JavaScript Basics

Mendel Rosenblum
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

- Seems like it was designed in a rush
  - Some “Good Parts”, some not so good
  - Got 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

```javascript
i = 3;
i = i * 10 + 3 + (i / 10);
while (i >= 0) {
    sum += i*i;  // Comment
    i--;
}
for (i = 0; i < 10; i++) {
}
/* this is a comment */
```

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

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

```javascript
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: Lexical/static scoping

Two scopes: Global and function local

```javascript
var globalVar;

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

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

```javascript
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)
  ```javascript
  function() {
    var val = "string";
    ...
    for(var i = 0; i < 10; i++) {
      var val = "different string"; // Hoisted to func start
  }
  ```
- 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
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)
  ```javascript
  function() {
    let val = "string";
    ...
    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

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

$$\text{MAX\_INT} = (2^{53} - 1) = 9007199254740991$$

Some oddities: NaN, Infinity are numbers

$$1/0 == \text{Infinity}$$
$$\text{Math.sqrt(-1)} == \text{NaN}$$

Watch out:

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

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

string type is variable length (no char type)

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

+ is string concat operator

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

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

```javascript
'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, "", null, undefined, and NaN
- Truthy:
  - Not falsy (all objects, non-empty strings, non-zero numbers, functions, etc.)
undefined and null

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

```javascript
var 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)

```javascript
typeof null == ‘object’
```

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

```javascript
var foobar = function foobar(x) {   // Same as 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 define)
- Can be called with variable arguments
  - 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

```javascript
var aFuncVar = function (x) {
    console.log('Called with', x);
    return x+1;
};

function myFunc(routine) { // passed as a param
    console.log('Called with', routine.toString());
    var retVal = routine(10);
    console.log('retVal', retVal);
    return retVal;
}

myFunc(aFuncVar);
```

Output
Called with function (x) {
    console.log('Called with', x);
    return x+1;
}
Called with 10
retVal 11
object type

- Object is an unordered collection of name-value pairs called **properties**
  ```javascript
  var foo = {};
  var bar = {name: "Alice", age: 23, state: "California"};
  ```
- Name can be any string:
  ```javascript
  var x = { "": "empty", "---": "dashes"};
  ```
- Referenced either like a structure or like a hash table with string keys:
  ```javascript
  bar.name or bar["name"]
  x["---"] // have to use hash format for illegal names
  ```
  ```javascript
  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:

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

- To remove use `delete`:

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

- To enumerate use `Object.keys()`:

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

```javascript
var 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

```javascript
var 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

```
var re = /ab+c/;  or  var 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 str has the substr HALT
/halt/i.test(str);  // Same but ignore case
/[Hh]alt [A-Z]/.test(str);  // Returns true if str either “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

```javascript
var str = "This has 'quoted' words like 'this';
var 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:
  ```javascript
  nonExistentFunction();
  ```
  Terminates execution with error:
  ```javascript
  Uncaught ReferenceError: nonExistentFunction is not defined
  ```

- Exception go up stack: Catch exceptions with try/catch
  ```javascript
  try {
    nonExistentFunction();
  } catch (err) { // typeof err 'object'
    console.log("Error call func", err.name, err.message);
  }
  ```
Exceptions - throw/finally

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

- Conventions are to throw sub-classes of Error object
  
  ```javascript
  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>`