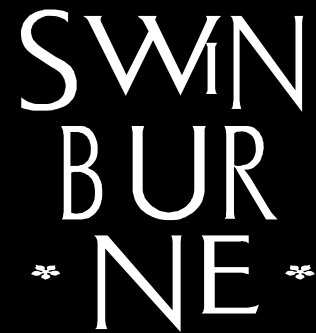




Web Application Development: Client-Side Processing – Java Script and DOM

Week 5



SWINBURN
UNIVERSITY OF
TECHNOLOGY

Content of this Lecture

- Core JavaScript
- Object-oriented JavaScript
- The Document Object Model (DOM)
- JavaScript and Events

Ajax (slide from Lecture 1)

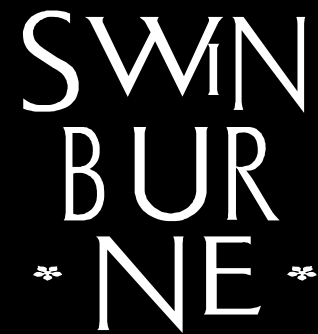
- Various techniques have been devised to resolve some of the issues of the classic web model
- The aim is to allow **dynamic change** in the interface presented to the user, and to allow the user to **continue interacting** with the system whilst the server is busy responding to an earlier request
- “AJAX” – **A**synchronous **J**avaScript **a**nd **X**ML – now refers to a collection of technologies that allow the above.

Client-Side

- Basically, in a web-based system, the client deals with the view and the controller in the MVC pattern; it can also engage in model-based processing (when a “thick-client” approach is used)
- The amount of processing to be done on the client has to be carefully assessed
- Usually, **DOM** (Document Object Model) is used to access and update the XHTML document hosted by a browser
- As such it can be manipulated using **JavaScript** , with the language directly giving access to the DOM features
- JavaScript allows us to program how the system will respond to user-initiated (and other) events, making use of **event-handlers**
- This lecture revises material dealing with these issues



Java Script Refresher



SWINBURN
UNIVERSITY OF
TECHNOLOGY

Javascript – Some Advice

- There are several JavaScript tutorials on the internet – for example <http://www.w3schools.com/JS/default.asp>
- There are several books available in the library
 - JavaScript Step by Step by Steve Suehring 2nd edition 2008 (416 pages) {on-line}
 - Advanced JavaScript, Third Edition by Chuck Easttom 2008 {on-line}
- Also refer back to JavaScript section in notes for “Internet Technologies” (provided on Canvas in the “Additional Material” folder in the Lectures section.)

JavaScript

- Not really related to Java, except that like Java, it inherits a lot of syntax and conventions from C.
- JavaScript is **object-based** – it supports objects, essentially supports classes, but does not have inheritance.
- JavaScript was designed to be a client-side web scripting language. JavaScript scripts (or programs) are interpreted by a JavaScript interpreter running in a web browser.
- A JavaScript program running in a browser knows about the browser and the document(s) contained in it, as defined objects that can be referenced, as variables with standard names, in any script.
- However, old browsers may not support the current version of the language.

JavaScript

- Within an HTML document, JavaScript code is contained within “script” tags.
- The JavaScript can be in-line, or the script tag can refer to an external JavaScript file.
- Generally we use separate files, although in some examples in these notes we use in-line code for the purposes of easier illustration of principles.

Syntax

- Basically inherited from C, Java
- Case-sensitive, e.g., pay attention to variable names
- Semicolons are optional (but recommended) for ending statements. **We adopt a STANDARD that mandates them!**
- Comments – are as in Java
 - // single-line comment
 - /* line 1 comments, ...
line n comments */

Variables & Types

- `var myBook;`
- `var myBook = "Beginning Ajax";`
- `myBook = "XML";`
- Primitive datatypes for variables to indicate the kind of data hold
 - ☐ Number (including integer, double, etc.)
 - ☐ String
 - ☐ Boolean (true, false)
 - ☐ undefined (a declared non-reference variable that has not been given a value has the undefined value)
 - ☐ null (an object reference variable has the value null when it does not refer to any object)
- Dynamic typing – the data type is determined at run time (compare with Java or C or C++, where variables have to be declared with a type (or class), and then the type is fixed)
 - ☐ `myBook = 123; // data type changed from String to Number`

Reference Datatypes

- Contain a reference to the a place of memory that holds the data
- Object type is a reference type, e.g., array

```
function addFirst(a) {  
    a[0] = a[0] + 5;  
}  
var myArray = [10, 20, 30];  
addFirst(myArray);  
alert("myArray is changed to " + myArray);
```

Operators – as in C, Java

- Assignment operator: **=**
- Arithmetic operators: **+**, **-**, *****, **/**, **%** (remainder)
- Comparison operators
 - **>**, **<**, **>=**, **<=**, **==**, **!=**
 - **===** (equal and the same type)
 - **!==** (not equal or not the same type)
- Logical operators: **&&** (AND), **||** (OR), **!** (NOT)
- Increment and decrement operators: **++**, **--**

Conditional Statements

■ if statement

- if (condition) {statements}
- if (condition 1) {
 ...
}
 else if (condition 2) {
 ...
 }
 else {
 ...
 }

■ switch statement

- switch (myVar) {
 case 'value 1': statements; break;

 case 'value n': statements; break;
 default: statements;
}

Loops

■ Four components

- ☐ Initialisation
- ☐ Test condition
- ☐ Iterator (counter)
- ☐ Loop body (optional *break* for jumping out and *continue* for getting into next iteration immediately)

■ Three forms (for loop, while loop, do-while loop)

```
for (var a=2; a<1000; a=a*2) { document.write("a = "+ a + "<br>"); }
```

```
var a=2;
while (a<1000) {
  document.write("a = "+ a + "<br>");
  a=a*2;
}
```

```
var a=2;
do {
  document.write("a = "+ a + "<br>");
  a=a*2;
} while (a<1000) ;
```

Functions

- A function contains code that will be executed by a call to that function (which can be a direct call, or a **call-back** when an event (eg a button click in the UI) occurs)
- A function has
 - a name (although there are **anonymous** un-named functions too, used for call-backs)
 - a body
 - optional parameters (input)
 - optional returned value (output)
- A function may cause side effects (ie affect global variables)

Function Examples

```
function displayMsg()  
{  
  alert("Hello World!")  
}
```

```
function displayMsgWithInput(subject)  
{  
  alert("I love " + subject + "!")  
}
```

```
function total(a) {  
  var r = 0;  
  for (var i=0; i<a.length; i++) {  
    r = r + a[i];  
  }  
  return r; // returns a value  
}
```

```
function doubleAll(a) {  
  for (var i=0; i<a.length; i++) {  
    a[i] = a[i]*2; // changes parameter  
  }  
}
```


Object-oriented JavaScript

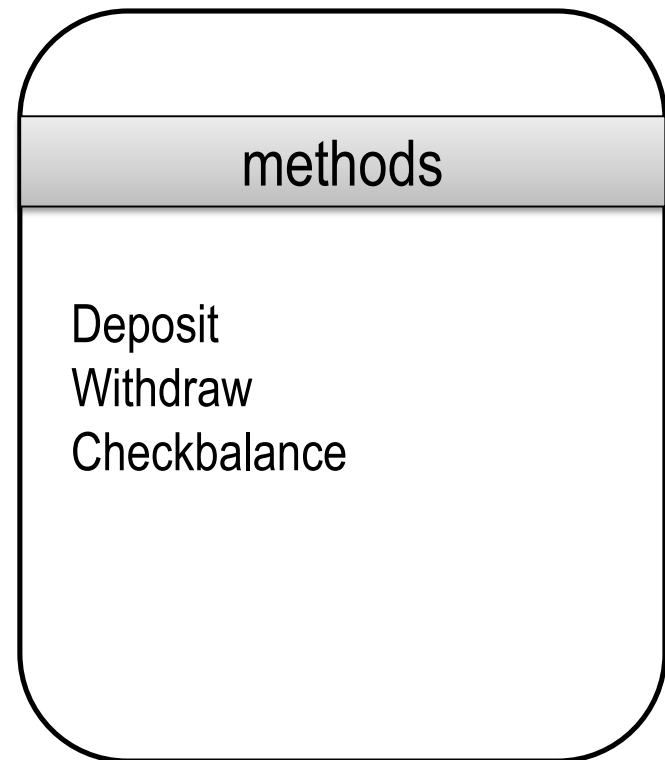
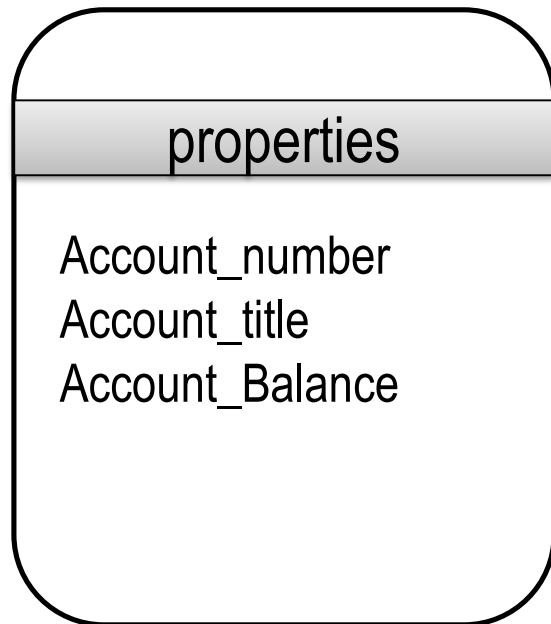
- JavaScript is actually object-based, supporting a collection of properties and methods for an object, **but no inheritance**
- There are four types of objects
 - Built-in objects
 - Browser objects – these refer to objects associated with the browser
 - Document objects – objects in the current document (see DOM)
 - User-defined objects
- User-defined objects let us model complex data types – sometimes necessary to hold data retrieved from the server

Object-oriented JavaScript

- Why should we care about objects?
 - We shall see that JavaScript objects are a convenient way to handle structured data sent from the server in the JSON format
 - We can bundle data structure and operations together, as in object-oriented languages, helping us to structure programs
 - Document and browser objects are “objects”, and need to be manipulated, so we need to understand the way to manipulate objects in JavaScript.
- In JavaScript, objects have properties (instance variables in Java) and methods (as in Java)
- They are accessed via the same “dot notation” as in Java

Example: Bank Account

■ properties and methods of BankAccount



simpleajax.js (in Lecture 1) – object: **xhr**.

```
var xhr = new XMLHttpRequest();  
xhr.open("GET", url, true);  
xhr.onreadystatechange = function() {  
    if (xhr.readyState == 4 && xhr.status == 200) {  
        place.innerHTML = xhr.responseText;  
    } // end if  
} // end anonymous call-back function  
xhr.send(null);
```

Object-oriented JavaScript

- JavaScript is actually object-based, supporting a collection of properties and methods for an object, **but no inheritance**
- There are four types of objects
 - Built-in objects
 - Browser objects – these refer to objects associated with the browser
 - Document objects – objects in the current document (see DOM)
 - User-defined objects
- User-defined objects let us model complex data types – sometimes necessary to hold data retrieved from the server

Built-in Objects

- Constructor method: *new*
 - `var myArray = new Array();` -- creates new array and assigns it to variable myArray
- String
 - `str.length`, `str.toUpperCase()`, `str.blink()` -- length is a property; others are methods
 - http://www.w3schools.com/jsref/jsref_obj_string.asp
- Date
 - `Date()`, `d.getDate()`, `d.setYear(2006)`, `d.toString()`
 - http://www.w3schools.com/jsref/jsref_obj_date.asp
- Math – a single object with many methods
 - `Math.abs(x)`, `Math.floor(y)`, `Math.random()`
 - http://www.w3schools.com/jsref/jsref_obj_math.asp

Built-in Object - Array

■ Defining an array

- ☐ `var myUnits=new Array();`
- ☐ `var myUnits2=new Array(3);`
- ☐ `var myUnits3=new Array("WAD", "Ajax", "XML");`

■ Adding values

- ☐ `myUnits[0] = "WAA"; myUnits[1] = "WS";`

■ Properties and methods

- ☐ `myUnits.length;`
- ☐ `myUnits.reverse(); myUnits.sort(); myUnits.push(); myUnits.pop();`

■ http://www.w3schools.com/jsref/jsref_obj_array.asp

Browser and Document Objects

■ JavaScript only runs in a browser

- When it runs, various objects relating to the browser that JavaScript is running in, and the HTML document that resides in the browser are automatically available, and can be used in scripts
- Using JavaScript commands, we can manipulate these objects, and consequently update the interface that is presented to the user; we can access browser windows, documents resident in these windows, the browsing history, and so on, thus allowing broad programmer control over the user-interface

Browser Objects

- Browser Object Model (BOM)
- A collection of objects, that interact with the browser window.
 - **window**: top level object in the BOM hierarchy
 - **history**: keep track of every page the user visits
 - **location**: contains the URL of the page
 - **navigator**: contains information about the browser name and version
 - **screen**: provides information about display characteristics
 - **document**: belongs to both BOM and DOM

Window Object Properties (Part)

Property	Description
closed	Returns whether or not a window has been closed
document	See Document object in DOM
history	See History object
length	Sets or returns the number of frames in the window
location	See Location object
name	Sets or returns the name of the window
opener	Returns a reference to the window that created the window
outerheight	Sets or returns the outer height of a window
pageXOffset	Sets or returns the X position of the current page in relation to the upper left corner of a window's display area
parent	Returns the parent window
self	Returns a reference to the current window
top	Returns the topmost ancestor window

Window Object Methods (Part)

Method	Description
alert()	Displays an alert box with a message and an OK button
blur()	Removes focus from the current window
close()	Closes the current window
confirm()	Displays a dialog box with a message and an OK and a Cancel button
createPopup()	Creates a pop-up window
focus()	Sets focus to the current window
moveBy()	Moves a window relative to its current position
moveTo()	Moves a window to the specified position
open()	Opens a new browser window
print()	Prints the contents of the current window
prompt()	Displays a dialog box that prompts the user for input
setTimeout()	Evaluates an expression after a specified number of milliseconds

History and Location Objects

■ History (wobj.history)

- ☐ `history.length`; // the number of elements in the history list
- ☐ `history.back()`; // loads the previous URL in the history list
- ☐ `history.forward()` ; // loads the next URL in the history list

■ Location (wobj.location)

- ☐ `location.href`; // sets or returns the entire URL
- ☐ `location.pathname`; // sets or returns the path of the current URL
- ☐ `location.assign(url)`; // loads a new document
- ☐ `location.reload()`; // reloads the current document
- ☐ `location.replace(url)`; // replaces the current document with a new one

User-Defined Objects

- There are several ways to define objects in JavaScript
 - ❑ Using a “constructor” function, where objects are formed by use of the keyword “new”. This way allows construction of several similar objects, and so is in some sense like defining a data type.
 - ❑ Using the “Object” constructor, and then assigning properties and methods to the object. This way just the one object is constructed, and this approach does not define a data type.
 - ❑ Using an “object literal”
 - ❑ Using an object prototype (the way most similar to type or class definition in other languages)

User-defined Objects using Constructor Function

```
var Car = function() {  
    this.wheels = 4; // specifies a property "wheels"  
    this.start = function() {  
        alert( "Vroom!" );  
    }; // defines a method called "start"  
};  
var myVW = new Car(); // declares a new Car object  
myVW.start();  
alert(myVW.wheels);
```

We can add a new property to an object after declaration :

```
myVW.engineSize = 2000;
```

■ note that this does not alter the definition of Car; other Cars that are constructed do not have an engineSize

■ We can destroy an object : myVW = null;

User-defined Objects using Constructor Function

- Note that this way of defining an object involves creating a new object every time “new” is invoked.
- Each object so defined has its own version of each method. *The method code is not shared between the different objects created.*
- This differs from the typical definition of a data type or a class in other languages.
- We will see later that there is a way to have the method code shared between all objects of a defined “type”.
- The ability to add properties to objects after their initial definition is totally unlike the situation found in languages like Java.

User-defined Objects using Object Constructor

- We can also create a user-defined object with the Object() constructor
 - `var member = new Object();` // this does NOT define a type, just a single // object, and at this stage, there are no properties or methods
- After this we can add properties
 - `member.name = "Donald Duck";`
`member.email = "dd@gmail.com";`
`member.isRegistered = true;`
- and we can define methods, first just defining a separate function
 - `function showMe() {alert("I'm here!");}` // at this stage, not yet a method
- and then assigning the function as a property of an object (in other words, in a sense a method IS a property)
 - `member.present = showMe;` // now assigned as a method of member
 - `member.present = function() {alert("I'm here!");}` // alternative approach
 - Later we can call `member.present();`

User-defined Objects using an Object Literal

- We can also create a new object by using an object literal, which can even include the code for a method

```
var member =  
{ name: "Donald Duck",  
  email: "dd@gmail.com",  
  isRegistered: true,  
  present: function () {alert("I'm here!");}  
};
```

- This way of representing an object lies at the heart of the JSON notation, which we will see is very useful in Ajax-based systems.

User-defined Objects using Prototypes

- Add new properties/methods

- Every object is based on a prototype object

```
function member (name, age)
```

```
{ this.name = name;
```

```
  this.age = age;
```

```
}
```

```
member.prototype.present = function () {
```

```
  alert("I'm here!");
```

```
}; // the method is defined and shared for all member objects
```

- We can also use a prototype to specify a property with a default value

```
member.prototype.isActive = true;
```

e.g. `var person1= new member("Peter", 30);`

```
person1.present();
```

```
alert(person1.name);
```

```
alert(person1.isActive);
```

contrast this with Slides 27-28, where we show an object defined using a constructor function.

Example – Creating and Using Objects

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  <title>Address Book</title>
  <style type="text/css">
    .ab {
      font-family: Verdana, Arial, Helvetica, sans-serif;
      font-size: small;
      color: #993300;
      background-color: #CCFFCC;
      padding: 5px;
      height: 100px;
      width: 350px;
      border: thin dotted #993300;
    }
  </style>
```

**In this example we use embedded
css and JavaScript for simplicity of
presentation only**

Example – Creating and Using Objects

```
<script type="text/javascript">
  function addressBookItem (fname, lname, email) {
    this.fname= fname;
    this.lname = lname;
    this.email = email;
  }

  addressBookItem.prototype.write = function() {
    var adrbook = "<p class='ab'>First Name: " + this.fname + "<br />";
    adrbook += "Last Name: " + this.lname + "<br />";
    adrbook += "Email Address: " + this.email + "</p>";
    document.write(adrbook);
  }
</script>
```

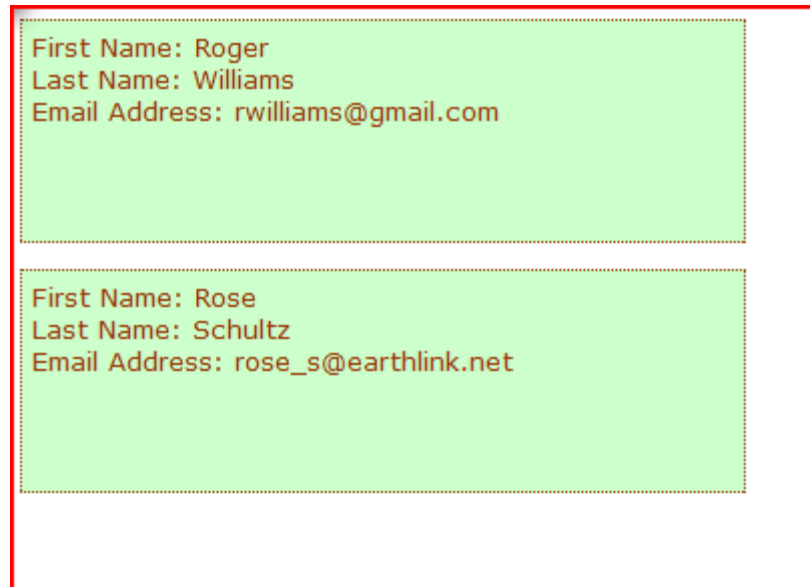
Here we are defining a “type” to represent a BookItem, which stores 3 pieces of data, and has a method that writes the data for the book into the browser document.

Example – Creating and Using Objects

```
</head>
<body>
  <script type="text/javascript">
    var aB1 = new addressBookItem('Roger', 'Williams', 'rwilliams@gmail.com');
    var aB2 = new addressBookItem ('Rose', 'Schultz', 'rose_s@earthlink.net');
    aB1.write();
    aB2.write();
  </script>
</body>
</html>
```

Here, in a script in the body of the browser document – which executes as soon as the body loads into the browser, we create two addressBookItem objects (in syntax similar to Java, or C#), and then call the method write on each of them.

When the HTML file is loaded



First Name: Roger
Last Name: Williams
Email Address: rwilliams@gmail.com

First Name: Rose
Last Name: Schultz
Email Address: rose_s@earthlink.net

<https://mercury.swin.edu.au/wlai/Lec5/example1.htm>



The Document Object Model (DOM)

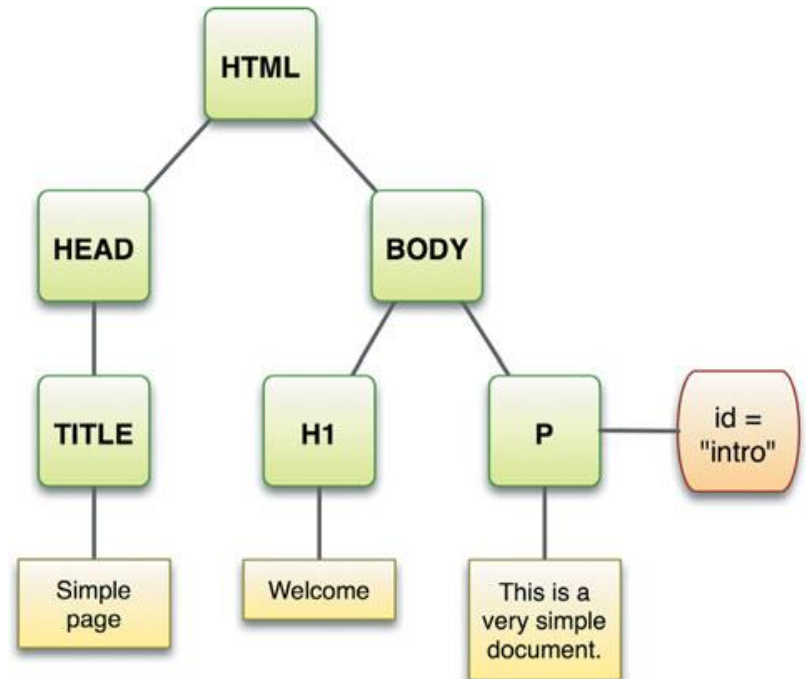
Document Object Model (DOM)

- W3C recommendation for advanced processing of both XML and HTML/XHTML documents
 - <http://www.w3.org/DOM/>
- The DOM is an API (application programming interface)
 - It is platform- and language-neutral (accessible in JavaScript, but also on a server using PHP, or other languages)
 - As a data structure, it has a tree-based structure
- It is designed to allow programs and scripts to dynamically *access* and *update* documents
 - Content (i.e. data in any XML/HTML document)
 - Structure (i.e. nesting of elements or element names)
 - Style (i.e. style sheets, XPath expressions ...)

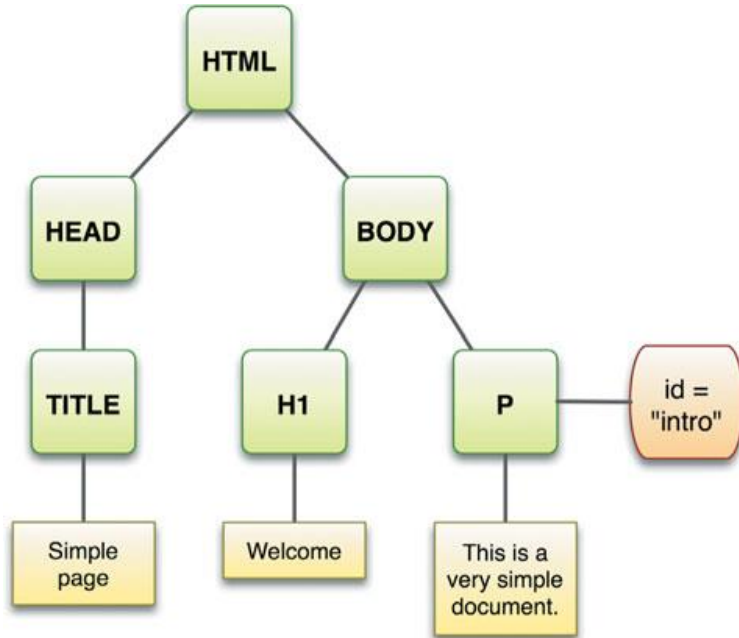
DOM Example – Tree Representation

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head><title>Simple page</title></head>
  <body>
    <h1>Welcome</h1>
    <p id="intro">This is a very simple document.</p>
  </body>
</html>
```

Note that because XHTML has all elements properly nested, the graph of the structure will be a tree



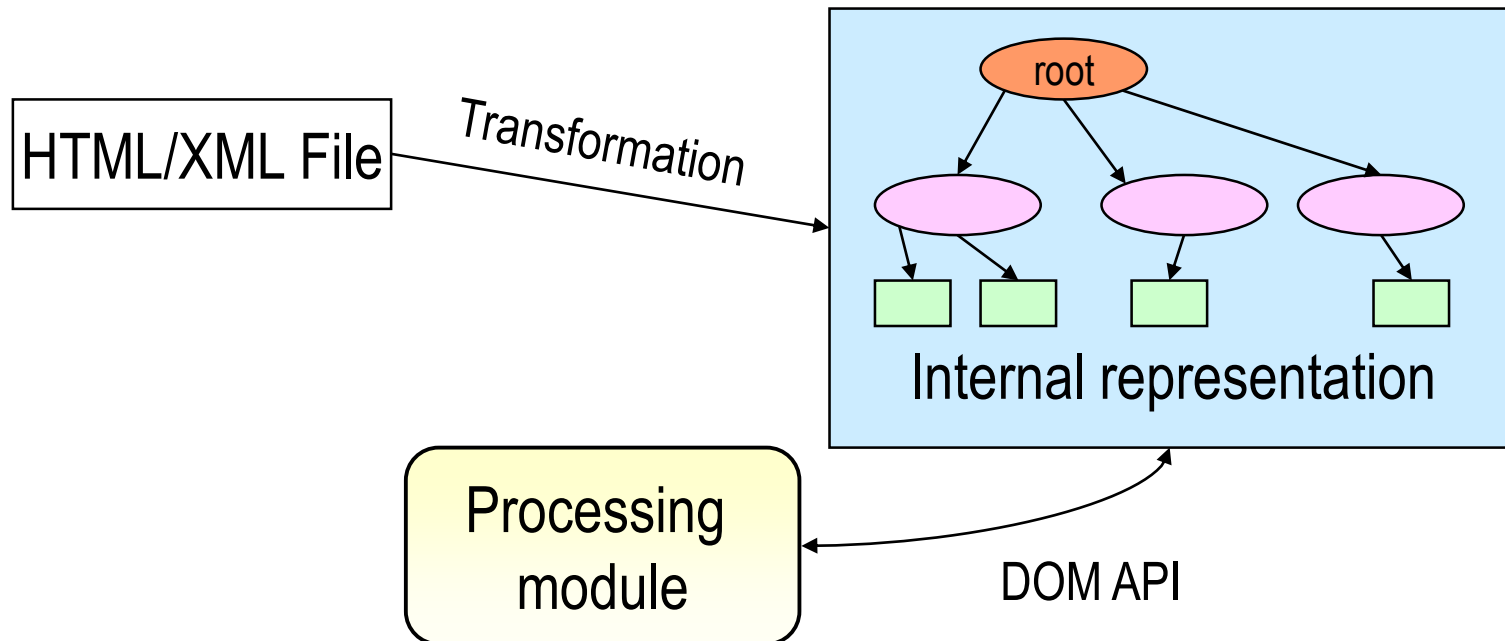
DOM Example



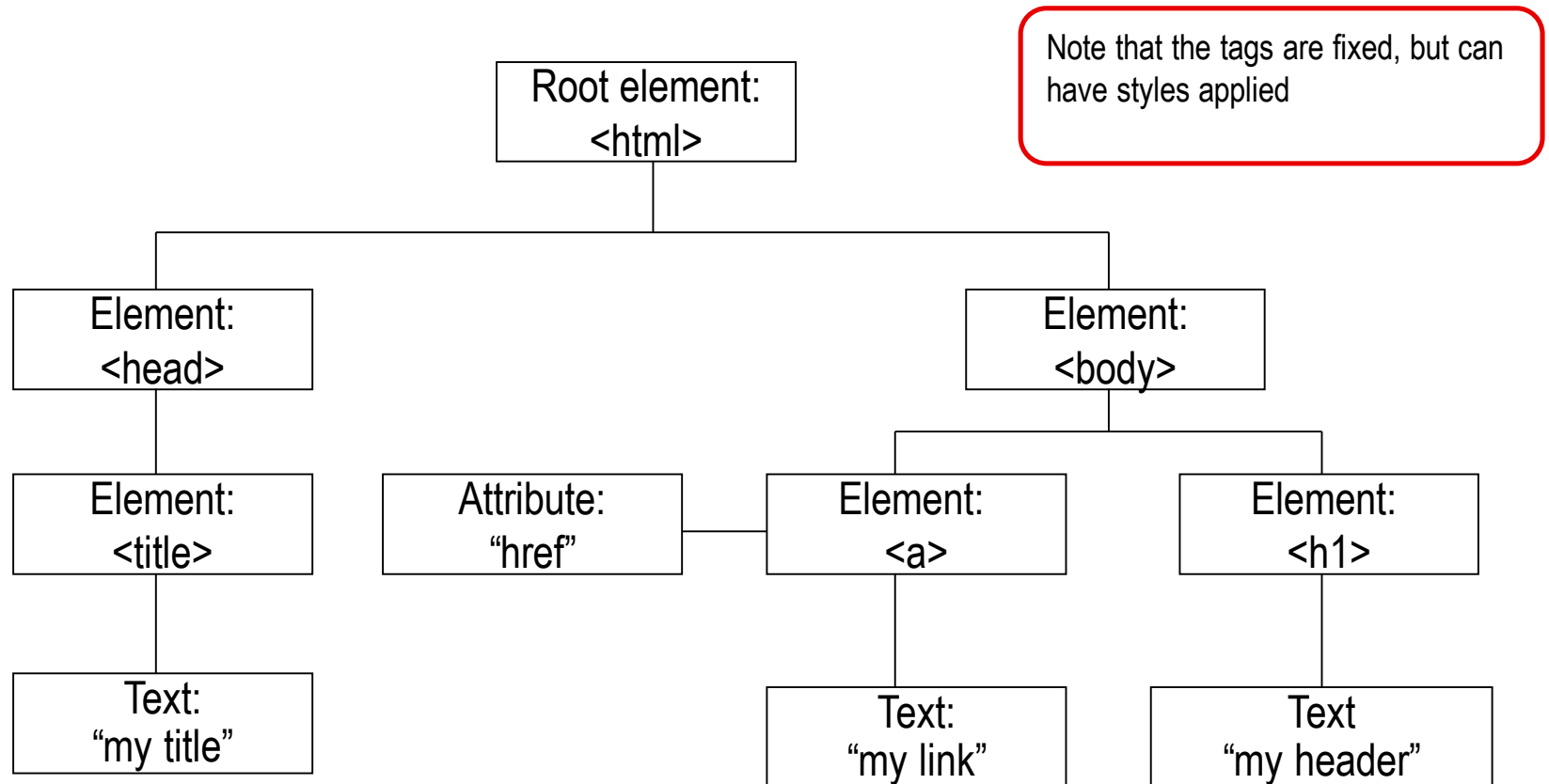
- The html element is the **parent** of the head element and the body element.
- The head and body elements are **siblings**.
- The head element is the **parent** of the title element.
- The title element is a **child** of the head element.

DOM – The Key Concept

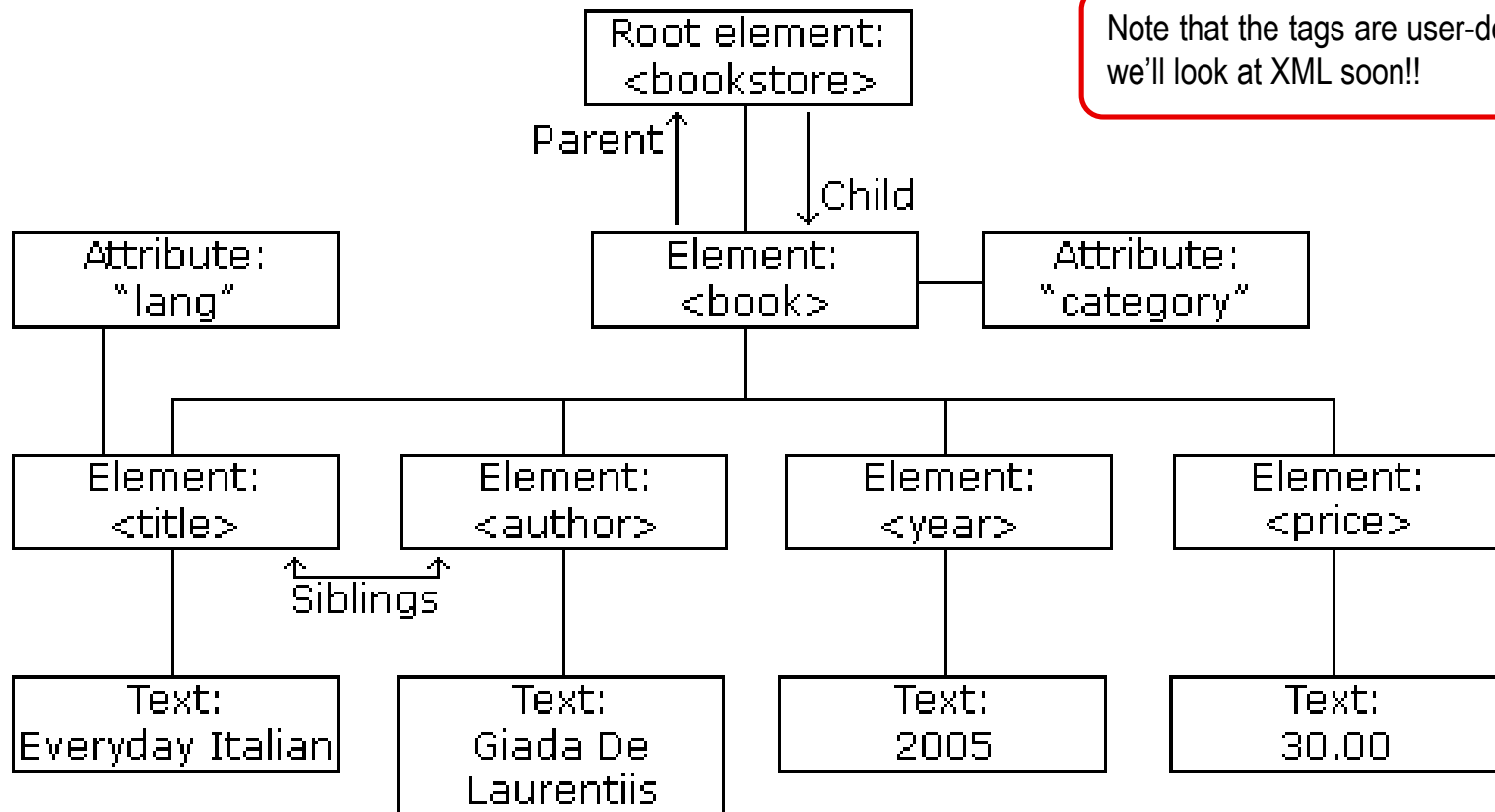
- Input: An HTML(XHTML)/XML document
- Internal representation: An **in-memory** "tree" data structure that can be accessed by JavaScript
- Access mode: A set of standard API calls



An HTML Tree



An XML Tree



DOM – Sample Code

```
// get list of all HTML body elements from document
var bodyels = document.getElementsByTagName("body");
// There is only one body element in a HTML 4.0 doc
// So bodyels will be a list with just one element
// We thus access the document body as follows
var docbody = bodyels.item(0);
// Set the CSS2 color property for the body to pink
docbody.style.color = "pink";
```

DOM Interfaces

- Everything in a document is a node
 - The entire document is a document node - **document**
 - Every HTML tag is an **element** node
 - The texts contained in the HTML elements are **text** nodes
 - Every HTML attribute is an **attribute** node
 - Comments are **comment** nodes

Element Type	Node Type
Element	1
Attribute	2
Text	3
Comment	8
Document	9

DOM Interfaces (Cont'd)

- A **Node Interface** is used to read and write the individual elements in the HTML (or XML) node tree.
- The **childNodes** property of an **element** node (including the **documentElement**) together with a loop construct can be used to enumerate each individual node for processing.
- Main functions of the DOM API
 - To traverse the node tree
 - To access and maybe change the nodes and their attribute values
 - To insert and delete nodes

Document Object Methods

- **createAttribute**(attributeName) creates an attribute node with the specified attribute name
- **createElement**(tagName) creates an element with the specified tagName
- **createTextNode**(text) creates a text node, containing the specified text
- **getElementsByTagName**(tagName) returns a (node) list of all nodes with the specified tag name
- **getElementsByName**(name) returns a list of all nodes with the specified name attribute
- **getElementById**(id) returns the node with the specified id attribute

Node Object Properties

- **attributes** returns a NamedNodeMap containing all attributes for this node
- **childNodes** returns a NodeList containing all the child nodes for this node
- **firstChild** returns the first child node for this node.
- **lastChild** returns the last child node for this node
- **nextSibling** returns the next sibling node. Two nodes are siblings if they have the same parent node
- **nodeName** returns the nodeName, depending on the type
- **nodeType** returns the nodeType as a number
- **nodeValue** returns, or sets, the value of this node, depending on the type
- **parentNode** returns the parent node for this node
- **previousSibling** returns the previous sibling node. Two nodes are siblings if they have the same parent node

Node Object Methods

- **appendChild(newChild)** appends the node newChild at the end of the child nodes for this node
- **cloneNode(boolean)** returns an exact clone of this node. If the boolean value is set to true, the cloned node contains all the child nodes as well
- **hasChildNodes()** returns true if this node has any child nodes
- **insertBefore(newNode,refNode)** inserts a new node, newNode, before the existing node, refNode
- **removeChild(nodeName)** removes the specified node, nodeName
- **replaceChild(newNode,oldNode)** replaces the oldNode, with the newNode
- **setAttributeNode(newAttr)** insert the new attribute newAttr to the node

Examples

- `document.getElementsByTagName("myElement")[0].parentNode`
- `document.getElementById("myId").parentNode`
- `document.getElementById("myId").childNodes[i]`
`document.getElementById("myId").childNodes.item(i)`
- `document.getElementById("myId").firstChild`
- `document.getElementById("myId").lastChild`
- `document.getElementsByTagName("myElement")[0]`
`document.getElementsByTagName("myElement").item(0)`
- `document.getElementsByTagName("myElement").item(`
`document.getElementsByTagName("myElement").childNodes.length - 1)`

Adding New Element to an Existing Page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Adding New Elements</title>
<style type="text/css">
body {
    font-family: Verdana, Arial, Helvetica, sans-serif;
    font-size: small;
    font-weight: bold;
}
.newDiv {
    background-color:#ffcccc;
}
</style>
```

**In this example we use embedded
css and JavaScript for simplicity of
presentation only**

Adding New Element to an Existing Page (Cont'd)

```
<script type="text/javascript">
function newOne() {
var newEl = document.createElement('p');
var newTx = document.createTextNode('This is the new paragraph.');
```

newEl.appendChild(newTx);

```
var newPara =document.getElementById('newText'); newPara.appendChild(newEl);
}
function newColor() {
document.getElementById('newText').style.color="red";
}
</script>
</head>
<body><p>This is the first paragraph on the page.</p>
<div class="newDiv" id="newText"></div>
<p>This is the next paragraph on the original page.</p>
<form>
<input type="button" value="Add a new paragraph" onclick="newOne()" /><br /><br />
<input type="button" value="Make it red" onclick="newColor()" />
</form>
</body>
</html>
```

The innerHTML Property

```
function newOne() {  
    // create a new p element to hold the new paragraph text  
    var newEl = document.createElement('p');  
    // create a new text node with the text 'This is the new paragraph'  
    var newTx = document.createTextNode('This is the new paragraph.');
```

// append the new text node to the new p element – ie, make it a child of the new p element

```
    newEl.appendChild(newTx);  
    // get the (existing) document element that has id 'newText'  
    var newPara = document.getElementById('newText');
```

//append the new p element to this

```
    newPara.appendChild(newEl);  
}
```

```
function newOne() {  
    var newPara = document.getElementById('newText');  
    newPara.innerHTML = "<p>This is the new paragraph.</p>";  
} // shorter, faster but not part of any standard
```



Java Script Event Model

JavaScript and Event Models

- Events are necessary to create interaction between JavaScript and HTML
- Events occur when either a user (eg clicks a button) or the browser does something (eg loads a page)
- We have also seen the `onReadyStateChange` event for the XHR object used in basic Ajax processing
- We can bind an event to an event handler (or “call-back” function) by using HTML attributes; the handler is the function that is called automatically when the event occurs.

```
<input type="button" value = "Click Me" onclick="myResponse()">
```

A diagram with two red boxes at the bottom labeled 'event' and 'handler'. Red lines connect the 'event' box to the 'onclick' attribute in the code snippet above, and the 'handler' box to the 'myResponse()' function value.

event


handler

Attribute	The event occurs when...
onabort	Loading of an image is interrupted
onblur	An element loses focus
onchange	The user changes the content of a field
onclick	Mouse clicks an object
ondblclick	Mouse double-clicks an object
onerror	An error occurs when loading a document or an image
onfocus	An element gets focus
onkeydown	A keyboard key is pressed
onkeypress	A keyboard key is pressed or held down
onkeyup	A keyboard key is released
onload	A page or an image is finished loading
onmousedown	A mouse button is pressed
onmousemove	The mouse is moved
onmouseout	The mouse is moved off an element
onmouseover	The mouse is moved over an element
onmouseup	A mouse button is released
onreset	The reset button is clicked
onresize	A window or frame is resized
onselect	Text is selected
onsubmit	The submit button is clicked
onunload	The user exits the page

Event Models

- Event bubbling: an event fires from the most-specific target to the least-specific target.
 - mouseover fires child element (p) first
 - If there is no such event in the child, but one in the parent, the parent event fires
- Event capturing: an event fires from the least-specific target to the most-specific target.
 - mouseover fires parent element (div) first

```
<div>  
  <p>Event order</p>  
</div>
```



```
addEventListener(event, function, useCapture); //The default value is false  
true/false
```

Event Registration

- **Inline event registration** by using HTML attributes
``
- **Traditional event registration**
`var myElement = document.getElementById('1stpara');`
`myElement.onclick = startNow;`
to remove `myElement.onclick = null;`
- **IE event registration // IE8 & earlier versions**
`var myElement = document.getElementById('1stpara');`
`myElement.attachEvent('onclick', startNow);`
`myElement.attachEvent('onclick', startNow2); // additional`
to remove `myElement.detachEvent('onclick', startNow);`
- **W3C DOM event registration**
`var myElement = document.getElementById('1stpara');`
`myElement.addEventListener('click', startNow, false);`
`myElement.addEventListener('click', startNow2, false); // additional`
to remove `myElement.removeEventListener('click', startNow, false);`

Event Registration - Example

This example displays a couple of paragraphs.

If you click on the FIRST paragraph, the background color changes, and a border appears round the paragraph.

If you click on the button, the paragraph changes back to its original state, and the click functionality is removed.

This is achieved by

- attaching an event-handler to the click event for the first paragraph, which causes the background and border change
- attaching an event-handler to the button that causes the first paragraph to have its background and border changed back, and which also removes the event-handler for the click event for the first para.

<https://mercury.swin.edu.au/wlai/Lec5/example3.htm>

Event Registration - Example

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Event Registration</title>
<script type="text/javascript">
function addHandler () {
    var addH = document.getElementById('p1');
    if (addH.addEventListener) {
        addH.addEventListener('click', addBord, false);
    }
    else if (addH.attachEvent) { // IE8 & earlier versions
        addH.attachEvent('click', addBord);
    }
}
```

In this example we use embedded CSS and JavaScript for simplicity of presentation only

Here we need code to handle browser event-model differences – we ‘ask’ if a particular “add event” method exists, and if it does, this identifies the browser and we call the method.

Event Registration - Example

```
function detHandler () {  
    var detH = document.getElementById('p1');  
    detH.style.border = "";  
    detH.style.backgroundColor = 'ffffff';  
    if (detH.removeEventListener) {  
        detH.removeEventListener('click', addBord, false);  
    }  
    else if (detH.detachEvent) { // IE8 & earlier versions  
        detH.detachEvent('click', addBord);  
    }  
}  
  
function addBord () {  
    var add = document.getElementById('p1');  
    add.style.border = '1px dotted #ff0000';  
    add.style.backgroundColor = 'ffff99';  
}  
</script>  
</head>
```

Event Registration - Example

```
<body onload="addHandler()">
<p id='p1'>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed scelerisque
odio non ligula. </p>
<p>Integer mollis, libero et facilisis hendrerit, tellus dui porttitor quam, vitae iaculis nisi
mauris ac dui. Vivamus volutpat sollicitudin est. </p>
<form>
<input type='button' value='No border, please!' onclick='detHandler()'>
</form>
</body>
</html>
```


Event Registration - Example

```
function detHandler () {  
    var detH = document.getElementById('p1');  
    detH.style.border = "";  
    detH.style.backgroundColor = 'ffffff';  
    if (detH.removeEventListener) {  
        detH.removeEventListener('click', addBord, false);  
    }  
    else if (detH.detachEvent) { // IE8 & earlier versions  
        detH.detachEvent('click', addBord);  
    }  
}  
function addBord () {  
    var add = document.getElementById('p1');  
    add.style.border = '1px dotted #ff0000';  
    add.style.backgroundColor = 'ffff99';  
}  
</script>  
</head>
```

```
function addBord (e) {  
    var add = e.target;  
    add.style.border = '1px dotted #ff0000';  
    add.style.backgroundColor = 'ffff99';  
}
```

-
- https://www.w3schools.com/jsref/event_target.asp

Get the element that triggered a specific event.

e.g.

```
function hide (e) {  
    var t = e.target || e.srcElement; // IE8 & earlier versions  
    t.style.visibility = 'hidden';  
}
```

Thank you!