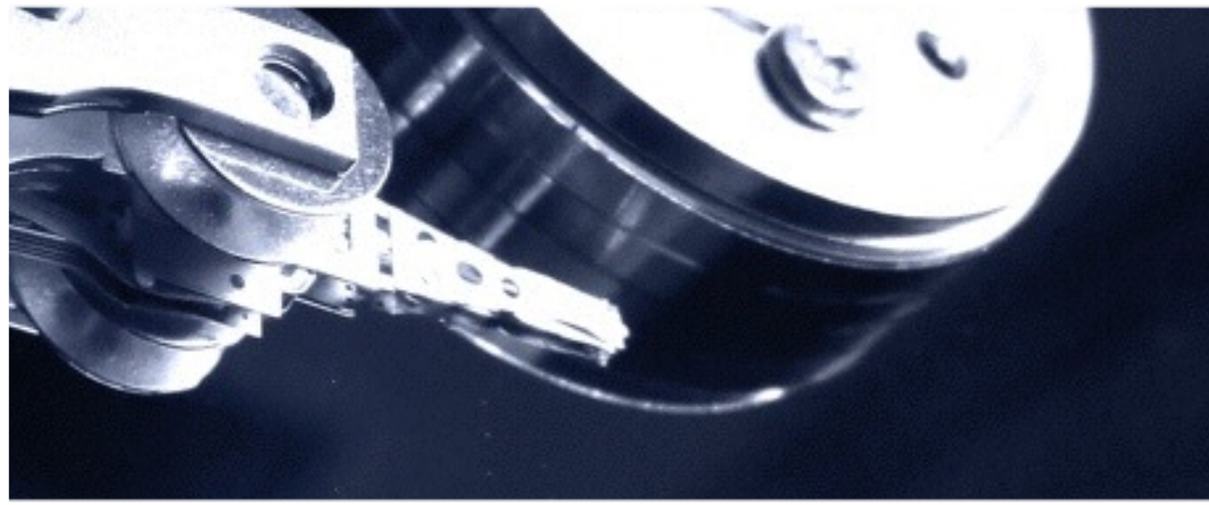
FULL SAIL
UNIVERSITY

web design and development



programming for web applications 1

courseMaterial.4

courseDirector

Fialishia O'Loughlin

foloughlin@fullsail.com

labSpecialist

Eric Silvay

esilvay@fullsail.com

courseMaterial.4

goal3.**Recap**

goal3.Recap

- review debugging assignment
- scope
- closure

courseMaterial4.**Objectives**

- course material
  - *BASIC object overview*
  - *introduction to the Document Object Module (DOM)*
  - *javascript events & callbacks (listeners and handler)*
  - *practice all the new materials*

- assignment
  - *fine tune the concepts from the course materials*

courseMaterial.**4**

basic.**Objects**

basic.**Objects**

- arrays and objects
  - javaScript, as an object-oriented language, was designed to have rich, mutable (i.e. returning an array of strings from a method) *objects*
  - objects are basically collections of **keys and values** for storing data
  - arrays and objects are both used to store multiple values
  - arrays vs objects:
    1. *arrays store by **numerical index***
    2. *objects store by a **key index***
       - ***keys** in objects are names (similar to variables), used to index a value inside the object*

basic.**Objects**

- setting an objects value
  - the most efficient way to create objects is by ***key: value*** pairings, also known as associative pairing
  - similar to making an array, we declare the object's properties inside the literal, separating by comma...

```
person = {"name":"bond", "age":35, "secretAgent":true};
```

key:value

```
person = {name:"bond", age:35, secretAgent:true};
```

key:value

basic.Objects

- setting an objects value (setters)
  - accessing or setting values can be done just like *array access:*

```
var house = {};
house["location"] = 'Winter Park, FL'; //set the key's value
```

key **value**

  - setting the key's value can also be done with *dot syntax:*

```
house.location = 'Winter Park, FL';
```

key **value**

basic.Objects

- nested objects
  - similar to arrays, we can nest objects inside objects

```
person = {birthday:{month:02, day:12}, name:"bond"}; //
setter
```

```
person["birthday"]["month"]; //returns '02' (getter)
```

```
person.birthday.month; //dot notation - returns '02' (getter)
```

basic.**Objects**

- methods and properties

```
var identifier = {};
```

  - objects are useful primarily for **properties** and **methods**:
    - **properties** are *variable/keys* that belong to an object only
    - **methods** are *function/keys* that belong to the object only
    - *these **members** are the foundation of an oop model*

basic.Objects

```
var fsStudent = {};                    //initialize variable
fsStudent.age = 22;                    //set the age property value
fsStudent.career = "Web Dev";          //set the career property
fsStudent.sayHello = function(){       //create a method called
sayHello
    alert("Hi!");
};
```

- in this example, we first initialize the object, then we created 2 properties for the object, and a method called sayHello - notice that the method is being created by an **assignment operator**, just like the properties

```
fsStudent.sayHello();
```

basic.Objects

- we can also access the methods and properties of an object using **[]**, by using their name as a **string** - all of the below are valid:

```
fsStudent.sayHello();                 //dot syntax
fsStudent["sayHello"]();              //index syntax
fsStudent.age;                        //dot syntax
fsStudent["age"];                     //index syntax
```

basic.**Objects**

- new constructor using **{ }**
  - we can also use the **object literal** syntax to create a new object and fill it at the same time
  - the **new *constructor*** is **{ }**.. here's the same object as the previous slide..

```
var fsStudent = {
    age: 22,
    career: "Web Dev",
    sayHello: function(){
        alert("Hi!");
    }
};
```

- in this syntax, **properties** and **methods** are assigned using **key:value** pairings
- similar to an array, each new **key:value** pair is separated with a **comma**

basic.Objects

- accessing object's properties
  - also keep in mind that since the keys can be strings, you could access the keys using string variables

```
var fsStudent = {
    age: 22,
    career: "Web Dev",
};

var myProp = "age";
```

`fsStudent[myProp]` = `fsStudent["age"]`

basic.Objects

- for in loop
  - although JavaScript lacks a *for-each* loop, we do have **for-in**
  - the **for in** statement lets us loop over all of the properties in an object

```
var myObj = {...};

for (var key in myObj) {
    alert( myObj[key] );
};
```

basic.Objects

- for in loop - example

```
var students
={name:"JamesBond",gender:"male",job:"student"};

for(var key in students){
   console.log('Key Name: ', key);
   console.log('Value of the key[',key,']:
',students[key]);
};
```

basic.**Objects**

- data types are all objects
  - most strictly-typed languages have clear separations in their data types and classical behavior
  - JavaScript is simpler:
    - ***numbers***, ***strings***, and ***booleans*** are the only separate data types in JavaScript
    - ***objects***, ***arrays***, ***regular expressions***, and ***functions*** are all considered to be **objects.**
  - in addition, anything that is not an object, *acts* like an object

```
myStr = 'James Bond';
alert(myStr.length)    //will return a 10
```

basic.Objects

- object literals
  - let's re-examine our data types - previously, we used the most common constructor for creating our variables, which is the **literal** syntax:

```
var myNum = 5;
```

  - we could also create this variable using its constructor equivalent:

```
var myNum = new Number(5);
```

  - both have the same result - the literal syntax is preferable

basic.Objects

- object constructors
  - using these constructors, we can also convert from one data type to another

```
var myNum = 1;                    //returns the number 1
myString = String(myNum);         //returns the string "1"
myBool = Boolean(myString);       //returns the boolean true
```

basic.**Objects**

- values as objects
  - because these values *act as* objects, the data types also have **methods** and **properties**, just like an object...
  - we've already looked at one, the **.length** property, which can be used on both *strings* and *arrays*

```
myStr = "Oh yes.";          //sets a string variable
alert(myStr.length);        //returns a 7

myArr = [6, 10];            //sets an array variable
alert(myArr.length);        //returns a 2
```

basic.Objects

- literals are objects
  - because the literals being created are also objects, we can use methods and properties directly on the literals themselves...

```
alert( "bond 007".length );          //returns a 8
```

basic.Objects

- more advance
  - an array of objects, inside of an object

```
var obj1 = {
    schoolName:'Full Sail',
    students:[
    {name:'Jane Doe', GPA:2.6},
        {name:'Albert Einstein', GPA:4.0},
        {name:'James Bond', GPA:3.9}
    ]
};
console.log(obj1.students.length);
```

courseMaterial.4

intro.DOM

intro.DOM

- what is the Document Object Model (DOM)?
  - the DOM is an API that every browser has
  - it is not a specification of javascript
  - the DOM exposes access to all the elements of a web document (including the browser window itself)

intro.**DOM**

- DOM history
  - Netscape created the original "*browser API*", which Microsoft copied for IE
  - Netscape and IE had different API models, so W3C stepped in
  - W3C drew up the standards they coined as **"The Document Object Model"** (**DOM** *for short)*
  - *Netscape's original API was included, and is now referred to as* ***Legacy DOM***

intro.DOM

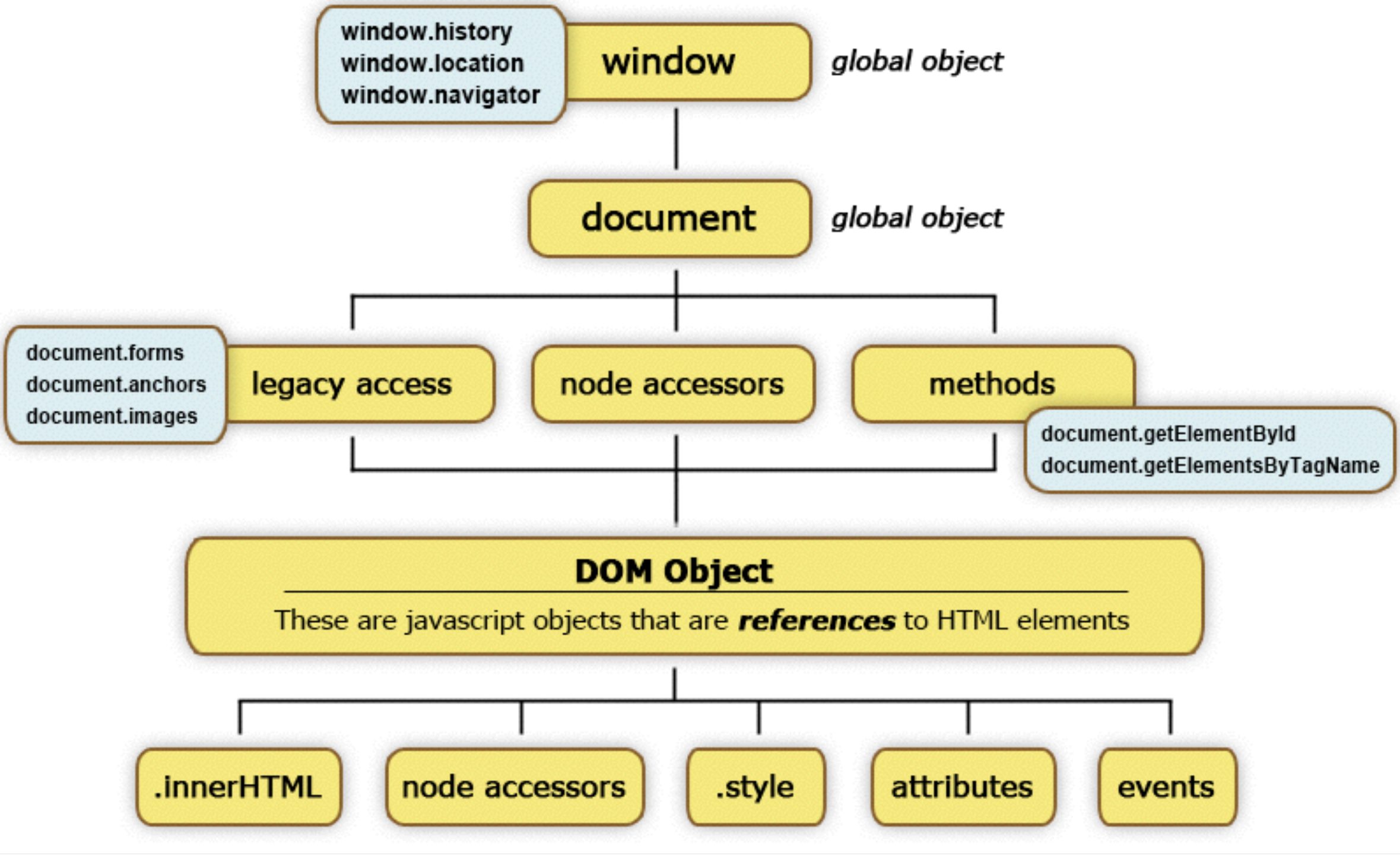
- W3C's DOM
  - the current DOM version is Level 3 *(2004)*, however most IE versions are missing components, and IE still has several proprietary methods
  - *for our purposes this month, we will focus on cross-browser only*
  - *we're also not going to worry about IE6 or earlier*

- *fun fact: IE was the first to implement W3C DOM specs (same with CSS)*

intro.DOM

- W3C's DOM API
  - *there are 3 key concepts for what the DOM API provides us:*
    - **XML tree:** every html element of the page is mapped into XML nodes
    - **API:** the API provides methods and properties for creating elements and searching for existing html elements
    - **Document Object References:** the DOM allows JavaScript to treat any html element as an ***object*** **-** hence, the "**Document Object Model**"

basic.DOM


window.history
window.location
window.navigator
window
global object
document
global object
document.forms
document.anchors
document.images
legacy access
node accessors
methods
document.getElementById
document.getElementsByTagName
DOM Object
These are javascript objects that are references to HTML elements
.innerHTML
node accessors
.style
attributes
events

basic.DOM

- “window” object
  - the window object represents an open window in a browser
  - **Note:** there is no public standard that applies to the window object, but all major browsers support it
  - http://www.w3schools.com/jsref/obj_window.asp

basic.DOM

- “document” object
  - each HTML document loaded into a browser window becomes a document object
  - the document object provides access to ALL HTML elements on a page, from within a script
  - **Tip:** The document object is also part of the Window object, and can be accessed through the window.document property.
  - **Note:** The document object can also use the properties and methods of the Node object.
  - http://www.w3schools.com/jsref/dom_obj_document.asp

basic.DOM

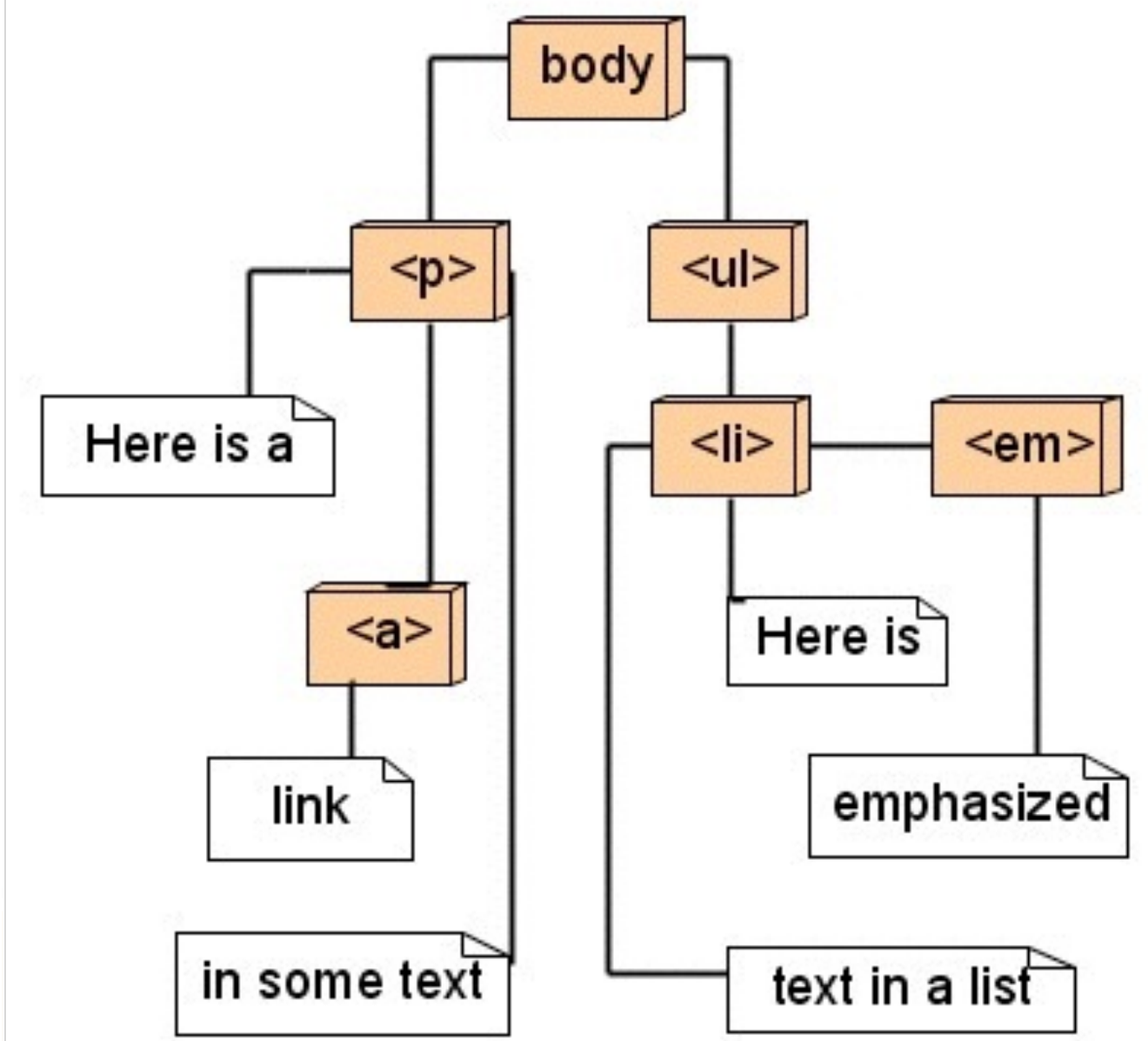
- “node” object
  - the node object represents a node in the HTML document
  - a node in an HTML document is:
    - the document
    - an element
    - an attribute
    - text
    - a comment
  - http://www.w3schools.com/jsref/dom_obj_node.asp

basic.DOM

- key concepts
  - using the DOM API gives us *DOM elements as object references*
  - we'll target html elements using the DOM search methods
  - begin our searches usually at an ID element
    - with a DOM reference, **.innerHTML** contains a string of its contents
    - access attributes of the html tag as object properties.
    - in event handlers, use **return false** to prevent browser defaults

```
<body>
  <p>Here is a <a href="">link</a> in some text.</p>
  <ul><li>Here is<em>emphasized</em>text in a list.</li></ul>
</body>
```

```
<body>
  <p>
    Here is a
    <a href="">
      link
    </a>
    in some text.
  </p>
  <ul>
    <li>
      Here is
      <em>
        emphasized
      </em>
      text in a list.
    </li>
  </ul>
</body>
```



basic.DOM

```
<ul id="nav">
  <li></li><li></li><li></li>
</ul>
```

- DOM Searching
  - searches by **id** can only return 1 element

```
var obj = document.getElementById("#nav");
```

firebug

```
<ul id="nav" />
```

  - However, **querySelectorAll** search will always return an array of results

```
var obj = document.querySelectorAll("#nav li");
```

firebug

```
[ <li />, <li />, <li /> ]
```

basic.DOM

- DOM Searching
  - searches by ID or class and only returns the first1 element: **querySelector**

```
var obj = document.querySelector(“#nav”);
```

  - the **getElementsByTagName** search will always return an array of results

```
var obj = document.getElementsByTagName(“a”);
```

basic.**DOM**

- **Key terms** to keep in mind:

- document object model (DOM)
- DOM API: *XML Nodes, Search Methods, HTML as Object References*
- DOM Collections
- document.*getElementById*
- document.*getElementsByTagName*
- document.*querySelectorAll*
- document.*querySelector*
- element.*innerHTML*
- DOM Events / Listeners & Handlers

basic.DOM

- Traversal
  - "Traversing" means finding other html elements from already selected element(s)
  - we use the node object for traversing, examples
    - firstChild
    - lastChild
    - parentNode
    - nextSibling
    - previousSibling
    - childNodes

basic.DOM

- DOM Manipulation
  - HTML elements also have attributes, things like "href", "src", "title", etc
  - to access these attributes, there are specific setter/getter methods
    - **setAttribute**
      - syntax: **element.setAttribute(attr, value)**
        - always initializes an attribute to a new value
    - **getAttribute**
      - syntax: **element.getAttribute(attr)**
        - always returns a string

courseMaterial.**4**

javascript.**Events**

javascript.**Events**

- listener and handler
  - there are 2 key aspects to any event, the **event listener** and the **event handler**
    1. the event listener is an property associated with the DOM that waits for the event trigger to occur (i.e click, mouseover), and then fires the *event handler*
    2. the **event handler** is the function that will execute when the event is fired

javascript.**Events**

- event.Listener
  - using dot syntax, the ***listener*** for any element is available as a property
  - the name of the listener is prefixed by word “on” (see the event name in the next slide), and must be all lowercase

```
element.onclick                 //click listener
```

```
element.onmousemove             //mousemove listener
```

Event Name	*Usable Elements*	*Triggered By*
change	input, select, textarea	element content has been changed, fires when element loses focus
focus	label, input, select, textarea, button	element is clicked on or tabbed to
blur	label, input, select, textarea, button	element loses focus
resize	window	user resizes window
scroll	window	user scrolls the window
submit	form	user clicks a “submit” input or presses Enter in a text field
keyup	focused element	user releases a keypress
keydown	focused element	user begins a keypress
mouseover	any visible element	mouse moves onto
mouseout	any visible element	mouse moves off of
mousedown	any visible element	mouse button depressed
mouseup	any visible element	mouse button released
mousemove	any visible element	mouse moves anywhere on the element
click	any visible element	user clicks the element

javascript.**Events**

- **click** event
  - the **click** event will be the most used event
  - there are 2 different click events **.onclick** and **.addEventListener**

```
element.onclick = myFn;
```

```
element.addEventListener('click', myFn, false);
```

javascript.**Events**

- **click** event (**.onclick**)
  - used where ONLY ONE click event is used - the **.onclick** will **NOT** work with multiple **.onclick** events in a single .js file
  - the **.onclick** works with all browsers, **.addEventListener** does not work in older versions of Internet Explorer, which uses .attachEvent instead.

javascript.**Events**

- **click** event (**.addEventListener**)
  - used where multiple click events are needed - the **.addEventListener** will work with multiple click events in a single .js file
  - the **.addEventListener** does not work in older versions of Internet Explorer (before version 9)
  - works on any DOM element, not just HTML elements

javascript.**Events**

- event.Handler
  - the ***handler*** for any element is simply a function
  - you can use a function by reference, or use a function literal

```
elementObj.onclick = myFn;
```

or

```
elementObj.onclick = function(){};
```

```
element.addEventListener('click', myFn, false);
```

or

```
element.addEventListener('click', function(){};, false);
```

javascript.**Events**

- event.Object
  - every event listener automatically passes an ***event object*** with information about the event.
  - the *handler* **must receive** it as an argument.
  - many developers will use the **e** as the function parameter

```
document.getElementById("uniqueID").onclick = function(e){
  console.log(e);
};
```

```
var myFn = function(e){};

element.addEventListener('click', myFn, false);
```

javascript.**Events**

- browser defaults
  - for most events, the browser will trigger a default action - for example, hovering over any element will create a tooltip out of the “title” or “alt” attribute if it exists
  - the one we care the most about is the <a> default action, which tells the window to go to the anchor’s **href** location
  - the window waits for a ***return*** to take place before calling the default
  - *an **.onclick** event function should always **return false** , and call **preventDefault()***

```
document.getElementById(“unique”).onclick =
function(e){
  e.preventDefault();
  return false;
};
```

javascript.**Events**

- browser defaults
  - *an **.addEventListener** event function should always **return false** , and call **preventDefault( )***

```
var myFn = function(e){};
  e.preventDefault();
  return false;
};

element.addEventListener('click', myFn, false);
```

javascript.**Events**

e.stopPropagation();	calling this method from inside an event handler will prevent the *Bubbling Phase* from triggering other events
e.preventDefault();	calling this method from inside an event handler will prevent the browser's default action *(such as following an href or the <form> action)*

- unless you specifically want *bubbling* to occur, using **return false** is the safest bet.

Assignment / Goal 4

- Goal4: Assignment: The Duel - Part III
  - You will use the same files you used for the Duel - Part 2, for this assignment.
    See FSO for the assignment instruction.
- Goal4: Assignment: Guessing Game
  - Log into FSO. This is where all your assignment files will be located as well as Rubrics and assignment instructions
- Commit your completed work into GitHub
  - As part of your grade you will need at least 6 reasonable GIT commits for each assignment.
- In FSO there is an announcement with “Course Schedule & Details” in the title, in that announcement you will see a “Schedule” link which has the due dates for assignments.