Table of Contents

1 – Reserved Words :

break	case	catch	continue
debugger	default	else	export
extends	for	function	if
instanceof	new	null	super
switch	this	try	typeof
var	abstract	boolean	byte
double	enum	final	implements
Int	interface	package	private
protected	set	short	static
ushort	void	assert	ensure
event	namespace	native	require
transient	user	volatile	

2 – Data Types : var x : DataType = initialValue;

var boolean Test1 = true; Boolean Test = false; // Takes values true or false
var byte Test2 = 0; // Represents integer in the Range[0 to 255]
var char Test3 = 'A' // Stored as 2 byte unicode
var decimal // Numbers as large as 7.9E+28 (positive or negative)
var double // Numbers as large as 1.79E+308 (positive or negative) and as small as 1E-323
var float // Numbers as large as 3.4E+38 (positive or negative) and as small as 1E-44
var int // Range from negative 2,147,483,648 to positive 2,147,483,647
var long // range from negative 9.2E+18 through 9.2E+18
var Number // Numbers as large as 1.79E+308 (positive or negative)
var sbyte // Range from negative 128 to positive 127,
var short // range from negative 32,768 to positive 32,767,
var test = "This is a string" // String
var uint // range from 0 to 4,294,967,295, inclusive
var ulong // range from 0 through about 1.8E+19.
var ushort // range from 0 to 65,535, inclusive.

var person = {name:'John',age:23};

person.name = "Henry";

person['age'] = 25;

3 – Objects :

ActiveXObject	arguments	Array	Boolean	Date	Enumerator
Error	Function	Global	Math	Number	Object
RegExp	String	VBArray			

4 – Operators :

There are three literals false, null and true.

+=	+	=	&=	&	<<	~	|=	|	>>
^=	^	,	= =	>	> =	= = =	! =	<	< =
! = =	?:	/ =	/	delete	in	+ +	- -	instanceof	<<=
&&	!	||	% =	%	* =	new	&	>>=	- =
-	typeof	> > >	void						

5 – Methods :

fromCharCode	abs	acos	anchor	apply	asin	atan
encodeURIComponent	atan2	atEnd	big	blink	bold	call
dimensions	ceil	charAt	charCodeAt	compile	concat	cos
decodeURIComponent	decodeURI	escape	eval	exec	exp	
	fixed	floor	fontcolor	fontsize	getDate	getDay
	getFullYear	getHours	getItem	getMilliseconds	getMinutes	getMonth
getTimezoneOffset	getSeconds	getTime	getUTCDate	getUTCDay	getVarDate	getYear
hasOwnProperty	indexOf	isFinite	isNan	italics	item	
isPrototypeOf	item	join	lastIndexOf	lbound	link	log
localCompare	match	max	min	moveFirst	moveNext	parse
parseFloat	parseInt	pop	pow	push	random	replace
reverse	round	search	setDate	setFullYear	setHours	shift
sin	slice	small	sort	splice	split	sqrt
strike	sub	substr	substring	sup	tan	test
toArray	toDateString	toFixed	toGMTString	toLowerCase	toPrecision	toString
toTimeString	toUpperCase	ubound	unescape	unshift	UTC	valueOf

6 – Modifiers :

abstract	expando	final	hide	internal	override
private	protected	public			

7 – Properties :

arguments	callee	caller	constructor	description	lastIndex
global	ignoreCase	index	Infinity	Input	lastMatch
lastParen	leftContext	length	LN10	LN2	LOG2E
MAX_VALUE	message	MIN_VALUE	multiline	name	NaN
number	PI	propertyIsEnurable	prototype	rightContext	
source	SQRT1_2	SQRT2	Undefined		

8 – Statements :

break	class	@cc_on	Comment	const	continue
debugger	do while	enum	for	for in	function
function get	function set	@if @elif	@else @end	import	Interface
Labeled	package	print	return	@set	static
super	switch	this	try catch finally	var	while
with					

9 – Directives :

@set @debug(on | off) // Turns the emission of debug symbols on or off
@set @position(line = 1) // Provides meaning position in error messages

10 – Definitions :

Here are some javascript definitions:

- Class – Defines the characteristics of the Object.
- Object – An Instance of a Class.
- Property – An Object characteristic, such as color.
- Method – An Object capability, such as walk.
- Constructor – A method called at the moment of instantiation.
- Inheritance – A Class can inherit characteristics from another Class.
- Encapsulation – A Class defines only the characteristics of the Object, a method defines only how the method executes.
- Abstraction – The conjunction of complex inheritance, methods, properties of an Object must be able to simulate a reality model.
- Polymorphism – Different Classes might define the same method or property.

11 – Code Snippets :

It's important to note that there are no classes in JavaScript. Functions can be used to somewhat simulate classes, but in general JavaScript is a class-less language. Everything is an object. And when it comes to inheritance, objects inherit from objects, not classes from classes as in the "class"-ical languages.

```
// First Javascript program
<html>
<body>
<script language="javascript" type="text/javascript">
      document.write("Hello World")
</script>
</body>
</html>
```

```
// Create an empty object using any one of the methods below
var empty = {}; var empty2 = new object();
```

```
// Create an empty array using any one of the methods below
var matrix = []; var matrix = new array();
```

```
// Define an empty class
function Person() { }
var person1 = new Person();
```

```
// Define a simple class using prototype
function Person(gender) {
  this.gender = gender;
}

Person.prototype.gender = '';

Person.prototype.sayHello = function () {
  alert ('hello');
};

var person1 = new Person('Male');

// call the Person sayHello method.
```

```
person1.sayHello(); // hello
```

```
// Method 1 - Define a class using a function
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = getAppleInfo;
}

function getAppleInfo() {
    return this.color + ' ' + this.type + ' apple';
}

var apple = new Apple('macintosh');
apple.color = "reddish";
alert(apple.getInfo());
```

```
// Method 2 - Methods defined inline
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

```
// Method 3 - Methods added to the prototype
function Apple (type) {
    this.type = type;
    this.color = "red";
}

Apple.prototype.getInfo = function() {
    return this.color + ' ' + this.type + ' apple';
};
```

```
// Creating and using an Interface
var Interface = function (objectName, methods) {
  // Check that the right amount of arguments are provided
  if (arguments.length != 2) {
        throw new Error ("Invalid arguments" + arguments.length");
        }
```

```
  // Create the public properties
  this.name = objectName;
  this.methods = [];

  // Loop through provided arguments and add them to the 'methods' array
  for (var i = 0, len = methods.length; i < len; i++) {
  // Check the method name provided is written as a String
  if (typeof methods[i] !== 'string') {
    throw new Error ("Methods should be of type string");
  }

  // If all is as required then add the provided method name to the method array
  this.methods.push(methods[i]);
  }
};

var test = new Interface('test', ['details', 'age']);

var properties = {
  name: "Mark McDonnell",
  actions: {
            details: function() {
                  return "I am " + this.age() + " years old.";
                  },
            age: (function(birthdate) {
                  var dob = new Date(birthdate),
                  today = new Date(),
                  ms = today.valueOf() - dob.valueOf(),
                  minutes = ms / 1000 / 60,
                  hours = minutes / 60,
                  days = hours / 24,
                  years = days / 365,
                  age = Math.floor(years)
                  return function() { return age; };
                  })("1981 08 30")
         }
};

// Create a Person constructor that will implement the above properties/methods
function Person(config) {
  // Pass in the methods we are expecting,
  // followed by the name of the Interface instance that we're checking against
  Interface.ensureImplements(config.actions, test);
  this.name = config.name;
  this.methods = config.actions;
}

// Create a new instance of the Person constructor...
var me = new Person(properties);
```

```
// ...and make sure the methods are working
alert(me.methods.age());
alert(me.method.details());
```

```
// Using an Abstract class
abstract class Car {
     abstract function Drive( );
}
class Honda extends Car {
     function Drive("I am driving");
}
```

```
// Extending a class
var Car = Class.Extend({
  setColor: function(clr){
    color = clr;
  }
});

var volvo = Car.Extend({
   getColor: function () {
      return color;
   }
});
```

```
// Using expando
expando class CExpandoExample {
   var x : int = 10;
}

// New expando class-based object.
var testClass : CExpandoExample = new CExpandoExample;
// New JScript Object.
var testObject : Object = new Object;

// Add expando properties to both objects.
testClass["x"] = "ten";
testObject["x"] = "twelve";

// Access the field of the class-based object.
print(testClass.x);      // Prints 10.
// Access the expando property.
print(testClass["x"]);   // Prints ten.
```

```
// Access the property of the class-based object.
print(testObject.x);     // Prints twelve.
// Access the same property using the [] operator.
print(testObject["x"]);  // Prints twelve.
```

```
// Using final
class CBase {
   final function methodA() { print("Final methodA of CBase.") };
   function methodB() { print("Non-final methodB of CBase.") };
}

class CDerived extends CBase {
   function methodA() { print("methodA of CDerived.") };
   function methodB() { print("methodB of CDerived.") };
}

var baseInstance : CBase = new CDerived;
baseInstance.methodA();
baseInstance.methodB();
```

```
class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();

class CBase {
   function methodA() { print("methodA of CBase.") };
   function methodB() { print("methodB of CBase.") };
}

class CDerived extends CBase {
```

```
   hide function methodA() { print("Hiding methodA.") };
   override function methodB() { print("Overriding methodB.") };
}


var derivedInstance : CDerived = new CDerived;
derivedInstance.methodA();
derivedInstance.methodB();

var baseInstance : CBase = derivedInstance;
baseInstance.methodA();
baseInstance.methodB();
```

```
// Using static
class CTest {
   var nonstaticX : int;        // A non-static field belonging to a class
instance.
   static var staticX : int;  // A static field belonging to the class.
}

// Initialize staticX. An instance of test is not needed.
CTest.staticX = 42;

// Create an instance of test class.
var a : CTest = new CTest;
a.nonstaticX = 5;
// The static field is not directly accessible from the class instance.

print(a.nonstaticX);
print(CTest.staticX);
```

```
// Using typeof
var index = 5;
var result = (typeof index === 'number');
```

Appendix :

Links for further reading:
http://msdn.microsoft.com/en-us/library/ye921ye4(v=vs.100).aspx

http://www.tutorialspoint.com/javascript/javascript_variables.htm

http://www.phpied.com/3-ways-to-define-a-javascript-class/

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript