1. Numbers, Strings, and Operators

Floating point numbers are approximations

- `0.1 + 0.2 // = 0.30000000000000004`
  - 0.1 + 0.2 is not 0.3, but 0.30000000000000004 because floating numbers are approximations.
  - This is not a particular JavaScript issue, but the floating point number issue in general.

Shifting a number is multiplying or dividing by 2

- `1 << 2; // = 4`
  - It is 4, as 1 is multiplied by 2 (shift left) two times (the 2 in 1 « 2). It is the same as 2^2.
  - 1 « 3 is 8, because $2^3 = 8$, and 1 « 4 is 16.

JavaScript is a weakly typed language.

JavaScript will change the type if necessary

- `"1, 2, " + 3; // = "1, 2, 3"`
  - JavaScript changes the number 3 into a string "3"
- `"Hello " + ["world", "!"]; // = "Hello world,!"`
  - JavaScript changes the array into a sequence of strings.
- `13 + !0;` // 14
  - !0 is interpreted 1 because it is added to a number
- "13" + !0; // '13true'
  - !0 is interpreted true as it is concatenated to a string.

Use === not ==

- `"5" == 5`; // = true
  - This is dangerous, use === instead.
  - `"5" === 5`; // = false
- `null == undefined`; // = true
  - This is also dangerous, use === instead
  - `null === undefined`; // = false

JavaScript copies some Java methods with a twist

- "This is a string".charAt(0); // = 'T'
- "Hello world".substring(0, 5); // = "Hello"
- "Hello".length; // = 5
  - `length` is a property in JavaScript, so don't use () like Java.

Falsy and Truthy with a twist

- false, null, undefined, NaN, 0 and "" are falsy; everything else is truthy.
- Note that 0 is falsy and "0" is truthy, even though 0 == "0".

2. Variables, Arrays and Objects

shift (delete first) and unshift (add last)

```
// ['Hello', 45, true, 'Hello']
myArray.shift(); // shift is delete the first element
// => [45, true, 'Hello']
myArray.unshift(3); // Add as the first element
// => [3, 45, true, 'Hello']
someVar = myArray.shift(); // Remove first element and return it
// => [45, true, 'Hello']
```

push (add last) and pop (delete last)

```
myArray.push(3); // Add as the last element
// => [45, true, 'Hello', 3]
someVar = myArray.pop(); // Remove last element and return it
// => [45, true, 'Hello']
```

JavaScript list can have hybrid values

- var myArray0 = [32,false,"js",12,56,90];
- myArray0.join(";"); // = "32;false;js;12;56;90"

slice and splice

- myArray0.slice(1,4); // = [false,"js",12] <- 1,2,3, not 4
- myArray0.splice(2,4,"hi","wr","ld");
  - [32, false, "js",12,56,90] => [32,false,"hi","wr","ld"]
  - remove elements from 2 to 2 + 4, and add three more elements.

JavaScript Object is not exactly the same as JSON

Two ways to access the value in JavaScript Object

- Keys are strings, but quotes aren't required if they're a valid JavaScript identifier. Values can be any type.
- To access values, we can use the bracket [...] or the " notation.

```
var myObj = {key1: "Hello", "key2": "World"};
console.log(myObj["key1"]) // give string as a key
console.log(myObj.key2) // give name as a property
```

- When the key has a space in it, it should be a string.

```
var myObj = {myKey: "myValue", "my other key": 4};
myObj["my other key"]; // = 4
```

There is no exception when accessing unset values

- myObj.myKey; // = "myValue"
- console.log(myObj['myKey'])
- myObj.myFourthKey; // = undefined
  - If you try to access a value that's not yet set, you'll get undefined.
  - Compare Python and JavaScript.

```
// Python
>>> a['a']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'a'
// JavaScript
> a = {}
{}
> a['a']
undefined
```

3. Logic and Control Structures

For Loop

- The for loop is similar to Java's for loop.

```
for (var i = 0; i < 5; i++){
    // will run 5 times
    console.log(i)
}
```

- It's not a good coding habit, but we can break out of the multiple loop using a label. It is recommended to use a function and return from the function, instead of break using a label.

```
outer:
for (var i = 0; i < 10; i++) {
    for (var j = 0; j < 10; j++) {
        if (i == 5 && j ==5) {
            break outer;
            // breaks out of outer loop instead of only the inner one
        }
    }
}
```

For each loop with 'of' and 'in' over iterables

- We can use the for each loop.
- In this example, we can iterate over an iterables (an array, for example) to get values using `of`.

```
var myPets = "";
var pets = ["cat", "dog", "hamster", "hedgehog"];
for (var pet of pets){ // of will return value
    myPets += pet + " ";
} // myPets = 'cat dog hamster hedgehog '
```

- String is another type of iterable, so we can use "of" to iterate over it.

```
var myPets = "";
var str = "Hello"
for (var p of str){ // of will return value
    myPets += p + " ";
} // myPets = 'H e l l o '
```

- When we use `in`, we get key values. In this example, we learn that the keys of an array is 0,1,2, ...

```
var myPets = "";
var pets = ["cat", "dog", "hamster", "hedgehog"];
for (var pet in pets){
    myPets += pet + " ";
} // myPets = '0 1 2 3 '
```

- We can use `const` instead of `var` to prevent

```
var myPets = "";
var pets = ["cat", "dog", "hamster", "hedgehog"];
for (var pet of pets){
    myPets += pet + " ";
    pet = "elephant" // This is an error
}
```

For each loop with 'in' over objects

- When we iterate over an object(dictionary), we get keys in `x`. So, we need to use it to access values.

```
var description = "";
var person = {fname:"Paul", lname:"Ken", age:18};
for (var x in person){ // get all the values
    description += person[x] + " ";
} // description = 'Paul Ken 18 '
```

Object functions

- We can use Object.values() function to get a values array of a dictionary.
- Be ware that we use `of` not `in` to get values.

```
description = "";
var person = {fname:"Paul", lname:"Ken", age:18};
for (var x of Object.values(person)){ // get all the values
    description += x + " ";
} // description = 'Paul Ken 18 '
console.log(description)
```

- We can use Object.keys() function to get a keys array of a dictionary.

```
description = ""
for (var x of Object.keys(person)){
    description += x + " ";
} // description = 'fname lname age '
console.log(description)
```

- We can use Object.entries() function to get an (key, value) array of array.

```
var entries = Object.entries(person)
// entries = [ [ 'fname', 'Paul' ], [ 'lname', 'Ken' ], [ 'age', 18 ] ]
description = "";
description = "";
for (const i in entries){
    description += entries[i][0] + " " + entries[i][1] + ",";
} // description = 'fname Paul,lname Ken,age 18,'
console.log(description)
```

Do While loop

```
var sum = 0;
var i = 0;
do {
   sum += i;
   i++;
} while (i < 5);
console.log(sum)
```

- We can use do while loop.

Shortcut

- We can use the `||` operator to set a default value.
- `undefined || A` is interpreted as `A`
- In this example, otherName is `undefined`. So, the "default" value is assigned instead.

```
var otherName
var name = otherName || "default";
```

4. Functions, Scope, and Closures

Bug

- When we separate the return and return value, undefined is returned. This may lead to a hard-to-find bug.

```
function myFunction(){
    return // <- semicolon automatically inserted here
    {thisIsAn: 'object literal'};
}
myFunction(); // = undefined
```

setTimeOut()

- We can use the setTimeOut(f, time) function to call the function f, after time later.
- 10 means 10 milliseconds, so for 5 seconds, we should give 5000.
- 'before' is printed out immediately, and 'after' will be printed in 10 milliseconds.

```
setTimeout(() => {
    console.log('after ');
}, 10);
console.log(' before ');
>> before
>> after
```

setInterval() and clearInterval()

- We can use setInterval() function to call functions repetidely.
- To finish the repetition, we need to For this we need to call clearInverval() function.
- We need to get an id and use it as an argument to the clearInverval() function.

```
var i = 0
function myFunction(){
    // this code will be called every 5 seconds
    console.log(`Hello ${i++}`)
    if (i == 10){
        clearInterval(id)
    }
}
```

```
var id = setInterval(myFunction, 50);
```

Scope

- The let is block scoped, so let variables are not accessible outside the block.
- The var is not block scoped, so it can be used

```
{
    let il = 1;
    var iv = 2;
}
// il; // error
iv; // 2
```

- It's the same as the block in a if statement.

```
if (true) {
    var ivf = 2;
}
ivf; // 2
```

IIFE (Immediately Invoked Function Expression)

- IIFE produces a lexical scope using function scoping.
- It prevents temporary variables defined in a function from polluting the global environments.
- In this example, the temporary variable is not accessible from outside of the block.

```
var permanent =  20;
// This has led to a common pattern of "IIFE",
// which prevents temporary variables from leaking into the global
// scope.
(function() {
    var temporary = 5;
    permanent = 10;
})();
// temporary; // raises ReferenceError
permanent; // = 10
```

Inner function and Closure

- sayHelloInFiveMilliSeconds() function uses its argument name to make a prompt string.
- Then, the prompt is used in the inner function.
- The setTimeout() function can call the inner function.

```
function sayHelloInFiveMilliSeconds(name){
```

```
    var prompt = "Hello, " + name + "!";
    function inner(){
        console.log(prompt);
    }
    setTimeout(inner, 500);
}
sayHelloInFiveMilliSeconds("Adam"); // will open a popup with "Hello, Adam!" in 5ms
```

5. More about Objects; Constructors and Prototypes

JavaScript Object

- myObj references an object whose element is a function.
- We can use arrow function.

```
var myObj = {
    myFunc: function(){
        return "Hello world!";
    }
};
var myObj = {
    myFunc: () => "Hello world!"
};
myObj.myFunc();
```

The `this` in a method and its context in an object

- this in any object means "this object." So, we can access other entities in the same object.
- So, the `this` in the myFunc references this object.

```
myObj = {
    myString: "Hello world!",
    myFunc: function(){
        return this.myString;
    }
};
myObj.myFunc(); // = "Hello world!"
```

- When we assign a variable to the object method myFunct and call it, the "this.myString" will be undefined because this in the myFunc() has no connection with the myObj.

```
// myObj.myFunc ---> [function object this]
//                                  ^
// myObj ---> [object with myString myFunc] <= myFunc knows this
```

```
var myFunc = myObj.myFunc;
myFunc(); // = undefined
```

- To resolve this issue, we explicitly bind myObj to myFunc.

```
var myFunc = myObj.myFunc.bind(myObj);
console.log(myFunc()); // Output: Hello world!
```

`this` in a function

- A function can be assigned to the object and gain access to it through `this`, even if it wasn't attached when it was defined.
- It is somewhat confusing, but we can understand it as it's a normal function, not a method attached to an object.
- Also, we can understand this is how we extend JavaScript features using the prototype approach.

```
var myOtherFunc = function(){
    return this.myString.toUpperCase();
};
myObj.myOtherFunc = myOtherFunc;
myObj.myOtherFunc(); // = "HELLO WORLD!"
```

- The otherFunction has two variables: argument s and `this`.
- We can use call() function to give an argument (myObj) for `this` and 2nd argument for s.

```
var anotherFunc = function(s){
    return this.myString + s;
};
anotherFunc.call(myObj, " And Hello Moon!");
```

- We can use this feature frequently in JavaScript.
- We should give a series of values, not an array to the Math.min function.

```
Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (uh-oh!)
```

Apply

- We can use call() that has both `this` and input argument.
- Remember that in functional programming, it should be in the higher-order function format
- (call anotherFunc myObj, "...") => anotherFunc.call(myObj, "...")
- However, in JavaScript, the order is reversed.

```
var anotherFunc = function(s){
    return this.myString + s;
};
```

```
anotherFunc.call(myObj, " And Hello Moon!");
// = "Hello World! And Hello Moon!"
```

- In this case, we can use the apply() function when the argument is in an array.

```
anotherFunc.apply(myObj, [" And Hello Sun!"]);
// = "Hello World! And Hello Sun!"
```

Math.min with an array input

- We should use multiple arguments to use Math.min; when we give an array argument, we will have an error.

```
Math.min(42, 6, 27); // = 6
Math.min([42, 6, 27]); // = NaN (uh-oh!)
```

- We can use apply to address this issue. Don't forget that Math is also an object.

```
Math.min.apply(Math, [42, 6, 27]); // = 6
```

- Or, we can use the ... operator to transform an array into a series of values.

```
Math.min(...[42, 6, 27])
```

bind

- The bind() function finds a function to an object.
- In this example, boundFunc knows the `this` of myObj object.

```
var boundFunc = anotherFunc.bind(myObj);
boundFunc(" And Hello Saturn!"); // = "Hello World! And Hello Saturn!"
```

Partially applying or currying

- We can use the bind function to apply only a part of the inputs.
- In this example, the doubler function binds the first argument as 2.
- So, we can give the second argument as an argument of the double function.

```
var product = function(a, b){ return a * b; };
var doubler = product.bind(this, 2);
doubler(8); // = 16
```

`this` in a Constructor

- A constructor is a function that can make a new object using a new keyword.
- It should have `this` to have fields.

```
var MyConstructor = function(){
    this.myNumber = 5;
};
myNewObj = new MyConstructor(); // = {myNumber: 5}
myNewObj.myNumber; // = 5
```

Inheritance in JavaScript (bad way)

- Let's say we have an object that we can inherit from.

```
var myPrototype = {
    meaningOfLife: 42,
    myFunc: function(){
        return this.myString.toLowerCase();
    }
};
```

- Then, we can set the **proto** to use myPrototype.

```
var myObj = {
    myString: "Hello world!"
};
myObj.__proto__ = myPrototype;
myObj.meaningOfLife; // = 42
myObj.myFunc(); // = "hello world!"
```

- We can even update the prototye of myPrototype.

```
myPrototype.__proto__ = {
    myBoolean: true
};
myObj.myBoolean; // = true
```

- Don't forget that we can change the prototype values as there's no copying involved.

```
myPrototype.meaningOfLife = 43;
myObj.meaningOfLife; // = 43
```

- The foreach loop will display all the prototypes connected.
- So, we need to filter out the prototype values using the hasOwnProperty() method.

```
for (var x in myObj){
    console.log(myObj[x]);
}
///prints:
// Hello world!
// 43
// [Function: myFunc]
```

```
// true

for (var x in myObj){
    if (myObj.hasOwnProperty(x)){
        console.log(myObj[x]);
    }
}
```

Constructors

- We can use `this` to specify the field in each instance.
- But the prototype values are common to all instances.

```
// constructors
function MyConstructor2 (name) { // constructor with an argument
    this.name = name;
    this.hello = function() {
      console.log('Hello ' + this.name)
    }
}
// all the this attributes are created for each instance
var person1 = new MyConstructor2('Sam');
var person2 = new MyConstructor2('John');
person2.hello() // 'Hello John'
person1.hello() // 'Hello Sam'
```

Extending using prototype (careful!)

- We can extend existing constructor's features using prototype.
- Be careful when we use it, as the `this` in the prototype is not instance's this; it's equivalent to a class varible in OOP.

```
MyConstructor2.prototype = {
    myNumber: 5,
    getMyNumber: function(){
        return this.myNumber;
    }
};
var myNewObj2 = new MyConstructor2();
myNewObj2.getMyNumber(); // = 5
console.log(myNewObj2.getMyNumber())
myNewObj2.myNumber = 6;
myNewObj2.getMyNumber(); // = 6
console.log(myNewObj2.getMyNumber())
```

Extending using prototype (it's OK)

- However, we can use `this` to refer a method, not a field, as we can share methods.

```
String.prototype.firstCharacter = function(){
    return this.charAt(0);
};
"abc".firstCharacter(); // = "a"
```

Object and primitive value comparison

- == compares if two values are printing the same thing.
- === compares if they are the same or not. Number object is different from the primitive number.

```
var myNumber = 12;
var myNumberObj = new Number(12);
myNumber == myNumberObj; // = true
myNumber === myNumberObj; // = false
```

- We can use `typeof` operator to get the type.

```
typeof myNumber; // = 'number'
typeof myNumberObj; // = 'object'
myNumber === myNumberObj; // = false
```

- So 0 is falsy, but Number(0) is not as it's an object.

How to implement newer features of JavaScript in an older subset

- Let's say we don't have Object.create not implemented, but we need to use it.
- We can use this "polyfilling" method to do that.

```
if (Object.create === undefined){ // don't overwrite it if it exists
    Object.create = function(proto){
        // make a temporary constructor with the right prototype
        var Constructor = function(){};
        Constructor.prototype = proto;
        // then use it to create a new, appropriately-prototyped object
        return new Constructor();
    };
}
```

Lambda expression

- Lambda expression was introduced in ES6.

- It does not support hoisting.

```
console.log(add(1, 8)) // This is an error as add should be defined.
const add = (firstNumber, secondNumber) => {
    return first number + secondNumber;
};
```

- Due to hoisting, this is OK.

```
console.log(add2(1, 8))
function add2(firstNumber, secondNumber) {
    return firstNumber + secondNumber;
};
```

- Also, we cannot use this in the lambda expression.

```
var o = {
    firstNumber: 1,
}
console.log(add3.call(o, 8))

function add4(secondNumber) {
    return this.firstNumber + secondNumber;
};

console.log(add4.call(o, 8))
```