JavaScript Basics

What is JavaScript?

From Wikipedia:

> ... **high-level**, **dynamic**, **untyped**, and **interpreted** programming language
>
> ... is **prototype-based** with **first-class functions**, …
>
> ... supporting **object-oriented**, **imperative**, and **functional programming**
>
> ... has an API for working with **text**, **arrays**, **dates** and **regular expressions**

- Not particularly similar to Java: More like C crossed with Self/Scheme
    - C-like statements with everything objects, closures, garbage collection, etc.
- Also known as ECMAScript

Some thoughts about JavaScript

- Example of a **scripting language**
    - Interpreted, less declaring of things, just use them (popular today: e.g. python)
- Seems like it was designed in a rush
    - Some “Good Parts”, some not so good
    - Got a bad reputation

- Many programmers use a subset that avoids some common problems
    - `"use strict";` tweaks language to avoid some problematic parts
- Language being extended to enhance things: New ECMAScript every year!
    - Transpiling common so new features used: e.g ECMAScript Version N, **TypeScript**
- Code quality checkers (e.g. jslint, jshint, eslint) widely used

Good news if you know C - JavaScript is similar

```
i = 3;

i = i * 10 + 3 + (i / 10);

while (i >= 0) { sum
       += i*i; i--;            // Comment
}


for (i = 0; i < 10; i++)                {

}
/* this is a comment */
```

```
if (i < 3) {
        i = foobar(i);
} else {
        i = i * .02;
}
```

Most C operators work:
`* / % + - ! >= <= > < && || ?:`

```
function foobar(i) { return i;}
```

continue/break/return

JavaScript has **dynamic** typing

```
var i;           // Need to define variable ('use strict';), note: untyped

typeof i == 'undefined'              // It does have a type of ‘undefined’

 i  =  32;                  // Now: typeof i == typeof 32 == 'number'

 i  =  "foobar";      // Now: typeof i == typeof 'foobar' == 'string'

 i  =  true;              // Now typeof i == 'boolean'
```

- Variables have the type of the last thing assigned to it
- Primitive types: undefined, number, string, boolean, function, object

Variable scoping with var: Lexical/static scoping

Two scopes: Global and function local

```
var globalVar;

function foo() {
    var localVar;
    if (globalVar > 0) { var
         localVar2 = 2;
    }
    // localVar2 is valid here
}
```

All var statements **hoisted** to top of scope:

```
function foo() {
    var x;
    x = 2;
// Same as: function
foo() {
     x = 2
     var x;
```

localVar2 is hoisted here but has value undefined

Var scope problems

- Global variables are bad in browsers - Easy to get conflicts between modules
- Hoisting can cause confusion in local scopes (e.g. access before value set)

```
function() {
  console.log('Val is:', val);
  ...
  for(var i  = 0; i < 10; i++) {
        var val = "different string"; // Hoisted to func start
```

- Some JavaScript guides suggest always declaring all varat function start
- ES6 introduced non-hoisting, scoped **`let`** and **`const`** with explicit scopes

  Some coding environments ban **`var`** and use **`let`** or **`const`** instead

Var scope problems

- Global variables are bad in browsers - Easy to get conflicts between modules
- Hoisting can cause confusion in local scopes (e.g. access before value set)

```
function() {
    console.log('Val is:', val);                    // Syntax error
    ...
    for(let i  = 0; i < 10; i++) {
            let val = "different string"; // Works
```

- Some JavaScript guides suggest always declaring all var at function start
- ES6 introduced non-hoisting, scoped `let` and explicit scopes

    Some coding environments ban `var` and use `let` or `const` instead

number type

numbertype is stored in floating point (i.e. double in C)

MAX_INT = (2^{53} - 1) = 9007199254740991

Some oddities: NaN, Infinity are numbers

1/0 == Infinity Math.sqrt(-
1) == NaN

Nerd joke: typeof NaN returns 'number'

Watch out:

(0.1 + 0.2) == 0.3 is false // 0.30000000000000004

bitwise operators (e.g. ~, &, |, ^, >>, <<, >>>) are 32bit!

string type

stringtype is variable length (no chartype)

```
let foo = 'This is a test';                    // can use "This is a test" foo.length
                                 // 14
```

+ is string concat operator

```
foo = foo + 'XXX'; // This is a testXXX
```

Lots of useful methods: indexOf(), charAt(), match(), search(), replace(), toUpperCase(), toLowerCase(), slice(), substr(), …

```
'foo'.toUpperCase() // 'FOO'
```

boolean type

- Either `true` or `false`
- Language classifies values as either **truthy** or **falsy**
    - Used when a value is converted to a boolean e.g. if (foo) { …)
- Falsy:

  false, 0, NaN, "", undefined, and null

- Truthy:

  Not falsy (all objects, non-empty strings, non-zero/NaN numbers, functions, etc.)

undefined and null

- **undefined** - does not have a value assign

  let x;   //   x has a value of undefined
  x = undefined;   // It can be explicitly store typeof x ==
  'undefined'

- **null** - a value that represents whatever the user wants it to

  Use to return special condition (e.g. no value)

  typeof null == ‘object’

- Both are falsy but not equal (null == undefined; null !== undefined)

Function type

```
function foobar(x) { if (x <=
        1) {
                return 1;
        }
        return x*foobar(x-1);
}
typeof foobar == ‘function’;                foobar.name == 'foobar'
```

- Function definitions are hoisted (i.e. can use before definition)
- Can be called with a different number arguments than definition
  - Array `arguments`variable (e.g. `arguments[0]`is first argument)
  - Unspecified arguments have value `undefined`
- All functions return a value (default is `undefined`)

"First class" function example

```
let aFuncVar = function (x) {
                                        console.log('Func called with', x); return x+1;
                                    };
      myFunc(aFuncVar);
function myFunc(routine) {                                                       // passed as a param
      console.log('Called with', routine.toString());

      let retVal = routine(10);
      console.log('retVal', retVal); return
      retVal;
}
```

Output

```
  Called with function (x) {
        console.log('Func called with’, x);
        return x+1;
  }

Func called with 10   retVal 11
```

object type

- Object is an unordered collection of name-value pairs called **properties**

  ```
  let foo = {};
  let bar = {name: "Alice", age: 23, state: "California"};
  ```

- Name can be any string: `let x ={ "": "empty", "---": "dashes"}`

- Referenced either like a structure or like a hash table with string keys:

  ```
      bar.name or  bar["name"]
      x["---"]              // have to use hash format for illegal names
      foo.nonExistent == undefined
  ```

- Global scope is an object in browser (i.e. `window[prop]`)

Properties can be added, removed, enumerated

- To add, just assign to the property:

```
let foo = {}; foo.name =
"Fred";                                    // foo.name returns "Fred"
```

- To remove use delete:

```
let foo = {name: "Fred"};
delete  foo.name; // foo is now an empty object
```

- To enumerate use Object.keys():

```
Object.keys({name: "Alice", age: 23}) = ["name", "age"]
```

Arrays

```
let anArr = [1,2,3];
```

Are special objects: typeof anArr == 'object'

Indexed by non-negative integers: (anArr[0] == 1)

Can be **sparse** and **polymorphic**: anArr[5]='FooBar'; //[1,2,3,,,'FooBar']

Like strings, have many methods: anArr.length == 3

push, pop, shift, unshift, sort, reverse, splice, …

Oddity: can store properties like objects (e.g. anArr.name = 'Foo')

Some properties have implications: (e.g. anArr.length = 0;)

Dates

```
let date = new Date();
```

Are special objects: typeof date == 'object'

The number of milliseconds since midnight January 1, 1970 UTC

Timezone needed to convert. Not good for fixed dates (e.g. birthdays)

Many methods for returning and setting the data object. For example:

date.valueOf() = 1452359316314 date.toISOString() = '2016-01-09T17:08:36.314Z'

date.toLocaleString() = '1/9/2016, 9:08:36 AM'

Regular Expressions

`let re = /ab+c/;` or `let re2 = new RegExp("ab+c");`

Defines a pattern that can be searched for in a string

String: `search()`, `match()`, `replace()`, and `split()`

RegExp: `exec()` and `test()`

Cool combination of CS Theory and Practice: CS143

Uses:

Searching: Does this string have a pattern I’m interested in?

Parsing: Interpret this string as a program and return its components

Regular Expressions by example - search/test

```
/HALT/.test(str);              // Returns true if string strhas the substr HALT
/halt/i.test(str);             // Same but ignore case
/[Hh]alt [A-Z]/.test(str); // Returns true if streither “Halt L” or “halt L”

 'XXX   abbbbbbc'.search(/ab+c/);          // Returns 4 (position of ‘a’)
 'XXX   ac'.search(/ab+c/);                // Returns -1, no match
 'XXX   ac'.search(/ab*c/);                // Returns 4

'12e34'.search(/[^\d]/);                   // Returns 2
'foo: bar;'.search(/...\s*:\s*...\s*;/);              //Returns 0
```

Regular Expressions - exec/match/replace

```
let str = "This has 'quoted' words like 'this'"; let re = /'[^']*'/g;

re.exec(str);        // Returns ["'quoted'", index: 9,            input: …
re.exec(str);        // Returns ["'this'", index: 29,           input: …
re.exec(str);        // Returns  null

str.match(/'[^']*'/g);          // Returns  ["'quoted'", "'this'"]

str.replace(/'[^']*'/g, 'XXX'); // Returns:
                                       'This has XXX words with XXX.'
```

Exceptions - try/catch

- Error reporting frequently done with exceptions
  Example:

  ```
      nonExistentFunction();
  ```

  Terminates execution with error:

  ```
      Uncaught ReferenceError: nonExistentFunction is not defined
  ```

- Exception go up stack: Catch exceptions with try/catch

  ```
  try {
     nonExistentFunction();
  } catch (err) {          // typeof err 'object' console.log("Error call func",
     err.name, err.message);
  }
  ```

Exceptions - throw/finally

- Raise exceptions with throwstatement

```
try {
      throw  "Help!";
} catch (errstr) {                  // errstr === "Help!"
      console.log('Got exception', errstr);
} finally  {
       // This block is executed after try/catch
}
```

- Conventions are to throwsub-classes of Errorobject

```
console.log("Got Error:", err.stack || err.message || err);
```

Getting JavaScript into a web page

- By including a separate file:

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

- Inline in the HTML:

```
<script type="text/javascript">
//<![CDATA[
Javascript goes here...
//]]>
</script>
```

ECMAScript

- New standard for ECMAScript released yearly
  - Relatively easy to get a new feature into the language
- Transpiling: Translate new language to old style JavaScript
  - Allows front-end software to be coded with new features but run everywhere.
  - For example: Babel. Check out: https://babeljs.io/en/repl new JS in -> old JS out
- Frontend frameworks are aggressively using new language features
  - React.js - Encourages use of newer ECMAScript features
  - Angular - Encourages Typescript - Extended JavaScript with static types and type checking

Lots of new features in ECMAScript

- Already seen a few
  - let, const, class, =>
- Here are a few more you might encounter:
  - Modules
  - Default parameters
  - Rest parameters ...
  - Spread operator ...
  - Destructuring assignment
  - Template string literals
  - Set, Map, WeakSet, WeakMap objects, async programming

Default parameters - Parameters not specified

Old Way

```
function myFunc(a,b) {
        a = a || 1;
        b = b || "Hello";
}
```

Unspecified parameters are set to undefined. You need to explicitly set them if you want a different default.

New Way

```
function myFunc (a = 1, b = "Hello") {

}
```

Can explicitly define default values if parameter is not defined.

Rest parameters ...

Old Way

```
function myFunc() {
        var a = arguments[0]; var b
        = arguments[1]; var c =
        arguments[2];
          arguments[N]
        //
}
```

Parameters not listed but passed can be accessed using the argumentsarray.

New Way

```
function myFunc (a,b,...theArgsArray) {

        var c = theArgsArray[0];

}
```

Additional parameters can be placed into a named array.

Spread operator ...

Old Way

```
var anArray = [1,2,3];
myFunc.apply(null, anArray);

var o = [5].concat(anArray).concat([6]);
```

Expand an array to pass its values to a function or insert it into an array.

New Way

```
var anArray = [1,2,3];
myFunc(...anArray);

var o = [5, ...anArray, 6];
```

Works on iterable types: strings & arrays

Destructuring assignment

Old Way

```
var a = arr[0]; var b
= arr[1]; var c =
arr[2];

var name = obj.name; var
age = obj.age;
var salary = obj.salary;

function render(props) { var
    name = props.name; var age
    = props.age;
```

New Way

```
let [a,b,c] = arr;

let {name, age, salary} = obj;

function render({name, age}) {
```

Template string literals

Old Way

```
function formatGreetings(name, age) { var str =
    "Hi " + name +
        " your age is " + age;
 ...
```

Use string concatenation to build up string from variables.

New Way

```
function formatGreetings(name, age) {

 let str =
     `Hi ${name} your age is ${age}`;
```

Also allows multi-line strings:

```
`This string has two lines`
```

Very useful in frontend code. Strings can be delimited by " ", ' ', or \` \`

For of

Old Way

```
var a = [5,6,7]; var
sum = 0;
for (var i = 0; i < a.length; i++) { sum += a[i];
}
```

Iterator over an array

New Way

```
let sum = 0;
for (ent of a) { sum
    += ent;
}
```

Iterate over arrays, strings, Map, Set, without using indexes.

Some additional extensions

- Set, Map, WeakSet, WeakMap objects
  - Defined interfaces for common abstractions
- async/await and Promises
  - Asynchronous programming help