JS Cheat Sheet

Objects

Creation

```
// OBJECT LITERAL

var empty_object = {};

var new_object = {
  "name" : "john",
  "age" : "29"
};

Every object has a constructor function and is associated to a prototype object
from which it can inherit properties.

Objects defined with an object literal are created using a generic constructor
function Object() {} (linked to the prototype object Object. Objects produced from
object literals are linked to Object.prototype.
To specify a proptotype the object Object can be extended with a new method:

if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    }
}

var new_obj = Object.create(prototype_obj);

To avoid looking at the prototype chain the hasOwnProperty method should
be used:

new_object.hasOwnProperty('name') // true

To loop over all the properties of an object in a specific order without going up the
prototype chain for is the prefered method:

var i;
var prop_order = [name,
                  age];
for (i = 0; i < prop_order.length; i += 1) {
    println(prop_order[i] + ' : ' + new_obj[prop_order[i]]);
}
```

```
// CONSTRUCTOR

var Person = function Person (age,gender) {
    this.age = age;
    this.gender = gender;
    this.getGender = function() {return this.gender;};
};

var alan = new Person(32,'Male');
```

```
// Object.create

var obj1 = {'name' : 'james', 'age' : 32};
var obj2 = Object.create(obj1, {'gender' : {value : 'male', writable : true}});

obj2.age // 32
obj1.hasOwnProperty('age')  // true
obj2.hasOwnProperty('age') // false

obj1.isPrototypeOf(obj2)  // true
```

```
// Constructor that fixes new operator absence
  function User(first, last) {
    if (!(this instanceof arguments.callee)) {
        return new User(first,last);
    }
    this.name = first + " " + last;
  }
```

Inheritance with prototype

```
function Person () {};
Person.prototype.talk = function() {};

function Player () {};
Player.prototype = new Person();

var player = new Player();
player.talk();
```

Descriptors set and get

```
var obj1 = {};

Object.defineProperty(obj1,'name',{configurable : true, value : 'mary', writable :
true});

Object.getOwnPropertyDescriptor (obj1,'name');

configurable: true
enumerable: false
value: "mary"
writable: true
__proto__: Object
```

Retrieval

```
new_object.name    // "john"
new_object["age"]   //  "29"
```

Update

```
new_object.name = 'alan';
new_object["age"] = '39';
```

Delete

```
delete new_obj.name;
```

Reference

```
var x = obj1;
x.name = 'Tom';
var name = obj1.name;  //name = 'Tom' because objects are passed by reference
```

JSON conversion

```
s = JSON.stringify(obj);
p = JSON.parse(obj);
```

Property attributes

```
Object.getOwnPropertyDescriptor(new_object,'name')
// Returns {value: 'alan', writable:true, enumerable:true, configurable:true}

Object.defineProperty(new_object, "name", { value: 'tommy' });
```

Functions

Definition

```
Functions in JavaScript are objects. Function objects are linked to
Function.prototype. Every function object is also created with a prototype property.
Its value is an object with a constructor property whose value is the function.

Function constructor runs some code like this:

this.prototype = {constructor: this};

var func = function () { return true;}

var proto = func.prototype;

Object
constructor: function () { return true;}
__proto__: Object


First class
behaves like a value :

var fortytwo = function() { return 42 };
var fortytwos = [42, function() { return 42 }];
var fortytwos = {number: 42, fun: function() { return 42 }};
42 + (function() { return 42 })();

Higher order
can take another function as an argument/can return a function:

function weirdAdd(n, f) { return n + f() }
weirdAdd(42, function() { return 42 });
```

Creation

```
function func1() {
  return true;
}

function factorial(x) {
  if (x <= 1) return 1;
  return x * factorial(x-1);
}

var square = function(x) { return x*x; }

data.sort(function(a,b) { return a-b; });

var tensquared = (function(x) {return x*x;}(10));

function hypotenuse(a, b) {
  function square(x) { return x*x; }
  return Math.sqrt(square(a) + square(b));
}
```

Invocation

```
// Method invocation pattern - this is bound to that object

var obj1 = {
    value : 1,
    inc : function (incVal) {
        this.value += typeof incVal === 'number' ? incVal : 1;}
}

// Function invocation pattern - this is bound to the global object

var sum = add(3, 4);

//accessing this in an inner function:

myObject.double = function () {
var that = this; // Workaround.
var helper = function () {
that.value = add(that.value, that.value);
};
helper(); // Invoke helper as a function. };

// Constructor invocation pattern - a new object is created and this is bound to it

var Quo = function (string) {
    this.status = string;
};

Quo.prototype.get_status = function () { return this.status;};

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status()); // confused

// Apply invocation pattern - sets the value of this

var statusObject = {
    status: 'A-OK'
};

var status = Quo.prototype.get_status.apply(statusObject);  // status is 'A-OK'
```

Arguments

```
Besides the invocation paramenters functions also receive two aditional
parameters (this and arguments)

var sum = function () { var i, sum = 0;
        for (i = 0; i < arguments.length; i += 1) {
            sum += arguments[i];
        }
        return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42)); // 108
```

Closure

```
Inner functions get access to the parameters and variables of the functions they
are defined within (with the exception of this and arguments)

A function has access to the context in which it was created.

var add_the_handlers = function (nodes) {
        var i;
        for (i = 0; i < nodes.length; i += 1) {
            nodes[i].onclick = function (e) {
alert(i); };
} };

//always displays the number of nodes because the handlers functions are bound
to the variable i not the value of the variable i when the function was created. To
correct it:

var add_the_handlers = function (nodes) {
        var i;
        for (i = 0; i < nodes.length; i += 1) {
            nodes[i].onclick = function (i) {
                return function (e) {
                    alert(e);
} ;}(i);
} };
```

Method chaining (methods return this)

```
shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();
```

Module pattern

```
The general pattern of a module is a function that defines private variables and
functions; creates privileged functions which, through closure, will have access to
the private variables and functions; and that returns the privileged functions or
stores them in an accessible place.

var serial_maker = function () {
    var prefix = '';
    var seq = 0;
    return {
        set_prefix: function (p) {
            prefix = String(p);
        },
        set_seq: function (s) {
            seq = s ;},
        gensym: function () {
            var result = prefix + seq; seq += 1;
            return result; }
    };
};

var seqer = serial_maker();
seqer.set_prefix = ('Q');
seqer.set_seq = (1000);
var unique = seqer.gensym();     // unique is "Q1000"
```

Asynchronicity

When code puts things in the queue

```
Steps:

1. evaluate code and populate queue with things to do, for example, setTimeout,
dom events, HTTP communication
2. execute code in the queue and queue up new events
3. when finished executing code check the queue again

Queue is checked only when there's nothing else to do. It never interrupts the
current flow.
```

Browser

Javascript runtime	webapis (DOM, ajax, setTimeout) run on different threads
heap - memory allocation	event loop - if the stack is empty pushes the first thing on the queue into the stack and runs it
call stack - single thread/one thing at a time	callback queue - when webapis are done callback are pushed to the queue
	render queue - has higher priority then callback queue but is constrained by the call stack being empty. To achieve optimal 60 fps the stack shouldn't take more then 16 ms to clear.