JS – The language of the web browser  
  
**Introduction:**

* Motivation: A language for the web browser.
* Good parts : functions, looase typing, dynamic objects, untyped, interpreted and a vey expressive object literal notion.
* Bad pats: programming model with gloabl variable.
* JS support *prototypicall inheritance*. It has a class-free object system in which objects inherit properties directly from other objects.
* JS depends on global variable for linkage.
* The standard that describes JS is ECMAScript programming language.  
  JSLing is a popular JS parser.
* Why JS is important – 1. It is the only language supported by all the web browser. 2. It is light weight & expressive.
* A very basic comparision between java and javaScript

|  |  |
| --- | --- |
| **Java** | **JavaScript** |
| Strongly-typed | Loosely-typed |
| Static | Dynamic |
| Classical (Class-based) | Prototypical |
| Classes | Functions |
| Constructors | Functions |
| Methods | Functions |

* Strict mode in javascript

It makes it easier to write secure 'javascript'. Strict mode changes previously accepted "bad syntax/parts" into real errors.

Ex-

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

**Grammer:**

* Whitespace is usually insignificant but used to improve the readiblity.
* Comments: Block comment /\*\*/ and line-ending comment //  
  Names: A letter optionally followed by one or more letters, digits, or underscore.
* Keywords:

|  |
| --- |
| Abstract boolean break byte case catch char class const continue  debugger default delete do double else enum export extends  false final finally float for function goto if implements import in instanceof int interface long native new null package private protected public return short static super switch synchronized this throw throws transient true try typeof var volatile void while with |

* Numbers: single number type – internally represented as 64-bit floating point.
* NaN is a number value that is the result of an operation that cannot produce a normal result. It can be detected using isNaN (number) function.
* *Infinity* represent all values larger than 1.79769313486231570e+308
* Operators

|  |
| --- |
| delete new typeof + - ! Unary operators  \* / % Multiplication, division, modulo  + - Addition/concatenation, subtraction  >= <= > < Inequality  === !== Equality  && Logical and  || Logical or  ?: Ternary |

**Objects:**

* Simple types : numbers, strings, bolleans (true and false), null, undefined.
* Object types : array, function, regular expression and object
* Standard built-in objects :

Array, Boolean, Date, Error, Function, JSON, Math, Number, Object, RegExp, String, Map, Set, WeakMap , WeakSet and more.

* Objects are class-free (meaning not associated with any class). The **prototype linkage feature** allows one object to inherit the properties of another. This features reduces object initialization time and memory consumption.
* Object literal: The **object LITERAL notion** provides a very convenient notion for creating new object values. Also know as JSON. An object literal is a pair of curly braces **{}** sorrounding zero or more name/value pairs.  
  Ex -   
   var stooge = {  
   “first-name”: ”Jerom”,  
   “last-name” : “Howard”  
   };  
  Objects can nest:  
   var flight = {

airline: "Oceanic",

number: 815,

departure: {

IATA: "SYD",

time: "2004-09-22 14:55",

city: "Sydney"

},

arrival: {

IATA: "LAX",

time: "2004-09-23 10:42",

city: "Los Angeles"

}

};

* Retrieval: Values can be retrieved from an object by wrapping a string expression in a [ ] suffix:  
  Ex -  
   stooge["first-name"]
* Attempt to retreive value from undefined will throw TypeError exception.  
   flight.equipmentor”
* Object update:  
  A value in an object can be updated by assignment. If the property name already

exists in the object, the property value is replaced:

stooge['first-name'] = 'Jerome';

If the object does not already have that property name, the object is augmented

* Passy by reference: Object are passed by reference. They are never copied.
* Prototype:  
  Every object is linked to a prototype object from which it can inherit properties. All objects created from object literals are linked to **Object.prototype**, an object that comes standard with JS. When you make a new object, you can select the object that should be its prototype.
* Delegation: The prototype link is used only in retreival. If we try to retreive a property value from an object, and if the object lacks the property name, then JS attempts to retreive the property value from the property object. An if that object is lacking the property, then it goes its property, and so on until the process finally bottoms out with Object.prototype. If the desired property exists nowhere in the prototype chain, then the result is the undefined value. This is called delegation. The prototype relationship is a dynamic relationship. If we add a new property to a prototype, that property will immediately be visible in all of the objects that are based on that protytype.
* Reflection:  
  The typedef operator and hasOwnProperty method can be used inspect an object.
* Enumeration: The for in statement can loop over all the property names in an object. The enumeration will include all the names including functions and prototype properties.
* Delete operator: The delete operator can be used to remove a proprty from an object. It will not touch any of the objects in the prototype linkage.
* Global abatement:  
  Glabal variables can be used to hold the gloabl information of the application. But global variables need to be used with caution. Best way to use global variable is to hold all the information in a single global variable.
* Function:  
  Funtions are first class objects, because they can have properties and methods just like any other object. Functions objects are linked to Function.prototype, which in turn link to Object.prototype
* Function LITERAL: Function objects are created with **function literals**. A function literal has four parts.  
  Part1 – The reserved word function.  
  Part2 – Function name. If name is not provided, it becomes anonymous.  
  Part3 – Set of parameters of the functions wrapped in a paranthesis.  
  Part4 – A set of stamement wrapped in a pair of curly baraces.
* Closure: Function can be defined inside of an other function. An inner function has access to the parameters and variables (with the exception of *this* and *arguments*) of the functions it is nested within. The function object created by a function literal contains a link to that outer context. This is called closure.
* Function invocation:   
  When a function is invoked along with its decalred parameters it receives two additional parameter: *this* and *argumets*. There are four patterns using which a function can be invoked – method invocation pattern, function invocation pattern, constructor invocation pattern, apply invocation pattern.
* The *this* object:

The value of this is determined by *the context in which a function is called*. It cann't be set by assignment during execution, and it may be different each time the function is called.

|  |  |
| --- | --- |
| Global Context | Outside of any function, in the global execution context, *this* referes to the global object, whether in the strict mode or not.  Ex-  this.a = 37;  console.log(window.a); // 37 |
| Function Context | Inside a function, the value of *this* depends on how the function is called.  Simple case:  function f1(){  return this;  }  f1() === window; // global object  In this case, the value of this is not set by the call. Since the code is not in strict mode, the value of this must always be an object so it defaults to the global object.  function f2(){  "use strict"; // see strict mode  return this;  }  f2() === undefined;  In strict mode, the value of this remains at whatever it's set to when entering the execution context. If it's not defined, it remains undefined. It can also be set to any value, such as null or 42 or "I am not this".  Arrow Functions:  In arrow functions, this is set lexically, i.e. it's set to the value of the enclosing execution context's this.  As an object method:  When a function is called as a method of an object, its this is set to the object the method is called on.  var o = {  prop: 37,  f: function() {  return this.prop;  }  };  console.log(o.f()); // logs 37  As a constructor:  When a function is used as a constructor (with new keyword), its this is bound to the new object being constructed.  Ex-  function C(){  this.a = 37;  }  var o = new C();  console.log(o.a); // logs 37  Call and apply:  Where a function uses the this keyword in its body, its value can be bound to a particular object in the call using the call or apply methods that all functions inherit from Function.prototype.  The bind method:  ECMAScript 5 introduced Function.prototype.bind. Calling f.bind(someObject) creates a new function with the same body and scope as f, but where this occurs in the original function, in the new function it is permanently bound to the first argument of bind, regardless of how the function is being used.  As a DOM event handler:  When a function is used as an event handler, its this is set to the element the event fired from (some browsers do not follow this convention for listeners added dynamically with methods other than addEventListener).  In an in-line event handler:  When code is called from an in–line handler, its this is set to the DOM element on which the listener is placed. |

* Method invocation pattern:

When a *function is stored as a property of an object* we call it a method. The dot (.)

operator is used for method invocation.

|  |
| --- |
| /\*  Create myObject. It has a value and an increment method. The increment method takes an optional parameter. If the argument is not a number, then 1  is used as the default.  \*/  var myObject = {  value: 0,  increment: function (inc) {  this.value += typeof inc === 'number' ? inc : 1;  }  };  myObject.increment( ); // 1  document.writeln(myObject.value);  myObject.increment(2); // 3  document.writeln(myObject.value); |

Note that the binding of *this* to the object happens at invocation time. This **very late binding** of *this* to the object happens at invocation time. This very late binding makes function that uses *this* very reusable.

* Function invocation pattern

When a function is not a property of an object, then it is invoked as a function.

Caution: When a function is invoked with this pattern, *this* is bound to the global object. If an innter function is invoked, the *this* it still bound to the global object whereas as the this is expected to be bound to the outer function !

Following is an work arond to that problem:

|  |
| --- |
| // Augment myObject with a double method.  myObject.double = function ( ) {  var that = this;  // Workaround.  var helper = function ( ) {  that.value = add(that.value, that.value);  };  helper( );  // Invoke helper as a function.  };  // Invoke double as a method.  myObject.double( );  document.writeln(myObject.getValue( ));  // 6 |

* Constructor invocation pattern

The new operator is used. If a function is invoked with the *new* prefix, then a new object will be created with a hidden link to the value of the function's prototype member, and this will be bound to that new object.

Example:

|  |
| --- |
| // Create a constructor function called Quo.  // It makes an object with a status property.  var Quo = function (string) {  this.status = string;  };  // Give all instances of Quo a public method  / called get\_status.  Quo.prototype.get\_status = function ( ) {  return this.status;  };  // Make an instance of Quo.  var myQuo = new Quo("confused"); // Here Quo serves as constructor  document.writeln(myQuo.get\_status( ));  // confused |

* Apply invocation pattern:

Javascript is a functional object-oriented language. In otherwords functions can have methods.

The apply method let us construct an array of arguments to use to invoke a function. It also let us choose the value of *this*. The apply method takes two parameters. First, value that should be bound to this. Second is an array of parameters.

Example:

|  |
| --- |
| // Make an array of 2 numbers and add them.  var array = [3, 4];  var sum = add.apply(null, array);  // sum is 7  // Make an object with a status member.  var statusObject = {  status: 'A-OK'  };  /\*  statusObject does not inherit from Quo.prototype, but we can invoke the get\_status method on statusObject even though statusObject does not have  a get\_status method.  \*/  var status = Quo.prototype.get\_status.apply(statusObject);  // status is 'A-OK' |

* Arguments: It makes it possible to write a function that takes an unspecified number of parameters. When a function is invoked it is supplied with an array called *argument* containing all the supplied parameters.
* Return : The return function can be used to cauase a function to return early.
* Exceptions:

|  |
| --- |
| // Throwing exception  var add = function (a, b) {  if (typeof a !== 'number' || typeof b !== 'number') {  **throw** {  name: 'TypeError',  message: 'add needs numbers'  };  }  return a + b;  }  // Handling exception using try-catch  // Make a try\_it function that calls the new add function incorrectly.  var try\_it = function ( ) {  **try** {  add("seven");  } **catch** (e) {  document.writeln(e.name + ': ' + e.message);  }  }  try\_it(); |

* Javascript allows the basic types of the language to be augmented. For example, adding a method to Object.prototype makes that method available to all objects.

This concept alos works for function,arrays,string,numbers,regular expressions, and boolans.

For example, by augmenting Function.prototype, we can make a method available to all functions.

* Recursion : Javascript supports recursion.
* Scope: Controlles the visibility and lifetimes of variables and parameters.

Does not have block scope but have function scope.

* Callback : It is easy to implement the callbacks.

Ex-netconf

|  |  |
| --- | --- |
| Naive way of handling callback:  request = prepare\_the\_request( );  response = send\_request\_synchronously(request);  display(response); | JS way of handling callback:  request = prepare\_the\_request( );  send\_request\_asynchronously(request, function (response) {  display(response);  }); |

* Module: The function and closure can be used to create module.
* Cascade:

Some methods do not have return values.

Ex – It is typical for methods that set or change the state of an object to return nothing. It we those methods return *this* instead of *undefined*. This will enable cascading.

In cascade we call many methods on the same object in sequence in a single statement.

Ex-

|  |
| --- |
| getElement('myBoxDiv').  move(350, 150).  width(100).  height(100).  color('red').  border('10px outset').  padding('4px').  appendText("Please stand by").  on('mousedown', function (m) {  this.startDrag(m, this.getNinth(m));  }).  on('mousemove', 'drag').  on('mouseup', 'stopDrag').  later(2000, function ( ) {  this.  color('yellow').  setHTML("What hath God wraught?").  slide(400, 40, 200, 200);  }).  tip('This box is resizeable'); |

* Currying : Functions are values, and we can manipulate function values in interesting ways.
* Memoization: Functions can use objects/arrays to remember the results of previous operations, making it possible to avoid unnecessary work. This optimization is called momoization.

**Inheritance:**

* JS provides a richer set of code resusable pattern. The set of possible code reuse pattern in Js is vast.
* In calssical languages, objects are instances of classes, a class can inherit from another class. JS is a prototypical language, which means that objects inherit directly from other object. The heritance is achieved through prototype chaining.
* Prototype chaining: Each object has an internal link to another object called it prototype. That prototype object has a prototype of its own, and so on until object is reached with *null* at its prototype. Null, by defination has no prototype, and acts as the final link in this prototype chain.
* Inheritance with prototype chaining:

Inheriting properties -

JS objects are dynamic “bag” of properties (referred to as *own properties*). JS objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototypes, and so on until either a property with a matching name is found or the end of the prototype chain is reached.

Inheriting methods -

JS does not have “method” in the form that class-based languages define them. In JS, any function can be added to an object in the form of a property. An inherited function acts just as any other property, including property shadowing. When an inherited function is executed, the value of *this* points to the inheriting object, not to the prototype object where the function is an own property.

* Different ways to create objects and the resulting prototype chain:

# Object created with syntax constructs:

|  |
| --- |
| var o = {a: 1};  // The newly created object o has Object.prototype as its [[Prototype]]  // o has no own property named 'hasOwnProperty'  // hasOwnProperty is an own property of Object.prototype.  // So o inherits hasOwnProperty from Object.prototype  // Object.prototype has null as its prototype.  // o ---> Object.prototype ---> null  var a = ["yo", "whadup", "?"];  // Arrays inherit from Array.prototype  // (which has methods like indexOf, forEach, etc.)  // The prototype chain looks like:  // a ---> Array.prototype ---> Object.prototype ---> null  function f(){  return 2;  }  // Functions inherit from Function.prototype  // (which has methods like call, bind, etc.)  // f ---> Function.prototype ---> Object.prototype ---> null |

# With a constructor

A constructor in JS is “just” a function that happens to be called with the *new* operator.

|  |
| --- |
| function Graph() {  this.vertices = [];  this.edges = [];  }  //  Graph.prototype = {  addVertex: function(v){  this.vertices.push(v);  }  };  var g = new Graph();  // g is an object with own properties 'vertices' and 'edges'.  // g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed. |

# With Object.create()

ECMAScript 5 introduced a new method: Object.create(). Calling this method creates a new object. The prototype of this object is the first argument of the function:

|  |
| --- |
| var a = {a: 1};  // a ---> Object.prototype ---> null  var b = Object.create(a);  // b ---> a ---> Object.prototype ---> null  console.log(b.a); // 1 (inherited)  var c = Object.create(b);  // c ---> b ---> a ---> Object.prototype ---> null  var d = Object.create(null);  // d ---> null  console.log(d.hasOwnProperty);  // undefined, because d doesn't inherit from Object.prototype |

# With the class keyword

ECMAScript 6 introduced a new set of keywords implementing classes. Although these

constructs look like those familiar to developers of class-based languages, they

are not the same. JavaScript remains prototype-based. The new keywords include

class, constructor, static, extends, and super.

|  |
| --- |
| "use strict";  class Polygon {  constructor(height, width) {  this.height = height;  this.width = width;  }  }  class Square extends Polygon {  constructor(sideLength) {  super(sideLength, sideLength);  }  get area() {  return this.height \* this.width;  }  set sideLength(newLength) {  this.height = newLength;  this.width = newLength;  }  }  var square = new Square(2); |

* Some important inheritance patterns are : Pseudoclassical, Object specifiers, Prototypical, Functional, Parts
* Ref : <https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain>

**Array:**

* Arrays are objects, inherited from the prototype Object.Array, containes the methods required for traversal and mutation operation.

Some command array operations:

|  |
| --- |
| // Create an Array  var fruits = ["Apple", "Banana"];  console.log(fruits.length); // 2  // The equivalent object literal notion  var fruiuts\_object = {  '0': 'Apple', '1': 'Banana'  };  // Acess (index into) an array item  var first = fruits[0];  // Apple  var last = fruits[fruits.length - 1];  // Banana  // Loop over an array  fruits.forEach(function (item, index, array) {  console.log(item, index);  });  // Apple 0  // Banana 1  // Add to the end of an array  var newLength = fruits.push("Orange");  // ["Apple", "Banana", "Orange"]  // Remove from the end of an array  var last = fruits.pop(); // remove Orange (from the end)  // ["Apple", "Banana"];  // Remove from the front of an array  var first = fruits.shift(); // remove Apple from the front  // ["Banana"];  // Add to the front of an array  var newLength = fruits.unshift("Strawberry") // add to the front  // ["Strawberry", "Banana"];  // Find the index of an item in the array  fruits.push("Mango");  // ["Strawberry", "Banana", "Mango"]  var pos = fruits.indexOf("Banana"); // 1  // Remove an item by index position  var removedItem = fruits.splice(pos, 1); // this is how to remove an item  // ["Strawberry", "Mango"]  // Copy an array  var shallowCopy = fruits.slice(); // this is how to make a copy  // ["Strawberry", "Mango"] |

**Regular expression:**

* Todo

**Methods:**

JS includes a small set of standard methods that are available on the standard types.

* Array

**array.concat(item...)**

The concat method produces a new array containing a shallow copy of this array with the items appended to it.

**array.join(separator)**

The join method makes a string from an array.

**array.join(separator)**

The join method makes a string from an array.

**array.pop( )**

The pop and push methods make an array work like a stack.

**array.push(item...)**

The push method appends items to the end of an array.

**array.reverse( )**

The reverse method modifies the array by reversing the order of the elements.

**array.shift( )**

The shift method removes the first element from an array and returns it.

**array.slice(start, end)**

The slice method makes a shallow copy of a portion of an array.

**array.sort(comparefn)**

The sort method sorts the contents of an array in place.

**array.splice(start, deleteCount, item...)**

The splice method removes elements from an array, replacing them with new items.

**array.unshift(item...)**

The unshift method is like the push method except that it shoves the items onto the front of this array instead of at the end.

* Function

**function.apply(thisArg, argArray)**

The apply method invokes a function, passing in the object that will be bound to this and an optional array of arguments.

* Number

**number.toExponential(fractionDigits)**

The toExponential method converts this number to a string in the exponential form.

**number.toFixed(fractionDigits)**

The toFixed method converts this number to a string in the decimal form.

**number.toPrecision(precision)**

The toPrecision method converts this number to a string in the decimal form.

**number.toString(radix)**

The toString method converts this number to a string.

* Object

**object.hasOwnProperty(name)**

The hasOwnProperty method returns true if the object contains a property having the name.

* RegExp

**regexp.exec(string)**

The exec method is the most powerful (and slowest) of the methods that use regular

expressions.

**regexp.test(string)**

The test method is the simplest (and fastest) of the methods that use regular expressions.

* String

**string.charAt(pos)**

The charAt method returns the character at position pos in this string.

**string.charCodeAt(pos)**

The charCodeAt method is the same as charAt except that instead of returning a string, it returns an integer representation of the code point value of the character at position pos in that string.

**string.concat(string...)**

The concat method makes a new string by concatenating other strings together.

**string.indexOf(searchString, position)**

The indexOf method searches for a searchString within a string.

**string.lastIndexOf(searchString, position)**

The lastIndexOf method is like the indexOf method, except that it searches from the end of the string instead of the front:.

**string.localeCompare(that)**

The localCompare method compares two strings.

**string.match(regexp)**

The match method matches a string and a regular expression.

**string.replace(searchValue, replaceValue)**

The replace method does a search and replace operation on this string, producing a new string.

**string.search(regexp)**

The search method is like the indexOf method, except that it takes a regular expression object instead of a string.

**string.slice(start, end)**

The slice method makes a new string by copying a portion of another string.

**string.split(separator, limit)**

The split method creates an array of strings by splitting this string into pieces.

**string.substring(start, end)**

The substring method is the same as the slice method except that it doesn’t handle the adjustment for negative parameters.

**string.toLocaleLowerCase( )**

The toLocaleLowerCase method produces a new string that is made by converting this

string to lowercase using the rules for the locale.

**string.toLocaleUpperCase( )**

The toLocaleUpperCase method produces a new string that is made by converting this

string to uppercase using the rules for the locale.

**string.toLowerCase( )**

The toLowerCase method produces a new string that is made by converting this string to lowercase.

**string.toUpperCase( )**

The toUpperCase method produces a new string that is made by converting this string to uppercase.

**String.fromCharCode(char...)**

The String.fromCharCode function produces a string from a series of numbers.

References:

[1] : Javascript Tutorial

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

[2] : <http://www.w3schools.com/js/>

[3] : Books

a. Javascript good and bad parts by Douglas Crockford