JavaScript

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 by Netscape
  - 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
- Code quality checkers (jslint and jshint) 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;   // 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

```javascript
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;
    }
}
```

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

```javascript
function foo() {
    var x;
    x = 2;
    // Same as:
    function foo() {
        x = 2
        var x;
    }
}
Var scope problems

- Global variables are bad in browsers
  Easy to get conflict between modules
- Hoisting can cause confusion in local scopes

```javascript
function() {
    for(var i = 0; i < 10; i++) {
        ....
    }
    for(var i = 0; i < 25; i++) { // Error: i already defined

- Some JavaScript guides suggest always declaring all var at function start
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

\[ \frac{1}{0} == \text{Infinity} \]
\[ \text{Math.sqrt(-1)} == \text{NaN} \]

Watch out: 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(), ...
**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
function type

```javascript
var foobar = function foobar(x) { // Same as function foobar(x)
    if (x <= 1) {
        return 1;
    }
    return x*fac(x-1);
}
typeof foobar == 'function' ; foobar.name == 'foobar'

- 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).
```
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
  ```

- 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

    var anArr = [1,2,3];

Are special objects: typeof anArr == 'object'

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

Can be parse 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;)

15
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:

  ```javascript
date.valueOf() = 1452359316314
  date.toISOString() = '2016-01-09T17:08:36.314Z'
  date.toLocaleString() = '1/9/2016, 9:08:36 AM'
  ```
Regular Expressions

```javascript
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.'
```
Object-oriented programming: methods

- A property of an object can be a function
  ```javascript
  var o = {count: 0};
  o.increment = function (amount) {
    if (amount == undefined) {
      amount = 1;
    } 
    this.count += amount;
    return this.count;
  }
  ```

- Method invocation:
  ```javascript
  o.increment(); // returns 1
  o.increment(3); // returns 4
  ```
this

● In methods this will be bound to the object

```javascript
var o = {oldProp: 'this is an old property'};
o.aMethod = function() {
    this.newProp = "this is a new property";
    return Object.keys(this);  // will contain 'newProp'
}
o.aMethod(); // will return ['oldProp','aMethod','newProp']
```

● In non-method functions:
  ○ this will be the global object
  ○ Or if "use strict"; this will be undefined
functions can have properties too

```javascript
function plus1(value) {
    if (plus1.invocations == undefined) {
        plus1.invocations = 0;
    }
    plus1.invocations++;
    return value + 1;
}
```

- `plus1.invocations` will be the number times function is called
Object-oriented programming: classes

Functions are classes in JavaScript: Name the function after the class

```javascript
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
    this.area = function() { return this.width*this.height; }
}
var r = new Rectangle(26, 14);   // {width: 26, height: 14}
```

Functions used in this way are called **constructors**:

```javascript
r.constructor.name == 'Rectangle'
```

```javascript
console.log(r): Rectangle { width: 26, height: 14, area: [Function] }
```
Object-oriented programming: inheritance

- Javascript has the notion of a **prototype** object for each object instance
  - Prototype objects can have prototype objects forming a **chain**

- On an object property read access JavaScript will search the up the prototype chain until the property is found
  - Effectively the properties of an object are its **own** property in addition to all the properties up the prototype chain. This is called prototype-based inheritance.

- Property updates are different: always create property in object if not found
  - Can lead to fun in AngularJS
Using prototypes

```javascript
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
}
Rectangle.prototype.area = function() {
  return this.width*this.height;
}
var r = new Rectangle(26, 14);   // {width: 26, height: 14}
var v = r.area();      // v == 26*14
Object.keys(r) == [ 'width', 'height' ] // own properties
```

Note: Dynamic - changing prototype will cause all instances to change
Inheritance

Rectangle.prototype = new Shape(...);

• If desired property not in Rectangle.prototype then JavaScript will look in Shape.prototype and so on.
Functional Programming

● Imperative:
  ```javascript
  for (var i = 0; i < anArr.length; i++) {
    newArr[i] = anArr[i]*i;
  }
  ● Functional:
  newArr = anArr.map(function (val, ind) {
    return val*ind;
  });
  ● Can write entire program as functions with no side-effects
  anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
Can mostly but not totally avoid functional style

- Asynchronous events done with callback functions

**Browser:**

```javascript
function callbackFunc() { console.log("timeout"); } 
setTimeout(callbackFunc, 3*1000);
```

**Server:**

```javascript
function callbackFunc(err, data) { console.log(data); } 
fs.readFile('/etc/passwd', callbackFunc);
```
Closures

An advanced programing language concept you need to know about

```javascript
var globalVar = 1;
function localFunc(argVar) {
    var localVar = 0;
    function embedFunc() {return ++localVar + argVar + globalVar;}
    return embedFunc;
}
var myFunc = localFunc(10); // What happens if a call myFunc()? Again?

- myFunc closure contains argVar, localVar and globalVar
```
Using Scopes and Closures

- Consider effect on the scopes of:

```
var i = 1;
```

Versus

```
(function () {
    var i = 1;
})(());
```
Using closures for private object properties

```javascript
var myObj = (function() {
    var privateProp1 = 1;  var privateProp2 = "test";
    function _setPrivate1(val1)  { privateProp1 = val1; }
    var compute = function() {return privateProp1 + privateProp2;}
    var setPrivate1 = function (val1) { _setPrivate1(val1); }
    return {compute: compute, setPrivate1: setPrivate1};
})();

typeof myObj;       // 'object'
Object.keys(myObj); // [ 'compute', 'setPrivate1' ]

What does myObj.compute() return?
```
Closures can be tricky with imperative code

// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}

* Ends up printing two files to console both starting with:
  file 2 has length
  Why?
Exceptions - try/catch

- Error reporting frequently done with exceptions
  Example:
  ```javascript
  nonExistentFunction();
  Terminates execution with error:
  Uncaught ReferenceError: nonExistentFunction is not defined
  ```
- Exception go up stack: Catching 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);
  ```
JavaScript Object Notation (JSON)

```javascript
var obj = { ps: 'str', pn: 1, pa: [1, 'two', 3, 4], po: { sop: 1 }};

var s = JSON.stringify(obj) =
    '{"ps":"str","pn":1,"pa":[1,"two",3,4],"po":{"sop":1}}'

typeof s == 'string'

JSON.parse(s) // returns object with same properties

- JSON is the standard format for sending data to and from a browser
```
JavaScript: The Bad Parts

Declaring variables on use - Workaround: Force declarations

```
var myVar = 2*typeoVar + 1;
```

Automatic semicolon insertion - Workaround: Enforce semicolons with checkers

```
return "This is a long string so I put it on it's own line";
```

Type coercing equals: `==` - Workaround: Always use `===`, `!==` instead

```
("" == "0") is false but (0 == ")" is true, so is (0 == '0')
(false == '0') is true as is (null == undefined)
```

with, `eval` - Workaround: Don't use
Some JavaScript idioms

- Assign a default value
  
  ```javascript
  hostname = hostname || "localhost";
  port = port || 80;
  ```

- Access a possible undefined object property
  
  ```javascript
  var prop = obj && obj.propname;
  ```

- Handling multiple `this`: self
  
  ```javascript
  var self = this;
  fs.readFile(self.fileName + fileNo, function (err, data) {
    console.log(self.fileName, fileNo);
  });
  ```
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>