtrusive JavaScript?

- Why do we want unobtrusive JS Code?
- allows separation of web site into three major categories:
  - content (HTML) - what is it?
  - presentation (CSS) - how does it look?
  - behavior (JavaScript) - how does it respond to user interaction?

Attaching an event handler using JavaScript code

```
// where element is a DOM element object
element.event = function;                                         JS
```

```
/* Example */
var button = document.getElementById("ok");
button.onclick = okayClick; /* <- LOOK: no () after func name*/
```

- It is possible to attach event handlers to elements' objects in your JavaScript code
  - notice that you do not put parentheses after the function's name! (see above)
- this is better style than attaching them in the HTML
- **QUESTION:** where should we put the above code?

When does JS code run?

```
<head>
<script src="myfile.js" type="text/javascript"></script>
</head>
<body> ... </body>                                          HTML
```

```
var x = 3;
function f(n) { return n + 1; }
function g(n) { return n - 1; }
x = f(x);    /* f() is called. x now is assigned 4, before the
webpage body has started rendering. */
```

- Your file's JS code runs the moment the browser loads the script
  - any variables are declared immediately
  - any functions are declared but not called, unless your global code explicitly calls them

When does my code run?

```html
<head>
<script src="myfile.js" type="text/javascript"></script>
</head>
<body> ... </body>
```
HTML

```javascript
// global code - start running as soon as it is linked
var x = 3;
function f(n) { return n + 1; }
function g(n) { return n - 1; }
x = f(x);
```
JS

- **at this point in time, the browser has not yet read your page's body**
  - none of the elements in your webpage have been created when the JS file is load.

A failed attempt at being unobtrusive

```
<html>
<head>
<script src="myfile.js"></script>
</head>
<body>
<p><button id="ok">OK</button><p>
</body>
</html>
```

<script> element runs the myfile.js code.

myfile.js

```
// global code
var button = document.getDocumentbyId("ok");
button.onclick = okayClick;
```

The following code is not in a function. It is global code. It tries to use document.getDocumentbyId() to get the button "ok", however, the HTML code hasn't even started on the <body> tag yet. So, there isn't a <button id="ok"> declared. This code will not work.

Solution: `window.onload` event

```
// this will run once the page has finished loading
function functionName() {
        element.event = functionName1;
        element.event = functionName2;
         ...
}
window.onload = functionName; // global code
                                                              JS
```

- we attach our event handlers right after the page is done loading
  - there is a global event called `window.onload` event that occurs at that moment
- in `window.onload` handler we attach all the other handlers to run when events occur

An unobtrusive event handler

```html
<!– look, no JavaScript! -->
<button id="ok">OK</button>
```

HTML

```javascript
// called when page loads; sets up event handlers
function pageLoad() {
       var button = document.getElementById("ok");
       button.onclick = okayClick;
}

function okayClick() {
       alert("booyah");
}

window.onload = pageLoad; // global code
```

JS

In this example, we assign the event "okayClick" using JS code, instead of the HTML page. This is considered unobtrusive.

Common unobtrusive JS errors

```
~~window.onload = pageLoad();~~ /* remember - don't put the () */
window.onload = pageLoad;
~~okButton.onclick = okayClick();~~
okButton.onclick = okayClick;
                                                     JS
```

- Remember, when we assign in the names of function, don't use the (), only the function name.

```
~~window.onLoad = pageLoad;~~     /* <- the L is not capital */
window.onload = pageLoad;
                                                            JS
```

- also, event names are all lowercase, not capitalized like other variables

Anonymous functions

```
/* look, no name! – we call this an anonymous function */
function() {
        statements;
}
                                                       JS
```

- JavaScript allows you to declare anonymous functions
- quickly creates a function without giving it a name
- can be stored as a variable, attached as an event handler, etc.

Anonymous function example

```
/* the example below is an anonymous function, notice there is
no name given to this function.  However, the
function is only called once when the "window.onload" event
occurs, so it is OK we don't give it name */

window.onload = function() {
       var okButton = document.getElementById("ok");
       okButton.onclick = okayClick;
};
function okayClick() {
       alert("booyah");
}
                                                          JS
```

We set window.onload = to an anonymous function.

Revisit style example

The HTML code below has an explicit link to "onclick" in it (i.e. obtrusive JS)

```
<p id="myP"> This paragraph has no style </p>
<p> <button onclick="myloadFunction();"> Change Style </button> </p>
```

This example is considered obtrusive, because we had to define the *onlick* event in the HTML. This means the HTML page needs to know about the JS file and function names, etc.

```
function myFunction() {

       var p = document.getElementById("myP");

       p.style.backgroundColor = "blue";/*change BG color to blue*/
       p.style.color = "yellow"; /*change text color to yellow*/

}
```

JS

Version2 (unobtrusive)

```
<p id="myP"> This paragraph has no style </p>
<p> <button id="myButton""> Change Style </button> </p>
```

```
function pageLoad() {
 var b  = document.getElementById("myButton");
 b.onclick = myFunction;
}

function myFunction() {

       var p = document.getElementById("myP");

       p.style.backgroundColor = "blue";/*change BG color to blue*/
       p.style.color = "yellow"; /*change text color to yellow*/

}

window.onload = pageLoad;
```

HTML code now has **no** JavaScript code for "onclick".

window.onload event is set to call function pageLoad().

pageLoad sets the buttons onclick to be myFunction()

Version 3 – with anonymous function

```
<p id="myP"> This paragraph has no style </p>
<p> <button id="myButton""> Change Style </button> </p>
```

```
window.onload = function() {
 var b  = document.getElementById("myButton");
 b.onclick = myFunction;
}

function myFunction() {

       var p = document.getElementById("myP");

       p.style.backgroundColor = "blue";/*change BG color to blue*/
       p.style.color = "yellow"; /*change text color to yellow*/

}
```

JS

HTML code now has no JavaScript code for "onclick".

window.onload event is set to an anonymous function that sets the onclick for the button.

Recap – unobtrusive JS

- Unobtrusive JS is general approach to JS programming that tries to avoid having JS code inside the HTML page
- When possible, try to write JS code in an unobtrusive manner
- Use the "windows.onload" event to assign all the event handlers

The DOM tree

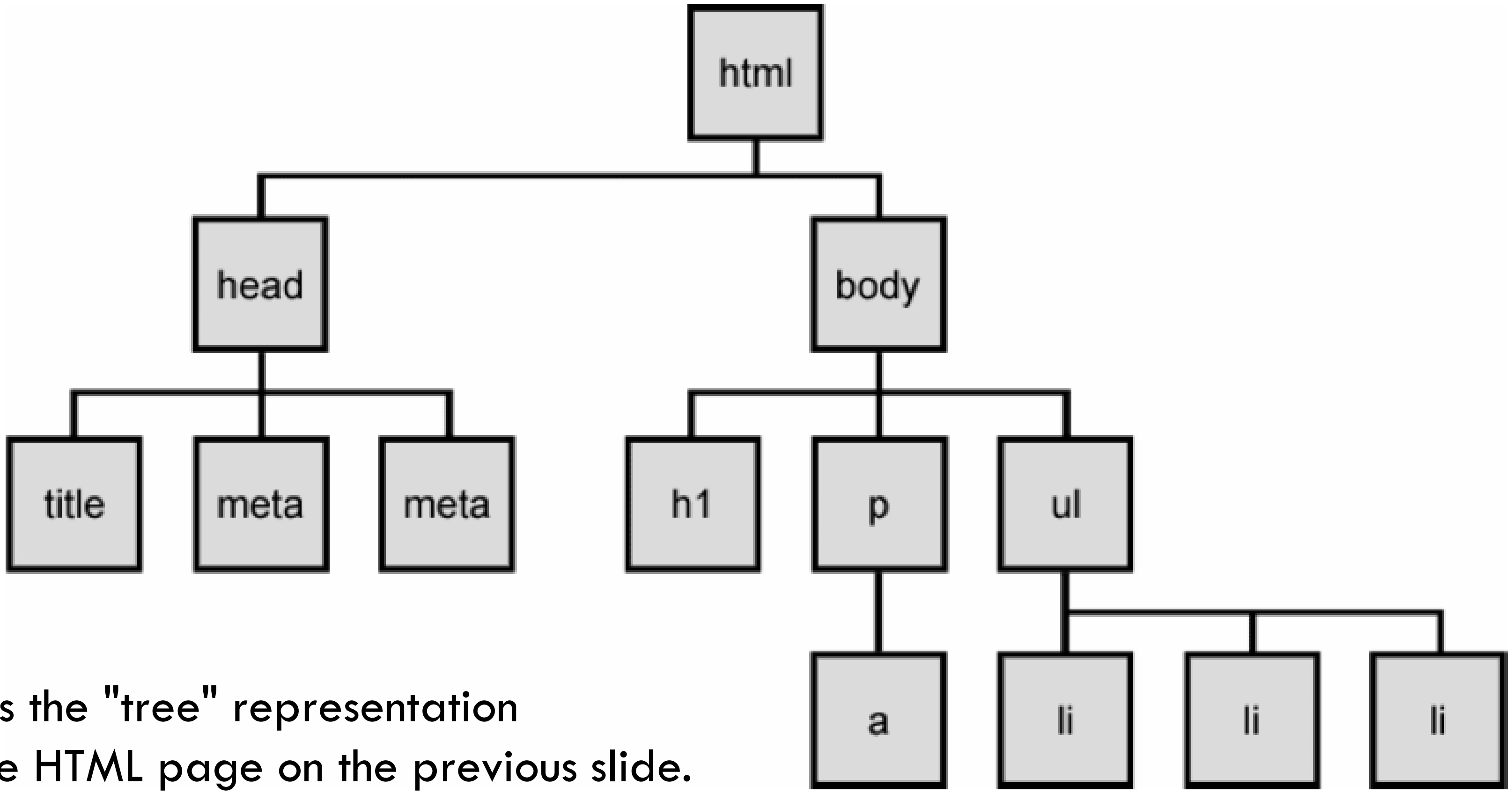
The document object model

- The DOM models an HTML page and its elements as a "tree" structure
- A tree is a type of "data structure" common in computer science to organize data
- Consider the next slide's HTML code

Consider the following HTML page

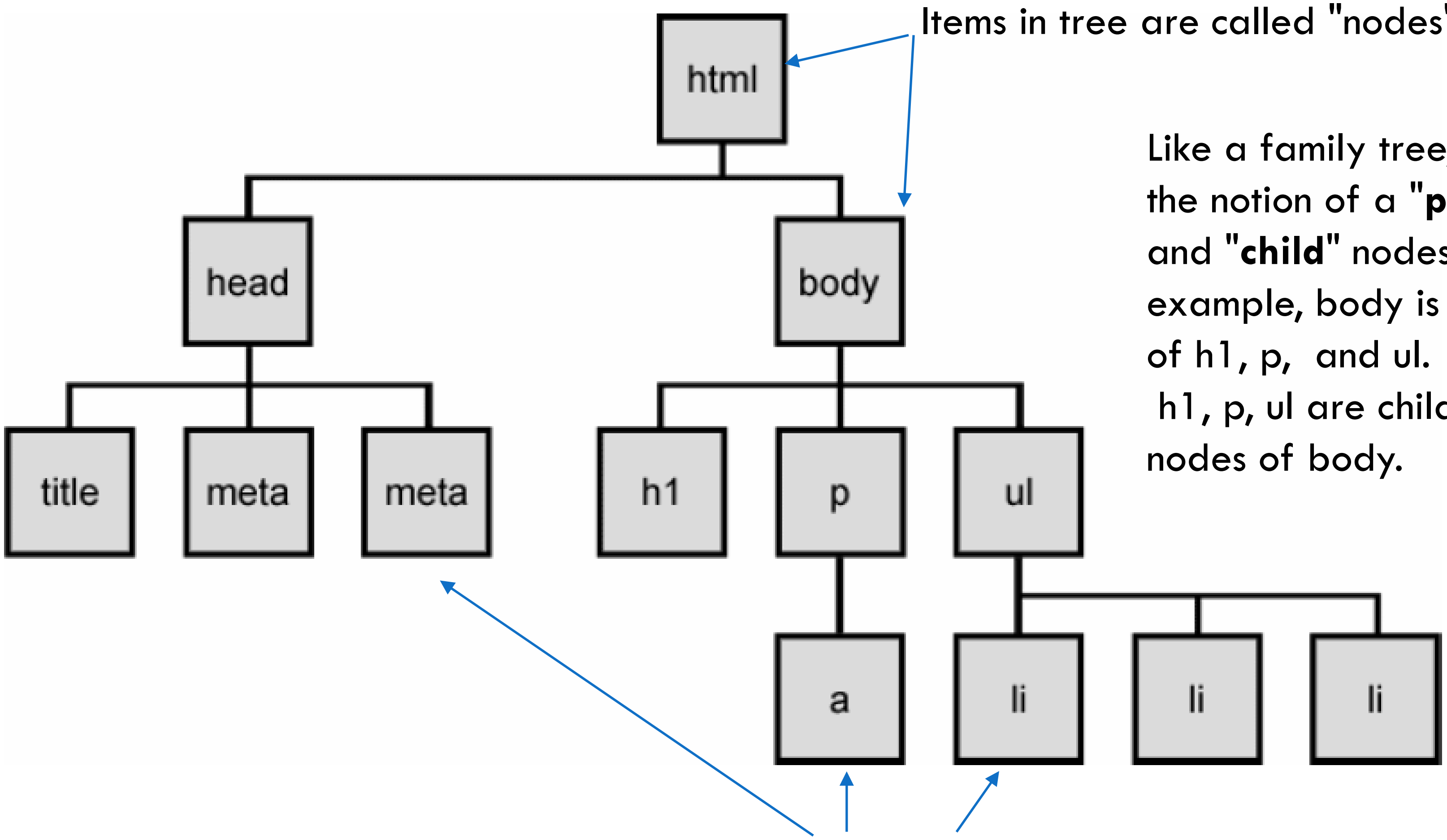
```
<!DOCTYPE html>
<html>
<head>
  <title> Page Title </title>
   <meta name="description" content="A really great web site">
   <meta charset="UTF-8">
</head>
<body>
 <h1> This is a heading </h1>
 <p> A paragraph with a <a href=http://www.google.com/> link </a>.</p>
 <ul>
    <li>a list item</li>
    <li>another item</li>
    <li>a third item</li>
 </ul>
</body>
</html>
```

The DOM tree of the HTML code



This is the "tree" representation
of the HTML page on the previous slide.

Tree terminology

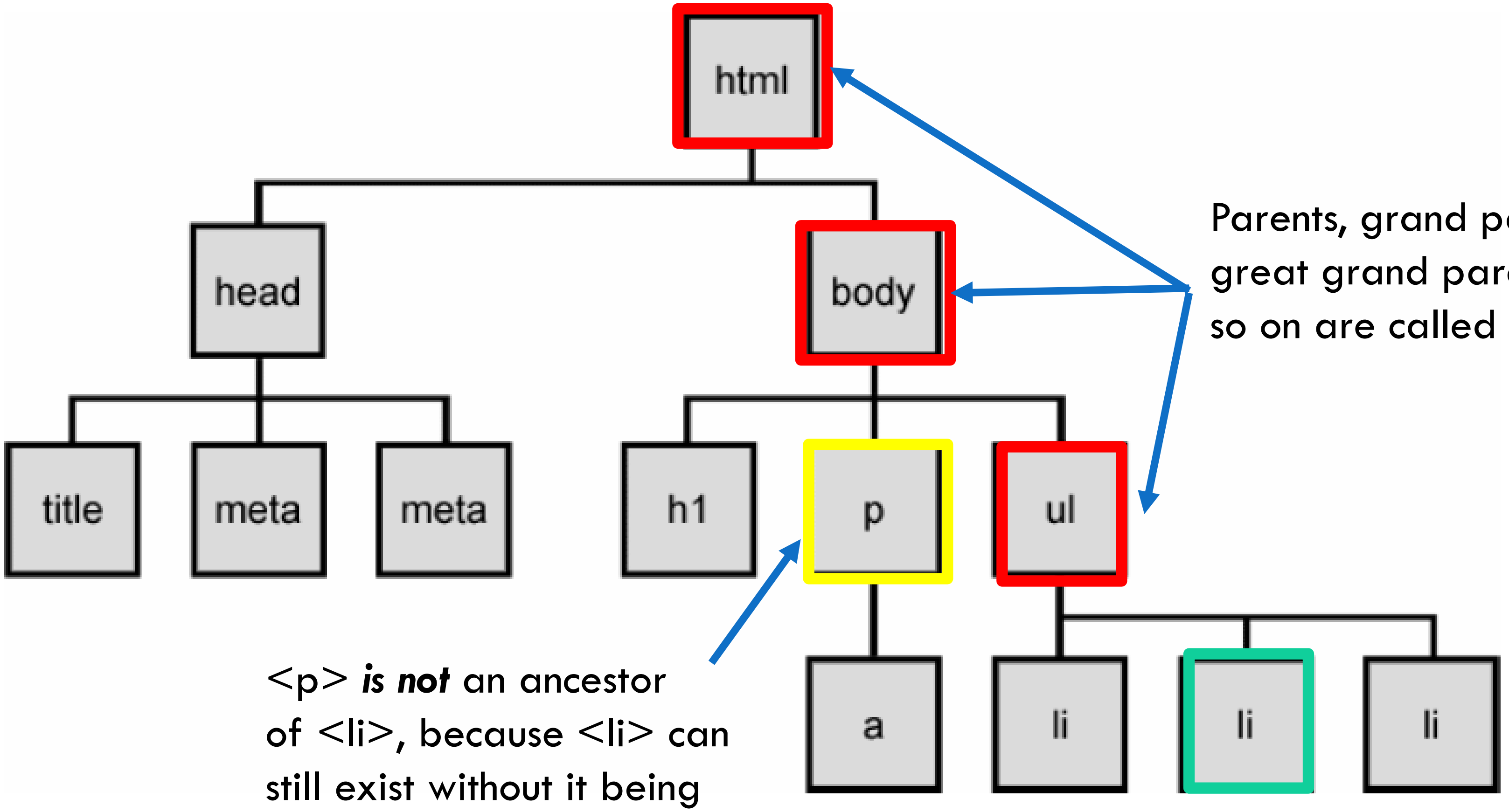


Items in tree are called "nodes".

Like a family tree, we have the notion of a "**parent**" and "**child**" nodes. For example, body is the parent of h1, p, and ul.
h1, p, ul are child nodes of body.

The nodes at the end of the tree are called "leafs"

More tree terminology



Parents, grand parents, great grand parents, and so on are called "Ancestors".

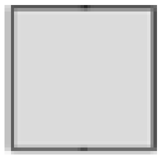
<p> ***is not*** an ancestor of <li>, because <li> can still exist without it being created.

All nodes in shown in "red" are ancestors of the <li> element in green.

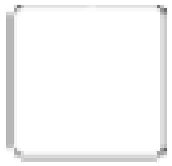
Types of DOM nodes

```
<p>
This is a paragraph of text with a
<a href="/path/page.html">link in it</a>.
</p>                                              HTML
```

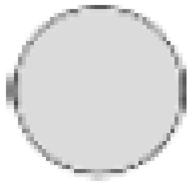
- element nodes (HTML tag)
  - can have children and/or attributes



- text nodes (text in a block element)



- attribute nodes (attribute/value pair)
  - text/attributes are children in an element node
  - cannot have children or attributes
  - not usually shown when drawing the DOM tree

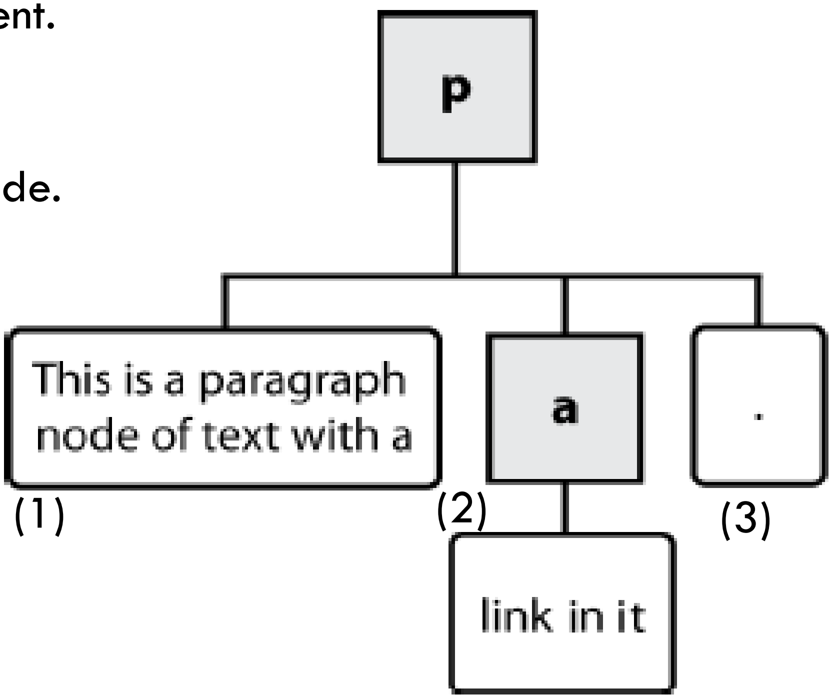


Types of DOM nodes

```
<p>
This is a paragraph of text with a
<a href="/path/page.html">link in it</a>.
</p>
```
HTML

Consider the DOM of a node.
This is a "mini-tree" considering on the <p> element.

It has three direct children: (1) a text node,
(2) another element (link), and (3) another text node.



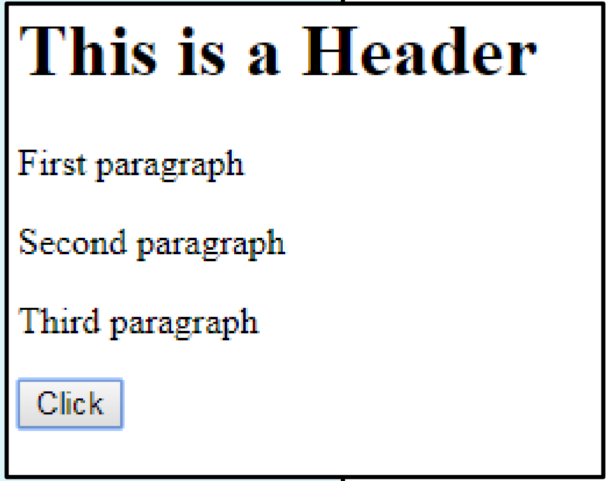
Traversing the DOM tree

name(s)	**description**
firstChild, lastChild	start/end of this node's list of children
***children**	array of all this node's children
nextSibling, previousSibling	neighboring nodes with the same parent
parentNode	the element that contains this node

We will do an example with the *children property.

DOM tree traversal example

```
<html>
<head>
    <title> My page</title>
    <meta charset="utf-8">
    <script src="dom_example1.js"
type="text/javascript"></script>
</head>
<body>
<h1>This is a Header</h1>
<div id="mydiv">
<p> First paragraph </p>
<p> Second paragraph </p>
<p> Third paragraph </p>
</div>
<button id="button"> Click </button>
</body>
</html>
```



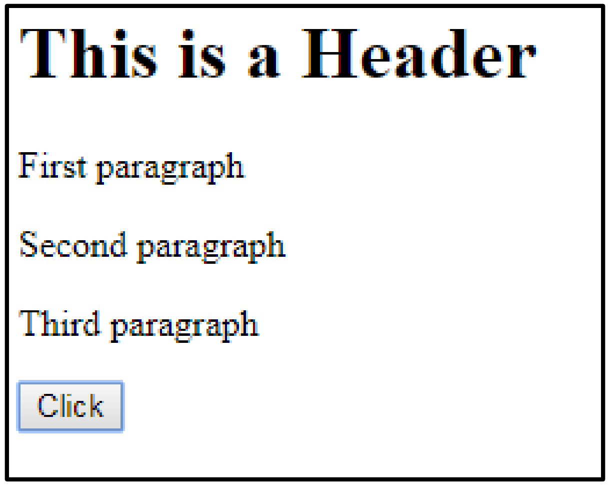
This is a Header

First paragraph

Second paragraph

Third paragraph

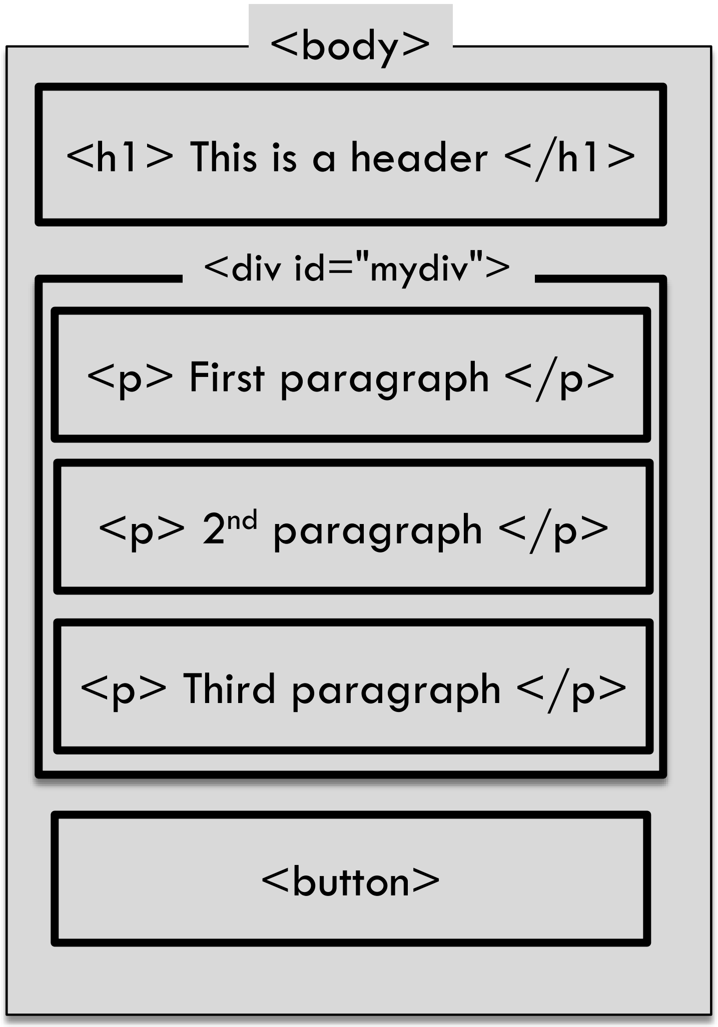
DOM tree traversal example


This is a Header

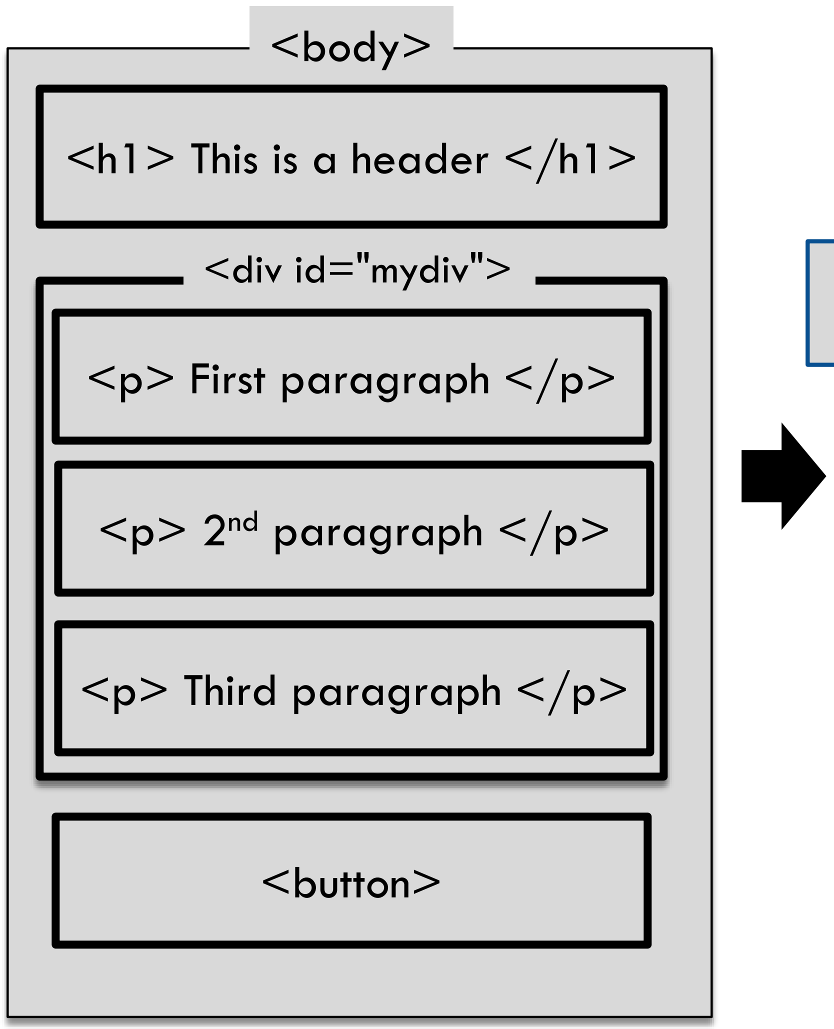
First paragraph

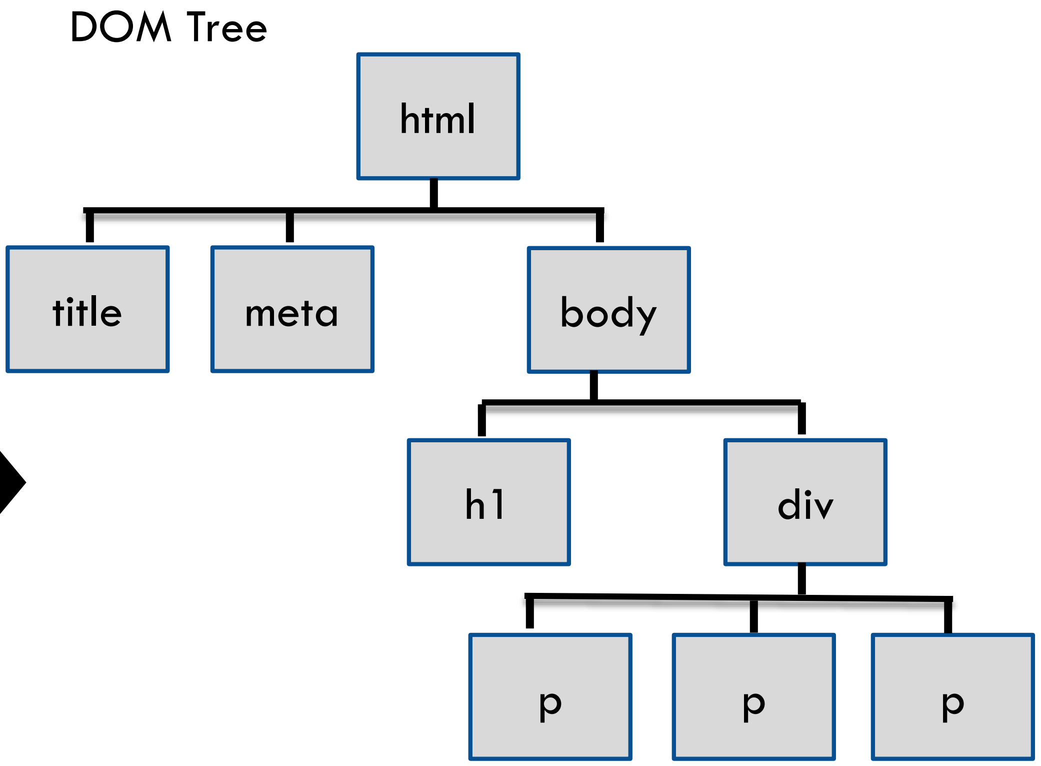
Second paragraph

Third paragraph



DOM tree traversal example





This is drawn without the "text" elements, but each "p" has a text element that we can access using "innerHTML".

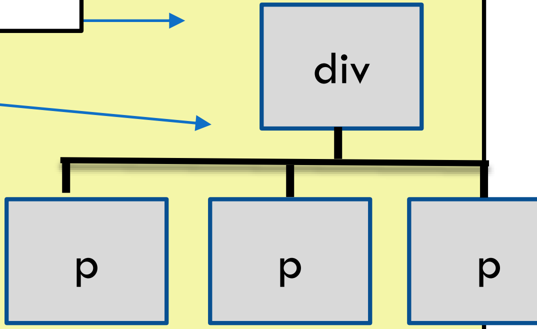
DOM tree traversal example

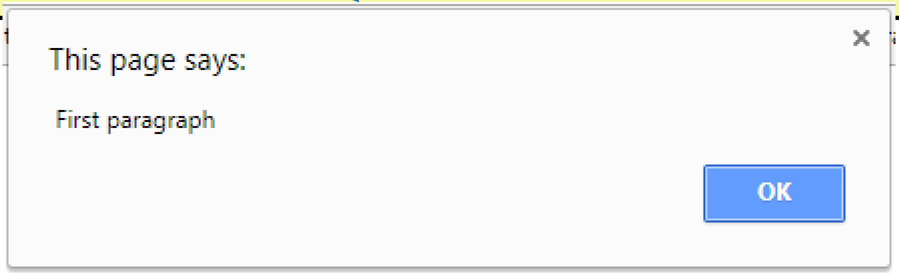
```
function walk()
{
  var divElement = document.getElementById("mydiv");
  var childElements = divElement.children;

  alert("mydiv element has " + childElements.length + " children");
  for(var i=0; i < childElements.length; i++)
  {
    alert(childElements[i].innerHTML);
  }
}
```

Gets the div element

returns an array with the children of div

Loops through the array and prints (using alert) the innerHTML of the paragraph

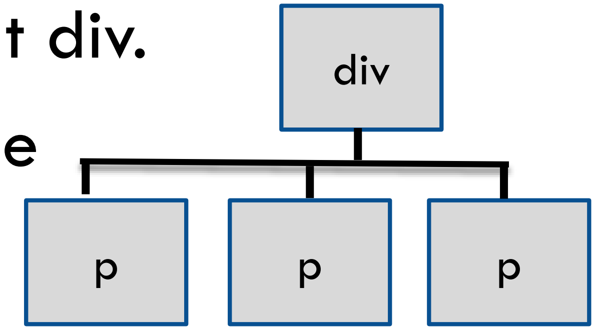



This page says:

First paragraph

Note: tree access

- In the previous example, since we started with
  `var divElement = document.getElementById("mydiv");`
- Our access to the DOM tree starts at div.
- As a result, we only had access to the "descendants" of div, not the full DOM tree



- You will see this word "descendants" in JS documentation

Selecting groups of DOM objects

- methods in document and other DOM objects for accessing descendants:

name	**description**
getElementsByTagName(name)	returns array of descendants with the given tag, such as "div"
getElementsByClassName(name)	returns array of descendants with the specified class name.

Getting all elements of a certain type

```
// all Paras is an array of element objects
var allParas = document.getElementsByTagName("p");
// we loop trough all elements and change their background
// to "yellow"
for (var i = 0; i < allParas.length; i++) {
       allParas[i].style.backgroundColor = "yellow";
}                                                       JS
```

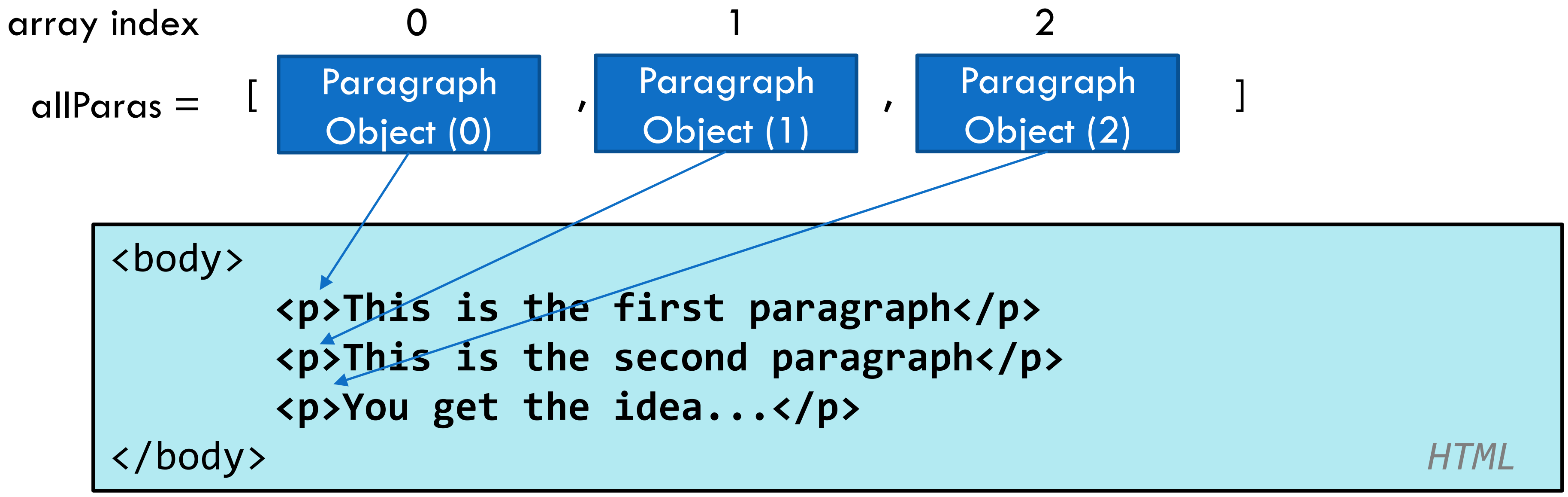
```
<body>
       <p>This is the first paragraph</p>
       <p>This is the second paragraph</p>
       <p>You get the idea...</p>
</body>                                                    HTML
```

In tis example, we use the document object, so the descendants are all elements in the HTML page with tag name "p". This results

Previous code explained

```
var allParas = document.getElementsByTagName("p");
```

This will find all the DOM element objects that are <p> elements in the HTML page and retur
them as an array. This array is assigned to the variable "allParas".



```
<body>
        <p>This is the first paragraph</p>
        <p>This is the second paragraph</p>
        <p>You get the idea...</p>
</body>
```

Previous code explained

Paragraph Object [i]

```
allParas[i].style.backgroundColor = "yellow";
```

allParas is an array of element objects.

.style accesses the style component of the element object.

.backgroundColor accesses the backgroundColor property of the style component.

Changes the background to yellow.

allParas[i] access the ith element in the array.

So, this statement is accessing a single element object. The element accessed depends on the value of the variable i.

```
<body>
        <p>This is the first paragraph</p>
        <p>This is the second paragraph</p>
        <p>You get the idea...</p>
</body>
```

Combining with getElementById()

```
var addressDiv = document.getElementById("address");
var addrParas = addressDiv.getElementsByTagName("p");
for (var i = 0; i < addrParas.length; i++) {
       addrParas[i].style.backgroundColor = "yellow";
}                                                     JS
```

```
<p>This won't be returned!</p>
<div id="address">
       <p>1234 Street</p>
       <p>Atlanta, GA</p>
</div>                                                  HTML
```

In this example, only the paragraphs contained **within** "addressDiv" are called. This is because "getElementsByTagName("p") is called from the div element.

Creating and Deleting Elements

Creating new nodes

name	description
document.createElement("tag")	creates and returns a new empty DOM node representing an element of that type
document.createTextNode("text")	creates and returns a text node containing given text

```
// create a new <h2> node
var newHeading = document.createElement("h2");
newHeading.innerHTML = "This is a heading";
newHeading.style.color = "green";
                                                   JS
```

- merely creating a node does not add it to the page
- you must add the new node as a child of an existing element on the page...

Modifying the DOM tree

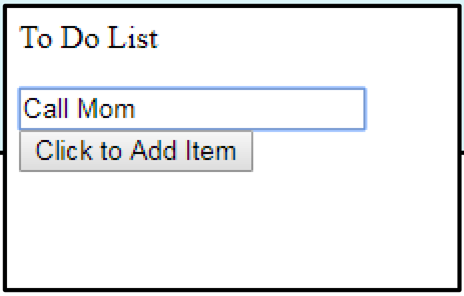
name	description
appendChild (node)	places given node at end of this node's child list
insertBefore(new, old)	places the given new node in this node's child list just before old child
removeChild(node)	removes given node from this node's child list
replaceChild(new, old)	replaces given child with new node

```
var div = document.getElementById("mydiv");
var p = document.createElement("p");
p.innerHTML = "A paragraph!";
div.appendChild(p);    /* append ads
                                                 JS
```

This would add a new paragraph to div with id=mydiv.

Example – adding items

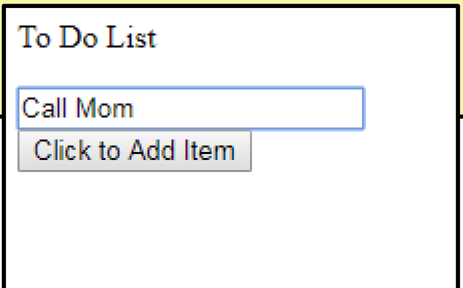
```
<html>
<head>
    <title> My page</title>
    <meta charset="utf-8">
    <script src="dom_example5.js"
type="text/javascript"></script>
</head>
<body>
  <p>To Do List</p>
  <input type="text" id="textToAdd" size="20"><br>
  <button id="button"> Click to Add Item</button>
  <ol id="list">
  </ol>
</body>
```

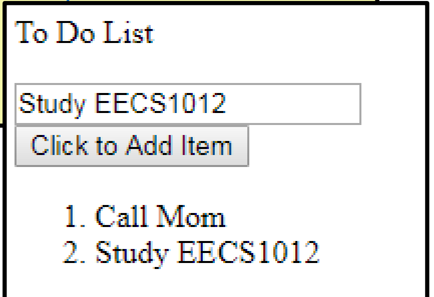

To Do List

Call Mom

Example – adding items

```
window.onload = function() {  /* This finds the button and sets the onclick function */
  var button = document.getElementsByTagName("button");
  button[0].onclick = insertItem;
}

function insertItem()
{
  var todoList = document.getElementById("list"); /* get list element */
  var textToAdd = document.getElementById("textToAdd"); /* get text input element */

  if (textToAdd.value != "")          /* if text input value isn't empty */
  {
    var newLi = document.createElement("li");        /* create a new li element */
    newLi.innerHTML = textToAdd.value;               /* set the innerHTML to the text input value */
    todoList.appendChild(newLi);                     /* add this to the DOM tree */
                                                     /* by appending to the list object */
  }
}
```

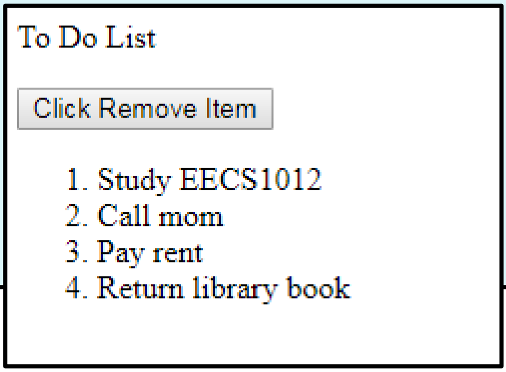




1. Call Mom
2. Study EECS1012

Example – delete items

```
<html>
<head>
   <title> My page</title>
   <meta charset="utf-8">
   <script src="dom_example4.js" type="text/javascript"></script>
</head>
<body>
  <p>To Do List</p>
  <button id="button"> Click Remove Item</button>
  <ol id="mylist">
            <li> Study EECS1012 </li>
   <li> Call mom </li>
   <li> Pay rent </li>
   <li> Return library book </li>
  </ol>
</body>
```

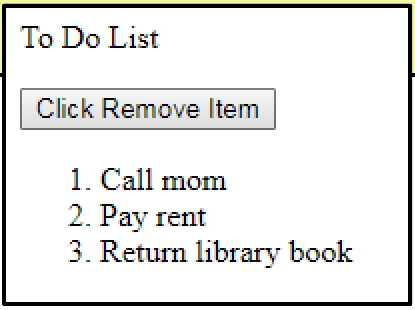


To Do List

1. Study EECS1012
2. Call mom
3. Pay rent
4. Return library book

Example – deleting items

```
window.onload = function() {  /* attaches the deleteListItem function to the button */
  var button = document.getElementsByTagName("button");
  button[0].onclick = deleteListItem;
}

function deleteListItem()
{
  var mylist = document.getElementById("mylist");  /* get list element */
  var pars = mylist.getElementsByTagName("li");    /* get all li elements in the list element */
  if (pars.length > 0)                             /* pars (array of li elements) is not 0 */
  {
    mylist.removeChild(pars[0]);                   /* remove the first li element from the */
  }                                                /* list element */
}
```


To Do List

Click Remove Item

1. Call mom
2. Pay rent
3. Return library book

Prototype Library

Problem with JavasScript

- JavaScript is a powerful language, but the DOM can be clunky to use
- JS is also not standardized
  - The same code doesn't work the same way on some browsers
- as a result, some developers have created a "library" (set of JS functions) call the "Prototype" library
  - There are others . . e.g., JQuery

Prototype Library

```
<script src="https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js"
type="text/javascript"></script>
```

- Prototype JS library adds many useful features to JS
  - Makes DOM easier to use
  - improves event-driven programming (next lecture)
  - works the same across many browsers
- To use the Prototype library, link to it as shown above
- Note this access the JS file as a URL

Prototype framework

```
<script src="
https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototy
pe.js "
type="text/javascript"></script>                              JS
```

- the Prototype JavaScript library adds many useful features to JavaScript:
  - many useful extensions to the DOM
  - added methods to String, Array, Date, Number, Object
  - improves event-driven programming
  - many cross-browser compatibility fixes
  - makes Ajax programming easier (seen later)

The $ function using Prototype

```
$("id")                                   JS
```

- returns the DOM object representing the element with the given id
- short for document.getElementById("id")
- often used to write more concise DOM code:

```
$("footer").innerHTML = $("username").value;
                                               JS
```

Example with prototype

prototype.html

```
<html>
<head>
<script src=" https://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js "
type="text/javascript"></script>
<script src="dom_example6.js" type="text/javascript"></script>
</head>
<body>
<h1>The Amazing Adder</h1>
<div>
        <input id="num1" type="text" size="3"> +
        <input id="num2" type="text" size="3"> =
        <span id="answer"></span> <br>
        <button onclick="compute();">Compute!</button>
</div>
</body>
</html>
```

prototype.js

```
function compute() {
   var num1 = $("num1").value;
   var num2 = $("num2").value;
  $("answer").innerHTML = parseInt( num1 ) + parseInt( num2 );
}
```

Prototype

HTML – (note: this example is obtrusive HTML . . but the purpose of this example is to who the prototype library in JS)

```
<input id="num1" type="text" size="3"> +
<input id="num2" type="text" size="3"> =
<span id="answer"></span> <br>
<button onclick="compute();">Compute!</button>
```

Using the $("element_id") we have much more compact JS code. It is also easier to make the connection back to the HTML page.

```
function compute() {
   var num1 = $("num1").value;
   var num2 = $("num2").value;
   $("answer").innerHTML = parseInt( num1 ) + parseInt( num2 );
}
```

$("id") is equivalent to calling document.getElementbyId("id").

Recap

- The DOM gives JS access to the underlying webpage
- The DOM tree is used to describe the HTML page
- This gives us the ability to modify, create, and delete elements in the tree (i.e. HTML page)
- The prototype library makes accessing document elements "cleaner"
- We will use the prototype library more in the next lecture on "events".