CSC309 Winter 2016
Lecture 3

Larry Zhang

Why Javascript

- Javascript is for dynamically manipulate the front-end of your web page.
  - Add/remove/change the content/attributes of an HTML element
  - Change the style of an HTML element
  - Detect user interactions with the web page.
  - …, pretty much everything with the front-end

How it works

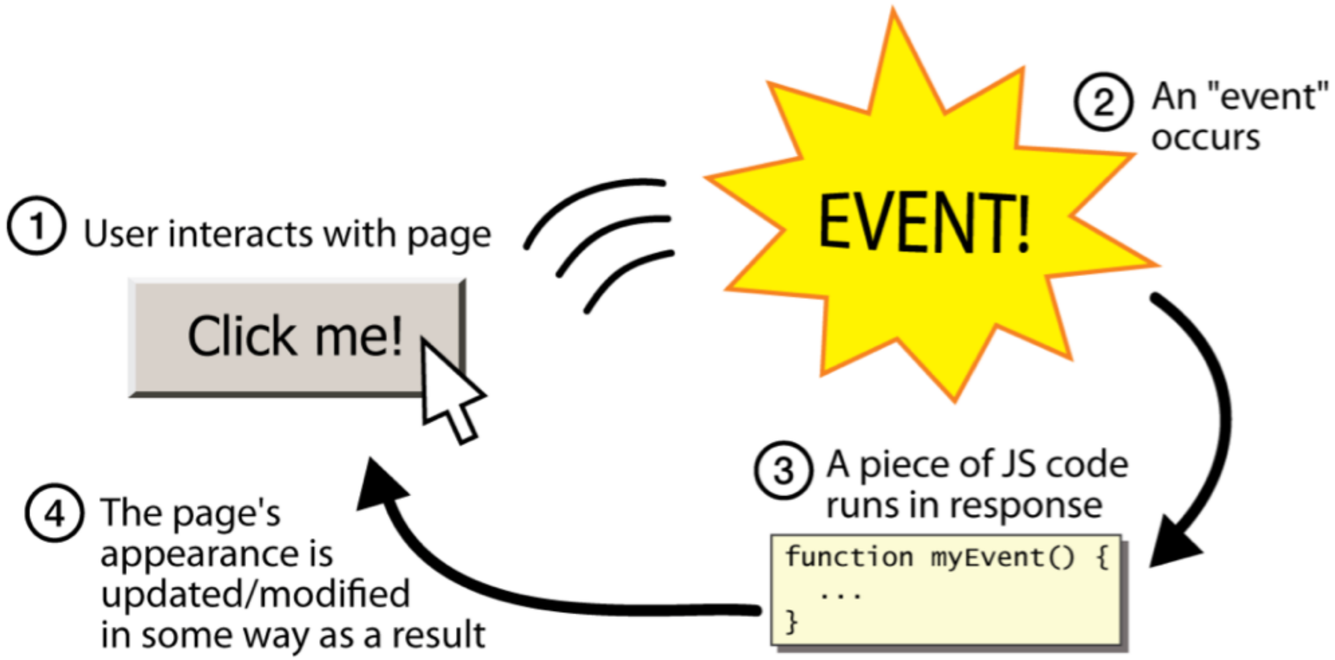
- it runs only on client side (the browser), i.e., it does not communicate with the server side (normally)
  - So Javascript can do fast UI
- It is interpreted language, interpreted by a **Javascript Engine** built in the browser.
  - V8 (Chrome), SpiderMonkey (Firefox), JavascriptCore (Safari)
- It is **event-driven**, i.e., all actions are triggered by a user **event**
  - e.g., user clicking on something, or page loaded
- It has nothing really to do with Java

The Iron Triangle

- **HTML**: defines the **content** of the web page
- **CSS**: specify the **style** of the web page
- **Javascript**: program the **behaviour** of the web page.

JS programming model: Event-Driven

- There is no “main” function
- Every function call is trigger by an event happening at the web page.
- If no event happens, the javascript are just like a static chunk of code and nothing happens.

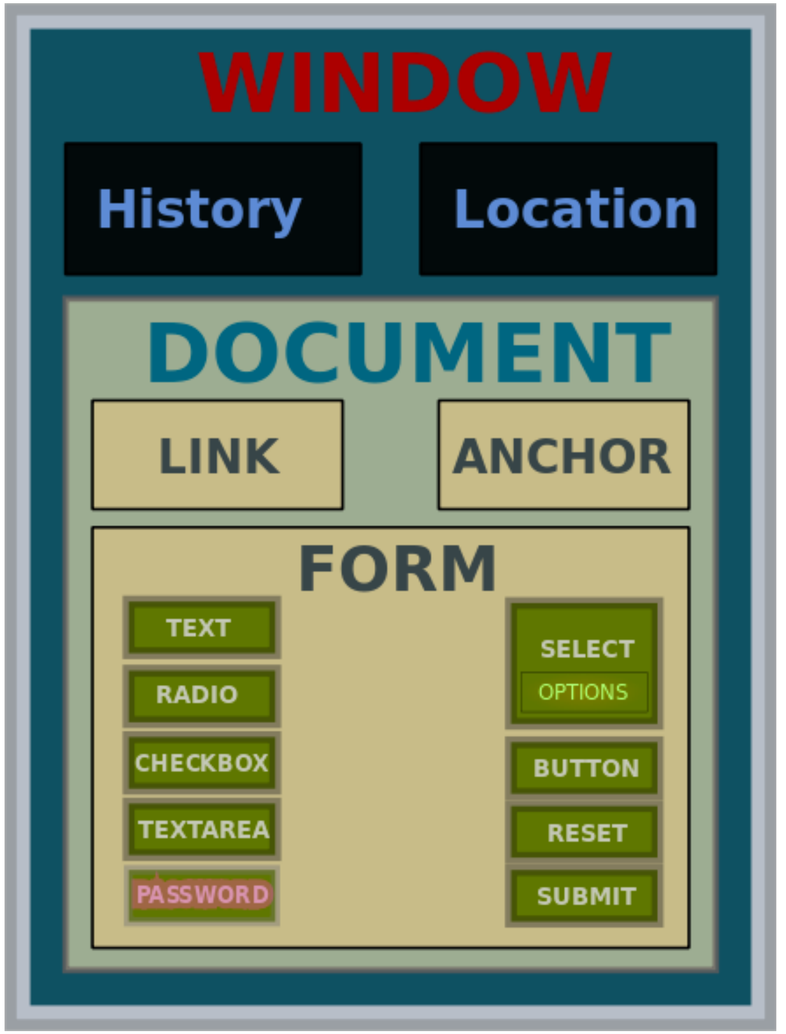


Before we start programming Javascript, we need to know the **environment** that is runs in.

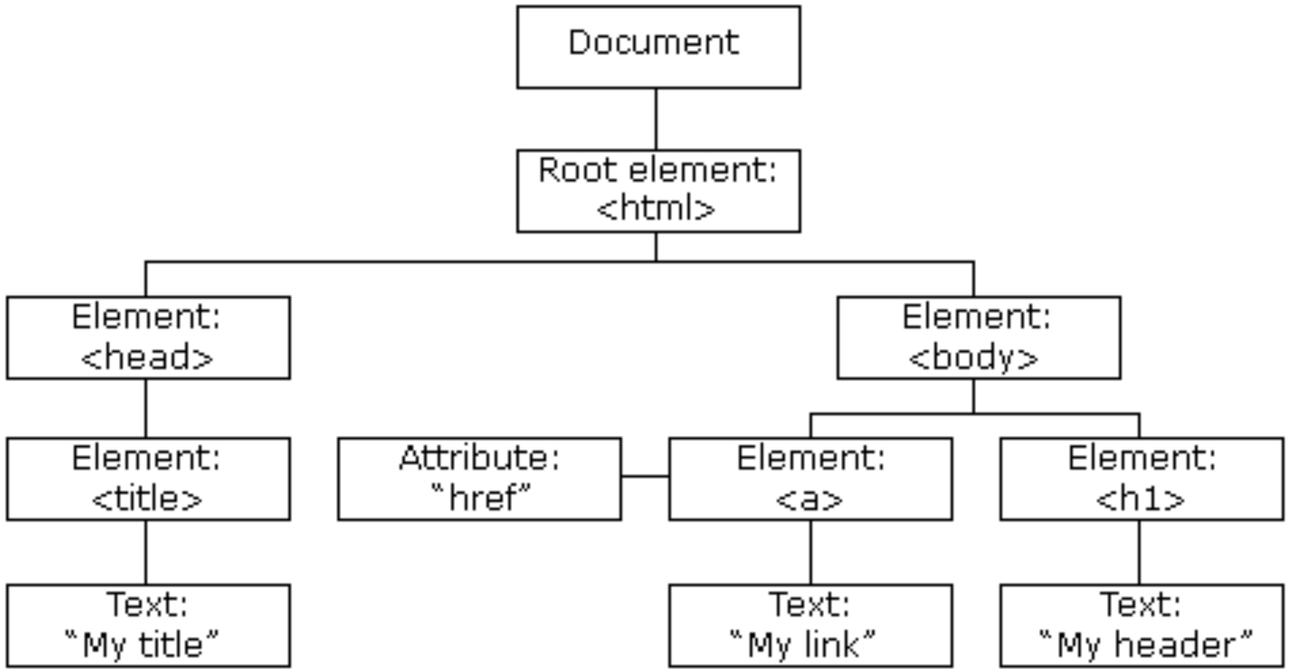
We know javascript can manipulate the HTML document, but how does JS get access to the elements in the HTML document?

Document Object Model (DOM)

- It is a cross-platform language-independent convention for representing and interacting with objects in HTML and XML document.
- Standard set by W3C.
- It is how the browser internally organized the objects in a loaded HTML document.
- It is the **programming interface** for HTML.
- The objects form a **DOM tree**.



Document Object Model (DOM)



Every element is an object.

Find an element and change it

- Find an HTML element
  - document.getElementById(id)
  - document.getElement**s**ByTagName(name)
  - document.getElement**s**ByClassName(name)
- Change HTML elements
  - element.innerHTML = new html content
  - element.attribute = new value
  - element.setAttribute(attribute, value)
  - element.style.property = new style

demo

Event handler: onclick

demo

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Page</h1>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

This action is trigger when the script is loaded.
What if I want the action to be triggered by a click on something?
Use the “onclick” handler.

Define a function

demo

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Page</h1>

<p id="demo">paragraph</p>

<button onclick='document.getElementById("demo").innerHTML =
"HAHA"'>I'm a button</button>

<script>
</script>

</body>
</html>
```

This is ugly, too much stuff in the line

Decoupling HTML and Javascript

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Page</h1>

<p id="demo">paragraph</p>

<button onclick="myfun();">I'm a button</button>

<script>
function myfun () {
    document.getElementById("demo").innerHTML = "HAHA";
}
</script>

</body>
</html>
```

demo

This is still ugly. Shouldn't the content definition (HTML) and the behaviour definition (Javascript) be completely **decoupled**?

The good code

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Page</h1>

<p id="demo">paragraph</p>

<button id="button1">I'm a button</button>

<script>
document.getElementById("button1").onclick = function () {
    document.getElementById("demo").innerHTML = "HAHA";
}
</script>

</body>
</html>
```

This part can now be put into a separate file.

Separate files

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Page</h1>

<p id="demo">paragraph</p>

<button id="button1">I'm a button</button>

<script src="myScript.js"></script>

</body>
</html>
```

```
// myScript.js

document.getElementById("button1").onclick = function () {
    document.getElementById("demo").innerHTML = "HAHA";
}
```

- Performance tip: it is typically a good idea to put the scripts at the bottom of the body, it can improve page load, since loading and compiling Javascripts can take time.
- Putting Javascript in a separate file also improve page load, since the script file can be cached.

Note

- In CSS we have **font-size**, **background-color**
- In JS when setting style we use **fontSize**, **backgroundColor**

Anonymous function

```
document.getElementById("button1").onclick = function () {
    document.getElementById("demo").innerHTML = "HAHA";
}
```

Named function

```
document.getElementById("button1").onclick = normalFunc;

function normalFunc () {
    document.getElementById("demo").innerHTML = "HAHA";
}
```

Readings

- More on DOM
  - http://www.w3schools.com/js/js_htmldom.asp
- More on Events
  - http://www.w3schools.com/js/js_events.asp

We’ve got an idea how to write Javascript. Now let’s learn the language a bit more systematically.

Variables

```
var name = expression;                                    JS
```

```
var clientName = "Connie Client";
var age = 32;
var weight = 127.4;                                       JS
```

- variables are declared with the *var* keyword (case sensitive)
- types are not specified, but JS does have types ("loosely typed")
  - `Number, Boolean, String, Array, Object, Function, Null, Undefined`
  - can find out a variable's type by calling `typeof`

Number types

```
var enrollment = 99;
var medianGrade = 2.8;
var credits = 5 + 4 + (2 * 3);
```
JS

- integers and real numbers are the same type (no int vs. double)
- same operators: + - * / % ++ -- = += -= *= /= %=
- similar precedence to Java
- many operators auto-convert types: "2" * 3 is 6

Math object

```
var rand1to10 = Math.floor(Math.random() * 10 + 1);
var three = Math.floor(Math.PI);
```
JS

- **methods:** abs, ceil, cos, floor, log, max, min, pow, random, round, sin, sqrt, tan
- **properties:** E, PI

Special values: null and undefined

```
var ned = null;
var benson = 9;
var caroline;
// at this point in the code,
// ned is null
// benson's 9
// caroline is undefined
```

JS

- `undefined` : has not been declared, does not exist
- `null` : exists, but was specifically assigned an empty or null value

Logical operators

- > < >= <= && | | ! == != === !==
- most logical operators automatically convert types:
  - 5 < "7" is true
  - 42 == 42.0 is true
  - "5.0" == 5 is true
- === and !== are strict equality tests; checks both *type* and *value*
  - "5.0" === 5 is false

if/else statements

```
if (condition) {
        statements;
} else if (condition) {
        statements;
} else {
        statements;
}
```
JS

- identical structure to Java's if/else statement
- JavaScript allows almost anything as a condition

Boolean types

```
var iLike190M = true;
var ieIsGood = "IE6" > 0; // false
if ("web devevelopment is great") { /* true */ }
if (0) { /* false */ }
```
JS

- any value can be used as a Boolean
  - "falsey" values: 0, 0.0, NaN, "", null, and undefined
  - "truthy" values: anything else
- converting a value into a Boolean explicitly:
  - `var boolValue = Boolean(otherValue);`
  - `var boolValue = !!(otherValue);`

for loop

```
var sum = 0;
for (var i = 0; i < 100; i++) {
        sum = sum + i;
}
```
JS

```
var s1 = "hello";
var s2 = "";
for (var i = 0; i < s.length; i++) {
        s2 += s1.charAt(i) + s1.charAt(i);
}
// s2 stores "hheellllloo"
```
JS

while loop

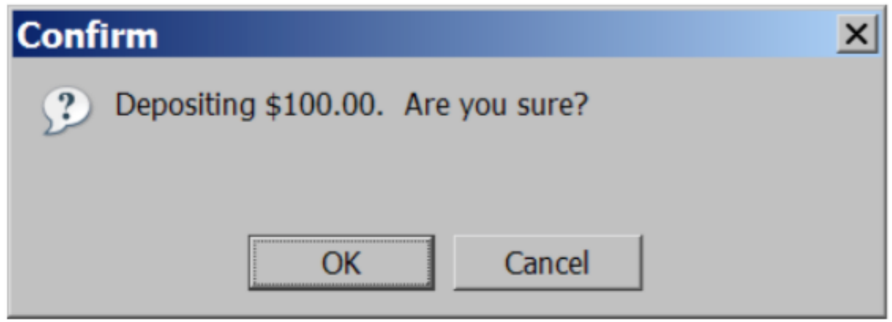
```
while (condition) {
       statements;
}
```
JS

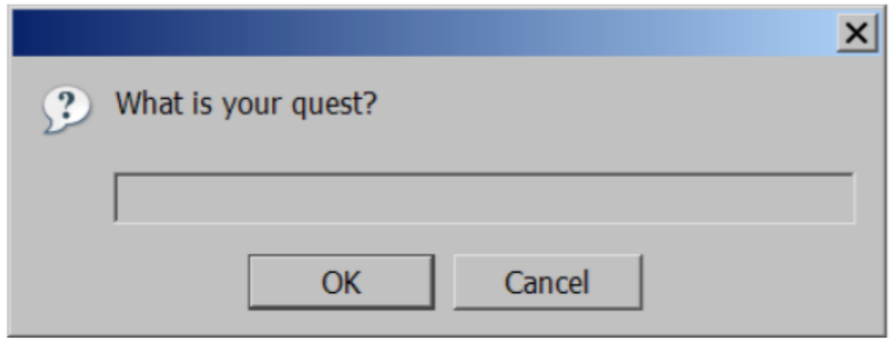
```
do {
    statements;
} while (condition);
```
JS

- break and continue keywords also behave as in Java

Pop-up

```
alert("message"); // message
confirm("message"); // returns true or false
prompt("message"); // returns user input string
```
JS





Arrays

```
var name = []; // empty array
var name = [value, value, ..., value]; // pre-filled
name[index] = value; // store element
```

JS

```
var ducks = ["Huey", "Dewey", "Louie"];
var stooges = []; // stooges.length is 0
stooges[0] = "Larry"; // stooges.length is 1
stooges[1] = "Moe"; // stooges.length is 2
stooges[4] = "Curly"; // stooges.length is 5
stooges[4] = "Shemp"; // stooges.length is 5
```

JS

Array methods

```
var a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
```

JS

- array serves as many data structures: list, queue, stack, ...
- methods: `concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift`
  - push and pop add / remove from back
  - unshift and shift add / remove from front
  - shift and pop return the element that is removed

String type

```
var s = "Connie Client";
var fName = s.substring(0, s.indexOf(" ")); // "Connie"
var len = s.length; // 13
var s2 = 'Melvin Merchant';
```
JS

- **methods:** `charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase`
  - charAt returns a one-letter String (there is no char type)
- length property (not a method as in Java)
- Strings can be specified with "" or ''
- concatenation with + :
  - 1 + 1 is 2, but "1" + 1 is "11"

More about strings

- escape sequences as in Java: \' \" \& \n \t \\
- converting between numbers and Strings:

```
var count = 10;
var s1 = "" + count; // "10"
var s2 = count + " bananas, ah ah ah!"; // "10 bananas, ah
ah ah!"
var n1 = parseInt("42 is the answer"); // 42
var n2 = parseFloat("booyah"); // NaN                    JS
```

- accessing the letters of a String:

```
var firstLetter = s[0]; // fails in IE
var firstLetter = s.charAt(0); // does work in IE
var lastLetter = s.charAt(s.length - 1);                JS
```

Split and join

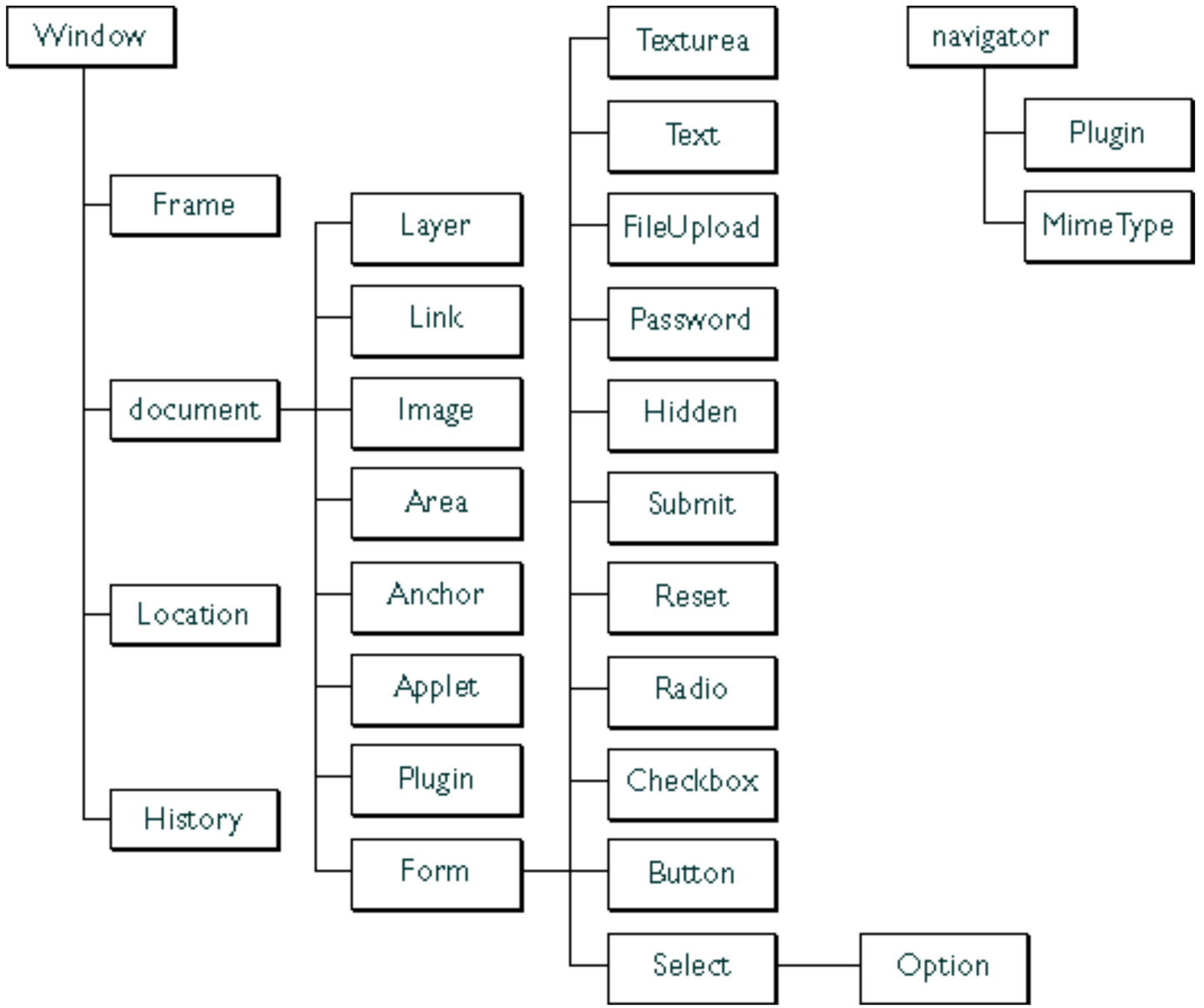
```
var s = "the quick brown fox";
var a = s.split(" "); // ["the", "quick", "brown", "fox"]
a.reverse();          // ["fox", "brown", "quick", "the"]
s = a.join("!");      // "fox!brown!quick!the"
```
JS

- split breaks apart a string into an array using a delimiter
  - can also be used with regular expressions (seen later)
- join merges an array into a single string, placing a delimiter between them

Reference:

http://www.w3schools.com/js/default.asp

Browser Object Hierarchy


Window
Frame
document
Location
History
Layer
Link
Image
Area
Anchor
Applet
Plugin
Form
Texturea
Text
FileUpload
Password
Hidden
Submit
Reset
Radio
Checkbox
Button
Select
Option
navigator
Plugin
MimeType

Global DOM

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

The **window** object

- *the entire browser window (DOM top-level object)*
- technically, all global code and variables become part of the window object properties:
  - document, history, location, name
- methods:
  - alert, confirm, prompt (popup boxes)
  - `setInterval`, `setTimeout` `clearInterval`, `clearTimeout` (timers)
  - `open`, `close` (popping up new browser windows)
  - `blur`, `focus`, `moveBy`, `moveTo`, `print`, `resizeBy`, `resizeTo`, `scrollBy`, `scrollTo`

The **document** object

- *the current web page and the elements inside it*
- **properties:**
  - `anchors, body, cookie, domain, forms, images, links, referrer, title, URL`
- **methods:**
  - `getElementById`
  - `getElementsByName`
  - `getElementsByTagName`
  - `close, open, write, writeln`

The **location** object

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

The **navigator** object

- *information about the web browser application*
- properties:
  - `appName, appVersion, browserLanguage, cookieEnabled, platform, userAgent`
- Some web programmers examine the navigator object to see what browser is being used, and write browser-specific scripts and hacks:

```
if (navigator.appName === "Microsoft Internet Explorer") {
...
```
JS

The **screen** object

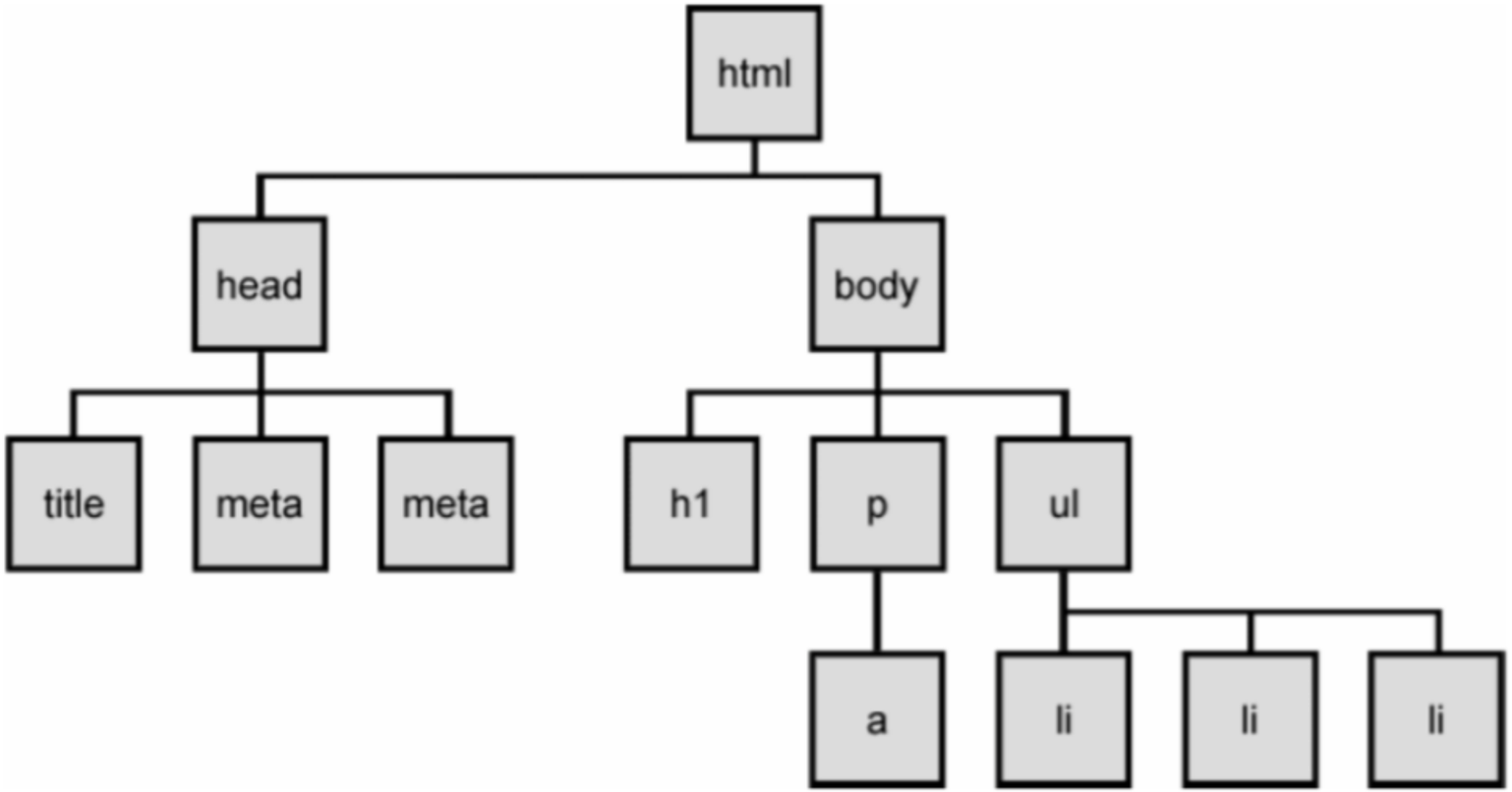
- *information about the client's display screen*
- properties:
  - `availHeight, availWidth, colorDepth, height, pixelDepth, width`

The **history** object

- the list of sites the browser has visited in this window
- properties:
  - `length`
- methods:
  - `back, forward, go`
- sometimes the browser won't let scripts view `history` properties, for security

The DOM Tree

The DOM Tree



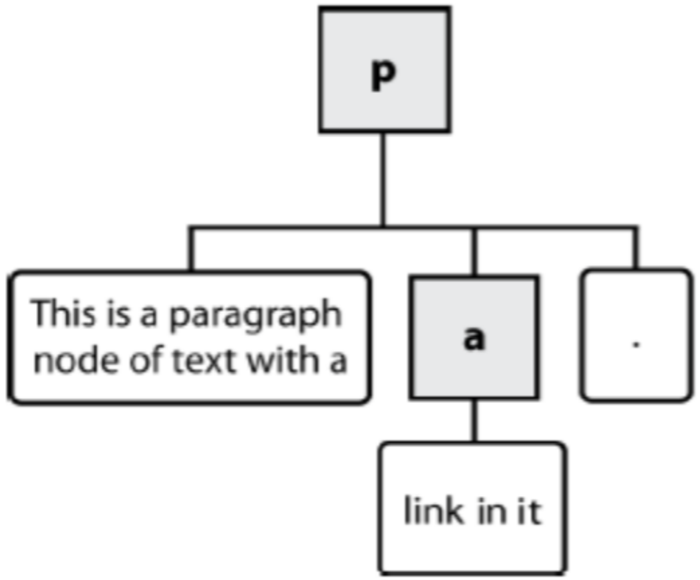
Types of DOM nodes

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

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

Types of DOM nodes

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



Traversing the DOM tree

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

Full list: https://developer.mozilla.org/en-US/docs/Web/API/Node

Create new nodes

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

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

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

Modifying the DOM tree

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

```
var p = document.createElement("p");
p.innerHTML = "A paragraph!";
document.getElementById("main").appendChild(p);
```
JS

Which way is better?

```JS
var p = document.createElement("p");
p.innerHTML = "A paragraph!";
document.getElementById("main").appendChild(p);
```

Better

vs

```JS
function slideClick() {
      document.getElementById(“thisslide").innerHTML +=
"<p>A paragraph!</p>";
}
```

- Hack
- Ugly, if the added element is complicated.
- error-prone, especially with a mixed ‘’ and “”
- can only add the end, cannot insert in the middle

When does the code run?

When does the code run?

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

```
// global code
var x = 3;
function f(n) { return n + 1; }
function g(n) { return n - 1; }
x = f(x);
```
JS

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

a failing example

```
<head>
<script src="myfile.js" type="text/javascript"></script>
</head>
<body>
<div><button id="ok">OK</button></div>
```
HTML

```
// global code
document.getElementById("ok").onclick = okayClick;
// error: $("ok") is null
```
JS

- Problem: global JS code runs the moment the script is loaded
- Script in head is processed before the page body is loaded
- No element with id=“ok” has been created yet.

a solution: window.onload

```
// called when page loads; sets up event handlers
function pageLoad() {
	document.getElementById("ok").onclick = okayClick;
}
function okayClick() {
	alert("booyah");
}
window.onload = pageLoad; // global code
```
JS

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

The keyword **this**

```
this.fieldName // access field
this.fieldName = value; // modify field
this.methodName(parameters); // call method
```

JS

- all JavaScript code actually runs inside of an object
- by default, code runs inside the global window object
  - all global variables and functions you declare become part of window
- the '*this*' keyword refers to the current object

The keyword **this**

```
function pageLoad() {
	document.getElementById("ok").onclick = okayClick;
	// bound to okButton here
}
function okayClick() { // okayClick knows what DOM object
	this.innerHTML = "booyah"; // it was called on
}
window.onload = pageLoad;
```

JS

- The event handler **okayClick** is **bound** to the element with id "**ok**"
- So this means the element with id "ok"
- No need to do getElementById() again.

- We’ve only covered a small part of things you can do with Javascript
- You should go through the reference to get a better sense what are all the things
- http://www.w3schools.com/js/default.asp
- When you need to do it, look it up exact how to
- Use the Developer Console

demo: making a game

http://www.cs.toronto.edu/~ylzhang/csc309w16/puzzle/game.html

- **Today we learned**
  - Javascript
- **Next week**
  - JQuery, Ajax; PHP