# Client Side Web Development

## Week 2

## Intro to JavaScript

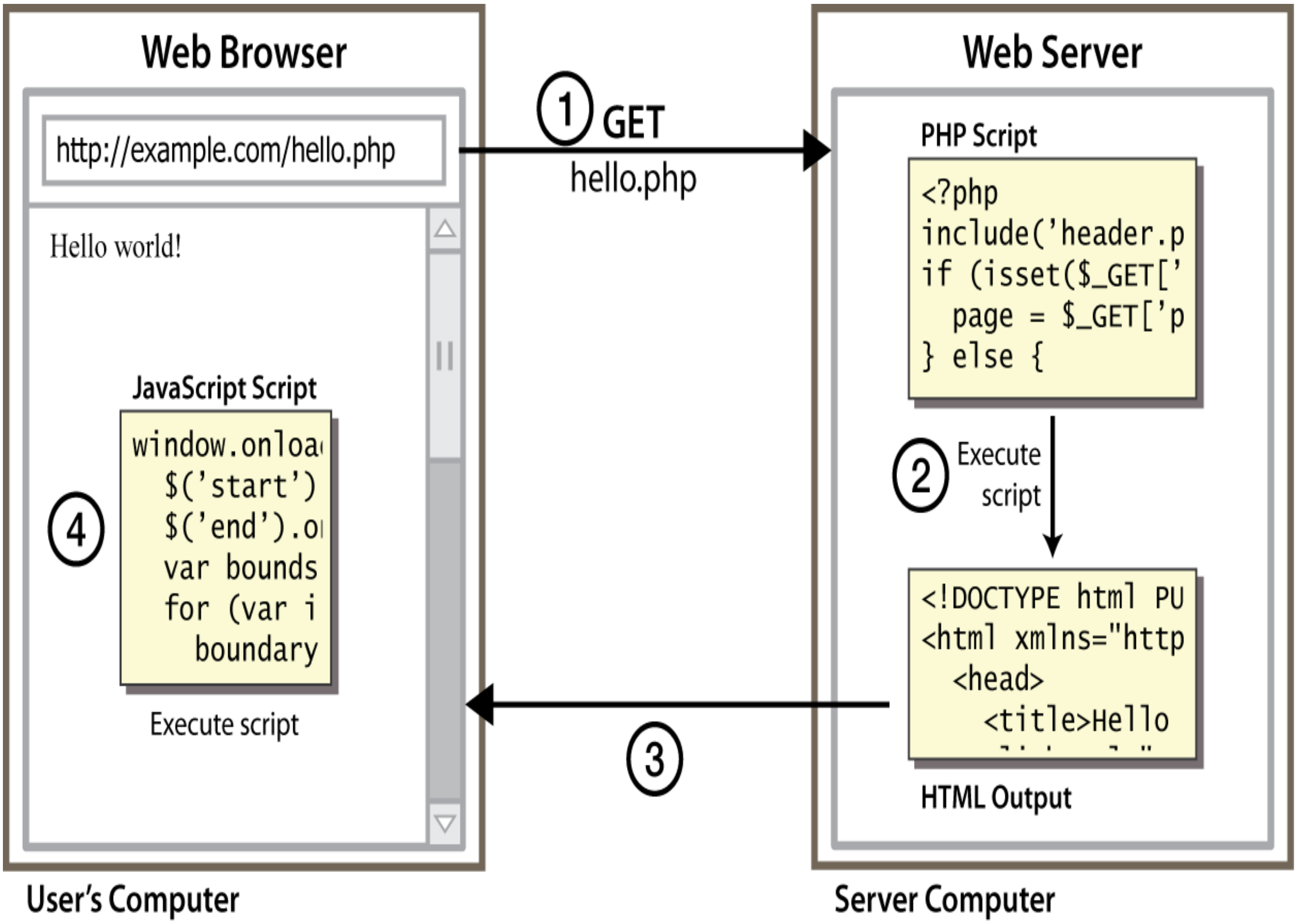
# Topics Covered

- Writing simple JavaScript programs.
- Using input and output statements
- Arithmetic operators.
- Decision-making statements.
- Relational and equality operators.
- Making and calling functions.
- Basic intro to arrays and objects.

CSWD

# Client Side Scripting

**The fundamental nature of the Web (a client-server) architecture transmitted over HTTP) hasn't changed.**



**Client-side scripting is always evolving – it's growing simpler, more nimble and easier to use. As a result, sites are faster, more efficient, and less work is left up to the server.**

# Why use client-side programming?

**Backend languages like PHP already can be used to create dynamic web pages.**

**Why do we use client-side scripting?**

- **client-side scripting (JavaScript) benefits:**
  - **usability: scripts are embedded within and interact with the HTML of your site, selecting elements of it, then manipulating those elements to provide a rich-interactive experience for your users**
    - **can modify a page without having to post back to the server (faster UI)**
    - **can interact with a cascading style sheet (CSS) file that styles the ways the page looks**
  - **efficiency: can make small, quick changes to page without waiting for server**
    - **less 'stress' on the server**
  - **event-driven: can respond to user actions like clicks and key presses**

# Client-side programming and frameworks

**Languages are almost always used in the context of their frameworks which make quick work of complicated tasks with libraries of pre-packaged, shareable code.**

**You'll have used one or a combination of these when building the front end of your sites.**

- **HTML & CSS**
  - **Core building blocks of any site**
  - **HTML dictates organization and content**
  - **CSS makes up the 'look and feel'**
- **JavaScript**
  - **JavaScript is client-side scripting**
  - **Nearly every site's front end is a combination of HTML, CSS & JS**

# Client-side programming and frameworks cont …

## JavaScript Frameworks.

- Angular.js: an incredibly robust JavaScript framework for data-heavy sites.
- jQuery, jQuery Mobile: fast small JS Object library that streamlines how JavaScript behaves across different browsers
- Node.js: a server-side platform that uses JavaScript, and is changing the way real-time applications can communicate with the server for faster response times and a more seamless user experience
- Bootstrap: a mobile-first framework that uses HTML, CSS and JavaScript to facilitate rapid responsive app development
- React: for user interface design
- Express, Backbone, Ember, Meteor, Ionic, Vue and many more
  - ➢ All worth checking out to some degree or other and some of you might already have your own favourite!

# Client-side programming and frameworks cont …

## JavaScript Frameworks.

- **Typescript: A compile-to-JavaScript language that is a superset of JavaScript, created by Microsoft**
- **VBScript & Jscript are Microsoft from-end scripting languages that run on the ASP.NET framework.**
    - ➢ **Jscript is their reverse-engineered version of JavaScript**

- AJAX ( Asynchronous JavaScript + XML)
    - ➢ **allows specific parts of a site to be updated without a full-page refresh by asynchronously connecting to the server/database/service and pulling JSON or XML based chunks of data**
    - ➢ **NB AJAX is not a framework but a collection of technologies**

# What is Javascript?

- a lightweight programming language ("scripting language")
  - used to make web pages interactive
  - insert dynamic text into HTML (eg user name)
  - react to events (eg page load user click)
  - get information about a user's computer (eg browser type)
  - perform calculations on user's computer (eg form validation)

- So it looks like it's necessary to embrace JavaScript as a first class development language in its' own right.
- This means learning the language in some depth, utilizing available libraries, and adopting mature development techniques.

Client-side scripting enhances functionality and appearance

# What is Javascript?

- a web standard (but not supported identically by all browsers)
- NOT related to Java other than by similar name and some syntactic similarities
- interpreted NOT compiled
- more relaxed syntax and rules than strongly-typed languages
  - fewer and "looser" data types
  - variables don't need to be declared
  - errors often go silent (few exceptions)
- key construct is the function rather than the class
  - first-class functions are used in many situations
- contained within a web page and integrates with its' HTML/CSS content

# 1 Overview of JavaScript

- Originally developed by Netscape by Brendan Eich, as LiveScript

- Became a joint venture of Netscape and Sun in 1995, renamed JavaScript

- Now standardized by the European Computer Manufacturers Association as ECMA-262 (also ISO 16262) – ECMAScript – ES6

- This material covers *client-side* JavaScript

- We'll call collections of JavaScript code *scripts*, not programs

- JavaScript and Java are only related through syntax – *they are different languages!*

- JavaScript is dynamically typed

- JavaScript's support for objects is very different to other programming languages eg Java, C#, PHP

# 1.1 Overview of JavaScript (continued)

- User interactions via forms are straightforward
    - ➢ jQuery can also make this easier

- The Document Object Model makes it possible to support dynamic HTML documents with JavaScript

```
var id = document.getElementById("hw");
id.innerHTML = "Hello World!";
```

- Much of what we will do with JavaScript is based on responding to specific events - event-driven

```
<button onclick="sayHi();">Click me!</button>
```

- JavaScript is an interpreted language rather than a compiled language like C#

# 1.2 Object Orientation and JavaScript

- JavaScript is NOT an object-oriented programming language

- Does not support class-based inheritance
    - ➢ Cannot support polymorphism

- Has prototype-based inheritance, which is different from other OO languages

- JavaScript objects are collections of *properties*, which are like the members of classes in Java and C#

- JavaScript has primitives for simple types

- The root object in JavaScript is `Object`
    - ➢ all objects are derived from `Object`

- All JavaScript objects are accessed through references

# 1.3 General Syntactic Characteristics

- All JavaScript sample scripts will be embedded in HTML documents

- Either directly, as in

```
<script type = "text/javaScript">
-- JavaScript script –
</script>
```

- Or indirectly, as a file specified in the `src` attribute of `<script>`, as in

```
<script type = "text/javaScript"
        src = "myScript.js">
</script>
```

optional
in HTML5

Code Conventions/Styling
http://javascript.crockford.com/code.html

*Language Basics*:

- *Identifier form*: begin with a letter or underscore, followed by any number of letters, underscores, and digits
- Case sensitive
- 25 reserved words, plus future reserved words
- Comments: both `//` and `/* … */`

# 1.3 General Syntactic Characteristics (continued)

- Scripts are usually hidden from browsers that do not include JavaScript interpreters by putting them in special comments

```
<!--
-- JavaScript script –
//-->
```

➢ Also hides it from HTML validators

- Semicolons can be a problem
    - ➢ They are "*somewhat*" optional

- *Problem*: When the end of the line can be the end of a statement  JavaScript puts a semicolon there – *automatic semi-colon insertion*

Q$^{2}$. http://aws.gcu.ac.uk/misc/qubed3/qubed.php?id=02101701

# 1.4 Primitives, Operations, & Expressions

- All primitive values have one of the five primitive types: *Number*, *String*, *Boolean*, *undefined*, or *null*

# 1.4 Primitives, Operations, & Expressions (continued)

- Number, String, and Boolean have wrapper objects (`Number`, `String`, and `Boolean`)

- In the cases of Number and String, primitive values and objects are coerced back and forth so that primitive values can be treated essentially as if they were objects

- Numeric literals – just like Java

- All numeric values are stored in double-precision floating point

- String literals are delimited by either `'` or `"`

  - ➢ Can include escape sequences (e.g., `\t`)
  - ➢ All String literals are primitive values

# 1.4 Primitives, Operations, & Expressions (continued)

- Boolean values are `true` and `false`

- The only null value is `null`

- The only undefined value is `undefined`

  ➢ Don't confuse null with undefined.

  ➢ undefined is used by JavaScript to tell you that something is missing

    ❖ e.g. variables with no initial value

  ➢ null is provided so you can determine when a value is expected but not available yet.

- JavaScript is dynamically typed – any variable can be used for anything (primitive value or reference to any object)

- The interpreter determines the type of a particular occurrence of a variable

- Variables can be either implicitly or explicitly declared

```
var sum = 0,
    today = "Wednesday",
    flag = false;
```

# 1.4 Primitives, Operations, & Expressions (continued)

- Numeric operators include `++`, `--`, `+`, `-`, `*`, `/`, `%`

- All operations are in double precision
  - ➢Same precedence and associativity as Java

- The `Math` Object provides `floor`, `round`, `max`, `min` and trig functions, etc.
  e.g., `Math.cos(x)`

- The `Number` Object
  - ➢ Some useful properties:
    `MAX_VALUE`, `MIN_VALUE`, `NaN`,
    `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `PI`
    - ❖ e.g., `Number.MAX_VALUE`
  - ➢ An arithmetic operation that creates overflow returns `NaN`
    - ❖ `NaN` is not == to any number, not even itself
      - ❑ `NaN == NaN` is false (How crazy is this?)
    - ❖ Test for it with `isNaN(x)`
  - ➢ `Number` object has the method, `toString`

# 1.4 Primitives, Operations, & Expressions (continued)

- ***String concatenation/catenation operator*** **+**

- ***Coercions***
  - ➢ Concatenation coerces numbers to strings
  - ➢ Numeric operators (other than +) coerce strings to numbers (if either operand of + is a string, it is assumed to be concatenation )
  - ➢ Conversions from strings to numbers that do not work return `NaN`

- **Explicit conversions**
  1. Use the `String` and `Number` constructors
  2. Use `toString` method of numbers
  3. Use `parseInt` and `parseFloat` on strings

- ***Sample string properties & methods*:**

  - ➢ `length` e.g., `var len = str1.length;`
  - ➢ `charAt(`position`)` e.g., `str.charAt(3)`
  - ➢ `indexOf(`string`)` e.g., `str.indexOf('B')`
  - ➢ `substring(`from, to`)` e.g., `str.substring(1,3)`
  - ➢ `toLowerCase()` e.g., `str.toLowerCase()`
  - ➢ `charCodeAt(`position`)` e.g., `str.charCodeAt(3)`

# 1.4 Primitives, Operations, & Expressions (continued)

- *The* `typeof` *operator*

  ➢ Returns `"number"`, `"string"`, or `"boolean"` for Number, String, or Boolean, `"undefined"` for undefined, `"function"` for functions, and `"object"` for objects and null

  ➢ `typeof(undefined) != typeof("undefined")`

- *Assignment statements* – just like C++ and Java

- The `Date` Object

  ➢ Create one with the `Date` constructor (no params)

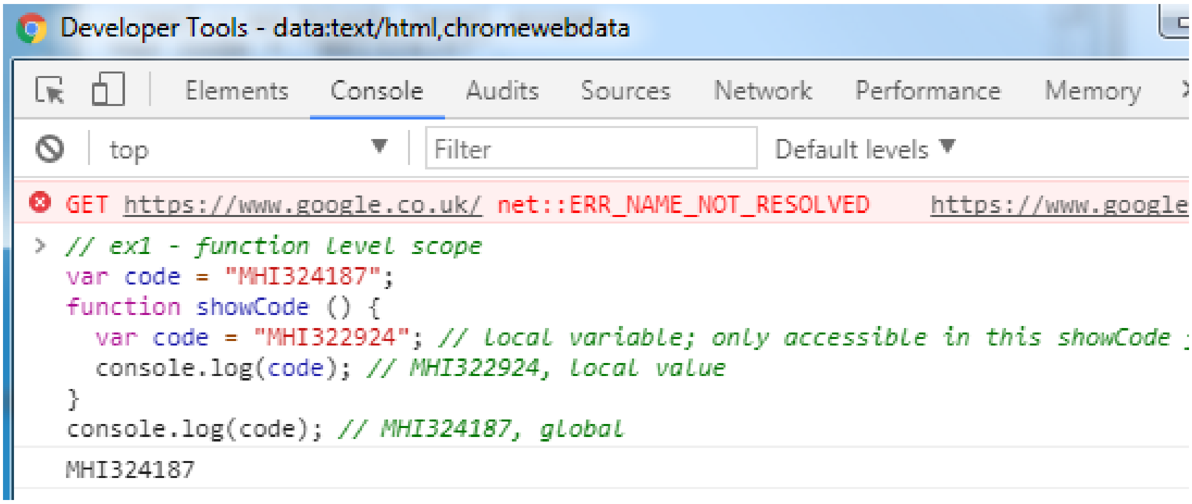
  ➢ Sample local time methods of `Date`:

  `toLocaleString` – returns a string of the date
  `getDate` – returns the day of the month
  `getMonth` – returns the month of the year (0 – 11)
  `getDay` – returns the day of the week (0 – 6)
  `getFullYear` – returns the year
  `getTime` – returns the number of milliseconds since January 1, 1970
  `getHours` – returns the hour (0 – 23)
  `getMinutes` – returns the minutes (0 – 59)
  `getMilliseconds` – returns the millisecond (0 – 999)

# 1.4.1 Scope

- **A variables scope is the context in which the variable exists**
  - **this essentially specifies where in the program a variable will have a value and where that value can be accessed/used**
  - **variables can have local scope or global scope**

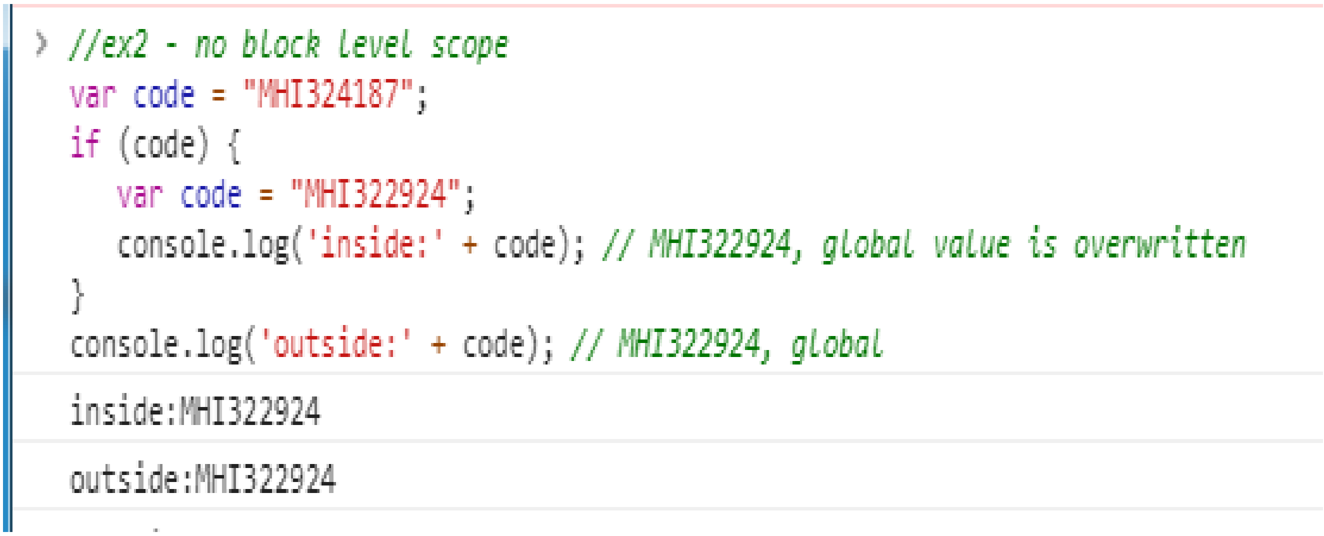
- **Local Variables (Function-level Scope)**
  - **unlike most programming languages, JavaScript does not have block-level scope (variables scoped to surrounding curly brackets) instead, JavaScript has function-level scope**
    - » **variables declared in a function are local variables and are only accessible within the function or by functions inside that function**



```
GET https://www.google.co.uk/ net::ERR_NAME_NOT_RESOLVED     https://www.google
> // ex1 - function level scope
  var code = "MHI324187";
  function showCode () {
    var code = "MHI322924"; // local variable; only accessible in this showCode :
    console.log(code); // MHI322924, local value
  }
  console.log(code); // MHI324187, global
  MHI324187
```

CSWD

# 1.4.1 Scope cont …



```
> //ex2 - no block level scope
  var code = "MHI324187";
  if (code) {
     var code = "MHI322924";
     console.log('inside:' + code); // MHI322924, global value is overwritten
  }
  console.log('outside:' + code); // MHI322924, global
```

```
inside:MHI322924
outside:MHI322924
```

- **All variables declared outside a function are in the global scope**
  - **in the browser, which is what we are concerned with as front-end developers, the global context or scope is the window object**

## Q$^2$. Assuming the following code:

```
if (!("code" in window)) {
   var code = "MHI322924";
   console.log('inside:' + code);
}
console.log('outside:' + code);
```

**What would you expect to see in the console window? Explain your answer.**

# 1.4.1 Scope (continued)

- **Do not pollute the GLOBAL scope**
  - to become proficient at JavaScript you have to avoid creating many variables in the global scope
  - simple example to be avoided

```
//these two variables are in the global scope but they shouldn't be
var name, code;
function fullName() {
   name = "CSWD"; code = "MHI322924";
   console.log('Fullname:' + name + ' ' + code);
}
```

- **Variable HOISTING.**
  - All variable declarations are hoisted (lifted and declared) to the top of the function if declared in a function or the  top of the global context if defined outside of a function
  - N.B. only variable declarations get hoisted not the variable initialization or assignments (when the variable is assigned a value)

# 1.5 Screen Output & Keyboard Input

- The JavaScript model for the HTML document is the `Document` object

- The model for the browser display window is the `Window` object
  - ➢ The `Window` object has two properties, `document` and `window`, which refer to the `Document` and `Window` objects, respectively

- The `Document` object has a method, `write`, which dynamically creates content

  - ➢ The parameter is a string, often catenated from parts, some of which are variables e.g.

```
document.write("Answer: " + result + "<br />");
```

  - ➢ The parameter is sent to the browser, so it can be anything that can appear in an HTML document (`<br/>`, but not `\n`)

- The `Window` object has three methods for creating dialog boxes, `alert`, `confirm`, and `prompt`

# 1.5 Screen Output (continued)

1. `alert("Hello World! \n");`

   - Parameter is plain text, not HTML

   - Opens a dialog box which displays the parameter string and an `OK` button
     - ➢ It waits for the user to press the `OK` button

2. `confirm("Do you want to continue?");`

   - Opens a dialog box and displays the parameter and two buttons, `OK` and `Cancel`

   - Returns a Boolean value, depending on which button was pressed (it waits for one)

3. `prompt("What is your name?", "");`

   - Opens a dialog box and displays its string parameter, along with a text box and two buttons, `OK` and `Cancel`

   - The second parameter is for a default response if the user presses `OK` without typing a response in the text box (waits for `OK`)

`screenOutput.html`

# 1.6 Control Statements

**Similar to C, Java, and C++**

**- Compound statements are delimited by braces, but *compound statements are not blocks***

**- *Control expressions* – three kinds**

**1. *Primitive values***

- **If it is a string, it is true unless it is empty or `"0"`**
- **If it is a number, it is true unless it is zero**

**2. *Relational Expressions***

- ***The usual six*: ==, `!=`, <, >, <=, >=**
- **Operands are coerced if necessary**
- **If one is a string and one is a number, it attempts to convert the string to a number**
- **If one is Boolean and the other is not, the Boolean operand is coerced to a number (1 or 0)**
- ***The unusual two*: === and `!==`**
  **Same as == and `!=`, except that no coercions are done (operands must be identical in type and value)**

# 1.6 Control Statements (continued)

2. *Relational Expressions* (continued)

- Comparisons of references to objects are not useful (addresses are compared, not values)

3. *Compound Expressions*

- The usual operators: `&&`, `||`, and `!`

- The `Boolean` object has a method, `toString`, to allow `Boolean` values to be printed (`true` or `false`)

- If a Boolean object is used in a conditional expression, it is false only if it is `null` or `undefined`

- *Selection Statements*

- The usual `if-elseif-else` (clauses can be either single statements or compound statements)

```
if ( x < y ) max = y;                  // simple
if ( x < y ) { max = y; }              // compound
```

# 1.6 Control Statements (continued)

- *Switch selection statement*

```
switch (control_expression) {
   case value_1:
      // value_1 statements
      break;
   case value_2:
      // value_2 statements
      break;
   …
   [default:
      // default statements]
      break;
}
```

flow-of-control stops executing at this point and leaves the switch

NB break; is optional

- The statements can be either statement sequences or compound statements
- The control expression can be a number, a string, or a Boolean
- Different cases can have values of different types
- default case is the code that executes when none of the other switch options match

# 1.6 Control Statements (cont …)

- *Loop statements: while, for, do ... while*

`while (`control_expression`)` statement or cmpnd

`for (`init`;` control`;` increment`)` statement or cmpnd

- init can have declarations, but the scope of such variables is the whole script

→ SHOW `date.js`

`do`

statement or compound

`while (`control_expression`)`

# 1.7 Object Creation and Modification

- Objects can be created with `new`

- The most basic object is one that uses the `Object` constructor, as in

```
var myObject = new Object();
```

- The new object has no properties of its' own

  ➢ A JavaScript object inherits from Object by default, unless you explicitly create it specifying null as its prototype, as in:

```
var myObject = new Object.create(null);
```

CSWD

# 1.7 Object Creation and Modification

(cont …)

Properties can be added to an object, any time

```
var myDog = new Object();
myDog.name = "Kim";
myDog.breed = "mongrel";
```

2 new properties name & breed

- Objects can be nested, so a property could be itself another object, created with `new`

- Properties can be accessed by dot notation or in array notation, as in

```
var dogName = myDog["name"];
delete myDog.breed;
```

- *Another Loop Statement (an iterator)*

  - `for` `(`identifier `in` object`)` statement or compound

```
for (var prop in myDog){
  console.log('Key:' + prop +
              ' Value:' + myDog[prop]);
}
```

We'll come back to objects later!

# 1.8 Arrays

- Objects with some special functionality

- Array elements can be primitive values or references to other objects

- Length is dynamic - the `length` property stores the length

- Array objects can be created in two ways, with `new`, or by assigning an array literal

```
var myList = new Array(24, "bread", true);
var myList2 = [24, "bread", true];
var myList3 = new Array(24);
```

- The length of an array is the highest subscript to which an element has been assigned, plus 1

```
myList[122] = "bitsy";  // length is 123
```

- Because the `length` property is writeable, you can set it to make the array any length you like, as in

```
myList.length = 150;
```

- Assigning a value to an element that does not exist creates that element

# 1.8 Arrays (continued)

- Array methods:

  - `join` – e.g., `var listStr = list.join(", ");`

  - `reverse`

  - `sort` – e.g., `names.sort();`
    - Coerces elements to strings and puts them in alphabetical order

  - `concat` – e.g., `newList = list.concat(47, 26);`

  - `slice`

```
listPart = list.slice(2, 5); // els. 2, 3, 4
listPart2 = list.slice(2); // els. 2 upwards
```

  - `toString`
    - Coerces elements to strings, if necessary, and catenates them together, separated by commas (exactly like `join(", ")`)

  - `push`, `pop`, `unshift`, and `shift`

**We'll also come back to arrays later!**

# 1.9 Functions

- **All the methods we've just seen are in effect JavaScript functions**
- **They encapsulate a block of code that performs a certain task**
- **these methods save us having to write the code**
  - **but how do we write or own functions?**

```
function function_name([formal_parameters]) {
  // body of function goes here
  return returnValue;   // optional
}
```

**- Return value is the parameter of `return`**

**- If there is no `return`, or if the end of the function is reached, `undefined` is returned**

**- If `return` has no parameter, `undefined` is returned**

```
function speak (msg) {
  console.log(msg);
}
var msg = speak("Hello Class");
console.log( msg );  // what do you see in console?

function say (msg, times) {
  var i = 0, tmpMsg = "";
  for (i = 0; i < times; i++ ) { tmpMsg += msg; }
  console.log( tmpMsg );
  return tmpMsg;
}
var msg = say("Hello Class“, 3);
console.log( msg ); // what do you see in console?
```

# 1.9 Functions

**Functions are objects, so variables that reference them can be treated as other object references**

- **If `fun` is the name of a function,**

```
function fun(){
  //
}
moreFun = fun;
moreFun();    // effectively a call to fun
```

**- Usual to place all function definitions in the head of the HTML document**

**- Remember that all variables that are either implicitly declared or explicitly declared outside functions are global**

- **Variables explicitly declared in a function are local**
- **Parameters are passed by value, but when a reference variable is passed, the semantics are pass-by-reference**

- **There is no type checking of parameters, nor is the number of parameters checked (excess actual parameters are ignored, excess formal parameters are set to `undefined`)**

- **All parameters are sent through a property array, `arguments`, which has the `length` property**

# 1.10 Constructors

- Used to initialize objects, but actually create the properties

```
function car(newMake, newModel, newYear){
   this.make = newMake;
   this.model = newModel;
   this.year = newYear;
}

myCar = new car("Ford", "Vauxhall", "1970");
```

- Can also have method properties

```
 function displayCar() {
   console.log("Make: "+ this.make +
     " Model: "+ this.model +
     " Year: "+ this.year);
}
```

- Now add the following to the constructor:

```
car.display = displayCar;
myCar.display();   // error - no function

myCar.display = displayCar;
myCar.display();   // does exist = ok

All to do with prototype inheritance - try it!
```

**We'll also come back to this later!**

# 1.11 Debugging JavaScript

- **Debugging is not an easy task but thankfully all modern browsers have a built-in JavaScript debugger that allows you to examine variables and set breakpoints to watch the values as the code runs**

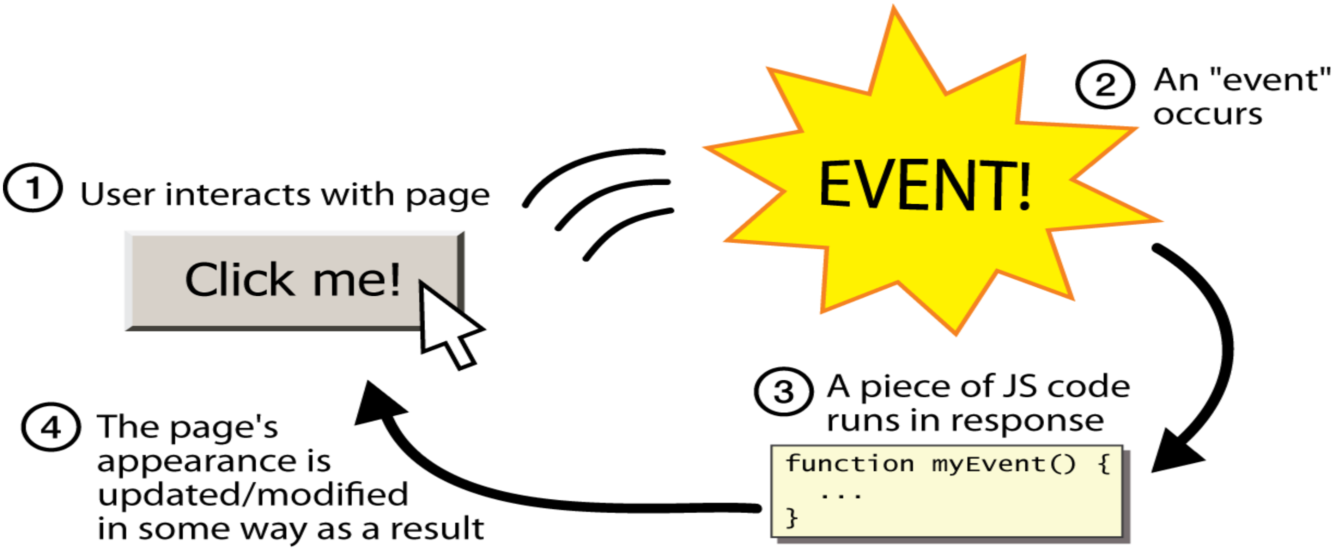
- **Different browsers start the debugger in different ways so check the browser that you're working with**

  - **try F12 or ctrl-shft-I**

  - **A small window appears to display script errors and variable values etc**
    - ➢ **debugger window can be detached from browser window**

  - **Remember to `Clear` the console after using an error message – avoids confusion**

# Event-driven JavaScript example



① User interacts with page

② An "event" occurs

③ A piece of JS code runs in response

```
function myEvent() {
   ...
}
```

④ The page's appearance is updated/modified in some way as a result

**HTML document**

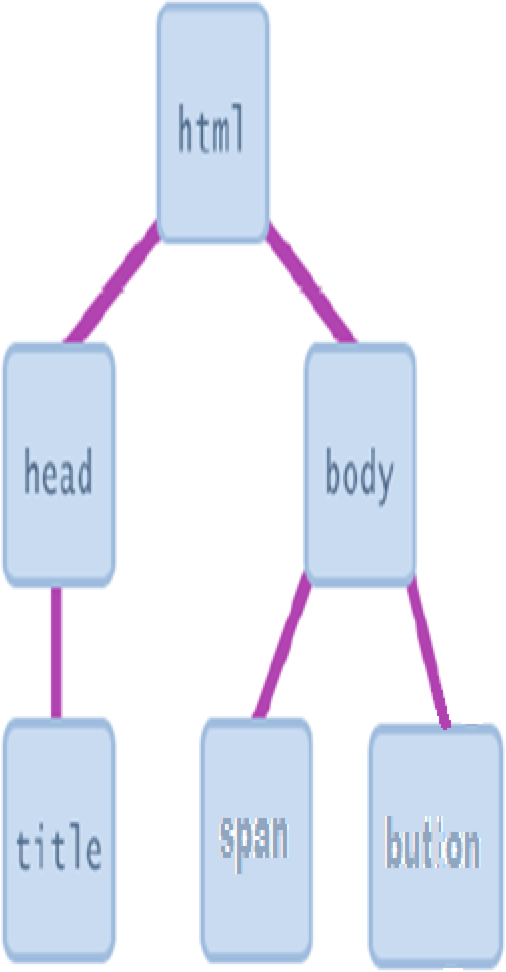
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>button click</title>

    <script src="eventHandler.js"></script>
</head>
<body>
   <span id="output">text will go
here</span>
   <button id="btn1"

      onclick="clickMe();">Click
Me!</button>
</body>
</html>
```

# Document Object Model (DOM)

- much JS code manipulates elements on an HTML page
- we can examine elements' state
  - ➢e.g. see whether a box is checked
- we can change state
  - ➢e.g. insert some new text into a div
- we can change styles
  - ➢e.g. make a paragraph red

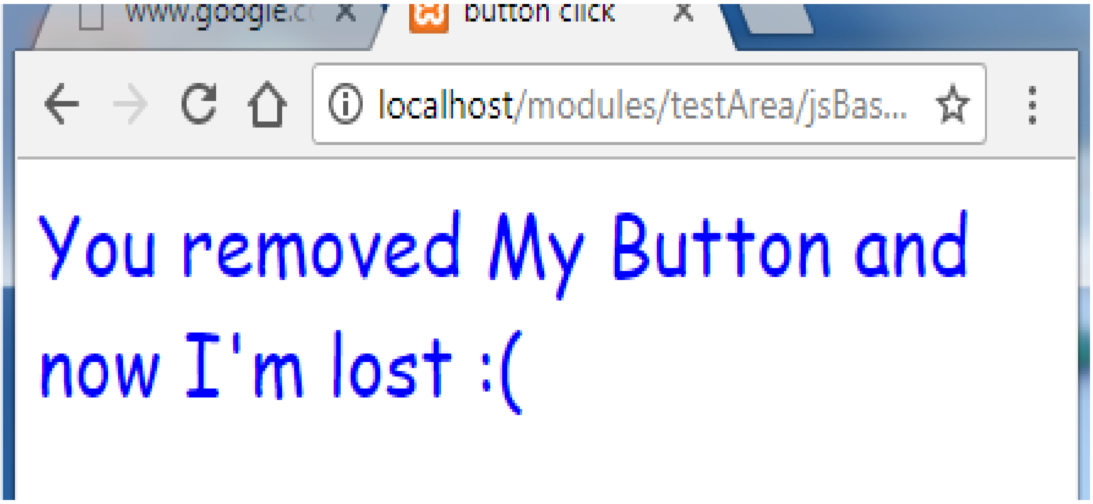


# Event-driven JavaScript example cont ...

```
function clickMe() {
    var span =
document.getElementById("output");
    span.style.fontSize = "22pt";
    span.style.fontFamily = "Comic Sans
MS";
    span.style.color = "green";
    span.innerHTML = "You removed My Button
and now I'm lost :(";
    var btn =
document.getElementById("btn1");
    btn.style.visibility = 'hidden';
}
```

<u>buttonClickHandler.js</u>

**After button clicked:**



You removed My Button and now I'm lost :(

# Summary

- Looked at introduction to basic syntax of JavaScript
  - Sequence
  - Selection
  - Repetition
  - Quick intro to objects, arrays and functions
    - » We’ll come back to these later in more depth

- Next Week

**HTML5**

**Pre class exercise:**

***Find out what graceful degradation is all about and find the name of its buddy ☺. Now compare the two approaches when designing for the web.***