Core JavaScript

Lab Help Notes

January 2016, Simon Roberts

Basics

JavaScript uses function scoping, not block scoping. Files are parsed from top to bottom in two passes, declarations are “hoisted” the first time through, effectively being placed at the top of the file or method that contains them. However, initialization statements are not moved.

It's recommended to put declarations at the top of a file or function to avoid confusion.

Execute a script in node.js using:

```
node <filename>
```

Load a script into a webpage

```
<script src='relative path to script.js'
type='text/javascript'></script>
```

Note: ***Do not omit*** the closing script tag!

JavaScript / API Documentation Sources

https://developer.mozilla.org/en-US/docs/Web/JavaScript

http://www.ecma-international.org/ecma-262/5.1/

http://devdocs.io/

Variable declaration

```
var <identifier> [ = <expression> ] ;
```

Identifiers should start with a letter, dollar, or underscore, combinations of letters and numbers may follow. Generally, reserve dollar and underscore for special purposes.

The `var` keyword should generally be used to avoid the variable being injected unintentionally into the global space. Using global space is a source of many errors and should almost always be avoided.

Basic Coding Conventions

- Function and variable names are

`camelCaseWithInitialLower`

- "Constants" are `ALL_CAPS_WITH_UNDERSCORE`
- K&R style braces (i.e. "`{`" goes at the *end* of the line)
- Functions having `InitialCapsCamelCase` should be reserved for "function constructors"
- Never omit final semicolons; the language says these are optional, but all kinds of nasty consequences can arise if they're omitted
- Consider including:
  `"use,strict";`
  at the start of every function, and perhaps at the start of every file. This causes JavaScript to change in ways that reduce the likelihood of "silent" errors.

Essential literal formats

`123`	decimal number
`0123`	octal number (avoid leading zeroes!) Note: JavaScript also supports hexadecimal and binary formats with 0x and 0b prefixes.
`12.3`	number
`12.3E+10`	number
`'x'`	Primitive `String`, may contain `"`
`"xyz"`	Primitive `String`, may contain `'`
`\u56CD`	Unicode character literals may be used in sourcecode (e.g. representing characters not printable in this machine) or in textual literals.
`true` / `false`	boolean

`/[a-z]+/i`	/ … / … defines a regular expression literal, matching the pattern defined between the slashes, with modifiers that follow. (This example matches one or more of 'a' through 'z', as a case insensitive match.
`'\n'` and others	good in `Strings`

`var an_array =` `  [ 1, 3, 5, 7 ];`	Array initialization literal

`var an_object = {`
`  name: “Fred”,`
`  age: 42`
`};`
	Object initialization literal

Primitives and Objects

JavaScript provides several primitive types/values:

`null`

`undefined`

`Boolean`

`Number` (64 bit floating point numeric)

`String` (character sequence)

Boolean, Number, and String exist as both objects and primitives (which will strict-compare as unequal!).

Generally, create primitives as:

```
var s = Number('99');
```

otherwise, strict-equality comparison will fail.

Essential Operators

`+`, `-`, `*`, `/`	Add, subtract, multiply, divide
`++`, `--`	Increment / decrement (prefix or postfix)
`+`	String concatenation, with conversion to string
`%`	Modulus (also known as “remainder”)
[<int>]	Array subscript selection
[<textval>]	Object element access
`delete x`	Delete the field / variable
`&&`, `||`, `!`	Logical and, or, and not (`&&`, `||` "short circuit")
`typeof`	Returns a text representation of operand's data type. Might be: object, string, number, function, undefined

Notes:

- Beware of type coercions with mismatched arguments, particularly with **+**.
- Precedence is broadly normal; use parentheses if unsure!
- Many operators support "assignment operator" forms:
  ```
  a += 20; // add 20 to a
  ```

Comparison Operators

<, <=, >=, >	Less, less or equal, greater or equal, greater
`===`	Strict equals (does not coerce types)
`!==`	Strict not equal (does not coerce types)

Notes:

- == and != are also equality comparison operators, but are

generally not recommended, because they perform type coercions that can lead to unexpected results.

- Equality tests test values of *variables*, which are likely to be references, *not* the objects those references point at. Primitives, however, behave as expected.
- JavaScript does not standardize a “semantic equivalence” function name.

Ternary / Conditional Operator

Set `x` to `"it is!"` if `test` is true, otherwise `"it's not"`:

```
var x = test ? "it is!" : "it's not";
```

Notes:

- The alternatives types do not have to be the same.
- The alternative values can be any valid expression.

Conditional Constructs

```
if ( <expr> ) { <code> }
[ else { <code> } ]

// switch works best with primitive
expressions
switch ( <control expression> ) {
case <expr1>:
  <code>
  break; // falls through without this!
case <expr2>:
case <expr3>:
// sequential cases allow "or" type behavior
...
[ default: ] // if no case matches
```

Iteration Constructs

```
while ( <test expr> ) { <code> }
```

```
do { <code> } while ( <test expr> );
```

C-style "for" loop

```
for ( <inits> ; <test expr> ;
      <increments> ) {
  <code>
}
```

`<test expr>`	Expression controlling loop repetition. If “truthy” the loop executes again.
`<inits>`	Variable declaration/initialization
`<increments>`	Expressions with side effects, e.g. increments

Loop over enumerable fields

```
for ( var <identifier> in <expr> ) {
  <code>
}
```

Example.

```
var person = {
  name: "Fred",
  age: 42
};
for (var field in person) {
  console.log("person[" + field + "] -> "
    + person[field]);
}
```

Notes:

- This iteration technique is generally considered unsafe for use with arrays, as features added to the Array prototype will show up too.
- Historically, `for each` has existed, but should not be used.

Abnormal Loop Execution

Loops may contain `break` and `continue` which allow abandoning a loop, or abandoning the rest of this iteration.

These can be used in nested loops with labels:

```
var x = 15;
outer: while (true) {
  for (var i = 0; i < 10; i++) {
    if (i % 3 === 0) continue;
    console.log(i);
    if (x-- === i) break outer;
  }
}
```

Declaring A Function

```
function <identifier> ( [<arguments>] ) {
  <code>
}
```

Notes:

- Function name must be legal identifier
- Function may use return keyword to return a value to the caller, or to exit before execution reaches the bottom of the function.
- Argument list is optional, form is comma separated list of variable-name identifiers
- Invocation does not have to match the supplied argument list:
  - Missing arguments result in parameter variable having value `undefined`
  - All arguments can be picked up in an array called `arguments`. Access by an index subscript, use `arguments.length` to determine how many were provided.
- Functions are objects. They can be treated like any other data, and can have properties (including other functions).

Example:

```
function makeMessage(name, isMale) {
```

```
  return "Greetings "
    + (isMale ? "Mr." : "Ms.") + name;
}
```

Invocation:

```
console.log(makeMessage("Fred", true));
```

Declaring A Method In A Literal Object

```
var myObj = {
  firstName: "Sheila",
  lastName: "Smith",
  getName: function(formal) {
    if (formal) {
      return "Ms. " + this.firstName
        + " " + this.lastName;
    } else {
      return this.firstName;
    }
  }
}
```

Note: ***Always*** refer to fields of an object using an instance prefix, such as **`this`**. If the prefix is omitted, you'll be referring to values in the current execution context, not an object. This often results in pushing values into the global execution context by mistake.

Invocation Examples:

```
console.log('Welcome '
  + myObj.getName(true));
console.log('Hi ' + myObj.getName(false));
```

Note: Methods on objects that will have multiple instances should typically be provided via the prototype mechanism.

Anonymous Function Declaration

A function is an expression (of type function) and does not need a name. This is similar to an object, which doesn't have an intrinsic name, only variables that refer to it, and its own identity.

Declare an anonymous function in an expression context (not a

statement context).

```
var aFunction = function(a,b) {
  console.log('a is ' + a + ' b is ' + b);
};
```

Notice this fails if not assigned to a variable (because this is a statement context, not an expression context):

```
function(a) { console.log(a);}; // FAILS
```

A function argument is an expression context too, so this works:

```
myButton.addEventListener('click',
  function(e) { console.log('click!'); }
);
```

Closures

Functions defined inside other functions may be passed as return values (or parts of composite return values). When this happens, the inner function retains access to the variables of the enclosing function scope even after the enclosing function has completed execution. This behavior is typically called a closure.

```
function getTestDividesBy(val) {
  return function (v) {
    // this function retains
    // access to val from the
    // call to the enclosing function
    return v % val === 0;
  }
}
var dividesByTwo = getTestDividesBy(2);
var dividesByThree = getTestDividesBy(3);
for (var i = 0; i < 10; i++) {
  console.log(i + ' divides by 2? '
    + dividesByTwo(i));
  console.log(i + ' divides by 3? '
    + dividesByThree(i));
}
```

Closures are often used to create “private” data space that is not accessible directly as fields of objects.

Immediately Invoked Function Expression

Scripts often need to create a storage “namespace” to work in, so as to avoid cluttering global space and risking collision with other scripts also using that namespace. Each function invocation potentially creates a closure, and a closure creates an excellent namespace.

To completely avoid cluttering the global space, we must avoid naming the function, or using a variable to refer to it. This is handled by creating an IIFE:

```
(function() {
  // variables created in this function
  // invocation persist as long as any
  // reference to them exists
  // functions created here can be attached
  // to other components,
  // e.g. as event listeners
}());
```

Or, as a more concrete example:

```
<input type="text" id="myInput">
<script type="text/javascript">
(function() {
  var myInput =
    document.getElementById("myInput");
  myInput.addEventListener('keyup',
    function(e){
    if (e.keyCode === 13) {
      console.log('read: ' + myInput.value);
    }
  });
}());
</script>
```

Note that the IIFE must be a function expression, it ***must*** be surrounded with parentheses, otherwise it will be treated as a syntax error in the attempt to create a function statement.

Creating Objects With Common Features

The object literal form creates a single object, and if called

repeatedly will create “new functions” each time. To avoid this, functions should be defined via the prototype of the object. Two approaches exist, using the “build from prototype” approach is perhaps preferred:

```
var prot = {
  name: 'unset',
  toString: function() {
    return 'My name is ' + this.name;
  }
}
Object.create(prot);
```

Note: this behavior is commonly wrapped in a factory function of some sort.

Handling Error Conditions

JavaScript provides an exception mechanism that uses `try`, `catch`, and `finally`. Any kind of data can be thrown, but an `Error` object is most appropriate generally. `Error` has a constructor that takes a message-string as an argument. The message should describe the problem in some way, and is available later as the `message` member of the object.

A single `catch` block may be provided, which receives anything thrown in the `try` block. The type of what was thrown may be tested with `typeof` or `instanceof`.

A `finally` block can be used for cleanup operations.

For example, if the method dodgy() might throw an Error, then this might be appropriate:

```
try {
  dodgy();
} catch (e) {
  console.log('dodgy broke, and reports: '
    + e.message);
} finally {
  console.log('doing cleanup');
}
```

Problems are reported to callers using the `throw` keyword, with the data to be thrown as an argument:

```
function dodgy() {
  if (Math.random() > 0.5) {
    throw new Error('randomly, that broke');
  }
}
```

Function Constructors

A function constructor is a function that has a `prototype` field on the function object itself. When invoked following the `new` keyword, an object is created. That object has its prototype set to the `prototype` field of the function, and then that object is passed into a call to the function itself as the `this` value. If the function does not explicitly return anything then the `this` value will be returned to the caller.

```
function Thing(color) {
  if (color) this.color = color;
}
Thing.prototype = {
  color: "white",
  toString: function() {
    return 'a ' + this.color + ' Thing';
  }
}
var aThing = new Thing("blue");
```

instanceof

Objects created in this way can participate meaningfully in `instanceof` tests. The `instanceof` operator takes two operands, the first is an object, the second is a function. If the `prototype` of the function that created the object is a prototype of the object, then this returns true.

`aThing instanceof Thing` → true

String Methods

Given:

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

Then:

`s.length` → 5

`s.charAt(4)` → `"o"`

`s.codePointAt(0)` → 104

`String.fromCodePoint(65, 66, 67)` → “ABC”

Notes:

65 is the UTF-8 code for the letter 'A', 66 is 'B', and 104 is 'h'

`s.indexOf('ll')` → 2

Note: return of -1 implies not found

`s.lastIndexOf('l')` → 3

`s.substr(3,2)` → `"lo"`

Notes:

- Second argument is count of characters to include
- Second argument is optional, get “to end” if omitted
- First argument can be negative, which measures from end of string

`s.toUpperCase()` → `"HELLO"` (also `toLowerCase`)

Compare Strings lexically using > and <

Split a string based on a regular expression:

```
'hello there how are you'.split(/[^a-z]+/i)
 → ["hello", "there", "how", "are", "you"]
```

Regular expression matching with capturing:

```
"1234 And this"
  .match(/([0-9]+) ([a-z]+).*/i)
```

→ ["1234 And this", "1234", "And"]

Array Methods

Array have zero-based index behavior, and have many useful behaviors. The behaviors can be invoked on literal arrays, variables, or expressions of array type.

`[1,2,3][0]` → 1

`[1,2,3].length` → 3

`[1,2,3].concat([9,8,7])` → [1, 2, 3, 9, 8, 7]

`[1,2,3].fill(0)` → ar1 now contains [0,0,0]

`[1,2,3].indexOf(2)` → 1

`[1,2,3].join(" : ")` → string value '1 : 2 : 3'

Given:

```
var ar1 = [2,1,3];
```

`ar1.pop()` → 3, ar1 is modified to [2,1]

`ar1.push(9)` → 4, ar1 is modified to [2,1,3,9]

`ar1.shift()` → 1, ar1 is modified to [1,3]

`ar1.sort()` → ar1 is now [1,2,3]

Arrays also support some functional programming behaviors, e.g.

```
[1,2,3].reduce(function(a,b){return a+b;})
      → 6
[1,2,3].filter(function(a){return a%2==0;})
      → [2]
```

Generating Visible Output

In node.js and most browsers:

```
console.log('some text');
```

```
document.write('some text');
```

In Java Nashorn (jjs):

```
print('some text');
```

In HTML pages:

```
<div id='myOutput'></div>
<script type="text/javascript">
  document
    .getElementById("myOutput")
    .innerHTML = 'Some output';
</script>
```

Reading User Input

In Java Nashorn (jjs):

1) Start jjs with the -scripting option

2) `var line = readLine('optional prompt: ');`

In HTML pages:

```
<input type="text" id="myIn">
<script type="text/javascript">
var myIn = document.getElementById("myIn");
myIn.addEventListener('keyup',
  function(e){
    if (e.keyCode === 13) {
      console.log('read: ' + myIn.value);
    }
  });
</script>
```

Note: Input on an HTML page is “event driven”. That means that the act of typing causes the calling of the function. You cannot

have your program call for input when it wants. In effect the cause and effect roles are reversed.

Generating Random Numbers

Generate a random number `x` such that `0 <= x < 1.0`

```
var x = Math.random();
```

Convert `String` And Other Types

Any `Object` Type → `String`

```
myObject.toString()
```

Any Type → `String`

```
"" + value
```

`String` to number types

```
var x = +"3.2"
```

Sorting An Array

If the array contains items that are naturally ordered (primitives) then:

```
var arr = [9,3,1,6,2,8,7,4,5];
arr.sort();
```

If the array contains items that do not have a natural sort order, then sort takes a function argument that takes two arguments, and returns the "difference" between them. The difference value should be numeric, and is effectively positive, negative or zero. The actual value is irrelevant, only the sign matters.

Note, this sort *modifies* the original list.

Get Date/Time Now

```
var d = new Date();
var day = d.getDate();
```

```
var dow = d.getDay(); // 0 = Sunday
var month = d.getMonth();
var year = d.getFullYear();
```

HTML Structure

```
<!doctype html>
<html>
  <head>
    … might contain script / css directives
  </head>
  <body>
    … home for document structure
  </body>
</html>
```

Basic HTML Structure / Elements

Some elements are “paired” and contain other, properly nested, elements. Some are stand-alone.

Nested elements example:

```
<div>
  <h1> heading </h1>
  <div>
    <span> text </span>
  </div>
</div>
```

Elements can have attributes that might be specific to the particular element, and generally often have a `class` attribute that is used by CSS styling and an `id` to facilitate easy identification in code:

```
<input type="button" id="my-button"
  class="red bordered" value="Click Me">
```

Note, id should be unique within the page.

Key Element Types

`div` – Defines a nestable region of the page, often used for styling, layout, or grouping.

`span` – Defines a nestable region of a text flow, often used for styling.

```
<input type="button" value="Click Me!" … >
```

A button. The button's label is defined by the `value` attribute.

```
<input type="text" … >
```

A text field.

`table`, `tr`, `td` – work together to define a table. Cells are surrounded by `td` (“table data”). A row is made up of multiple `td` elements surrounded by `tr` (“table row”). The table is made up of multiple `tr` elements surrounded by `table`. E.g.

```
<table>
  <tr><td>Fred</td><td>3000</td></tr>
  <tr><td>Sheila</td><td>5000</td></tr>
</table>
```

`ul`/`ol`, `li` – work together to make lists, either unnumbered (`ul`), or numbered (`ol` → ordinal). The entire list is surrounded by either `ul` or `ol`, then each list item is surrounded with `li`. E.g.

```
<ul>
  <li>First</li>
  <li>Second</li>
</ul>
```

CSS Styling

Define in a style block in `<head>`:

```
<head>
  <style type="text/css">
    .red {
      background-color: red;
    }
  </style>
</head>
```

Or, define in a file, suppose this is “`mystyle.css`”:

```
.blue {
  background-color: green;
```

```
}
```

Then include the file in the <head> block:

```
<link rel="stylesheet" type="text/css"
  href="mystyle.css">
```

Style rules take the form of a selector, followed by a block containing properties and values.

Key Selectors:

`xxx` – matches an element of type xxx (e.g. `div`)

`#xxx` – matches an element with id xxx

`.xxx` – matches an element with class xxx

`[xxx]` – matches an element with an attribute xxx, note, user-defined attributes are permitted. The specification calls for them to start `data-`. E.g. `<span data-xyz="true"` ...

Key Properties:

`background-color:` sets a background color, e.g. `red`

`border:` sets a border width, style, and color. E.g. `border: 1px solid black;`

`color`: sets a text color, e.g. `blue`

`font-family:` sets a font, or set of alternative fonts, e.g. `"Times New Roman", Georgia, Serif;`

`font-size:` sets a font size e.g. `250%;`

Inline Script Declaration

Define an inline script either in the <head> block, or at the end of the <body> block (usually, to avoid blocking the whole page):

```
<script type="text/javascript">
 var btn = document.getElementById('btn');
 …
</script>
```

External Script Declaration

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

Do not omit the “closing” tag.

Finding DOM Elements

The `document` global variable gives access to the DOM's tree structure

Each node has ***fields*** to access other nodes:

`parentNode`, `previousSibling`, `nextSibling`,

`firstChild`, `lastChild`, `childNodes`

Find elements by tag type:`document.getElementsByTagName("h1")`

Find elements by class name
`document.getElementsByClassName("red")`

Find element(s) matching a CSS selector
`document.querySelector(".green")`
`document.querySelectorAll(".green")`

Find elements by id
`document.getElementById("mybutton");`

Finding Text Of DOM Elements

Contents of a container type node (e.g. div):
`node.innerHTML`

Contents of a text area, or label text on a button
`node.value`

(Other node types might respond to `nodeValue`, `textContent`, `innerText`)

Modifying The DOM

Create / prepare new DOM elements:

```
var para = document.createElement("p");
para.innerHTML = "This is a line";
para.className = "red";
```

Manipulate Element Tree Structure:

```
root.appendChild(para);

root.removeChild(existing);

root.insertBefore(new, beforeThisChild);

root.replaceChild(new, replactThisChild);
```

Some Useful Properties Of Nodes

`n.className` → space separated sequence of classes on this node

`n.classList.add/remove/toggle/contains` allows easy manipulation of classes.

`n.id` → id of this element

`n.checked` → is this item “checked”, applicable to radio buttons and similar

`n.style` → object containing style properties, e.g. `backgroundColor`, `border`, `color`, `fontFamily`, `fontSize`

Notes:

- Styles read this way might not be accurate as the overall layout can change things. Use `window.getComputedStyle(n)` to get accurate info on what's currently happening.
- Style property names change hyphenated forms to camel

case forms, so `background-color` becomes `backgroundColor` etc. Also, some change from e.g. `float` → `cssFloat`.

Events

Different components might report different event types, but in general many—including `<input type="button"`—report a '`click`' event when the mouse is clicked on them. Also `<input type="text"` elements issue an '`input`' event with each change in text.

Events can be sent to registered handlers, which are functions. The function will be provided with an argument, which is the event object.

Event handler should be added to the node in JavaScript code (older ways include embedding `onclick="alert('clicked')"` type code in the HTML, but this is not recommended.

Add an event handler like this:

```
n.addEventListener('click',
  function (e) { ... });
```

The first argument is the name of the event to be handled, the second is the event handler function. The argument to the event handler is the event object.

The 'target' field of an event is the object the event applies to. So, for example, the text currently in a text input can be determined by the construction:

```
e.target.value
```

AJAX

Sending an HTTP request and handling successful response from the server may be performed like this:

```
var req = new XMLHttpRequest();
req.open('GET', 'path/to/data');
req.setRequestHeader('Accept',
  'application/json');
req.addEventListener('load', function(e){
  if (req.status === 200) {
    console.log(req.responseText);
    console.log('headers: '
      + req.getAllResponseHeaders());
  }
}
req.send();
```

Notes:

- The data format received from a request should depend on the Accept header, but `XMLHttpRequest` simply treats the data as text.
- Requests that carry entity data from client to server, such as POST and PUT requests, may be sent. Place the entity data into the argument of the `send` method call.

JSON

JSON conversions may be performed as:

```
var data = JSON.parse(jsonTextData);
var jsonText = JSON.stringify(anObject);
```