JavaScript

Table of Contents

1 Introduction 4

1.1 Who should read this text? 4

2 JavaScript basics 5

2.1 Getting started 5

2.2 External JavaScript files 5

2.3 Interacting with the document 5

3 ECMAScript 7

3.1 Early History 7

3.2 Standards 7

3.3 Reliable links 7

3.4 Practical start 7

3.5 Core JavaScript syntax 8

3.6 Debugging and the JavaScript Console 8

3.7 Documentation 9

3.8 Primitives 10

3.9 JavaScript outside a browser 11

3.10 Caution - Not A Number is a number 11

3.11 Caution – type coercion 12

3.12 Caution – truthy and falsy 13

3.13 Caution – function hoisting 13

3.14 Caution – variable hoisting 14

3.15 Namespaces 14

3.16 Anonymous functions 15

3.17 Objects as namespaces 15

4 Type and objects 17

4.1 Objects 17

4.2 The global object 17

4.3 Creating objects 18

4.4 Object properties 19

4.5 Internal properties 19

4.6 No classes in JavaScript 20

4.7 Primitives and object references and aliases 20

4.8 Prototypes and inheritance 20

4.9 Functions 21

4.10 Primitives and auto-boxing 21

4.11 Functions as objects 22

4.12 Closures 22

4.13 Dot and [ ] notation 22

4.14 A dynamic language 23

4.15 Keyword 'this' 23

5 The pre-built objects 25

5.1 Number 25

5.2 Boolean 25

5.3 String 25

5.4 Date 25

5.5 Math 25

5.6 Function 25

5.7 Array 26

5.8 Regular Expressions 26

6 OOP in JavaScript 28

6.1 Classes in JavaScript 28

6.2 Inheritance and Polymorphism 29

6.3 Encapsulation – private members 29

7 Functional programming 31

7.1 First-class and higher order functions 31

7.2 Pure functions 31

7.3 Recursion 31

7.4 Eager and lazy evaluation ... 32
7.5 Strict functional programming ... 32
7.6 Maximum element of a list ... 32
7.7 Total of a list ... 33
7.8 Reverse a string ... 33
7.9 Remove an element from a list ... 33
7.10 Sort a list ... 33

8 The DOM ... 34

8.1 The global object ... 34
8.2 The Browser Object Model ... 34
8.3 The DOM ... 34
8.4 DOM levels and IDL ... 35
8.5 Accessing nodes ... 35
8.6 Inserting and deleting nodes ... 36
8.7 Changing attributes ... 36
8.8 Events ... 36
8.9 JSON ... 37
8.10 AJAX ... 37

9 Data structures ... 40

9.1 A linked list ... 40
9.2 Stack ... 40

1 Introduction

1.1 Who should read this text?

JavaScript as a programming language is a product of the browser wars between Netscape and Microsoft in the 1990s. It is often used by graphics designers who have no background in Computer Science. Consequently it is often given a rather low status compared with languages like C.

However, this is not appropriate. JavaScript does have a formal specification (as ECMAScript 5.2). It is an interpreted language which can be used in several paradigms – structured programming, OOP and functional programming. It is dynamic – code can change itself at runtime.

For client-side programming, its great advantage is the fact that all browsers contain interpreters, so there are no problems about distributing code.

This text is not intended for people learning to program. It assumes that

1. You are familiar with the basic programming concepts of variable, type, control constructs and so on.

2. You know a 'proper' language like C or Java

3. You know html and css, and can write web pages using a text editor and understand how to load them into a browser to test them.

4. You have some idea of server-side scripting, say in php, and database use with SQL, say with mySQL.

For example, you should know what an associative array is.

There is *no reference material* here. That would be pointless. Links are given to complete references to the standards. Web links are much more appropriate for that than a text like this. This means the text is pretty short.

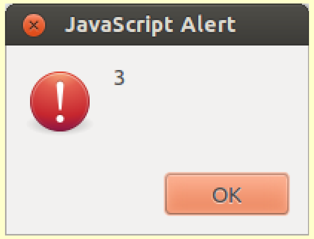
2 JavaScript basics

2.1 Getting started

This section is intended as an informal introduction to coding in JavaScript. More details are provided in the next section.

Try this:

```
<!DOCTYPE html>
<html>
<head>
<title>Title</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
<script>
   x = 1;
   y = 2;
   z = x + y;
   alert(z);
</script>
</head>
<body>

</body>
</html>
```



When this loads, it outputs as shown. We have two variables x and y. We add them, in z, and the alert function displays the result. Try experimenting with the script.

2.2 External JavaScript files

A script can be placed in the head of an html page (or anywhere else). But this has disadvantages. We might have a useful script which we want to use in every page, which would mean including it in every page. As for CSS, it is much more effective to have the script in a separate file, and link it to any web page in which we want to use it. For example

```
<!DOCTYPE html>
<html>
<head>
<title>Title</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
<script src="js1.js"></script>
</head>
<body>
</body>
</html>
```

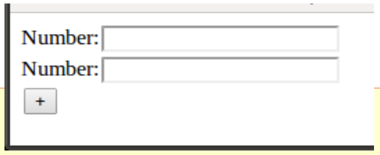
and the file js1.js is

```
var incomeTax=3+4;
alert(incomeTax);
```

2.3 Interacting with the document

JavaScript will usually interact with an html document.

Suppose we have an html body like this:



```
<body>
<form>
Number:<input type="text" id="num1"><br>
Number:<input type="text" id="num2"><br>
<input type="button" value="+" onclick="add()"><br>
<div id="result"></div><br>
</form>
</body>
```

The idea is to have a simple calculator which adds the two numbers and displays the result. The button has an onclick attribute onclick="add()". That means when it is clicked, a JavaScript add function is called.

The function is:

```
function add()
{
box1=document.getElementById("num1");
x=parseFloat(box1.value);
box2=document.getElementById("num2");
y=parseFloat(box2.value);
z=x+y;
result = document.getElementById("result");
result.innerHTML=z;
}
```

We will go through this line by line.

The first thing to do it to get a reference to the first input field:

```
box1=document.getElementById("num1");
```

document.getelementbyid searches the document for a matching id. So box1 is the first input box.

Next, we need the number in it. box1.value gets that. But it treats it as a string – a sequence of characters, rather than a number. ParseFloat changes a string to a number. So

```
x=parseFloat(box1.value);
```

means x is the number in the first box.

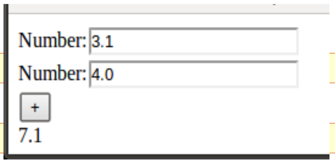
The next two lines are the same for the next box. So x and y are the two numbers, and z is their total.

```
result = document.getElementById("result");
```

gets the div with id 'result', then

```
result.innerHTML=z;
```

puts the number in it:


Number: 3.1
Number: 4.0

7.1

Try adding subtract multiply and divide buttons.

3 ECMAScript

The evolution of languages: FORTRAN is a non-typed language. C is a weakly typed language. Ada is a strongly typed language. C++ is a strongly hyped language.
--Ron Sercely

3.1 Early History

In 1995 Netscape developed a scripting language, named LiveScript, to use in their browser, Navigator. Its release coincided with the first release of Java by Sun, and marketing people at Netscape thought this sounded good, so they switched the name to JavaScript at the last moment. JavaScript has no other connection with Java other that this marketing ploy.

This was the time of the 'browser wars' between Netscape and Microsoft. Netscape now had a browser which could do things Internet Explorer could not do, so Microsoft quickly responded with their own version, named JScript, in 1996.

3.2 Standards

Just because the standard provides a cliff in front of you, you are not necessarily required to jump off it.
--Norman Diamond

The Netscape/Microsoft battle was interesting for standards. Both wanted a better browser, and hence a different browser. But if scripts were handled differently, they would only work on one browser, so there was pressure for a standard. In 1997 ECMA (European Computer Manufacturer's Association) produced a standard, known as ECMAScript.

The current standard is ECMAScript 5.1, released in 2011, and this text will use that standard. All current browsers in widespread use support it, and for most, have done so for several versions back.

So this text should really be titled ECMAScript not JavaScript.

3.3 Reliable links

JavaScript is heavily used by web page designers, who often have backgrounds in graphic design and not in computer science. Consequently many blogs and 'tuts' are written by people who have a very shallow understanding of the language. This means Googling often leads to articles which only relate to simple aspects of the language and are often incorrect.

Here are some links to material which is authoritative and reliable:

The ECMAScript standard : http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

The Mozilla Developer Network JavaScript pages at https://developer.mozilla.org/en-US/docs/Web/JavaScript/ This has a reference section and a language guide.

Dmitry Soshnikov's pages, like http://dmitrysoshnikov.com/ecmascript/javascript-the-core/

Douglas Crockford's site http://javascript.crockford.com/

3.4 Practical start

This is so you can get on and write some scripts and try things out. JavaScript is a general purpose interpreted language, but overwhelmingly it is used in a web page in a browser. We can put a script in a page in three ways.

The first way is to put the code in an event handler. Some html elements have attributes which are event handlers – which specify what should happen if the user does something. The most obvious is when a button in clicked. So we can say:

```
<!DOCTYPE html>
<html>
<head>
<title>Title</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
<link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
<input type="button" value="click here" onclick=".. JavaScript here..">
</body>
</html>
```

Some code to try out would be alert('You clicked')

This has the disadvantage that it mixes up html and script. It also means if you need to do the same thing at different places, you have to copy the code. A better way is to put the code in a script element, usually in the head element:

```
<!DOCTYPE html>
<html>
<head>
<title>Title</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
<link rel="stylesheet" type="text/css" href="styles.css">
<script>
.. JavaScript here ..
</script>
</head>
<body>
</body>
</html>
```

Before version 5, html required you to say what language the script was in. It now defaults to JavaScript. The script is executed as the page is loaded – so that script would be executed before <body> is loaded.

But suppose we have several pages which all require the same scripting? This corresponds to the same issue with styles, and the fix is the same – put it in a separate file, which can be linked to by multiple html pages

```
 <!DOCTYPE html>
<html>
<head>
<title>Title</title>
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
<link rel="stylesheet" type="text/css" href="styles.css">
<script src="js1.js" >
</script>
</head>
<body>
</body>
</html>
```

Then the script is simply in the file 'js1.js'

3.5 Core JavaScript syntax

JavaScript syntax is based on C, with a few variations.

Code is structured as global scope, or function scope.

JavaScript can run in normal mode or in strict mode – set by saying 'use strict'; at the start of a script, or function.

In normal mode variables do not have to be declared before use. If you refer to a new variable, a variable of that name is simply created. This is a problem. If you have a variable named incomeTax, and you say

incometax=4;

then now you have two variables, and no syntax error. Using strict mode will treat this as a reference error, which you will see in the console.

Strict mode also makes some other changes to JavaScript semantics – test your code.

Another issue is 'use strict' must be the first statement. Suppose you say in a web page

<script src="js1.js"></script>

<script src="js2.js"></script>

Then js1 and js2 are concatenated. Suppose js1 start 'use strict' – then everything is in strict mode. But suppose js2 starts use strict, and js1 does not. Then *nothing* is in strict mode.

Statements must end with a semi-colon. However if you miss out one, the interpreter will in effect insert one, if it is appropriate, according to an obscure set of rules. Good practice is to always put a semi-colon at the end.

= is the assignment operator

== compares values

=== compares values *and types*

The interpreter will often do an implicit type conversion with ==, in JavaScript called 'type coercion'. Examples follow. It is safer to use ===.

There are some other pitfalls, explained below.

The type system is initially confusing. There are a set of primitive types (such as string and number), and objects. But there are no classes. There are some pre-built objects, and some of these are prototypes for object-based versions of the primitives – like String and Number. A section below covers types and objects.

3.6 Debugging and the JavaScript Console

As soon as we started programming, we found to our *surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*

Maurice Wilkes 1949

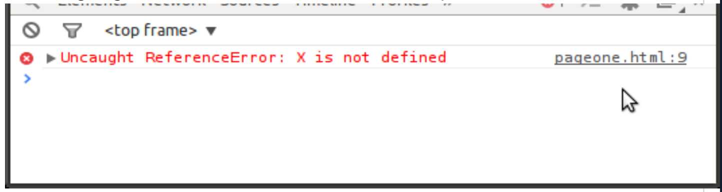
Try this:

```
<script>
x=3+4;
alert(X);
</script>
```

You expect 7. You get nothing. Not even an alert box. Where's the bug?

The problem is we've defined x, and we are outputting X, and JavaScript is case-sensitive – x and X are different.

Is there any way we can get help in finding bugs like this? The answer is the JavaScript console, which most browsers will display somehow. In Chrome it is in Tools..JavaScript console.



```
<top frame> ▼
Uncaught ReferenceError: X is not defined                pageone.html:9
>
```

You still need to do some work. You must read the error message very carefully 'Uncaught ReferenceError. X is not defined' and try to work out what it means. The console tells you where the error occurred – in pageone.html at line 9. After some thought you should see you have typed X when it should have been x.

Use the JavaScript console.

You can use alert(something) to display output, but this is modal and you have to click OK – no good for outputting 100 values. Two alternatives are

document.write(something);

which writes into the web page, or

console.log(something)

which outputs to the JavaScript console. The console has advantages that it displays uncaught errors and where they are, as shown.

3.7 Documentation

Java has javadoc, the tool which generates doucmentation as html pages from comments in the correct format.

There is no standard equivalent for JavaScript, but jsdoc is often used.

Download and install – how depends on the operating system.

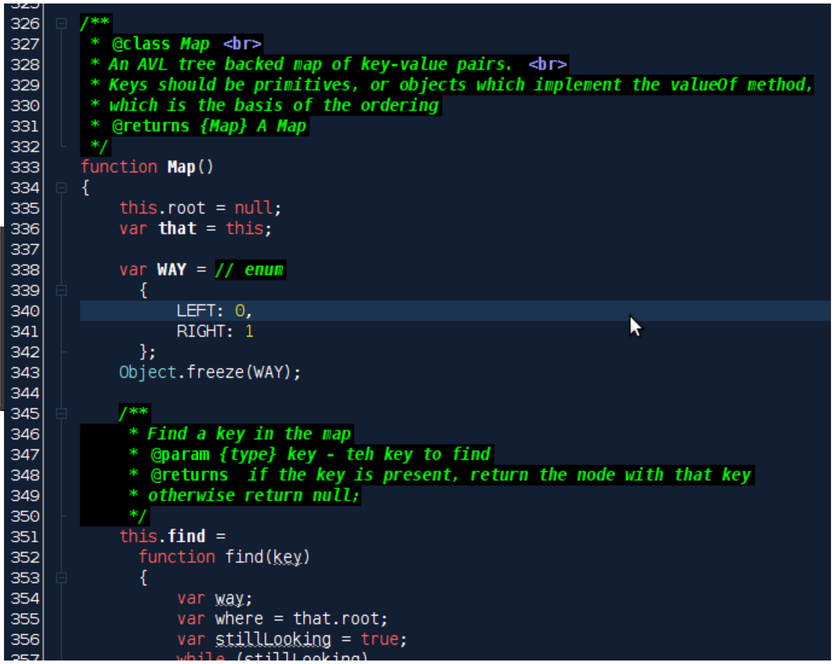
To use it, put comment documentation in the .js files as shown below. To create the web pages, do this from the command line:

```
jsdoc file.js -d=doc
```

That creates documentation from the file file.js, and puts it in the folder named doc.

Documenting comments comes from /* comment that start with a * - in other words that start

```
/**
*
```

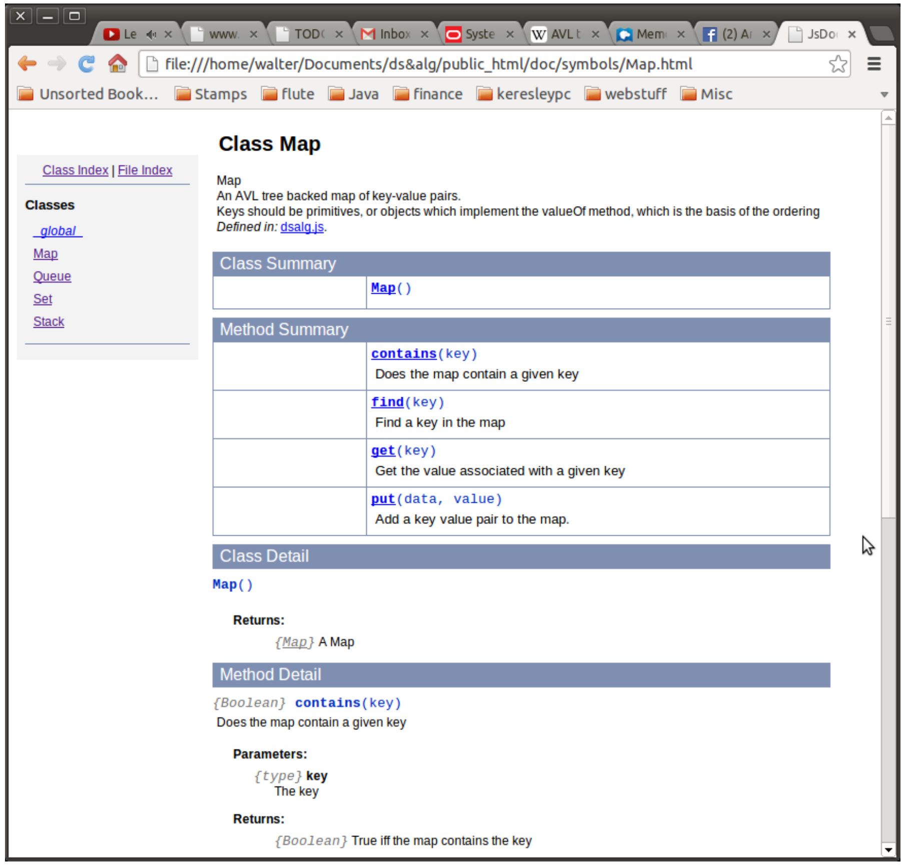


```
/**
 * @class Map <br>
 * An AVL tree backed map of key-value pairs.  <br>
 * Keys should be primitives, or objects which implement the valueOf method,
 * which is the basis of the ordering
 * @returns {Map} A Map
 */
function Map()
{
    this.root = null;
    var that = this;

    var WAY = // enum
      {
         LEFT: 0,
         RIGHT: 1
      };
    Object.freeze(WAY);

    /**
     * Find a key in the map
     * @param {type} key - teh key to find
     * @returns  if the key is present, return the node with that key
     * otherwise return null;
     */
    this.find =
      function find(key)
      {
         var way;
         var where = that.root;
         var stillLooking = true;
```

and you can use various *tags to format them.

For example, here is something to document a class and a method (the JavaScript is explained later):

This generates web pages which look like:



file:///home/walter/Documents/ds&alg/public_html/doc/symbols/Map.html

Class Index | File Index

Classes

global
Map
Queue
Set
Stack

Class Map

Map
An AVL tree backed map of key-value pairs.
Keys should be primitives, or objects which implement the valueOf method, which is the basis of the ordering
Defined in: dsalg.js.

Class Summary

	`Map()`

Method Summary

	`contains(key)` Does the map contain a given key
	`find(key)` Find a key in the map
	`get(key)` Get the value associated with a given key
	`put(data, value)` Add a key value pair to the map.

Class Detail

`Map()`

Returns:
`{Map}` A Map

Method Detail

`{Boolean} contains(key)`
Does the map contain a given key

Parameters:
`{type}` **key**
The key

Returns:
`{Boolean}` True iff the map contains the key

3.8 Primitives

We can have different kinds of data, like whole numbers, decimals, dates, times, colors, sound, currency and so on. The idea of data type is fundamental in programming.

In JavaScript there are 5 primitive types – String, Number, Boolean, Undefined and Null. In addition to primitive types, there are objects.

String is a sequence of characters, like “Hello”. Each character is represented by a code point (UTF-16).

Number is a.. well, a number. Some languages have a special whole number type (an integer or int.) In JavaScript number always has decimals (represented as a 64bit IEEE 754 value). There are three special values. NaN means 'not a number' – like the square root of a negative number (the paradox that 'not a number' is a number is just the start of the craziness associated with NaN). The other two are positive infinity and negative infinity. They are there to annoy mathematicians, who would say that infinity is not a number.

Boolean just has two possible values – true and false.

Undefined has one value – undefined. Any variable which has not been given a value has the value undefined.

Null has just one value – null.

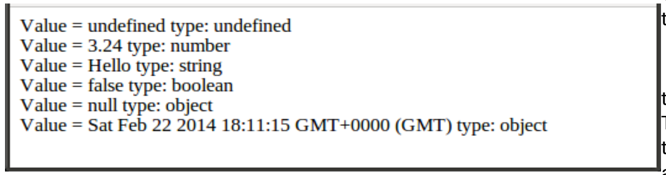
All other values are *objects*

In some languages (like Java) variables have a definite type, set when the variable is declared. The type cannot be changed.

In JavaScript, ***values have type. Variables do not have a type***. We can assign the same variable to different values, with different type.

For example

```
'use strict';
var x;
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
x=3.24;
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
x="Hello";
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
x=false;
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
x=null;
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
x=new Date();
document.write("Value = "+x+" type: "+typeof(x)+"<br>");
```



```
Value = undefined type: undefined
Value = 3.24 type: number
Value = Hello type: string
Value = false type: boolean
Value = null type: object
Value = Sat Feb 22 2014 18:11:15 GMT+0000 (GMT) type: object
```

The type of null is “object” because the standard at says so at 11.4.3.

typeof is slightly misleading. Typeof x looks like it gives you the type of x, but it does not, since x is a variable and variables don't have type. In fact it gives you the type of *the value of* x.

Date is a pre-defined object – which is actually a function. In the above code we use the function as a constructor, to make another object, which x is assigned to.

So as well as the primitives, there are some built-in predefined objects, and we can define our own additional ones.

Some of the built-in objects are Object, Function, String, Number, Boolean, Math and Date.

So there is a pairing between some primitive types and built-in objects – number and Number, string and String, boolean and Boolean. These are like Java wrapper classes, int and Integer.

3.9 JavaScript outside a browser

Usually JavaScript is executed by the interpreter embedded in a browser, but there is no reason why it cannot be executed by an interpreter outside a browser. Several are available, including

nodejs – built from Chrome's runtime,and intended to run server-side. See nodejs.org

SpiderMoney and its JavaScript shell, js. This is from Mozilla and runs inside Firefox and elsewhere. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Rhino from Oracle. This was originally from Netscape and compiled to Java bytecode. It is now managed by Mozilla, and there is an Oracle version.

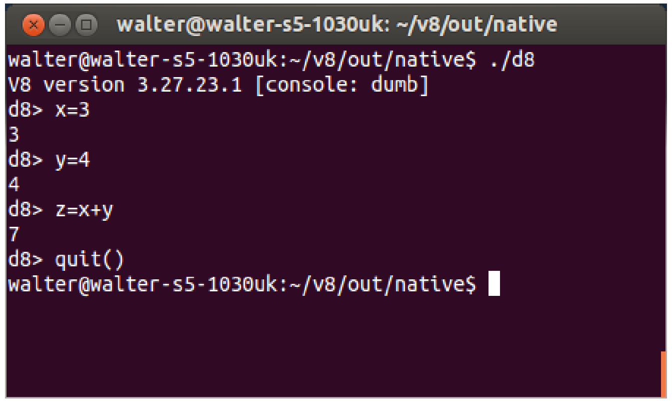
There is also v8, from Google. This has a shell called d8.

On Linux, v8 can be downloaded and built as follows:

```
svn checkout http://v8.googlecode.com/svn/trunk/ ./v8
cd v8
make dependencies
make native
```

The first command uses subversion, which you can download and install first if needed.

This creates a folder out in v8, and in that a folder named native. In that there is an executable named d8. For example:



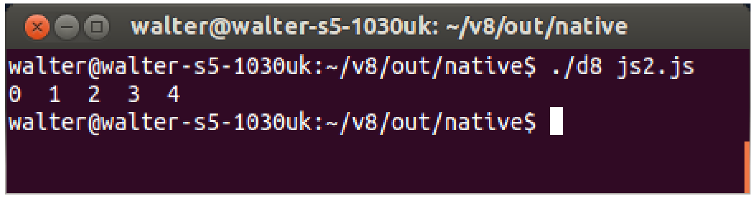
```
walter@walter-s5-1030uk:~/v8/out/native$ ./d8
V8 version 3.27.23.1 [console: dumb]
d8> x=3
3
d8> y=4
4
d8> z=x+y
7
d8> quit()
walter@walter-s5-1030uk:~/v8/out/native$
```

The shell outputs a prompt (d8>). You type in a JavaScript expression, and it is executed. The value is output. So 'x=3' has the value 3 and as a side-effect, x gets the value 3. Same for 'y=4' and 'z=x+y'. quit() exits the shell.

So using it that way is not very convenient. Better is to write some JavaScript in a text file. Then you can invoke d8 and pass the script file to it. For example, if js2.js is:

```
for (var i=0; i<5; i++)
    write(i, " ");
write("\n");
```

then:



```
walter@walter-s5-1030uk:~/v8/out/native$ ./d8 js2.js
0  1  2  3  4
walter@walter-s5-1030uk:~/v8/out/native$
```

The 'write' is non-standard, d8-specific.

3.10 Caution - Not A Number is a number

JavaScript follows Java in that as far as possible it uses C-style syntax. So assignments statements and control constructs are very similar. However there are some pitfalls.

NaN is an example. NaN is Not A Number – for example, what you get when you divide 0 by 0.

Numbers are represented as IEEE754 64bit floating point form. There are around 184 million million million possible values of that. Of those, 9 000 million million are reserved for NaN, not a number. This follows from the IEEE standard, where NaNs have all exponent bits 1, and significant bits not all zero – so there are a lot of NaN values.

There is a buit-in function isNaN:

```
var x=0/0;
console.log(x); // NaN
console.log(typeof x); // number
console.log(isNaN(x)); // true
```

But this actually involves type coercion, which in other languages is called a cast – a conversion of a value from one type to another. See the next section.

However, what isNaN actually does is to return true iff its argument, when coerced to a number, has the value NaN. If the coercion fails, it gives false. This does not seem entirely clear:

```
console.log(isNaN(NaN)); // true
console.log(isNaN("NaN")); // true
console.log(isNaN(undefined)); // true
console.log(isNaN({})); // true (empty object )
console.log(isNaN(true)); // false
console.log(isNaN(null)); // false
console.log(isNaN(3)); // false
```

It might be thought that you could just say if (x===NaN) ..:

```
var x=0/0;
if (x===NaN)
    console.log("x is NaN");
else
    console.log("x is not NaN"); // always get this
```

In fact NaN===NaN is always false (if you think that makes no sense at all, I agree). NaN is the only value for which x===x is false, so the standard actually suggests this as the way to detect NaN

```
var x=0/0;
if (x===x)
       console.log("x is not NaN");
else
       console.log("x is NaN"); // get this
```

ECMAScript 6 adds an isNaN() to the Number object:

```
var x = "Hello";
console.log(Number.isNaN(x)); // false
x="4";
console.log(Number.isNaN(x)); // false
x=4;
console.log(Number.isNaN(x)); // false
x=0/0;
console.log(Number.isNaN(x)); // true
```

The bad news is that support for ECMA 6 is very patchy (March 2014 – FireFox 30 is winning).

3.11 Caution – type coercion

As in C, '=' assigns a value, and '==' tests for equality:

```
console.log(x=1); // outputs 1
console.log(x==1); // outputs true
```

However

```
console.log(1=="1"); // also outputs true, which is probably unexpected
```

According to the standard, the rules of == are as follows (if you think what follows is totally confused and impossible to remember, you are correct):

x==y is true or false depending on:

1. If Type(**x**) **is the same as** Type(**y**), then

    1.If Type(x) is Undefined, return true: *undefined == undefined*

    2.If Type(x) is Null, return true: *null == null*

    3.If Type(x) is Number, then

        1.If x is NaN, return false: *NaN != NaN*

        2.If y is NaN, return false: *NaN != NaN*

        3.If x is the same Number value as y, return true: *2 == 2*

        4.If x is +0 and y is −0, return true: *0 == 0*

        5.If x is −0 and y is +0, return true: *0 == 0*

        6.Return false: *2 != 1*

    4.If Type(x) is **String**, then return true if x and y are **exactly the same sequence of characters** (same length and same characters in corresponding positions). Otherwise, return false: *"a" == "a" but "a" != "b" and "a" != "aa"*

    5.If Type(x) is Boolean, return true if x and y are both true or both false. Otherwise, return false: *true == true and false == false but true != false and false != true*

    6.Return true if x and y refer to the same object. Otherwise, return false:*var o = {}; o == o but o*

!= {} and {} != {} and [] != [] ... etc etc, all objects are eqeq only if it's the same

2.If **x is null and y is undefined**, return **true**:
null == undefined

3.If x is undefined and y is null, return true:
undefined == null

4.If Type(x) is Number and Type(y) is String, return the result of the comparison x == ToNumber(y): *2 == "2"*

5.If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y: *"2" == 2*

6.If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y: *false == 0 and true == 1 but **true != 2***

7.If Type(y) is Boolean, return the result of the comparison x == ToNumber(y)

8.If Type(x) is either String or Number and Type(y) is Object, return the result of the comparison x == ToPrimitive(y):
ToPrimitive means implicit valueOf call or toString if toString is defined and valueOf is not

Type coercion happens when according to the above, values are converted to strings or numbers or whatever.

This makes == very problematic to use. It is an evil twin, and it is better to use its good twin, which is ===. This simply returns true iff x and y are the ***same type*** and the ***same value***.

For example

```
console.log(1===1); // true
console.log("1"==="1"); // true
console.log(1==="1"); // false
console.log(undefined===undefined); // true
console.log(undefined===null); // false
var x;
console.log(x===undefined); // true
```

=== is slightly slower than ==. But slow code which works is better than fast code which fails.

3.12 Caution – truthy and falsy

The values of boolean type are true and false. But in a boolean expression, other values coerce to true or false:

truthy	falsy
true	false
objects	null
arrays	undefined
Non-empty strings	NaN
Non-zero numbers	0
	Empty string

As a result, JavaScript programmers talk about 'truthy', meaning values which will coerce to true:

```
if (new Date()) console.log("Object is truthy");
if ([1,2,3]) console.log("Array is truthy");
if ("false") console.log("'false' is truthy");
if (7) console.log("7 is truthy");

if (!null) console.log("not null is truthy");
if (!undefined) console.log("not undefined is truthy");
if (!(0/0)) console.log("Not NaN is truthy");
if (!"") console.log("Not empty string is truthy");
```

3.13 Caution – function hoisting

The definition of a function is in effect shifted from where it is actually given, to the top of the script.

For example, consider this:

```
  'use strict';
  console.log(f1(2));
  // console.log(f2(2)); Uncaught TypeError: Property 'f2' of object [object Object]
is not a function


  function f1(x)
```

```
{
    return x * 3;
}

var f2 = function(x) // make f2 a function
{
    return 4 * x;
};

console.log(f2(2)); // this is ok - get 8
```

So we can invoke f1 before (in the script) the definition of f1 has been reached. In effect the interpreter makes two passes of the source, and in the first pass shifts all function definitions to the top.

But the lines starting

```
var f2 = function(x) // make f2 a function...
```

are not shifted up – as executable statements they cannot be moved. So invoking f2 at the start produces this exception – f2 is actually undefined there. Invoking it after the assignment is OK.

So function definitions are 'hoisted' to the top of the script

3.14 Caution – variable hoisting

Variables are hoisted just like functions.

A variable declared at global scope is available everywhere, but it is hidden by a variable declared in local scope (in a function)

```
var x;
x=1;
function bar() {
    var x=2;
    console.log(x); // 2
}
bar();
console.log(x); //1
```

'x' is declared in the function, so the local value 2 is used there. At global scope the global value 1 is used. This is as expected.

But the JavaScript interpreter makes two passes of the source code, and shifts all var declarations to the top of the scope. This is called variable hoisting. So the above is the same as:

```
x=1;
function bar() {
    var x=2;
    console.log(x); // 2
}
var x;

bar();
console.log(x); //1
```

Declaring x anywhere in global scope is the same – the interpreter hoists it to the top of the scope.

Variable hoisting can sometimes be misleading. For example:

```
var foo = true;
function bar() {
    if (foo===undefined) { // it is undefined
        var foo = false;
    }
    else
        {
        var foo = true;
    }
    console.log(foo); // false
}
bar();
```

foo is declared again inside the function scope, so after hoisting this is equivalent to:

```
var foo = true;
function bar() {
    var foo;
    if (foo===undefined) { // it is
        foo = false;
    }
    else
        {
        foo = true;
    }
    console.log(foo); // false
}
bar();
```

So foo is local to the function, and at the if, it is undefined. So false is assigned to it, and that's what we see.

Because of variable hoisting, many people suggest you should var declare variables at the top of their scope. This means that the way that you see it will be the same as what the interpreter actually runs.

3.15 Namespaces

It is a good idea to use meaningful identifier names – in JavaScript, that means variables and functions with names which reflect what they represent.

A problem linked with this is that we may well choose an identifier name which has already been used elsewhere in code, and re-using it may corrupt a value.

The idea of a namespace fixes this – a namespace limits the region of code where an identifier name applies. In Java, packages and classes form namespaces. For example, there are two Date classes – one for dates in general, and another for dates in SQL databases. No problem, since one is java.util.Date, and the other is java.sql.Date, in a different package.

Unfortunately JavaScript has no such modularity, and everything goes into one namespace. This even applies if you have two or more external script files, so if the html says

```
<script src="one.js"></script>
<script src="two.js"></script>
```

then var x in one.js is the same as var x in two.js.

Initially JavaScript would just supply a few dozen lines of code in a web page, but today we might use several unrelated libraries with thousands of lines of code, and the namespace problem is significant.

Anonymous functions provide one solution.

3.16 Anonymous functions

There are two aspects to a function:

1. A piece of code we can invoke several times
2. A local scope

The first point is the usual reason to define a function. An anonymous function abandons the first, and is there for the second.

For example:

```
(function() {
    var k=2+3;
    console.log(k);
})();
```

This has defined a function, with no name:

```
(function() {
    var k=2+3;
    console.log(k);
})
```

and then invoked it:

```
..();
```

We cannot invoke it again, without repeating the source code, since it has no name. But we have done some processing, without introducing anything into the global namespace – the function has no name, and var k is local to the function.

We can pass values into an anonymous function through parameters, and use return to get a value passed back:

```
var value = (function(n) {
    var total=0;
    for (var i=1; i<=n; i++)
        total+=i;
    return total;
})(3);
console.log(value); // 6
```

Or, inside the function we can assign to a global value:

```
var k;
(function(n) {
    var total=0;
    for (var i=1; i<=n; i++)
        total+=i;
    k=total;
})(3);
console.log(k); // 6
```

Suppose we want the processing to be done recursively? The code must call itself – which means it must have a name to be called. We can do this anonymously if the named code is an inner function with local scope:

```
(function(n)
{
    console.log(fib(n)); //5

    function fib(m)
    {
        if (m===1 || m===2)
            return 1;
        return(fib(m-1)+fib(m-2));
    }
})(5);
```

3.17 Objects as namespaces

Another way to deal with the namespace issue is to use an object as a namespace. Variables and functions we want to define and assign to are created as properties of the object:

```
var NS = {}; // make NS to be an empty object

NS.x = 27; // we want a variable x restricted to this namespace
NS.f = function() // and a function
{
    console.log("Executing function f");
};

NS.f(); // use the function
```

Suppose other code also uses the name NS? We can check:

```
  var NS=7; // other code uses NS

  if (NS!==undefined) throw "Namespace clash on NS";
  // if we get here, NS is free to use..
  var NS = {}; // make NS to be an empty object
..
```

A variation on this is to use a namespace which is obscure – following the Java approach for package naming, it could be the reverse of the domain name of the organisation, such as:

```
  var comMilnerWalter = {}; // make obscure namespace

  comMilnerWalter.x = 27; // we want a variable x restricted to this namespace
  comMilnerWalter.f = function() // and a function
  {
      console.log("Executing function f");
  };
```

However, when we want to use f, we have to say

```
  comMilnerWalter.f();
```

which is a lot of typing. We can avoid this by setting up an alias:

```
  var NS = comMilnerWalter; // alias
  NS.f(); // use the function
```

4 Type and objects

When there is no type hierarchy you don't have to manage the type hierarchy.

—Rob Pike

There are a set of ideas which only make sense together. Here they are. In JavaScript

1. Values have a type

2. Variables do not have type. As their values change, the type of the value of a variable might change.

3. There are some primitive types, and the 'object' types, as shown below.

4. An object is a bundle of named values, called properties. An object is like an associative array of property names and values.

5. A function is an object. This means some objects are functions. It also means we can do functional programming.

6. There are no classes.

7. Each object has a link to another 'prototype' object, up a chain, ending at null at the top.

8. An object inherits the properties of its prototype.

9. There are a set of pre-built global objects, like Number and String, which provide convenient starting points for the tops of prototype chains. Some of these correspond to the primitive types of number, string and so on. All functions have the Function object as their prototype.

10. Auto-boxing and unboxing happens.

The JavaScript type hierarchies are:

Primitive types (String, Number, Boolean, Undefined and Null)

Objects

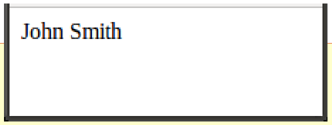
Pre-built objects – Object, and objects descending from this – Function, String, Boolean, Number, Math and Date

Programmer-defined objects. These can descend from one of the above, or null.

4.1 Objects

An object is a bundle of named property values:

```
var x=new Object();
x.firstName="John";
x.lastName="Smith";
x.rateOfPay=9.55;
document.write(x.firstName+" "+x.lastName);
```



```
John Smith
```

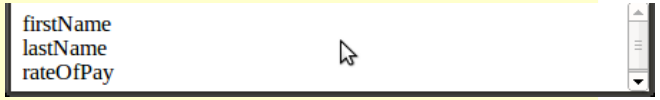
So an object is a bit like a C struct – a set of fields. But there are differences. The fields do not have type (their values do). And we can add further fields simply by assigning to them.

Or we can use an equivalent array syntax;

```
var x = {firstName:"John", lastName:"Smith", rateOfPay:9.55};
document.write(x['firstName']+" "+x['lastName']);
```

We can iterate through the properties of an object using for..in (this may not show all of them – see later):

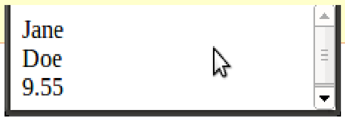
```
var y=new Object();
y.firstName="Jane";
y.lastName="Doe";
y.rateOfPay=9.55;

for (var prop in y)
   document.write(prop+"<br>");
```



```
firstName
lastName
rateOfPay
```

or we can get the property values:

```
for (var prop in y)
   document.write(y[prop]+"<br>");
```



```
Jane
Doe
9.55
```

4.2 The global object

Before execution starts the host environment (normally the browser) constructs what is known as the 'global object'. We can refer to this object using 'this'. For example:

```
console.log(this);
```

outputs:

```
Window{top:Window,window:Window,location:Location,external:Object,chrome:O
bject…}
```

so in a browser, the global object is the window – part of which is the html document. This is how JavaScript can access the web page elements.

In fact 'global variables' are in fact properties of this global object:

```
var someField =4;
for (var prop in this)
    console.log(prop, this[prop]);
```

outputs:

```
top
Window {top: Window, window: Window, location: Location, external: Object,
chrome: Object…}
window
Window {top: Window, window: Window, location: Location, external: Object,
chrome: Object…}
..
document #document
someField 4
prop prop
..
```

Note that when 'prop prop' is being output, the value of the variable prop is prop, and the prop property of the window is 'prop'.

Or we can refer to it as

```
var someField =4;
console.log(window.someField); // 4
```

4.3 Creating objects

There are several ways to do this. One way is to just assign to a list of name-value pairs:

```
var myObject = {
    x: 2,
    y: 3,
    func : function() {return this.x+this.y;}
};
console.log(myObject.x, myObject.y, myObject.func()); // 2 3 5
```

So myObject has three properties. x and y are primitives, and func is a function. Inside a function definition, 'this' means the object executing the function (outside it means the global object). Functions are objects.

A second way is to use the built-in object named Object as a prototype, and add fields to it, like

```
var myObject = new Object();
myObject.x=2;
myObject.y=3;
myObject.func = function() { return this.x+this.y; };
console.log(myObject.x, myObject.y, myObject.func()); // 2 3 5
```

Suppose we wrote

```
myObject.func = function() { return this.x+this.Y; };
```

Then myObject.func() is NaN – a special number value meaning 'not a number' . We get the same for

```
var a=2;
var b;
console.log(a+b);
```

Here b is undefined, and adding undefined to 2 gives NaN. The issue is we have referred to the Y property of myObject, but it does not exist. We get no syntax error, no exception or error message – just crazy output. Difficult to debug.

That way actually uses Object as a constructor function. This is a bit clearer if we use our own function as a constructor -

```
function MyObject(a,b)
{
    this.x=a;
    this.y=b;
    this.func = function() {return this.x+this.y;};
}

var myObject = new MyObject(2,3);
console.log(myObject.x, myObject.y, myObject.func()); // 2 3 5
```

This in effect is the equivalent of a class in Java, since it allows us to make a set of objects with the same pattern. There are no actual classes in JavaScript.

A fourth way is to use Object.create. This takes an object as argument, and it creates another object which has the argument as its prototype. The new object *inherits* the properties of the prototype:

```
function MyObject(a,b)
{
    this.x=a;
    this.y=b;
    this.func = function() {return this.x+this.y;};
}

var baseObject= new MyObject(2,3);
var myObject = Object.create(baseObject);
console.log(myObject.x, myObject.y, myObject.func()); // 2 3 5
```

4.4 Object properties

We can create properties by just assigning to them:

```
var x={}; // empty object
x.f=2; // add property
console.log(x.f); // 2
```

But in fact properties have attributes which can be more closely controlled than this. We can do this with Object.defineProperty, which allows us to control the attributes of a property:

```
var x={}; // empty object
Object.defineProperty(x,"f",
{
    value:2,
    configurable: false,     // can we alter property attributes or delete it?
    enumerable: true,        // does it appear in a for .. in.. property enumeration
    writable:false           // can we assign to it

});

console.log(x.f);            // 2
// delete(x.f);              // try to remove property - get Uncaught TypeError:
Cannot delete property 'f' of #<Object>

for (var p in x)
    console.log(p);          // f
// x.f=1;   // try to assign to it - get Uncaught TypeError: Cannot assign to read
only property 'f' of #<Object>
```

Objects also inherit properties through the prototype chain – so for example

```
var a={};        // apparently empty object
Object.prototype.x=4;     // now all objects have Object prototype in their
             //chain have a property x
console.log(a.x);         // 4
```

4.5 Internal properties

In addition to own properties and inherited properties, an object will have internal properties. The ECMA standard sets out a list of internal properties which all objects must have. One example is [[extensible]]. This notation is used to show that the property cannot be directly referred to by code. The internal [[extensible]] property flags whether an object can have extra properties added to it.

Sometimes there are methods which enable indirect access to internal properties. For example

```
var a={}; // object
Object.preventExtensions(a); // can't add more properties
```

```
a.x=4; // gets Uncaught TypeError: Can't add property x, object is not extensible
```

Note that:

```
var a={}; // object
a.extensible=false;
a.x=4; // no problem
```

This is because the 'normal' property .extensible has just been added, and has no connection with the internal [[extensible]] property.

The required internal properties which all objects must have are:

Internal property	Idea	Accessed
[[Prototype]]	The prototype of the object	Object.get
[[Class]]	'A specification defined classification of objects'	getString()
[[Extensible]]	Whether its extensible	Object.pre
[[Get]]	Returns the value of a given property	Set by Obj
[[Get own property]]	Returns a given property's descriptor	Object.get
[[Put]]	Sets the value of a given property to the suppliedvalue	Set by Obj
[[CanPut]]	Whether a property value can be set	Object.get
[[HasProperty]]	Whether the object has a given property	Object.get
[[Delete]]	Remove a given property	Delete key
[[Default value]]	The standard says 'Returns a default value for the object. '	
[[DefineOwnProperty]]	Adds or modifies a property with a property descriptor	Object.def

For example:

```
var base = new Object();
var sub = Object.create(base);

console.log(Object.getPrototypeOf(sub)); // Object {} [[prototype]]
console.log(sub.toString()); // [object Object]  [[class]]
```

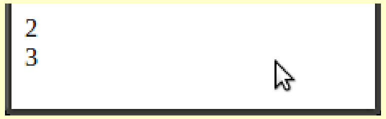
```
Object.defineProperty(sub, "x", {get: function() {
        return 5;
    },
    set: function(newValue) {
    },
    enumerable: true,
    configurable: true});

sub.x = 4;
console.log(sub.x); // 5 - using the [[get]]
console.log(Object.getOwnPropertyDescriptor(sub, "x")); // Object {get: function,
set: function, enumerable: true, configurable: true}
delete sub.x;                                          // [[delete]]
console.log(Object.getOwnPropertyDescriptor(sub, "x")); // undefined
```

4.6 No classes in JavaScript

In the last section we created two objects which had the same set of fields. In Java or C++ we would say they would instantiate the same class. But not in JavaScript, so there are no classes in JavaScript.

To create objects like this, with the same fields, we need to repeat some code. That is what a function does. So we use a function, together with the keyword new, to create an object:

```
function TypeOne(value) {
   this.field1 = value;

}

var x = new TypeOne(2);
var y= new TypeOne(3);
document.write(x.field1+"<br>");
document.write(y.field1+"<br>");
```

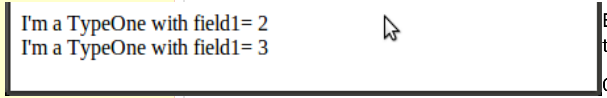


```
2
3
```

This is like Java, except the constructor has become the entire class definition.

What about methods? A function is an object. So define a function in the constructor, and assign it to a field of the object:

```
 function TypeOne(value) {
   this.field1 = value;
   this.toString=toString1;


   function toString1() {
      var s = "I'm a TypeOne with field1= " + value;
      return s;
   }
}

var x = new TypeOne(2);
var y= new TypeOne(3);
document.write(x.toString()+"<br>");
document.write(y+"<br>");
```



```
I'm a TypeOne with field1= 2
I'm a TypeOne with field1= 3
```

We've called the function toString1, then assigned that to a property toString. The we can say x.toString() to invoke the method. We could have named the function toString, but we show we don't have to.

In fact toString is a method of Object, which is inherited by all other objects, and here we have over-ridden it. This is discussed later. ToString returns a string representation of the object (as in Java), and is invoked when a string is required. So we could say document.write(y+"<br>"); and the toString is invoked for us.

4.7 Primitives and object references and aliases

As in Java, the semantics of variables with primitive values and object values are different. For example:

```
var x = 3;
var y = x;
x++;
console.log(y); // still 3
```

but contrast that with

```
x=new Object();
x.f1=3;
y=x;
x.f1++;
console.log(y.f1); //now 4
```

In the first case, y=x means we get a new number, and the value of x is copied to y – we have two 3s. Changing x does not affect y.

But is the second, we've only got one object, with a property f1. X refers to that object, and y=x makes y refer to the *same* object (not a new copy). So if we change x, y seems to change – in fact its the same thing.

So a variable with an object value is in effect like a pointer in C or a reference in Java.

When we said x=y, then x and y refer to the same thing - in other words x is an alias of y. This is sometimes useful.

4.8 Prototypes and inheritance

Every object has a link to another, its prototype, except those at the top of the chain, for which the link is null.

Objects inherit the properties (and values) of their prototypes, as shown here:

```
 // create object x with 2 properties, f1 and f2
var x = { f1:1, f2:2 };

// create new object y, which has x as its prototype
var y= Object.create(x);
y.f3=3; // add property f3


// make z with prototype y
var z=Object.create(y);
z.f4=4; // add property f4

// list all properties of z
for (var prop in z)
    console.log(prop+" = "+z[prop]);
/* outputs
f4 = 4
f3 = 3
f1 = 1
f2 = 2
 */

// list own properties, not inherited ones
var ownProps = Object.getOwnPropertyNames(z);
for (var i=0; i<ownProps.length; i++)
    console.log(ownProps[i]+ " = "+ z[ownProps[i]]);
/* outputs just
 f4 = 4
 */
```

4.9 Functions

The standard use of functions is like this

```
'use strict';
function average(x,y)
{
   var total=x+y;
   return total/2;
}

var a=1;
var b=2;
document.writeln(average(a,b));
```

```
1.5
```

This is pretty much like C – we define a function, then call it. There is no return type or parameter types, because variables don't have type.

There is global scope and local scope as expected.

Parameters are passed by value (a copy is passed):

```
function fooBar(x, y) {
   x++;
   y.field++;
}

var x = 1;
var obj = new Object();
obj.field=1;
fooBar(x, obj);
document.writeln(x + " " + obj.field);
```

```
1 2
```

This is the same as Java (with the same confusion). A copy of x is passed into fooBar. The copy is incremented. No effect on the original. A copy of obj is passed in as the second parameter. But obj is a reference – a pointer to the object. So in fooBar y points to the same object as obj did, and if we change its fields, we are changing the fields of obj as well, since its the same object.

4.10 Primitives and auto-boxing

There are primitive types for string boolean and number. There are also pre-built String Boolean and Number objects. These are not the same:

```
 var s1 = "abcde";
console.log(typeof s1); // 'string'

var s2=new String("abcde");
console.log(typeof s2); // 'object'
```

The primitive types are just immutable values. The objects also have methods (inherited from their prototypes).

For example a String has a slice method:

```
var s2=new String("abcde");
console.log(s2.slice(1,4)); // 'bcd'
```

In Java terms, String is a wrapper for a primitive string.

JavaScript also has autoboxing. When a property of a primitive is referred to, a temporary object of the wrapper type is created and used – and is then discarded.

For example:

```
var s1 = "abcde";
console.log(typeof s1); // 'string'
console.log(s1.slice(1,4)); // bcd
```

s1 is a primitive string. We make a reference to the slice method – but a primitive does not have any methods. So a temporary String object is created on s1, the slice method of that object is used, then the object is discarded.

This might be confusing. For example:

```
s1.prop1=4;
console.log(typeof s1.prop1); // undefined
```

s1 is a string primitive not an object, so it can't have a property. But s1.prop1 refers to it – so a temporary String object is created, a property prop1 is added to it, and a value of 4 assigned to it. That temporary object is then discarded – so s1.prop1 is now undefined.

4.11 Functions as objects

We can say:

```
var f = function fooBar(x)
{ return x+1; };

document.write(f(1));
```

This is similar to before, except that we've said var f = function..

So we've assigned a function to an object. In other words we can treat a function as an object.

4.12 Closures

A closure is a code unit (like a function) which retains the values from its environment, even after those values have changed, or disappeared. A closure in effect retains a frozen copy of the values in its environment when it was constructed.

In JavaScript a closure is usually an inner function inside another function. For example:

```
 function Outer(inc)
{
    this.adder = function adder(val)
    {
        return val + inc;
    };
}

// make an Outer object.
// assign add1 to its adder property (a function)
var add1 = (new Outer(1)).adder;
console.log(add1(2)); // 3
// make another Outer object.
// assign add2 to its adder property
var add2 = (new Outer(2)).adder;


console.log(add2(2));  // 4
console.log(add1(5));  // 6
```

In function adder, we add 'inc' which is a parameter supplied to the outer function Outer. Usually the lifetime of a parameter is just the lifetime of its function execution. But an instance of adder remembers the value of inc when it was created, even if there is a subsequent call with a different value of inc.

We can get a similar effect without a parameter:

```
var inc=0;

function Outer()
{
    this.x=inc;
    var that=this;
    this.adder = function adder(val)
    {
        return val + that.x;
    };
}

// make an Outer object.
// assign add1 to its adder property (a function)
inc = 1;
var add1 = (new Outer()).adder;
console.log(add1(2)); // 3
// make another Outer object.
// assign add2 to its adder property
inc=2;
var add2 = (new Outer()).adder;


console.log(add2(2));  // 4
console.log(add1(5));  // 6
```

It would make more sense to say:

```
return val + this.x;
```

but this inside an inner function goes wrong. This hack to avoid it is thanks to Crockford.

Closures are useful to imitate private class members.

4.13 Dot and [ ] notation

These are not equivalent.

```
a.x=0;
```

creates a property of object a named x (if it does not already exist).

```
a[x]=3;
```

creates a property named *the value of* x. The difference is that using [ ] the item is evaluated.

If x is undefined, this creates a property whose name is undefined:

```
var a = {};
a.x=0;
a[x]=3;
var s="Properties of a : ";
for (var p in a)
    s+=p+" : "+a[p]+" ";
console.log(s); // Properties of a : x : 0 undefined : 3
```

Rather bizarrely, we can delete this property with undefined as a name, then create some more:

```
delete a.undefined;

for (var x=0; x<3; x++)
a[x]=x+1;
s="Properties of a : ";
for (var p in a)
    s+=p+" : "+a[p]+" ";
console.log(s); // Properties of a : 0 : 1 1 : 2 2 : 3 x : 0
```

4.14 A dynamic language

In a conventional static language, the programmer writes the code, and then, interpreted or compiled, that code is executed.

In a dynamic language, the code can change itself at run-time. JavaScript is a dynamic language.

For example

```
var a={};
var name=prompt("Enter name", "x");
var value=prompt("Enter value", "5");
a[name]=value;
```

This adds a property to object 'a'. However the name and value of this property are not determined until runtime.

Another example:

```
var name = prompt("Name: ", "squareFunc");
var arg = prompt("Arg:", "x");
var body = prompt("Body", "return x*x;");
this[name] = new Function(arg, body);
console.log(this[name](2));
```

This creates a new function and executes it. But the name, argument and body of the function are not known until runtime.

In this code, 'this' refers to the 'global object'. this[name] adds a property to this object, and is equivalent to saying

```
var squarefunc = ...
```

but that fixes the function name to squarefunc. This way, the function name can be chosen at runtime.

Traditionally, dynamic languages with code which modifies itself, is associated with LISP and AI.

4.15 Keyword 'this'

The keyword 'this' in a function is a reference to the object executing the function (but what if the function is executing at top-level with no object? Discussed in a while).

For example:

```
          function func(a,b) // define func
          {
            this.field=a;
            return a+b;
          }


         var obj={}; ///make empty object

         obj.f=func; // set func as a property of obj
         console.log(obj.f(2,3)); // invoke f - get 5
         console.log(obj.field);  // 2
```

The object obj is executing the function – so this.field=a sets the field property of obj to be a = 2.

If the function is used as a constructor, 'this' refers to the object being constructed:

```
          function func(a,b) // define func
          {
            this.field=a;
            return a+b;
          }

         var obj= new func(4,5); // make  object
         console.log(obj); // func {field: 4}
         console.log(obj.field);  // 4
```

If the function is invoked by using call or apply, these have the first argument for the 'this' value:

```
          function func(a,b) // define func
          {
            this.field=a;
            return a+b;
          }

         var obj= {};
         console.log(func.call(obj,6,7)); // 13
         console.log(obj.field);  // 7
```

But what happens if the function is called directly, at the top level, not on an object? Or if you refer to 'this' not in a function?

The answer is different in strict mode or not. Not in strict mode, 'this' refers to the global object, which in a browser is window:

```
          function func(a,b)
          {
            this.field=a;
            return a+b;
          }

          console.log(func(1,2)); // 3
          console.log(this); // Window {top: Window, window: Window,...
          console.log(window.field); // 1
```

But in strict mode, 'this' is undefined:

```
Uncaught TypeError: Cannot set property 'field' of undefined
```

which is at

```
this.field=4;
```

5 The pre-built objects

When execution starts, there is a global object already created. In a browser, this is 'window'. As well as browser and document-related properties, this also has a set of pre-built objects. Some of these relate to the primitive types, whilst others are in effect bundles of useful library code:

```
console.log(window.Number); // function Number() { [native code] }
```

This section looks at some of them. Consult web references for an exhaustive list.

5.1 Number

A Number object wraps a primitive number value. For example:

```
        console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
        var num = new Number(123.456);
        console.log(num.toFixed(2)); // 123.46
        console.log(num==123.456); // true,using type coercion
        console.log(num===123.456); // false - different types
```

5.2 Boolean

This wraps a primitive boolean type:

```
        var bool = new Boolean(false);
        console.log(bool.toString()); // false
        console.log( bool==false);  // true - type coercion
        console.log(bool===false);  // false - different types
        if (bool) // bool is defined, and this is truthy
        {
          console.log("Surprise!"); // Surprise!
        }
        if (bool==false)
        {
          console.log("Expected");
        }
```

5.3 String

A String object is different from a string primitive:

```
        var str =new String("One Two Three \" \u0634"); // Arabic
```

```
        console.log(str.toString()); // One Two Three " ش

        console.log(str.toUpperCase().toString()); // ONE TWO THREE " ش
        console.log(str.charAt(1)); // n
        console.log(str.slice(0,3)); // One
        console.log(str.indexOf("Two"));  // 4
        var words=str.split(" "); // words is an array
        console.log(words[2]); // Three
```

5.4 Date

Like Java, a JavaScript date object wraps an ineger which represents a moment in time that number of milliseconds past (or before) midnight on 1st January, 1970:

```
         var d1=new Date(); //now
        console.log(d1); // Sun Jun 08 2014 14:05:04 GMT+0100 (BST)
        var d2=new Date(1000); // 1000 milliseconds past Unix epoch
        // which is Midnight Jan 01 1970
        console.log(d2); //Thu Jan 01 1970 00:00:01 GMT+0000 (GMT)
        var d3=new Date(1990, 1,2);
        console.log(d3); // Fri Feb 02 1990 00:00:00 GMT+0000 (GMT)
        console.log(d3.getMonth());  // 1 (Jan = 0 )
```

5.5 Math

A collection of constants and standard functions:

```
console.log(Math.PI);// 3.141592653589793
var PiBy2=Math.PI/2;
console.log(Math.sin(PiBy2)); // 1
console.log(Math.cos(PiBy2)); // 6.123031769111886e-17
console.log(Math.pow(2,3)); // 8 = 2 to the power 3
console.log(Math.random()); // 0.897883019875735 random number >=0 and < 1
var val = 10+Math.floor(21*Math.random()); random integer from 10 to 30 inclusive
console.log(val);
console.log(Math.ceil(5.8)); 6 = smallest integer > 5.8
```

5.6 Function

The Function object is a function constructor which constructs functions.

It takes a set of string arguments, which are the function arguments, followed by a string which is the function body. For example:

```
var bigger = new Function("a", "b", " if (a>b) return a; else return b;");
console.log(bigger(4,5)); // 5 - uses type coercion passing numbers to strings
```

This example could also have been done (faster) by defining the function in code:

```
function adder(a,).....
```

The Function approach enables the dynamic creation of functions at runtime, which is sometimes useful.

5.7 Array

The Array object is a constructor function to make array objects. This seems simple at first sight:

```
var array = new Array(3);
array[0]=1;
array[1]=28;
array[2]="Hello";
for (var index=0; index<array.length; index++)
  console.log(index+" : "+array[index]);
```

But you can also say:

```
var array = new Array(3);
array[0]=1;
array[1]=28;
array[2]="Hello";
array["test"]=99;
console.log(array["test"]); // 99
```

This suggests that these are actually associative arrays or maps, consisting of key-value pairs, with the key being any type, not just integer.

But :

```
var array = new Date();
array[0]=1;
array[1]=28;
array[2]="Hello";
array["test"]=99;
console.log(array["test"]); // 99
```

JavaScript objects are collections of property names and values, and we are using this, with the x[y] notation. This is true for Date or Object or any other object, and this behaviour is common to them all, not just Array instances.

So what kind of array are they? Fixed or variable length? Homogeneous or hetero (mixed element type)? Associative array or not? We have to look at the ECMAScript standard to see what they are. At 15.4 it says:

> *Array objects give special treatment to a certain class of property names. A property name P (in the form of a String value) is an array index if and only if ToString(ToUint32(P)) is equal to P and ToUint32(P) is not equal to 2^{32}1. A property whose property name is an array index is also called an element.*

So a JavaScript array is an object which treats some properties in a special way. If the property is a[p], and when you turn p from a string to an integer and back again you get back to where you were (in other words p looks like an integer) then a[p] is an array element, which is treated differently from other object properties.

The length property is one more than the largest index (which is read/write). This means they can be sparse arrays:

```
var array = new Array();
array[0]=1;
array[1]=28;
array[99]="Hello";
console.log(array.length); // 100
console.log(array[70]); // undefined
array.length=2;
console.log(array[99]); // undefined
```

So JavaScript arrays are objects which treat property names which are integers (or integer-like strings) as elements. They are variable length, heterogeneous and can be sparse.

Some examples of the way array elements are handled:

```
 var array = new Array();
array[0]=1;
array[1]=28;
array[9]="Hello";
console.log(array.toString()); // 1,28,,,,,,,,Hello
var another = new Array(5,6,7,8,9);
console.log(array.concat(another).toString()); // 1,28,,,,,,,,Hello,5,6,7,8,9
console.log(array.join(" - ")); // 1 - 28 -  -  -  -  -  -  -  - Hello
console.log(array.pop()); // Hello
array.push(7);
console.log(array.toString()); // 1,28,,,,,,,,7
array.reverse();
console.log(array.toString()); // 7,,,,,,,,28,1
console.log(array.shift()); // 7
console.log(array.toString()); // ,,,,,,,28,1
var sl = another.slice(1,3);
console.log(sl.toString()); // 6,7
var jumble = new Array(7,1,8,3,9,4,5);
console.log(jumble.sort().toString()); // 1,3,4,5,7,8,9
```

Check the standards for the rest.

5.8 Regular Expressions

The RegExp constructor function creates regular expressions.

A regular expression is a string which can be used to search for a match in another string. For example:

```
var regExp=new RegExp("abc");
console.log( regExp.test("qwe abc ert")); // true - contains abc
console.log( regExp.test("qwe ert"));     // false - does not
```

But that could be done by the String indexOf() method. A reg exp can be far more flexible. For example:

```
var regExp=new RegExp("a+bc*"); // one or more a's, then one b, then zero or more c's
console.log( regExp.test("qwe aaaab ert")); // true
console.log( regExp.test("qwe bbbcccccc ert"));     // false
console.log( regExp.test("aaaabcccccc ert"));     // true
```

or

```
var regExp=new RegExp("^[0-9]+.*[a-z]$"); // starts with 1 or more digits :  ^[0-9]
// then any number of any characters except newline : .*
// ends with a lower case letter a to z : [a-z]$
console.log( regExp.test("01234 what !")); // false
console.log( regExp.test("1t"));     // true
console.log( regExp.test("9 erT"));     // false
console.log( regExp.test("9 ! t"));     // true
```

An alternative notation is /like this/

```
var regExp=/[0-9]/; // a digit 0 to 9

console.log( regExp.test("0")); // true
console.log( regExp.test("a12"));     // true
console.log( regExp.test("a"));     // false
```

Check the reference for other special meanings.

6 OOP in JavaScript

Object-oriented programming uses several key concepts:

- Objects as bundles of code and data
- Classes as types of objects
- Inheritance of class members in subclasses
- Over-riding of methods in sub-classes to provide polymorphism
- Restricted access to members to provide encapsulation

JavaScript is not pure OO. But then neither is Java (which has primitives) and C++ which retains global data and procedural programming from C.

In fact JavaScript is uses prototype-based inheritance, which we can use to form patterns which provide full OO.

6.1 Classes in JavaScript

There aren't any (despite the [[class]] inner property).

If we create objects like:

```
var person1 =
  {
      name: "John",
      age: 23
  };
```

then we would have to take care to create all Person instances with the same members. But if we use a function constructor, we in effect have a class:

```
function Person(name, age)
{
    this.name=name;
    this.age=age;
}
```

```
var p1=new Person("John",23);
var p2=new Person("Jane", 42);
for (var p in p1)
    console.log(p,p1[p]); // name John age 23
```

We are following the convention of having the 'class' name starting upper case.

We can include method members as well:

```
function Person(n, a)
{
    this.name=n;
    this.age=a;
    this.display = f;
    var that=this;

    function f()
    {
        console.log(that.name+" "+that.age);
    }
}

var p1=new Person("John",23);
var p2=new Person("Jane", 42);
p1.display(); // John 23
```

This uses the Crockford 'that=this' hack.

Another way to do this is using the prototype. When we define a function (which is an object), another object is constructed, and the prototype property of the function object points to it.

Then when we create a new object using the constructor function, it inherits what is in the prototype. So if we add a member to the prototype, all instances get a copy:

```
function Person(n, a)
{
    this.name=n;
    this.age=a;
};


Person.prototype.display =function ()
    {
        console.log(this.name+" "+this.age);
    };

var p1=new Person("John",23);
var p2=new Person("Jane", 42);
p1.display(); // John 23
console.log(Object.getPrototypeOf(p1));   // Person {display: function}
```

Or instead of adding a new method to the automatically created prototype, we can assign it to a new object. And we can have the initialisation of the data members as a method in it:

```
function Person()
{
};


Person.prototype =
  {display: function ()
    {
        console.log(this.name+" "+this.age);
    },
    init: function(n,a)
    {
        this.name=n;
        this.age=a;
    }
  };

var p1=new Person();
p1.init("John", 23);
p1.display(); // John 23
```

6.2 Inheritance and Polymorphism

We need to be able to

- define a superclass with fields and methods
- define a subclass which inherits the superclass members
- add members to the subclass
- override superclass members in the subclass to get polymorphism

Here we go:

```
// define superclass
function  SuperClass(param)
{this.field1=param;
};
// put two methods in the prototype
SuperClass.prototype.method1 = function(x) {return x*x; };
SuperClass.prototype.method2 = function() {console.log("Super method"); };

// define the subclass
function SubClass(param) {
 this.field1=param;  // constructor not inherited
 this.field2=0; // extra field
};
// inheritance chain
SubClass.prototype=Object.create(SuperClass.prototype);
// override method2
SubClass.prototype.method2 = function() {console.log("Sub method"); };
// add new method
SubClass.prototype.method3 = function() {console.log("New method"); };


var ob1=new SuperClass(2); // superclass instance
console.log(ob1.field1); // 2
console.log(ob1.method1(3)); // 9
ob1.method2(); // super method
var ob2=new SubClass(3); // subclass instance
console.log(ob2.field1); // 3
console.log(ob2.method1(4)); // 16
ob2.method2(); // sub method
ob2.method3(); // new method
```

Note we cannot say

```
SubClass.prototype=SuperClass.prototype;
```

since then these would be two references to *the same object*. When we said

```
SubClass.prototype.method2 = …
```

that would have replaced method2 in the single object – in other words replaced the superclass method2 as well. Instead we use Object.create to creat a new object, which has Superclass.prototype as its prototype. This is the heart of the inheritance.

6.3 Encapsulation – private members

A key aspect of OOP is encapsulation. It is important that the data bundled into an object cannot be freely accessed without constraint. The problem with global data as in C was that any function would have access to it. Consequenly any function might accidentally corrupt that global data. So when any new function was added, all existing functions had to be re-tested, in case the new function affected the global data in inappropriate ways.

Can we set up data members which cannot be accessed directly, but only through methods which can validate any changes? Yes:

```
function MyClass()
{
    var x = 2; // private member

// getter and setter:
    this.getX = function getService() {
        return x;
    };

    this.setX = function setX(val) {
        x = val;
    };
}

var obj = new MyClass();
console.log(obj.x);               // undefined - can't access directly
console.log(obj.getX());          // initial 2
obj.setX(3);
console.log(obj.getX());          // now its 3
var obj2 = new MyClass();         // new instance
console.log(obj2.getX());         // initial 2
console.log(obj.getX());          // still 3
```

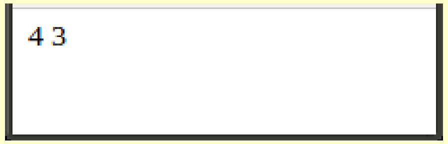
7 Functional programming

7.1 First-class and higher order functions

A first-class function is a function defined in a programming language which can be treated as an object – not simply as a process or code unit. We have already seen JavaScript doing this.

First order functions take *data values* as arguments. Higher order functions take *functions* as arguments. For example:

```
function square(x) {
   return x * x;
}
function inc(x) {
   return x + 1;
}

var fObject = function f1(f2, x) {
   return f2(x);
};

document.writeln(fObject(square,2));

document.writeln(fObject(inc,2));
```

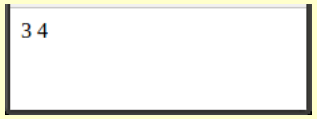


```
4 3
```

So fObject takes two arguments. The first is a function and the second is some data. What fObject does is to apply the first argument to the second. So fObject is a higher order function.

7.2 Pure functions

```
var adder=0;
var fObject = function f1(x) {
   adder++;
   return x+adder;
};
document.writeln(fObject(2));
document.writeln(fObject(2));
```



```
3 4
```

So fObject returns its argument plus something. But it also has a side-effect -namely it increments the adder, which is a global value. As a consequence, fObject does different things each time we invoke it.

This means it is *not* a pure function. A pure function has no lasting side-effects. It does not change the state of the system when it is invoked. Consequently it always returns the same result when used with the same argument.

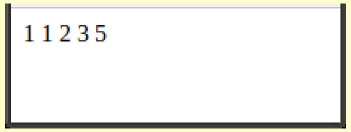
What's so good about that? We always know what a pure function will do. It does not matter what has already happened. Consequently it is easy to test – while functions which are not pure are basically impossible. They might work 99 times and fail on the 100th. And for program proof of correctness, impure functions are impossible.

We can avoid impure functions by ensuring by making sure no global data is changed.

7.3 Recursion

Such as:

```
var fObject = function fib(n) {
   if (n==0 || n==1) return 1;
   else return fib(n-1)+fib(n-2);
};

for (var n=0; n<5; n++)
document.writeln(fObject(n));
```



```
1 1 2 3 5
```

Recursion is widely used in functional programming.

While this is the classic example of recursion, it is also the classic case for recursion being inefficient. The same value is re-calculated several times.

For example, suppose we want fib(20)

so we find fib(19) and fib(18)

fib(19) asks for fib(18) and fib(17). But we've already found fib(18)

fib(18) asks for fib(17) and fib(16). But fib(19) also found fib(17).

We fix the inefficiency by using memoization. When asked to calculate a value, we first look in a map to see if we have already worked it out. If so, we use that. Else we work it out, and add it to the map.

A JavaScript object is a ready-made map:

```
var fibMap = {}; // empty map

var fObject = function fib(n) {
    if (fibMap[n] == undefined)
    {
        var result;
        console.log("Calculating fib of "+n);
        if (n == 0 || n == 1)
            result = 1;
        else
            result = fib(n - 1) + fib(n - 2);
        fibMap[n] = result;
        return result;
    }
    else
        return fibMap[n];
};

for (var n = 0; n <7; n++)
```

```
    console.log(fObject(n));
```
This outputs:

```
Calculating fib of 0
1
Calculating fib of 1
1
Calculating fib of 2
2
Calculating fib of 3
3
Calculating fib of 4
5
Calculating fib of 5
8
Calculating fib of 6
13
```

Each value is only calculated once.

On the other hand, invoking fObject might change fibMap, so it is not a pure function.

7.4 Eager and lazy evaluation

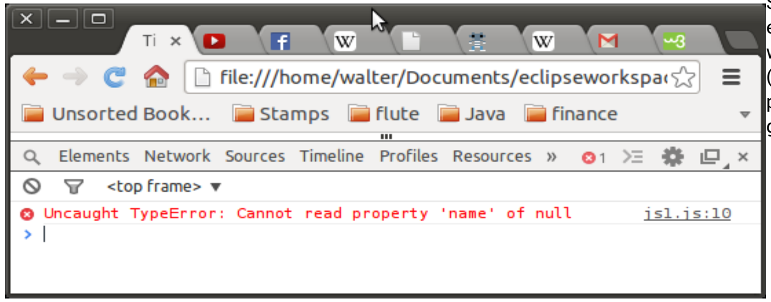
Suppose we have this:

```
function getLength(n) {
   return n.length;
};

var x=null;
document.writeln(getlength(new Array(1,2,3,x.name)));
```
So the function getLength returns the length of the array you pass it. And we pass it an array with 4 elements. What will happen? There are two tactics

1. Count the array elements. Do not evaulate them (yet), because we don't need to. This is lazy evaluation – not working things out until you must.

2. Evaluate the items in the array, then count them. This is eager evaluation – work things out as soon as you can, even if you later don't actually need the values. This is eager evaluation.

This is what actually happens:



```
Uncaught TypeError: Cannot read property 'name' of null          js1.js:10
```

So its doing eager evaluation – trying to work out x.name (and failing) before it passes the array to getLength.

7.5 Strict functional programming

When a computer is running a program, different inputs produce different outputs. This is what a function does – its maps a domain (of inputs) to a range (of outputs). So we can think of a program as a function. In strict functional programming, we actually write the program as a function – or a set of interacting functions.

In strict functional programming, we only have pure functions, with no global state. We have *no variables,* only parameters. And since we have no variables, we have no assignments. We look at some examples.

7.6 Maximum element of a list

The conventional style would use the 'biggest so far' idea, initialising this to the first, and iterating through the rest of the list :

```
function biggest(list)
{
    var maxSoFar=list[0];
    for (var i=0; i<list.length; i++)
        if (list[i]>maxSoFar)
            maxSoFar=list[i];
    return maxSoFar;
}

console.log(biggest([5,3,8,1,4,3,2])); // 8
```
However this uses variables and assignments. How to do it without them? Recursively. In pseudo-code:

if the list is only 1 element long, that's the largest

otherwise, compare the first to the largest in the rest of the list, and return whichever is larger.

When we do the *largest in the rest of the list* the list is smaller. Eventually we get down to a single element, and that's it.

Here is a version like this:

```
function biggest(list)
{
    if (list.length==1) return list[0];
    if (list[0]> biggest(list.slice(1))) return list[0];
    else return biggest(list.slice(1));
}

console.log(biggest([5,3,8,1,4,3,2])); // 8
```

We have no variables and no assignments.

Unfortunately it is very inefficient. If the first item is not the smallest, then we have worked out biggest(list.slice(1))) twice. We can't assign it to a variable to remember it. But we can do this:

```
function bigger(a,b)
{
    if (a>b) return a; else return b;
}
function biggest(list)
{
    if (list.length==1) return list[0];
    return bigger(list[0], biggest(list.slice(1)));
}

console.log(biggest([5,3,8,1,4,3,2])); // 8
```

7.7 Total of a list

We could have a running total variable, initialised to 0, and adding in each item in the list. Or, we can do it without variables:

```
function total(list)
{
    if (list.length==1) return list[0];
    return list[0]+total(list.slice(1));
}
console.log(total([1,0,2,1])); // 4
```

7.8 Reverse a string

```
function reverse(list)
{
    if (list.length==1) return list[0];
    return reverse(list.slice(1))+list[0];
}
```

```
console.log(reverse("sdrawkcab")); // backwards
```

7.9 Remove an element from a list

The idea is

if the first element is the item to remove, return the rest of the list

else return the first followed by remove the item from the rest

```
function removeFirst(list, item)
{
    if (list[0] == item)
        return list.slice(1);
    else
        return [list[0]].concat(removeFirst(list.slice(1), item));
}

console.log(removeFirst([1, 2, 3, 4], 2)); // [1,3,4]
```

7.10 Sort a list

The idea is

if the list is 1 item long, return it

else return the biggest, followed by the sorted list with the biggest removed

```
 function sort(list)
{
if (list.length==1) return list[0];
return [biggest(list)].concat(sort(removeFirst( list, biggest(list))));
}

console.log(sort([1, 2, 3, 4])); // [4,3,2,1]
```

8 The DOM

The DOM, the document object model, is the API through which JavaScript code can reference the web page displayed in the browser.

8.1 The global object

When JavaScript execution starts, a 'global object' already exists, with a set of properties, some of which are objects (some of which are functions). The keyword 'this' refers to that object.

Compare this code:

```
            'use strict';
            var object = {}; // make an empty object
            object.x = 3;     // create a property x, and assign a value
            console.log(object.x); // get 3
```

with this:

```
            var x = 4;
            console.log(this.x); // 4
```

It looks like x is a 'stand-alone' value, but in fact it has been added as a property of the global object, and we can refer to it as this.x. But we can also omit the this, and just say 'x'.

8.2 The Browser Object Model

If JavaScript is executing in a browser, this global object is called the Browser Object Model, the BOM. There is no standard for this – so different browsers will put different properties in this object. One of the properties is document, and this is an object representing the web page being displayed. This is structured according to a standard, known as the DOM, the Document Object Model.

However there are properties of the BOM which are common to all browsers: window (of the browser), screen, location (url the browser is displaying), history (browsing history), navigator (the browser itself), timing, and cookies, but the details may vary. Go to http://www.w3schools.com/ to see the properties of these objects, and which browsers support what. For example:

```
            window.open("http://google.com"); // open a new browser window - if pop-ups
are not blocked
            console.log(navigator.userAgent); // userAgent sent to server in http request
= name of browser
            console.log(screen.width); // total screen width
            history.back(); // load previous page
            location.assign("http://waltermilner.com"); // open url in current browser
tab
```

8.3 The DOM

Unlike the BOM, here is a w3c standard of the DOM – which is defined in a language-agnostic way, so that it can be used from Python, for example, as well as from JavaScript.

When the browser loads a web page, it constructs a tree representing the document's structure and content. The DOM is the standard for this tree representation. For example, the console output from this page:

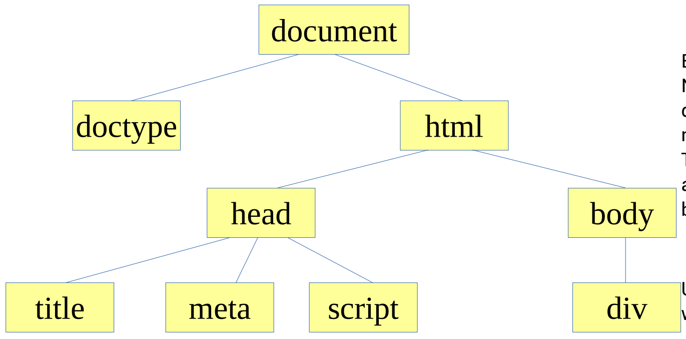
```
<!DOCTYPE html>
<html>
      <head>
            <title>Title</title>
            <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
            <script>
         function traverse(node)
         {

             var children = node.childNodes;
             if (children.length===0)
               return;
             console.log("Elements below : "+node.nodeName);

             for(var i=0; i<children.length; i++) {
                  console.log(children[i].nodeName+" : type "+ children[i].nodeType);
               }
          console.log("-------------------------");
          for(var i=0; i<children.length; i++) {
                  traverse(children[i]);
               }
        }
            </script>
      </head>
      <body onload="traverse(document)">
            <div>Hello world</div>
      </body>
</html>
```

is

```
Elements below : #document
html : type 10
```

```
HTML : type 1
-------------------------
Elements below : HTML
HEAD : type 1
#text : type 3
BODY : type 1
-------------------------
Elements below : HEAD
#text : type 3
TITLE : type 1
#text : type 3
META : type 1
#text : type 3
SCRIPT : type 1
#text : type 3
-------------------------
Elements below : TITLE
#text : type 3
-------------------------
Elements below : SCRIPT
#text : type 3
-------------------------
Elements below : BODY
#text : type 3
DIV : type 1
#text : type 3
-------------------------
Elements below : DIV
#text : type 3
-------------------------
```

So this does a recursive traversal of the tree. The tree is:



Each item in the tree is a node. Nodes have a type. Type 10 is a doctype. Type 1 is an 'element node'. Type 3 is a 'text node'. These text nodes are the returns and any other white space between elements in the html.

Usually we are not interested white space, and

```
var children = node.children;
```

rather than childNodes will skip text elements.

8.4 DOM levels and IDL

The DOM is maintained by w3c, and the levels (1 to 3) are versions. The levels are split into sections. The current version is level 3, split into Load and Save, Validation ,and Core (2004). In addition there is the 2008 Element Traversal Specification (which includes the .children to skip text nodes). DOM 4 is currently being developed. WhatWG.org calls DOM4 the 'Living Standard'.

Because the Dom is accessible from any programming language, it is defined in terms of IDL – Interface Definition Language – which states the properties and events of each type. The JavaScript equuivalent is pretty obvious.

8.5 Accessing nodes

Rather than traversing the whole document tree, we might want to access certain nodes. We can do that in several ways:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
      function access()
      {
       var node1 = document.getElementById("div one");
       console.log(node1.innerHTML); // Hello world
       var allNodes =document.documentElement.children;
       // document.documentElement is html part of the document, missing
                  the doctype node
       // .children gives an array of the childnodes of the html
       // allNode[0] is the head, and allNodes[1] is the body
       console.log(allNodes[1].innerHTML); // html of the body element
       var s = document.querySelector(".special");
       // get first node matching the CSS selector = class special
       console.log(s.innerHTML); // Special class
     }
        </script>
    </head>
    <body onload="access()">
        <div id="div one">Hello world</div>
        <div id="div two">Two</div>
        <div class="special">Special class</div>
    </body>
</html>
```

8.6 Inserting and deleting nodes

For example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
      function insert()
      {
        var three = document.getElementById("three");
        three.insertAdjacentHTML('beforebegin', '<div>Two</div>');

     }
        </script>
    </head>
    <body onload="insert()">
        <div id="one">One</div>
        <div id="three">Three</div>
    </body>
</html>
```

and

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
      function remove()
      {
        var three = document.getElementById("three");
        three.remove();
     }
        </script>
    </head>
    <body onload="remove()">
        <div id="one">One</div>
        <div id="three">Three</div>
    </body>
</html>
```

8.7 Changing attributes

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
      function change()
      {
        var three = document.getElementById("three");
        three.setAttribute("align", "center");
     }
        </script>
    </head>
    <body onload="change()">
        <div id="one">One</div>
        <div id="three">Three</div>
    </body>
</html>
```

The attribute could be a style.

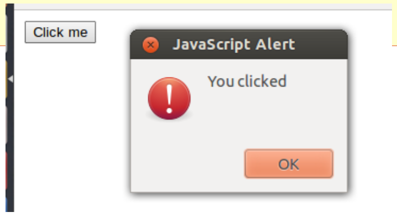
To change to a different style sheet, use `document.selectedStyleSheetSet`

8.8 Events

A event is something that happens, often due to user actions. Each html element has a set of events which can happen to it, and an attribute of an event-handler (a JavaScript function call or other code) can be set for it. We have already seen the use of the body onload event.

For buttons the onclick event is usually handled:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
          function clickHandler()
          {
              alert("You clicked");
          }
        </script>
    </head>
    <body >
    <button onclick="clickHandler()">Click me</button>
    </body>
</html>
```

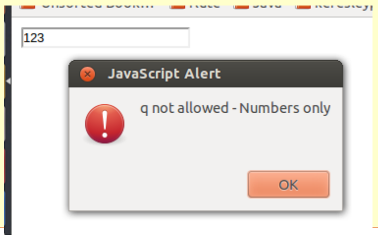


You clicked

For textfield input, 3 events may happen when a key is hit – keydown, keyup and keypress. So if you type capital A, you would have keydown SHIFT, keydown 'a', keyup 'a', keyup SHIFT, and keypress 'A'. So usually keypress is the one to handle..

Suppose we want to allow only digits to be typed into a text field:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
    function keyHandler(event)
    {
    var keyCode=event.keyCode;
    if (keyCode<48 || keyCode>58)
     {
     alert(String.fromCharCode(keyCode)+" not allowed - Numbers only");
          event.preventDefault();
     }
   }
        </script>
    </head>
    <body >
    <input type="text"
onkeypress="keyHandler(event)"></input>
    </body>
</html>
```

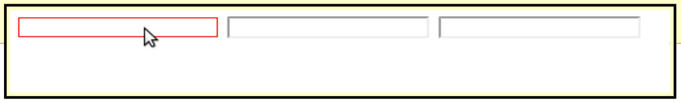

123

JavaScript Alert

q not allowed - Numbers only

For mouse events, we can code mouseover, mouseout and mouseclick. For example, so change border color on mouse over a text field:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Title</title>
        <meta http-equiv="Content-Type" content="text/html;charset=utf-8" >
        <script>
          function over(id)
          {
             var node=document.getElementById(id);
             node.setAttribute("style", "border: thin red solid;");
          }
          function out(id)
          {
          var node=document.getElementById(id);
          node.setAttribute("style", "border: thin black solid;");
          }
        </script>
    </head>
    <body >
    <input type="text" id="1" onmouseover="over(1)" onmouseout="out(1)"></input>
    <input type="text" id="2" onmouseover="over(2)" onmouseout="out(2)"></input>
    <input type="text" id="3" onmouseover="over(3)" onmouseout="out(3)"></input>
    </body>
</html>
```



8.9 JSON

JSON is an acronym for JavaScript Object Notation. JavaScript values are either primitives or objects. JSON is a simple way to represent them all, as Strings.

There are just two methods. JSON.parse takes a string and converts it to an object. JSON.stringify does the reverse - object to string. For example:

```
var str = '{ "a" : "10", "b" : "20" }'; // string form of an object
var obj = JSON.parse(str); // make the object
for (var p in obj) // go through the properties of the object
  console.log(p + " : " + obj[p]); // a:10 b:20
var str2 = JSON.stringify(obj);
console.log(str2); // {"a":"10","b":"20"}
```

Note how we are pairing up the two ' and ' and the various " and " in

```
'{ "a" : "10", "b" : "20" }'
```

JSON provides a simple way for sending object data over the net in a simple text format, without using a binary format..

8.10 AJAX

When the user interacts with a web page (say selecting an item in a drop-down list) we might need the web page contents to change. This may require communication with a web server (for example, to fetch data from a database). Then the server might send an altered web page.

For example, this might be a flight booking site. When the user selects a destination airport, we need to query a database on a remote server, and display the results.

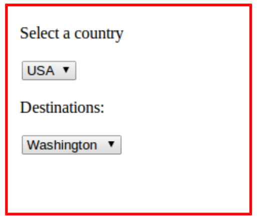
Fetching and displaying an *entire web page* may well be slow, and wasteful, if only a single element needs to be altered. This is the motivation for using Ajax.

The idea is:

1. In the web page, we have an event listener which calls a JavaScript function – such as selecting an item from a drop-down list.

2. The JavaScript function gets the data of what the user has done, and sends an XMLHttpRequest() to the server. This is sent by http, and will run some script on the server. The server will send data back – which could be in XML format, JSON or anything else. The request specifies what should be done when the response is received.
3. The request is asynchronous. In other words, the web page just sits there doing nothing (apparently), until the server response is received. When it is, the code specified in the request is executed. This usually alters a node in the DOM to display something new.

So we want to have this:



Whe the user selects a country, we need to fill the destinations drop down list with relevant airports. This data would come from a very large database, which is frequently changing, so needs to be held in a server-side database, not in the web page. But we do not want to reload the entire web page in response.

The web page body is:

```
    <body >
        <p>Select a country</p>
        <select id = "country" onchange="process()">
            <option value="India">India</option>
            <option value="USA">USA</option>
            <option value="UK">UK</option>
        </select>
        <p>Destinations:</p>
        <select id="result">
        </select>
    </body>
```

In reality the 'country' list would have been filled from a database. When the user makes a selectio from this, the function 'process' is called:

```
            function process() // responds to option change select
            {
                var node = document.getElementById("country");
                // get option
                var str = node.options[node.selectedIndex].value;
                makeAjaxCall(str);
            }
```

so this just gets the item selected, then calls the function which makes the Ajax call:

```
            function makeAjaxCall(str)
            {
                var req = new XMLHttpRequest();
                req.open('get', 'ajax.php?val=' + str);

                // Track the request, as a call-back function
                req.onreadystatechange = function() {
                    if (req.readyState === 4) { // when done
                        if (req.status === 200) { // is OK
                            display(req.responseText); // Handle response
                        } else {
                            alert('Error: ' + req.status); // An error
                        }
                    }
                };

                // Send the request
                req.send(null);
            }
```

When the call is made, it calls the url ajax.php?val=USA or whatever the user selects. In other words we call script ajax.php, using the GET method, and pass the value selected in the query string. The script ajax.php is on the same server as the web page.

Before actuall sending it, we set up the callback function invoked when the request state changes (onreadystatechange). So we do not need a loop listening for when we get a response (underlying interpreter code does this). The function (if (req.readyState === 4)...) will be called when the response is received. This calls display:

```
            function display(text) // display Ajax response
            {
                var node = document.getElementById("result");
                // remove any current options
                var itemCount = node.length;
                for (var item = 0; item < itemCount; item++)
                    node.remove(0);
                // Ajax response is csv like Heathrow,Manchester,Luton
                // split on comma
                var options = text.split(",");
                // add each to result select element
                for (var p = 0; p < options.length; p++)
                {
                    var option = document.createElement("option");
                    option.text = options[p];
                    node.add(option);
                }
            }
```

The script is just:

```
<?php

// Set the content type
header('Content-Type: text/plain');

$str=$_GET["val"];
if ($str=="USA")
{
    echo("New York, Los Angeles, Washington");
}
if ($str=="India")
{
    echo("Delhi, Mombai, Kolkotta");
}
if ($str=="UK")
{
    echo("London, Manchester, Luton");
}

?>
```

In reality this would query a database, but the Ajax functionality is indifferent – the script just gets an input and sends back a response. The script is php – it could have been any language the server can handle. The response is simple text, structured as a comma separated list of values. It could have been XML, or JSON, or anything else, if we have matching JavaScript code to handle whatever format comes back.

9 Data structures

To illustrate how Javascript can implement core Computer Science ideas, we show how we can code a linked list, and then a stack backed by the linked list.

9.1 A linked list

We start be defining a node suitable for use in double-linked lists. Each node has pointers to the next and the previous node, and a field for the data. The fact that variables have no type enables us to have a node which will hold any type.

We also define two methods – one to link this node to another, and one to output this node and any subsequent ones:

```
            // node constructor
            function Node(data) {
                this.data = data;
                this.next = null;
                this.previous = null;
                this.link = link;
                this.print = print;

                function link(anotherNode) {
                    this.next = anotherNode;
                    anotherNode.previous = this;
                }

                function print()
                {
                    console.log(this.data);
                    if (this.next !== null)
                        this.next.print();
                }
            }
```

and we can use this like

```
    var node1 = new Node(23);
    var node2 = new Node(34);
    node1.link(node2);
    node1.print(); // 23 34
```

Then we can use this to code a linked list. A list just has one field – a reference to the head node. We also define methods to append a new node at the end of the list, and print the whole list:

```
function LinkedList() {
                this.head = null;
                this.append = append;
                this.print = print;

                function append(val)
                {
                    var newNode = new Node(val);
                    // append to empty list?
                    if (this.head === null)
                    {
                        this.head = newNode;
                        return;
                    }
                    // else find the last node
                    var where = this.head;
                    while (where.next !== null)
                        where = where.next;
                    where.next = newNode;
                    newNode.previous = where;
                }



                function print()
                {
                    var where = this.head;
                    var str="List: ";
                    if (where === null)
                    {
                        console.log("Empty list");
                        return;
                    }
                    while (where !== null)
                    {
                        str+=where.data + " ";
                        where = where.next;
                    }
                    console.log(str);
                }

            }
```

which we could use as:

```
  var node1 = new Node(23);
  var node2 = new Node(34);

  var list = new LinkedList();
  list.append(node1);
  list.append(node2);
  list.print(); 23 24
```

9.2 Stack

Then a stack could be:

```
function Stack()
{
    this.data = new LinkedList();
    this.push = push;
    this.pop = pop;

    function push(value)
    {
        this.data.append(value);
    }

    function pop()
    {
        // throw exception if empty
        if (this.data.head === null)
        {
            throw "Stack underflow";
        }// else find the last node
        var where = this.data.head;
        while (where.next !== null)
            where = where.next;
        // remove it - is it the only node?
        if (where.previous === null)
            this.data.head = null; // now empty list
        else
            where.previous.next = null;
        // and return value
        return where.data;
    }
}
```

used as:

```
var stack = new Stack();
stack.push(1);
stack.push(2);
stack.push(3);
console.log(stack.pop()); // 3
console.log(stack.pop()); // 2
console.log(stack.pop()); // 1
console.log(stack.pop()); // stack underflow
```