**ObjectScript**

Last Updated: April 14, 2023

Primary Source: https://javascript.info/

Language Version: ECMAScript 2022, Node.js 19.9.0

Major Update: April 2023

Minor Update: None yet.

Additional Notes:

1. What’s JS ?

It’s a language that can be written directly in webpages and almost all modern web browsers can execute it. All that is required to run js code is a JavaScript engine, which is V8 in Chrome/Opera/Edge and Spidermonkey in Firefox.

In browsers, the “script” is parsed then compiled into machine code and executed, and it is heavily optimized.

It is a pretty “safe” language, as it doesn’t have low-level access. Still, the capability of JS itself varies depending on the environment executing it, for browsers JS can manipulate the webpage, interact with web servers, get set cookies, remember “user data” etc. On servers (like in Node.js), it can do other things like File I/O etc.

That said, JS on browser has many limitations imposed to enforce security, such as not being able to see contents of another tab in the browser, no access to OS, strict browser managed access to peripherals, not being able to connect to other domains unless explicitly allowed by both domains, etc. These are not present in JS outside the scripts in webpages.

1. Useful Links
2. MDN for anything JS, it is not the whole ECMA spec sheet but it is the closest to it with practical and concise examples.
3. <https://caniuse.com/> To check what feature is supported in what browser.
4. <https://kangax.github.io/compat-table> Similar
5. Execution

For browsers, any .html file that uses a <script> tag with inline js or external js file as source can execute a js file.

For server-side, or locally, we can use node <filename.js> to execute it using Node.js.

Modern language that is compiled by JIT compilers of the browsers. Doesn’t give access to pointers, memory addresses or OS tools. Also known as ECMAScript since they are the ones implementing new features to the languages.

Everything is either an Object or a derivative of one. Might as well call it ObjectScript rather than JavaScript.

# JS automatically inserts ‘;’ after proper expression ending, through automatic semicolon insertion, but it’s better to put them manually to avoid random errors.

# ‘use strict’: Used to tell browsers to only use modern js which is off by default and has many language specific fixes. Include this string globally or locally for the respective changes.

# Automatic Type Inferring: var, let and const are used.

As a good practice, one variable should only hold value for 1 thing, meaning a numberOfEagles variable should only hold number of eagles, its name should indicate what value it holds and it should hold only that thing and changes to that thing. Number of eagles in a class A should go in numberOfEaglesA and even if the use of this variable is over, number of eagles in class B should go in a different container.

# Var vs Let: Var declared variables are ‘hoisted’, i.e they automatically go to the top of the block. They even pierce through blocks. So for this function,

Function test()

{

Pp=”hello”

If(false)

{

Var pp;

}

Console.log(pp);

}

It works and we can even access pp outside the function. Redeclarations are ignored, so if var pp is declared twice, then the second one is ignored but if it has a value then that is copied to the variable. Let doesn’t work like this, which is a better and well-defined behavior.

# Data types: let and var allow dynamic types, so value types can change for the same variable. All numeric expression in js return either a value or NaN, but not errors. There are 7 primitive types and 1 special type (Object), they are:

Number: Any number, floats included. JS supports ‘e’ (10^9), so 1e9 (1\*10^9) works. The toString(<base>) method returns the number in string in base 2,8,16,36 or default 10. On raw numbers we call any method using ‘..’, so 123..toString(), that’s because the first ‘.’ is for decimal.

BigInt: <number>n. Not supported in IE.

String: “inside here “or ‘here ‘or `here`, the difference is that ` allows ${expr or var or method call etc.} inside it but the other 2 don’t. Immutable by default.

Bool: true false.

Null: null means nothing. Can be stored.

Undefined: means value is not present in a container.

## Object: Unlike C# Objects aren’t a parent type of every type. They are much like a dictionary, but more powerful. Properties (keys) are of 2 types, data properties and accessor properties. The difference between them is elaborated later.

Syntax: let <varname>= new Object(); or let <varname>={ };

Keys are converted to strings but they can be accessed as-is too. That’s why reserved keywords can be keys too.

Let temp= {

Name: “value”,

0: 2,

“long key”: 2000,

[varname]:”val”,

Temp2,

Hello(){…}

Hi: function(){…}

};

We can access the values with either temp.Name or temp[“Name”] or temp[0] (only because ‘0’ as a key is read both as a string and as an int). temp[“Name”] is preferred since “long key” can’t be accessed otherwise. The [varname] allows us to create a key with a variable’s value. Temp2 is shorthand, it means a key with Temp2 name will be created with value stored in Temp2 variable outside. For all other keys, values must be specified or it’s an error. Functions can be stored in the given 2 ways.

<func Name>: function(){…}

<functionname>(){…}

Delete temp[“long key”]; to delete a key

Since keys can store undefined or null, it will return undefined for both keys that have undefined and keys that don’t exist, so it’s better to use ‘in’ keyword to check if value exists in a key.

By default, if all keys are ints or can be converted to ints then they are sorted in ascending order.

Symbol: Symbols are unique identifiers that can be used to define unique values. Their main use is in Objects, a symbol as a key ensures that anyone outside the scope can never access it. For..in loop doesn’t loop over symbol keys but Object.assign() does read them , however even then outside the global scope they can’t be accessed. Symbols aren’t implicitly converted to string, <symbol variable>.toString() needs to be called manually.

Syntax: let <varname>= Symbol(<optional description name in string>);

This symbol is unique, even if another variable with same description is made, it is not == this variable.

If description is specified, then <varname>.description returns the description string otherwise undefined.

Symbol.for(<description>); returns a Symbol(<description>) (from the global Symbol Registry) that is the same internally as the Symbol(<description>) (if it exists, if it doesn’t then it creates it and stores it in the global Symbol Registry)

Symbol.keyFor(<symbol>); returns the description if the symbol exists in the globabl Symbol Registry, if it doesn’t then undefined is returned.

For an object that has symbol keys, it can use Object.getOwnPropertySymbols to get an iterable that only has the Symbol keys. Another method is Reflect.ownKeys(obj) which does the same.

Function: Function data type is used to hold processable blocks. It’s a sub-type of Object. There are 2 ways of creating functions, function declaration and function expression. Function declaration require a function name and are initialized and ‘hoisted’ but expressions are not.

Param types aren’t specified, just names. If a variable in function is declared as the same name as a variable outside the function scope then it hides/shadows it. A function that has no or empty return, is equal to undefined. Nested functions are also supported.

Syntax: function <name> (<params>){…}

Unlike C++, functions can be called before they are declared in the code (given that they are declared and in the same scope), because JS creates functions in an initialization phase and then runs the code.   
Functions can be assigned to variables because they are a type. These are called function expressions and unlike function declarations, these functions are not prepared in initialization phase.

Syntax: let <varname>= function(<params>){…};

Arrow-functions work just like in dart.

Syntax: let <varname>= (<params>) => <expr>; , however they even support multi-line expressions, we need to provide return explicitly in that case.  
Syntax: let <varname>=(<params>)=> {…};

We can create an immediately executing function using function expression (called IIFE, immediately invoked function expressions) in the following ways:

(function(){…})();

(function(){…}());

!function(){…}();

+function(){…}();

Normal function:

Function <name>(){…}

Function expression:

Let pp=function(){…};

Since Function are a type of Object they can be used like Objects.

For example:

Function test(a,b,…rest) {…}

A pre-defined property is length, if we call test.length we will get 2, i.e the length of the parameters (doesn’t account for rest parameter). ‘name’ is yet another property and it returns a ‘contextual name’, i.e compiler fills in this property by context. In this case, test.name will return ‘test’ string, it works even in function expressions (returns variable name). However fails in some situations like let pp=[test()] , pp[0].name will return “”.

We can set our own properties like,

Function test(){

Return Test.count++;

}

Test.count=0;

Test(); //returns 1

Test(); //returns 2 and so on.

A variable named ‘count’ inside test() will not be the same as a property name, they are stored differently.

• NFE: Named Function Expression, these are useful as they allow the function expr to call itself (recursion). The name given to a function expression is not visible outside the function itself.

For ex:

Let hello=function (value){

If(value)

hello(!value);

};

Let welcome=hello;

Hello=null;

Welcome(); will give error as the function it’s calling inside is hello() which doesn’t have any function now.

We can add names to the function expression to allow it to avoid this issue.

Let hello=function hi(value){

If(value)

Hi(!value);

}

Will hence never give that error as it can call itself.

Hi(22); //will throw since hi is not visible outside the function itself.

There’s also ‘new Function(<optional params>,<string>)’; This is different from function object constructor call ( where new <functionName>(<params>) is the syntax). This creates a function expression automatically, fills it’s body with the given string and assigns it to a variable.

For example:

Let f2=new Function(‘a’,’b’,’return a+b’);

f2(1,2); //returns 3 and works. However, this function gets access to only the global LE (global scope) and not any scope in between. That’s because of minifier, a special program that is run to minify JS variable names, this program very useful and often used. Since the <string> body of the function can be taken at runtime, it will be unable to access any local variables because their names would be minified at compile time.

Functions and all Objects have another hidden property set automatically, which is prototype, this property holds another Object and by default has 1 property called constructor which holds a reference to the Function Object. The prototype property is not the same as [[Prototype]] as this is just a property but works to assist the [[Prototype]]. The Object in the prototype is referenced to [[Prototype]] when new <T>() is called, new T() sets the [[Prototype]] of the returned Object to reference the Object in the prototype. So now we have [[Prototype]] Object and all the defined properties in the Function Object.

For example:

Function test(){}

Test.prototype.constructor==test(); returns true;

Let testObj= new test.constructor(); //this is called a function constructor.

We can replace prototype Object and that removes constructor.

T.prototype is also, an Object, hence it also holds a [[Prototype]]. This is set to null by default but if it is assigned then we can link multiple [[Prototype]]s together. This is useful in classes.

• Array: Arrays in JS are created either with [ ] or with new Array(), unlike Number(), new doesn’t create an Object for Arrays. Can hold values of different types.

• Object to primitive conversion: An object can only convert to a number or a string (bools can be calculated using numbers). By default, object has no method for string or numbers so it returns “[Object object]” which for numbers mean NaN, we can overload the primitive conversion. The overloading method must be inside the Object body. There are 3 return methods, for string conversion, for number conversion and for default conversion (i.e the compiler is unsure if it needs to call string or number conversion method, like if obj+obj is called, ‘+’ exists for both strings and numbers so it calls for a default method).

Using 2 ways,

[Symbol.toPrimitive](hint){…}; //here hint can be “string” or it can be “number” or “default”, we have to return either a number or a string from here. There should only be 1 of this method otherwise the last one is used and all previous ones are discarded.

toString(){…}; //this can also be used to return a value, however it can return anything, even nothing in which case undefined is returned.

valueOf(){…}; // this is meant to be used after toString(), toString() will only be called if a string conversion is required or if valueOf() is not defined. Meaning if [Symbol.toPrimitive](hint){…} is not defined, Number(object); will call valueOf() if it exists otherwise toString() will be called as the last option, if even that doesn’t exist then NaN is returned. ValueOf is also called for default.

Priority: [Symbol.toPrimitive](hint){…} then toString() + valueOf(). Meaning if all 3 of these exist, then only the Symbol method is called.

For example:

Let anObj={

Name: “hex”,

Value:0,

[Symbol.toPrimitive](hint){

If (hint==”string”)

Return this.Name;

Return this.Value;

}}

Now if we call anObj+anObj, compiler calls the Symbol method and passes “default” to hint, for which it returns 0.

• Very basic or pure Objects: Objects that have their [[Prototype]] set to null are called such Objects. The don’t inherit any method, no toString(), valueOf() etc. We can create them by using let <varname>=Object.create(null); and now this Object can even store a key named \_\_proto\_\_ which would be a getter/setter by default.

•Shallow Copy: Object.assign(<target object>,<source object>…); copies all values from all the source objects into target object, overwrites value for keys that already exist. However for values that are objects, references are copied. Returns an object with all keys and values, even from the target. So Object.assign() returns a reference to all keys and values. This is also useful for implementing ‘mixins’ , since Object.assin() returns an Object with all properties, we can create a ‘mixin’ Object and define methods in it then use ‘assign’ to get a sum of Objects.

• Reference copy: Object.create(Object.getPrototypeOf(<obj>), Object.getOwnPropertyDescriptors(<obj>)); returns an Object which has reference to all the properties and their descriptors.

• typeof(<varname>): returns type.

• Conversion: By default, JS automatically converts values into the container type. For mathematical operations, values are converted to numbers, for functions that accept strings as parameters, other types are converted to string and so on.

• Manual Conversion:

String: String(<value>)

Number: “6”\*”2” becomes 6\*2 automatically, if it fails then NaN is returned. Number(<value>) for manual conversion.

Undefined or NaN becomes NaN,

Null becomes 0,

True and false become 1 and 0,

Strings are trimmed for spaces and if the resultant string is a number, it is returned, if string is empty/spaces 0 is returned, else NaN is returned.

Boolean: Boolean(<value>),

Undefined, NaN, null, 0, empty string, or no value return false, else true.

•In expressions, the operator precedence goes from left to right / brackets (if any).

For ‘+’ both string and number use the same operator so there are few rules,

If an operand is string and ‘+’ is used the operation is converted to string, then any ‘+’ operators further right in the expression act like string concatenation.

If an operand is string but any other operator such as ‘- ‘or ‘\*\*’ etc. is used then string operand is converted to number.

Any operation with the operators, ‘+’,’- ‘, ‘\*’, ‘/’ or ‘\*\*’ can only return Number, NaN or String. If there is only 1 operand, unary operation on the operand after conversion to number is performed (if that fails, NaN is returned).

In the expression,

String + any operand = String

But for all other operands = Number or NaN (even for unary operations, so +”0” returns number)

For example:

True+4+null+NaN+false = NaN

“1” + 0+2 –“1”+2-1=102

“1”+0+2-“1”+2-1+”1”=”1021”

True+5+null+false=6

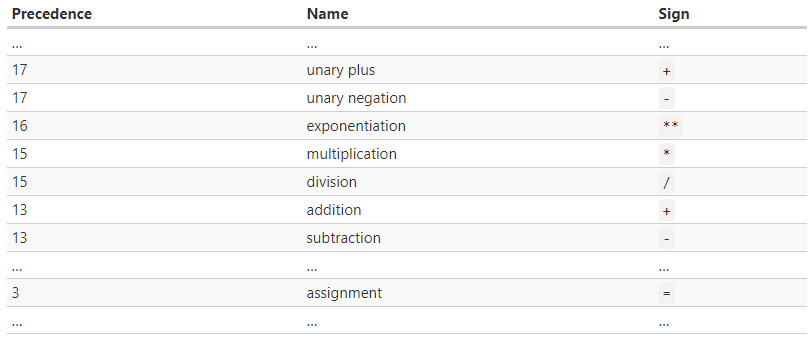
True\*false+null\*”ss”=NaN

True\*false+null\*null=0

“True”+true+5=Truetrue5

And so on.

• Operator Precedence:



Opinion: The way these works are as follows, JS solves expressions from left to right and iteratively compares operators till the end of expression and makes a precedence list. Then starts operating on the operators from the highest precedence operator to the lowest (which is ‘,’). Brackets are always at the top of the list. So for, let a= 2+3\*4/2; , /\*+= will be precedence list and it will solve sequentially using this list. Answer is 8.  
  
• JS support chaining assignment. A=b=c=2; is hence valid.

We can even do

Let a=2;

Let b=3;

Let c=3+(a=b+2);

Here c will be 8 and a will be 5.

• ‘,’ operator: Used to evaluate multiple expr in same line. Returns the last expr value. Precedence is 1.

Usage:  
let b=3;

Let a=(b=4,5); puts 5 in a and 4 in b.

If we don’t use () then it evaluates to a=b=4,5 in which case first 4 is assigned to b then b is assigned to a and 5 is ignored.

• Comparison operators: Shittiest comparison thanks to dynamic types, NaN, undefined and null.

3 types of comparison,

>=,<=,>,<: Both operands are converted to numbers first and then compared, for strings their ascii values are compared iteratively (for the same index positions) , the first bigger ascii value is the greater one. So (“19999”>”2”) evaluates to false.   
Undefined gets converted to NaN and NaN comparison always returns false.  
null gets converted to 0.

==, !=: If the type of the operands is not the same then both are converted to numbers and then compared. 0==false is true because both are converted to numbers.

Undefined == null true, undefined and null return true for ‘==’ only for each other.

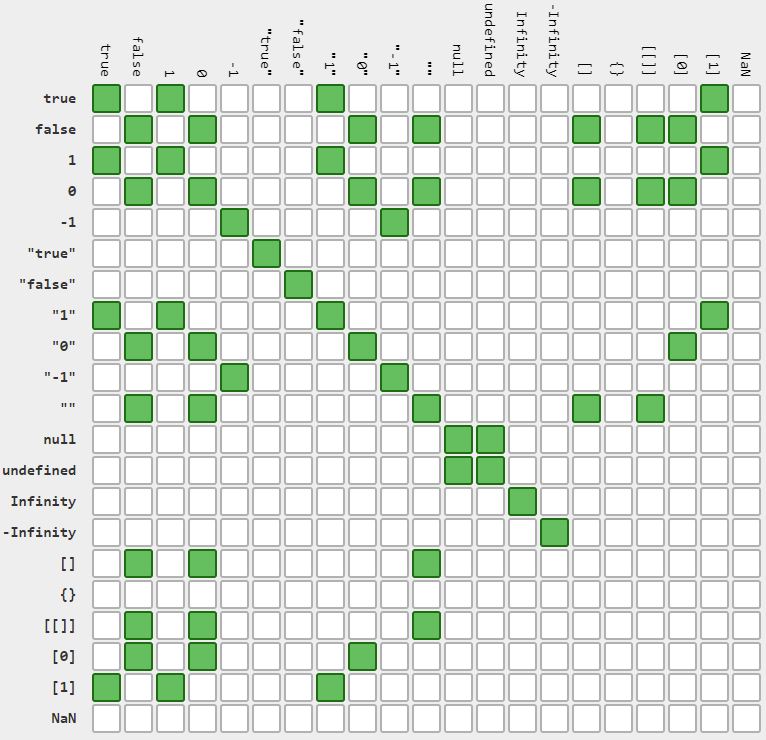
NaN == undefined false,

NaN == null false,

Null==0 false, null and undefined are not converted on ‘==’ ever.

===, !==: Strict check, checks type and value. Here no conversions are performed at all. Undefined=== null also returns false here.

• == comparison table: Best to use strict equality check everywhere.



• Conditional operators are same as in dart, if else, switch and ternary are all supported.

• Logical Operators: These work in a weird way; in it they return a value. Expressions are evaluated from left to right, and && has more precedence than ||. Even so these expressions work in a different way and hence can short-circuit, meaning they can return a value without evaluating the full expression.

&&: Returns the first operand which returns false after being Boolean(<value>) ed. If none is found then the last operand is returned.

||: Similar to && but returns the first operand which returns true after being Boolean(<value>) ed. If none is found then last operand is returned.

The if-else-? then apply another Boolean() over these expressions and then proceed.

For example:

Let a= (1 && 0 && null); returns 0 because Boolean(0) is false. If this same expression is in an if() condition, the returned 0 is passed into Boolean().

Let b= (1 || 0 && null); returns 1, here since || is used and even if && (which has higher precedence) is used after it, the first true returning value, i.e 1 is returned without even evaluating the later expression. This is called short-circuiting.

Technically, these aren’t comparing 2 operands at all.

• ??: Works same as in dart, it’s the null-coalescing operator. If the operand on the left is undefined or null then the operand on the right is returned. Has a slightly higher precedence than || but lower than &&.

• Loops: For, while and do {} while(); are all supported. Break and continue are also supported, one new thing that JS adds is <label>: {}. This allows inner break or continue to jump to this loop’s current state.

For example:

jumpyBoi: while(true)

{

While(true)

{

Break jumpyBoi;

}

}

Breaks out of the loop right after the label. Same works for continue. Normal break/continue would’ve only worked on the loop right above and not the parent loop, that’s where labels help.

• this keyword: this in js is used to accessed the current or parent object’s keys.

For example:

Let temp={

Hello:”vake”,

Hi(){

Console.log(this.Hello); // we could also use temp.Hello but that will always access temp , even if object is copied to another object.

}

}

Temp.Hi(); // “vake”,

However, if we use function(){…} inside a function then this inside cannot access the object above it. To counter this we use arrow function, as that allows this to access parent object. Arrow functions do not have a scope of their own, they extend the scope of the parent Function or Object.

'use strict';

let a= {

    hi(){

        let pp= function(){ return this.namse};

        return pp();

    },

};

console.log(`a is ${a.hi()}`);

This creates an error, to resolve we use arrow function and assign that to pp.

• new: New calls an object’s constructor. This is really important since a ‘constructor’ is not a vague term, a ‘constructor’ is an actual property inside an Object’s prototype property, the constructor holds reference to an Object and that Object’s reference is returned with its [[Prototype]] set to reference the Object in the Prototype itself. That is calling an Object’s constructor and this is prototypical inheritance which happens automatically.

This is how the older environments worked, right now the Object in the ‘constructor’ is just a placeholder. New doesn’t use that Object and instead calls the main Object itself, the prototype is referenced to [[Prototype]] but that’s all.

For declared functions it creates an empty object, assigns it to ‘this’ and if there’s an empty or no return at the end of the function then returns the object automatically, this is called a constructor function, if return is specified then that value is returned and this is discarded.

New.target inside the function returns undefined if the function is called without a ‘new’ keyword.

For example:

Function pp(name){  
this.name=name;

This.age=this.name>”A” ? 18: 10;

}

Let lol=new pp(“BC”); //we can omit () if there are no parameters.

console.log(Lol.name);// “BC”.

Let lol2=new function(){

This.name:”alpha”,

};

Lol2.name //works, here the function is executed right then so instead of returning a function type, an object is returned.

• Optional chaining: Used to return undefined if the left operand is undefined and exists. Much like null-check operator in dart. Can be used for methods, arrays and variables.

Example:

Let a=null;

Let b= a?.();

Assumes a is a function but since it is null returns undefined to b, otherwise it would have resulted in an error.

• Object Wrapper: The methods that are available on primitives, such as .toUpperCase() are not defined like in c++ where a class holds them, instead these are available in an Object held by prototype, that Object is then referenced by [[Prototype]] as internally new Object() is called and this passes prototype’s Object as reference to the [[Prototype]]. This temporary Object is called an Object Wrapper and the methods are predefined. Since it is a temporary Object, even though we can use a primitive like an Object to assign a key and value, it doesn’t return it as it gets deleted.

• Primitive Object: We can create an object of primitive using new. It automatically includes toString() and valueOf() methods.

For example:

Let temp=new Number(2); is an object that has a Symbol key with a value that is 2 along with toString and valueOf methods.

It has the same value as Number(2) but not the same type. However when any method is called on this value then an ‘Object Wrapper’ is created and that is internally created using ‘new’ so a the prototype in Number() is referenced and all the methods defined for Number are accessible by a caller.

We can override the internal methods, but doing so affects it at the global scope.

Number.prototype.toString= function(){

Return this;

} //will override any number’s toString method (and return the same number back, like Number(2).toString() will return 2) in the prototype, which is inherited automatically by any new Number(); .

We can create our own type in an Object, reference the prototype for a given type and then insert our own methods for a non-global use case.

•For..of: For of calls the Symbol.iterator function defined for an Object once and then receives an Object, this Object must be an iterator-object.

• For..of vs for..in: For of loops over values, where for in loops over keys. So, for an array, for in would loop over indexes. For..of’s behavior is defined through Symbol.iterator so it could return anything, but for..in always loops over keys in an Object.

• Iterators: To define an iterator for an Object, it must have a method next() and this method should return an Object with value and done key defined.

For example: This is a valid iterator object definition.

let pp =

{

  0: "abc",

  1: "ddd",

  2: 45,

  Length: 3,

  [Symbol.iterator]() {

    return {

      obj: this,

      currentValIndex: 0,

      next() {

        if (this.currentValIndex < this.obj.Length)

          return {

            done: false,

            value: this.obj[`${this.currentValIndex++}`],

          }

        else

          return {

            done: true,

          }

      }

    }

  }

}

• Map: new Map(); maps can be store keys of any type. Recommended to use get set instead of [ ] to access keys.

• Set: new Set(); just like array but we don’t use [ ] to declare sets.

• WeakMap: Just like Map but it allows only Objects as keys, and doesn’t allow iterations over the entire storage. This is to help with Garbage Collection where Map may have an Object as a key it will still hold the data of the Object even if the reference Object is set to null, WeakMap prevents that and if an object is set to null then it is nulled in the memory as well.

• WeakSet: Just like weakmap, but for sets. Only allows Objects as values.

• Spread operator: Just like variadic template in C++, we have a type of variable in JS that can hold all the arguments. The ‘…’ operator unpacks any array or Object. We can use it to grab all the remaining arguments in functions (called rest parameter there) or just unpack an array somewhere. This operator only works on Objects that implement iterable-object.

For example:

Function test(a, b,…rest) {…}

Test(1,4,5,6,7,8); // will work, here a =1, b=4 and rest will hold all the remaining args as an array. It must always be at the end of the argument list.

• arguments: In every function, there is by default an ‘arguments’ parameter that captures all the arguments.

For example:  
Function test(a, b,…rest) {…}

Test(1,4,5,6,7,8); // now if we call arguments[2] inside the function, it will return 5. It captures values even if they are caught by other parameters. So, we can pass values to a function that doesn’t accept any. Arrow functions don’t have ‘arguments’ as they don’t have a scope of their own.

• Destructuring: JS can ‘destructure’ arrays, objects, maps and sets. That is, for an Object it can unpack it and assign it to variables.

For example:

Let obj={

name: “aaaa”,

value: 23,

};

Let [name,value]= obj; //and now name holds “aaaa” and value holds 23. It works for arrays and the other similar types.

Let [a1=”hello”,a2=”no”, a3]=[“Julu”]; // stores julu in a1, no in a2 and undefined in a3. This is called default assignment.

Let [obj1, , obj2]=[2,3,5,6,7]; // stores 2 in obj1, discards 3 and stores 5 in obj2 then discards the rest.

Let [val1,…val2]=[2,3,4,5,6,7]; //stores 2 in val1 and creates an array to store the rest of the values in val2.

Let {val3,…val4}= {…}; // same as above but for Objects. However, the variable name ‘val3’ must be same as a key in the Object it is destructuring. Even if ‘…’ wasn’t used, only keys with the same name as the variables are assigned.

Let val5,val6;

[val5,val6]=[val6,val5]; // using destructuring we can do smart swap, i.e swap values in variables without a temporary variable.

({val5,val6}={…}); //here ( ) are used so as to make JS use destructuring and not treat the { } as just simple code blocks.

Let {

Size:{

Width,

Length,

},

Title=”none”,

} = {

Size: {

Width: 20,

Height: 30,

}

}; //nested destructuring + default assignment, here an Object is being destructured and since it has an Object inside it, we specify the key’s name to open it up. So the variables created are width, length and title.   
Basically, the syntax is {

<incomingParameterName>: <variableName>= <optionalValue>,

}

We can use destructuring in function calls to unpack Objects.

Function test1({ size: {width:w=400},title=”ad”}={ }){…} //In this destructuring, we can call test1() or test1(<obj>); and both will work, the obj needs to have a key title and a key size which has an object as value with width as a key, if it doesn’t the w=400 and title will be “ad” , if the object isn’t passed it default assigns an empty object. Either way, w and title will be exposed to the function.

• JSON: JSON.parse(<string>,<optional reviver method>); is used to convert string to an Object. We can define a function that accepts a key and value , it will be ran on each key in the Object in the string and return a value which parse will attach to the key.   
We can define toJSON(){…} in an Object so that will be called when JSON.stringify(<obj>,<optional replacer>, <optional space>); is called on the Object. By default, stringify will convert an Object into a string will all keys and values, (unless the key has a value that is the same object which results in a loop or circular dependency, in which case it fails). We can define a replacer function which does the same as the reviver method above, however replacer can also be an array of strings. These strings are keys that stringify will not leave, all key names not defined in this array are left out.

• Execution Context: It is an internal data structure that contains details about the execution of a function, where the control flow is at any given point of time, the variables in the function and ‘this’.

• Lexical Environment: This is an internal data structure as well, usually it contains the same things as Execution Context known as Environment Record here but it works for every { } block, it also contains reference to an outer Lexical Enviroment. The global scope is hence known as Global Lexical Environment.

This means every variable is just a property in an Enviroment Record Object and r/w on the variable’s value is hence a change of value in this Object. And when we access a variable, it is searched in the current LE, then the one above it and so on.

We can’t access this ‘specification Object’ (defined in the language spec) as the compiler performs many optimizations and hence it is basically invisible.

Basically, when a function is called, it automatically receives a [[Environment]] property, this holds the reference to the function that called it, which is the above LE. For global functions, it means that it gets the Global LE as reference. Now when a variable is to be found, it looks in the current block, if not found then it searches in the [[Environment]], and then it’s [[Environment]] and so on. This is why nested functions can access variables in the parent functions.

Function test()

{

Let count=0;

Return function(){

Return count++;

}

}

Let pp=Test();

Pp(); returns 1

Pp(); returns 2 and so on.

All functions in JS are hence Closures, i.e they can ‘remember’ their outer variables and access them. The only exception are new Function(<string>);.

• globalThis, window and global: Depending on the platform (web : window, vanilla js globalThis, node: global), this Object can access all the global variables and functions. ‘Var’ hoists the variables and then this Object can access it, it doesn’t work for ‘let’.

•setTimeout: Used to set a timeout for a function, i.e it will run after the given duration.

Syntax:

Let timerId=setTimeout(<function reference or function String>, <optional delay in ms>, <optional …args>);

• setInterval: Same as setTimeout but it runs a function every given ms later. This doesn’t include function’s own execution time. Meaning if duration is 100ms but func takes 200ms to complete it will have started another function. Better to nest setTimeouts for those tasks. Syntax is same as timeout.

These timer objects are run after the script has ended, meaning they are run after the execution of all the other functions etc.

For example:

let pos=2;

setTimeout(()=> console.log(`value is ${pos}`),1000);

pos=3;

console.log(`The end`);

will print 3. That’s why 0 delay timer objects are used to run something after everything else.

• clearTimeout: clear an interval or timeout. It is recommended to call this method on the object holding the interval or timeout object because these functions keep a reference of [[Environment]] meaning all the environments in the call stack are saved and not dumped by the garbage collection till one of these objects exist.

• <function name>.call(<object>, <…args>): We can call a function using this method. The object that is provided here becomes the context for the function and hence ‘this’ works in the function.

For example:

Function test(a,b){…}

Test.call({ }, 1,2); // works and test gets an Object to reference to for this. Normally we would use new <function name> but this is an alternative.

•<function name>.apply(<object>,<args array>): Similar to call, but this is faster due to compiler optimizations. However it only accepts an array of arguments, where call could take any number of arguments or objects that implemented iterators.

• Method borrowing: We can ‘borrow’ a method from another Object and apply it to another type. We do this by,

<data type>.<method name>.call(<argument that has implemented the data type>); Now these arguments will be passed to the method of the data type and they would be treated as if <arguments> were of that data type.

For example:

[ ].join.call([1,2,3]); //returns “1,2,3”. It would have worked anyway but for all other cases if Object is passed as argument it must be array-like.

• Function Binding: We can ‘bind’ a context to a function so that it can always access a ‘this’ context without sending objects with call or apply.

Syntax:

Let <variable name>=<function name>.bind(<object>,<args>);

Once done we can now call the function anywhere with <variable name>()and it will have reference to this Object, the args are optional but if provided it passes them to the function each time it is called (we cannot pass already specified arguments).

For example:

Function test(a,b)

{

Return a+b;

}

Let binder=test.bind(null,2); //here null is the object reference, since we are providing an argument binder is a ‘partial function’ as we can call part of test from now.

Binder(3); // will return 5. If we provide more arguments they are ignored.

• Property flags and descriptors: Every property in an Object can have property flags, these flags decide what permissions we have over the property of an Object. By default, there are 4 property flags, they are:

Value: The value of the property.

Writable: <bool value> allows value of the property to be changed,

Enumerable: <bool value> allows value of the property to be listed in loops.

Configurable: <bool value> allows the modification of property flags and deletion of the property. If this is set to false it cannot be changed back to true, only writable flag can be turned to false after this is set to false.

All these are set to true for any property.

We can get property descriptors (the flags for an Object) using

Let <varname>= Object.getOwnPropertyDescriptor(<object>, <property name>);

This returns an Object (called descriptor) which holds a copy of the flags.

We can set new flags using Object.defineProperty(<object>,<property name>, <descriptor object>);

For example:

Let user={};

Object.defineProperty(user,’name’,{ ‘value’: “lmao”}); , sets the ‘value’ flag to “lmao” for the ‘name’ property in user object.   
If the property exists, it simply updates the descriptor for it, however if it doesn’t then it sets the default 3 property flags to false (value is required, it cannot be left empty).

Object.defineProperties(<object>,{<prop1>:<descriptor>,<prop2>:<descriptor> }; sets multiple properties and their descriptors at once.

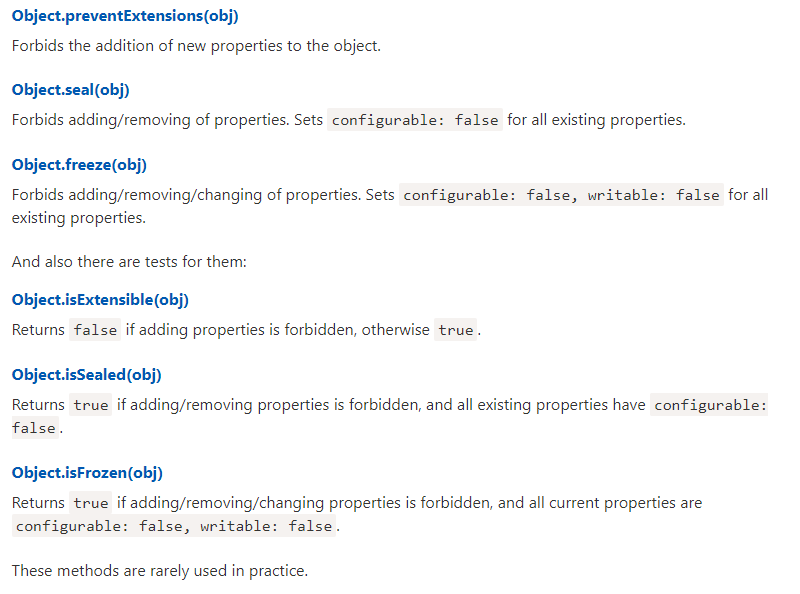
Object.defineProperties(user,{  
‘hello’:{ value:’woo’,writable:true},

‘bye:{ value:’nii’,configurable:true,},

};

And so on.

There are other Object methods to work on multiple properties at once.



• Object accessor properties: Objects can have getter and setter functions, these are called accessor properties and unlike data properties they do not have value or writable flag but get, set, enumerable, and configurable.

For example:

Let user={

firstName: null,

lastName:null,

get fullName(){

return `${this.firstName} ${this.lastName}`;

}

set fullName(value){

[this.firstName,this.lastName]=value.split(‘ ‘);

}

}

User.fullName=”a k”;

User.fullName; //returns “a k”.

fullName is an accessor property so it doesn’t use ().

To set a new one,

Object.defineProperty(user,’age’, {

Get(){

Return this.userAge;

}

Set(value)

{

This.userAge=”yabibi”;

}

};

Since userAge didn’t exist and was created with the setter it’s called smart getter/setter.

• [[Prototype]]: Objects have another hidden property called [[Prototype]], this is used to inherit properties from another Object and can only have a value of null or an Object. It’s called Prototypical Inheritance. It doesn’t copy or references them in an Object, it only references the given Object. When we request value of a property on an Object, it is first searched for in the Object, if not found then searched in the Object defined in the [[Prototype]], if it is still not found then the [[Prototype]] of that Object is searched and so on until an Object with its [[Prototype]] is found to be null, throws if it is a circular search, i.e an Object references an already looked up Object. It is recommended to use Object.get/setPrototypeOf(<obj>); to get or set [[Prototype]] of an Object, we can use \_\_proto\_\_ keyword as well(it is a getter/setter on the internal [[Prototype]]) but its support is waning. One Object can only have reference to 1 Object and it is read-only, i.e if we modify a value for a given key then it is written to the current Object and not in its parent. Accessor properties are no exception as the rule with ‘this’ keyword is that it has the reference of the calling Object.

For example:

Let user={

firstName: ”Aan”,

lastName: “Khan”,

get fullName(){

return `${this.firstName} ${this.lastName}`;

}

set fullName(value){

[this.firstName,this.lastName]=value.split(‘ ‘);

}

};

Let anotherUser={

Age:20,

\_\_proto\_\_:user,

};

anotherUser.firstName; //returns “Aan”

anotherUser.firstName=”Bcn”; // creates a property named firstName in anotherUser and sets it to Bcn.

anotherUser.fullName; //returns “Aan Khan”.

anotherUser.fullName=”p p”; // creates firstName and lastName in anotherUser and sets them to p.

[[Protoype]] properties are iterated in for in loops but Object.keys does not. To check if a key exists in the current Object we can use <object>.hasOwnProperty(<key>); , returns true if key in current Object false otherwise.

Object.get/setPrototypeOf(<obj>); and new Object.create(<obj>,<descriptor>); do the same thing except an optional descriptor Object can be passed which will add properties to the prototype Object and not the constructor Object. Meaning that the <obj> remains unchanged.

• class: A class in JS is just a kind of function, it’s typeof also returns the same. However, they are a bit more advanced.

Syntax:

Class <name>{

Constructor(<params>){…}

}

With objects created using, new <name>();

New calls the constructor method here. This constructor is not the same as the ‘constructor’ in the prototype, that constructor is just a placeholder but this constructor is actually called on Class Object creation.

Class may be a type of function but there are a few differences, class properties are non-enumerable by default. Classes have ‘use strict’; by default. And an internal property [[isClassConstructor]] is set to true for class Objects.

Just like functions, classes can have class expressions and named class expressions.

For Example:

Let test=class {…};

Or

Let test= class Test{…};

Getter/setter in class work just like in Objects.

All methods in a class are referenced to prototype.

We can have class fields too (we can’t use let or var as they are let by default), however, they are set on individual Objects and not in the prototype.

So, for,

Class A{

Name=”lol”;

sayHo(){…};

}

A.name; //returns lol.

A.sayHo(); //fails.

A.prototype.sayHo(); //works.

A.prototype.name; //fails

Let test=new A();

Test.sayHo(); //works

Test.name; //works

Test.\_\_proto\_\_.mm(); //works. Note: \_\_proto\_\_ holds the reference to the Object that is in prototype of A{}.

• Extend: JS supports classical inheritance, (only a single class can be inherited), but hierarchical inheritance works. When a class is ‘inherited’ the [[Prototype]] of T.prototype gets the K.prototype Object as reference.

Extend also works on functions that return a class.

Syntax: class <name> extends <name2>{…}

For example:

Class A{

}

Class B extends A{

}

B.prototype.\_\_proto\_\_==A.prototype; //returns true.

Now if an Object of B tries to call a method or a field, it is first searched in B, then in B’s prototype and then in B’s prototypes’ prototype. This means fields, which are not referenced by prototype are invisible to inheriting classes. This also means, if a method is found in B.prototype then it will be called, essentially breaking the search and if a method with same name is in A then that is ’hidden’.

‘This’ can still access the current Object’s methods and fields.

Function f(){

Return Class{

sayHi(){…}

}

}

Class A extends f(){…} //works as functions are a subtype of Object.

• Super: super() calls the constructor of the parent class, super.<method>() calls a method in the parent class. Super can only be called by a child class. As usual, arrow functions don’t have a LE hence they don’t have their own super.

By default, if we don’t define a constructor in a child class then it is automatically inserted.

So B becomes,

Class B{

Constructor(…args){

Super(…args);

}

It is necessary to call super() in the constructor (if we define it for class A and class B). super() must be called before accessing this in the constructor. The reason for this behavior is, when new <T>() is called it checks for an internal flag [[ConstructorKind]], and if this flag is set to “derived” (which is set in a child class) then it doesn’t provide a new Object to the constructor method (it provides a new Object to new function() because it has this flag set to some other value). The child Object’s constructor can’t call this since an Object hasn’t been passed to it, that’s why super() is called, as it creates an Object, passes it to parent class and then sets this to the Object.

• Overriding trick:

For,

Class A{

Name=”a”;

Hi(){

Console.log(“a”);

}

Constructor()

{

Console.log(this.Name);

}

}

Class B extends A{

Name=”b”;

Hi(){

Console.log(“b”);

}

}

New B(); //prints a

New A(); //prints a

But if console.log(this.Name); is replaced by this.Hi();

New B(); //prints b

New A(); //prints a

The ‘trick’ here is due to how super initializes the Objects.

This is how ‘super’ call works:

1. Initialize all child class methods.
2. Initialize all parent class fields.
3. Initialize all child class fields.

Since our constructor in A is called at 2 it can’t see the overriding class fields.

• super vs call forwarding: Normally we can ‘call’ a method and pass our own context for it to use. But sometimes that is not enough.

For example:

Let obj1={

Hi(){

Return this.name;

}

}

Let obj2={

\_\_proto\_\_:obj1,

Hi()

{

This.\_\_proto\_\_.Hi.call(this);

}

}

Let obj3={

\_\_proto\_\_:obj2,

Hi()

{

This.\_\_proto\_\_.Hi.call(this);

}

}

Obj3.Hi(); //fails. As it goes into an infinite call loop, using ‘this’ and call forwarding we pass obj3’s [[Prototype]] (which is obj2 due to us manually defining the \_\_proto\_\_) to obj2, so it calls obj3’s [[Prototype]] again and infinitely. To resolve this, JS has an [[HomeObject]] internal property which is always set to the same Object. It can be accessed using super.

So replacing the 2 this.\_\_proto\_\_.Hi.call(this); with super.Hi(); will return the correct value.

Super will look into [[HomeObject]]’s prototype to determine it’s parent and then go there.

This means, functions and all Objects that are passed around ‘remember’ their original references. Also meaning that if a method in an Object calls its super, then no matter where the method is called from it will always look at that Object’s parent.

That is why, if a <property>:<function> (){…} is created in an Object, a ‘super’ in it will not be able to access [[HomeObject]] of the Object, instead we use <property>(){…} for the same.

• Computed method names: Objects can have strings as method names, these are computed at compile time.

Syntax: [<string>](<params>){…}

For example:

Let obj={

[‘say’+’ho’](){}

}

Obj.sayho(); or obj[‘sayho’](); // works.

• Static methods: These methods are just like class fields and are there for each instance rather than being in the prototype.

Syntax: static <methodName>(){…}

Or

<obj>.<methodName>=function(){…} does the same thing.

Static classfields can also be declared similarly.

However there’s a difference between static fields and normal fields, static fields are inherited, i.e they are referenced in the [[Prototype]] object.

• factory method: There is no factory keyword but using static methods we can do the same.

Static createNewObj(){

Return new this();

}

This will call the constructor and return a new Object.

• Private, public and protected: By default, all methods and fields are public in js. Protected properties of an Object are prefixed with ‘\_’ and private properties are prefixed with ‘#’. However, private properties aren’t implemented in the language yet and protected properties are just for syntax and letting others know.

• Built-in Type inheritance: We can extend built-in classes. All methods that are called on the object of this child class (and belong to the built-in class) return an object of child class by default.

So,

Class A extends Array{

Dance()

{…}

}

Let a=new A(1,2,3,4);

Let b=a.filter(…);

B.dance(); works, since A is returned by filter.

This is because the built-in methods call the T.prototype.constructor(); and since it references A in this case that is returned. We can override this behavior by,

Class A extends Array{

Dance(){…};

Static get [Symbol.species](){

Return Array;

}

}

Let a=new A(1,2,3,4);

Let b=a.filter(…);

B.dance(); will not work anymore.

Built-in classes inherit from Object’s prototype but not from Object. That is why methods like Object.keys() won’t work with Array.keys();

• instanceOf: Returns bool for if an Object is an instance of another Object, works for inherited objects too.

Syntax: <obj> instanceOf <obj>;

Also works for functions

New function() instanceOf function ; //true

Array instanceOf Object; //true

instanceOf calls a static [Symbol.hasInstance] and returns the returned bool.

Let obj1={

Static [Symbol.hasInstance](obj)

{

If obj.stuff=blabla

Return true;

Return false;

}

}

Let a=obj1;

A instanceOf obj1 ; // will call the given static symbol property.

If this method isn’t defined then it compares,

Obj.\_\_proto\_\_ to obj2.\_\_proto\_\_

Obj.\_\_proto\_\_.\_\_proto\_\_ to obj2.\_\_proto\_\_; and so on until either it is found or prototype of an object is found to be null.

<Obj>.isPrototypeOf(<obj>); works the same way.

• toString: This method is meant to return a value from Symbol.toStringTag.

So, for

Let obj2=Object.prototype.toString;

Let arr=[];

Obj2.call(arr); // will print [Object Array]

Let num=Number();

Obj2.call(num); //will print [Object Number]

We can define our own Symbol.toStringTag for an Object.

Let obj3= {

[Symbol.toStringTag]: “obj3”,

}

Obj2.call(obj3); will print [Object obj3].

• try catch finally: Same as in any language.

Try{

//sync code, any errors go to catch. Sync errors mean if there is an error inside a setTimeout or interval then they are not caught.  
}

Catch(err) //we can put without () and that means err obj isn’t needed.

{

///stuff. Err.name, err.message and err.stack are available by normal Errors.

}

Finally{…} //finally is optional, but if we put it, then it is ran regardless of an error. Meaning even if try{} has a return inside it will skip the return.

Throw new <error> to throw.

We can throw any Object, classes are better as their properties can be read by the catch.

We can also extend Error.

Class someError extends Error{

Constructor(message)

{

Super(message);

This.name=”someError”;

}

}

Throw new someError(“yahoo”); will work.

Browsers can read an onerror property,

Syntax: Window.onerror=function(message,url,line,col,error){…}

On an error that kills the process this method is ran.

• callbacks: Functions passed to functions are called callbacks, usually callbacks also include an error parameter, this helps mitigate errors better. However callbacks can create callback-hell, meaning a deep nesting of these functions is a bad practice of coding as the debugging is very time-taking.

• Promise: Promises are a modern way of handling async code. A function is passed to Promise which accepts 2 parameters (called an executor function), resolve and reject (these are both functions). In the function itself we have to call either resolve or reject after we are done with any task. Then the function in either resolve or reject is ran. The executor is ran right where it is defined, i.e Promise starts right where it is declared. The resolve and reject functions are optional but have to be provided (in the .then method of a Promise object) by us if we want to use the value / error generated by an executor function.

Syntax:  
let <varname>=new Promise(function(res,rej){…}).then(function(resValue){…},function(rejValue){…});

In the executor,

Res(<value>); and Rej(<value>); call the first and the 2nd function in the then respectively, they also pass the value provided to these functions.

Promises are also an Object and internally have [[PromiseState]] and [[PromiseResult]] properties, the state holds if the executor is done /resolve is called (has the value “fulfilled” for done and “rejected” if there was an error or reject was called). The result holds the value we pass to resolve or rejected.

For errors, both .catch(function(err){…}) and the function in then for reject can be used.

The .finally(<function with no args>); method is used to do something after the executor and the then blocks have completed. If we have a .finally before a .then then the value ‘passes through’ and the then is ran before the finally.

The .then, .catch and .finally methods wait for a promise to end. So,

Let promise=new Promise(function(res,rej){

Res(12);

});

promise.finally(()=> console.log(“done”));

promise.then((res)=> console.log(res));

works, prints 12 then “done”.

• Promise chaining: We can chain a bunch of promises very easily by using then, these ‘then’ are called ‘thenables’, thenable is an class Object that implements a ‘then’ method as an executor function.

Class Thenable {

Constructor(num)

{

This.num=num;

}

Then(res,rej)

{

…

}

}

New Promise(res => res(23)).then(result=> {return new Thenable(result);}).then(result=>console.log(result));

This is promise chaining, by default returning a Promise from a then is how we do promise chaining, but we can use a thenable too.

The flow here is,

1. Executor in Promise
2. After it completes successfully calls resolve and passes 23
3. The then catches the 23 and returns a ‘thenable’, it could also return another promise.
4. The next then catches the thenable Object’s result (if it called a resolve in this case) and processes it.

Let promise=new Promise(…);

Promise.then();

Promise.then(); // is not promise chaining, then’s are all consumers and in this case after promise completes all the consumers are given the value.

For errors, a single catch is at the end of a promise chain is good enough.

A good practice is to always return a promise Object from an async block even if we don’t consume it.

• Promise API: The Promise in js provides some other methods for handling Promises.

Promise.all([<promises or values>].then(); We can pass an iterable (usually an array) of promises/ other values and after all those promises finish (it waits for every one of them to finish as it processes them parallely) or there’s an error/reject() call in the middle the then() is called. If all of the promises resolve then their values are passed as an array (in the same index as their respective promises in the iterable) to the then, we can pass raw values or normal functions in the iterable and their values are passed as well. If there is an error or reject is called by a promise then it doesn’t wait for the rest of the promises to complete and passes the reject value or error to the then.

For ex.

Let promise=Promise.all([1,2,new Promise(function(res,rej){…}),4]).then(resVal=>…); //here resVal will be [1,2,<something>,4];

Promise.fulfilled([…]).then(function(res)=>{…}); This is the same as promise.all except it gathers all resolve and rejected values, even errors. They are passed as an array to the then and have their status property exposed. So, for res[0].status, it could be “fulfilled” while res[1] could have “rejected” and so on. For objects with “fulfilled” status we get .value and for “rejected” we get .reason.

Promise.race([…]).then(…); Similar to promise.all but waits for any one promise to complete only. Then it passes it’s resolve or reject value to then.

Promise.any([…]).then(…); Similar to promise.race() but this promise handler waits for the first ‘fulfilled’ promise, i.e if the fastest promise returns a reject/error then it waits for the next one and passes the first ‘fulfilled’ promise’s value to then. However if all the promises fail then a special error Object ‘AggregateError’ is created and it passes all the errors/reject values as an iterable to the reject/catch method.

Promise.resolve(function); This is rarely used but implies that the promise will only return a resolved function.

Promise.reject(function); Same as resolve but for returning only reject.

• Microtask queue or PromiseJobs: All async tasks in JS are scheduled after the main script is ran and that’s due to this internal queue where FIFO is followed and all promises are executed sequentially after the script.

Promises are scheduled in the microtask queue however everything else such as setTimeout are scheduled in the macrotask queue. After normal events, microtasks are executed then macrotasks and if there are multiple microtasks, they are scheduled right after macrotasks. The priority is:

1. Synchronous events
2. Microtasks
3. Macrotasks

We can use queueMicrotask(<func>); to add a function to the microtask queue, i.e., have it processed as soon as possible but without delaying the synchronous code.

• Async/Await: JS also provides async and await just like in dart, these ensure that a task is completed before proceeding. Normally the microtask queue runs after the script but with async and await it pauses for tasks to complete (it does other unrelated stuff while it is waiting).

Syntax:

Asnyc <function>{… let something=await <promise>;} async functions return a promise. If a function is processed successfully the return value is wrapped in a resolve, else a reject.

Async ()=>{…} also works.

Awaits work with ‘thenables’.

• Generators: JS supports generators like dart. But unlike dart they don’t keep running, they just return a lazy iterable with all the yield values. For..of to loop over a genny’s values. Internally a genny is just an Object that implements an iterator, that’s how for..of works on it.

Syntax: function\* <func name>(){ yield <val>or return;} We can use return but it’s value is ignored. Empty yields return undefined. The last yield puts the ‘done’ property in iterator object as true.

Function\*(){…} for anonymous functions;

\*()=>{…}; for arrow functions.

Essentially, any generator returns an array of values. But their ‘yields’ aren’t processed until the next() is called on a generator function, this is why they return a ‘lazy iterable’.

let test =

{

    0: "abc",

    1: "ddd",

    2: 45,

    length: 3,

    \*[Symbol.iterator]() {

        for (let elem = 0; elem < this.length; ++elem)

            yield elem;

    }

}

console.log([...test]);

We can implement our iterator easily using a generator.

• Yield\*: Using yield\* we can call a generator function and all its yields are ‘transparently’ forwarded as if all of them were yielded from the caller function. This is also called generator composition.

Yield in JS is like a 2-way street, i.e it can pass values back into the generator (not as args).

For example:

Function\* test(){

Let result=yield “yoo”;

Console.log(result): //prints no.

Yield “hoe”;

}

Let testVar=test();

Console.log(testVar.next().value); //prints yoo

Console.log(testVar.next(“no”).value); //prints hoe

testVar.throw(<err>); //will pass an error object back to the generator where a try catch block can catch it. Yield can also pass an error.

• [Symbol.asyncIterator]: Just like a normal iterator JS also provides an asyncIterator. The difference here is that the next() is instead an async next(){…}, and the for..of is for await(..of..);

For example:

let test =

{

  0: "abc",

  1: "ddd",

  2: 45,

  Length: 3,

  [Symbol.asyncIterator]() {

    return {

      obj: this,

      currentValIndex: 0,

      async next() {

        if (this.currentValIndex < this.obj.Length)

          return {

            done: false,

            value: this.obj[`${this.currentValIndex++}`],

          };

        else

          return {

            done: true,

          };

      }

    };

  }

};

(async()=>{

    for await(let elem of test)

        console.log(elem);

})();

Since next is async now we can have awaits inside it.

• Async Generator:

Syntax: async function\* <name> (){…} // yield stuff like normal, await stuff like normal.

Start the genny and consume it like an asyncIterator (for await of)as it is the same, an iterator on Promises. This means unlike normal gennys, we have to use await genny.next();

Async Generator Iterator: We can use async gennys for asyncIterators.

let test =

{

    0: "abc",

    1: "ddd",

    2: 45,

    length: 3,

    async \*[Symbol.asyncIterator]() {

        for (let elem = 0; elem < this.length; ++elem)

            yield elem;

    },

};

(async()=>{

    for await(let elem of test)

        console.log(elem);

})();

• Modules: Modules are parts of any file that we can either export or import. Only exported methods/variables/classes are seen by any file that imports them and it can edit the script (only for runtime). In browsers, imported module files are executed only once and that is at the first import. Modules always ‘use strict’;. In browsers, modules can only import modules. Meaning we cannot interact with them using normal scripts (because of the microtask queue which puts module and their loading after the entire page and scripts are loaded). So, we can only import inside module tagged script in HTML.

Syntaxes: export <method or variable>

Export { methods/variables/classes }; //this is for marking multiple of the variables/methods as exported.

Export{method as m, variable as v}; //exports the methods as a given name.

Export default <methods/variables/classes> ; This passes a single default entity to the importers. There can only be one export default per file and it’s better to not mix it with normal exports as that beats the purpose of a single export. Export default entities can choose to not have a name.

For example:

//1.js

Export default function(){…}

2.js

import {default as <someName>} from <filelocation.js>;

or

import <any given name> from <filelocation.js>. //even if we give a default export a name it is ignored we can simply assign any name to the default entity using this import.

Other ways to import:

Import { methods/variables/classes } from <filelocation.js>;

Import \* as <obj> from <filelocation.js>; // puts entire imported modules into an Object.

Import {method as m, class as temp} from <filelocation.js>; // imports the module and changes the invoke name to the given name.

So if a file imports another script, then the imported script is first evaluated, then it becomes ready to be used by the importer. If there are multiple imports, then they all get the same reference from the memory.

For example:

//1.js

Export anObj={

Name:”lol”,

}

//2.js

Import ‘./1.js’;

anObj.name=”no”;

//3.js

Import ‘./1.js’;

Console.log(anObj.name); prints no.

Re-export: We can export imported entities to provide a level of abstraction, meaning importing a single file can import multiple files as the file re-exports or forwards the deeper files.

Syntax:

Import { methods/variables/classes } from <filelocation.js>

Export { methods/variables/classes, default as <something> }; // default needs to be re-exported with a name otherwise it is ignored.

Export \* from <filelocation.js> // cleaner syntax for re-exporting, however default exports are ignored.

All import and export statements must be in global scope.

For browsers, we import modules using <script type=”module”> </script>// since this is a script tag we can insert js inside it. However, each script module has its own independent scope, so if multiple of these tags are defined, they have their own scopes. But these module tag scripts don’t have a top level ‘this’, normal script tags bring that.

In browsers, module tag scripts are loaded after the entire page is loaded.

<script async type=”module”> … </script> is a script module that loads parallely with other modules, meaning it doesn’t get processed sequentially and so even if browser puts scripts in a queue, it processes along with other scripts.

<script nomodule>…</script> is for compatibility with browsers that don’t support modules. If browsers don’t support modules then the scripts inside are ran.

• Dynamic import: We can import modules at runtime using special ‘import()’ syntax. This is not a method but something like super(). For dynamic import, exports are done as usual but import returns a promise object with the loaded file. Dynamic imports don’t need type=”module” in script tag.

Syntax:

Let <varname>= await import(‘<filelocation.js’); something is an Object with all the exported methods/variables/classes as properties.

<varname>.default holds the default exported method/variable/class.

It can work with destructuring too,

Let {<var1>,<var2>}= await import(‘<filelocation.js>’);

• Proxy: Proxy in JS is a ‘transparent’ Object that wraps another Object and intercepts any call to the Object.

Syntax:

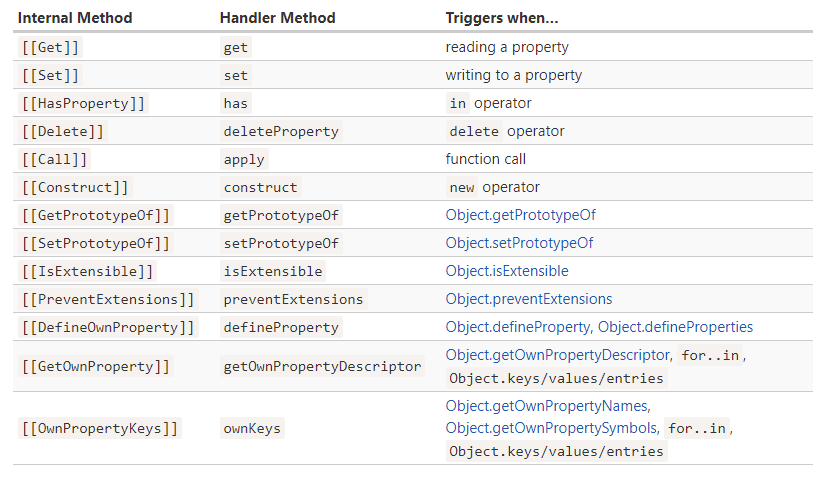
Let proxy=new Proxy(<obj>,<handler>);

If handler object is empty or null then all calls are transparently forwarded.

Proxy.test=5;

<obj>.test; //returns 5.

A handler is a method that intercepts the calls using ‘traps’, which are properties that are invoked on their corresponding internal calls on the proxy object.



For some properties, there are invariants. These are rules the values in properties must follow.

Get method: get(<obj>,<property>,< optional receiver>){…} receiver object is used as ‘this’ inside the get method.

For example:

let nums=[243,51,3];

let proxy=new Proxy(nums,{

    get(target,val)

        {

            console.log(`nums is ${val}`)

            if(val in target)

                return target[val];

            else

                return 0;

        }

})

console.log(proxy[2]);

//prints 3

Here val (as key in properties) is looked for.

It is recommended to hold the reference of an Object only in a proxy or it’s own variable but not both. So, we should hold proxy in ‘nums’ in the above example.

Set method: set(<obj>,<prop>,<value>,<optional receiver>){…} For set, there’s an invariant and it is that setters in proxies should return true or false for if insertion was successful in the Object or not.

For example:

Let nums=[1,2,15];

Nums=new Proxy(nums,{

Set(target, prop,val){

If(typeOf val == “number”)

{Target[prop]=val;

Return true;

}

Else

Return false;

}

});

Nums.push(2); //works

Nums.push(“”); //fails with TypeError

We can use proxies for other operations such as providing wrappers as well.

Function del(obj,delay){

Return new Proxy(obj, {

Apply(target,thisArgs, args){

setTimeout(()=> target.apply(thisArgs,args),delay);

}

});

Function sayHi(name)

{

Console.log(`namaewa ${name}`);

}

Let hello=del(sayHi,2000);

Hello(“lol”); //prints namaewa lol after 2000ms.

We could have used normal Wrapper function here but it wouldn’t forward any properties to the main Object.

Proxies may be transparent but they cannot pass a ‘===’ check.

Proxies cannot reference/access internal methods of an Object or private fields (in proposal).

• Reflect: For every method that a proxy has, reflect has the same method available on any Object. Reflect allows us to call operators (such as new, delete) as functions.

For example:

Let user={};

Reflect.set(user, ‘name’, “something”); //the args are same as methods in proxy as well. In essence, reflect allows us to transparently forward an operation to the main Object.

The main use of reflect is with proxies, since proxies don’t forward the caller context to the internal methods (target has the reference to the Object given to a proxy). For inheritance this causes an issue as ‘this’ in a parent Object receives context of the Object in the proxy no matter which child calls it.

let user = {

    \_name: "ho",

    get name() {

        return this.\_name;

    }

};

user = new Proxy(user, {

    get(target, prop, reciever) {

        return target[prop];

    },

});

let admin = {

    \_name: "no",

    \_\_proto\_\_: user,

};

console.log(admin.name);

//prints ho where “no” is expected. To fix this we simply return Reflect.get(…arguments); from the get proxy and reflect uses the receiver to pass the receiver context automatically.

For maps and some other data structures, simple get and set aren’t used internally to store data. Instead, internal slots like [[MapData]] are used which proxy can’t access directly, nor does it reference. So, we use Reflect and bind to get the data.

Let map=new Map();

Map=new Proxy(map, {

Get(target, prop, reciever)

{

//Return target[prop] or Return Reflect.get(…arguments); will cause an error. Instead,

Let value=Reflect.get(…arguments); // to call internal [[Get]].

Return typeOf value == ‘function’ ? value.bind(target) : value; // This will check if the property passed returned a function, i.e an internal method then it will bind the target’s context to it so any further calls on the Object will not search for the method in the proxy object but in the target object.

}

});

• Revocable Proxy: This type of proxy can revoke access to the Object. Meaning it cannot be accessed anymore after getting revoked.

Syntax:

Let {<proxyVariable>,<revokeMethodVariable>}=Proxy.revocable(<target>, <handler>);

We can access the Object normally using <proxyVariable>.<someProp>. But as soon as we call <revokeMethodVariable>(); any further calls will return Error.

Since <proxyVariable> and <revokeMethodVariable> are 2 separate variables we can store the <revokeMethodVariable> as a property in the <proxyVariable> to easily access it anywhere or use a weakMap.

• eval: eval is a function that evaluates a string and processes it. It is unadvised to use this as any variables used inside it are not minified and if they are in a scope then they are not minified for that entire scope.

Syntax: eval(<any line>);

To use only the global scope call window.eval(). To use eval with variables (and minifiers), use new Function(<args string>,<body as string>);

• Currying: Currying is a transformation of functions from f(a,b,c) to f(a)(b)(c). Basically, we return function from function to convert a function into a partial function.

For example:

Function curry(func)

{

Return function curried(…args)

{

If(args.length>=func.length)

{

Return func.apply(this, args)

}

Else

{

Return function(…args2)

{

Return curried.apply(this,args.concat(args2));

}

}

}

}

Function sum(a,b,c)

{

Return a+b+c;

}

Let currySum=curry(sum);

currySum(1,2,3); //prints 6

currySum(1)(2,3); //prints 6

• Reference type: This is a hidden type used by JS to pass context (‘this’) to any method.

When we call obj.method();, it internally takes context as obj using the ‘.’ notation. This can be resolved at compile time but at runtime this cannot be resolved and causes error.

For example:

Let user={

Name:”lol”,

Hi(){

Return this.name;

}

Bi()

{

Return this.name + “ya”;

}

}

(User.name==”no” ? user.hi : user.bi)(); //error

The ‘.’ returns not a function but a special reference type.

Which is

(<base>,<property name>,<isStrict>); When () is called on this reference type it gets the correct ‘this’ but as in our example the () is appended later it has lost ‘this’.

• XMLHttpRequest: It is a built-in browser object that allows HTTP requests from Javascript. It can operate on any data and not just XML.

Syntax:

Let <varname>=new XMLHttpRequest();

<varname>.open(<method>,<url>,<optional async>,< optional user>, < optional password>);

Method: “GET” or “POST” or some other method,

url: takes URL() object.

Async: <bool>, true default and false for sync call. Sync call pauses the execution of other items and is considered a bad practice.

User,pass

Open doesn’t open anything, it’s just a name.

<varname>.send(<optional body>); sends the request. It should be the last method to be called on the object. In optional body we can give a FormData object as well. We can get instantiate it like FormData(<form element>); or FormData() and append values to the object manually as it is an Object. The optional body can be anything.

Has 4 events and their eventHandlerAttributes. Load, error, timeout and progress. Load is triggered if data gets fully downloaded even on HTTP400 or so, error if that failed and progress triggers periodically. For timeout, we can specify duration in ms as it is a property and its event is triggered after no response in the given duration.

<varname>.upload. This object has its own events and eventHandlerAttributes. For POST requests these are useful. The events are:

loadstart – upload started.

progress – triggers periodically during the upload.

abort – upload aborted.

error – non-HTTP error.

load – upload finished successfully.

timeout – upload timed out (if timeout property is set).

loadend – upload finished with either success or error.

And can be assigned listeners with <varname>.upload.on<event>=function(){}

<varname>.responseType=<value>; to set the response format.

Value can be:

“” : default, empty string.

text: string,

arraybuffer: Arraybuffer() obj,

blob: Blob(),

document: get as XML doc or HTML based on MIME type.

json: as json.

<varname>.setRequestHeader(‘<name>’, ‘<value>’); to set custom headers. Once set, the value never gets removed for the given header of this object. Any further calls don’t overwrite, but append on the value for the given header.

<varname>.getResponseHeader(<name>); to get one.

<varname>.getAllResponseHeaders(); to get all. Returns a single string with /r/n separated pairs.

<varname>.readyState returns the ready state of the object.

0: unsent,

1: opened (or rather, configured),

2: headers received,,

3: Loading, for progress.

4: Done.

onReadyState eventHandlerAttribute is used to check progress periodically.

<varname>.status for httpstatus code. 0 for any non http error.

<varname>.abort() cancels a request and fills 0 in statusCode.

<varname>.withCredentials=<bool>, true to send cookies and HTTP-authorization for CORS.

• Cookie: It is a small string of data stored in the browser, the string has ‘key=value’ pairs with ‘;’ seperators. It’s sent to a browser using Set-Cookie HTTP Header in the response.

Document.cookie is a getter/setter accessor to append key value pairs to the string. Since it is a string and not an Object we have to add values like,

Document.cookie=”key=value”;

The key and value can support all the characters however it is recommended to use encodeURIComponent(<val>); to get a string which can be put in the cookie.

Document.cookie=encodeURIComponent(key)+”=”+encodeURIComponent(value); is hence used. (We don’t run it over ‘=’ since that seperates keys and values but encoding converts it to something else).

A key value pair in a cookie shouldn’t exceed 4KB size and a cookie cannot have 20+ pairs.

Cookies have several options which browsers use before accessing a website.

path=/; Means the cookie will be used and visible to all pages on a domain, /<x> would mean only /<x> and pages inside it will be able to access this cookie.

domain= x.com; sets the visibility of cookie to the domain and all subdomain such as T.x.com. We can’t set domain to a domain other than the current domain.

expires=Tue, 19 Jan 2038 03:14:07 GMT; sets the expiry of a cookie to a desired date, the format must be same. If this or max-age is not set then cookie is deleted on browser close, which makes them session cookies. Also deleted if expires has a date before the current day.

max-age=<seconds>; is an alternative. Deleted if value is 0 or -ve.

secure; Just specifying this value means a cookie can’t be used in between protocols. So <https://x.com’s> cookie won’t be visible to <http://x.com>

samesite=strict/lax; samesite without value is strict. This is used to prevent XSRF. If it is strict then cookie isn’t sent to the domain from any other domain. Even if the domain is open in another window, lax just allows other windows.

httpOnly; if set prevents javascript to open a cookie. Document.cookie hence returns null.

Since keys in cookies are usually encoded, we can’t directly access them. So we use this helper function,

function getCookie(name) {

let matches = document.cookie.match(new RegExp(

"(?:^|; )" + name.replace(/([\.$?\*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + "=([^;]\*)"

));

return matches ? decodeURIComponent(matches[1]) : undefined;

}

Updating:

function setCookie(name, value, options = {}) {

options = {

path: '/',

// add other defaults here if necessary

...options

};

if (options.expires instanceof Date) {

options.expires = options.expires.toUTCString();

}

let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);

for (let optionKey in options) {

updatedCookie += "; " + optionKey;

let optionValue = options[optionKey];

if (optionValue !== true) {

updatedCookie += "=" + optionValue;

}

}

document.cookie = updatedCookie;

}

setCookie('user', 'John', {secure: true, 'max-age': 3600});

Deleting:

function deleteCookie(name) {

setCookie(name, "", {

'max-age': -1

})

}

Document.cookie may set values in a cookie but it is binded to that domain, however if the site being visited sends a Set-Cookie header with a cookie that has another domain then that cookie can be accessed by that domain later.

• Storage: Used to store key value pairs for persistence. LocalStorage is used for long term storage (surviving browser/os reboots) whereas sessionStorage is used for a single session (surviving tab refresh but not reboots). Each storage object is only visible on the same origin, i.e., port+domain+protocol. Both objects provide same methods and properties, which are:

setItem(key,value);

getItem(key);

key(index); gets key at index

removeItem(key);

clear(); remove everything

length;

For example

localStorage.setItem(‘a’,1);

We can use these objects like Objects to set their properties but it doesn’t fire a necessary event which is normally triggered.

Key value must be both strings.

For..of doesn’t work on these objects, for..in with hasOwnProperty (to filter out setItem etc.), Object.keys(localStorage) and traditional index iterative loop works.

‘storage’ event is fired whenever storage objects have a change, but the eventHandler for them only catches events on another window.

For example:

window.onstorage=event=>{...};

will get triggered if any of the storage objects get changed in the same origin but different window/tab.

event.storageArea contains which Object (local/session) had the change. Event has details of the changes as well.

It’s called same-origin inter-window communication.

• Popups and window methods:

Open a popup with window.open(‘<url>’, name, params);

name is a defined string that goes into window.name, if a window with the ‘name’ is already open then that window is brought into focus else new one is opened.

params: Configure new window settings, Single string with ‘,’ separated params. The params are:

left/top: Co-ordinates of top left corner of window

width/height: Can’t be < a certain number.

menubar: bool, show or hide the browser menu

toolbar: bool, show/hide the navigation bar (back/forward etc.)

location: bool, show/hide URL field

status: bool, show/hide statusbar

resizable: bool, allow resizing

scrollbar: bool, allow scrolling

All bools are yes/no instead of true/false.

If params is not defined or values are not specified then default params are used, bools are ‘no’, width/height/left/top is same as last window opened

An opened popup is can navigate (change URL) and send messages back to opener window, the opener window can send messages as well. CORS is applied, if they aren’t from same origin then this is limited (all content of either window is not visible to another and only location of popup can be changed but it cannot be read).

Most browsers block popups if they are called outside of event handlers. An exception is when popups are opened through setTimeout and have a timeout of <=2000ms.

let newWin= window.open(…);

Window.open returns a window reference, allowing us to change and read its properties.

newWin.focus(); to focus the window.

newWin.blur(); to unfocus the window

newWin.close(); to close a window, newWin.closed to get status in bool.

Window config methods:

newWin.moveBy(x,y); move on x and y axis relative to the window.

..moveTo(x,y)

..resizeBy(width,height)

..resizeTo(width/height)

..scrollBy(x,y)

..scrollTo(x,y)

<elem>.scrollIntoView(top=true): Scroll the window to present the elem at top/bottom.

These methods only work reliably when the popup is opened by the window and has no additional tabs.

Events:

..onResize event.

..onScroll event.

..onblur event.

..onfocus event.

window.opener returns the window reference for the opener window, null for all windows except popups.

• Cross-window Communication: Rules and methods for effective Cross-Window Communication.

Same Origin Policy:

2 URLs are said to be same-origin if they have same protocol, domain and port. Domain includes the sub-level domain, tld and domain.

This policy is what affects cross window comms the most, it has to be true for all features to work.

Sub-level domain can still be ignored. Same-site can be configured to allow changes in sub-level domain, to do so set,

document.domain=’site.com’;

now any site with any sub-level domain (www/xyz/zzz etc.) can be used as long as the tld and domain are the same.

Iframe: An Iframe tag (<iframe src=’some url’ id=’myIframe’ name=’myName’> </iframe>) in an html document host a separate window embedded into the html page itself. It has it’s own document and window objects. It is like a popup except a new window isn’t opened and the current webpage holds the content.

myIframe.contentWindow; returns window reference for it.

myIframe.contentDocumet (shorthand for myIframe.contentWindow.document); to get doc ref. An iframe already has a doc even before the loading is complete, but it is flushed as soon as the loading completes and new document is inserted into it. To get the right doc, only call this method in an onload event handler.

Iframe follows the same origin policy just like popups.

Events:

..onload

..contentWindow.onload

The diff between the both is that the second one isn’t accessible for another origin sites while the first one is.

Collection: window.frames

window.frames[<int>] returns an iFrame object of the current window. The int index is specified to mention which one if there are multiple, 0 is the first one and the one encountered the first from top to bottom of the html page and so on.

window.frames.myName; returns an iframe with the given ‘name’ attribute.

window.parent: Returns immediate parent window.

window.top: Returns the topmost parent window.

Sandbox: An iframe can specify a ‘sandbox’ attribute, it applies limitations on what the iframe window can do. No value for this attribute applies the strictest limitations, we can specify limitations as well, these are space separated keys in a string that only need be mentioned to be activated:

‘allow-same-origin’

‘allow-top-navigation’ : allow changing parent.location.

‘allow-forms’: allow submitting forms

‘allow-scripts’ : to runs in iframe

‘allow-popups’ : allow window.open in iframe

.. and a few more.

Messaging: Despite Same-Origin, windows can send/receive messages.

<myIframe element>.postMessage(data, targetOrigin); data can be anything, it is cloned using structured serialization algorithm. targetOrigin can be used to specify which domain we want to accept the message from. ‘\*’ to allow all.

Event:

..onmessage: Triggered when a message is received and is from the target origin. The event has data, origin and source properties, source has the window ref for the source. We can use source.postMessage(…) to reply.

X-Frame-Options: A header that is set by the server serving the webpage that can have 3 values, DENY, SAMEORIGIN or ALLOW-FROM <domain>. This header defines if the webpage can be opened in an iframe or not. Hence why, Iframe src=twitter.com will return ‘refused to connect’.

It is recommended to use this header and ‘samesite’ cookie to disallow clickjacking attack on your website.

# Major References

1. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>
2. <https://www.ecma-international.org/publications/standards/Ecma-262.htm>