

Standardised by W3C

Standardised by ECMA

**HTML**

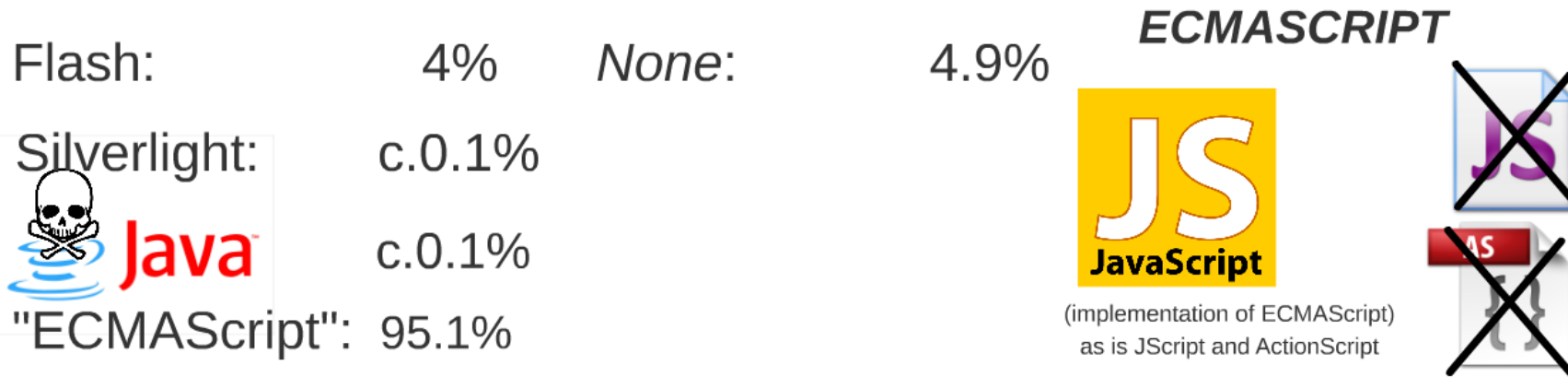
**CSS**

**JS**



DOM

# Client Side Processing



Source: W3Techs.com ~ January 2019

JavaScript attempts to work with the web page. There's a console; but the way in which the effects of JavaScript are felt by the user are the ways in which it interacts with the web page itself.

All the processing is done on the user's computer. This is outsourcing of computation from a design point of view

# Overlap of W3C and ECMA is DOM

The overlap is in the Document Object Model

All browsers must parse HTML and create a tree like structure out of it.

For instance Chrome and Opera use Blink while Firefox uses Gecko and Internet Explorer, Trident.

But if you want to edit this structure after the fact? Some consistent model with which to work with is necessary.

# DOM Standardisation

DOM Level 0 (Created by Netscape at same time as JavaScript) to be used directly by the language for use primarily with html forms and images.

Microsoft created its own DOM Level 0 (and flavour of JavaScript, namely JScript) as the two companies competed through the 90s.

W3C eventually helped create a standardised DOM Level 1. Later versions, with extended functionality (particularly in relation to XML) were subsequently released. We are currently on DOM Level 4.

# Implications

A standardised DOM means you have a universally accepted tree structure for scripting to work with: so if you wish to get an array of every link on a web page, or insert additional tags in an existing web page, there's a guaranteed structure with which to work.

# What will we cover?

JavaScript is huge.

JavaScript has several popular frameworks

- What ways is JavaScript similar to what you've seen before?
- What ways is JavaScript different from what you've seen before?
- What is the purpose of JavaScript?
- Avoiding the bad aspects of JavaScript

# What won't we cover?

Development of full-blown web apps/games

Angular or React

AJAX (in practice)

JavaScript OOP

Intermediary languages (CoffeeScript etc.)

Creating custom HTML elements

Generic programming stuff

# JavaScript is easy in all the wrong ways

Want to leave out a semicolon?

Sure

Want to forget about scope?

Okay

Mash together different types?

Er... is it okay if it makes  
no sense?

The principle of encouraging development...  
even if that development was awful

Is JavaScript a bad language?

No. Like all languages, you'd only use it in  
certain circumstances.



# When to use JavaScript?

**Data visualisation**

**Web development**

**Full stack development**

You can do **everything** in JavaScript.


- Dynamic parts of a website
  - A website can re-write itself using JavaScript
- Some visual aspects

# What do you already know?

- Variables
- Scope, parentheses, statements end with a semicolon...
- Loops, logical statements, switch, *and so on*

unless you are a  
Python head

```
if (flagIsTrue){  
    for (i = 0; i < 10; i++) {  
        j+=i;  
    }  
}  
else flagIsTrue = 1;
```

**C / C++**  
 **JavaScript**

- Functions
- Objects

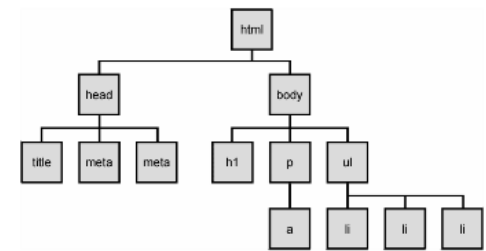
**no pointers**

# Working with the DOM

One of the main purpose of JavaScript is to work with the Document Object Model API

This means that a JavaScript script can access any node on the DOM tree.

What is a node? HTML element

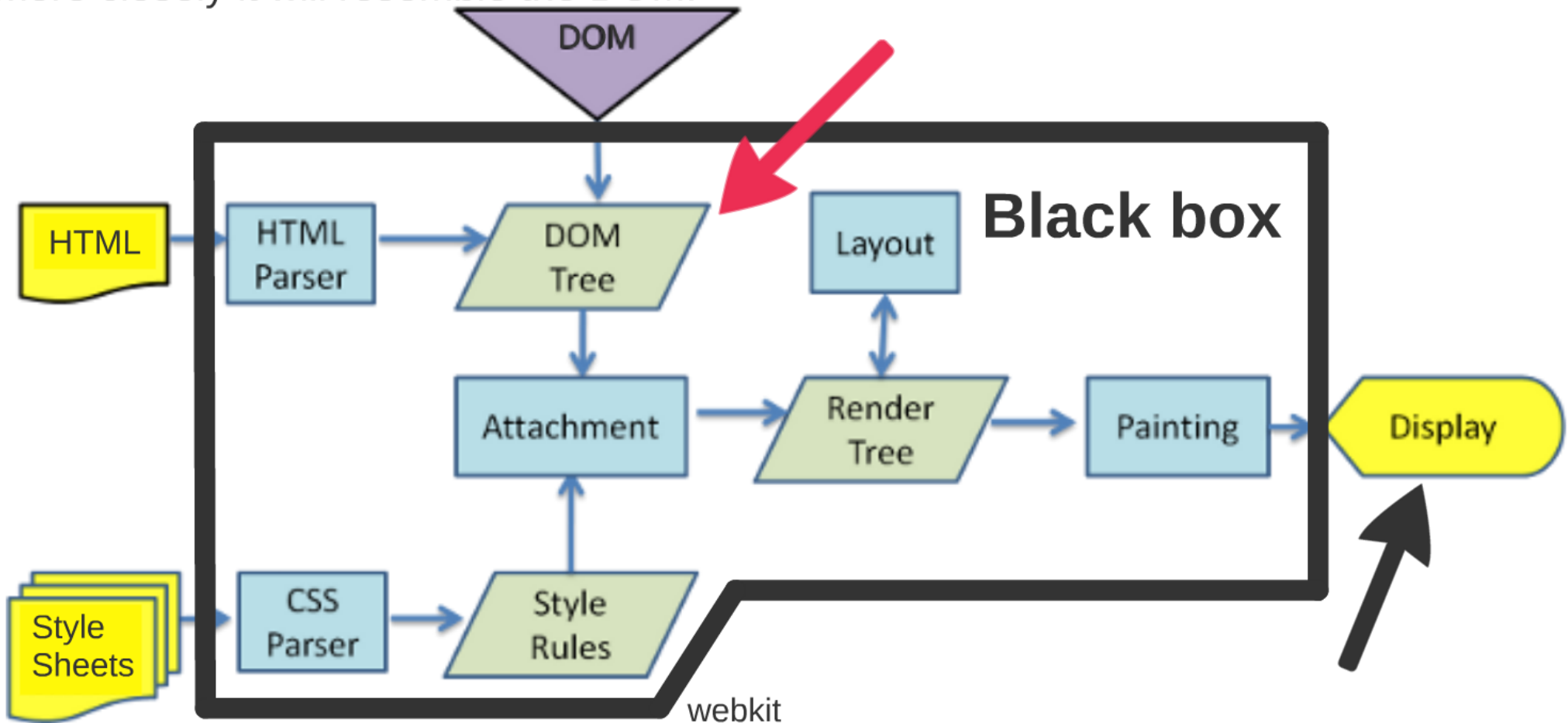


Possible to use the information of a node.

Possible to overwrite the information in a node!

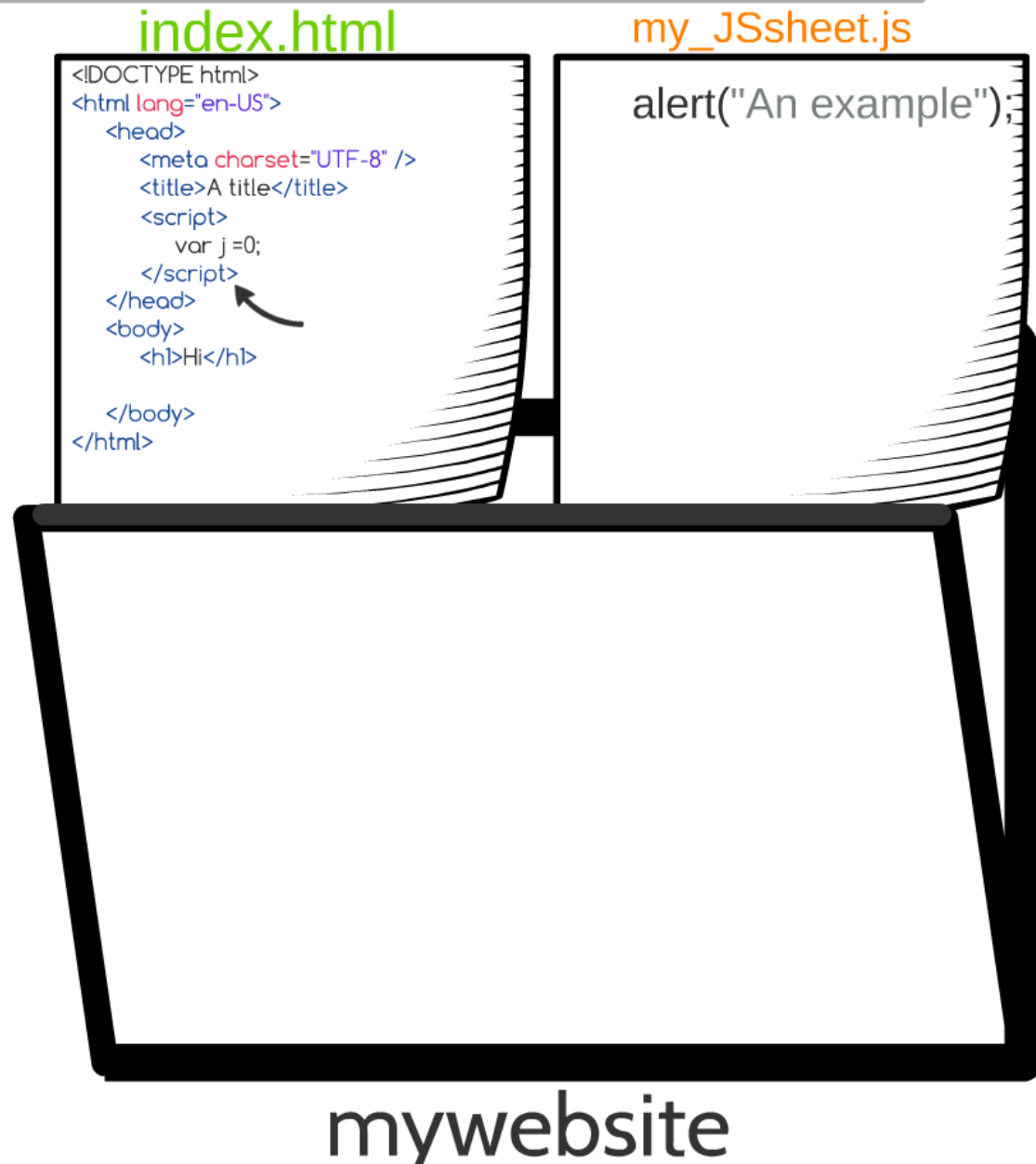
# What does a browser do?

Browser engines are able to parse HTML (along with other things like CSS and JavaScript). Browsers read HTML code and from it construct the DOM. Afterwards JavaScript can manipulate the page through the DOM. The better the HTML code, the more closely it will resemble the DOM.



# 3 "levels" of JavaScript

- external script
- "inline"  
(within script tags)
- inline  
(inside HTML elements)



# Inline JavaScript

JavaScript embedded within HTML elements is generally to be avoided.

```
<a href="#" onclick="alert('Hi')">Click Me</a>
```



This will technically work but

- Difficult to read.
- Heavier HTML code.
- Lack of caching.
- Poor accessibility.
- Difficult code maintenance.

# Getting your system running

Want to get your code running and doing stuff?

Make a HTML page.

HTML element: `<script> </script>`

Sometimes given attribute: `type="text/javascript"`

This is okay

*Sometimes given attribute: `language="JavaScript"`*


*Not okay*

Your standard code can go within script tags. No boilerplate it required.

The code is interpreted and run by the browser.

The console is found easily using "inspect element".

# Hello world



dynamic

```
<script>  
  var t = 5;  
  console.log("Hello world" + t);  
</script>
```

*String concatenation*

Hello world5

t (above) is a double (if you are interested)  
All numbers in JavaScript are doubles.  
In general you don't have to worry about type.



# Back to the DOM

As discussed already, the DOM consists of nodes.

Inserting, deleting, accessing, and editing nodes are all possible via JavaScript.

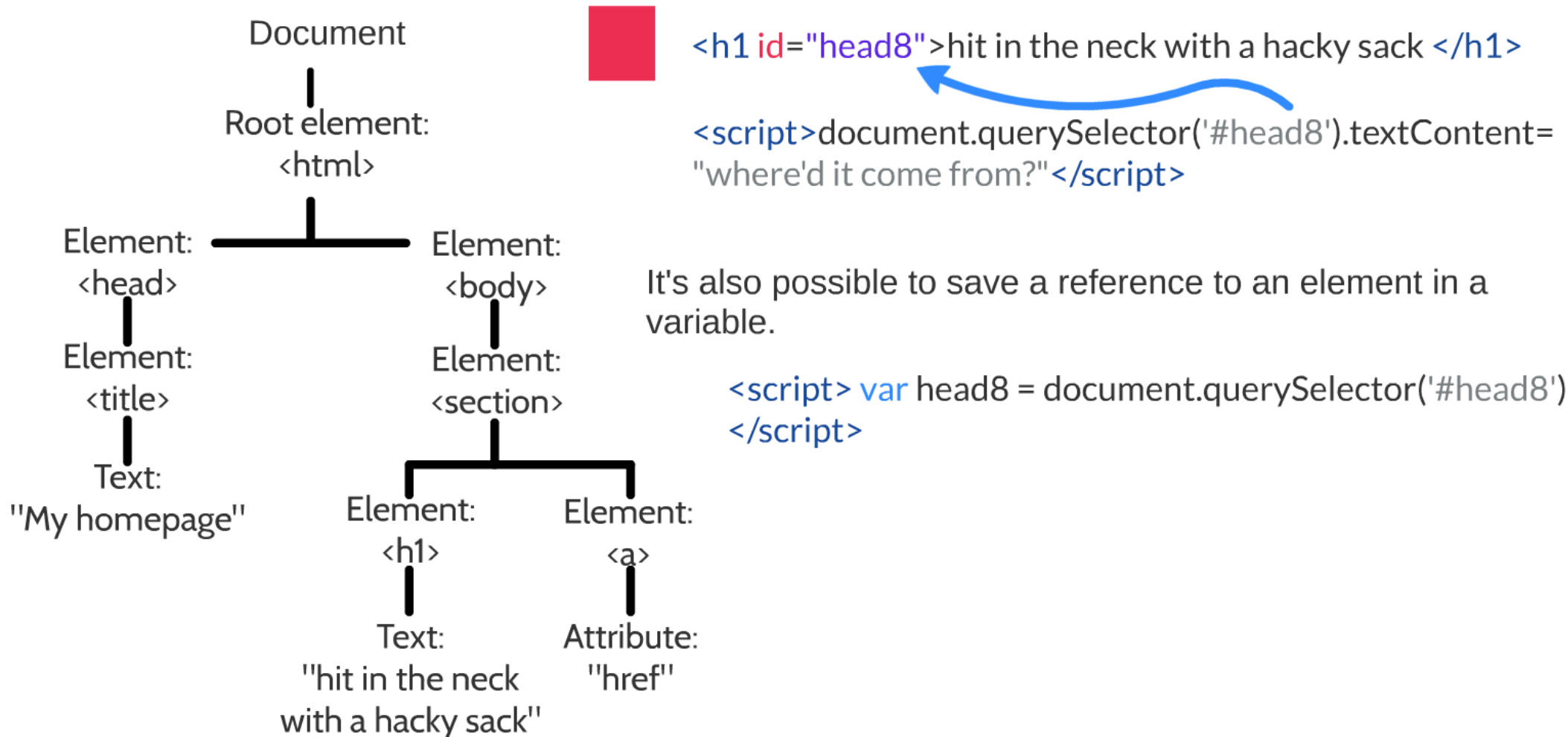
JavaScript can't access DOMs outside its own window (or tab) in the browser.

There are lots and lots of methods and properties associated with the DOM, including some experimental ones.

We'll be looking at some of the more basic ones

# Back to the DOM

Valid text and attribute nodes are attached to html elements on the DOM.



# Loading the script


JavaScript is parsed and executed as it is encountered. Traditionally no other action can take place as this is occurring, including the parsing of HTML and the construction of the DOM.

This means that a JavaScript script can refer to parts of the page that it does not yet know about.

It will fail, as the script will assume that the element being searched for does not exist. One solution is to put any JavaScript code at the bottom of the page.

Alternatively you can test to ensure the page has been fully loaded before execution using `window.onload`

# Loading the script




```
<script>document.querySelector('#head8');  
// huh? </script>
```

```
<h1 id="head8">hit in the neck with a hacky  
sack</h1>
```

This code won't work because the script is attempting to access a HTML element that it does not yet know exists.

# Loading the script



```
window.onload = function() {  
  
    // create a couple of elements  
  
    var para = document.createElement("blockquote");  
    var sum_text = document.createTextNode("Never  
gonna give you up");  
    para.appendChild(sum_text);  
    document.body.appendChild(para);  
}
```

This code will only run when the window (i.e. web page) has fully loaded.

# noscript element

The `<noscript></noscript>` element has a simple purpose. It displays any HTML contained **within** it **if** scripting is disabled.

People may have scripting disabled for increased security or to save resources.

Consider usability. Also, always have a fallback in case JavaScript fails.

# Making variables

Naturally you will want to save and use values, and clearly you will need to use variables to be able to do this.

There are four ways to declare variables

**var**      **const**      **let**      [nothing]

These are **not** to do with the type of the variable, but instead, its *scope*.

Type, like in Python, is guessed by the interpreter. There is no compiler type checking. It is **loosely typed**.

# Hoisting && Scope

Declarations are moved to the top of the current scope by the JavaScript interpreter; meaning the top of the current function or script. All functions and variables are hoisted.

This means that functions can be called before they are declared

**JavaScript does not have block level scope.**

Nested blocks are always able to access variables in higher levels!

Reason for introduction of 'let' into ECMA Script in 2015.



# Hoisting && Scope



```
stupid();
```

```
function stupid(){  
  function whatareyouactuallyserious(){  
    myFish.splice(2, 0, "MAGICMOONFISHY");  
  }  
  whatareyouactuallyserious();  
}
```

```
var myFish = ['cod', 'red herring', 'kipper', 'salmon'];  
console.log(myFish);
```

```
(5) ["cod", "red herring", "MAGICMOONFISHY",  
     "kipper", "salmon"]
```

# var vs let

Since the beginning of JavaScript variables could be declared with the **var** keyword

More recent concerns about some of the scoping issues I mentioned in earlier slides prompted the introduction of **let** keyword in ECMAScript 6

**var**

No block scope

hoisted

Can be redeclared

**let**

Block scope

hoisted, but suffers from the 'temporal dead zone'

Cannot be redeclared

# True and false

In an attempt to make JavaScript easier to learn, consequently the normal equality operator is a bit broken.

```
console.log(1 == '1');  
console.log(1 == [1]);
```

true  
true

An empty string is false,  
but an empty array is true.

```
var z="false";  
if(z)  
    console.log(z);
```

false

Never use `==` or `!=`

Always use `===` or `!==`

These sibling operators are the same,  
but do not perform type coercion

# NaN

NaN is only thing that will not work with ===

NaN is !== NaN

NaN means "Not a number" and is returned whenever a math function fails.

  
isNaN(Math.sqrt(-1)) true

Equality comparisons with NaN will **always** yield false. This can be problematic.

**isNaN()** function can be used to test if something is a NaN!