

Exercise: battleship game

- **Goal:** Sink the browser's ships in the fewest number of guesses. You're given a rating, based on how well you perform
- **Setup:** When the game program is launched, the computer places ships on a virtual grid. When that's done, the game asks for your first guess
- **How you play:** *The browser will prompt you to enter a guess and you'll type in a grid location. In response to your guess, you'll see a result of "Hit", "Miss", or "You sank my battleship!" When you sink all the ships, the game ends by displaying your rating*

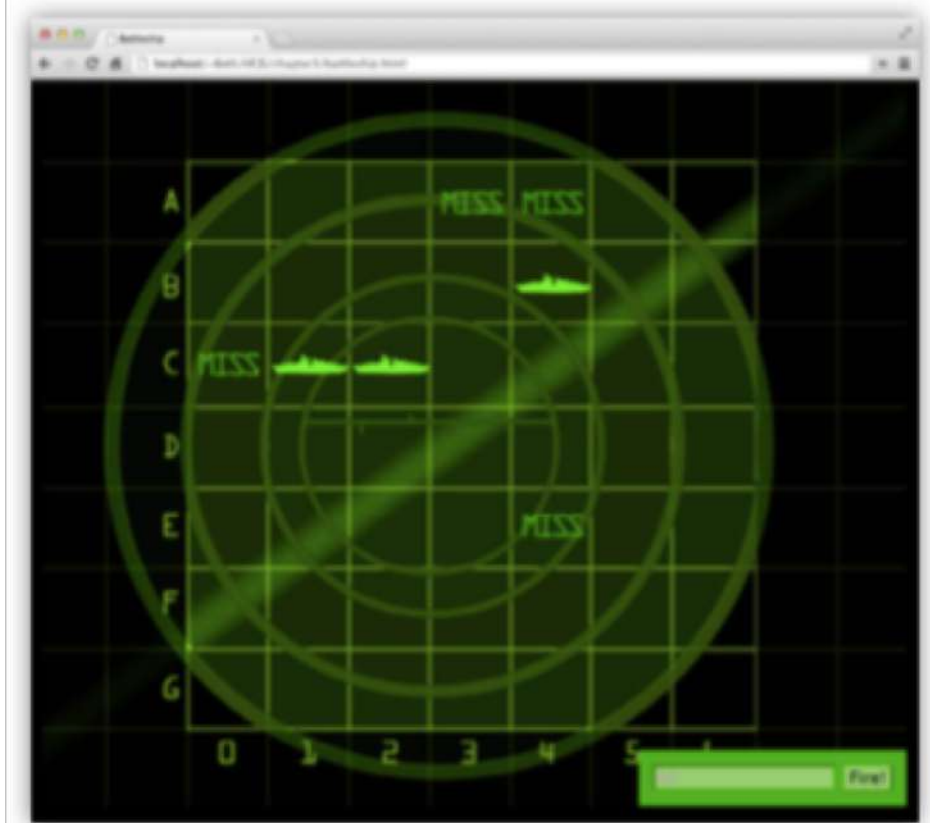
Simplified battleship

- Objective: 7x7 graphical version with three ships, but...
- We're going to start with a nice 1-D grid with seven locations and one ship to find

Instead of a 7x7 grid, like the one above, we're going to start with just a 1x7 grid. And, we'll worry about just one ship for now.



Notice that each ship takes up three grid locations (similar to the real board game).



High-level design

- 1** User starts the game
 - A** Game places a battleship at a random location on the grid.
- 2** Game play begins

Repeat the following until the battleship is sunk:

 - A** Prompt user for a guess ("2", "0", etc.)
 - B** Check the user's guess against the battleship to look for a hit, miss or sink.
- 3** Game finishes

Give the user a rating based on the number of guesses.

Details

- **Representing the ships**

- Keep in mind the virtual grid
- The user know that the battleship is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code

- **Getting user input**

- Use the **prompt** function. Whenever we need to get a new location from the user, we'll use prompt to display a message and get the input, which is just a number between 0 and 6, from the user

- **Displaying the results**

- Use alert to show the output of the game

Sample game interaction



Battelship

```
<!doctype html>  
<html lang="en">
```

```
<head>
```

```
<title>Battleship</title>
```

```
<meta charset="utf-8">
```

```
</head>
```

```
<body>
```

```
<h1>Play battleship!</h1>
```

```
<script src="battleship.js"></script>
```

```
</body>
```

```
</html>
```

← The HTML for the Battleship game is super simple; we just need a page that links to the JavaScript code, and that's where all the action happens.

↖ We're linking to the JavaScript at the bottom of the <body> of the page, so the page is loaded by the time the browser starts executing the code in "battleship.js".

Battleship (2)

DECLARE three *variables* to hold the location of each cell of the ship. Let's call them location1, location2 and location3

DECLARE a *variable* to hold the user's current guess. Let's call it guess

DECLARE a *variable* to hold the number of hits. We'll call it hits and *set* it to 0

DECLARE a *variable* to hold the number of guesses. We'll call it guesses and *set* it to 0

DECLARE a *variable* to keep track of whether the ship is sunk or not. Let's call it isSunk and *set* it to false

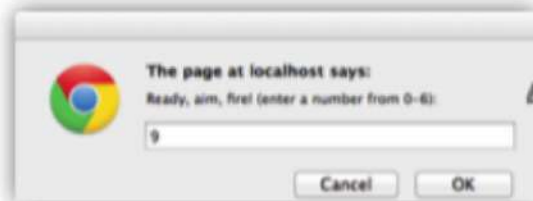
```
var randomLoc = Math.floor(Math.random() * 5);  
var location1 = randomLoc;  
var location2 = location1 + 1;  
var location3 = location1 + 2;  
var guess;  
var hits = 0;  
var guesses = 0;  
var isSunk = false;
```

Battleship (3)

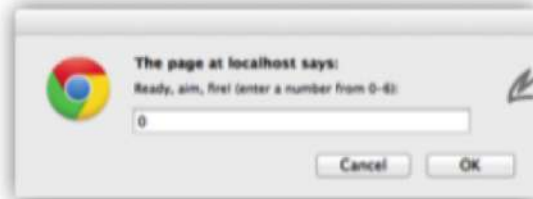
```
while (isSunk == false) {  
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");  
    if (guess < 0 || guess > 6) {  
        alert("Please enter a valid cell number!");  
    } else {  
        guesses = guesses + 1;  
        if (guess == location1 || guess == location2 || guess == location3) {  
            alert("HIT!");  
            hits = hits + 1;  
            if (hits == 3) {  
                isSunk = true;  
                alert("You sank my battleship!");  
            }  
        } else {  
            alert("MISS");  
        }  
    }  
}  
  
var stats = "You took " + guesses + " guesses to sink the battleship, " +  
            "which means your shooting accuracy was " + (3/guesses);  
alert(stats);
```


Here's what our game interaction looked like.

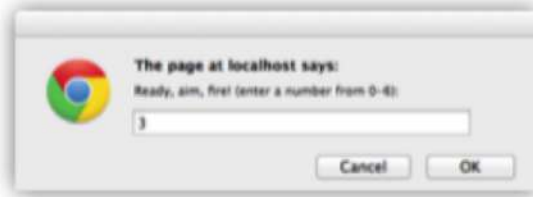
Test it!



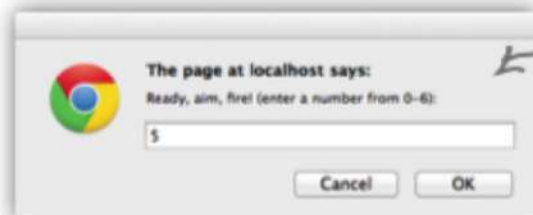
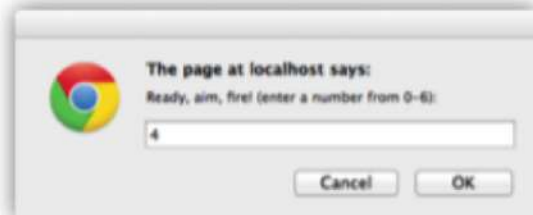
First we entered an invalid number, 9.



Then we entered 0, to get a miss.

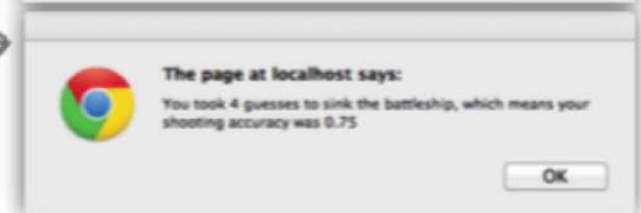
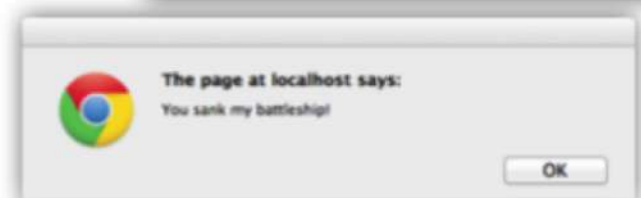
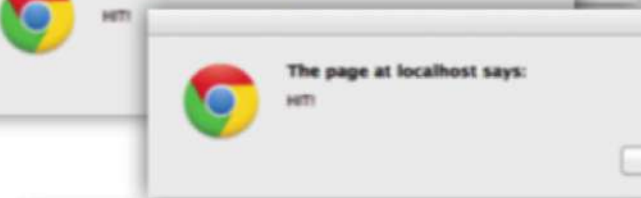


But then we get three hits in a row!



On the third and final hit, we sink the battleship.

And see that it took 4 guesses to sink the ship with an accuracy of 0.75.



Oggetti

- Gli oggetti sono tipi composti che contengono un certo numero di **proprietà** (attributi)
 - Ogni proprietà ha un **nome** e un **valore**
 - Si accede alle proprietà con l'operatore **'.'** (punto)
 - Le proprietà non sono definite a priori: *possono essere aggiunte dinamicamente*
- Gli oggetti vengono creati usando l'operatore **new**:
var o = new Object()
- **Attenzione:** `Object()` è un costruttore e non una classe
- Le classi non esistono e quindi i due concetti non si sovrappongono come avviene in Java

Costruire un oggetto

- Un oggetto appena creato è completamente vuoto non ha ne proprietà né metodi
- Possiamo costruirlo dinamicamente
 - appena assegniamo un valore ad una proprietà la proprietà comincia ad esistere
- Nell'esempio sottostante creiamo un oggetto e gli aggiungiamo 3 proprietà numeriche: x, y e tot:

```
var o = new Object();  
o.x = 7;  
o.y = 8;  
o.tot = o.x + o.y;  
alert(o.tot);
```

Costanti oggetto

- Le costanti oggetto (**object literal**) sono racchiuse fra parentesi graffe **{ }** e contengono un elenco di attributi nella forma: **nome:valore**
var nomeoggetto = {prop1:val1, prop2:val2, ...}
- Usando le costanti oggetto creiamo un oggetto e le proprietà (valorizzate) nello stesso momento
- I due esempi seguenti sono del tutto equivalenti:

```
var o = new Object();  
o.x = 7;  
o.y = 8;  
o.tot = 15;  
alert(o.tot);
```

```
var o = {x:7, y:8, tot:15};  
alert(o.tot);
```

Example (isobject.html)

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

<p>Creating a JavaScript Object.</p>

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

<script>
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>

</body>
</html>
```

Example 2

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

<p>Creating and using an object method.</p>

<p>An object method is a function definition, stored as a property
value.</p>

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

<script>
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};

document.getElementById("demo").innerHTML = person.fullName();
</script>
</body>
</html>
```


Array

- Gli array sono tipi composti i cui elementi sono accessibili mediante un indice numerico
 - l'indice parte da zero (**0**)
 - Non hanno una dimensione prefissata (simili agli ArrayList di Java)
 - Espongono attributi e metodi
- Vengono istanziati con **new Array([dimensione])**
- Si possono creare e inizializzare usando delle costanti

array (array literal) delimitate da **[]**:

var varname = [val,val2,...,valn]

- Es. **var a = [1,2,3];**
Possono contenere elementi di tipo eterogeneo:
- Es. **var b = [1,true,"ciao",{x:1,y:2}];**

makePhrase.html

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <title>makePhrase</title>
6   <script>
7     function makePhrases() {
8       var words1 = ["24/7", "multi-tier", "30,000 foot", "B-to-B", "win-win"];
9       var words2 = ["empowered", "value-added", "oriented", "focused", "aligned"];
10      var words3 = ["process", "solution", "tipping-point", "strategy", "vision"];
11      var rand1 = Math.floor(Math.random() * words1.length);
12      var rand2 = Math.floor(Math.random() * words2.length);
13      var rand3 = Math.floor(Math.random() * words3.length);
14      var phrase = words1[rand1] + " " + words2[rand2] + " " + words3[rand3];
15      alert(phrase);
16    }
17    makePhrases();
18  </script>
19 </head>
20 <body></body>
21 </html>
```

Example (cars.html)

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

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

<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>

</body>
</html>
```

Example 2

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

<p>The length property returns the length of an array.</p>

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

<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.length;
</script>

</body>
</html>
```

Example: delete

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

<p>The delete operator deletes a property from an object.</p>

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

<script>
var person = {
  firstname:"John",
  lastname:"Doe",
  age:50,
  eyecolor:"blue"
};
delete person.age;
document.getElementById("demo").innerHTML =
person.firstname + " is " + person.age + " years old.";
</script>

</body>
</html>
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

<p>The in operator returns true if the specified property is in the specified object.</p>

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

```
<script>
```

```
// Arrays
```

```
var cars = ["Saab", "Volvo", "BMW"];
```

```
// Objects
```

```
var person = {firstName:"John", lastName:"Doe", age:50};
```

```
document.getElementById("demo").innerHTML =
```

```
  ("Saab" in cars) + "<br>" +
```

```
  (0 in cars) + "<br>" +
```

```
  (1 in cars) + "<br>" +
```

```
  (4 in cars) + "<br>" +
```

```
  ("length" in cars) + "<br>" +
```

```
  ("firstName" in person) + "<br>" +
```

```
  ("age" in person) + "<br>" +
```

```
  // Predefined objects
```

```
  ("PI" in Math) + "<br>" +
```

```
  ("NaN" in Number) + "<br>" +
```

```
  ("length" in String);
```

```
</script>
```

```
</body>
```

```
</html>
```

Example: in.html

Example: instanceof

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

<p>The instanceof operator returns true if the specified object is an instance of the specified object.</p>

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

```
<script>
```

```
var cars = ["Saab", "Volvo", "BMW"];
```

```
document.getElementById("demo").innerHTML =
```

```
    (cars instanceof Array) + "<br>" +
```

```
    (cars instanceof Object) + "<br>" +
```

```
    (cars instanceof String) + "<br>" +
```

```
    (cars instanceof Number);
```

```
</script>
```

```
</body>
```

```
</html>
```

Example: void.html

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

<p>
<a href="javascript:void(0);">
  Useless link
</a>
</p>

<p>
<a href="javascript:void(document.body.style.backgroundColor='red');">
  Click me to change the background color of body to red.
</a>
</p>

</body>
</html>
```

The **void** operator evaluates an expression and returns **undefined**. This operator is often used to obtain the undefined primitive value, using "void(0)" (useful when evaluating an expression without using the return value).

Oggetti e Array

- Gli oggetti in realtà sono **array associativi**
 - strutture composite i cui elementi sono accessibili mediante un **indice di tipo stringa** (nome) anziché attraverso un **indice numerico**
- Si può quindi utilizzare anche una sintassi analoga a quella degli array
- Le due sintassi sono del tutto equivalenti e si possono mescolare

```
var o = new Object();  
o.x = 7;  
o.y = 8;  
o.tot = o.x + o.y;  
alert(o.tot);
```

```
var o = new Object();  
o["x"] = 7;  
o.y = 8;  
o["tot"] = o.x + o["y"];  
alert(o.tot);
```

Stringhe

- Non è facile capire esattamente cosa sono le stringhe in JavaScript
- Potremmo dire che mentre in Java sono oggetti che sembrano dati di tipo primitivo in JavaScript sono **dati di tipo primitivo che sembrano oggetti**
- Sono sequenze arbitrarie di caratteri in formato UNICODE a 16 bit e sono immutabili come in Java
- Esiste la possibilità di definire costanti stringa (**string literal**) delimitate da apici singoli ('ciao') o doppi ("ciao")
- È possibile la concatenazione con l'operatore **+**
- È possibile la comparazione con gli operatori **< > >= <=** e **!=**

Example

- When comparing two strings, "2" will be greater than "12", because (alphabetically) 1 is less than 2.

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

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

<script>
document.getElementById("demo").innerHTML = "2" < "12";
</script>

</body>
</html>
```

Stringhe come oggetti?

- Possiamo però invocare metodi su una stringa o accedere ai suoi attributi
- Possiamo infatti scrivere
 - `var s = "ciao";`
 - `var n = s.length;`
 - `var t = s.charAt(1);`
- Non sono però oggetti e la possibilità di trattarli come tali nasce da due caratteristiche:
 - Esiste un tipo wrapper `String` che è un oggetto
 - *JavaScript fa il boxing in automatico come C#*
 - quando una variabile di tipo valore necessita essere convertita in tipo riferimento, un oggetto box è allocato per mantenere tale valore

String Basics

- You can use double or single quotes

```
var names = ["joe", 'jane', "john", 'juan'];
```

- Strings have length property

```
"Foobar".length → 6
```

- Numbers can be converted to strings

- Automatic conversion during concatenations.

```
var val = 3 + "abc" + 5; // result is "3abc5"
```

- Conversion with fixed precision

```
var n = 123.4567;
```

```
var val = n.toFixed(2); // result is 123.46  
(not 123.45)
```

String Basics (Continued)

- Strings can be compared with `==`

```
"foo" == 'foo' // returns true
```

- Strings can be converted to numbers

```
var i = parseInt("37 blah");  
      // result is 37 - ignores blah
```

```
var d = parseFloat("6.02 blah");  
      // result is 6.02 - ignores blah
```

Core String Methods

- Simple methods

- Charat, indexof, lastindexof, substring, tolowercase, touppercase

`"Hello".charAt(1); → "e"`

`"Hello".indexOf("o"); → 4 // returns -1 if no match`

`"hello".substring(1,3); → "el"`

`"Hello".toUpperCase(); → "HELLO"`

- Methods that use regular expressions

- match, replace, search, split

- HTML methods

- anchor, big, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, sup

`"test".bold().italics().fontcolor("red")`

`→ '<i>test</i>'`

- These are technically nonstandard methods, but supported in all major browsers

JavaScript Data Types

```
var length = 16;           // Number
var lastName = "Johnson"; // String
var cars = ["Saab", "Volvo", "BMW"]; // Array
var x = {firstName:"John", lastName:"Doe"}; // Object
```

```
var x;           // Now x is undefined
var x = 5;       // Now x is a Number
var x = "John";  // Now x is a String
```

```
var carName = "Volvo XC60"; // Using double quotes
var carName = 'Volvo XC60'; // Using single quotes
```

```
var answer = "It's alright"; // Single quote inside double quotes
var answer = "He is called 'Johnny'"; // Single quotes inside double quotes
var answer = 'He is called "Johnny"'; // Double quotes inside single quotes
```


The typeof Operator

- You can use the JavaScript typeof operator to find the type of a JavaScript variable:

```
typeof "John"           // Returns string
typeof 3.14              // Returns number
typeof NaN               // Returns number
typeof false            // Returns boolean
typeof [1, 2, 3, 4]      // Returns object
typeof {name:'John', age:34} // Returns object
typeof new Date()        // Returns object
typeof function () {}   // Returns function
typeof myCar             // Returns undefined (if myCar is not declared)
typeof null              // Returns object
```



Tipi valore e tipi riferimento

- Si può distinguere fra **tipi valore** e **tipi riferimento**
 - Numeri e booleani sono tipi valore
 - Array e Oggetti sono tipi riferimento
 - Per le stringhe abbiamo una situazione incerta
 - **Pur essendo un tipo primitivo si comportano come un tipo riferimento**
- 

Funzioni

- Una funzione è un frammento di codice JavaScript che viene definito una volta e usato in più punti
- Ammette parametri che sono privi di tipo
- Restituisce un valore il cui tipo non viene definito
- La mancanza di tipo è coerente con la scelta fatta per le variabili
- Le funzioni possono essere definite utilizzando la parola chiave **function**
- Una funzione può essere assegnata ad una variabile

```
function sum(x,y)
{
    return x+y;
}
```

```
var s = sum(2,4) ;
```

Funzioni

Again, these are parameters;
they are assigned values when
the function is called.

```
function bark(name, weight) {  
  if (weight > 20) {  
    console.log(name + " says WOOF WOOF");  
  } else {  
    console.log(name + " says woof woof");  
  }  
}
```

And everything inside the function
is the body of the function.

```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Bark</title>
6   </head>
7   <body>
8   </body>
9   <script>
10  function bark(name, weight) {
11    if (weight > 20) {
12      console.log(name + " says WOOF WOOF");
13    } else {
14      console.log(name + " says woof woof");
15    }
16  }
17  bark("rover", 23);
18  bark("spot", 13);
19  bark("spike", 53);
20  bark("lady", 17);
21  </script>
22 </body>
23 </html>
```

Test the bark function

```
    }  
    }  
    bark("rover", 23);  
    bark("spot", 13);  
    bark("spike", 53);  
    bark("lady", 17);
```

We just did this... →

...so do this next. →

rover says WOOF WOOF

spot says woof woof

spike says WOOF WOOF

lady says woof woof

> |

Costanti funzione e costruttore Function

- Esistono **costanti funzione** (function literal) che permettono di definire una funzione e poi di assegnarla ad una variabile con una sintassi decisamente inusuale:

```
var sum =  
    function(x,y) { return x+y; }
```

- Una funzione può essere anche creata usando un costruttore denominato **Function** (le funzioni sono quindi equivalenti in qualche modo agli oggetti)

```
var sum =  
    new Function("x","y","return x+y;");
```

- Es: **sum(4, 3)** ritorna il valore 7

JavaScript is pass-by-value

- JavaScript passes arguments to a function using ***pass-by-value***. What that means is that each argument is *copied* into the parameter variable

- 1 Let's declare a variable `age`, and initialize it to the value 7.

```
var age = 7;
```



- 2 Now let's declare a function `addOne`, with a parameter named `x`, that adds 1 to the value of `x`.

```
function addOne(x) {  
    x = x + 1;  
}
```



3

Now let's call the function `addOne`, pass it the variable `age` as the argument. The value in `age` is copied into the parameter `x`.

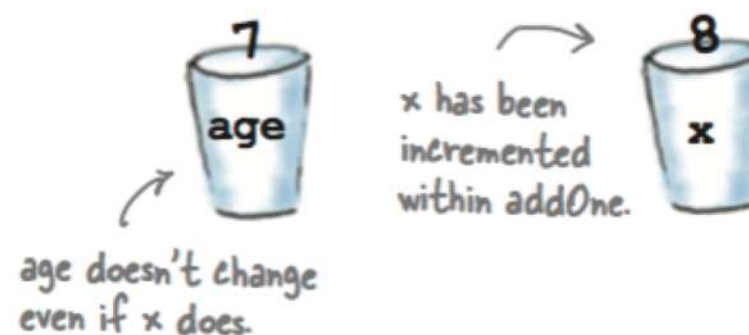
```
addOne (age) ;
```



4


Now the value of `x` is incremented by one. But remember `x` is a copy, so only `x` is incremented, not `age`.

```
function addOne(x) {  
  We're incrementing x. → x = x + 1;  
}
```





Passing objects to functions

- When an object is assigned to a variable, that variable holds a **reference** to the object, not the object itself
 - When you call a function and pass it an object, you're passing **the object reference**, not the object itself. So using our pass by value semantics, a copy of the reference is passed into the parameter, and that reference remains a pointer to the original object
 - **if you change a property of the object in a function, you're changing the property in the original object**
- 

Putting Fido on a diet...

- The dog parameter of the loseWeight function gets a copy of the reference to fido. So any changes to the properties of the parameter variable affect the object that was passed in:

```
var dog = {name: "charlie", weight: 10, ...}
```

When we pass fido into loseWeight, what gets assigned to the dog parameter is a copy of the reference, not a copy of the object. So fido and dog point to the same object.

The dog reference is a copy of the fido reference.



```
function loseWeight(dog, amount) {  
    dog.weight = dog.weight - amount;  
}
```


```
alert(fido.name + " now weighs " + fido.weight);
```

So, when we subtract 10 pounds from dog.weight, we're changing the value of fido.weight.





Variable scope

- **Globals live as long as the page**
 - **Local variables typically disappear when your function ends**
 - Local variables are created when your function is first called and live until the function returns (with a value or not)
- 

Metodi

- Quando una funzione viene assegnata ad una proprietà di un oggetto viene chiamata metodo dell'oggetto
- La cosa è possibile perché, come abbiamo visto, una funzione può essere assegnata ad una variabile
- In questo caso all'interno della funzione si può utilizzare la parola chiave **this** per accedere all'oggetto di cui la funzione è una proprietà

```
var o = new Object();  
o.x = 15;  
o.y = 7;  
o.tot = function() { return this.x + this.y; }  
alert(o.tot());
```

Costruttori

- Un costruttore è una funzione che ha come scopo quello di costruire un oggetto
- Se viene invocato con **new** riceve l'oggetto appena creato e può aggiungere proprietà e metodi
- L'oggetto da costruire è accessibile con la parola chiave **this**
- In qualche modo definisce il tipo di un oggetto

```
function Rectangle(w, h)
{
  this.w = w;
  this.h = h;
  this.area = function()
    { return this.w * this.h; }
  this.perimeter = function()
    { return 2*(this.w + this.h); }
}
```

```
var r = new Rectangle(5,4);
alert(r.area());
```

Proprietà e metodi statici

- JavaScript ammette l'esistenza di proprietà e metodi statici con lo stesso significato di Java
- Non esistendo le classi sono associati al costruttore
- Per esempio, se abbiamo definito il costruttore Circle() che serve per creare oggetti di tipo cerchio, possiamo aggiungere l'attributo PI in questo modo:

```
function Circle(r)
{
    this.r = r;
}
Circle.PI = 3.14159;
```

- Anche in Javascript esiste l'oggetto **Math** che definisce solo metodi statici corrispondenti alle varie funzioni matematiche

Function example: AreaAndDistance

- You can put your functions anywhere in your JavaScript file
- JavaScript doesn't care if your functions are declared before or after you use them

```
function setup(width, height) {  
    centerX = width/2;  
    centerY = height/2;  
}  
function computeDistance(x1, y1, x2, y2) {  
    var dx = x1 - x2;  
    var dy = y1 - y2;  
    var d2 = (dx * dx) + (dy * dy);  
    var d = Math.sqrt(d2);  
    return d;  
}  
function circleArea(r) {  
    var area = Math.PI * r * r;  
    return area;  
}  
setup(width, height);  
var area = circleArea(radius);  
var distance = computeDistance(x, y, centerX, centerY);  
alert("Area: " + area);  
alert("Distance: " + distance);
```


Events (only intro)

- An HTML event can be something the browser does, or something a user does
- Here are some examples of HTML events:
 - An HTML web page has finished loading
 - An HTML input field was changed
 - An HTML button was clicked
- Often, when events happen, you may want to do something
- JavaScript lets you execute code when events are detected
- HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements

```
<!DOCTYPE html>
```

buttontime.html

```
<html>
```

```
<body>
```

```
    <p>Click the button to display the date.</p>
```

```
    <button onclick="displayDate()">The time is?</button>
```

```
<script>
```

```
    function displayDate() {
```

```
        document.getElementById("demo").innerHTML = Date();
```

```
    }
```

```
</script>
```

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

```
</body>
```

```
</html>
```

Istruzioni

- Un **programma JavaScript** è una **sequenza di istruzioni**
- Buona parte delle istruzioni JavaScript hanno la stessa sintassi di C e Java
- Si dividono in:
 - **Espressioni** (uguali a Java): assegnamenti, invocazioni di funzioni e metodi, ecc.
 - **Istruzioni composte**: blocchi di istruzioni delimitate da parentesi graffe (uguali a Java)
 - **Istruzione vuota**: punto e virgola senza niente prima
 - **Istruzioni etichettate**: normali istruzioni con un etichetta davanti (sintassi: ***label: statement***)
 - **Strutture di controllo**: if, for, while, ecc.
 - **Definizioni e dichiarazioni**: var, function
 - **Istruzioni speciali**: break, continue, return

For

```
var scores = [60, 50, 60, 58, 54, 54,  
              58, 50, 52, 54, 48, 69,  
              34, 55, 51, 52, 44, 51,  
              69, 64, 66, 55, 52, 61,  
              46, 31, 57, 52, 44, 18,  
              41, 53, 55, 61, 51, 44];  
  
for (var i = 0; i < scores.length; i++) {  
    var output = "Bubble solution #" + i +  
                 " score: " + scores[i];  
  
    console.log(output);  
}
```

← All we've done is update where we increment the loop variable with the post-increment operator.

For (2)

- La struttura **for/in** permette di scorrere le proprietà di un oggetto (e quindi anche un array) con la sintassi:
- **for (variable in object) statement**

```
var x;  
var mycars = new Array();  
mycars[0] = "Panda";  
mycars[1] = "Uno";  
mycars[2] = "Punto";  
mycars[3] = "Clio";  
for (x in mycars)  
{  
    document.write(mycars[x]+"<br />");  
}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<p>The best way to loop through an array is using a standard for loop:</p>
```

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

```
<script>
```

```
    var fruits, text, fLen, i;
```

```
    fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
    fLen = fruits.length;
```

```
    text = "<ul>";
```

```
    for (i = 0; i < fLen; i++) {
```

```
        text += "<li>" + fruits[i] + "</li>";
```

```
    }
```

```
    text += "</ul>";
```

```
    document.getElementById("demo").innerHTML = text;
```

```
</script>
```

```
</body>
```

L'oggetto globale e funzioni predefinite

- In JavaScript esiste un oggetto **globale implicito**
- Tutte le variabili e le funzioni definite in una pagina appartengono all'oggetto globale
- Possono essere utilizzate senza indicare questo oggetto
- Questo oggetto espone anche alcune funzioni predefinite:
 - **eval(expr)** valuta la stringa expr (che contiene un'espressione Javascript)
 - **isFinite(number)** dice se il numero è finito
 - **isNaN(testValue)** dice se il valore è NaN
 - **parseInt(str [,radix])** converte la stringa str in un intero (in base radix – opzionale)
 - **parseFloat(str)** converte la stringa str in un numero

Array operations

- In javascript, arrays can have methods
 - Not functions to which you *pass* arrays, but methods *of* arrays

```
var nums = [1,2,3];  
nums.reverse(); → [3,2,1]
```

```
[1,2,3].reverse(); → [3,2,1]
```

- Most important methods
 - push, pop, concat
 - sort
 - forEach
 - map
 - filter
 - reduce

Push, pop, concat

- Push

```
var nums = [1,2,3];  
nums.push(4);  
nums; → [1,2,3,4]
```

- Pop

```
var val = nums.pop();  
val; → 4  
nums; → [1,2,3]
```

- Concat

```
var nums2 = nums.concat([4,5,6]);  
nums2; → [1,2,3,4,5,6]  
nums; → [1,2,3]
```

Pop (example)

```
<!DOCTYPE html>
<html>
<body>
  <p>The pop method removes the last element from an array.</p>
  <button onclick="myFunction()">Try it</button>
  <p id="demo"></p>
  <script>
    var fruits = ["Banana", "Orange", "Apple", "Mango"];
    document.getElementById("demo").innerHTML = fruits;
    function myFunction() {
      fruits.pop();
      document.getElementById("demo").innerHTML = fruits;
    }
  </script>
</body>
</html>
```

Sort

- With no arguments (default comparisons)
 - Note the odd behavior with numbers: they are sorted lexicographically, not numerically

```
["Hi", "bye", "hola", "adios"].sort();  
→ ["Adios", "bye", "hi", "hola"]
```

```
[1, -1, -2, 10, 11, 12, 9, 8].sort();  
→ [-1, -2, 1, 10, 11, 12, 8, 9]
```

Foreach

- Calls function on each element of array. Cannot break “loop” partway through
 - Lacks option to run in parallel that java 8 has

- Examples

```
[1,2,3].forEach(function(n) { alert(n); });
```

- Pops up alert box in page 3 times showing each number

```
[1,2,3].forEach(alert);
```

- Same as above.

- Summing an array (but reduce can also be used)

```
var nums = [1,2,3];
```

```
var sum = 0;
```

```
Nums.forEach(function(n) { sum += n; });
```

```
sum; → 6
```

Map

- Calls function on each element, then accumulates result array of each of the outputs. Returns new array; does not modify original array.
 - Like the java 8 “map” method, but not as powerful since the Javascript version does not support lazy evaluation or parallel operations.
- Examples

```
function square(n) { return(n * n); }
```

```
[1,2,3].map(square);
```

```
→ [1,4,9]
```

Filter

- Calls function on each element, keeps only the results that “pass” (return true for) the test. Returns new array; does not modify original array.
 - Like the java 8 “filter” method, but not as powerful since the javascript version does not support lazy evaluation or parallel operations.
- Examples

```
function isEven(n) { return(n % 2 == 0); }  
[1,2,3,4].filter(isEven); → [2, 4]
```


Reduce

- Takes function and starter value. Each time, passes accumulated result and next array element through function, until a single value is left.
 - Like the java 8 “reduce” method, but not as powerful since the javascript version does not support lazy evaluation or parallel operations.
- Examples

```
function add(n1,n2) { return(n1 + n2); }
```

```
function multiply(n1,n2) { return(n1 * n2); }
```

```
function bigger(n1,n2) { return(n1> n2 ? n1 : n2); }
```

```
var nums = [1,2,3,4];
```

```
var sum = nums.reduce(add, 0);    // 10
```

```
var product = nums.reduce(multiply, 1);    // 24
```

```
var max = nums.reduce(bigger, -Number.MAX_VALUE);    // 4
```

More array methods

- Slice

- Returns sub-array

`[9,10,11,12].slice(0, 2); → [9,10]`

`[1,2,3].slice(0); → [1,2,3] // makes copy of array`

- Reverse

- Reverses array (returns it, but also changes original)

`[1,2,3].reverse(); → [3,2,1]`

- Indexof

- Finds index of matching element

`[9,10,11].indexOf(10); → 1`

`[9,10,11].indexOf(12); → -1`

Splice

- The **splice()** method can be used to add new items to an array:

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");
```

- The first parameter (2) defines the position **where** new elements should be **added** (spliced in).
- The second parameter (0) defines **how many** elements should be **removed**.
- The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.



```
<!DOCTYPE html> (splice.html)
```

```
<html>
```

```
<body>
```

```
<p>The splice() method adds new elements to an array.</p>
```

```
<button onclick="myFunction()">Try it</button>
```

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

```
<script>
```

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
```

```
document.getElementById("demo").innerHTML = fruits;
```

```
function myFunction() {
```

```
    fruits.splice(2, 0, "Lemon", "Kiwi");
```

```
    document.getElementById("demo").innerHTML = fruits;
```

```
}
```

```
</script>
```

```
</body>
```

```
</html>
```

Esempio: carProps.html

```
<script>
```

```
var fiat = {  
  make: "Fiat",  
  model: "500",  
  year: 1957,  
  color: "Medium Blue",  
  passengers: 2,  
  convertible: false,  
  mileage: 88000,  
  started: false,  
  
  start: function() {  
    this.started = true;  
  },  
  
  stop: function() {  
    this.started = false;  
  },  
}
```

Esempio: carProps.html (cont)

```
drive: function() {  
    if (this.started) {  
        alert(this.make + " " +  
              this.model + " goes zoom zoom!");  
    } else {  
        alert("You need to start the engine first.");  
    }  
}  
};  
  
fiat.start();  
fiat.drive();  
  
for (prop in fiat) {  
    console.log(prop + ": " + fiat[prop]);  
}  
  
</script>
```

⌘ make: Fiat

⌘ model: 500

⌘ year: 1957

⌘ color: Medium Blue

⌘ passengers: 2

⌘ convertible: false

⌘ mileage: 88000

⌘ started: true

⌘ start: function () {
 this.started = true;
}

⌘ stop: function () {
 this.started = false;
}

⌘ drive: function () {
 if (this.started) {
 alert(this.make + " " +
 this.model + " goes zoom zoom!");
 } else {
 alert("You need to start the engine first.");
 }
}



JavaScript

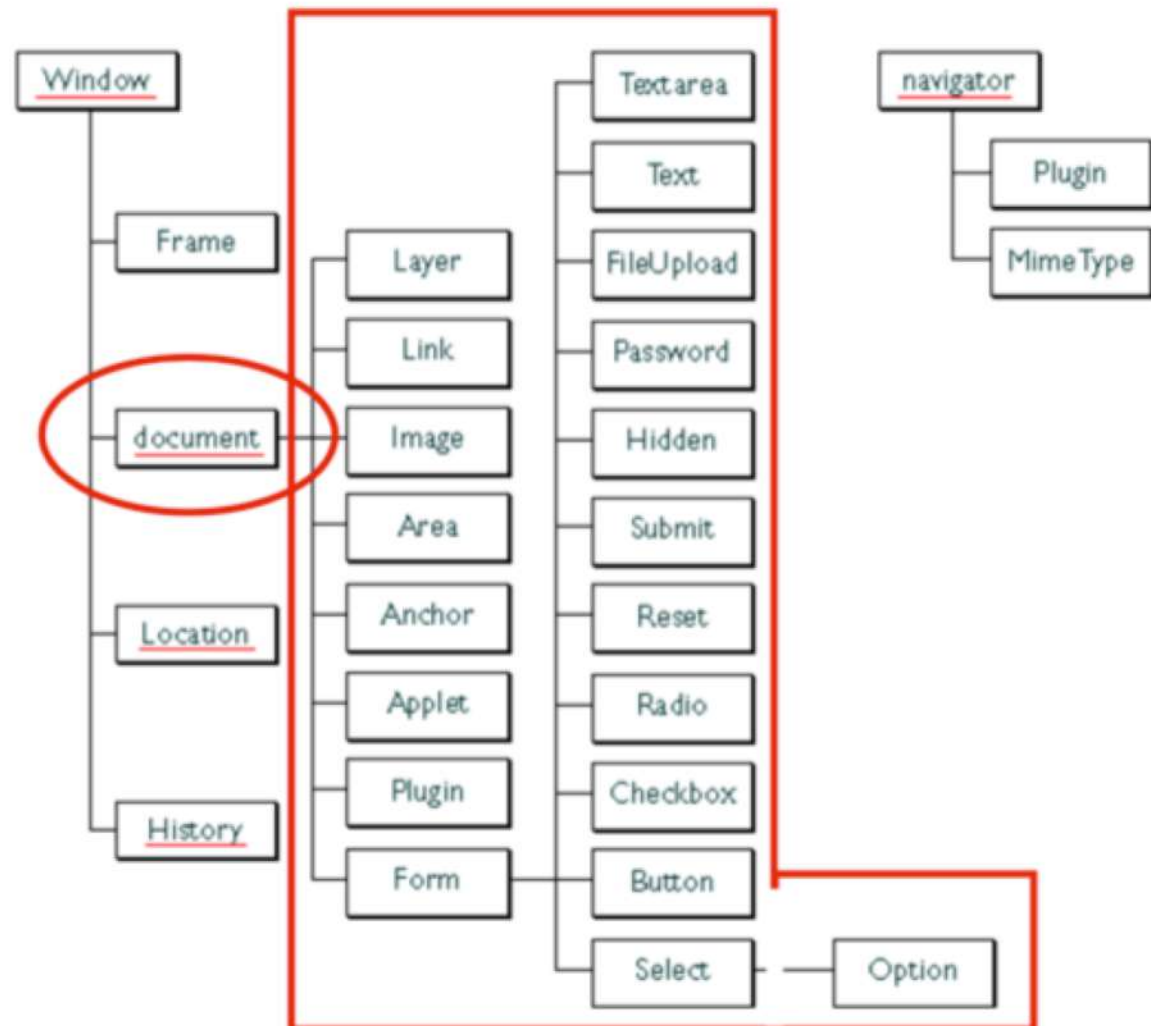
Fiat 500 goes zoom zoom!

OK

Browser Objects

- Per interagire con la pagina HTML , Javascript utilizza una gerarchia di oggetti predefiniti denominati Browser Objects e DOM Objects

La gerarchia che ha come radice document corrisponde al DOM



Costruzione dinamica della pagina

- La più semplice modalità di utilizzo di JavaScript consiste nell'inserire nel corpo della pagina script che generano dinamicamente parti della pagina
- Bisogna tener presente che questi script vengono eseguiti solo una volta durante il caricamento della pagina e quindi **non si ha interattività con l'utente**
- L'uso più comune è quello di generare pagine diverse in base al tipo di browser o alla risoluzione dello schermo
- La **pagina corrente** è rappresentata dall'oggetto **document**
- Per scrivere nella pagina si utilizzano i metodi **document.write()** e **document.writeln()**

Rilevazione del browser

- Per accedere ad informazioni sul browser si utilizza l'oggetto **navigator** che espone una serie di proprietà:

Proprietà	Descrizione
appName	Nome in codice del browser (poco utile)
appName	Nome del browser (es. Microsoft Internet Explorer)
appVersion	Versione del Browser (es. 5.0 (Windows))
cookieEnabled	Dice se i cookies sono abilitati
platform	Piattaforma per cui il browser è stato compilato (es. Win32)
userAgent	Stringa passata dal browser come header user-agent (es. "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1;)") È possibile esplorare la proprietà userAgent per mobile browser quali iPhone, iPad, o Android.

```
<html>
  <body>
    <script>
      document.write('Hello '+navigator.appName+'!<br>');
      document.write('Versione: '+navigator.appVersion+'<br>');
      document.write('Piattaforma: '+navigator.platform);
    </script>
  </body>
</html>
```

Rilevazione delle proprietà dello schermo

- L'oggetto **screen** permette di ricavare informazioni sullo schermo
- **screen** espone alcune utili proprietà tra cui segnaliamo **width** e **height** che permettono di ricavarne le dimensioni

```
<html>
  <body>
    <script>
      document.write('Schermo:
        '+screen.width+'x'+screen.height+' pixel<br>');
    </script>
  </body>
</html>
```

Schermo: 1360x768 pixel

Window Size

- Two properties can be used to determine the size of the browser window
- Both properties return the sizes in pixels:
 - **window.innerHeight** - the inner height of the browser window (in pixels)
 - **window.innerWidth** - the inner width of the browser window (in pixels)
- A practical JavaScript solution (covering all browsers):

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

<script>
var w = window.innerWidth
    || document.documentElement.clientWidth
    || document.body.clientWidth;

var h = window.innerHeight
    || document.documentElement.clientHeight
    || document.body.clientHeight;

var x = document.getElementById("demo");
x.innerHTML = "Browser inner window width: " + w + ", height: " + h + ".";
</script>
```

Modello ad eventi ed interattività

- Per avere una reale interattività bisogna utilizzare il meccanismo degli eventi
- JavaScript consente di associare script agli eventi causati dall'interazione dell'utente con la pagina
- L'associazione avviene mediante attributi collegati agli elementi della pagina HTML
- Gli script prendono il nome di gestori di eventi (**event handlers**)
- Nelle risposte agli eventi si può intervenire sul DOM modificando dinamicamente la struttura della pagina (DHTML)

DHTML = JavaScript + DOM + CSS

Eventi- 1

Evento	Applicabilità	Occorrenza	Event handler
Abort	Immagini	L'utente blocca il caricamento di un'immagine	onAbort
Blur	Finestre e tutti gli elementi dei form	L'utente toglie il focus a un elemento di un form o a una finestra	onBlur
Change	Campi di immissione di testo o liste di selezione	L'utente cambia il contenuto di un elemento	onChange
Click	Tutti i tipi di bottoni e i link	L'utente 'clicca' su un bottone o un link	onClick
DragDrop	Finestre	L'utente fa il drop di un oggetto in una finestra	onDragDrop
Error	Immagini, finestre	Errore durante il caricamento	onError
Focus	Finestre e tutti gli elementi dei form	L'utente dà il focus a un elemento di un form o a una finestra	onFocus
KeyDown	Documenti, immagini, link, campi di immissione di testo	L'utente preme un tasto	onKeyDown
KeyPress	Documenti, immagini, link, campi di immissione di testo	L'utente digita un tasto (pressione + rilascio)	onKeyPress
KeyUp	Documenti, immagini, link, campi di immissione di testo	L'utente rilascia un tasto	onKeyUp

Eventi-2

Evento	Applicabilità	Occorrenza	Event handler
Load	Corpo del documento	L'utente carica una pagina nel browser	onLoad
MouseDown	Documenti, bottoni, link	L'utente preme il bottone del mouse	onMouseDown
MouseMove	Di default nessun elemento	L'utente muove il cursore del mouse	onMouseMove
MouseOut	Mappe, link	Il cursore del mouse esce fuori da un link o da una mappa	onMouseOut
MouseOver	Link	Il cursore passa su un link	onMouseOver
MouseUp	Documenti, bottoni, link	L'utente rilascia il bottone del mouse	onMouseUp
Move	Windows	La finestra viene spostata	onMove
Reset	Form	L'utente resetta un form	onReset
Resize	Finestre	La finestra viene ridimensionata	onResize
Select	Campi di immissione di testo (input e textarea)	L'utente seleziona il campo	onSelect
Submit	Form	L'utente sottomette il form	onSubmit
Unload	Corpo del documento	L'utente esce dalla pagina	onUnload

Gestori di evento

- Come si è detto, per agganciare un gestore di evento ad un evento si utilizzano gli attributi degli elementi HTML

- La sintassi è:

<tag eventHandler="JavaScript Code">

- Esempio:

<input type="button" value="Calculate" onClick='alert("Calcolo")'/>

- È possibile inserire più istruzioni in sequenza, ma è meglio definire delle funzioni (in testata)
- È necessario alternare doppi apici e apice singolo

**<input type="button" value="Apri sesamo!"
onClick="window.open('myDoc.html','newWin')">**

The onkeyupEvent (onkeyup.html)

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
    var x = document.getElementById("fname");
    x.value = x.value.toUpperCase();
}
</script>
</head>
<body>
```

Enter your name: <input type="text" id="fname"
onkeyup="myFunction()">

<p>When you type a letter|, a function is triggered which
transforms the input text to upper case.</p>

```
</body>
</html>
```

The onmouseover and onmouseout Events

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

<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-
color:#D94A38;width:120px;height:20px;padding:40px;">
Mouse Over Me</div>

<script>
function mOver(obj) {
    obj.innerHTML = "Thank You"
}

function mOut(obj) {
    obj.innerHTML = "Mouse Over Me"
}
</script>

</body>
</html>
```

setInterval and clearInterval

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

<p>A script on this page starts this clock:</p>
<p id="demo" onclick="stopTimer()"></p>

<script>
var myVar = setInterval(function(){ myTimer() }, 1000);

function myTimer() {
    var d = new Date();
    var t = d.toLocaleTimeString();
    document.getElementById("demo").innerHTML = t;
}

function stopTimer() {
    clearInterval(myVar);
}
</script>

</body>
</html>
```

Esempio: calcolatrice

```
<head>
  <script type="text/javascript">
    function compute(f)
    {
      if (confirm("Sei sicuro?"))
        f.result.value = eval(f.expr.value);
      else alert("Ok come non detto");
    }
  </script>
</head>
<body>
  <form>
    Inserisci un'espressione:
    <input type="text" name="expr" size=15 >
    <input type="button" value="Calcola"
      onClick="compute(this.form)" ><br/>
    Risultato:
    <input type="text" name="result" size="15" >
  </form>
</body>
```

Inserisci un'espressione: 3*2

Calcola

Risultato: 6

Esplorare il DOM: Document

- Il punto di partenza per accedere al Documento Object Model (DOM) della pagina è l'oggetto **document**
- **document** espone 4 collezioni di oggetti che rappresentano gli elementi di primo livello:
 - **anchors[]**
 - **forms[]**
 - **images[]**
 - **links[]**
- L'accesso agli elementi delle collezioni può avvenire per indice (ordine di definizione nella pagina) o per nome (attributo **name** dell'elemento):
 - document.links[0]**
 - document.links["*nomelink*"]**
- Può essere scritta anche come **document.*nomelink***

Document

- Metodi:
 - **getElementById()**: restituisce un riferimento al primo oggetto della pagina avente l'id specificato come argomento
 - **write()**: scrive un pezzo di testo nel documento
 - **writeln()**: come write() ma aggiunge un a capo
- Proprietà:
 - **bgColor**: colore di sfondo
 - **fgColor**: colore di primo piano
 - **lastModified**: data e ora di ultima modifica
 - **cookie**: tutti i cookies associati al document
 - rappresentati da una stringa di coppie: nome-valore title: titolo del documento
 - **URL**: url del documento

Document (2)

- **getElementById()** returns the element that has the ID attribute with the specified value
- **getElementsByClassName()** returns a NodeList containing all elements with the specified class name
- **getElementsByTagName()** returns a NodeList containing all elements with a specified name
- **getElementsByTagName()** returns a NodeList containing all elements with the specified tag name

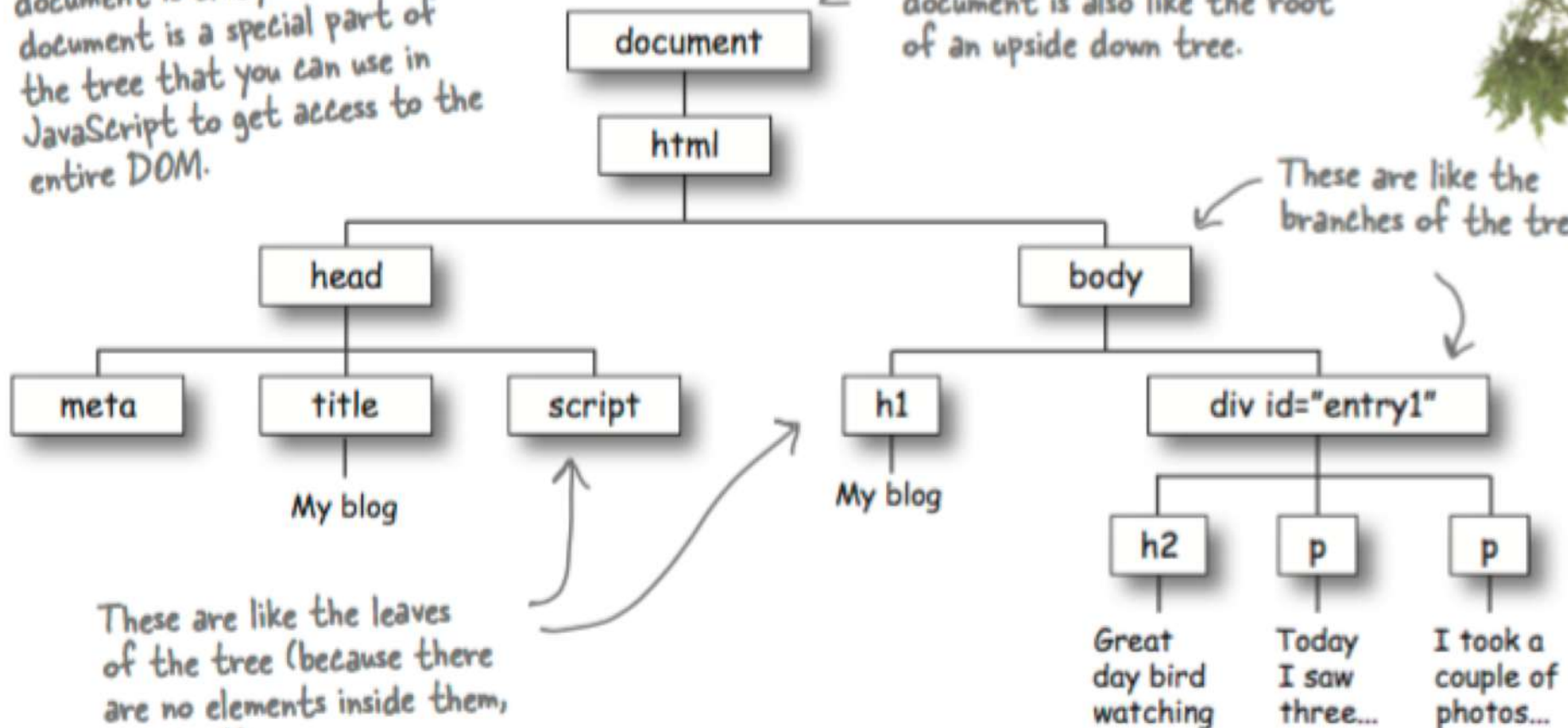
```
var x = document.getElementsByTagName("P");
var i;
for (i = 0; i < x.length; i++) {
    x[i].style.backgroundColor = "red";
}
```

The DOM

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>My blog</title>
  <script src="blog.js"></script>
</head>
<body>
  <h1>My blog</h1>
  <div id="entry1">
    <h2>Great day bird watching</h2>
    <p>
      Today I saw three ducks!
      I named them
      Huey, Louie, and Dewey.
    </p>
    <p>
      I took a couple of photos...
    </p>
  </div>
</body>
</html>
```


document is always at the top.
document is a special part of
the tree that you can use in
JavaScript to get access to the
entire DOM.

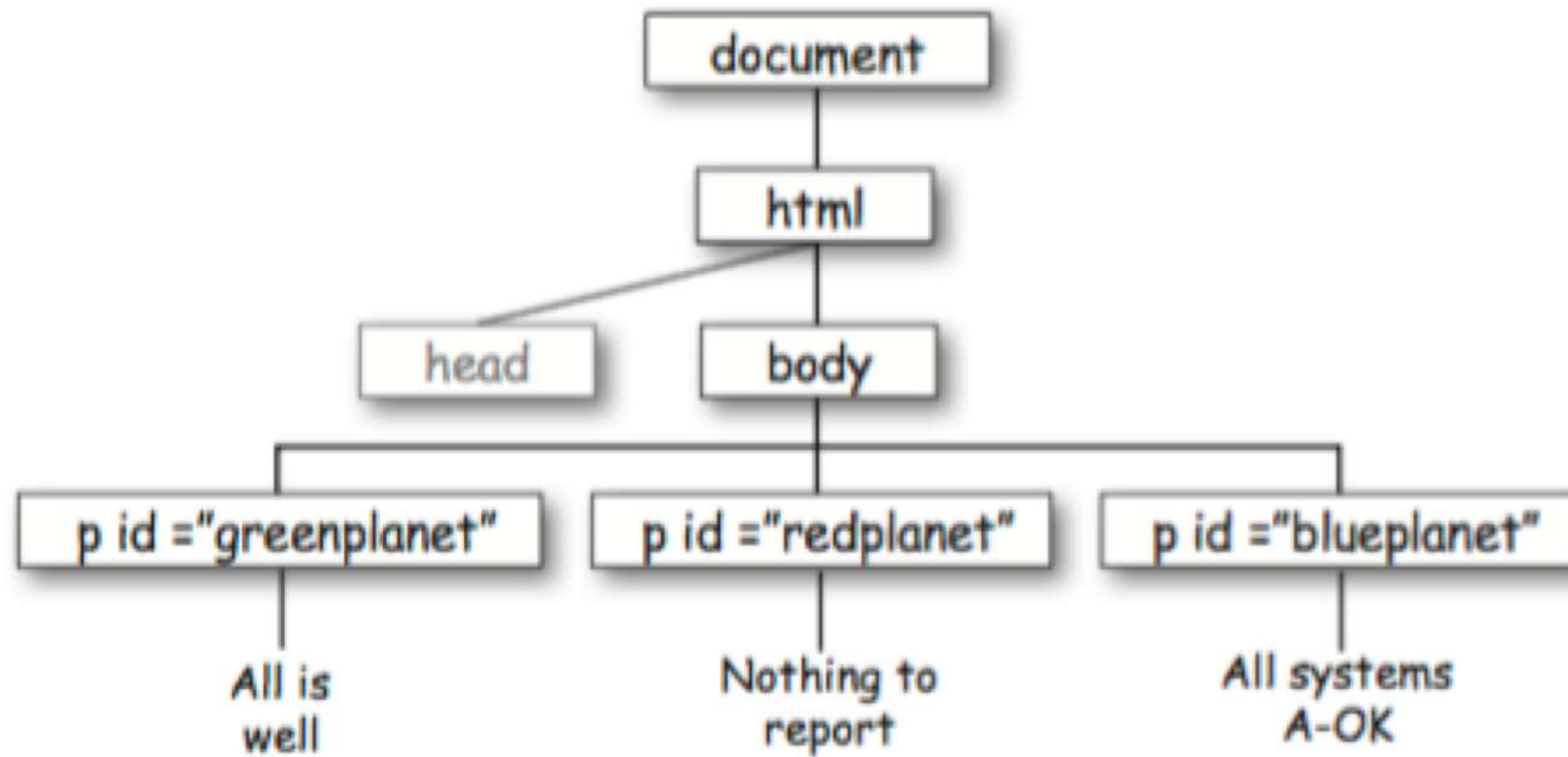
document is also like the root
of an upside down tree.



These are like the leaves
of the tree (because there
are no elements inside them,
just text).

The DOM includes the content of the page as well as the
elements. (We don't always show all the text content when
we draw the DOM, but it's there).

Example



We're assigning the element to a variable named planet.

Here's our call to `getElementById`, which seeks out the "greenplanet" element and returns it.

↓

```
var planet = document.getElementById("greenplanet");
```

↓

And in our code we can now just use the variable planet to refer to our element.

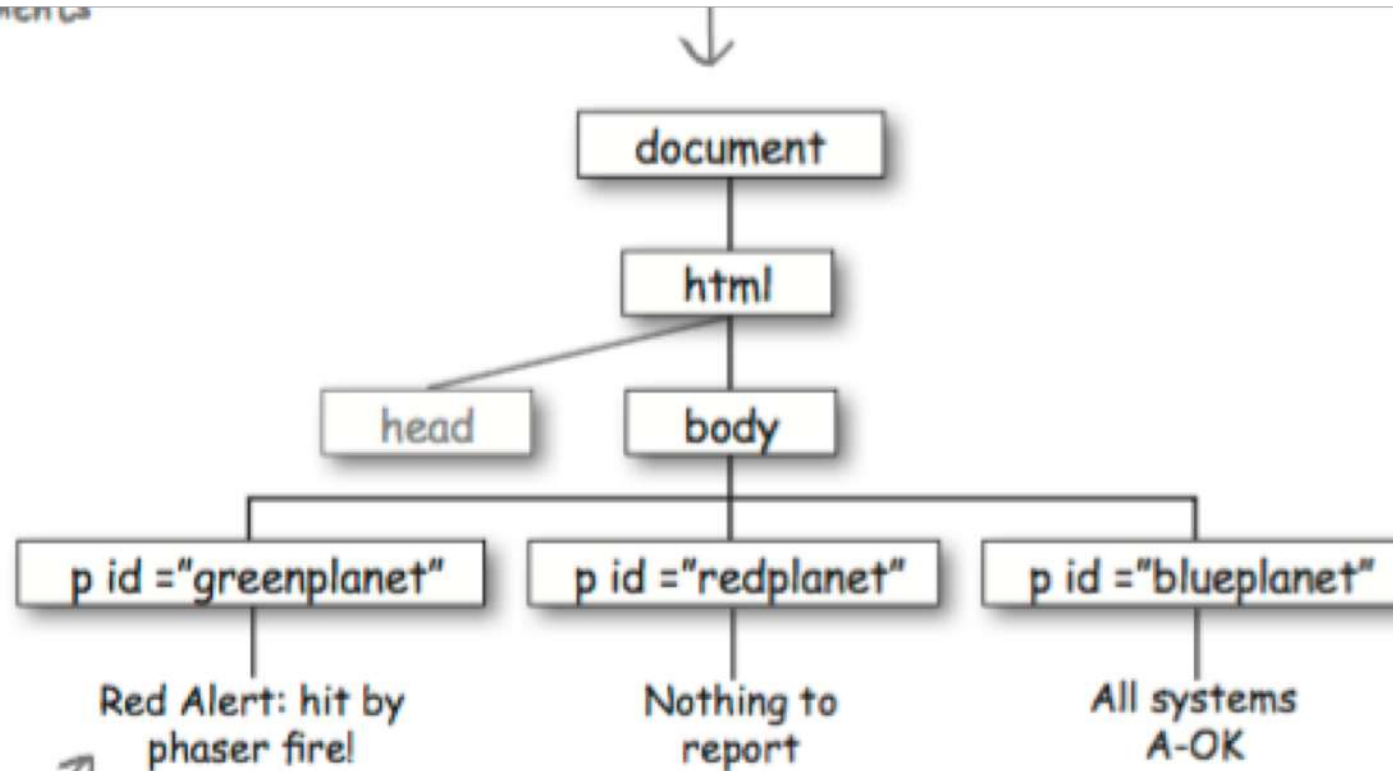
↓

```
planet.innerHTML = "Red Alert: hit by phaser fire!";
```

We can use the `innerHTML` property of our planet element to change the content of the element.

We change the content of the greenplanet element to our new text... which results in the DOM (and your page) being updated with the new text.

The dome has been changed



Any changes to the DOM are reflected in the browser's rendering of the page, so you'll see the paragraph change to contain the new content!

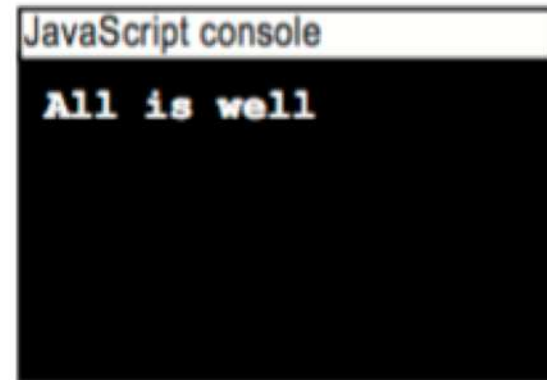
Finding the inner HTML

- The innerHTML property is an important property that we can use to read or replace the content of an element

```
var planet = document.getElementById("greenplanet");  
console.log(planet.innerHTML);
```

We're just passing the planet.innerHTML property to console.log to log to the console.

The content of the innerHTML property is just a string, so it displays just like any other string in the console.



Changing the inner HTML

```
var planet = document.getElementById("greenplanet");  
planet.innerHTML = "Red Alert: hit by phaser fire!";  
console.log(planet.innerHTML);
```

Now we're changing the content of the element by setting its `innerHTML` property to the string "Red Alert: hit by phaser fire!"

JavaScript console

Red Alert: hit by
phaser fire!

So when we log the value of the `innerHTML` property to the console we see the new value.

And the web page changes too!



Important!

- When you're dealing with the DOM it's important to execute your code only *after* the page is *fully loaded*. If you don't, there's a good chance the DOM won't be created by the time your code executes
- First create a function that has the code you'd like executed *once the page is fully loaded*
- Next, you take the **window object**, and assign the function to its **onload** property

Onload (the “page is loaded” event)

```
<script>
```

```
function init() {
```

```
    var planet = document.getElementById("greenplanet");
```

```
    planet.innerHTML = "Red Alert: hit by phaser fire!";
```

```
}
```

```
window.onload = init;
```

```
</script>
```

First, create a function named `init` and put your existing code in the function.

You can call this function anything you want, but it's often called `init` by convention.

Here's the code we had before, only now it's in the body of the `init` function.

Here, we're assigning the function `init` to the `window.onload` property. Make sure you don't use parentheses after the function name! We're not calling the function; we're just assigning the function value to the `window.onload` property.

How to set an attribute with setAttribute

- Element objects have a method named `setAttribute` that you can call to set the value of an HTML element's attribute

We take our element object.

`planet.setAttribute("class", "redtext");`

Note if the attribute doesn't exist a new one will be created in the element.

And we use its `setAttribute` method to either add a new attribute or change an existing attribute.

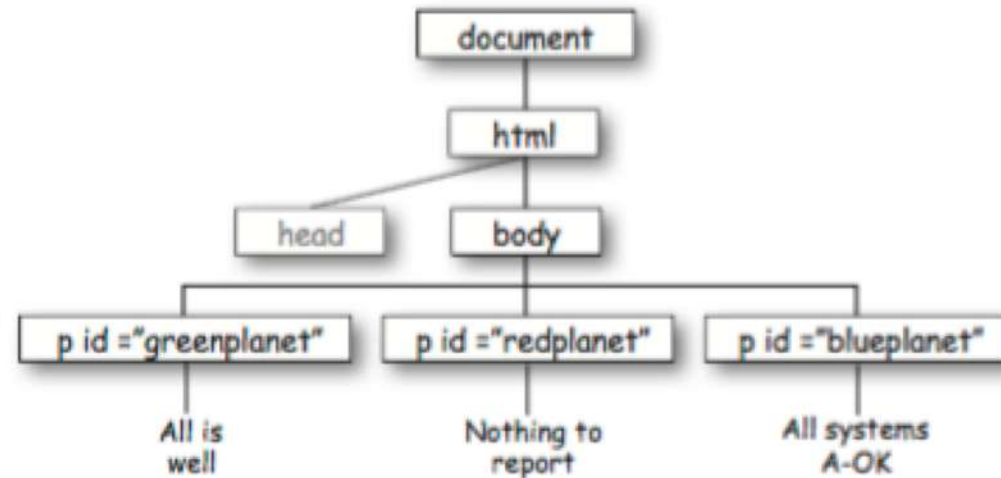
The method takes two arguments, the name of the attribute you want to set or change... ... and the value you'd like to set that attribute to.

```
element.setAttribute("style", "background-color: red;");
```

```
element.style.backgroundColor = "red";
```

Before...

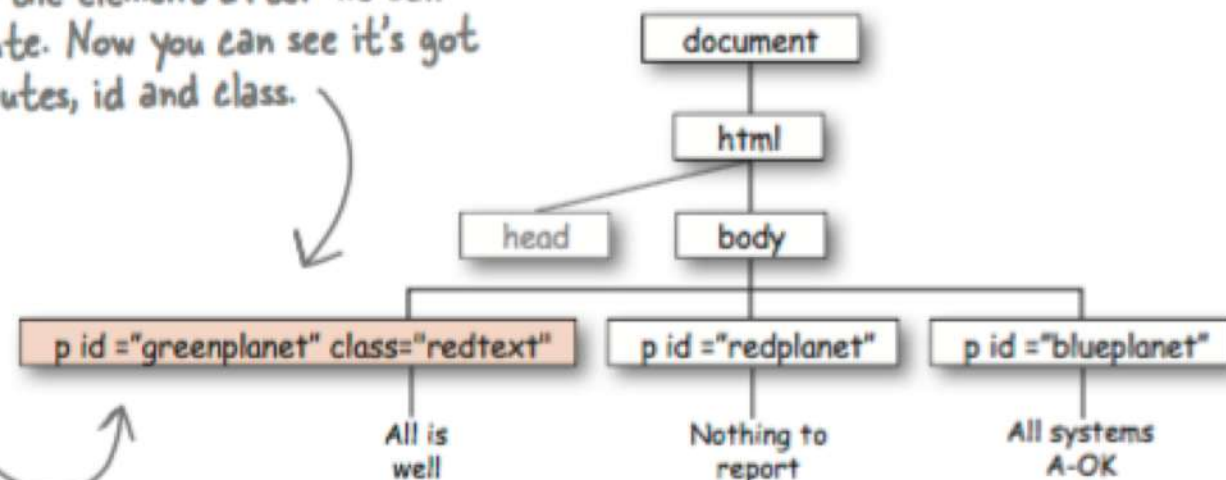
Here's the element before we call the `setAttribute` method on it. Notice this element already has one attribute, `id`.



And After

And here's the element after we call `setAttribute`. Now you can see it's got two attributes, `id` and `class`.

Remember, when we call the `setAttribute` method, we're changing the element object in the DOM, which immediately changes what you see displayed in the browser.



<head>

<meta charset="utf-8">

<title>Planets</title>

<style>

.redtext { color: red; }

</style>

<script>

function init() {

var planet = document.getElementById("greenplanet");

planet.innerHTML = "Red Alert: hit by phaser fire!";

planet.setAttribute("class", "redtext");

}

window.onload = init;

</script>

</head>

We've got the redtext class included here so when we add "redtext" as the value for the class attribute in our code, it turns the text red.

And to review: we're getting the greenplanet element, and stashing the value in the planet variable. Then we're changing the content of the element, and finally adding a class attribute that will turn the text of the element red.

We're calling the init function only when the page is fully loaded!

DOM is good for...

Get elements from the DOM

- Use `document.getElementById`
- Use tag names, class names and attributes to retrieve not just one element, but a whole set of elements (say all elements in the class “on_sale”)
- Get form values the user has typed in, like the text of an input element

Create and add elements to the DOM

- You can create new elements and you can also add those elements to the DOM. Of course, any changes you make to the DOM will show up immediately as the DOM is rendered by the browser

DOM is good for... (2)

Remove elements from the DOM

- You can also remove elements from the DOM by taking a parent element and removing any of its children
- You'll see the element removed in your browser window as soon as it is deleted from the DOM

Traverse the elements in the DOM

- Once you have a handle to an element, you can find all its children, you can get its siblings (all the elements at the same level), and you can get its parent. The DOM is structured just like a family tree!

Navigating Between Nodes

- You can use the following node properties to navigate between nodes with JavaScript:
 - **parentNode**
 - **childNodes[nodenum]**
 - **firstChild**
 - **lastChild**
 - **nextSibling**
 - **previousSibling**
- **Warning !**
 - A common error in DOM processing is to expect an element node to contain text
 - In this example: `<title>DOM Tutorial</title>`, the element node `<title>` does not contain text. It contains a text node with the value "DOM Tutorial"
 - The value of the text node can be accessed by the node's `innerHTML` property, or the `nodeValue`

Creating new HTML Elements - insertBefore()

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

    <ul id="myList">
      <li>Coffee</li>
      <li>Tea</li>
    </ul>

    <p>Click the button to insert an item to the list.</p>

    <button onclick="myFunction()">Try it</button>

    <p><strong>Example explained:</strong><br>First create a LI node,<br> then create a Text node,<br> then
      append the Text node to the LI node.<br>Finally insert the LI node before the first child node in
      the list.</p>

    <script>
      function myFunction() {
        var newItem = document.createElement("LI");
        var textnode = document.createTextNode("Water");
        newItem.appendChild(textnode);

        var list = document.getElementById("myList");
        list.insertBefore(newItem, list.childNodes[0]);
      }
    </script>

  </body>
</html>
```

Removing Existing HTML Elements

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

    <!-- Note that the <li> elements inside <ul> are not indented
         (whitespaces).
         If they were, the first child node of <ul> would be a text node
         -->
    <ul id="myList"><li>Coffee</li><li>Tea</li><li>Milk</li></ul>

    <p>Click the button to remove the first item from the list.</p>

    <button onclick="myFunction()">Try it</button>

    <script>
      function myFunction() {
        var list = document.getElementById("myList");
        list.removeChild(list.childNodes[0]);
      }
    </script>

  </body>
</html>
```


JavaScript can Change HTML Attributes

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

<h1>JavaScript Can Change Images</h1>



<p>Click the light bulb to turn on/off the light.</p>

<script>
function changeImage() {
    var image = document.getElementById('myImage');
    if (image.src.match("bulbon")) {
        image.src = "pic_bulboff.gif";
    } else {
        image.src = "pic_bulbon.gif";
    }
}
</script>

</body>
</html>
```

JavaScript Can Change HTML Styles (CSS)

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

<h1>What Can JavaScript Do?</h1>

<p id="demo">JavaScript can change the style of an HTML
element.</p>

<button type="button" onclick="myFunction()">Click Me!</button>

<script>
function myFunction() {
    var x = document.getElementById("demo");
    x.style.fontSize = "25px";
    x.style.color = "red";
}
</script>

</body>
</html>
```