Part II
More on JavaScript

Pre-DOM Model

Two Object Models?

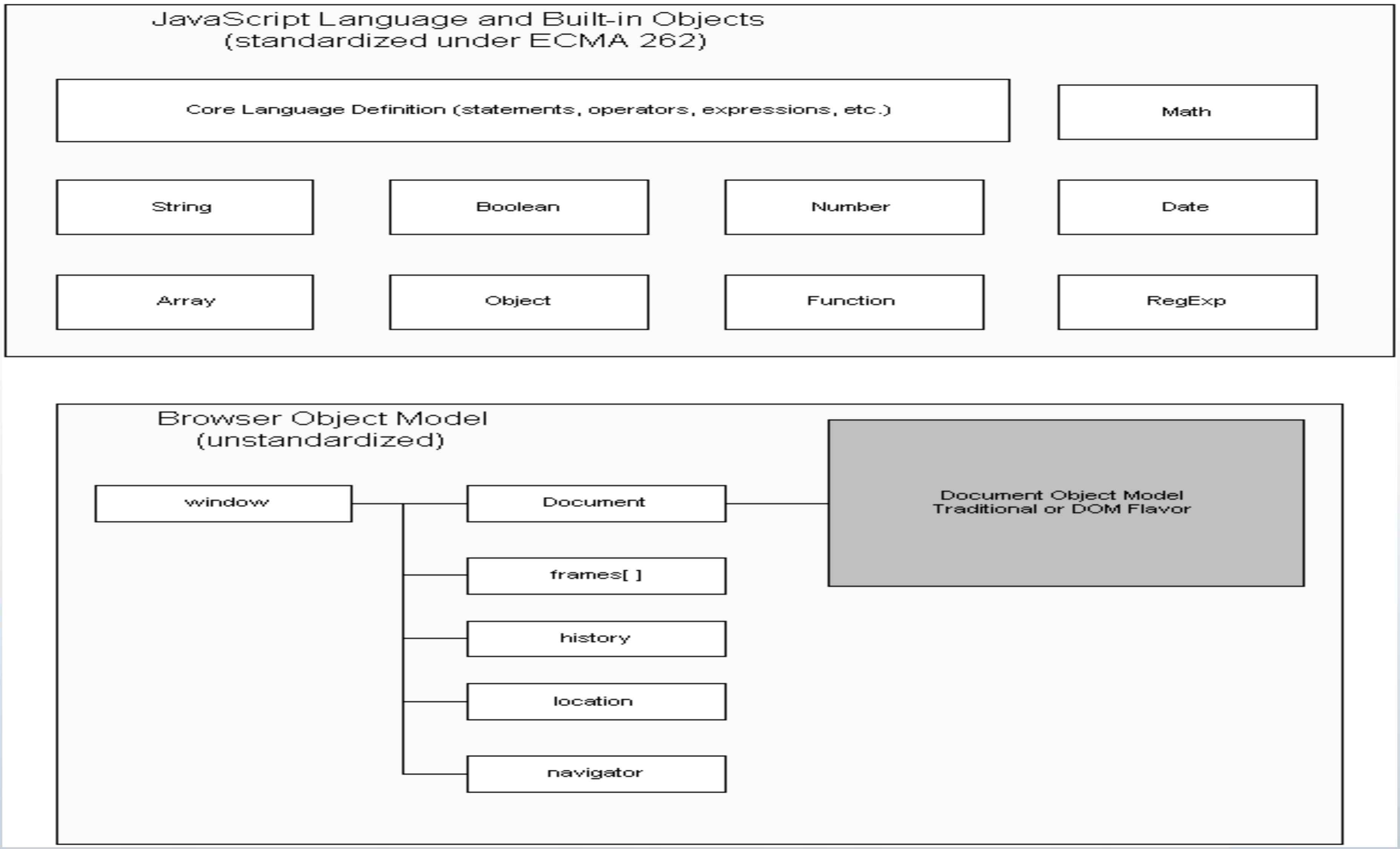
- An object model defines the interface to the various aspects of the browser and document that can be manipulated by JavaScript.
- In JavaScript, two primary object models are employed
  1. a browser object model (BOM)
     - The BOM provides access to the various characteristics of a browser such as the browser window itself, the screen characteristics, the browser history and so on.
  2. document object model (DOM).
     - The DOM on the other hand provides access to the contents of the browser window, namely the document including the various HTML elements ranging from anchors to images as well as any text that may be enclosed by such elements.

The Ugly Truth

- Unfortunately, the division between the DOM and the BOM at times is somewhat fuzzy and the exact document manipulation capabilities of a particular browser's implementation of JavaScript vary significantly.

The Big Picture

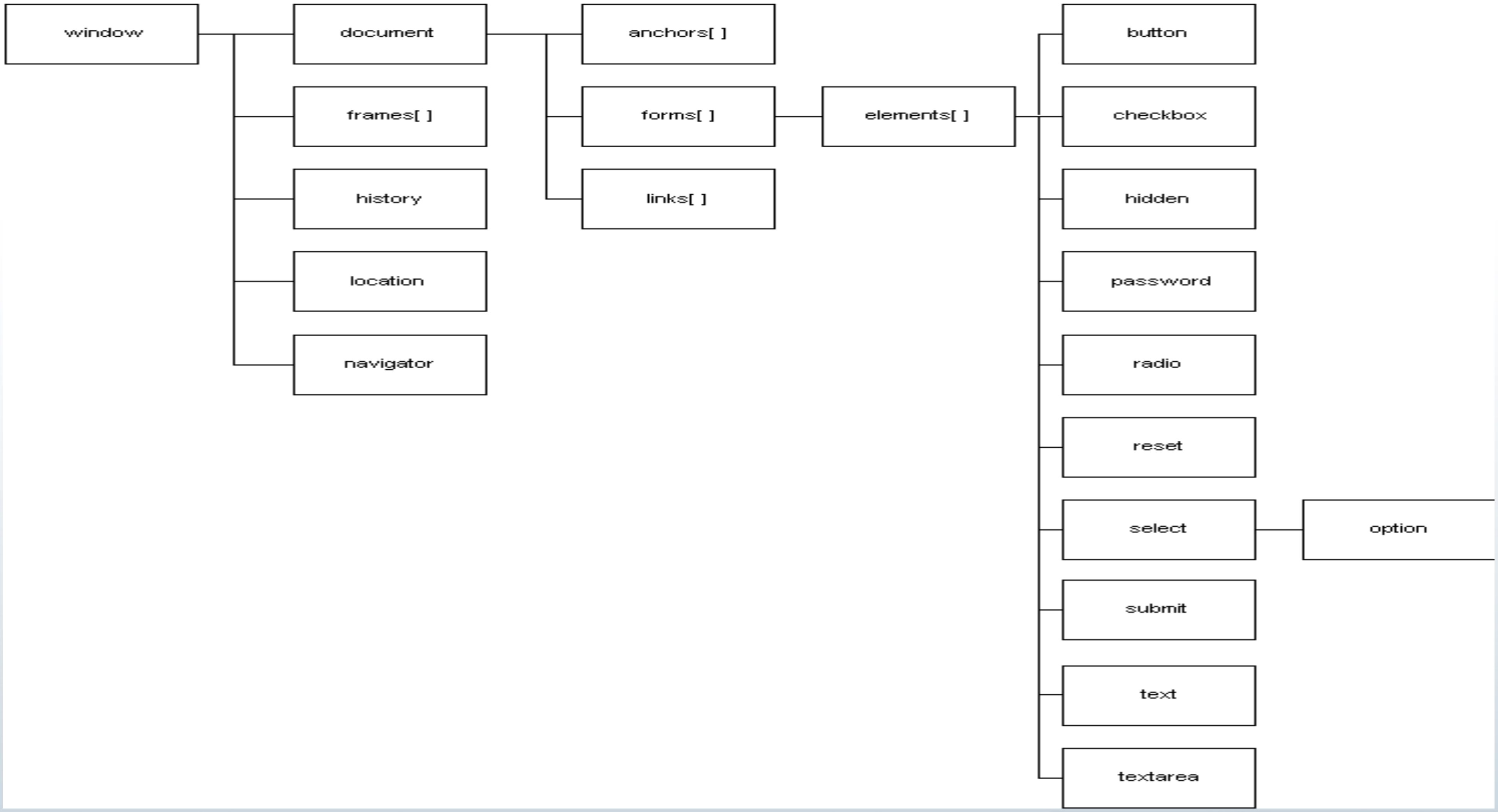
- Looking at the "big picture" of all various aspects of JavaScript including its object models. We see four primary pieces:
  1. The core JavaScript language (e.g. data types, operators, statements, etc.)
  2. The core objects primarily related to data types (e.g. Date, String, Math, etc.)
  3. The browser objects (e.g. Window, Navigator, Location, etc.)
  4. The document objects (e.g. Document, Form, Image, etc.)


JavaScript Language and Built-in Objects
(standardized under ECMA 262)
Core Language Definition (statements, operators, expressions, etc.)
Math
String
Boolean
Number
Date
Array
Object
Function
RegExp
Browser Object Model
(unstandardized)
window
Document
frames[ ]
history
location
navigator
Document Object Model
Traditional or DOM Flavor

~~Four~~ Five Models

- By studying the history of JavaScript we can bring some order to the chaos of competing object models. There have been four distinct object models used in JavaScript including:
  1. Traditional JavaScript Object Model (NS 2 & IE 3)
  2. Extended Traditional JavaScript Object Model (NS 3)
  3. Dynamic HTML Flavored Object Models
     1. a. IE 4
     2. b. NS 4
  4. Traditional Browser Object Model + Standard DOM (Ffox,Chrome,etc.)
  5. HTML5 Model!

Traditional Object Model



Overview of Core Objects

Object	**Description**
Window	The object that relates to the current browser window.
Document	An object that contains the various HTML elements and text fragments that make up a document. In the traditional JavaScript object model, the **Document** object relates roughly the HTML **<body>** tag.
Frames[ ]	An array of the frames in the Window contains any. Each frame in turn references another **Window** object that may also contain more frames.
History	An object that contains the current window's history list, namely the collection of the various URLs visited by the user recently.
Location	Contains the current location of the document being viewed in the form of a URL and its constituent pieces.
Navigator	An object that describes the basic characteristics of the browser, notably its type and version.

Document Object

- The **Document** object provides access to page elements such as anchors, form fields, and links as well as page properties such as background and text color.
- Consider
  - document.alinkColor, document.bgColor, document.fgColor, document.URL
  - document.forms[ ], document.links[ ], document.anchors[ ]
- We have also used the methods of the **Document** object quite a bit
  - document.write( ) , document.writeln( ), document.open( ), document.close( )

Object Access by Document Position

- HTML elements exposed via JavaScript are often placed in arrays or collections. The order of insertion into the array is based upon the position in the document.
- For example, the first **<form>** tag would be in document.forms[0], the second in document.forms[1] and so on.
- Within the form we find a collection of elements[ ] with the first **<input>, <select>** or other form field in document.forms[0].elements[0] and so on.
- As arrays we can use the length property to see how many items are in the page.
- The downside of access by position is that if the tag moves the script may break

Object Access by Name

- When a tag is named via the **name** attribute (HTML 4.0 - **<a>, <img>,** embedded objects, form elements, and frames) or by **id** attribute (pretty much every tag) it should be scriptable.
- Given

```
<form id="myform" name="myform">
   <input type="text" name="username" id="username">
</form>
```

we can access the form at window.document.myform and the first field as window.document.myform.username

Object Access by Associative Array

- The collection of HTML objects are stored associatively in the arrays.
- Given the form named “myform” we might access it using

```
window.document.forms[“myform”]
```

- In Internet Explorer we can use the **item( )** method like so

```
window.document.forms.item(“myform”)
```

Modern Access Solutions

- document.getElementById()
  - id is not a name replacement completely, think form fields (name-value pairs)
- document.getElementsByClassName()
- document.querySelectorAll()
- $() this is a wrapper function folks!
- Be careful though modern doesn’t always equal better as we’ll see…things are still a mess at times and speed is the main thing…if older works everywhere and is faster why avoid it?

Event Models

- JavaScript reacts to user actions through event handlers (code associated with a particular event or object and event in the page)
- Common events include Click, MouseOver, MouseOut, etc.
- Events can be registered through HTML event handlers like onclick or via JavaScript directly
  - <input type="button" value="press" onclick="alert('hi')">
  - document.onload = new Function("alert('hi')");
- We'll see events primarily with links, form items and mouse movement

All Together

- Once document objects are accessed either by user event or script event we can then modify the various properties of the elements.
- The following examples on the next slides show reading and writing of form fields as a demonstration of this.

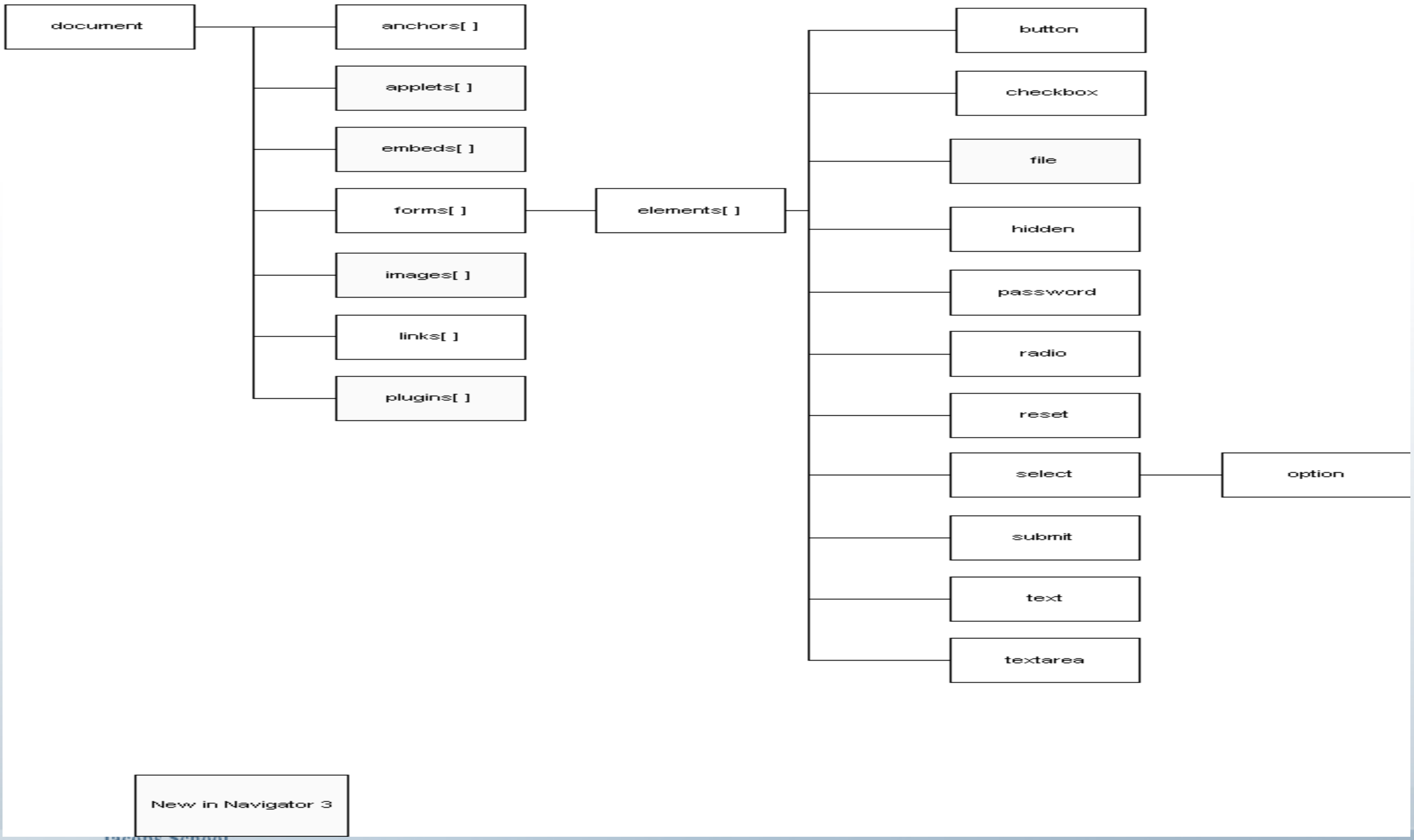
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
    <html>
    <head>
    <title>Meet and Greet</title>
    <script type="text/javascript">

    function sayHello()
    {
     var theirname=document.myform.username.value;
     if (theirname !="")
      alert("Hello "+theirname+"!");
     else
      alert("Don't be shy.");
    }
    </script></head><body>
    <form name="myform" id="myform">
    <b>What's your name?</b>
    <input type="text" name="username" id="username"  size="20"><br><br>
    <input type="button" value="Greet" onclick="sayHello()">
    </form>
    </body>
    </html>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
          "http://www.w3.org/TR/html4/loose.dtd">
    <html><head><title>Meet and Greet 2</title>
    <script type="text/javascript">
    function sayHello()
    {
     var theirname = document.myform.username.value;
     if (theirname != "")
      document.myform.response.value="Hello "+theirname+"!";
     else
      document.myform.response.value="Don't be shy.";
    }
    </script></head><body>
    <form name="myform" id="myform">
    <b>What's your name?</b>
    <input type="text" name="username" id="username"  size="20">
    <br><br>
    <b>Greeting:</b>
    <input type="text" name="response" id="response" size="40">
    <br><br>
    <input type="button" value="Greet" onclick="sayHello()">
    </form></body></html>
```

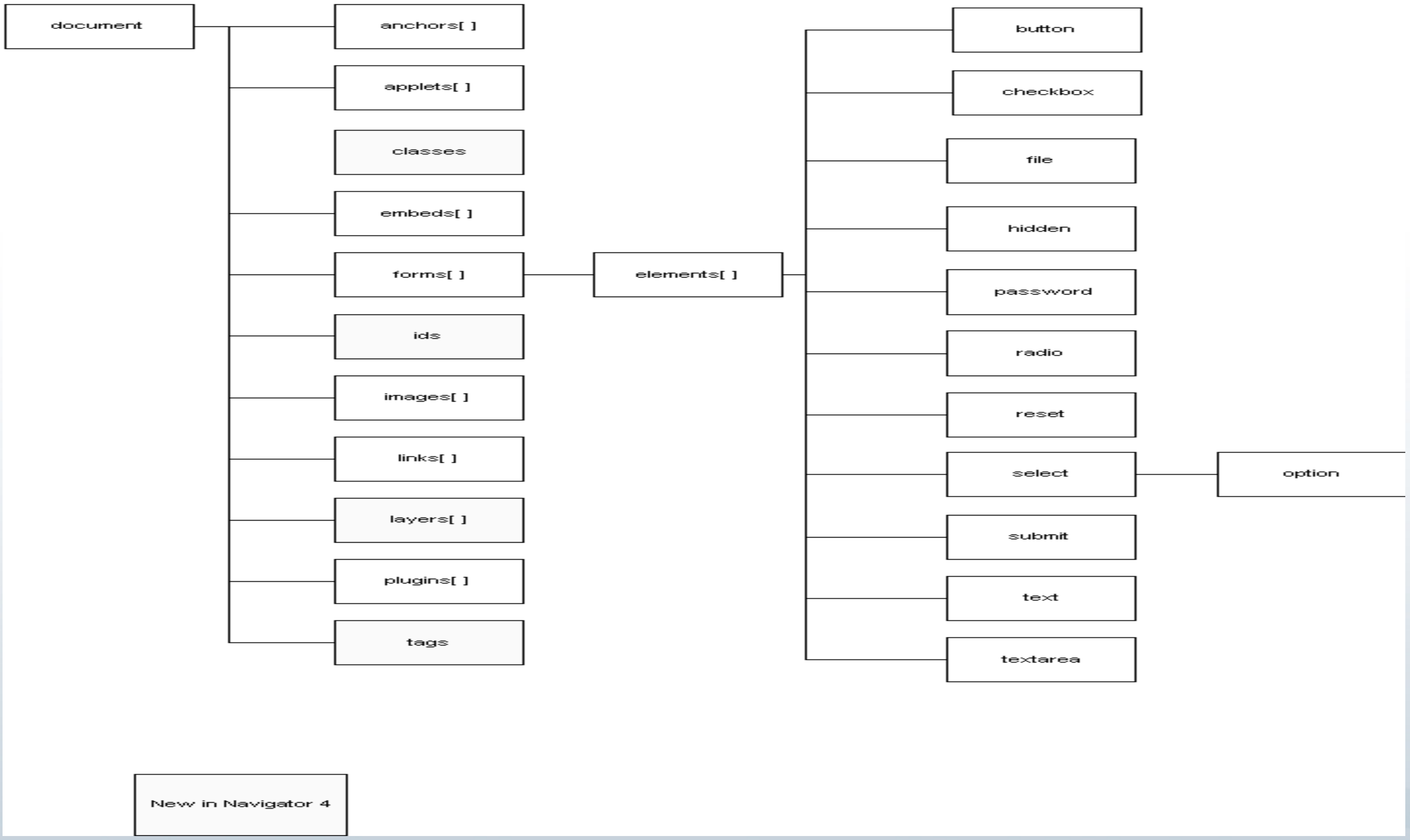
The Object Models

- The next few slides present the various object models supported pre-standard DOM. In JavaScript 1 we focus primarily on the Netscape 3 DOM with some introduction to the non-standard DHTML object models.

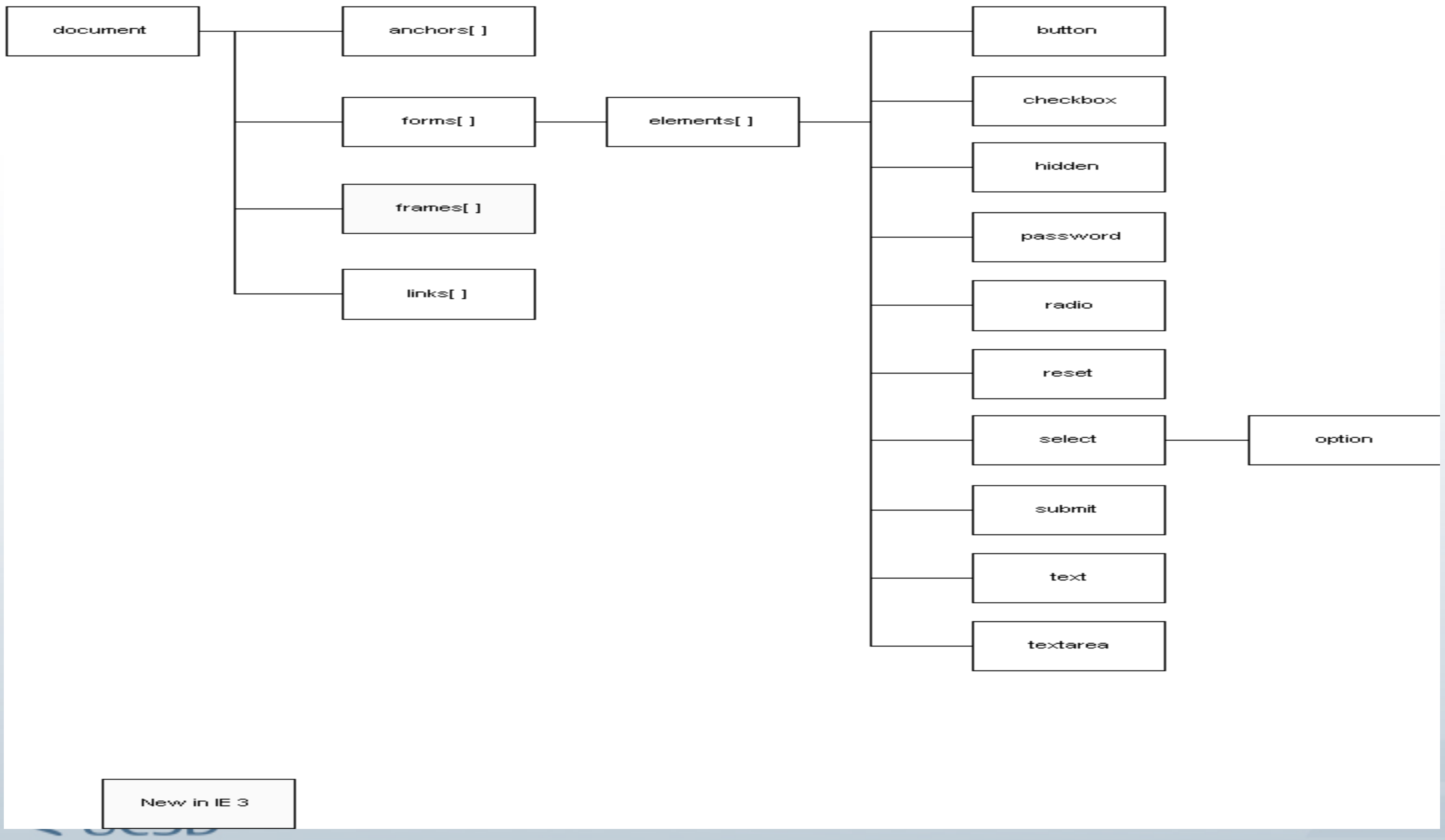
Specific Object Models: Netscape 3



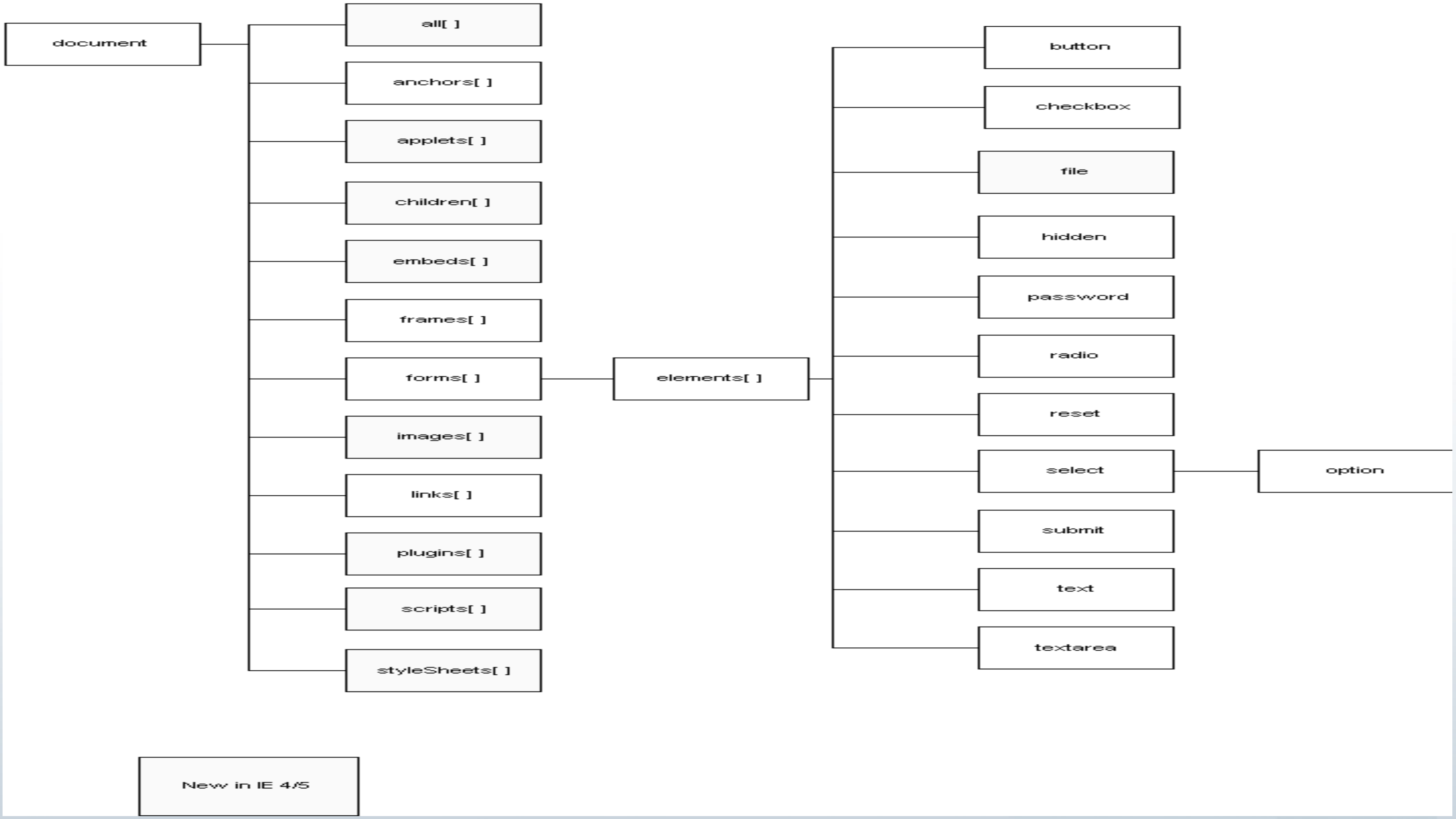
Specific Object Models: Netscape 4



Specific Object Models: Internet Explorer 3



Specific Object Models: Internet Explorer 4



The Cross Browser Nightmare

- The problem we face with JavaScript is that each object model is different
- Somehow we either have to find a common ground (traditional model), use object detection, use browser detection, pick a particular object model like IE and stick with it or just hope the standards work out
- We'll see with the rise of the Document Object Model (DOM) that someday maybe only certain BOM features will be non-standard and all browsers will have the same ability to manipulate page content.

The Standard Document Object Model

DOM Flavors

- The Document Object Model or DOM is a standard that maps HTML and XML documents into objects for manipulation by scripting languages such as JavaScript
- The DOM comes in the following flavors:
  - DOM Level 0 – roughly equivalent to NS3's object model. Often called traditional or classic object model
  - DOM Level 1 – Maps all the HTML elements and provides generic "node" manipulation features via the document object.
  - DOM Level 2 – Maps all CSS properties

Note: The later DOM levels also support the earlier objects so "classic" scripts should work under DOM

DOM Flavors Contd.

- Another breakdown of the DOM is
  - DOM Core – core features for node manipulation (create, delete, movement, etc.)
  - DOM HTML – bindings to HTML tags (HTMLParagraph, etc.)
  - DOM CSS – bindings to CSS properties
  - DOM Events – event handling support
  - DOM XML – bindings to deal with user defined XML languages

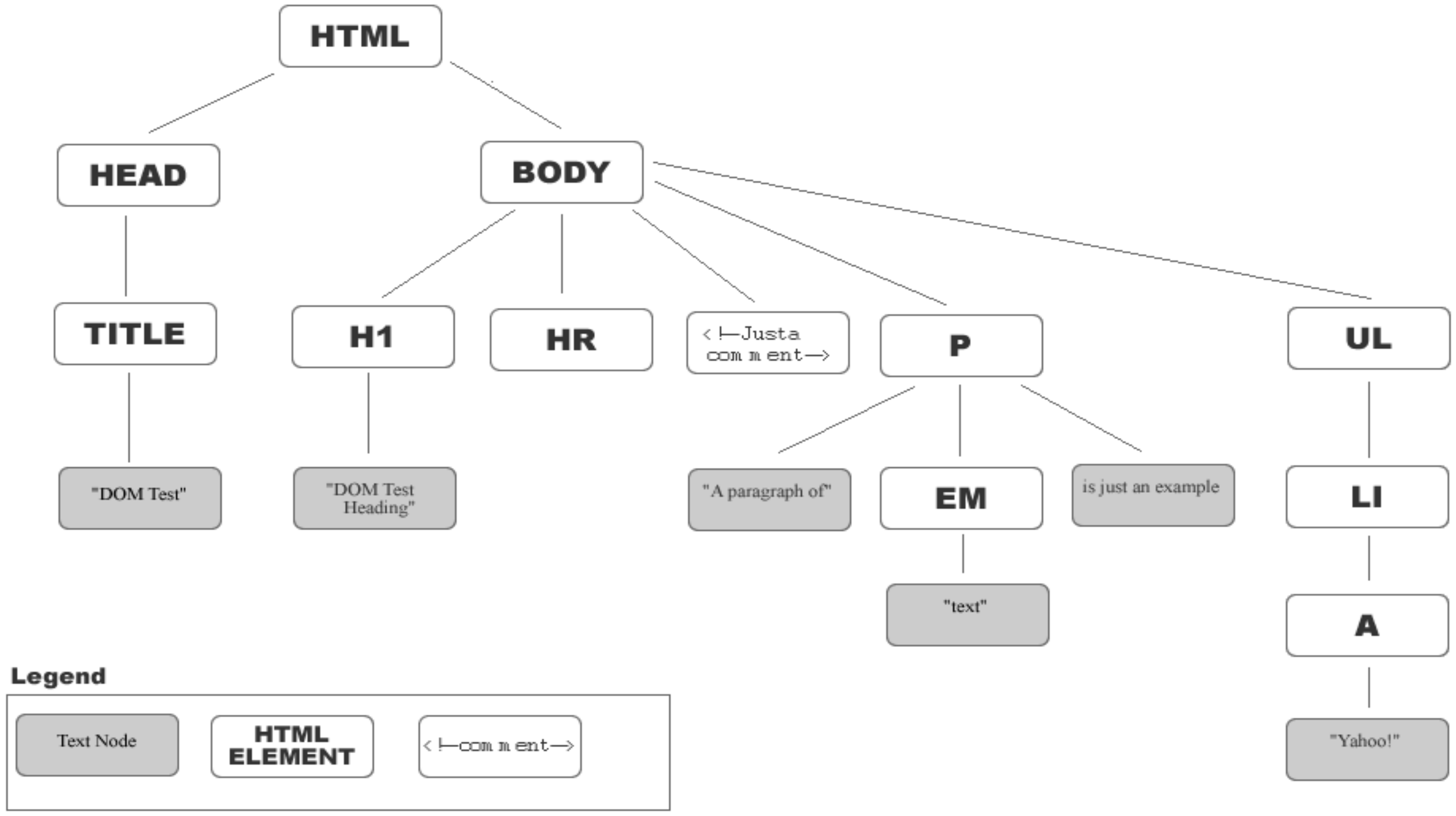
- *Today's modern browsers support DOM Core, DOM HTML, and a good portion of DOM CSS. However, DOM events and DOM XML are not consistently supported*

Document Trees

- The key to understanding the DOM is how an HTML document is modeled as a tree. Consider

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head><title>DOM Test</title></head>
<body>
<h1>DOM Test Heading</h1>
<hr>
<!-- Just a comment -->
<p>A paragraph of <em>text</em> is just an example</p>
<ul>
   <li><a href="http://www.yahoo.com">Yahoo!</a></li>
</ul>
</body>
</html>
```

Modeled Document Tree



Looking at the Tree

- The tree structure follows the structured nature of HTML. <html> tags encloses <head> and <body>. <head> encloses <title> and so on.
- Each of the items in the tree is called generically a node
- Notice that are different types of nodes corresponding to HTML elements, text strings, and even comments. The types of nodes relevant to most JavaScript programmers is shown on the next slide.

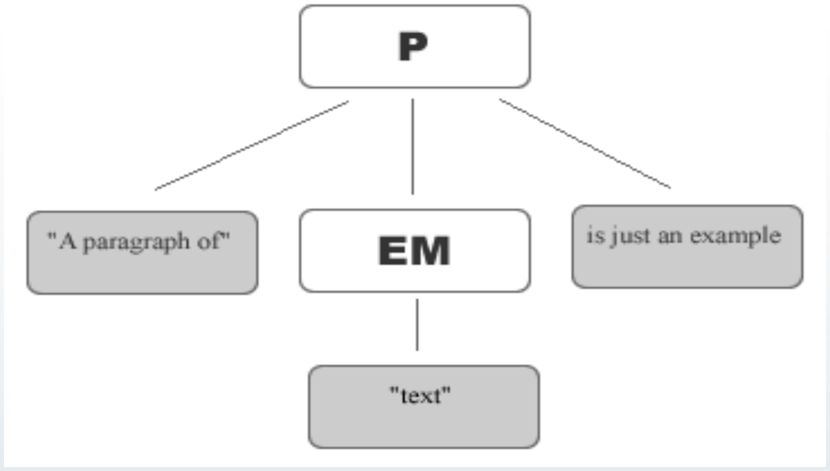
Node Types

Node Type Number	Type	Description	Example
1	Element	An HTML or XML element.	<p>…</p>
2	Attribute	An attribute for an HTML or XML element.	align="center"
3	Text	A fragment of text that would be enclosed by an HTML or XML element	This is a text fragment!
8	Comment	An HTML comment	<!-- This is a comment -->
9	Document	The root document object, namely the top element in the parse tree	<html>
10	DocumentType	A document type definition	<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

Node Relationships

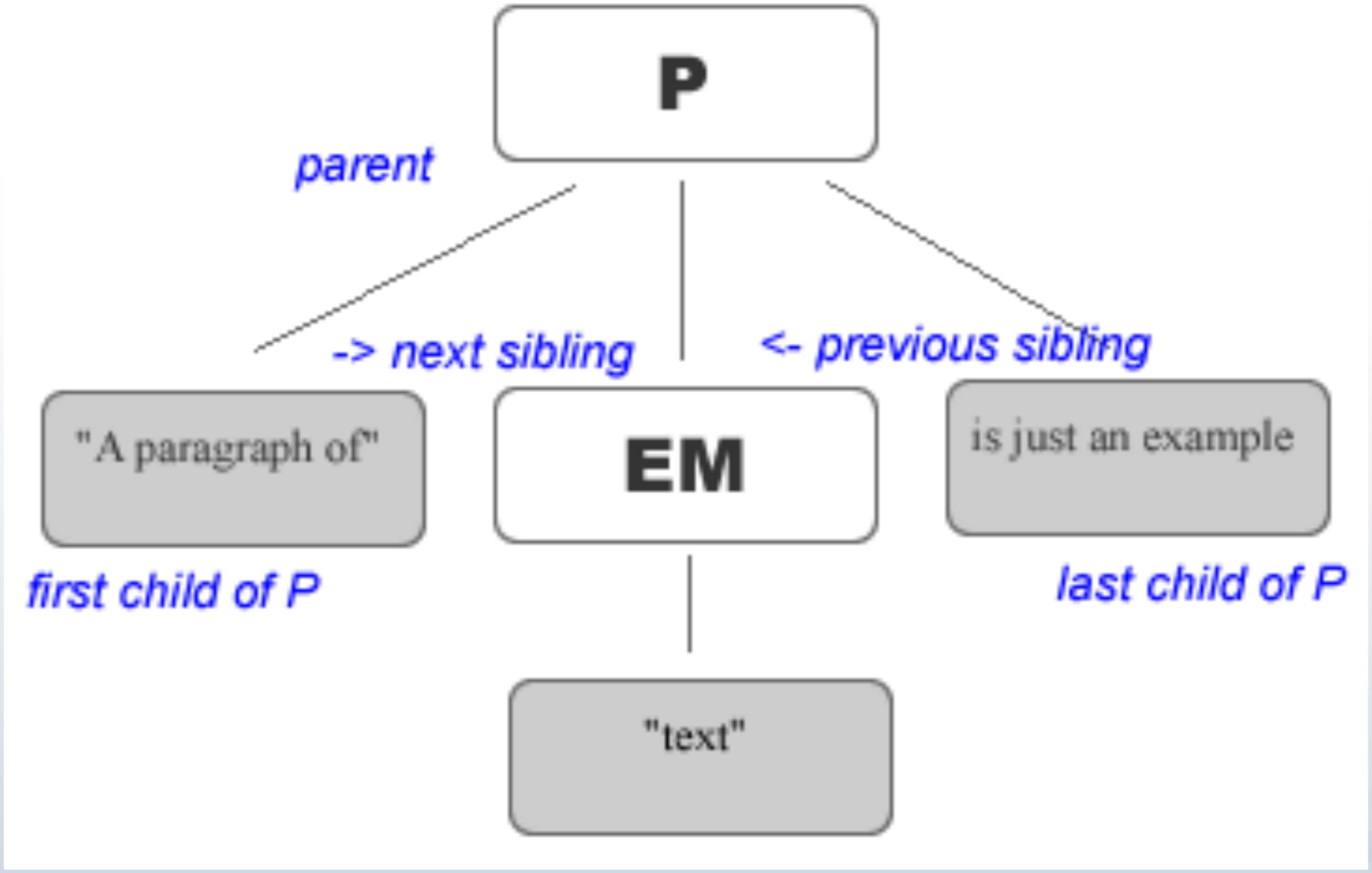
- Look at the tree for

  <p>A paragraph of <em>text</em> is just an example</p>



Notice that the **<p>** tag has three direct children and one "grandchild" Also make sure you understand the sibling and parent relationships. The DOM relies on them

Node Relationships Contd.



Accessing Nodes

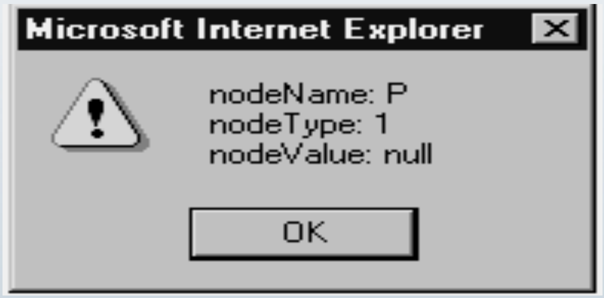
- The easiest way to access nodes in a document tree is via the getElementById( ) method for the **Document** object.
- In order to use the method we need to name our tags using the core attribute **id** like so

```
<p id="p1" align="center">A paragraph of
<em>text</em> is just an example</p>
```

Accessing Nodes Contd.

- Using document.getElementById(‘p1’) we are returned an DOM **Element** object that corresponds to the appropriate node in the tree.

```
var currentElement = document.getElementById('p1');
 var msg = "nodeName: "+currentElement.nodeName+"\n";
 msg += "nodeType: "+currentElement.nodeType+"\n";
 msg += "nodeValue: "+currentElement.nodeValue+"\n";
 alert(msg);
```



```
nodeName: P
nodeType: 1
nodeValue: null
```

Accessing Nodes Contd.

- Notice the node value to be *1* (an element), the type *P* corresponding to the HTML <p> tag, and the **nodeValue** is *null*.
- The reason for the *null* value is that you have to look at a text node to see the text within the parent tag. We now need to learn how to move around the tree. Fortunately there are some generic node properties that make this very easy as summarized on the next slide.

DOM Node Properties

DOM Node Properties	**Description**
nodeName	Contains the name of the node
nodeValue	Contains the value within the node, generally only applicable to text nodes
nodeType	Holds a number corresponding to the type of node, as given in Table 10-1
parentNode	A reference to the parent node of the current object, if one exists
childNodes	Access to list of child nodes
firstChild	Reference to the first child node of the element, if one exists
lastChild	Points to the last child node of the element, if one exists
previousSibling	Reference to the previous sibling of the node; for example, if its parent node has multiple children
nextSibling	Reference to the next sibling of the node; for example, if its parent node has multiple children
attributes	The list of the attributes for the element
ownerDocument	Points to the HTML **Document** object in which the element is contained

Basic Movement

- Using the common node properties you should be able to move around a tree that you know the structure of

```
 var currentElement = document.getElementById('p1');
currentElement = currentElement.firstChild;
currentElement = currentElement.nextSibling;
currentElement = currentElement.parentNode;
```

- This simple script would end up right were it started from assuming that the starting node had at least two children.

Basic Movement Contd.

- We need to be careful though when we don’t know the tree structure ahead of time
- Use simple conditionals to protect yourself from moving “off” tree

```
if (current.hasChildNodes())
   current = current.firstChild;

if (current.parentNode)
   current = current.parentNode;
```

- You should be able to easily write a safe tree traversal system once you know the core properties and how to do **if** statements

getElementsByName

- Related to getElementById( ) is the DOM method getElementsByName( ) which deals with HTML elements identifed by the name attribute including: **<form>**, **<input>**, **<select>**, **<textarea>, <img>, <a>, <area>,** and **<frame>**

- Elements using **name** actually didn't have to have globally unique names thus the DOM method getElementsByName( ) returns a list of nodes with the passed name as shown here looking for something called 'mytag'

```
tagList = document.getElementsByName('myTag');
 for (var i = 0; i < tagList.length; i++)
    alert(tagList[i].nodeName);
```

getElementsByClassName and Other Ideas

- These methods are NOT part of the DOM standard yet but are part of HTML5 however even if not it is relatively easy to implement them
    1. Find all elements in the tree via a walk
    2. Compare the class values with the class or selector being searched for
    3. If compares add to return list else move on

- Of course while it would be easy enough to implement such routines their performance can be quite slow on large documents

- Most all browsers now implement getElementsByClassName natively so you should use an if to see if this feature is in place before using your own. We also note that querySelectorAll( ) and similar library defined functions (ex. $()) allow us to find elements by CSS selector for example $('p.foo > em') might return a list of elements that meet this CSS rule

Traditional JavaScript Collections

- For backwards compatibility the DOM supports some object collections such as document.forms[ ] , document.images[ ] and so forth which were commonly supported amongst JavaScript aware browsers.

Collection	**Description**
document.anchors[ ]	A collection of all the anchors in a page specified by **<a name=””> </a>**
document.applets[ ]	A collection of all the Java applets in a page
document.forms[ ]	A collection of all the **<form>** tags in a page
document.images[ ]	A collection of all images in the page defined by **<img>** tags
document.links[ ]	A collection of all links in the page defined by **<a href=””> </a>**

Generalized Element Collections

- Under the DOM you can create an arbitrary collection of elements using getElementsByTagName( )

```
allparagraphs = document.getElementsByTagName('p');
```

- You can use many of these methods on nodes themselves to find the elements within a particular element

```
allparagraphsinbody = document.body.getElementsByTagName('p');

para1=document.getElementById('p1');
emElements = para1.getElementsByTagName('em');
```

Common Tree Starting Points

- Rather than using a built-in collection or a named starting point you may simply want to start at a well know common tree position such as :
- document.documentElement
  - should be the **<html>** tag
- document.body
  - **<body>** tag
- document.doctype
  - should be the <!doctype> statement but may not be and has limited value

Creating Nodes

- You can create nodes and then insert them into the document tree

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

- Of course you may have to then create text nodes to put inside of elements

```
newText = document.createTextNode('Hello there');
```

- Then we will attach things together and attachto the document

```
newNode.appendChild(newText);
document.body.appendChild(newNode);
```

Create Node Methods

Method	**Description**	**Example**
createAttribute(*name*);	Creates an attribute for an element specified by the string *name*. Rarely used with existing HTML elements since they have predefined attribute names that can be manipulated directly.	myAlign = document.createAttribute(“align”);
createComment(*string*);	Creates an HTML/XML text comment of the form <!-- *string* --> where *string* is the comment content.	myComment = document.createComment(“Just a comment”);
createElement(*tagName*)	Creates an element of the type specified by the string parameter *tagName*	myHeading = document.createElement(“h1”);
createTextNode(*string*)	Creates a text node containing *string*.	newText = document.createTextNode(“Some new text”);

Insert and Append Methods

- The two methods for node attaching are insertBefore(*newChild*, *referenceChild*) and appendChild(*newChild*)

- These methods run on a node object, for example

```
newText = document.createTextNode('Hi!');
currentElement = document.body;
insertPt = document.getElementById('p1');
currentElement.insertBefore(insertPt,newText);
```

Copying Nodes

- Use the cloneNode( ) method to make a copy of a particular node. The method take a Boolean argument which indicates if the children of the node should be cloned (a deep clone) or just the node itself

```
var current = document.getElementById('p1');
 newNode = current.cloneNode();
 newSubTree = current.cloneNode(true);
```

Deleting Nodes

- The Node object's removeChild(child) method is useful to delete a node out of the tree. You need to run this node on the parent of the object you are interested in deleting

```
var current = getElementById('p1');
 currentParent = current.parentNode;
 currentParent.removeChild(current);
```

- *Note: The removeChild( ) method does return the node object removed.*

Replacing Nodes

- You can also replace a node using replaceChild(*newchild*, *oldChild*)
- The replaceChild( ) method will destroy the contents of the node replace and does not side effect the old value

Modifying Nodes

- You can't modify an element directly but you can modify its contents particularly text nodes. Given

  ```
  <p id="p1">This is a test</p>
  ```

  Use

  ```
  textNode = document.getElementById('p1').firstChild;
  ```

  then set the textNode's data property

  ```
  textNode.data = "I've been changed!";
  ```

- There are a variety of DOM methods like appendData( ), deleteData( ), insertData( ), replaceData( ), splitText( ), and substringData( ) that can be used, but since the data value is just a string you might want to resort to commonly understood String object methods.

Modifying Attributes

- Attributes can be manipulated by DOM methods like getAttribute(*name*), setAttribute(*attributename*, *attributevalue*) and removeAttribute(*attributeName*) that work off a particular Node object. You can also check for the existence of attributes using the hasAttributes( ) method.
- Most people do not use these DOM methods but directly modify the attributes of the tag like so

  <p id="p1" align="left">This is a test</p>

- You would use

  current = document.getElementById('p1');
  current.align = 'right';

The DOM and HTML

- What you should begin to recognize now is the key to the DOM in most Web pages is understanding HTML
- The various properties of a node correspond directly to its HTML attributes. For example given a paragraph tag **<p>** it corresponds to an HTMLParagraphElement with the following properties align, id, className, title, lang, and dir. Notice the mapping from HTML attributes to object properties is nearly one-to-one except for some situations like the **class** attribute which would be a reserved word and thus is renamed className under the DOM.
- Two word attributes like **tabindex** are represented in the DOM in typical programming camel back form (e.g. tabIndex)

The DOM and HTML

- The ramification of this relationship between HTML and JavaScript via the DOM is that the language can now manipulate any arbitrary HTML element in anyway, but it does require a well formed document otherwise the results can be somewhat unpredictable
- Suddenly, knowing how to do HTML properly actually matters. Even WYSIWYG editors will have to modified to ensure 100% validatable markup to ensure correct JavaScript operation
- The intersection with CSS is very similar and covered under DOM Level 2

The DOM and CSS

- The **style** attribute for an HTML element allows style sheets properties to be set inline. The DOM allows access to this attribute's value, for example given

  ```
  <p id="p1" style="color: red">Test</p>
  ```

  then

  ```
  theElement = document.getElementById('p1');
  theElement.style.color = 'green';
  ```

- What we see is like HTML the various CSS properties map to DOM names directly, so **font-size** becomes fontSize, **background-color** becomes backgroundColor, and so on. There are only one or two exceptions to this conversion.

The DOM and CSS Contd.

- We can manipulate the className and id properties of an element as well to effect a style sheet change
- We can access the complete style sheet using the document.styleSheets[ ] collection and then looking at the cssRules[ ] collection within each **<style>** tag. You can addRule( ), removeRule( ) and insertRule( ) on an given style sheet as well as change the various properties and values.
- Be careful this aspect of the DOM Level 2 is poorly implemented so far in browsers and in IE you may find that non-standard approaches work better

DOM Conclusions

- The DOM represents the possibility for easier cross-browser scripting, but it also requires mastery of CSS and HTML to be used properly
- Some aspects of the BOM are actually easier to use than the DOM
  - Consider creating nodes or manipulating text contents, some programmers find using properties like innerHTML, innerText, outerText, and outerHTML to be far easier than making nodes one by one
- A great deal of legacy code using BOM objects like IE’s document.all[ ] style exist and would have to be ported. This will take time!

Chapter 10

Event Models

Traditional Event Model

- Event Binding with HTML attributes
  - <p onclick=”alert(‘Stop that!’);”>Click me if you can!</p>
  - Casing often camel case in old style HTML - <p onClick=”...”>
  - As HTML attributes case doesn’t matter, these are part of HTML standard
  - HTML4 defines: onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onreset, onselect, onsubmit, and onunload
  - Example: http://javascriptref.com/3ed/ch11/coreeventsattrs.html

HTML5 Event Attributes

- HTML5 introduces a number of events some of which are IEisms and some new to handle native audio/video and markup based form checking
- Some of interest: oncontextmenu, ondrag, ondrop, oninput, oninvalid, onmousewheel, onplay, onprogress, onratechange, onreadystatechange, onseeked, onafterprint, onbeforeprint, onprint, onbeforeunload, onerror, onhashchange, onmessage, ononline, onoffline, onpopstate, onscroll, onstorage
- The book has a full discussion, but given the transitory nature of HTML5 likely there are a number of new ones

Traditional Event Binding

- Simple Example

```
<p id="p1">Please click me!</p>
<script>
document.getElementById("p1").onclick = function ()
{
    alert("Hey stop clicking me!");
};
</script>
```

- Must wait for element to be defined,
- Can’t do multiple binds obviously

Traditional Event Binding

- Wouldn’t work - just second fires

```
<p id="p1">Please click me!</p>
<script>
function click1() { alert("First click handler"); }
function click2() { alert("Second click handler"); }
window.onload = function () {
 document.getElementById("p1").onclick = click1;
document.getElementById("p1").onclick = click2;
};
```

- Can fix in code if you controlled it

```
document.getElementById("p1").onclick = function ()
{ click1(); click2(); };
```

Old Multi-Bind Solution

- Easy enough to address the multi-bind problem
- Make a function that checks current handler & makes new function with old & new handler added to it.
- Example:http://javascriptref.com/3ed/ch11/oldmultieventbind.html

```
function addEvent(obj,event,handler) {
var oldHandler = obj[event];
if (typeof obj[event] != "function")  {
  obj[event] = handler;
} else { obj[event] = function () {
                if (oldHandler) { oldHandler.apply(); }
                   handler.apply();
}}};
```

Event Handler Scope Details

-
  ```
  <script>
  window.id = "theWindow";
  </script>
  <p id="theParagraph" onmouseover="alert(this.id);">Mouse over me!</p>
  ```

-
  ```
  <script>
  window.id = "theWindow";
  function showID() { alert(this.id); }
  </script>
  <p id="theParagraph" onmouseover="showID();">Mouse over me!</p>
  ```

-
  ```
  <script>
  window.id = "theWindow";
  function showID(el) { alert(el.id); }
  </script>
  <p id="theParagraph" onmouseover="showID(this);">Mouse over me!</p>
  ```

Return Values

- Returning true or false to an event handler can change the default behavior

```
<a href="http://www.google.com/" onclick="return false;">
Try to leave</a>
<a href="http://www.w3.org/" onclick="return
confirm('Leave site and proceed to W3C?');">W3C</a>

<form action="handleform.php" onsubmit="return
validateForm(this);">
<!-- form details omitted -->
</form>
```

Firing Events Manually

- In general you can fire an event that a user can trigger themselves

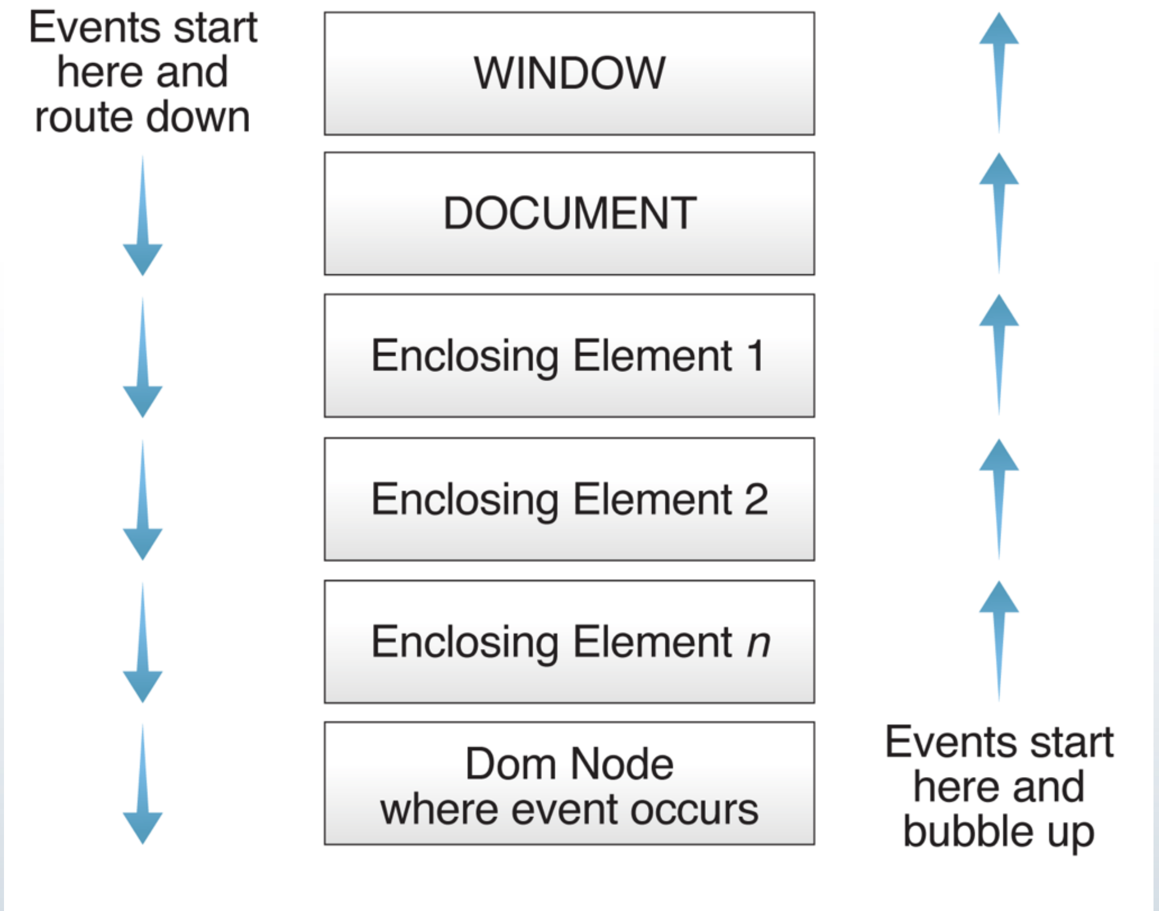
```
<form name="form1">
<input type="button" name="button1" value="Press Me"
onclick="alert('Hey there');">
</form>
<script>
// click the button programmatically
document.form1.button1.click();
</script>
```

- For security reasons some things are not triggerable or not in the same manner as the user would issue it - ex: file upload, mouse movement

Overview of Modern Event Models

Feature	Traditional Model	Netscape 4 Model	Internet Explorer 4–8 Model	DOM2 Model
To bind a handler…	XHTML attributes or direct assignment, `obj.onevent = function`	XHTML attributes, `captureEvents()`	XHTML attributes, `attachEvent()`	XHTML attributes, `addEventListener()`
To detach a handler…	Set XHTML attribute to `null` with script	Set XHTML attribute to `null` with script, `releaseEvents()`	Set XHTML attribute to `null` with script, `detachEvent()`	Set XHTML attribute to `null` with script, `removeEventListener()`
The `Event` object…	N/A	Implicitly available as `event` in attribute text, passed as an argument to handlers bound with JavaScript	Available as `window.event`	Passed as an argument to handlers
To cancel the default action…	Return `false`	Return `false`	Return `false`	Return `false`, `preventDefault()`
How events propagate	N/A	From the `Window` down to the target	From the target up to the `Document`	From the `Document` down to the target and then back up to the `Document`
To stop propagation…	N/A	N/A	cancelBubble	`stopPropagation()`
To redirect an event…	N/A	`routeEvent()`	`fireEvent()`	`dispatchEvent()`

Overview of Modern Event Models Contd.


Events start here and route down
WINDOW
DOCUMENT
Enclosing Element 1
Enclosing Element 2
Enclosing Element *n*
Dom Node where event occurs
Events start here and bubble up

Dom events go either way, but are usually handled in “bubble up” fashion.

Internet Explorer's Proprietary Model

- attachEvent() and detachEvent()
- object.attachEvent("event to handle", eventHandler);
- object.detachEvent("event to stop handling", eventHandler);
- Examples
  - http://javascriptref.com/3ed/ch11/attachevent.html
  - http://javascriptref.com/3ed/ch11/detachevent.html

IE Model Contd. - Event Object

- Transient Event object made available via a global variable event
- The event object contains values like
  - Event target values like srcElement, fromElement, toElement
  - Pixel coordinates (clientX, clientY, screenX, screenY, x, y)
  - Modifier keys - altKey, shiftKey, keyCode, ctrlKey
- Example: http://javascriptref.com/3ed/ch11/eventattributesIE.html

IE Model Contd. - Event Bubbling

```
<script>
function gotClick(who)  {
   document.getElementById("results").innerHTML += who + " got the
click <br>";
}
</script>
</head>
<body onclick="gotClick('body');">
<table onclick="gotClick('table');"> <tr onclick="gotClick('tr');">
  <td onclick="gotClick('td');"><p onclick="gotClick('p');">Click on the <b
onclick="gotClick('b');">BOLD TEXT</b> to watch bubbling in action!
</p> </td></tr></table><hr> <br>
<p id="results"> </p>
</body>
```

- Example: http://javascriptref.com/3ed/ch11/iebubble.html

IE Model Contd. - Event Creation

- var myEvent = document.createEventObject([eventObjectToClone])
- var evt = document.createEventObject (window.event);
  evt.button = 1;
  evt.clientX = Math.floor(Math.random()*800);
  evt.clientY = Math.floor(Math.random()*600);
- Then send event with fireEvent()
  document.body.fireEvent("onclick", evt);
- Example: http://javascriptref.com/3ed/ch11/eventsIE.html

UCSD Jacobs School

DOM Event Model - Adding Events

- object.addEventListener(event, handler, capturePhase); where:
  - object is the node to which the listener is to be bound.
  - event is a string indicating the event it is to listen for.
  - handler is the function that should be invoked when the event occurs.
  - capturePhase is a Boolean indicating whether the handler should be invoked during the capture phase (true) or bubbling phase (false).
- Example:http://javascriptref.com/3ed/ch11/addeventlistener.html
- Note: You are in charge of tracking what listeners are bound - no listListeners()

DOM Event Model - Removing Events

- object.removeEventListener(event, handler, capturePhase); where:
  - object is the node to which the listener is to be removed.
  - event is a string indicating the event it is to stop listening for.
  - handler is the function that should be removed when the event occurs.
  - capturePhase is a Boolean indicating whether the handler should be invoked during the capture phase (true) or bubbling phase (false).

- Example:http://javascriptref.com/3ed/ch11/removeeventlistener.html

Event Model Abstraction - POC

```
function addListener(obj, eventName, listener) {
if (obj.addEventListener) {
  obj.addEventListener(eventName, listener, false);
} else {
   obj.attachEvent("on" + eventName, listener);
 }
}

function removeListener(obj, eventName, listener) {
  if (obj.removeEventListener) {
      obj.removeEventListener(eventName, listener, false);
   } else {
              obj.detachEvent("on" + eventName, listener); }
}
var el = document.getElementById( ‘p1’ );
addListener(el, “click”, handleClick);
```

DOM Event Model - Event Object

- The DOM Event object contains similar items to the IE Event object (pixel, key, target element, etc.) though the names are slightly different
- See Table 11-8 - p. 446 for info and note bubbles, cancelable, currentTarget, eventPhase, isTrusted, relatedTarget, timeStamp, target, and type
- A big difference is that you do not access this object using a global variable instead it is passed to the event handler function being invoked
- Example:http://javascriptref.com/3ed/ch11/eventattributes.html

DOM Event Model - Event Control

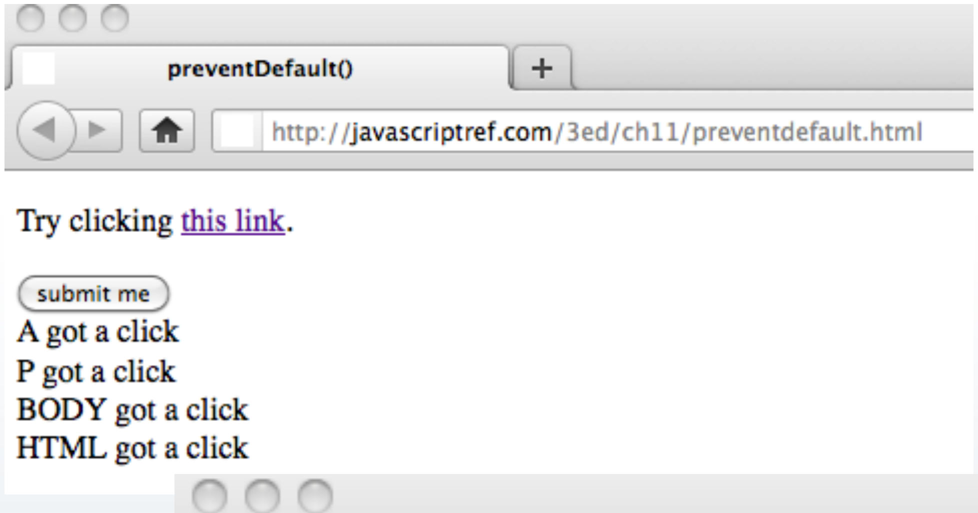
- Preventing Default Actions is beyond just returning false

```
<p>Try clicking <a href="http://www.javascriptref.com">this
link</a>.</p> <form action="http://www.javascriptref.com"
method="get">
<input type="submit" value="submit me">
</form>
<script>
  function killClicks(event) { event.preventDefault(); }
// kill all default click actions!
document.addEventListener("click", killClicks, true);
</script>
```

- Note though that the event may not cause the default action but it will continue up the DOM tree unless told otherwise

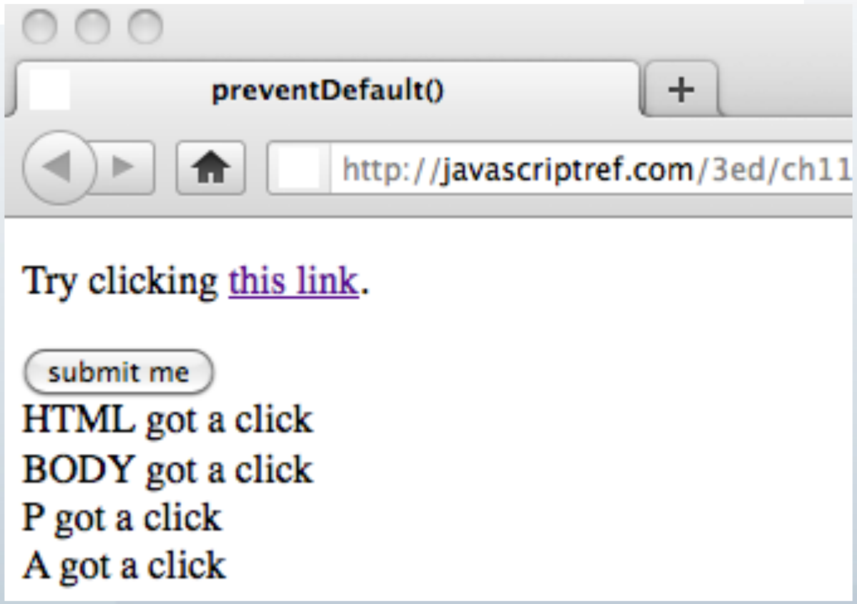
DOM Event Model - Event Control Contd.

- You can control the propagation of an event using `event.stopPropagation()`
- Example: http://javascriptref.com/3ed/ch11/stoppropagation.html
- Of course as shown here the direction of propagation changes depending on how you decide to register the event



Try clicking this link.

A got a click
P got a click
BODY got a click
HTML got a click



Try clicking this link.

HTML got a click
BODY got a click
P got a click
A got a click

DOM Event Creation

- Synthetic events made with `document.createEvent()`

```
evt = document.createEvent(“HTMLEvents”);
```

- Once created you make the event passing it a variety of values to populate the event object properly

```
evt.initEvent(“click”,”true”,”true”);
  // this syntax can get wild
```

- Finally find a node and dispatch the event to it

```
currentNode.dispatchEvent(evt);
```

- Example:

http://javascriptref.com/3ed/ch11/createevent.html

DOM4 Event Creation Changes

- Making events in the DOM is one of its more convoluted areas
- DOM4 introduces a more sensible event constructor syntax.

```
document.getElementById("p2").addEventListener("mouse
   over", function () { var evt = new Event("click",
   {bubbles:true,cancelable:true});
   document.getElementById("p1").dispatchEvent(evt);},
   false);
```

- Example:http://javascriptref.com/3ed/ch11/createevent-constructor.html
- Syntax has changed but not really functionality and developers should be cautious for browser support with this approach

DOM Event Notes - isTrusted

- Events generated by browser or direct user action are “trusted” and thus isTrusted property on Event object is true
- Synthetic events are triggered by code (maybe malicious XSS) so they are not trusted and isTrusted is false.
- Example:http://javascriptref.com/3ed/ch11/istrusted.html
- Do not be naive though reliance on this property and code that may check it is dubious given the dynamic nature of JavaScript

Event Types - Mouse Events

- Mouse events are defined under the MouseEvent interface and include:
  click, dblclick, mousedown, mouseenter, mouseleave, mousemove, mouseout, mouseover, mouseup
- Example:http://javascriptref.com/3ed/ch11/mouseevents.html
- Creation of mouse events is messy!
  var evt = document.createEvent("MouseEvent");
  initMouseEvent(type, bubbles, cancelable, view, detail, screenX, screenY, clientX, clientY, ctrlKey, altKey, shiftKey, metaKey, button, relatedTarget)
- Example:http://javascriptref.com/3ed/ch11/createmouseevents.html
- Mouse wheel handling is troubling:http://javascriptref.com/3ed/ch11/mousewheel.html

UCSD Jacobs School

UI Events

- "UI event" includes:
  DOMActivate, abort, error, load, resize, scroll, select, and unload
- Synthetic UI events are a bit easier than some
  var evt = document.createEvent("UIEvent");
  evt.initUIEvent(*type*, *bubbles*, *cancelable*, *views*, *detail*);

  where
  type is a string representing the particular event to create, such as "DOMFocusIn".
  bubbles is a Boolean value indicating whether or not the event should bubble.
  cancelable is a Boolean value indicating whether or not the event should be cancelable.
  view is the event's AbstractView. You should pass the Window object here.
  detail indicates event-specific details for the spawned event.

- Example:http://javascriptref.com/3ed/ch11/createuievents.html

Other Events

- Keyboard Events:http://javascriptref.com/3ed/ch11/keyboardevents.html
- Text Events: http://javascriptref.com/3ed/ch11/textinput.html
- Mutation Events:http://javascriptref.com/3ed/ch11/mutationevents.html
- Non-Standard Events: http://javascriptref.com/3ed/ch11/oncopy.html
- Custom Events:http://javascriptref.com/3ed/ch11/customevent.html http://javascriptref.com/3ed/ch11/customevent-constructor.html
- onebeforeunload, onreadystatechange, onhashchange, onmessage

Library Rise Phase 1

- Well you can write this native DOM but that is no fun. Maybe you want to smooth things out?
- 1st Gen Libraries – Solved browser inconsistencies in the DOM, Events and Ajax
  - jQuery, YUI, Prototype, MooTools, etc. (jQuery won!)
- Time passes and things change
  - Too much success leads to problems
    - Sloppy Inclusions and Bad Code Flow (thus need structure)
    - DOM no longer a big deal smoothing but updating becomes problem (remove some stuff, add virtual DOM)
    - Things get slow (remove stuff, go native instead)

Library Rise Phase 2

- 2nd Generation Libraries
  - AngularJS, Ember, React, VueJS, etc.
- More focused on larger scale app building
  - MVC pattern focused (maybe not a hot idea)
  - Just the patterns and conventions though are helpful
- Databinding and updating become a big deal
- Overhead starts to become a more serious problem

Back to the Native? Phase 3

- Today we see that native components (Custom Elements, Shadow DOM, etc.) might blow out what many frameworks do
  - Ex: Polymer (React, VueJS, and Angular all pointing way as well)
- The mobile constraint is showing that frameworks may not be worth it (depending on situation)
- But the network is taking center stage again (cue Ajax lecture!)