HTML 5 Shooters

Ted Hagos
ted@thelogbox.com

November 2012

Contents

This is front matter

Preface

1 | HTML Basics

HTML stands *Hypertext Markup language*. It is not a programming language like Java, Javascript or C++ are programming languages, but it is a language nonetheless. A markup language consists of pair of tags. These tags are instructions to whatever will read HTML document. Browsers do read HTML documents, so the markup is intended (mostly) for browsers. There are other consumers of HTML documents but for our purposes we will consider only browsers. `What is HTML`

Markup tags are instructions, a processor like a web browser will know what to do with the markups, e.g. `<h1>Heading One</h1>` tells the browser the text "Heading One" should be rendered differently from the default font and weight of the paragraph body, it should be rendered with more weight and size emphasis. HTML documents can contain both marked up and non-marked up (regular text)

HTML documents are ubiquituous, they have a wide range of use, from documentation, apps etc, but most of the time, HTML documents are used for building web pages.

HTML tags are special words written in between a pair of angle brackets; the less than and greater than signs e.g. `|<emph>emphasis</emph>|`. They come in pairs, notice that the other tag has forward slash? That is how to close a tag. Not all browsers will be strict in processing the HTML tags, some browsers will let you get away with improperly written tags, but that is bad form—strive to always write proper HTML. HTML tags are also known as HTML elements, whenever you see the term HTML element, it means exactly the same thing as an HTML tag `HTML tags`

HTML documents are text files. They can be created by any editor capable of processing plain text files such as Notepad, TextEdit, nano, pico, GEdit. While some people swear that a plain text editor is the only thing you need to create web pages, I cannot endorse that. When you start working with non-trivial projects, a decent and capable editor will go a long way. While you haven't settled on your own editor, this list can you started `HTML files are just text files`

1. **CoffeeCup** http://www.coffeecup.com/free-editor/
2. **HTML Kit** http://www.htmlkit.com
3. **NotePad++** http://notepad-plus-plus.org
4. **PSPad** http://www.pspad.com
5. **Scite** http://opensource.ebswift.com/SciTEInstaller/
6. **eTextEditor** - http://e-texteditor.en.softonic.com
7. **Smultron** (Commercial Software, for OSX)

8. **TextMate** (Commercial Software, for OSX)
9. **Sublime2** (Commercial Software, for OSX)
10. **Geany** (Linux software)

There are others like Aptana Studio, Netbeans etc, but they are more than just plain editors, they are IDEs (Integrated Development Environment). You can try them too, but you need to invest time in learning how to use them properly. A plain text editor is low on footprint and low on ceremony, just get up and go.

1.1 Structure of an HTML document

Listing 1.1: HTML Document Structure

```
<!DOCTYPE html>
<html>
    <head>
    </head>

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

The `<html>` element is the outermost element in a document, there can only be one pair of this element inside an HTML source file. There are two elements that can be nested inside the **html tag**, the `<head> and the <body>` elements

All the visible things that you want to show on an HTML page goes to the **body** element. For example, ordered lists, un-ordered lists, headings, paragraphs, tables.

Things written inside the **head** element will not be visible on the HTML page. That is not to say they are less important. Some of the things you may find on the **head** element are *script*–for embedding Javascript codes, *link*–for CSS amongst other things, *title* etc

The `<!DOCTYPE>` is not an HTML tag, it is an instruction to the processor (the web browser) that contains which version of HTML the document is using. This declaration is necessary because there is more than one version of HTML. The HTML technology has been evolving for the past two decades. The current version of HTML is 5, but it doesn't mean that all web pages are now using HTML version 5. Some web pages are still coded using previous versions of HTML, that is the reason why the browser needs to know which version it is dealing with. The following are the versions of HTML:

1. **HTML** - 1991
2. **HTML+** - 1993
3. **HTML 2.0** - 1995
4. **HTML 3.2** - 1997
5. **HTML 4.01** - 1999
6. **XHTML 1.0** - 2000

7. **HTML5** - 2012

8. **XHTML5** - 2013

The W3C site keeps an updated list of HTML versions [1].

The way to declare a DOCTYPE in HTML 5 is `<!DOCTYPE html>`, you can already use this now. Current browsers will look at this doctype and switch to standards mode. What this means is that the HTML5 pages you will write today can last for a very long time.

If you want to look at doctype declarations for other versions of HTML, W3C keeps an updated page on DOCTYPES. [2]

[1] HTML versions http://www.w3schools.com/html/html_intro.asp
[2] DOCTYPE declarations http://www.w3schools.com/tags/tag_doctype.asp

2 | HTML5 Introduction

HTML5 will be the new standard for HTML, it is the result of the cooperation between the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG).

The goals of HTML5 are:

1. New features are based on HTML, CSS, DOm and JavaScript
2. Reduce dependencies on external plugins e.g. Flash
3. Improved error handling
4. Favor markup over scripting (when possible)
5. Device independence
6. Development process should be publicly visible

2.1 New features

1. Canvas elements
2. Video and Audio elements
3. Local storage
4. Content specific elements like article, footer, header, nav and section
5. New form controls e.g. calendar, date, email, url and search

3 | Structural Elements

4 | JavaScript

JS (JavaScript) is a loose, dynamically typed and interpreted scripting language. It began life as LiveScript (1995); when it was a project at Netscape Communications. At some point in time, it was renamed JavaScript (roughly around the time when plugin support for the Java language was added to Netscape Navigator).

Java and JavaScript are not related to each other, not even remotely, they simply share a commonality in their names.

JavaScript's popularity maybe attributed to it's ubiquity amongst web pages (client side JavaScript), but it is actually being used in lots of places other than web pages. JS can be ran on the server (Node.JS), it can be used to do MVC type development of web apps (Backbone.JS) and it can also be used to stitch together a bunch of Java apps together (Rhino); these are just some of the other uses of JavaScript.

`ECMAScript 262 Specification`

JavaScript as we know it today, is the scripting language consistent with ECMAScript. ECMAScript is a standardized scripting language according to ECMA specification 262 and ISO/IEC 16262. ECMAScript exist in the form of JavaScript, ActionScript and JScript. If you are looking for language specification of JavaScript, you can look at the ECMAScript 262 specification

4.1 Code Structure

1. **Variable** - a temporary storage for something. It can hold any type of data
2. **Literals** - e.g. 10 is Number literal, "Hello" is a string literal
3. **Expression** - involves operand(s) and operators. e.g. *a + b or 1 + a or 15 % 12*
4. **Statement** - similar to expressions. They are (optionally) terminated by semicolon. They can be written either inside a function or outside a function
5. **Function (method)** - a programming block that (usually) have a name. A programming block is a bunch of statements grouped together and enclosed inside a pair of curly braces
6. **Source file** - contains JS functions and statements. Usually has a *.js* extension

4.2 Data Types and Variable Declaration

You need to know what kinds of stuff you can create, define and manipulate in JS.

1. Number
2. Boolean
3. String
4. Array
5. Object
6. Function
7. undefined
8. null
9. Object

The first three are simple data types in JS, the rest are not so simple (don't worry, we will try to define the *not so simple* types a bit down the road)

Number - 10, 2, 100 (integers), 2.3, 1.2 (floats) are numbers. JS does not discriminate whether it is of integral or float type. There's more; 0xFF, 0xCAFEBABE (hex or octal) are also considered numbers, and so are base 8 numbers (i.e. 0377)

When you try to store a value that is more than what a Number can hold, it can result to either *Infinity* or *-Infinity*. You can also get some weird results is *NaN* (not a number). The NaN can result if you try to do some illegal operations, i.e. divide something by zero.

Boolean - either *true* or *false*

String - A sequence of UNICODE characters, although, there is no concept of character arrays in JS. You can define Strings using either the single quote or double quote technique. i.e. "Hello World" is as good as 'Hello World'. You may insert escape characters in Strings like `"Hello\n World"`.

We will deal with the other data types a little bit later, these three are actually enough to keep us busy for now.

You will usually bother with data types when you are either declaring variables or returning something from functions. If you need to see for your eyes how the JS runtime really treats this variable, this small code snippet should help

```
/*

A small test on how JS treats types

*/

var s = "hello";
var int = 1;
var float = 1.0;
var boolean = true;
var vnull = null;
var vundefined = undefined;
```

```
var obj = new Object();
var arr = new Array();

function x() {
    return 1;
}

function y() {
    return
}

function z() {

}

// Let's see the typeof results

console.log("s " + typeof(s));
console.log("int " + typeof(int));
console.log("float " + typeof(float));
console.log("boolean " + typeof(boolean));
console.log("vnull " + typeof(vnull));
console.log("vundefined " + typeof(vundefined));
console.log("obj " + typeof(obj));
console.log("arr " + typeof(arr));
console.log("function x " + typeof(x()));
console.log("function y " + typeof(y()));
console.log("function z " + typeof(z()));
console.log("wildcard " + typeof(wildcard));
/*

RETURNS:
s string
int number
float number
boolean boolean
vnull object
vundefined undefined
obj object
arr object
function x number
function y undefined
function z undefined
wildcard undefined
*/
```

4.3 Functions

A JS function is named code-block. A code block is a group of statements tucked inside a pair of opening and ending curly brace.

```
/*
File name: hello.js
The hello world program in JavaScript
*/

foo();
```

```
function foo() {
    var s = "Hello World\n";
    console.log(s);
}
```

If you installed node.js (and I hope you did) you can run try this out on the command line using **$ node hello.js**

The hello.js program is actually more verbose than it has to be, it could have been written as

```
console.log("Hello World");
```

I wrote it that way to demonstrate a few more JS characteristics. /* */ are comments. Comments are non-executable codes, the interpreter will ignore them, but they are useful in putting reminders or markers in your code. It can help others read your code as well.

Functions are defined in JavaScript by using the word **function** then give it a name (hopefully a descriptive one), put open and close parentheses immediately after the function name, then put the pair of curly braces.

The *console.log()* command is the STDIO library of Node.JS (although some browsers include this as part of their JavaScript runtime). This command routes the output to stderr and stdout (the command line screen)

```
CANONICAL FORM OF THE FUNCTION

function <name> (<parameters) {
    // statements
[return <value>]
}
```

Functions can contain a *return* statement, take note of the following though;

1. Calling a function without a return statement - returns undefined
2. Calling a function with return statement, but does actually return anything - returns undefined
3. Calling a function with a return statement and a proper return value - returns the value

CAUTION: careful how you write your curly braces, JavaScript's defaults might produce unexpected results, see the following code

```
function goo()
{

    return
    {                                   // this line is unreachable
        objprop: "Some obj property";
    }

}
```

The statement will be unreachable because JavaScript would have inserted a semi-colon right after the **return** statement. The object literal is unreachable as a return value.

Functions are different from JS statements. Statements are executed at once as soon as it is encountered by interpreter. Functions on the other hand are *compiled* first, they are not executed when first encountered, intead, it will be parsed and stored for later use.

Theres a lot more to JavaScript functions but we will stop here for now. The other characteristics of functions are better explained when we are the context of Object Oriented JavaScript and some more advanced concepts.

4.4 Keywords

When you create identifiers, e.g. variable names, function names and class names, you need to stay away from certain words–the JavaScript reserved words. Here they are;

- break case catch continue debugger default
- delete do else finally for function if in
- instanceof new return switch this throw try
- typeof var void while with implements
- interface let package private protected
- public static yield class enum export
- extends import super

Read the ECMA specification for keywords, I just listed them here for convenience.

4.5 Arithmetic Operators

These operators behave exactly as they behave in your calculator. + for adding, - subtracting, * multiplying and / for performing division. The only other operator in the arithmetic arsenal is the modulo (%) operator. It also performs divsion, but instead of giving you the quotient, it gives you the remainder. e.g. *156 % 12* gives you 3

4.6 Boolean operators

These operators allows you to compare things logically and do some equivalence tests. Boolean expresssions yields either true or false.

The *&&* (logical AND), || (logical OR) and the *unary NOT !* operator allows you to perform logical comparissons.

The **&&** operator yields true only if both operands are result to true. If you are coming from another language where this symbol has a short-circuit behavior, be careful. This operator evaluates both operands before yielding the truth value.

```
/*
    Simple test for && operator
*/


console.log(a() && b());

function a() {
    console.log("inside a");
    return true;
}

function b() {
    console.log("inside b");
    return false;
}

RESULT:

inside a
inside b
false
```

the **||** operator on the other hand requires that only one of the operands is true for the expression to return true.

```
/*
    Simple test for || operator
*/


console.log(a() || b());

function a() {
    console.log("inside a");
    return true;
}

function b() {
    console.log("inside b");
    return false;
}

RESULT:

inside a
true
```

The || operator did not even evaluate function b() before returning the truth value.

The other boolean operators **== != > < <= >=** and **!** are straight forward, you can read them in the ECMAScript specification if you prefer.

4.7 Control of program flow

You will need to know how to direct program flow. There are 3 ways; execute statements one after another (sequencing), execute some statements when some condition is true (branching) and performing some statements repeatedly while some conditions are true (looping or iteration).

4.7.1 if-else statement

if-else is way to branch or reroute program control. When the expression inside the **if()** construct yields true, the statement inside the if block is executed.

```
var i = 10;

if (i == 20) {
    console.log("true");
}
else {
    console.log("nah!");
}

RETURNS nah!
```

Be careful when writing your comparisson operator. Here's a slightly modified version of the code above

```
var i = 10;

if (i = 20) {
    console.log("true");
}
else {
    console.log("nah!");
}

RETURNS true
```

Can you spot the difference? The expression inside the if() construct is improperly written as an assignment rather than a test for equality–but JavaScript went to evaluate it anyway. Be careful with this one because this mistake is so easy to make. The reason this returned **true** is because an arithmetic assignment operation returns the zero value, and if you use a number where a boolean expression is expected, the number is convered to the *true* value.

4.7.2 Switch Statement

Switch statement is another way of branching. It is a bit more concise than the if-else statement, it allows for a different kind of construction.

```
var s = "hello";
```

```
var ret;

switch (s) {

    case "he":
        ret = "he";
        break;
    case 1:
        ret = 1;
        break;
    case true:
        ret = "true";
        break;
    case "Hello".toLowerCase():
        ret = "Hello";
        break;
    default:
        ret = "nothing";
}

console.log(ret);

/*

RETURNS "Hello"

*/
```

The values listed in *case* statements can be Strings, booleans or numbers (a bit of a relief if you are coming the Java language where the case is restricted to integer values). Always add the *break* keyword inside the case statement lest you will have unintended consequences–without the break keyword, the program will overflow to the remaining statements of the *switch construct* even if a match has already been found.

4.7.3 While Statement

While statement is a looping mechanism. It requires a boolean value (or an expression that yields a boolean value) inside the **while()** construct. The statements inside the while block will be executed repeatedly as long as the expression inside while() is true. When the expression is no longer true, the loop terminates.

```
var timetoexit = false;
var counter = 0;

while (!timetoexit) {
    console.log("counter = " + counter);
    if (counter++ == 10) timetoexit = true;
}
```

You can use the **break** keyword to break out of loops, like this

```
var timetoexit = false;
var counter = 0;

while (!timetoexit) {
```

```
    console.log("counter = " + counter);
    if (counter++ == 10) break;
}
```

The break keyword causes the program flow to ignore all the statements (if there are any) immediately following the *break* keyword and fall right outside the end of the while block where the program flow will resume.

Another keyword related to loop structures is the **continue** keyword. It causes program flow to jump also

```
var timetoexit = false;
var counter = 0;

while (!timetoexit) {

    console.log("counter = " + counter);
    if (++counter == 10) {
        timetoexit = true;
        continue;
    }
    console.log("*");
}
```

Unlike *break* though, **continue** causes program flow to ignore all the remaining statements until the end of the while block and jump right away to the beginning of the while loop—forcing the condition to get re-evaluated; this is the reason why our last asterisk never got printed.

4.7.4 For Statement

For statement is another looping mechanism. The for loop like the while loop allows you to iterate and perform a group of statements over and over again.

```
CANONICAL FORM

for ([where to begin] ; [where to stop] ; [step]) {

}
```

A key concept in using the for loop is the counter (the step). Loop will perform all the statements inside the block for a finite and determined number of times. Each time that the whole block is evaluated, the step value is incremented. The incremented step value is then compared to the ending value, and when the ending value is reached, the for loop will then terminate. The program control will fall over to first statement immediately after the for loop block.

```
for (i = 1 ; i < 10 ; i++) {

    console.log(i);

}
```

The code sample above defines the starting point at 1 (i = 1), it will stop the loop execution when *i < 10*. Each time the *for* loop is evaluated, the i variable will be incremented by 1 (i++).

Here is a another sample of the for loop.

```
var util = require('util')

var i = 1;
var j = 1;

for (i = 1 ; i < 10; i++) {

    for (j =1 ; j < 10; j++) {

        util.print(i * j + "\t");

    }

    util.print("\n");
}


/*

RESULT

1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
4   8   12  16  20  24  28  32  36
5   10  15  20  25  30  35  40  45
6   12  18  24  30  36  42  48  54
7   14  21  28  35  42  49  56  63
8   16  24  32  40  48  56  64  72
9   18  27  36  45  54  63  72  81

*/
```

This code requires the node.js library *util*. We can't use the STDIO (console.log) here because it automatically adds a CRLF (carriage return-line feed) while *util.print()* library does not. Node.JS has many libraries for convenient use, you just need to do the *require(libname)* statement if you want to use it.

4.8 Complex data types

JS has some containers (stuff that can contain other stuff, that can contain other stuff …). We will take a look at two of them right now—Arrays and Objects. These are not simple types like Numbers or Strings or Boolean. Instead of representing single values, Objects and Arrays can represent a collection of values.

4.8.1 Object

Lots of things are considered objects in JavaScript, even arrays and functions are objects, but let's not get distracted by that right now. Think of an object like a dictionary or the telephone yellow pages. It is just a collection of stuff. This stuff though has a format, it comes in pairs. The first part of the pair is a key, the other part of the pair is a value. Using the yellow pages analogy, the key is probably a person's name or name of a business. The value is the address or the phone number. The key has to be unique, you cannot have more than one key with exactly the same name.

The value part of each pair could be anything. It could be simple data (Numbers, Booleans, Strings) or it could be complex data (Arrays, Objects or functions).

There are a couple of ways to create objects in JS.

1. Using object literals
2. Using the new keyword against the Object type
3. Constructor functions

Try your best to ignore number 3 above (at least for now) – we will circle around that later when we get to JavaScript OOP.

4.8.2 Using Object literals

```
var obj = {} // this one constructs an empty object
var pt  = {x:0, y:0} // an object with simple data types as members
var person = {

    name: "John Doe",
    email: "johndoe@somewhere.com",

} // an object that has two members, both of which are Strings
```

When you use object literals to initialize objects, don't forget to separate the pairs using comma. You can add properties (the key-value pair) to objects by simply declaring the member–using the dot notation–and assigning them values

```
person.date_hired = "some date value";
person.address = "some address";
```

Similarly, you can access values of the object properties using the dot notation as well, although this is not the only way.

```
console.log(person.name);
console.log(person.address);
```

JavaScript's use of dot notation to access object properties and method will feel very familiar with devs because a lot of programming languages utilizes this notation too. What might not feel familiar will be the fact that JS can also access object properties using square brackets. Like this

```
person["name"];
person["address"];
```

JS objects are actually associative arrays—I need to clarify one point, when I write **Array** (capitalized) I am referring to the JavaScript complex type. When I write **array** (lowercase) I am referring to the array data structure in general, what I mean by this is a collection of rows (and sometimes columns). An object looks like an array because it a collection of **named** values, this is the reason you can use the square braces approach to reference a member of a specific object.

4.8.3 Arrays

An Array is also a collection of values. Unlike an object, an Array is a collection of *ordered values*, not *named values*. Each member of an Array is called an element. Each element is denoted by a numeric position in the array–the position is called an index. Like objects, Arrays can be created using a variety of ways;

```
var a = []; //an empty array
var b = [1,2,3,4,5,6,7]; //an array with 7 elements
var c = ["Hello", "World", 1, 2.0, 0xFF]; //an array with members of diff types
var d = [[1,1], [2,2],[3,3]]; // an array with members, which are also arrays.
```

The other way to create an Array is to use the **Array()** constructor

```
var a = new Array(); //equivalent to var a = []
var b = new Array(1,2,3,4,5,6,7);
var c = new Array("Hello", "World", 1, 2.0, 0xFF);
var d = new Array(new Array(1,1), new Array(2,2), new Array(3,3));

var e = new Array(5); //with 5 new elements
```

If you call the Array constructor with a single integer as parameter, it means you are specifying the length of the Array–you are not declaring an array with a single element, and the number you passed is the value of the first element.

Array members can be retrieved by accessing the individual elements. Arrays are zero-based, hence the first element is index 0.

```
b[0] =  8; //assigns the value 8 to the first element of the Array
console.log(b[2]); //prints the value of the third element (remember, zero based)
```

The characteristic of arrays having index positions makes them ideal for iterating using a for-loop. You cannot access members of an array using the dot notation–like you did when you accessed members of an object.

4.9 Anonymous Functions

We have already taken a look at functions earlier. At first glance, functions in Javascript does not seem too special. Like in any other language, we used it to contain a bunch of statements and gave it a name. This name, when we call them at a later time, will execute

the set of commands inside its block. We use the function simply as an encapsulation unit for a sequence of statements.

Functions in JavaScript has some twists that may seem weird at first (depending on where you are coming from). The dynamic characteristic of the language presents a lot of possibilites–and some suprises too. Take a look at the following code

```
function foo() {
    return 10;
}

console.log(foo());     //nothing surprising here,
                        //it will return 10

foo = "Hello World";    //This is allowed in JS,
                        //because it is not strongly typed
console.log(foo);       //This will now output "Hello World".
                        //It has already forgotten that
                        //it was a function
                        //before
console.log(foo());     //this one will throw an error,
                        //because foo is no longer a function
```

JavaScript function names are not that different from ordinary variables. You can assign them values, just like how you would an ordinary variable. Similarly, you can assign functions to variables; like this

```
var foo = function() {
    return 10;
}

console.log(foo()); //returns 10
```

This is another way to create a function. Notice that the function on the right hand side of the equal sign does not have a name. Notice also that we can assign it to a regular variable named "foo" then invoked **foo()**. What we have just done was create an anonymous function and assigned it to variable *foo*. It is called an anonymous function, for obvious reasons, it does not have a name. There are other ways to create functions in JS, but we will stop here because we don't have any further use for another way to create a function, what we have is quite enough for now.

4.10 Objects, again

A few things we need to remember (or know for the first time) about objects.

Objects in JavaScript are reference types

Object in JavaScript are not value types, you can declare as many reference variables pointing to the exact same object

```
//filename: multiref.js

var obj1 = {
    name: "firstobject"
```

```
}

var obj2 = obj1; // both obj1 and obj2
                    //reference var is pointing to the same object

console.log(obj2.name); //exact same property as obj1

obj2.anothername = "still first object";
console.log(obj1.anothername); //obj1 should have "anothername" as well,
                    //because object 2 is pointing to the same object
                    //being referenced by obj1
```

Some object's structure can be modified (mutable). An array is actually an object, it is a good example of an object that is mutable. Mutability means that as you modify the content or structure of the object, it does not spawn or create newer objects. It modifies the original object that was created. Mutable types

And then, some objects cannot be modified (immutable). The String objects that you create are such examples of immutable objects in JS. To add this list, Numbers and Booleans (when wrapped) are also immutable. What this means is that if you apply transformations to immutable objects, they will not modify the original object that was created. The transformation will cause the object to create a new object. immutable types

```
var assert = require('assert');

var s = "A String";
var stemp = s;  //s and stemp points to "A String"

assert(stemp == s, "checking is s is equal to stemp"); // result is ok

s = s + "Hello World";

assert(stemp == s, "checking is s is equal to stemp"); // result not ok
```

If Strings are mutable, then the second assert would not have failed. The references "s" and "stemp" are simply no longer pointing to the same object. Any transformation that you apply on an immutable object will cause the creation of a new object, the original content plus any transformation you applied will now be contained inside the new object. By the way, **assert** works pretty much like a test for truth or falsity, but it behaves violently. When the expression inside the assert (the asserted value) does not evaluate to true, your code will fail with a pretty ugly run-time error. Defensive programmers use this to test their codes mercilessly, better that the code fails during development time, rather than the code failing in front of hte users.

4.10.1 Pass by Reference

```
function Base() {

    this.prop1 = "Property 1"

}

var a = new Base();     // Property 1
```

```
console.log(a.prop1);

paramPass(a);
console.log(a.prop1);   // Altered prop1


function paramPass(arg) {
    arg.prop1 = "Altered prop1";
    console.log("While inside ParamPass : Base.prop1 = " + arg.prop1 );
    // this outputs Altered prop1
    return 0;
}
```

When you pass object references to functions as parameters, whatever changes you make to the object (that was passed) will endure even after the function goes out of execution scope.

4.11 Objects with Functions

Objects are easy to create, construct and modify in JS. Adding a property to an object takes nothing more than declarating the name of the property (attached to the object using the dot notation), then simply assigning a value to to it. Just don't forget that objects in JS are like hash tables, each member of the object is a pair of data, the first one is the key and the second one is the value.

You can put any type of member inside an object. A member could be any data type in JS, simple data or complex data—which means you can put another object inside another object. This is the part we need to make a leap on our JS knowledge. **Functions** in JS are objects as well, that's right. Which means, we can put functions inside objects, and when functions are inside objects, they become methods of the objects.

```
// file: obj3.js

var employee =  {

    name: "Ted",
    work: function() {
        console.log(this.name + " is working")
    }
}

employee.work();
```

The preceding code sample is straightforward, it is a simple creation of an object using object literals. The second property though, used an anonymous function which was assigned to the property named "work". When we invoked the *work()* property as function—as we should, because it is a function, it then executed whatever was defined inside the function block.

If we need another object, we need to repeat whatever we did on the first employee object, like so;

```
var employee =  {

    name: "Ted",
    work: function() {
```

```
        console.log(this.name + " is working")
    }
}

employee.work();

var rommel = {};
rommel.name = "Rommel";
rommel.work = function() {
    console.log(this.name + " is working");
}

rommel.work();
```

While this is perfectly legal to do, the idiom used in creating the *rommel* object is bit bitter to the taste, surely, there has to be a way to achieve some sort *Object factory* idiom in JS. The code is too malleable (it could be a good thing though). It seems that following this idiom, one can pretty much add properties as you go along. I am sure there is a class of problems where this kind of flexibility and malleability is desired, probably necessary even. Coming from a language that supported classical OOP (like Java), it can send some chill on the spine knowing that I could easily change properties of my object anywhere in the code. Fortunately, JS is very orthogonal language—there are lots of ways of achieving same results, so let's look for something more palatable for classical OO programmers.

```
// Employee object: Template

function Employee(name,id, dept) {
    this.name = name;
    this.id = id;
    this.dept = dept;

    this.work = function () {
        console.log(this.name + " is working")
    }

}

// Company object: Template

function Company() {
    this.employees = [];

    this.recruit = function(employee) {
        this.employees.push(employee);
    }

    this.listEmployees = function() {
        arrlength = this.employees.length;
        for (i = 0; i < arrlength; i++) {
            console.log(this.employees[i]);
        }
    }
}


var ted = new Employee("Ted H", 39, "swd" );
var kit = new Employee("Kit A", 31, "swd");
var pepper = new Employee("Pepper F", 2, "admin");
```

```
var rd = new Company();


rd.recruit(ted);
rd.recruit(kit);
rd.recruit(pepper);

rd.listEmployees();
```

The preceding code feels a bit firmer than the one before it. This code used the function as some sort of constructor (actually, it is called exactly that, a function constructor). JavaScript does not have class mechanisms like Java or C++. Code reuse is not achieved by class extensions, but rather by Object extensions. By the way, this last code we just did sure looks a lot like the class mechanism, if you could just get over the fact that it said **function** rather than *class*.

There a couple of things we need to talk about in this code, specifically they keyword **this**, but before we go there, let's look at another code, it is exactly the *Employee* and *Company* constructor code also, but with a bit of modification.

```
// Company.js

// Employee object: Template

function Employee(name,id, dept) {
    this.name = name;
    this.id = id;
    this.dept = dept;
}

Employee.prototype.work = function () {
    console.log(this.name + " is working")
}


// Company object: Template

function Company() {
    this.employees = [];
}

Company.prototype.recruit = function(employee) {
    this.employees.push(employee);
}

Company.prototype.listEmployees = function() {
    arrlength = this.employees.length;
    for (i = 0; i < arrlength; i++) {
        console.log(this.employees[i]);
    }
}

var ted = new Employee("Ted H", 39, "swd" );
var kit = new Employee("Kit A", 31, "swd");
var pepper = new Employee("Pepper F", 2, "admin");

var rd = new Company();
```

```
rd.recruit(ted);
rd.recruit(kit);
rd.recruit(pepper);

rd.listEmployees();
```

4.12 Inheritance

You can use JavaScript pretty much in a procedural way–what I mean by that is, without using coarse grained abstractions for interesting objects of your problem domain–but as soon as you find yourself coding way past 300 lines of code (give or take a 100 lines), you might find yourself in a (pretty) mess. The code becomes incresingly complex and increasingly difficult to change or introduce new features. Soon you need to find a way to compartmentalize the mess, you will still deal with the mess, but with OO abstractions, the messes are in bigger boxes. Inheritance is one of the bigger concepts in OO programming, here's one way to do them in JS.

```
var Person = function() {
    this.work = function() {
        console.log("Person working");
    }
    this.managing = function() {
        console.log("should not be implemented here");
    }
}

var Manager = function() {
    this.managing = function() {
        console.log("am managing");
    }
}

Manager.prototype = new Person;

var m = new Manager();
m.work();       // Person working
m.managing();   // am managing

console.log("Keys : " + Object.keys(m));
console.log("isPrototypeOf manager : " + Person.prototype.isPrototypeOf(m));
console.log("getPrototypeOf m : " + Object.getPrototypeOf(m));


console.log("m instanceof Manager  :" + (m instanceof Manager)); //true
console.log("m instanceof Person   :" + (m instanceof Person)); //true
console.log("m instance of Object  :" + (m instanceof Object)); //true
console.log("m instanceof Function :" + (m instanceof Function)); //false
```

JavaScript is an OO language but it is not class-based. There are no templates (classes) you can use to create objects. Objects, in JavaScript are created using other objects; more specifically, they are created using the function object. There is a way to create an object in JS using a literal, but we are not interested in those, we are interested in objects created using constructor functions.

When you create a constructor function (like Manager and Person, in the examples above), they will have a property called *prototype*. A prototype is an object whose prop-

erties are shared by all the objects that have that prototype . In the example above, Manager inherits from Person via the code **Manager.prototype = new Person**. Whatever Person have, Manager will also have.

To override an inherited method, simply redefine (reimplement) the method in inheriting class (the child class), that makes the method polymorphic. When you invoke a method against an object, the method will be searched (first) on the child class, if the method is not found, the search will go up the prototype chain until the method definition is found.

5 | Client-side JavaScript

JavaScript is quite ubiquotous nowadays. You can find it in quite a few places, on mobile apps, server side programs, glue codes for Java programs etc. The most common place you will find JavaScript code is within web pages (HTML pages).

```
<!DOCTYPE html>

<html>
<body>

<script>document.write("Hello JavaScript");</script>

</body>
</html>
```

JavaScript lives inside the *script* tag of an HTML page, in this example, the JavaScript code is inline, meaning it is written on the same source file where the HTML codes are also written in.

```
<!DOCTYPE html>

<html>
<body>

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

</body>
</html>
```

In this example, the script element has an src attribute which is given a value "hellojavascript.js"—this is a JavaScript source file. You can write whatever legal JavaScript statements in that source file.

The code document.write() , when executed, will erase the whole content of the HTML page (specifically the whole of the body element) and then replace it with whatever you supplied as parameter to the write() method—that's probably not what you would like to do, so be careful using that command, you normally see that code on introductory codes of JavaScript on the browser so that you can quickly get a tactile feel of JS coding. In order to produce dynamic HTML with JavaScript you will need to know a little bit about the DOM (Document Object Model).

An HTML page is made up of tags (elements) which in turn can contain other elements. If element1 contains element2 and element3, the relationship between these elements is that; element2 and element3 are child nodes of element1, another way of saying it

is that, element1 is the parent node of element2 and element3. If you try to picture the relationship of HTML elements to one another, it might resemble something like a tree structure. The trunk, the main node is the <html> element, and then it will have branches (the <head> and <body>) elements. The head and body elements can in turn contain other elements like script, style, input, textarea, options, canvas etc.

The document object model allows you programmatic access to these elements and their attributes. Navigating the DOM is an essential skill of a JavaScript programmer. Below is a rough pictorial representation of DOM (Level 0)

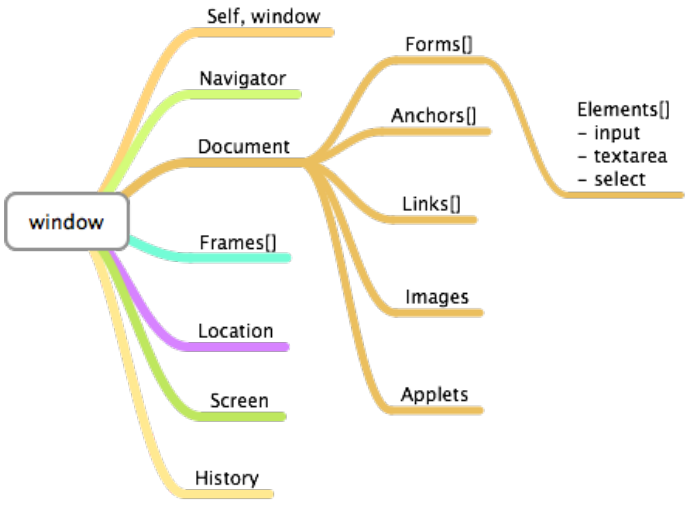

Figure 5.1: DOM Level 0

Each node in the DOM is an object, so you can use the dot notation to reference each element. The root element of the DOM is the window object, this is the most global space in client-side JavaScript. It has many child nodes, you have already seen one of them in action a while back, the document object. When we wrote document.write(), we actually meant window.document.write(), but we don't have the write the window object explicitly.

I already mentioned that each node in the DOM is an object, which means if you want to do something to one of elements, you need to know their methods so you can do something meaningful to them.

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <div id="watch">

        </div>
```

```
        <script>

        var out = document.getElementById("watch");
        out.innerHTML = "Hello JavaScript";

        </script>

    </body>
</html>
```

The document object has a method—which you will use quite a lot—the getElementById(). It is used to locate an HTML element that has a specific ID. When the element is found, a programmatic reference to that object is returned. After that, you can call the methods of the HTML element. In the example above, a div named "watch" was defined in the HTML. You can use document.getElementById("watch") to obtain an object reference to that div just like what we did on the sample code. The innerHTML is a property of any div element, this is a read-write property so you can pretty much use it like this

```
var x = document.getElementById("watch");
x.innerHTML = "Writing to the div";  // writes to the div

var y = x.innerHTML; // reading the runtime content of the div
```

Let's re-arrange the codes in our sample

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
    </head>
    <body>
        <script>

        var out = document.getElementById("watch");
        out.innerHTML = "Hello JavaScript";

        </script>

        <div id="watch">

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

In this version, the script comes before the div definition, if you run this code, you will see nothing—because there will be an error. The variable out will contain null. The reason for this is because we tried to obtain an object reference to a div element which does not exist yet. The div "watch" must be defined and loaded in the DOM before we can obtain a programming reference to it. It is time to step back a little bit and try to get some more understanding of how the browser, DOM, and JavaScript works.

When a web page is received by browser, it will go to work by scanning through all the HTML elements in the page. It will be interrupted shortly when it encounters elements that have an src attribute, because that attribute means that whatever information follows the src, it is not part of the current document being rendered—the browser will make another roundtrip requesting the resource from the server, then coming back

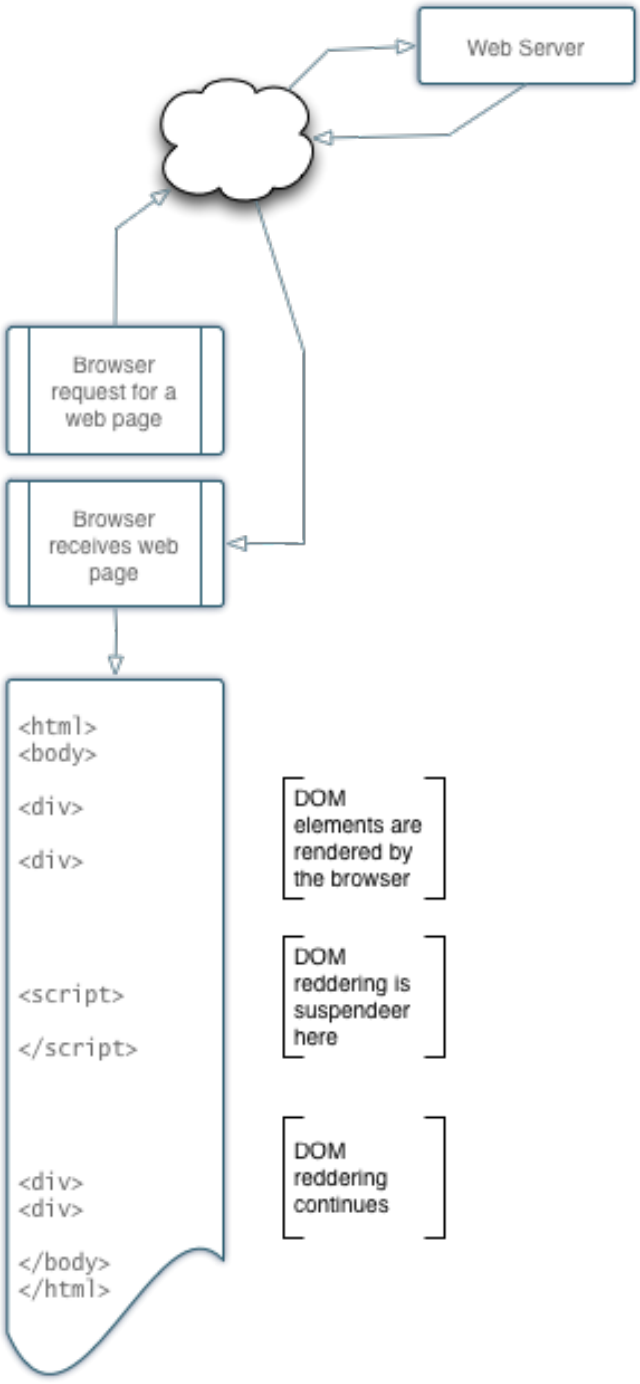

Figure 5.2: How DOM is processed

again. This is the reason why sometimes, the browser seems to load for a very long time, because it is trying to complete all the elements in the web page. If the element has an src attribute, it will be fetched before the browser can complete rendering the complete DOM.

The other occassion when the browser will stop rendering the DOM is when it encounters the `<script>` element. JavaScript has the ability to alter the DOM, that is why the browser needs to suspend operation when a JavaScript code is running. When the script is done, the DOM rendering can resume.

You need to keep this interplay between the DOM, JavaScript and the browser in mind so that you always know the state of the browser and the DOM before you accessing any element of the DOM.

This does not mean though that you have to position your scripts on very select places of the HTML page just to ensure that your code is synchronized with the HTML elements. Not only is that an ardous task, it is also bad form. What we need right now is to use a little bit of callbacks.

Callbacks are simply functions in JavaScripts, you probably know them better as events. Each element of the DOM have built-in events that you can already use. Let's start with the **window** object, we'll rewrite the previous code sample using some events of the window object.

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
        <script>

        window.onload = function(){
            var x = document.getElementById("watch");
            x.innerHTML = "Writing to the DIV watch";
        }

        </script>
    </head>
    <body>

        <div id="watch">

        </div>

    </body>
</html>
```

This is a lot different now, the script element is on the head portion of the HTML page, obviously none of the HTML elements have been rendered at this point, but code works just fine. This is because we have overriden the onload event of the window* object. By assigning a callback function to the onload member of window, we are in fact telling it which function to execute when all the elements of DOM have already been loaded, because at that point, the DOM is pretty stable, all of the elements would have been rendered.

5.1 Event Programming

The basic idea is when a user does something, e.g. click, right-click, types something, hovers the mouse over an area, your code will do something meaningful, useful or cool (or all of the above). These useful codes reside on your JavaScript functions. Hence, the trick is to bind a function (a function that you define) to an established event of an HTML element.

Events can be bound to JavaScript functions in a couple of ways.

5.1.1 Binding the event using the attribute of an HTML element

```
<!DOCTYPE html>

<html>
    <head>
        <title></title>
        <script>

        function btnClick() {
            var x = document.getElementById("writetome");
            x.innerHTML = "Hi World";
        }

        </script>

    </head>
    <body>
        <form>
        <input type="button" value="Click me to see" onclick="btnClick()">
        </form>

        <div id="writetome"></div>
    </body>
</html>
```

The onClick attribute of the input element is assigned a name of the JavaScript function. When the user clicks the button, btnClick() will be called. This is fine if you are experimenting on JavaScript, but this style of coding is obstrusive, it sullies the HTML code.

5.1.2 Using callbacks programmatically

```
<!DOCTYPE html>

<html>
<head>
    <title></title>
    <script>

    window.onload = function(){
        var btn = document.getElementById("btn");
        btn.onclick = function(){
            document.getElementById("writetome").innerHTML = "Hi World";
        }
    }
```

```
    </script>
</head>
<body>
    <form>
    <input type="button" value="Click me to see" id="btn"/>
    </form>

    <div id="writetome"></div>
</body>
</html>
```

The second version of our code is a bit cleaner. There are no programming instructions on the input element, just styling and bare instructions. The onclick event was bound to the button entirely inside the **script** element. All of our logic are wrapped inside the window.onload callback because we need the DOM to have completely rendered before we get a reference to both the div and the button input element.

Most input elements of the DOM have the onclick event, you can even use it on document object itself, if you want to.

Below is a list of the common events you might need for JavaScript programming. The list is not exhaustive, nor is it complete. It is listed only to get you started.

1. **onclick** - works on lots of HTML elements, there is no ondoubleclick or onrightclick, that is handled using some other trickeries of the event object. I will get to that, but not now
2. **onmousedown, onmousemove, onmouseover, onmouseup** - the events are pretty self explanatory, you can probably guess already when these events are triggered, no need to spell it out. This event works on lots of elements too, like the onClick
3. **onchange** – usually found on input, select and textarea elements. This event is triggered when the value of the element has changed and it has lost the focus
4. **onkeyup, onkeydown, onkeypress** - if you want to handle keyboard events

5.1.3 Sample code, onkeyup

```
<!DOCTYPE html>

<html>
<head>
    <title></title>
    <script>

    window.onload = function(){

        var txtobj = document.forms[0].elements["txtarea1"];

        txtobj.onkeyup = function(){
            document.getElementById("writetome").innerHTML = txtobj.value;
        }
    }
    </script>
</head>
```

```
<body>

<form name="objform">
    <textarea name="txtarea1" rows="10" cols="100">

    </textarea>

    <p>
</form>
<div id="writetome">
    This text will change
</div>
</body>
</html>
```

Everytime the text inside the textarea element changes, we get the value of textarea (txtobj.value) then we assign that to the innerHTML property of the div

5.1.4 Handling the right click

```
<!DOCTYPE html>

<html>
<head>
    <title></title>

    <style>
        #thearea {
            border:4px solid gray;
            width: 500px;
            height: 200px;
        }
        #watch {
            border:5px solid green;
            padding-top: 50px;
            width:500px;
        }
    </style>

    <script>

    window.onload = function(){

        var objarea = document.getElementById("thearea");

        objarea.oncontextmenu = function(){
            return false;
        }

        objarea.onmousedown = function(){

            var watch = document.getElementById("watch");
            var x = event.clientX;
            var y = event.clientY;
            var msg = "X : " + x + " | Y: " + y;

            switch(event.which){
                case 1: // left click
                    watch.innerHTML = "Left " + msg;
                    break;
```

```
                case 2: // middle click
                    watch.innerHTML = "Midle " + msg;
                    break;
                case 3: // right click
                    watch.innerHTML = "Right " + msg;
                    break;
            }
        }

    }

    </script>
</head>
<body>

<div id="thearea">Right click inside the box. Then right click outside
the box. The context menu will not be triggered inside this div
</div>
<p>
<div id="watch"></div>
</body>
</html>
```

We need to override the **oncontextmenu** function so that the default context menu will not kick in. The key to detecting which mouse button was used to click is the **which** property of the event object—note though that there have been inconsistencies on the values of which, sometime in the past, I remember 0 = left click, 1 = middle and 2 = right; Do your proper testing, you have been warned.

5.1.5 Tracking the mouse position

The X and Y position of the mouse is always reported by the event object, in the following code sample, the mouse position is captured during the **onmousemove** of a div element.

```
<!DOCTYPE html>

<html>
<head>
    <title></title>

    <style>
        #thearea {
            border:4px solid gray;
            width: 500px;
            height: 200px;
        }
        #watch {
            border:5px solid green;
            padding-top: 50px;
            width:500px;
        }
    </style>

    <script>

    window.onload = function(){

        var objarea = document.getElementById("thearea");
```

```
        objarea.onmousemove = function(){

            var xpos = event.clientX;
            var ypos = event.clientY;
            var msg = "X : " + xpos + " | Y : " + ypos;

            document.getElementById("watch").innerHTML = msg;
        }
    }

    </script>
</head>
<body>

<div id="thearea">Just hover your mouse inside box</div>
<p>
<div id="watch"></div>
</body>
</html>
```

The event object has two members, clientX and clientY which corresponds to exact position of mousepointer at the time of capture. You need to be careful when using event.clientX and event.clientY*, these may not be the same from one browser to another– which is why its worthwhile using JavaScript libraries that takes away our worries of cross-browser coding, like JQuery; which is coming up next.

6 | HTML5 Canvas

HTML5 Canvas is one of new additions to the HTML specification. By now, it must have occured to you already that HTML5 does not refer to a single thing, it refers to a lot of things. There are markups or tags and a bunch of APIs—which are all exposed via the use of JavaScript. HTML5 Canvas is one of those of lots of things under the HTML5 umbrella.

The canvas is drawable region within the HTML document. You can draw pixels anywhere you like in this region, you can color it up too. It is simple to use the canvas, the API contains methods which allow you to produce some drawing primitives like arc, text and rectangle. These drawing APIs are all accessible via JavaScript.

7 | JQuery

8 | TypeScript

9 | CoffeeScript

10 | HTML5 Audio

Before HTML5, audio files were played in the web page using browser plugins, there were lots of them. Some plugins would support some browsers, and some other plugins would support some other browsers.

In HTML5, the element is intended to play audio files on the web page without the aid of plugins.

The basic declarations for HTML5 audio is illustrated in the code below

Listing 10.1: HTML5 audio element

```
<!DOCTYPE html>

<html>
  <head>
    <title></title>
  </head>
  <body>
    <audio controls>
      <source src='aria.mp3' type='audio/mpeg'>
        Your browser does not support the audio element <br/>
        of HTML5
    </audio>
  </body>
</html>
```

The structure for element is straightforward, inside the audio tag, is tag, the attributes of the source tag is where you specify the URL and the type for the audio file.

HTML5 is still nascent (even at the time of this writing), there are three audio formats currently supported (not by all browsers though). The table below summarizes support of each browser on the three audio formats, OGG Vorbis, WAV and MPEG

BROWSER	MP3	WAV	OGG
IE9+	yes	no	yes
Chrome	yes	yes	yes
FireFox	no	yes	yes
Safari	yes	yes	no
Opera	no	yes	yes

Figure 10.1: Browser support

Each , ideally, should contain only one element, although nothing prevents you from embedding more than one *source* element inside an *audio* element, only the first source element defined will be visible (and playable) in the web page.

`One source file per <audio> element`

If you intend to place more than one audio file in a web page, you should consider defining more than one *audio* element, once for each *source*

Listing 10.2: HTML5 audio element

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>
  <body>
    <audio controls>
      <source src='aria.mp3'>
      <source src='jewish-life.m4a'>

        Your browser does not support the audio element <br/>
        of HTML5

    </audio>

    <audio controls>
      <source src='jewish-life.m4a'>
        Your browser does not support the audio element <br/>
        of HTML5
    </audio>
  </body>
</html>
```

By far, you have seen that by including the **controls** argument in the *audio* element, you can display the default audio controls to allow the user to manage the playback of the audio files. If you change the **controls** to **autoplay**, the audio file will begin playback as soon as the webpage is loaded. Be CAUTIOUS about using this, this can potentially interfere with the experience of the users, especially those who rely on audible feedback for page navigation; not to mention, autoplaying audio files is downright annoying.

`autoplay`

There are other attributes that you can use with the *audio* element, below is the list of these attributes, two of which you have already seen from earlier examples.

1. **autoplay** - Sets the media clip to play upon creation or query whether it is set to autoplay.
2. **loop** - Returns true if the clip will restart upon ending or sets the clip to loop (or not loop).
3. **currentTime** - Returns the current time in seconds that has elapsed since the beginning of the playback. Sets currentTime to seek to a specific position in the clip playback.
4. **controls** - Shows or hides the user controls, or queries whether they are currently visible.
5. **volume** - Sets the audio volume to a relative value between 0.0 and 1.0, or queries the value of the same.
6. **muted** - Mutes or unmutes the audio, or determines the current mute state.
7. **autobuffer** - Tells the player whether or not to attempt to load the media file before playback is initiated. If the media is set for auto-playback, this attribute is ignored.

10.1 Controlling audio programmatically

The element is already usable out of box. With just a few simple declarations, you can an sound dimension to your web pages. Like many other things in HTML5, can be programmed, via *JavaScript* of course.

Listing 10.3: HTML5 programming audio

```
<!DOCTYPE html>

<html>
  <head>
    <title>HTML5 Audio Demo</title>

    <style>
    #button {
      width: 100px;
      height: 100px;
      border: 2px solid gray;
    }

    #watch {
      border: 1px dotted gray;
    }
    </style>

    <script>

    window.onload = function() {

      var button = document.getElementById("button");
      var clip = document.getElementById("clip");
      var watch = document.getElementById("watch");

      button.onclick = function() {
        var duration = clip.duration;
        var ended = clip.ended;
        var filename = clip.currentSrc;

        clip.play();

        watch.innerHTML = "Duration : " + duration + "<br/>" +
         "Ended : " + ended + "<br/>" + "Filename : " + filename;
      }
    }

    </script>

  </head>
  <body>

    <div id='button'>
      play
    </div>

    <audio id='clip'>
      <source src='aria.mp3'>
      <source src='jewish-life.m4a'>

        Your browser does not support the audio element <br/>
        of HTML5

    </audio>

    <div id='watch'>
      Hello
    </div>

  </body>
</html>
```

11 | HTML5 Video

12 | Introduction to GEO Location

13 | WebWorkers

14 | WebSockets

15 | Introduction to Server Sent Events

16 | HTML5 Local Storage

17 | Tools

Bibliography

[1] James L Williams, *Learning HTML 5*. Addison Wesley, 2011.

[2] David Flanagan, *JavaScript, The definitive guide*. O'Reilly, 2006.

[3] Larry Ullman *Modern Javascript, Development and Design* O'Reilly

[4] Rob Hawkes *Foundation HTML5 Canvas* O'Reilly 2010

[5] Stroyan Stephanov *Javascript Patterns* O'Reilly 2010

[6] Douglas Crockford *Javascript, The Good Parts* O'Reilly

[7] Steve Fulton & Jeff Fulton *HTML 5 Canvas* O'Reilly year

[8] Steve Mcaw *The Small CoffeeScript Book*

[9] Microsoft *TypeScript Language Specification v.08* Microsoft 2012.

[10] Matthew David *HTML5 Desining Rich Internet Applications* Focal Press 2010.

[11] *HTML 5 Rocks* http://html5rocks.com

[12] W3C *HTML 5 Specification* W3C http://www.w3.org/TR/html5/

[13] W3C Schools *HTML Basic* http://www.w3schools.com/html/default.asp

[14] ECMA International *ECMAScript 262 Specification* http://www.ecma-international.org/publications/standards/Ecma-262.htm