Introduction to JavaScript

Materials

- The information here is dominantly derived from *JavaScript: The Complete Reference* 2nd and 3rd editions
- There are plenty of other good resources online about JavaScript, but there are also quite poor discussions about it filled with bad info – you've been warned
- Tools
  - Browsers (Firefox, IE, Opera, Safari, etc.)
  - Debugging and DOM focused toolbars (Web Developer Toolbar, Firebug, etc.)
  - Text editor that is JS aware – WebStorm, Coda, Dreamweaver, Eclipse, etc.

Intro

- JavaScript is premier client-side scripting language used in Web development
  - Note especially
    - Client side
    - Focus on web development
    - Scripting
    - No limits though! desktop widgets, node,js, etc.
- Part of the client-side 'triangle' consisting of (X)HTML, CSS and of course JavaScript
  - Manipulation of mark-up and style via the *document object model* or DOM

Helloworld

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8">
<title>JavaScript Hello World</title>
</head>
<body>
<h1>First JavaScript</h1>
<hr>
<script>
  document.write("Hello World from JavaScript!");
</script>
</body>
</html>
```

HTML – JavaScript Intermixture

- The interplay between (X)HTML and JavaScript can be tricky at first

```
<script>
// Careful on tag and script intermixture
<strong>
  document.write("Hello World from JavaScript!");
</strong>
</script>
```

- Instead you would do

```
<script>
 document.write("<strong>Hello World from    JavaScript!</strong>");
</script>
```

- or even

```
<strong>
<script>
 document.write("Hello World from JavaScript! ");
</script>
</strong>
```

JavaScript's Silent Failings

- Most browsers will give minimal feedback that a JavaScript failure is occurring
  - Look in the lower left corner of the status bar in IE to double click on the warning icon
  - You may see in Mozilla browsers a status bar message like JavaScript errors occurred or similar
- Make sure you can turn on your browser's error reporting
  - IE (Tools > Internet Options > Advanced)
  - Mozilla (use javascript: URL or (Tools > JavaScript Console)
- A little experiment: Browse the Web with JavaScript error reporting on

Adding JavaScript to HTML Documents

- There are four standard ways to include script in an (X)HTML document:

1. Within the **<script>** element
2. As a linked file via the **src** attribute of the **<script>** element
3. Within an (X)HTML event handler attribute such as **onclick**
4. Via the pseudo-URL javascript: syntax referenced by a link

Note: There may be other approaches but they are non-standard

<script> Tag

- The <script> tag (<script> … </script>) in all major browsers interprets contents as JavaScript unless one of the following occurs:
  - Inclusion of language attribute
    - <script language="VBS"> … </script>
  - Inclusion of type attribute
    - <script type="text/javascript"> … </script>
  - The **type** attribute is W3C recommended, **language** more common and in many ways more useful
  - HTML5 pretty much pushes dropping both, so the likelihood of a multi-script language Web is unlikely
  - However…note that other languages that compile (translate really) to JS is a very real possibility – see https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

Generalized Top-to-Bottom Execution

- You can use as many **<script>** tags as you like in both the **<head>** and **<body>** and they are executed sequentially.

- ```
  <h1>Ready start</h1>
  <script>
      alert("First Script Ran");
  </script>
  <h2>Running...</h2>
  <script>
      alert("Second Script Ran");
  </script>
  <h2>Keep running</h2>
  <script>
      alert("Third Script Ran");
  </script>
  </h1>Stop!</h1>
  ```

- The demo is trivial but should push the point that if you are running code that the blocking, network fetch, and interaction between HTML/CSS/JS engines is kind of important. Guessing at what is going to happen is a dangerous proposition
  ```

Script in <head>, then called in <body>

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8">
<title>JavaScript in the Head</title>
<script>
   function alertTest() {
    alert("Danger! Danger! JavaScript Ahead");
   }
</script>
</head>
<body>
<h2>Script in the Head</h2>
<hr>
<script>
   alertTest();
</script>
</body>
</html>
```

Script Masking and <noscript>

- Script Hiding using HTML and JavaScript comments
  - <script type="text/javascript">
    <!--
    put your JavaScript here
    //-->
    </script>
  - Avoids printing script onscreen in non-script aware browsers, really of limited value given newer browsers and your end goal of code separation
  - However, cost is minor if you must inline script in markup

- **<noscript>** Element
  - Useful to provide alternative rendering in browsers that have script off or don't support script
  - <noscript>
    Either your browser does not support JavaScript or it is currently disabled.
    </noscript>

  - Next example shows a technique to keep non-JavaScript supporting visitors out of your site/app

<noscript> Example

```
<!DOCTYPE html>
<html>
<head>
<meta charset=”UTF-8”>
<title>JavaScript Masked</title>
</head>
<body>
<script>
<!-
   // example code here
 document.write("Congratulations! If you see this you have
   JavaScript.");
//-->
</script>
<noscript>
 <h1 class="errorMsg">JavaScript required</h1>
 <p>Read how to <a href="/errors/noscript.html">rectify this
   problem</a></p>
</noscript>
</body>
</html>
```

Meta-Refresh Trick

- Add a <meta> refresh within a <noscript> in the head to redirect non-JS users right-away to an error page

```
<noscript>
    <meta http-equiv="Refresh" content="0;URL=/errors/noscript.html">
</noscript>
```

- Possible Downsides
  - Won't validate – W3C oversight? Fixed now or soon?
  - Redirects turned off at certain security settings with JS
  - Not bot friendly, but is that a good thing? Depends on if a site or an app you are applying this to. Most apps shouldn't be bot traversed anyway.

Event Handler Attributes

- (X)HTML defines a set of event handler attributes related to JavaScript events such as **onclick**, **onmouseover**, etc. which you can bind JavaScript statements to.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>JavaScript Events</title>
</head>
<body onload="alert('page loaded');">
<form action="#" method="get">
 <div id="formfields">
    <input type="button" value="press me" onclick="alert('You pressed my
    button!');">
 </div>
</form>
<p><a href="http://www.yahoo.com" onmouseover="alert('hi');">Yahoo!</a></p>
</body>
</html>
```

Event Handler Attributes

- Event handler attributes are closely intertwined with the HTML so that has some considerations
  - Good News: You can clearly see what is bound to what element
  - Bad News: You have really intermixed your code into your markup in a deep way. This seems to go against the idea of separation of concerns and may lead to difficultly in maintenance later on

- Also consider that if you are binding things here this way it is not advanced event binding
  - <input type="button" id="myBtn" value="Press Me" onclick="someFunc">
  - Doesn't provide the easy ability to add other things to the element, control the event itself, etc. as compared to using code binding like

    document.getElementById('myBtn').addEventListener('click', someFunc, false);

    Of course all this begs the question of do you really want things being able to easily bind in without you as the developer being aware of them and accounting for it?

JavaScript Pseudo-URLs

- You can also use the JavaScript pseudo-URL to trigger a script statements
- For example

```
<a href="javascript: alert( 'hi' );">Click me</a>
```

- You can also type such a URL directly in the browser's location box, for example

```
javascript:alert(5+9)
```

- Be aware that JavaScript pseudo-URLs do not degrade well in non-JavaScript aware situations and may confuse a user when some links load pages and others cause actions. Use with caution, but useful in "app" specific cases

Linked Scripts

- Like linked style sheets you can store JavaScript code in a separate file and reference it
  - Use a .js file
  - Contains only JavaScript
  - Store these files like images in a common directory in your site (e.g. /scripts)
  - Linked scripts can be cached and “clean up” HTML documents
  - Linked scripts do have problems under the obvious condition of network failure or extreme slowness
    - Unless used excessively which would cause lots of round-trips external script code is the preferred way to separate code from markup
  - Linking to a remote script <script src=“http://somesite.com/somescript.js”></script>
    - Dangerous? That site has performance, security impact on you?

Linked Script Example

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Linked Script</title>
<script src="danger.js"></script>
</head>
<body>
<form action="#" method="get" id="form1">
 <div id="formfields">
   <input type="button" name="button1" id="button1"
          value="press me" onclick="alertTest();">
 </div>
</form>
</body>
</html>
```

Linked Script Example Contd.

In file danger.js you would have simply

```
function alertTest( )
 {
   alert("Danger! Danger!");
 }
```

Improved Linked Script Example

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Linked Script</title>
<script src="danger.js"></script>
</head>
<body>
<form action="#" method="get" id="form1">
 <div id="formfields">
   <input type="button" name="button1" id="button1"
          value="press me" >
 </div>
</form>
<script src="events.js"></script>
</body>
</html>
```

In events.js we have document.getElementById('button1').onclick=function () { alertTest; };

Better Fully Decouple Example

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Linked Script</title>
<script src="danger.js"></script>
</head>
<body>
<form action="#" method="get" id="form1">
 <div id="formfields">
   <input type="button" name="button1" id="button1"
          value="press me“>
 </div>
</form>
</body>
</html>
```

Better Fully Decoupled Example

In danger.js

```
function alertTest( )  {
      alert("Danger! Danger!");
    }

window.onload = function ()  {
 document.getElementById('button1').onclick=function ()    {
        alertTest(); };
}
```

Still not there!

- What happens if another included JS does?
  `window.onload = function() { /* other stuff */ }`
- What about other included code has `function alertTest() { }` or a variable of similar name in their code?
- Big Challenge - JavaScript shares namespace with other included scripts
  - Real possibilities - Name collisions, event collisions, misbehaving or even malicious code

Take 2 Fully Decoupled Example

In danger.js

```
var UCSD = { };   // wrapper object

UCSD.alertTest = function ( )  {
  alert("Danger! Danger!");
}

// employ approach to set listeners or preserve old handlers
window.addEventListener("load", function () {
  document.getElementById('button1').addEventListener("click",
  UCSD.alertTest, true);
  }, true);
```

Multiple <script> Tags

- Commonly developers reference multiple JS files separately

```
<script type="text/javascript" src="lib1.js"></script>
<script type="text/javascript" src="lib2.js"></script>
```

- **It is questionable the value of this practice given that each request adds a network round-trip and that the .js files will share the same namespace**
- **Idea**

```
<script type="text/javascript" src="alllibs.js"></script>
```

  - **Code organization and caching is cited instead but analysis of both claims is specious at best**
- **Mantra: "Code for yourself - prep for delivery"**
  - **Need a build step then. Consider something like Gulp**

Defensive Coding 101

- If our script is going to play nicely on the Web we must be very defensive
  - Don't bash and watch out for being bashed!
- Encapsulate code and assume the worst is a good idea
- Potential Concerns
  - Variable and Function name conflicts
  - Load order and network concerns
  - Catastrophic errors thrown without handling
  - Event rebinding
  - Browser quirks!

Defensive Coding 101

- Variable Collision
  - Code in browser based JavaScript shares the same namespace.
    - If you define a variable say `num` and later on a script goes and does the same their `num` will overwrite yours. The reverse can also happen.
  - You must avoid global variables that may be bashed

```
var num = 5;  // bad idea!
```

  - Stemming

```
var JSREF_num = 5;
```

  - Object Wrapper

```
var JSREF = { };
JSREF.num = 5;
```

Defensive Coding 101

- Immediate Invoke Function Expression
  - (function () {
    // doing some stuff trying to stay non-global
    })();

- Often just used for a function that does something and then “explodes” However, we can use a return or JS weak-pass by references to do more
- Downside – callstack is going to be polluted with anonymous function calls all over (hard to trace)

Defensive Coding 101

```
// Revealing Modules Pattern Example

(function( cse134 ) {
    var prof = "Powell"; // This is private

    cse134.nameOfClass= "CSE 134";  // watch out for class or className
    cse134.sayHi = function() {
        console.log(  " Prof " + prof + " says hi everybody!");
    };
}( window.UCSD = window.UCSD || {} ));

console.log( window.UCSD );           // limit global pollution
console.log( window.UCSD.nameOfClass );     // Public property view
console.log( window.UCSD.sayHi() ); // Public method access private var
console.log( window.UCSD.prof );    // Private variable no access

// or

var UCSD2 = (function () { …
             return stuff;})();
```

Defensive Coding 101

- Event Collision
  - Depending on how events are added it is also possible to overwrite and existing event handler.

```
window.onload = function () {
 /* going to bash an existing one */
}
```

- Safe Loader Code or better yet use addEventListener…except…

```
var JSREF = { };
 JSREF.addLoadEvent =
function(newFunction) {
  var oldFunction = window.onload;
  if  (typeof window.onload != "function") {window.onload = newFunction; }
  else { window.onload = function () {
            if (oldFunction) { oldFunction(); }
                    newFunction();
         };
        }
};
```

History of JavaScript

- JavaScript first introduced in 1995
  - Invented by Netscape
  - Originally called LiveScript
  - Renamed JavaScript when first beta in Netscape 2
  - Not really related to Java
- Microsoft supports clone of JavaScript called JScript
  - First introduced in Internet Explorer 3
- Standards oriented JavaScript called ECMAScript
- The ideas of DHTML and Ajax add even more confusion
- JavaScript used both client and server-side and within browsers and outside browsers
- Some of the bad aspects of JavaScript are really the bad aspects of the environment in which it is used

Preview of JavaScript, (X)HTML, and CSS Link

- JavaScript very much relies on markup and CSS in browsers, in fact it manipulates objects that are created by the **correct** use of tags and style properties

- For example, the **document** object contains objects and collections corresponding to many of the tags in the (X)HTML document.
  - `document.forms[ ], document.images[ ], document.links[ ],` etc.
  - We can always jump directly to the object using something like `document.getElementById( )` under a DOM compliant browser

- If you tags aren't correct then the corresponding JavaScript objects are you are attempting to instantiate will not be created
- Note also that naming is quite important here as well

Simple Example 1 of Interplay

```
<!doctyle HTML>
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Simple DOM 1</title>
<script type="text/javascript">
function showField() {
  alert(document.form1.field1.value);
 }
</script>
</head>
<body>
<form action="#" method="get" id="form1" name="form1">
   <input type="text" name="field1" id="field1">
   <input type="button" name="button1" id="button1"
          value="press me" onclick="showField();">
</form>
</body>
</html>
```

Simple Example 2 of Interplay

```
<html>
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<title>Simple DOM 2</title>
</head>
<body>
<p id="p1" style="color: red">Hello there</p>
<form>
   <input type="button" value="left"
   onclick="document.getElementById('p1').align='left';">
   <input type="button" value="center"
   onclick="document.getElementById('p1').align='center';">
   <input type="button" value="right"
   onclick="document.getElementById('p1').align='right';">
   <br><br>
   <input type="button" value="red"
   onclick="document.getElementById('p1').style.color='red';">
   <input type="button" value="blue"
   onclick="document.getElementById('p1').style.color='blue';">
   <br><br>
   <input type="button" value="Big"
   onclick="document.getElementById('p1').style.fontSize='xx-large';">
   <input type="button" value="Small"
   onclick="document.getElementById('p1').style.fontSize='xx-small';">
</form></body></html>
```

Syntax Overview

Basic Features

- Script Execution order
  - Top to bottom
  - **<head>** before **<body>**
  - Can't forward reference outside a **<script>** tag
- JavaScript is case sensitive
  - HTML is not, XHTML is
  - "Camelback" style document.lastModified
  - Remember **onClick**, **ONCLICK**, **onclick** doesn't count since that is HTML

Basic Features Contd.

- Whitespace
  - Whitespace is generally ignored in JavaScript statements and between JavaScript statements but not always consider
    - x = x + 1 same as x =x + 1
    - s = typeof x; is same as s=typeof x but it not the same as s=typeofx; or s= type of x;
  - Return character can cause havoc
  - Given white space support by JavaScript some developers favor “minification”

Basic Features Contd.

- Statements
  - A script is made up of individual statements
  - JavaScript statements are terminated by returns or semi-colons (;)
  - So `x = x+1;` `alert(x);` same as `x = x+1` `alert(x)`
  - Prefer to use semi-colons because if you reduce returns you run into problems
    `x=x+1 alert(x)` throws an error while `x=x+1;alert(x);` does not.

Blocks

- To group together statements we can create a block using curly braces { }. In some sense this creates one large statement

- Blocks are used with functions as well as larger decision structures like if statements

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

```
if (x > 10)
     {
      x= 0;
        y = 10;
         }
```

Comments and Formatting

- When writing JavaScript you may want to include a comment
  - `/* This is a multiple line style comment */`
  - `// This is a single line comment`
- You also may want to format your script for nice reading or you may want to crunch it for fast download?

Variables

- Variables store data in a program
- The name of a variable should be unique well formed identifier starting with a letter and followed by letters or digits
- Variable names should not contain special characters or white space
- Variable names should be well considered
  - X versus sum
  - Some rules of programming might not follow on the Web?
  - Hungarian notation like sMyName or strMyName and oMyObject or objMyObject or … might be useful given the weak typed nature of JavaScript. We also might prefix for name spacing or to show global.

Variables Contd.

- Define a variable using the var statement
  - `var x;`
- If undefined a variable will be defined on its first use
- Variables can be assigned at declaration time
  - `var x = 5;`
- Commas can be used to define many variables at once
  - `var x, y = 5, z;`

- Under ES6 we get const and let to define values as well. This does require transpiling so for ubiquitous use we may want to avoid for a bit.

Variable Name Fun

- Given the nature of JS we see some interesting name effects
  - `$` is allowed and many use this for a function that is short-hand for the wordy `document.getElementById()`

    ```
    var node = document.getElementById('field1');
    Becomes var node=$('field1');
    ```

  - Environment leads to interesting practices
    - Obfuscation needs may push us to variables like `O1001010011`
    - Speed leads to short variable names
    - Name spacing leads to prefixes `PINT_temp` or as we see later object literals `var PINT = { } PINT.temp;`

Basic Data Types

- Every variable has a data type that indicates what kind of data the variable holds
- Basic data types in JavaScript
  - Strings ("thomas", 'x', "Who are you?")
    - Strings may include special escaped characters
      - 'This isn\'t hard'
    - Strings may contain some formatting characters
      - "Here are some newlines \n\n\n and tabs \t\t\t yes!"
  - Numbers (5, -345, 56.7, -456.45677)
    - Numbers in JavaScript tend not to be complex (e.g. higher math)
  - Booleans (true, false)
- Also consider the values null and undefined as types

Weak Typing

- JavaScript is a weakly type language meaning that the contents of a variable can change from one type to another.
  - Some languages are more strongly type in that you must declare the type of a variable and stick with it.
- Example of dynamic & weak typing a variable initially holding a string can later hold a number

```
x = "hello"; x = 5; x = false;
```

- While weak typing seems beneficial to a programmer it can lead to problems

Type Conversion

- Consider the following example of weak typing in action

```
document.write(4*3);
document.write("<br>");
document.write("5" + 5);
document.write("<br>");
document.write("5" - 3);
document.write("<br>");
document.write(5 * "5");
```

- You may run into significant problems with type conversion between numbers and strings use functions like **parseFloat()** or **parseInt()** to deal with these problems

Dealing with Type

- You can also use the `typeof` operator to figure out type

```
var x = "5";
alert (typeof x);
```

- Be aware that using operators like equality or even `+` may not produce expected results

```
x=5;
y = "5";
alert(x == y)
```

Produces a rather interesting result. We see the inclusion of a type equality operator (`===`) to deal with this

Remember the Prof's law of "weird stuff" – focus on constructs in new languages you learn that are odd to you to understand interesting aspects of the language that may vary from what you know from Java, C/C++, etc.

Composite Types

- JavaScript supports more advanced types made up of a collection of basic types.
- Also called reference types
  - See demo with Numbers and Arrays

- Arrays
  - An ordered set of values grouped together with a single identifier
- Defining arrays
  - `var myArray = [1,5,1968,3];`
  - `var myArray2 = ["Thomas", true, 3, -47];`
  - `var myArray3 = new Array();`
  - `var myArray4 = new Array(10)`

Arrays

- Access arrays by index value
  - `var myArray = new Array(4)`
  - `myArray[3] = "Hello";`
- Arrays in JavaScript are 0 based given
  - `var myArray2 = ["Thomas", true, 3, -47];`
  - myArray2[0] is “Thomas”, myArray[1] is true and so on
  - Given new Array(4) you have an array with an index running from 0 - 3
  - To access an array length you can use *arrayName*`.length`
    - `alert(myArray2.length);`

Objects

- An object is a collection of data types as well as functions in one package

```
var stooge1 = { names : ["Larry","Curly","Moe"] , number : 3}
var stooge2 = { names : ["Larry","Curly","Moe"] ,
                showNumber : function() {alert(this.names.length);}  }
alert(stooge1.names);
stooge2.showNumber();
```

- We see the typical access notation similar to Java using "."

  objectname.propertyname *objectname.method()*

- Underneath everything in JavaScript are objects.
  - We have actually been using these ideas everywhere, for example document.write("hello") says using the **document** object invoke the **write()** method and give it the string "hello" this results in output to the string
  - Even a global variable is simply a property of the window object

    var x = 5; alert(window.x);

Working with Objects

- There are many types of objects in JavaScript
  - Built-in objects (primarily type related)
  - Browser objects (navigator, window, etc.)
  - Document objects (forms, images, etc.)
  - User defined objects
- Given the need to use objects so often shortcuts are employed such as the with statement

```
with (document)
    {
     write("This is easier");
     write("This is even easier");
    }
```

- Be careful with ambiguity here what if we have a function write() defined in the global space? Under strict JS with is not
- We also see the use of the short cut identifier this when objects reference themselves
- User defined objects are often used in JS simply as wrappers to implement a form of namespacing

Regular Expressions

- JavaScript implements regular expressions via the RegExp object
  - Introduced in JavaScript 1.2 and Jscript 3.0
  - Very similar to Perl implementations
  - String methods often use regexes
- Literal syntax patterns surrounded by //

```
var pattern = /dog/;
  /* Any string containing dog would match pattern
  including hot dog, doggy, etc. */
```

We can also create the regexe using the

- RegExp() constructor

```
var pattern = new RegExp("dog");
```

Some REGEXs in JS examples

- To test a pattern use the `test( )` method

```
    var pattern = new RegExp('dog');
    document.write(pattern.toString() +' test dog =
'+ pattern.test('dog') + "<br />");
```

- The `exec()` method is used to test a given pattern and get more information like offset values

```
var myPattern = /http:.*/;
myPattern.exec('http://www.pint.com');
```

- Also used in a number of String methods

```
alert("JavaScript regular expressions are
powerful!".search(/pow.*/i));
var s = "Hello. Regexps are fun.";
s = s.replace(/\./, "!"); // replace first . with a !
alert(s);
```

Expressions and Operators

- Make expressions using operators in JavaScript
- Basic Arithmetic
  - + (addition), - (subtraction/unary negation), / (division), * (multiplication), % (modulus)
- Increment decrement
  - ++ (add one) -- (subtract one)
- Comparison
  - >, <, >=, <= , != (inequality), == (equality), === (type equality)
- Logical
  - && (and) || (or) ! (not)

More Operators

- Bitwise operators (&, |, ^)
  - Not commonly used in JavaScript except maybe cookies?
  - Shift operators (>> right shift, << left shift)
- String Operator
  - + serves both as addition and string concatenation
  - document.write("JavaScript" + " is " + " great! ");
  - You should get familiar with this use of +
- Be aware of operator precedence
  - Use parenthesis liberally to force evaluations
  - var x = 4 + 5 * 8 versus x = (4+5) * 8

Flow Control

- Basic program execution control handled in JavaScript using the **if** statement

- **if** (*expression*)
  true-case

  or

  if (*expression*)
  true-case;
  else
  false-case;

```
if (x > 10)
    alert("x bigger than 10");
  else
    alert("x smaller than 10");
```

More on If Statements

- You can use **{ }** with **if** statements to execute program blocks rather than single statements

```
if (x > 10)
    {
       alert("X is bigger than 10");
       alert("Yes it really is bigger");
     }
```

- Be careful with ;'s and **if** statements

```
if (x > 10);
    alert("I am always run!? ");
```

Switch Statements

- **If** statements can get messy so you might consider using a **switch** statement instead

- **switch** (*condition*)
  {
  case (value) : *statement(s)*
  break;
  …
  default: *statement(s)*;
  }

- The **switch** statement is not supported by very old JavaScript aware browsers (pre-JavaScript 1.2), but today this is not such an important issue

Switch Example

```
var x=3;
switch (x)
{
  case 1: alert('x is 1');
          break;
  case 2: alert('x is 2');
          break;
  case 3: alert('x is 3');
          break;
  case 4: alert('x is 4');
          break;
  default: alert('x is not 1, 2, 3 or 4');
}
```

Loops

- JavaScript supports three types of loops: **while**, **do/while**, and **for**
- Syntax of while:

while(*condition*)
  statement(s)

- Example:

```
var x=0;
while (x < 10)
{
  document.write(x);
  document.write("<br>");
  x = x + 1;
}
document.write("Done");
```

Do Loop

- The difference between loops is often when the loop condition check is made, for example

```
var x=0;
   do
    {
     document.write(x);
     x = x + 1;
   } while (x < 10);
```

- In the case of **do** loops the loop always executes at least once since the check happens at the end of the loop

For Loop

- The most compact loop format is the **for** loop which initializes, checks, and increments/decrements all in a single statement

```
for (var x=0; x < 10; x++)
    {
      document.write(x);
    }
```

- With all loops we need to exercise some care to avoid infinite loops. See example from class
- Careful with the **var** statement above, that is not locally scoped x to the loop, you could use a **let** statement to do that but that should be used with caution as it is new to JS, we'll discuss ES6 and transpiling in class

For/In Loop

- One special form of the for loop is useful with looking at the properties of an object. This is the **for/in** loop.

```
for (var aProp in window)
    {
      document.write(aProp)
      document.write("<br>");
    }
```

- Note: Class demo where the prof showed enumerating built-in objects for browsers and how they varied from Chrome to Firefox.
- Use the hasOwnProperty() with object enumeration to distinguish between own and inherited properties/methods

Loop Control

- We can control the execution of loops with two statements: **break** and **continue**
- **break** jumps out of a loop (one level of braces)
- **continue** returns to the loop increment

```
var x=0;
while (x < 10)
{
   x = x + 1;
   if (x == 3)
     continue;

   document.write("x = "+x);
   if (x == 5)
     break;
}
document.write("Loop done");
```

Input/Output in JavaScript

- Special dialog forms
  - Alert
    - alert("Hey there JavaScript coder! ");
  - Confirm
    - if (confirm('Do you like cheese?')
      alert("Cheese lover");
      else
      alert("Cheese hater");
  - Prompts
    - var theirname = prompt("What's your name? ", "");

Input/Output in JavaScript Contd.

- Writing to the HTML document
  - document.write()
  - document.writeln()

- Writing should be done before or as the document loads.
- In traditional JavaScript the document is static after that, though with the DOM everything is rewritable
- Since we are writing to an (X)HTML document you may write out tags and you will have to consider the white space handling rules of (X)HTML

Eval() is evil?

- var aString = “alert(‘hi’)”;
  eval(aString);

- Direct access to the JS interpreter
- Consider the value now of toString() on various different objects and functions – serialization
- “use strict”;
  - Look into this to keep yourself from using such constructs
- Be aware that other constructs like setTimeout(“code”, 0) are the same really. Construct isn’t the problem really…it’s the use!

Function Basics

- **function** *functionname*(*parameterlist*)
  {
  statement(s)
  }

  where
  - *Functioname* must be well-formed JavaScript identifier
  - *Parameterlist* is a list of JavaScript identifiers separated by commas. The list may also be empty

Function Example 1

Simple function with no parameters

```
function sayHello()
 {
   alert("Hello there");
 }
sayHello();      // invoke the function
```

- *Note: You generally will be unable to call a function before it is defined. This suggests that you should define your functions in the <head> of your (X)HTML document. However, in some JavaScript implementations you can forward reference with the same <script> block.*

Function Example 2: Parameters

```
function sayHello(someName) {
    if ((someName) && (someName.length > 0))
       alert("Hello "+someName+", I'm a function!");
    else alert("Don't be shy. Functions are fun.");
}

  sayHello("George");
  sayHello();

  sayHello(3); // ??? Use typeof
```

Example 3: Multiple Parameters & Return

```
function addThree(arg1, arg2, arg3)
  {
   return (arg1 + arg2 + arg3);
   // watch it weak typing!
}


var x = 5, y = 7, result;
result = addThree(x,y,11);
alert(result);
```

Example 4: Multiple Returns

```
function myMax(arg1, arg2)
   {
      if (arg1 >= arg2)
        return arg1
      else
        return arg2;
     }
```

***Note**: Functions always return some value whether or not a **return** is explicitly provided. Usually it is a value of **undefined**. You should notice this in node or the console*

Parameter Passing

- Primitive Data types are passed by value, in other words a copy of the data is made and given to the function

```
  function fiddle(arg1)
   {
       arg1 = 10;
       document.write("In fiddle arg1 = "+arg1+"<br />");
   }
var x = 5;
document.write("Before function call x = "+x+"<br />");
fiddle(x);
document.write("After function call x = "+x+"<br />");
```

Parameter Passing 2

- Composite types are passed by reference in JS

```
function fiddle(arg1)
   {
        arg1[0] = "changed";
        document.write("In fiddle arg1 = "+arg1+"<br>");
   }

var x = ["first", "second", "third"];
document.write("Before function call x = "+x+"<br>");
fiddle(x);
document.write("After function call x = "+x+"<br>");
```

- Note the reference is “weak” reference as in only change innards of composites not whole thing. If change whole instance the semantics is pass by value.

Global and Local Variables

- A *global variable* is one that is known throughout a document
- A *local variable* is limited to the particular function it is defined in
- All variables defined outside a function are global by default
- Variables within a function defined using a **var** statements are local

Global and Local Example

```
// Define x globally
var x = 5;
function myFunction()
{
  document.write("Entering function<br> x="+x+" <br>");
  document.write("Changing x <br>");
  x = 7;
  document.write("x="+x+"<br> Leaving function<br>");
}
document.write("Starting Script<br>");
document.write("x="+x+"<br>");
myFunction();

document.write("Returning from function<br>");
document.write("x="+x+"<br>");
document.write("Ending Script");
```

Local Variable Example

```
function myFunction()
{
  var y=5;  // define a local variable

  document.write("Within function y="+y);
}

myFunction();
document.write("After function y="+y);
```

***Note**: This example will throw an error, but that's the point. You could use an **if** statement to avoid problems like*

```
if (window.y)
 document.write("After function y="+y);
else
 document.write("Y is undefined");
```

Readability Challenges

- Be careful when you have local and global variables of the same name, you may get an undesirable readability of similar names

```
var x = "As a global I am a string";
function maskDemo()
{
 var x = 5;
 document.write("In function maskDemo x="+x+"<br />");
}

document.write("Before function call x="+x+"<br />");
maskDemo();
document.write("After function call x="+x+"<br />");
```

- Consider using prefix like `g` for global or the stem of the function something is local to
- In general the weak typing and scope rules of JavaScript can hurt readability so you might aim to mitigate that with naming conventions. Hungarian notation like var strMyName, numMyBalance, boolLikeJS; are good ideas given the nature of JavaScript and can assist you in debugging.

Local Functions

```
function testFunction()
    {

     function inner1() {   document.write("testFunction-inner1<br>");   }

     function inner2() {  document.write("testFunction-inner2<br>");    }

      document.write("Entering testFunction<br>");
      inner1();
      inner2();
      document.write("Leaving testFunction<br>");
     }


document.write("About to call testFunction<br>");
testFunction();
document.write("Returned from testFunction<br>");

/* Call inner 1 or inner2 here and error */
inner1();
/* no no no testFunction.inner1() thinking either! */
```

Closures

- In JavaScript a closure is an inner function that retains variables outside of the place it was created. This is going to be fundamental in Ajax. Important slide. Read up on Closures online as well!

```
function whatIsIt () {
   // Local variable that ends up within closure
   var num = 1;
   var showIt = function() { alert(num); }
   num++;
   return showIt;
}

var test = whatIsIt();
test();

alert(test.toString());
/* reflect the source shows the function but not the
environment */
```

Functions as Objects

- Like nearly everything in JS, functions are objects and can be created and accessed as such

```
var sayHello = new Function("alert('Hello
   there');");
sayHello();
```

- This allows us to even reuse functions in an interesting way.

```
var sayHelloAgain = sayHello;
sayHelloAgain();
```

Functions as Objects

- You can also define functions with parameters in this fashion.

```
var sayHello2 = new Function("msg","alert('Hello there
   '+msg);");
sayHello2('Thomas');
```

- The general syntax is

var *functionName* = **new Function**("*argument 1*",..."*argument n*", "*statements for function body*");

Useful Function Features

- As objects you can reference the length of functions, thus find out the number of arguments

```
function myFunction(arg1,arg2,arg3)
  {
    // do something
  }
  alert("Number of parameters for myFunction
  = "+myFunction.length);
```

Arguments and Length

- You can examine not just defined arguments but actual passed parameters

```
function myFunction()
{
 document.write("Number of parameters defined =
   "+myFunction.length+"<br />");
 document.write("Number of parameters passed =
   "+myFunction.arguments.length+"<br />")
 for (i=0;i<arguments.length;i++)
   document.write("Parameter "+i+" = "+myFunction.arguments[i]+"<br
   />")
}
myFunction(33,858,404);
```

Variable Arguments

- Given arguments and length you can write more adaptive functions that take variable arguments

```
function sumAll()
{
  var total=0;

  for (var i=0; i< sumAll.arguments.length; i++)
    total+=sumAll.arguments[i];

  return(total);
}

alert(sumAll(3,5,3,5,3,2,6));
```

Literal and Anonymous Functions

```
function Robot(robotName) {
   this.name = robotName;
   this.sayHi = function () { alert('Hi my name is
'+this.name); };
   this.sayBye = function () { alert('Bye!'); };
   this.sayAnything = function (msg) { alert(this.name+'
says '+msg); };

   //return this; // would that make it more clear?
}

var fred = new Robot("fred");
fred.sayHi();
fred.sayAnything("I don't know what to say");
fred.sayBye();
```

Recursive Functions

- JS supports recursive functions that call themselves
- Everybody's favorite factorial n! = n*(n-1)*(n-2) * … 1

```
function factorial(n)
     {
        if (n == 0)
          return 1;
        else
           return n* factorial(n -1);
      }
alert(factorial(5));
```

Objects in JavaScript

- Objects in JavaScript fall into four groups:
  - Built-in objects (ex. String, Date, Array, etc.)
  - Browser objects (ex. Window, Navigator)
  - Document objects (ex. Document, document.forms[], etc.)
  - User-defined objects

- User defined objects are often not used by novice JavaScript programmers or in the cases where such approaches don't improve things but obviously become quite important as you build larger and larger Web applications
  - Will there be a trade-off to Object use in JS?

Object Creation

- Use the new( ) operator along with the appropriate constrictor function

```
var city = new String();
var city = new String("San Diego");
var city2 = "San Diego";
alert(typeof city);
alert(typeof city2);
```

- Of course we will often be making our own objects

```
var city = new Object();
city.name = "San Diego";
city.sunny = true;
```

Object Destruction

- We don't have to do much as within browser implementation of JavaScript garbage collection cleans up unused object references we may have

```
var myString = new String('Larry Fine');
myString = new String('Moe Howard');
// Larry Fine string eventually garbage collected
```

- Better to hint to the browser you are done with memory though.

```
var city = new Object();
city = null;  // hint to garbage collector
```

Properties

- A *property* is named data contained in an object

```
var myString = new String('hello there');
alert(myString.length);
```

- Properties that are not set return undefined

- Instance properties can be added to a particular object and deleted as well

```
var myString = new String('hi');
myString.simpleExample = true;
alert(myString.simpleExample);
delete myString.simpleExample;
alert(myString.simpleExample);  // undefined
```

Array Style Property Access

- You can also use [ ] to access a property of an object by name

```
var myString = new String('hi');
alert(myString['length']);
myString['simpleExample']=true;
alert(myString['simpleExample']);
delete(myString['simpleExample']);
alert(myString['simpleExample']);
```

- The main uses of this style of syntax is when the property name is a variable or if the property name contains spaces.

Methods

- Object properties that are functions are called *methods*
- You can use the standard . operator as well as the array syntax to access a method but add the ( ) to invoke the method

```
var myString = new String("Hi there");
alert(myString.toUpperCase());
alert(myString['toLowerCase']());
```

- Like properties you can set instance methods for a particular objects
  - `myString.sayWhat = function() {alert('What?')};`
  - `myString.sayWhat();`

Enumerating Objects

- The for-in loop can be used to enumerate object properties (and sometimes methods)

```
for (prop in window.navigator)
 document.write('window.navigator["' + prop + '"] = ' +
  window.navigator[prop] + '<br />');
```

- Some browsers may not enumerate methods and some objects particularly built-ins are not enumeratable - just instances are

```
var myString = new String('hi');
myString.foo = true;
for (prop in myString)
  document.write('myString["' + prop + '"] = ' +
myString[prop] + '<br />');
for (prop in String)
  document.write('String["' + prop + '"] = ' + myString[prop] +
'<br />');  // nothing
```

Objects as Reference Types

- All JavaScript types can be categorized as primitive or reference types
- Reference types include Objects, Arrays, and Functions and since they can be large they do not contain the actual data but references to the place in memory that holds the data

```
var city = new Object();
city.name = 'San Diego';
city2 = city;
city2.name = 'Los Angeles';
alert("city.name = " + city.name); // shows Los Angeles
```

Reference Types Contd.

- Given the way reference types work you will find them useful with functions which usually pass parameters by value in the case of primitive types

```
function fiddle(x)
    {
     // x can be changed if reference not if primitive
    }
```

Comparing Objects

- When comparing primitive types things work as expected, but with objects which are reference types you are comparing a reference to data not the data itself so the comparison may fail

```
var str1 = "abc";
var str2 = "abc";
alert(str1 == str2); // true

str1 = new String("abc");
str2 = new String("abc");
alert(str1 == str2); // false

alert(str1.valueOf() == str2.valueOf() ); // true
```

User-Defined Object Intro

- Generic objects are used to create user-defined data types

```
var robot = new Object();
robot.name = "Zephyr";
robot.model = "Guard";
robot.hasJetPack = true;
alert(robot.name + " " + robot.model);

robot.attack = function () { alert('Zap!'); };
robot.attack();
```

Object Literals

- Object literals have been supported since JS 1.2 and are defined as { name1 : value1, name-n: value-n;}

```
var robot2 = { name: "Arnold",
  model: "Terminator",
  hasJetPack: false,
  attack: function() {alert('Hasta la vista!');}
 };
 alert(robot2.name + " " + robot2.model);
 robot2.attack();
```

Object Literals Contd.

- You can build quite large objects this way if you like

```
var jetpack = true;
var robot3 = { name: null,
               hasJetpack: jetpack,
               model: "Protocol Droid",
               attack: function() { alert("Ahhhh!"); },
               sidekick: {  name: "Spot",
                            model: "Dog",
                            hasJetpack: false,
                            attack: function() { alert("CHOMP!"); }
                         }
             };
robot3.name = "C-3PO";
alert(robot3.name + " " + robot3.model);
robot3.sidekick.attack();
```

Objects as Associative Arrays

- You can also use an associate array style for object access (or you could say associative arrays are just objects?)

```
var customers = new Object();
customers["John Doe"] = "123 Main St SD, CA";
alert(customers["John Doe"]);
```

- The advantage here is that you can use property names not known until run time

```
customerName = prompt("Enter name", "");
customerAddress = prompt("Enter address", "");
customers[customerName] = customerAddress;
alert(customerName);
alert(customers[customerName]);
```

Constructors

- Constructors are special functions that prepare new instances of an object for use
  - A constructor contains an object prototype that defines the code and data that each object instance has by default

```
function Robot() { }  // caps just convention
var guard = new Robot();

function Robot(needsToFly)
  {
     if (needsToFly == true)
         this.hasJetpack = true;
     else
        this.hasJetpack = false;
  }
  // create a Robot with hasJetpack == true
  var guard = new Robot(true);
  // create a Robot with hasJetpack == false
  var sidekick = new Robot();
```

Prototypes

- Every object has a *prototype* property that gives it its structure. The prototype is a reference to an **Object** describing the code and data that all objects of that type have in common

```
Robot.prototype.hasJetpack = false;
 Robot.prototype.doAction = function() {  alert("Watch it!"); };

 function Robot(flying)
 {
    if (flying == true)
       this.hasJetpack = true;
 }


 var guard = new Robot(true);
 var canFly = guard.hasJetpack;
 guard.doAction();
```

Using Prototypes

- Prototypes are of course shared so you can do some interesting things

```
String.prototype.getThirdChar = function()
{
    return this.charAt(2);
}
```

  - Now you can invoke this method as you would any other built-in **String** method:

```
var c = "Example".getThirdChar();  // c set to 'a'
```

- Sometimes you don't even need to do prototypes though esp. if you just want to improve some built-in object instances
  - `document.writeln = function(s) { document.write(s + "<br />\n") };`
  - `document.writeln('he there');`
  - `document.writeln('thomas powell');`

Inheritance via the Prototype Chain

- Inheritance in JavaScript is achieved through prototypes.
- Derive a new object type from a type that already exists inheriting all the properties of their own type in addition to any properties embodied in their parent.

```
function UltraRobot(extraFeature)
  {
     if (extraFeature)
        this.feature = extraFeature;
  }
  UltraRobot.prototype = new Robot();
  UltraRobot.prototype.feature = "Radar";
var guard = new UltraRobot("Performs Calculus");
var feature = guard.feature;
var canFly = guard.hasJetpack;
guard.doAction();
```

Common Properties and Methods

Property	Description
prototype	Reference to the object from which it inherits non-instance properties
constructor	Reference to the function object that served as this object's constructor
toString()	Converts the object into a string (object-dependent behavior)
toLocaleString()	Converts the object into a localized string (object-dependent behavior)
valueOf()	Converts the object into an appropriate primitive type, usually a number
hasOwnProperty(*prop*)	Returns **true** if the object has an instance property named "*prop*", **false** otherwise
isPrototypeOf(*obj*)	Returns **true** if the object serves as the prototype of the object *obj*
propertyIsEnumerable(*prop*)	Returns **true** if the property give in the string *prop* will be enumerated in a **for/in** loop

OOP Use in JavaScript

- JavaScript does support the four aspects of OOP
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

- JS OOP programming quality and acceptance is all over the place
  - Could it be because of the types of applications built?
  - Could it be the type of user writing code? What lang the know?
  - Is the more prototype based style of JS OOP not like class based enough for people? (ES6 is changing all that)
  - Does it mean a language has to be used only one way in order to be useful? Are there impacts to using it (ex: performance)?
  - We leave these questions for you to answer yourself depending on the project you are working on. Hopefully you see that speaking a language like JavaScript will be much more than just the syntax

Summary

- JavaScript supports a basic syntax very similar to C but it is far from it beyond that
- It is a weakly typed language
- It has a limited set of data types
- It is very object flavored and can do function style coding as well but it does not force object-oriented programming nor FP on programmers
- There are many host environment details (browser, nodejs, etc.) that can lead to trouble. Get lang right first to know where to place blame!
- Next up applied JS with the DOM!