JavaScript Transition

Reference Notes

Draft 1. February 2017, Simon Roberts

Getting Going

Options:

- Enter code fragments directly in the Chrome browser's console.
- 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

Variables, as opposed to objects, are entirely ***untyped*** and can be used for multiple different types in the lifetime of a single variable.

Variables are generally references to objects, and the object knows its type and capabilities. Runtime errors arise if a type is misused. Note that some primitive types exist.

```
[ var | let | const ] <identifier>
     [ = <expression> ] ;
```

Four variable declarations forms exist, "naked" (i.e. use without declaration), `var`, `let`, and `const`. Note that `const` is ***not*** a modifier.

Avoid naked / undeclared variables: they are created and placed in the global scope, possibly overwriting another definition!

The `var` declaration creates ***function scoped*** variables. The two-pass interpreter causes the declaration to be "hoisted" but any associated initialization statements are ***not*** moved.

The `let` and `const` forms create "block scoped" variables (this is a newer form).

A `const` declaration must be initialized and creates a constant value. Note that if the value is a reference, `const` does not prevent mutation of the target object.

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

Basic Coding Conventions

- Consider including:
  `"use strict";`
  at the start of every function, or (better) at the start of every file. This causes JavaScript to change in ways that reduce the likelihood of "silent" errors.
- 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"
- It's a good idea to terminate statements with semicolons consistently

Literal formats

Decimal, octal, and hexadecimal literals follow the conventions of C, C++, Java etc. Binary format e.g. `0b0101010` is also supported.

Text strings may be enclosed in ***either*** single or double quotes

(allowing strings containing the “other” quote type). Text can include C style escape sequences, such as `\n`.

`\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.

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

`let 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)

Note that `Boolean`, `Number`, and `String` can exist as either objects and primitives.

Generally, create ***primitives*** using literal formats, or as:

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

If the “constructor form” is used for data that has a primitive form:

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

strict-equality comparisons will fail.

Objects are essentially maps / associative arrays. Field names are keys in the map. Object field access may be as:

```
anObject.x = 99;
```

or:

```
anObject["keys can be expressions"] = 99;
```

Key / value pairs can be added and deleted in the object dynamically.

Object key types are ***not*** limited to textual types

Essential Operators

Many C/C++/Java operators are essentially the same in JavaScript:

`+`, `-`, `*`, `/` , `%`, `++`, `--`, `[`<int>`]`, `&&`, `||`, `!`, `?:`

`+`	String concatenation, with conversion to string (via a method called `toString`)
`[`<expr>`]`	Object element access
`new X(`<args>`)`	Construct instance of `X` using (optional) arguments. Note that the form `X()` does not construct, it merely invokes a function.
`delete x`	Delete the field / variable
`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; parentheses recommended!

Comparison Operators

<, <=, >=, >	Less, less or equal, greater or equal, greater, behave as expected, and produce a boolean
`===`	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 function name for a "semantic equivalence" test.

Flow Control Constructs

Basic flow controls are as C/C++/Java:

`if` / `else`

`switch` / `case` / `break` / `default`

`while`, `do while`, `for`

Note: expressions of **`case`** ***need not*** be constant. This allows the possibility of multiple matches. The first will prevail.

Loop Over Keys Of 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 iterates ***keys***, not values.
- This iteration technique is generally inappropriate for arrays. First, it iterates keys, not values. Second, features added to the array, or **`Array.prototype`** will be enumerated too.
- Historically, **`for each`** has existed, but should not be used.
- Newer JavaScript introduces **`for of`**, which works with value of arrays (and other types that are "iterators"—which are not discussed here).

Labeled `break` / `continue` For Loops

Loops support **`break`** and **`continue`** which 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-like object 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));
```

Variable Length Arguments

The `arguments` variable facilitates variable argument lists. Also, the spread/rest operator … can be used at the end of an argument

list to collect all subsequent actual parameters into a true array:

```
function addressMessage(message, ...names) {
  for (let n of names) // concatenate names
```

Functions As Data

In JavaScript, a function is a legitimate value for a variable, function argument, or return. A function is actually just a special, executable, kind of object. This provides syntactic support for "functional programming" style.

Declaring A Method In A Literal Object

Traditional syntax:

```
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 can result in pushing values into the global execution context by mistake.

ECMAScript 6 Syntax:

```
{
  doStuff() {
    //...
  }
}
```

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!'); }
);
```

ECMAScript 6 Arrow Functions (Lambda Expressions)

Function expressions have two simplified forms. For a function requiring a block body (usually more than one statement):

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

and for a function that simply returns an expression:

```
(a,b) => a+b
```

In either case, parentheses are optional around the argument list if that argument list has exactly one element:

```
a => console.log('a is ' + a)
```

Avoid using arrow functions to define member functions in object literals (they do not handle **`this`** in the expected way).

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.

Warning: Although a nested function retains access to variables in the enclosing scope, the value of **`this`** in the enclosing scope is not usable. If needed, **`this`** should be explicitly assigned to a new local variable, and that local variable used instead of **`this`** in the nested function. (This warning does not apply to nested arrow functions.)

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.

Control Of Fields

JavaScript omits a `private` keyword. Equivalent access protection may be provided using closures instead of objects.

`Object` Methods Controlling Fields

`Object.defineProperty` (and `Object.defineProperties`) can make a field immutable, and/or “hidden” (but ***not*** inaccessible), or alternatively be defined in terms of accessor and/or mutator functions.

```
Object.defineProperty(
     <target-object>, <key>, <descriptor>);
```

The <descriptor> object may contain these fields:

`configurable`	The field may be deleted, or its type altered
`enumerable`	The field is reported in `for in` loops
`value`	The field’s value

`writable`	The field may be assigned
`get`	The accessor function for the field
`set`	The mutator function for the field

Note that `value` and/or `writable` may not be combined with `set` and/or `get`. That is, ***either*** specify storage and an optional write-protection, ***or*** specify access/mutate functions for the field.

`Object.freeze(<target>)` prevents any changes to the object <target> from this point forward.

`Object.seal(<target>)` prevents any changes in the set of properties, without altering the writeability of the existing fields.

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, and if present, is invoked for all flows (successful, unsuccessful-recovered, unsuccessful-unrecovered).

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');
  }
}
```

JavaScript Inheritance

Inheritance is radically different in JavaScript compared with C++/Java etc.

There is no class concept—and no class inheritance—associated with an object. Instead, if the object itself does not define a field or method that is requested, the runtime system checks the prototype of the object. If still not found, the prototype of the prototype is checked all the way up to the prototype of Object.

The prototype, and its contents, is dynamically variable (by assignment at runtime) and is a per-object feature, not a class oriented thing. This behavior is much closer to the tradition of the *Strategy Pattern* (and therefore more flexible, and arguably better, even though unfamiliar).

Read/write access to the prototype is generally available through a field `__proto__` (note there are ***two*** underscores at each end of the name)

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

Inheritance With Prototypes

The object that is used as the prototype for a function constructor likely has its own prototype. This creates an “inheritance chain”, as the search for fields and functions will proceed up the chain until it reaches the Object prototype at the top.

Class Syntax In ECMAScript 6

ECMAScript 6 provides a new syntax that permits templating of objects using a style more consistent with traditional class-based object oriented languages, such as Java, C#, and others. Note that this feature is mostly “syntactic sugar”; that is, it does not change the underlying prototype-based language, nor does it really create classes in the way that other languages understand them.

```
class Person {
  constructor(name) {
    Object.defineProperty(
```

```
      this, 'iname', {value: name});
//  this.iname = name;
  }

  get name() {
    return this.iname;
  }

  toString() {
    return "Person{name: "
      + this.iname + "}";
  }
}
```

Notes:

- The constructor function is referred to as `constructor()`. It differs from a traditional constructor function in that it cannot be called without `new`.
- Methods are defined in a different syntax, without the use of the keyword `function`.
- Accessor methods may be labeled get, in which case, they are invoked as fields, for example:
  `new Person("Fred").name`
  rather than:
  `new Person("Fred").name()`

Class Inheritance

The ECMAScript 6 Class syntax supports a single implementation inheritance model, syntactically comparable with languages like C++ and Java.

```
class Employee extends Person {
  constructor(name, position) {
    super(name);
    Object.defineProperty(this,
      '_position', {value: position});
  }

  get name() {
```

```
    return super.name + ' who is a '
      + this._position;
  }

  get position() {
    return this._position;
  }

  toString() {
    return 'Employee{' + super.toString() +
      ', position: ' + this._position + '}';
  }
}
```

Notes:

- In a constructor `super(...)` passes values to the parent class constructor.
- Methods, including `get`/`set` property methods, may be overridden.
- Parent class fields and behaviors can be accessed using the `super.xxx` and `super(xxx)` syntaxes.
- Despite the syntax, this is still a prototypal inheritance mechanism.

Functional Programming Concepts

Function expressions are commonly passed as arguments into functions and/or returned from functions. A function can create and return a new function derived from argument function.

Behavior as an argument

Example, sorting an array can accept “ordering behavior” as an argument:

```
let names = [ "Fred", "Jim", "Sheila" ];
names.sort(function (a,b) { return b.length-
a.length;});
```

Using bind To Derive Behavior

"Partial Application" is a technique that allows a function to create a new, derived, function. The function prototype defines a **`bind()`** method which provides for this.

```
let add = function(a,b){ return a+b; };
let addFive = add.bind(null, 5);
console.log(addFive(9));
```

In this example, the bind method creates a new function (referred to by addFive, which is equivalent to:

```
function addFive(b) { return add(5, b); }
```

The first argument to **`bind`** is the this context object that will appear as **`this`** in the invoked behavior. In this case, it's `null`, as we have no need of it.

Mix-ins

Mix-In is a term that represents the idea of adding behavior (or fields) to an object "from below", rather than the more familiar adding "from above" that is implied by inheritance. JavaScript provides two tools that provide practical approaches to this.

Mix-Ins "By Hand"

If an object (or its prototype) is not sealed, then new fields and functions can be written to the object (or its prototype) at runtime. This can be used to create a "mix-in" effect.

The method `Object.assign(<target>, … sources)` duplicates enumerable, own, fields from each of `sources` into `<target>` (and returns that target object).

Note that this mix-in mechanism modifies an existing object (or possibly the prototype shared by many objects).

Mix-Ins From Class Expressions

In ECMAScript 6, class definitions can be expressions, in much the same way that functions can. They can also be anonymous. This allows an interesting mechanism that create a mix-in.

```
let Nameable = (x) => class extends x {
  get name() {
    return 'Call me: ' + this.name;
  }
}
```
Notes:

- `Nameable` is a function that takes a class expression and returns a class expression.
- The returned class expression defines a subclass of the provided class (called `x`), which has been extended by the addition of the features provided in the class body that follows; in this example a name accessor function.
- This mix-in approach creates a new class from which instances can be created.

Usage

To use this, define the class that includes the mix-in like this:

```
class NamePerson extends Nameable(Person){};
let np = new NamePerson();
```

Note that in this case, the `NamePerson` may be given additional features in the class body (between the curly braces).

Alternatively, the new class can be left anonymous:

```
let np = new (Nameable(person))();
```

JavaScript Asynchrony

JavaScript is essentially single-threaded, but in many environments (e.g. in the browser, and in node.js) it is “event driven”. This means that the code runs from top to bottom performing initialization and setup, and attaching “call-back handlers” to objects that can trigger an event that occurs later (such as UI input devices, and network requests). Each callback handler is a function expression, and the infrastructure (the event loop) invokes the function when the triggering event arises.

Example, log a message on the console 2 seconds after setup:

```
setTimeout(
```

```
  function() {console.log(“Hello!”);}
  , 2000);
```
Given this, the system will configure the specified behavior, but continue executing. After two seconds, a triggering event will be submitted, and when the event processor has completed setup processing, and processing of any earlier events, then the event handler will invoke the anonymous function, printing the message.

Callbacks are very common in JavaScript code, but can become messy. The “Promises API” was developed to help clean up programs with non-trivial asynchronous requirements.

Promises

ECMAScript 6 formalizes the idea of a “promise”. Essentially this allows client software to make a request for a long-running operation, such as an HTTP request, without blocking the event thread (which would cause the entire user interface to freeze up).

The promise works by allowing the caller to make a request and provide two callback methods, one is called after the request completes successfully, the other will be called if the request fails.

Promises allow for a sequence of such operations, passing data down the sequence in an easy-to-read and easy-to-code form.

Handling a Promise

A Promise is generally handled through its `then` method. The `then` method takes two arguments, each is a function. If the Promise completes successfully (“resolves”), then the first function will be invoked. If the Promise completes unsuccessfully (“rejects”), then the second function is invoked. Assuming doStuff() returns a promise:

```
doStuff().then(
  function(data) {
    console.log("success: " + data);
  }, function(errordata) {
    console.log("failure: " + errordata);
  });
```

Success-only or Failure-only

It's not essential to have both success and failure handling methods provided to a `then` block. If a handler is not provided for the result state, then the `then` block is simply skipped. This makes most sense when `then` blocks are chained.

Chaining `then` blocks

Promises allow the chaining of `then` processing. For example, assuming operation returns a promise:

```
operation()
  .then(success1, failure1)
  .then(success2)
  .then(undefined, recovery3)
  .then(success 4, failure4);
```

The processing function, for success or failure, is called with the data from the previous step.

Each processing function can return a simple data item, or another Promise.

If another promise is returned, then it will complete asynchronously either as success (`resolve`) or failure (`reject`).

If a processing function returns simple data, it be taken to be a success response.

Each step in the chain processes with either the success or failure handler, based on the prior step's result.

If a failure handler returns a promise that resolves, or a simple object, then the following handler will execute on the success path. This facilitates retry type behavior.

Using Promises With Older APIs

Modern APIs for long-running or asynchronous operations might return promises directly, but those that do not can be interfaced to the Promise, as in this example:

```
function delay2secs(x) {
  return new Promise(
    function(resolve, reject) {
```

```
      setTimeout(function(){resolve(x);},
                 2000);
    }
  );
}
```

Note that the call to `resolve(x)` passes the "result" (`x` in this case) back to the Promise, which will in turn pass that data into the success (first) function of the `then` handler. If the `reject` method is called, the same basic behavior results, except that the failure (second) method of the `then` handler is invoked.

Basic API Reference

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 x = 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();
```

`Set` and `Map` Classes

Features of the `Set` class: `size`, `add`, `clear`, `delete`, `entries`, `forEach`, `has`, `values`

Features of the `Map` class: `size`, `clear`, `delete`, `entries`, `forEach`, `get`, `has`, `keys`, `set`

A `Map` can be iterated using `for of`, in which case it yields a series of elements each of which is a key/value pair in a two-element array.

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);
```