JavaScript language a- z cheat sheet

Here is an A - Z list of some Javascript idioms and patterns. The idea is to convey in simple terms some features of the actual Javascript language (rather than how it can interact with DOM). Enjoy...

Array Literals
An array literal can be defined using a comma separated list in square brackets.

```
var months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun',
              'jul', 'aug', 'sep', 'oct', 'nov', 'dec'];
console.log(months[0]); // outputs jan
console.log(months.length) // outputs 12
```

Arrays in javascript have a wide selection methods including push() and pop(). Suppose the world got taken over by a dictator who wanted to get rid of the last month of the year? The dictator would just do...

```
months.pop();
```

And of course, the dictator will eventually want to add a month after himself when everyone will have to worship him:

```
months.push("me");
```

Callbacks
Since functions are objects, they can be passed as arguments to other functions.

```
function peakOil(callback) {
    //... code
    callback();  // the parentheses mean the function is executed!
}

function changeCivilisationCallback(){
   //...
}

// Now pass the changeCivilisationCallback to peakOil.
// Note: no changeCivilisationCallback parentheses because it is not
// executed at this point.
// It will be excuted later inside peak oil.
peakOil(changeCivilisationCallback);
```

In the example above, the *chanceCivilisationCallback* callback function is invoked by *peakOil*. Logic could be added to check if the energy returns from solar panels and wind farms were sufficient in which case another callback, other than *changeCivilisationCallback* could be added.

Configuration Object
Instead of passing around a bunch of related properties...

```
function addCar(colour, wheelsize, regplate) {...}
```

Use a configuration object

```
function addCar(carConf) {...}

var myCarConf = {
    colour: "blue",
    wheelsize: "32",
    regplate: "00D98788"
```

```
};
addCar(myCarConf);
```

The use of a configuration object makes it makes it easier to write clean APIs that don't need to take a huge long list of parameters. They also means you are less likely to get silly errors if parameters are in the wrong order.

Closures

There are three ways to creats objects in Javascript: using literals, using the constuctor function and by using a closure. What closures offer that the other two approaches do not is *encapsulation*. Closures make it possible to hide away functions and variables.

```
var counter = function(count) {
    console.log(">> setting count to " + this.count);
    return {
        getCount: function(){
           return ++count;
        }
    }
}

mycounter = counter(0);
console.log(mycounter.getCount());  // outputs 1
console.log(mycounter.getCount());  // outputs 2
console.log(mycounter.getCount());  // outputs 3
console.log(mycounter.getCount());  // outputs 4

// Same again with offset this time.
mycounterWithOffset = counter(10);
console.log(mycounterWithOffset.getCount());  // outputs 11
console.log(mycounterWithOffset.getCount());  // outputs 12
console.log(mycounterWithOffset.getCount());  // outputs 13
console.log(mycounterWithOffset.getCount());  // outputs 14
```

Note: The closure is the object literal returned from annoymous function. It "closes" over the count variable. No-one can access it except for the closure. It is encapsulated. The closure also has a sense of state. Note also how the it maintains the value of the counter.

Constructor Functions (Built in)

There are no classes in Javascript but there are construtor functions which use the *new* keyword syntax similar to the class based object creation in Java or other languages. Javascript has some built-in constructor functions. These include Object(), Date(), String() etc.

```
var person = new Object();  // person variable is an Object
person.name = "alex";  // properties can then be dynamically added
```

Constructor Functions (Custom)

When a function is invoked with the keyword *new*, it is referred to as a Constructor function. The *new* means that the new object will have a hidden link to value of the function's *prototype* member and the *this* keyword will be bound to the new object.

```
function MyConstrutorFunction() {
    this.goodblog = "dublintech.blogspot.com";
}

var newObject = new MyConstrutorFunction();
console.log(typeof newObject);    // "object"
console.log(newObject.goodblog);  // "dublintech.blogspot.com"

var noNewObject = MyConstrutorFunction();
console.log(typeof noNewObject);  // "undefined"
console.log(window.tastes);       // "yummy"
```

The convention is that constructor functions should begin with a capital letter. Note: if the *new*keyword is not used, then the *'this'* variable inside the function will refer to the global object. Can you smell a potential mess? Hence why the capital letter convention for constructor functions is used. The capital letter means: "I am a constructor function, please use the *new* keyword".

Currying

Currying is the process of reducing the number of arguments passed to a function by setting some argument(s) to predefined values. Consider this function.

```
function outputNumbers(begin, end) {
    var i;
    for (i = begin; i <= end; i++) {
        print(i);
    }
}
outputNumbers(0, 5);  // outputs 0, 1, 2, 3, 4, 5
outputNumbers(1, 5);  // outputs 1, 2, 3, 4, 5
```

Suppose, we want a similar function with a fixed "begin" value. Let's say the "begin" value was always 1. We could do:

```
function outputNumbersFixedStart(start) {
    return function(end) {
        return outputNumbers(start, end);
    }
}
```

And then define a variable to be this new function...

```
var outputFromOne = outputNumbersFixedStart(1);
outputFromOne(3);  1, 2, 3
outputFromOne(5);  1, 2, 3, 4, 5
```

Delete Operator

The delete operator can be used to remove properties from objects and arrays.

```
var person = {name: 'Alex', age: 56};
// damn I don't want them to know my age remove it
delete person.age;
console.log("name" in person);  // outputs true because it is still ther
console.log("age" in person);   // outputs false


var colours = ['red', 'green', 'blue']
// is red really in the array?
console.log(colours.indexOf('red') > -1);  // outputs true.
// remove red, it's going out of fashion!
delete colours[colours.indexOf('red')];
console.log(colours.indexOf('red') > -1);  // outputs false
console.log(colours.length) // length is still three, remember it's java
```

You cannot delete global variables or prototype attributes.

```
console.log(delete Object.prototype)  // can't be deleted, outputs fals
function MyFunction() {
    // ...
}
console.log(delete MyFunction.prototype) // can't be deleted, outputs fal

var myglobalVar = 1;
console.log(delete this.myglobalVar)   // can't be delete, outputs false
```

Dynamic Arguments

Arguments for a function do not have to be specifed in the function definition

```
function myFunction(){
   // ... Note myfunction has no arguments in signature
   for(var i=0; i < arguments.length; i++){
       alert(arguments[i].value);
   }
}

myFunction("tony", "Magoo");  // any argument can be specified
```

The arguments parameter is an array available to functions and gives access to all arguments that were specified in the invocation.

for-in iterations

for-in loops (also called enumeration) should be used to iterate over nonarray objects.

```
var counties = {
    dublin: "good",
    kildare: "not bad",
    cork: "avoid"
}

for (var i in counties) {
    if (counties.hasOwnProperty(i)) { // filter out prototype properties
        console.log(i, ":", counties[i]);
    }
}
```

Functions are literals

This is an important one for people coming from a Java background. Functions do not need to have names. They can be anonymous, they can be passed into and returned from other functions without any needing a name - they can be treated *literally*. When a Java developer sees a function, they can't help thinking they are analogous to Java methods. But, Java methods can never be anonymous, they can be never be passed to or returned from other methods. They can be wrapped in an anonymous object defined on the fly; but they need that object that in many cases does nothing else - the methods themselves can never be treated *literally*. JavaScript ability to treat functions literally gives it a lot of expressive power.

Function declaration

In a function declaration, the function stands on its own and does not need to be assigned to anything.

```
function multiple(a, b) {
    return a * b;
} // Note, no semi colan is needed
```

Function expressions

When function is defined as part of something else's definition, it is considered a function expression.

```
multiply = function multiplyFunction(a, b) {
```

```
    return a * b;
}; // Note the semi colan should always be placed after the function

console.log(multiply(5, 10)); // outputs 50
```

In the above example, the function is named. It can also be anonymous, in which case the name property will be a blank string.

```
multiply = function (a, b) {
   return a * b;
}; // Note the semi colan should always be placed after the function

console.log(multiply(5, 10)); // outputs 50
```

Functional Inheritance

Functional inheritance is mechanism of inheritance that provides encapsulation by using closures. Before trying to understand the syntax, take an example first. Suppose we want to represent planets in the solar system. We decided to have a *planet* base object and then several planet child objects which inherit from the base object. Here is the base planet object:

```
var planet = function(spec) {
    var that = {};
    that.getName = function() {
        return spec.radius;
    };
    that.getNumberOfMoons()= function() {
        return spec.numberOfMoons;
    };
    return that;
}
```

Now for some planets. Let's start with Earth and Jupiter and to amuse ourselves let's add a function for Earth for people to leave and a function to Jupiter for people arriving. Sarah Palin has taken over and things have got pretty bad!!!

```
var earth = function(spec) {
    var that = planet(spec);   // No need for new keyword!
    that.peopleLeave = function() {
        // ... people leave
    }
    return that;
}
```

```
var jupiter = function(spec) {
    var that = planet(spec);
    that.peopleArrive = function() {
       // .. people arrive
    }
    return that;
}
```

Now put the earth and jupiter in motion...

```
var myEarth = earth({name:"earth",numberofmoons:1});
var myjupiter=jupiter({name:"jupiter",numberofmoons:66});
```

The three key points here:

1. There is code reuse.
2. There is encapsulation. The name and numberOfMoons properties are encapsulated.
3. The child objects can add in their own specific functionality.

Now an explanation of the syntax:

1. The base object planet accepts some data in the *spec* object.
2. The base object planet creates a closures called *that* which is returned. The *that* object has access to everything in the spec object. *But*, nothing else does. This provides a layer of encapsulation.
3. The child objects, *earth* and *jupiter*, set up their own data and pass it to base planet object.
4. The planet object returns a closure which contains base functionality. The child classes receive this closure and add further methods and variables to it.

Hoisting

No matter where var's are declared in a function, javascript will "hoist" them meaning that they behave as if they were declared at the top of the function.

```
mylocation = "dublin"; // global variable
function outputPosition() {
    console.log(mylocation);  // outputs "undefined" not "dublin"
    var mylocation = "fingal" ;
    console.log(mylocation);  // outputs "fingal"
}
outputPosition();
```

In the function above, the var declaration in the function means that the first log will "see" the mylocation in the function scope and not the one declared in the global scope. After declaration, the local mylocation var will have the value "undefined", hence why this is outputted first. Functions that are assigned to variables can also be hoisted. The only difference being that when functions are hoisted, their definitions also are - not just their declarations.

Immediate Function Expressions

Immediate function expression are executed as soon as they are defined.

```
(function() {

    console.log("I ain't waiting around");

}());
```

There are two aspects of the syntax to note here. Firsty, there is a () immediately after the function definiton, this makes it execute. Secondly, the function can only execute if it is a function expression as opposed to a function declaration. The outer () make the function an expression. Another way to define a an immediate function expression is:

```
var anotherWay = function() {
    console.log("I ain't waiting around");
}()
```

JSON

JavaScript Object Notation (JSON) is a notation used to represent objects. It is very similar to the format used for Javascript Object literals except the property names must be wrapped in quotes. The JSON format is not exclusive to javascript; it can be used by any language (Python, Ruby etc). JSON makes it very easy to see what's an array and what's an object. In XML this would be much harder. An external document - such as XSD - would have to be consulted. In this example, Mitt Romney has an array describing who might vore for him and an object which is his son.

```
{"name": "Mitt Romney", "party": "republicans", "scary": "of course", "
```

Loose typing

Javascript is loosely typed. This means that variables do not need to be typed. It also means there is no complex class hierarchies and there is no casting.

```
var number1 = 50;
var number2 = "51";

function output(varToOutput) {
    // function does not care about what type the parameter passed is.
    console.log(varToOutput);
}
output(number1);  // outputs 50
output(number2);  // outputs 51
```

Memoization

Memoization is a mechanism whereby functions can cache data from previous executions.

```
function myFunc(param){
    if (!myFunc.cache) {
        myFunc.cache = {}; // If the cache doesn't exist, create it.
    }
    if (!myFunc.cache[param]) {
        //... Imagine the code to work out result below
        // is computationally intensive.
        var result = {
            //...
        };
        myFunc.cache[param] = result;  // now result is cached.
    }
    return myFunc.cache[param];
}
```

Method

When a function is stored as a property of an object, it is referred to as a method.

```
var myObject {
    myProperty: function () {
       //...
       // the this keyword in here will refer to the myObject instance.
       // This means the "method" can read and change variables in the
       // object.
    }
}
```

Modules

The goal of modules is to enable javascript code bases to more modular. Functions and variables are collated into a module and then the module can decide what functions *and* what variables the outside world can see - in the same way as encapsulations works in the object orientated paradigms. In javascript we create modules by combining characteristics of closures and immediate function expressions.

```
var bankAccountModule = (function moduleScope() {
    var balance = 0; //private
    function doSomethingPrivate(){  // private method
        //...
    }
    return { //exposed to public
        addMoney: function(money) {
            //...
        },
        withDrawMoney: function(money) {
            //...
```

```
        },
        getBalance: function() {
            return balance;
    }
}());
```

In the example above, we have a bank account module:

- The function expression *moduleScope* has its own scope. The private variable balance and the private function doSomethingPrivate, exist only within this scope and are *only* visible to functions within this scope.
- The moduleScope function returns an object literal. This is a closure which has access to the private variables and functions of moduleScope. The returned object's properties are "public" and accesible to the outside world.
- The returned object is automatically assigned to bankAccountModule
- The immediate function ()) syntax is used. This means that the module is initialised immediately.

Because the returned object (the closure) is assigned to *bankAccountModule*, it means we can access the *bankAccountModule* as:

```
bankAccountModule.addMoney(20);
bankAccoumtModule.withdrawMoney(15);
```

By convention, the filename of a module should match its namespace. So in this example, the filename should be bankAccountModule.js.

Namespace Pattern

Javascript doesn't have namespaces built into the language, meaning it is easy for variables to clash. Unless variables are defined in a function, they are considered global. However, it is possible to use "." in variables names. Meaning you can pretend you have name spaces.

```
DUBLINTECH.myName = "Alex"
DUBLINTECH.myAddress = "Dublin"
```

Object Literal Notation

In javascript you can define an object as collection of name value pairs. The values can be property values or functions.

```
var ireland = {
    capital: "Dublin",
    getCapital: function () {
        return this.capital;
    }
};
```

Prototype properties (inheritance)

Every object has a prototype object. It is useful when you want to add a property to all instances of a particular object. Suppose you have a constructor function, which representent Irish people who bought in the boom.

```
function IrishPersonBoughtInTheBoom(){
}

var mary = new IrishPersonBoughtInTheBoom ();
var tony = new IrishPersonBoughtInTheBoom ();
var peter = new IrishPersonBoughtInTheBoom ();
...
```

Now, the Irish economy goes belly up, the property bubble explodes and you want to add a debt property to all instances of this function. To do this you would do:

```
IrishPersonBoughtInTheBoom.prototype.debt = "ouch";
```

Then...

```
console.log(mary.debt);   // outputs "ouch"
console.log(tony.debt);   // outputs "ouch"
console.log(peter.debt);   // outputs "ouch"
```

Now, when this approach is used, all instances of IrishPersonBoughtInTheBoom share the save copy of the debt property. This means, that they all have the same value as illustrated in this example.

Returning functions

A function always returns a value. If return is not specified for a function, the *undefined* value type will be returned. Javascript functions can also return some data or another function.

```
var counter = function() {
    //...
    var count = 0;
    return function () {
        return count = count + 1;
    }
}

var nextValue = counter();
nextValue();   // outputs 1
nextValue();   // outputs 2
```

Note, in this case the inner function which is returned "*closes*" over the count variable - making it a *closure* - since it encapsulates its own count variable. This means it gets its own copy which is *different* to the variable return by *nextValue.count.*

this keyword

The *this* keyword in Java has different meanings, depending on the context it is used. In summary:

- In a method context, *this* refers to the object that contains the method.
- In a function context, *this* refers to the global object. Unless the function is a property of another object. In which case the *this* refers to that object.
- If *this* is used in a constructor, the *this* in the constructor function refers to the object which uses the constructor function.
- When the *apply* or *call* methods are used the value of *this* refers to what was explictly specified in the apply or call invocation.

typeof

typeof is a unary operator with one operand. It is used to determine the types of things (a bit like getClass() in Java). The values outputted by typeof are "number", "string", "boolean", "undefined", "function", "object".

```
console.log(typeof "tony");          // outputs string
console.log(typeof 6);               // outputs number
console.log(typeof false);                  // outputs boolean
console.log(typeof this.doesNotExist);   // outputs undefined if the glob
console.log(typeof function(){});    // outputs function
console.log(typeof {name:"I am an object"});  //outputs object
console.log(typeof ["I am an array"]) // typedef outputs object for array
console.log(typeof null)              // typedef outputs object for nulls
```

Some implementations return "object" for typeof for regular expressions; others return "function". But the biggest problem with typeof is that it returns object for null. To test for null, use strict equality...

```
if (myobject === null) {
    ...
}
```

Self-redefining functions

This is a good performance technique. Suppose you have a function and the first time it is called you want it to perform some set up code that you never want to perfom again. You can execute the set up code and then make the function redefine itself after that so that the setup code is never re-excuted.

```
var myFunction = function () {
    //set up code only to this once
    alert("set up, only called once");

    // set up code now complete.
    // redefine function so that set up code is not re-executed
    myFunction = function() {
         alert("no set up code");
    }
}
myFunction();  // outputs - Set up, only called once
myFunction();  // outputs - no set up code this time
myFunction();  // outputs - no set up code this time
```

Note, any properties added to the set up part of this function will be lost when the function redefines itself. In addition, if this function is used with a different name (i.e. it is assigned to a variable), the re-definition will not happen and the set up code will re-execute.

Scope

In javascript there is a global scope and a function scope available for variables. The *var* keyword does not need to be used to define variable in the global scope but it must be used to define variable in the local function scope. When a variable is scoped to a local function shares the name with a global variable, the local scope takes precedence - unless var was not used to declare the local variable in which case any local references are pointing to the global reference. There is no block scope in javascript. By block we mean the code between {}, aka curly braces.

```
var myFunction = function () {
var noBlockScope = function ( ) {
    if (true) {
        // you'd think that d would only be visible to this if statement
        var d = 24;
    }
    if (true) {
        // this if statement can see the variable defined in the other i
        console.log(d);
    }
}
noBlockScope();
```

Single var pattern

You can define all variables used by a function in one place. It is ensures tidy code and is considered best practise.

```
function scrum() {
    var numberOfProps = 2,
        numberOfHookers = 1,
```

```
        numberOfSecondRows = 2,
        numberOfBackRow = 3
    // function body...
}
```

If a variable is declared but not initialized with a value it will have the value *undefined.*

Strict Equality

In javascript it is possible to compare two objects using ==. However, in some cases this will perform type conversion which can yield unexpected equality matches. To ensure there is strict comparison (i.e. no type conversions) use the === syntax.

```
console.log(1 == true)    // outputs true
console.log(1 === true)   // outputs false
console.log(45 == "45")   // outputs true
console.log(45 === "45")  // outputs false
```

Truthy and Falsey

When javascript expects a boolean, you may specify a value of any type. Values that convert to true are said to be truthy and values that convert to false are said to be falsey. Example of truthy values are objects, arrays, functions, strings and numbers:

```
// This will output 'Wow, they were all true'
if ({} && {sillyproperty:"sillyvalue"} && [] &&
        ['element'] && function() {} && "string" && 89) {
   console.log("wow, they were all true");
}
```

Examples of falsey values are empty strings, undefined, null and the value 0.

```
// This will out put: 'none of them were true'
if (!("" || undefined || null || 0)) {
    console.log("none of them were true");
}
```

Undefined and null

In javascript, the undefined value means not initialised or unknown where null means an absence of a value.

References

1. JavaScript patterns Stoyan Stefanov
2. JavaScript, The Definitive Guide David Flanagan
3. JavaScript, The Good Parts Doug Crockford.