SIU
CARBONDALE

SOUTHERN ILLINOIS UNIVERSITY
DEPARTMENT OF
COMPUTER SCIENCE

CS 350/491
WEB APPLICATION DEVELOPMENT

JAVASCRIPT AND HTML DOCUMENTS

S IVERSITY
D T OF
C SCIENCE

JavaScript Execution Environment

- JavaScript executes in a browser
- *The window object* represents *the window* that is displaying *the document*
  - **Sitting on the top of the hierarchy and serving as the default context object**
  - **All Window properties are automatically visible to all scripts**
  - **Global variables are actually added properties to the Window object!!**
- *The document (sub)object* represents *the document* that is currently being displayed in *the window*
  - **Is a property of the Window object**

Document Object Model – the DOM

- **DOM Levels**
  - DOM 0: informal, early browsers
  - **DOM 1: XHMTL/XML structure (1998)**
  - **DOM 2: event model, style interface, traversal (2000)**
  - DOM 3: content model, validation (2004), not to be discussed by this course
- **DOM specifications describe <u>an abstract model</u> of a document and its elements**
  - **Each HTML doc is mapped to a tree structure**
  - **Elements mapped to nodes objects and attributes to properties**
  - ***Methods* are the main *interfaces***
  - **Different languages will need to *bind* the interfaces to their specific implementations**
    - **The internal representation may not be tree-like (doesn't matter!)**
    - **In JS, <u>data</u> are represented as properties and <u>operations</u> as methods**

A Quick Illustrative Example

- <input type=“text” name=“address”>
- The *object* representing this element has two properties:
  - The *type* property will have value “text”
  - The *name* property will have value “address”

Element Access in JavaScript

- *Elements* in XHTML are mapped to *objects* in JavaScript which implements the DOM, so elements must be accessed *via the notion of an object*
- Objects can be addressed in several ways:
  - Using the `forms` and the `elements` arrays defined in DOM 0
    - Individual elements are specified by array index
    - The index may change when the form changes (problematic!)
  - Using the *name* attributes of forms and form elements
    - Names now causes validation problems
    - Yet, *names are required* on form elements for providing data to the server
  - Using *getElementById* with id attributes is recommended!!
    - An id attribute value must be unique in the document

Using the *forms* array

- Consider this simple form:

```
<form action = "">
<input type = "button" name  = "pushMe">
 </form>
```

- **This input element can be referenced as**

  ```
  document.forms[0].elements[0]
  ```

Using the *name* Attribute

- **If using the *name* attributes, then all elements from the referenced element up to the body must have a name attribute – the reference link cannot be broken!**
- **This violates XHTML standards in some cases (!)**
- **Example**

```
<form name  = "myForm" action = "">
   <input type  = "button" name = "pushMe">
</form>
```

  - **Reference to the input object**

```
document.myForm.pushMe
```

Using *id* Attributes

- **Must first set the id attribute of the element!**

```
<form action = "">
   <input  type="button" id=“turnItOn">
</form>
```

- **Then use *getElementById***

```
document.getElementById(“turnItOn")
```

  - Advantage – no need to care the intermediate elements!
  - This is the preferred access method!

Events and Event Handling

- *Event-driven programming* **is a style of programming in which pieces of code, called** ***event handlers*****, are activated when certain** ***events*** **occur**
- **Events represent activities in the environment including, especially, user actions, such as moving the mouse or typing on the keyboard**
- **An** ***event handler*** **is a program segment designed to respond to a certain event when it occurs**
- **Events are represented by JavaScript as objects**
- **Proper associations between event sources/generators and the event handlers must be explicitly set**
  - **Note: not every handler is responsive to every event!**
- **The association is achieved by** ***Event handler registration*** **in 2 ways:**
  - **Assign (a handler) to an event attribute of elements – the HTML way**
  - **Assign to a property of a DOM object – the DOM way (in JS code)**

Common Events & Event Attributes of Tags

Event	*Tag Attribute*
`blur`	`onblur`
`change`	`onchange`
`click`	`onclick`
`focus`	`onfocus`
`load`	`onload`
`mousedown`	`onmousedown`
`mousemove`	`onmousemove`
`mouseout`	`onmouseout`
`mouseover`	`onmouseover`
`mouseup`	`onmouseup`
`select`	`onselect`
`submit`	`onsubmit`
`unload`	`onunload`

Events, Attributes and Tags

- **A tag may carry multiple event attributes**
- **The same event attribute may appear in different tags**

  E.g., The ***onclick*** attribute can be in both <a> and <input>

- **A text element gets focus in three ways:**

  1. **When the user puts the** *mouse cursor* **over it and presses left button**
  2. **When the user** *tabs* **to the element**
  3. **When executing the *`focus`* method (of the element) thru JS code**

- **Losing focus is another event, i.e., the *blur* event**

Setting a Handler

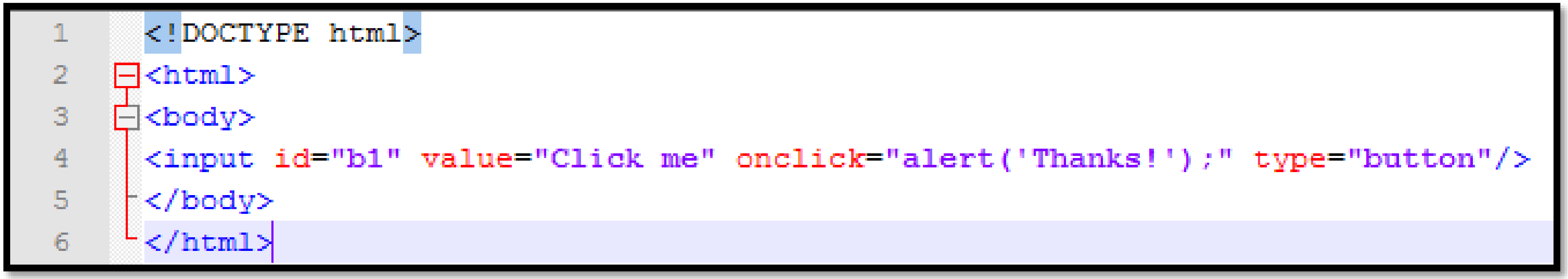
- Use an event attribute specifying a JavaScript command:

```
<input type=“button” name=“myButton”
   onclick=
   “alert(‘You clicked the button!’)”/>
```

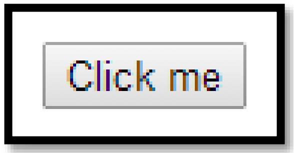
- More often, a user-defined function is used which may enclose more than a single JavaScript statement, e.g.,

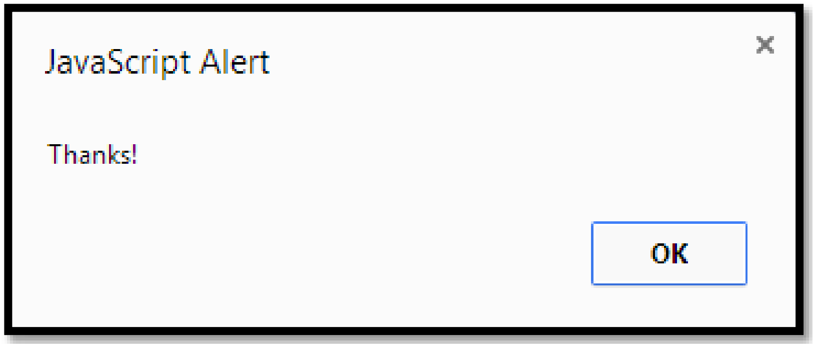
```
<input type=“button” name=“myButton”
   onclick=“myHandler()”/>
   // don’t forget the parentheses!!
```

Events handlers



```
<!DOCTYPE html>
<html>
<body>
<input id="b1" value="Click me" onclick="alert('Thanks!');" type="button"/>
</body>
</html>
```





JavaScript Alert

Thanks!

Handling Events from Button Elements

- **An event can be <u>registered</u> to a tag in <u>two ways:</u>**

```
<input type=“button” name=“freeOffer”
   id=“freeButton”/>
```

1. Assigning to the event <u>attribute of the element</u>

```
<input type=“button” name=“freeOffer”
  id=“freeButton”
  onclick=“freebuttonHandler()”/>
```

2. Assigning to the <u>property of the object</u> (via JavaScript code)

```
document.getElementById(“freeButton”).onclick =
    freeButtonHandler
```

- **Note that the function name, a reference to the function (object), is assigned**
- **Using freeButtonHandler() would assign the <u>returned value</u> of the function call!!**

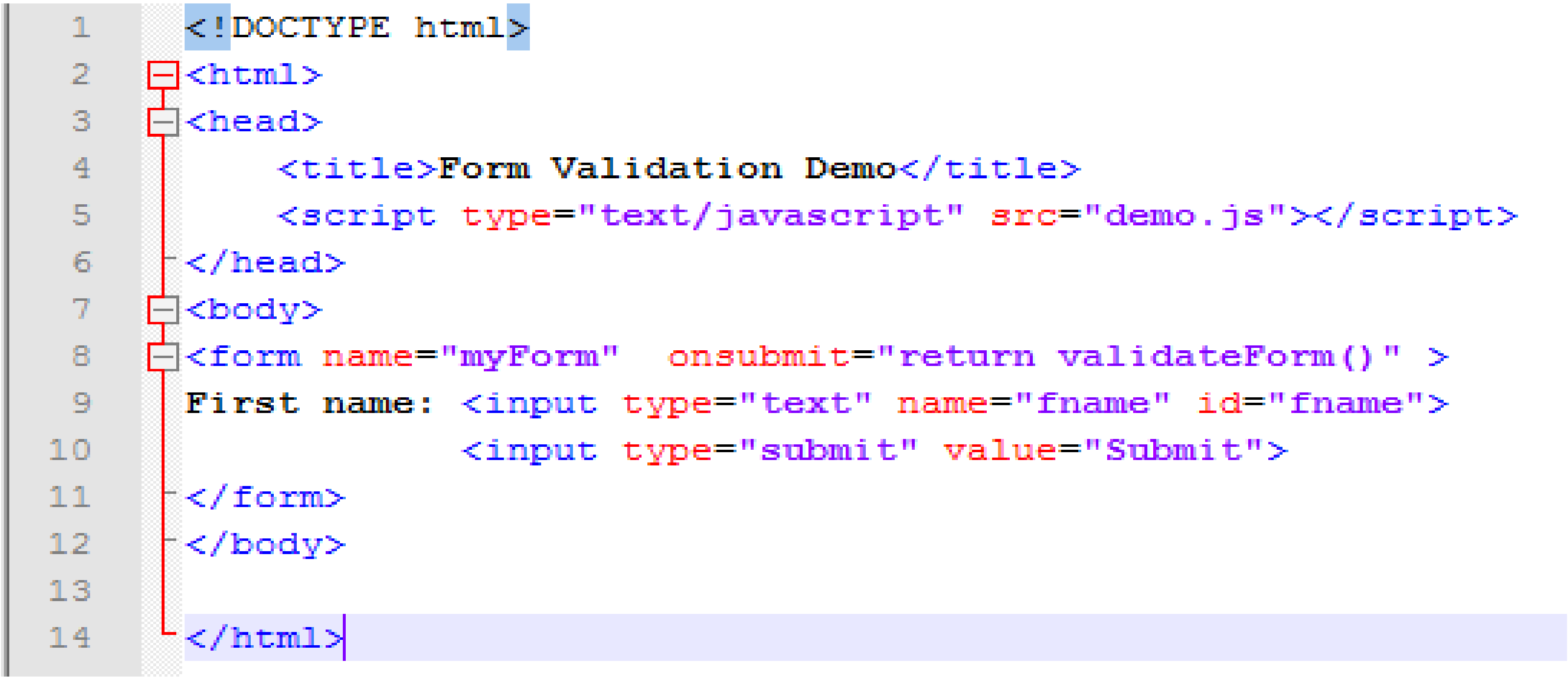
Comparison of the 2 Registration Methods

- **“HTML way” assigns to an attribute is more flexible and allowing passing parameters without creating an anonymous function**
- **“DOM or JS way” assigns to a property – preferred!**
  - **This way helps separate HTML from JS code**
  - **Allows reassignment later if the handler needs to be changed through JS code and gives *more dynamics***
  - ***No parameter passing*, though as a drawback, can be indirectly obtained by assigning through an *anonymous* function**

Validating Form Input

- **Validating data using JavaScript provides quicker user interaction**
- **Validity checking on the server requires a *round-trip*:**
  - **Server checks the data and then responds with an appropriate error page**
- **Must properly *Handling* a data validity error in a "customer-centered" manner – customer interest first, e.g.:**
  - **Pre-focus the field in question**
  - **Highlight the text for easier editing**
- **Note: if an event handler returns false, the default action is not taken by the browser**
  - **This can be used in a Submit button's event handler to check validity and not submit if there are problems**

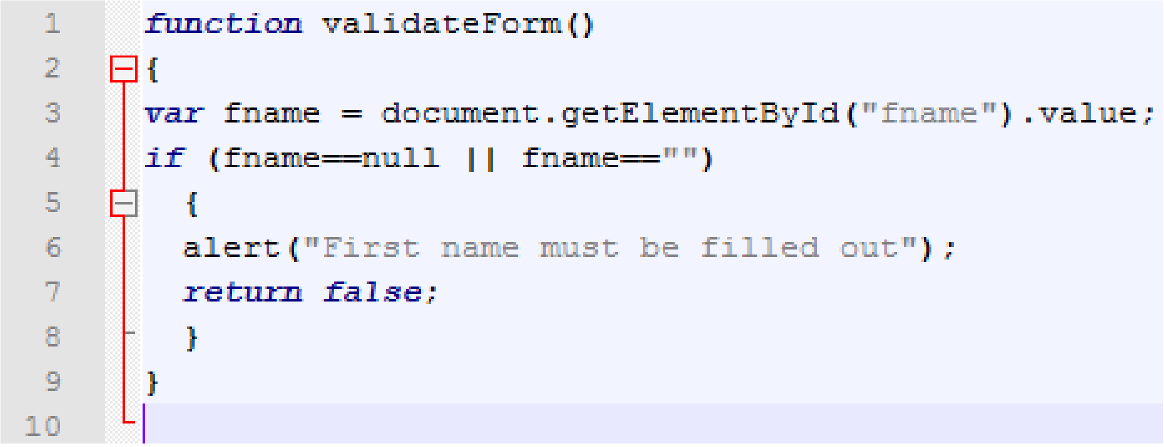
Form Validation Demo



```html
<!DOCTYPE html>
<html>
<head>
    <title>Form Validation Demo</title>
    <script type="text/javascript" src="demo.js"></script>
</head>
<body>
<form name="myForm"  onsubmit="return validateForm()" >
First name: <input type="text" name="fname" id="fname">
            <input type="submit" value="Submit">
</form>
</body>

</html>
```

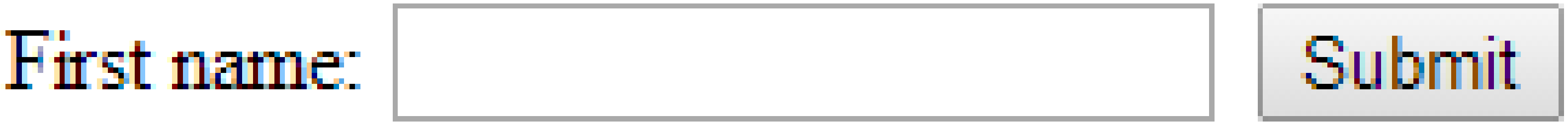
demo.js

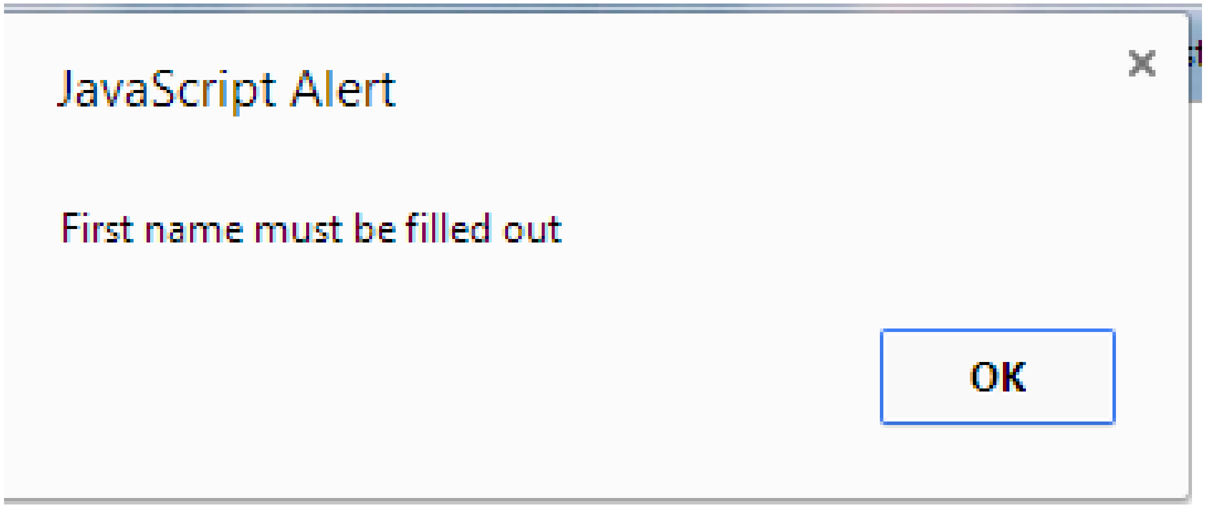


```javascript
function validateForm()
{
var fname = document.getElementById("fname").value;
if (fname==null || fname=="")
  {
  alert("First name must be filled out");
  return false;
  }
}

```

Form Validation




JavaScript Alert

First name must be filled out

DOM 2 Event Model

- DOM Level 2 is defined in modules
- The Events module defines several sub-modules
  - HTMLEvents and MouseEvents are common
- In DOM 2, events are handled explicitly through the event object (while *implicitly* in DOM 1)
- The event object is passed as a parameter to an event handler (which is called listener in DOM 2)
  - Properties of the event object carry information about the event
  - Some event subtypes may extend the interface to include more specific information relevant to the *subtype*
    - For example, a mouse event will include the location of the mouse at the time of the event occurred
      - *DOM level 2* events interfaces form a hierarchy (supporting *subtyping*)

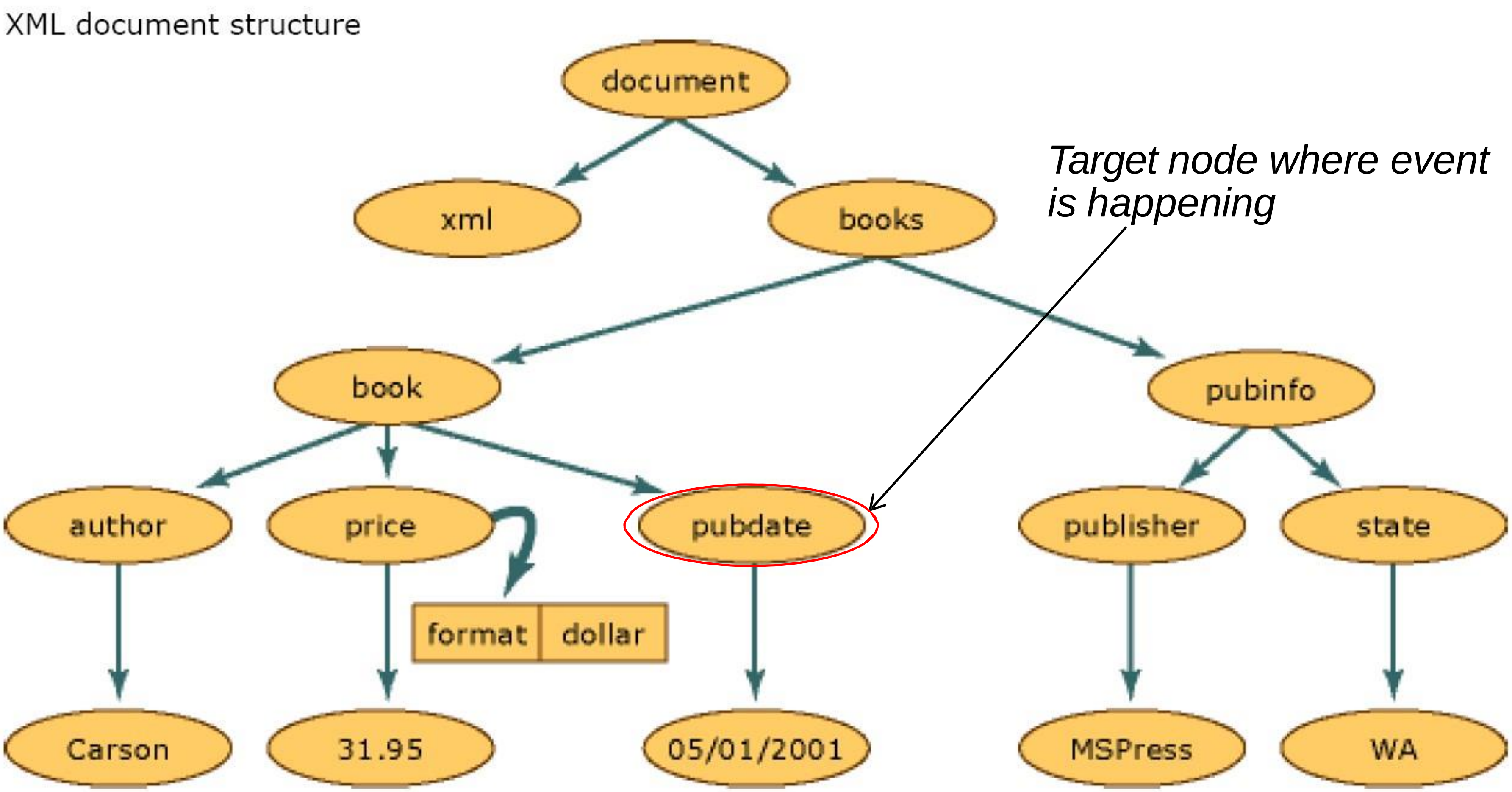
DOM 2 Event Interface Hierarchy

- org.w3c.dom.events.DocumentEvent
- org.w3c.dom.events.Event
    - **org.w3c.dom.events.MutationEvent**
    - **org.w3c.dom.events.UIEvent**
        - **org.w3c.dom.events.MouseEvent**
- org.w3c.dom.events.EventListener
- org.w3c.dom.events.EventTarget

Event Processing Flow

- **DOM 2 defines a process for determining which handlers to invoke for a particular event**
- **The process has three phases for most events**
- **The three phases are regarding the DOM tree structure**
- **In the DOM tree structure, the node where the event happened is called the *target node***
  - **When an event is happening, a corresponding event object representing *that happening event* is created and passed to all registered listeners.**
- [Illustration of the process of event propagation – next slide]

An Example DOM Tree Structure



Target node where event is happening

Event Propagation in DOM 2

- Three traversal phases then occur in turn
- In the *capturing phase* each node from the document root to the target node, in order, is examined.
  - For every node on the way from the root to the target node, if a registered listener for the event is found and enabled, it is invoked immediately
- *Target node phase*: all listeners registered for the event at the target node (if any) are then invoked.
- In the *bubbling phase* each node on the way from the target node to the document root is examined in order
  - If there is a listener registered for the event, whether it is enabled or not, the listener is executed
  - Some event types are not allowed to bubble: load, unload, blur and focus
    - Note "Whether enabled" only affects the capturing phase!

Event Propagation, cont'

- **When a handler is executed, the properties of the event object (as auto-passed parameter) provide the context:**
  - **The `currentTarget` property refers to "the current node" encountered on the way from the root to the event generating node (this is a "moving target")**
  - **The `target` property refers to the node at which the event was originated (there is only one event initiating or generating target)**
- **One major advantage of this scheme over DOM 0 & 1 is that one event can now simultaneously trigger off multiple event handlers to respond to the same event**

Event Handler Registration in DOM 2

- **In DOM 2, listeners must be explicitly registered to node objects using the *addEventListener* method:**
  - **It is a method of any potential event generating object**
  - **It takes three parameters**
    - **A string naming the event type**, e.g., *mouseup*, *submit*, etc.
    - **The listener**: usually the name of a handler/listener function
    - **A Boolean value** specifying whether the handler **enabled**:
      - true **–listener can be called during the capturing phase**
      - false **– the listener cannot be called during capturing phase (but can still be called at the target or the bubbling phases)**
  - **Events and listeners generally have a Many to Many relation**

The `navigator` Object

- Properties of the `navigator` object allow the script to determine the characteristics of the browser in which the script is executing
- The `appName` property gives the name of the browser
- The `appVersion` gives the browser version

DOM Tree Traversal and Modification

- **Each element in an XHTML document has a corresponding object in the DOM representation**
- **The ELEMENT’s object has methods to support:**
  - **Traversing the DOM tree – visiting each of the document nodes in a certain way, e.g., to left, right, up, or down, and jumping …**
  - **Modifying the document**
    - **For example, removing and inserting child nodes**

DOM Tree Traversal

- Various properties of an `Element` object are references to related nodes (objects):
  - `parentNode` references the parent node of the `Element`
  - `previousSibling` and `nextSibling` connect the children of a node into a list
  - `firstChild` and `lastChild` reference children of an Element
  - …

DOM Tree Traversal, cont’d

- **For example, an unordered list with** id = myList**, the following code accesses its list items:**

```
    var dom = document.getElementById("myList");
    var listItems = dom.childNodes.length;
    document.write("Number of list items is: " +
                    listItems + "<br />");
```

- **Read more about traversal at**

  http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html

DOM Tree Modification Methods

- insertBefore(newChild, refChild)
- replaceChild(newChild, oldChild)
- removeChild(oldChild)
- appendChild(newChild)

More details are left for interested students to explore themselves again at

http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html

Characters and Character-Classes

- ***Metacharacters*** **have special meaning in regular expressions**
  - **\ | ( ) [ ] { } ^ $ * + ? .**
  - **These characters may be used literally by escaping them with ‘\’**
- **Other characters represent *themselves***
- **A period matches any single character**
  - **/f.r/ matches *for* and *far* and *fir* but not *fr***
- **A *character class* matches one of a specified set of characters**
  - **[*character set*]**
  - **List characters individually: [abcdef]**
  - **Give a range of characters: [a-z]**
  - **Beware of [A-z]**
  - **^ at the beginning negates the whole class**

Predefined character classes you can use

Name	Equivalent Pattern	Matches
\d	[0-9]	A digit
\D	[^0-9]	Not a digit
\w	[A-Za-z_0-9]	A word character (alphanumeric)
\W	[^A-Za-z_0-9]	Not a word character
\s	[\r\t\n\f]	A whitespace character
\S	[^ \r\t\n\f]	Any non-whitespace character

Sub-pattern repetition

- A pattern can be repeated for a *fixed number* of times by following it with a pair of braces enclosing a count
- Besides, a pattern can be repeated by the following special characters:
  - `*` indicates zero or more repetitions of the previous pattern
  - `+` indicates one or more of the previous pattern
  - `?` indicates zero or one of the previous pattern
- Examples
  - `/\(\d{3}\)\d{3}-\d{4}/` might represent a phone number
  - `/[$_a-zA-Z][$_a-zA-Z0-9]*/` matches identifiers

Anchors

- **Anchors in regular expressions match *positions* rather than characters**
    - **Anchors are 0 width and may not take multiplicity modifiers**
- **Anchoring to the beginning/end of a string**
    - **^ at the beginning of a pattern matches the beginning of a string**
    - **$ at the end of a pattern matches the end of a string**
        - **Note: The $ in /a$b/ matches a $ character**
- **Anchoring at a word boundary**
    - **\b matches the position between a word character and a non-word character or the beginning or the end of a string**
    - **/\bthe\b/ will match ‘the’ but not ‘theatre’ and will also match ‘the’ in the string ‘one of the best’**

Pattern Modifiers

- Pattern modifiers are specified by characters following the closing slash “/” of a pattern
- Modifiers modify the way a pattern is interpreted/used
- The x modifier causes to ignore whitespaces in the pattern
    - This allows better (convenient) formatting of the pattern
    - \s as a defined class still retains its meaning, unaffected!
- The g modifier indicates “global” (explained later)

Other Pattern Matching Methods

- The *replace* method takes a pattern parameter and a string parameter
  - It replaces a match in the target string with the second parameter
  - A g modifier on the pattern causes multiple replacements
- *Parentheses* in patterns mark *sub-patterns*
  - The pattern matching machinery will remember the parts of matched substrings that correspond to sub-patterns
- The *match* method takes one pattern parameter
  - Without a g modifier, the return is an array of the (one whole) match and parameterized sub-matches
  - With a g modifier, the return is an array of all (only) whole matches
- The *split* method splits the object string using the pattern that specifies the split points