Data Types

1. Primitive
   1. String
   2. Number
   3. Boolean
   4. Null
   5. Undefined
2. Objects
   1. Objects
   2. Arrays
   3. Functions
3. Loosely type
4. Type of
5. NaN -> type of NaN, NaN === NaN

Loosely type

var a = ‘2’
var b = 2

if(b == a)
Conversion

Everything is loosely typed
So for that typeof
typeof (‘1’) string
typeof (1) number

Array

Var a = [ ];

Var a = new Array();

Var a = [1,2,3];

A.length

```
Var a = [1, 2 , 3 , “hello world”];
```

It works because array internally is an object so in object you can have any key value pair.

```
Var a = [1, 2 , 3 , “hello world”, function(){console.log()}];
```

```
a[3]() works
```

Array of functions;

Empty an array

```
A = [1,2,3];

B  =  A;

A = [];
```

Makes array empty

Problem is a and b are referring to the same memory b would still point to memory even when a points to empty it won't be garbage collected.

```
A.lentgth = 0
```

Memory reference is gone;

Function in function

```
var a = function

function try(fn) {
  console.log(fn());
}

try(function(){return 8});


Var try = Function(){
  Return 8;
}

try();
```

Javascript everything is an object

```
Var obj ={a:1, b:2};
```

Accessing the values of an object

```
Obj.a
obj[‘a’]
```

Why we do this ?
This is because we would want to get data where we know the name of the property then it would be possible with

```
Var z= ‘a’;

Obj[z];
```

Ways to create an object

```
Var obj = {};

Var obj = new Object();
```

Object is a constructor function
Array
String

Each constructor function has methods inside it.

```
Var dog =  {
	“Name”: “golu”,
	“Bread”: “pom”
}
```

Object inside object to create JSON

```
Var dog =  {
        “Name”: “golu”,
        “Bread”: “pom”,
Owner: {
        “Name”: “Siddharth”
        }
}
```

Function inside an object

```
Var dog =  {
        “Name”: “golu”,
        “Bread”: “pom”,
        Bark: function() {
                console.log(‘Dog barks’);
        }
}
```

Dog.bark

```
Var dog =  {
        “Name”: “golu”,
        “Bread”: “pom”,
        Bark: function() {
                console.log(this.name+”  ”+“ barks’);
        }
}
```

This and concatination

```
Function Dog() {
        console.log(a);
        console.log(arguments);
}

Dog(1);
```

Arguments is an array like structure

What is an array like structure

```
Function Dog(name, breed) {
	This.name = name;
	This.breed = breed;
}
```

```
Var d = new Dog(‘golu’, ‘pom’);
```

Default constructor function that we created

What is prototype

hasOwnProperty false as it is borrowed method.

```
Dog.prototype.barks = function() {
	console.log(‘barks’);
}
```

```
d.barks();
```

__proto__

Very powerful used exactly for inheritence

`null` **means empty or non-existent value which is used by programmers to indicate “no value”. null is a primitive value and you can assign null to any variable. null is not an object, it is a primitive value. For example, you cannot add properties to it. Sometimes people wrongly assume that it is an object, because typeof null returns "object".**

`undefined` **means, value of the variable is not defined. JavaScript has a global variable** `undefined` **whose value is "undefined" and** `typeof undefined` **is also "undefined".**

Remember, undefined is not a constant or a keyword. undefined is a type with exactly one value: undefined. Assigning a new value to it does not change the value of the type undefined.

typeof(null) “object”

typeof(undefined) “undefined”

Null == undefined

NaN == NaN false check for NaN

typeof(NaN) is number

+’abc’

NaN

Difference between == and ===

Type conversion in ==
Type check in ===

Null == undefined true
Null === undefined false

1 == ‘1’ true

1 === ‘1’ false

Convert arguements objects to array

Array.prototype.slice.call(arguments);

```
function isTwoPassed(){
  var args = Array.prototype.slice.call(arguments);
  return args.indexOf(2) != -1;
}

isTwoPassed(1,4) //false
isTowPassed(5,3,1,2) //true


function abc() {
  console.log(a);
}


function abc() {
  var a = 10;
  console.log(a);
}
```

Hoisting

```
var z = 2;
function check() {
  console.log(z);
  var z = 10;
}

check();



function log(){
  var args = Array.prototype.slice.call(arguments);
  args.unshift('(app)');
  console.log.apply(console, args);
}

log('my message',’hello’);
log('my message', 'your message'); //(app) my message your message
```

```
var animal = {
  eats: true,
  walk: function() {
    alert("Animal walk");
  }
};

var rabbit = {
  jumps: true,
  __proto__: animal
};

console.log(rabbit);

var longEar = {
  earLength: 10,
  __proto__: rabbit
};

console.log(longEar);

rabbit.walk();
```

```
var animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

var rabbit = {
  __proto__: animal
};

rabbit.walk();

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};
```

```
var a = {
  x: 10,
  calculate: function (z) {
    return this.x + this.y + z;
  }
};

var b = {
  y: 20,
  __proto__: a
};

var c = {
  y: 30,
  __proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```

```
// a constructor function
function Foo(y) {
  // which may create objects
  // by specified pattern: they have after
  // creation own "y" property
  this.y = y;
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:

// inherited property "x"
Foo.prototype.x = 10;
```

```
// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

// let's show that we reference
// properties we expect

console.log(

  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is a
  // reference to the constructor function itself;
  // instances "b" and "c" may found it via
  // delegation and use to check their constructor

  b.constructor === Foo, // true
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo, // true

  b.calculate === b.__proto__.calculate, // true
  b.__proto__.calculate === Foo.prototype.calculate // true

);
```

```
function Mammal(name){

    this.name=name;
    this.offspring=[];
}
Mammal.prototype.haveABaby=function(name){
    var newBaby=new Mammal(name +" "+this.name);
    this.offspring.push(newBaby);
    return newBaby;
}


function Cat(name){
    this.name=name;
}
Cat.prototype = new Mammal();        // Here's where the inheritance occurs
Cat.prototype.constructor=Cat;       // Otherwise instances of Cat would have a constructor of Mammal




var someAnimal = new Mammal('Mr. Biggles');
var myPet = new Cat('Felix');
console.log('someAnimal is '+someAnimal.name);   // results in
console.log('myPet is '+myPet.name);             // results in


myPet.haveABaby('Big');                    // calls a method inherited from Mammal
myPet.haveABaby('Small');                    // calls a method inherited from Mammal


console.log(myPet.offspring.length);        // shows that the cat has one baby now
console.log(myPet.offspring);          // results in '[Mammal "Baby Felix"]'
```

Closure in Javascript

Closure is retaining the scope of a variable even after the function has returned.

```
function makeWorker() {
  var name = "Pete";

  return function() {
    alert(name);
  };
}

var name = "John";

// create a function
var work = makeWorker();

// call it
work();
```

```
function makeCounter() {
  var count = 0;

  return function() {
    return count++; // has access to the outer "count"
  };
}

var counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

```
function makeCounter() {
  var count = 0;
  return function() {
    return count++;
  };
}

var counter1 = makeCounter();
var counter2 = makeCounter();

alert( counter1() ); // 0
alert( counter1() ); // 1

alert( counter2() ); // 0 (independent)
```

Closure inside loops

```
 for(var i = 0; i < 10; i++) {
    setTimeout(function() {
      console.log(i);
    }, 10);
}

for(var i = 0; i < 10; i++) {
    setTimeout((function(i) {
      console.log(i);
    })(i), 10)
}

for(var i = 0; i < 10; i++) {
  setTimeout(console.log.bind(console, i), 10);
}
```

In other words, a **closure** gives you access to an outer function's scope from an inner function. In **JavaScript**, **closures** are created every time a function is created, at function creation time. To **use** a **closure**, define a function inside another function and expose it.

```
Self invoking functions in js

(function(){
      console.log(1)
})()

Usage to create scopes


Currying
```

```
function add(x,y){
     if(arguments.length > 1) {
          return x+y;
     }else if(arguments.length == 1){
    return function (y){
        return x+y;
    }
     }
}

add(2,3);

add(2)(3);
```

```
var obj = {   // every method returns obj---------v
    first: function() { console.log('first');   return obj; },
    second: function() { console.log('second'); return obj; },
    third: function() { console.log('third');   return obj; }
}
```

```
obj.first().second().third();
```

```
Inheritance in es6

class User {
     constructor(name, age) {
     this.name = name;
    this.age = age;
  }

  incrementAge() {
     return ++this.age;
  }

}

var u = new User('sid', 25);
alert(u.incrementAge());
alert(u.incrementAge());

class Admin extends User {
  constructor(name, age, role) {
    super(name, 25);
    this.role = role;
  }

}

var a= new Admin('sid', 25, 'admin');
alert(a.incrementAge());
alert(a.incrementAge());
```

https://codesandbox.io/s/keen-hopper-g65evm?file=/src/index.js

```
Arrow functions
```

```
Spread operator
```

```
class
```

```
var
let
const
```

```
hoisting in var
block scope in let const
redeclaration in var
```

```
String.prototype.reverse = function(){
return this.split('').reverse().join('');
}

var str = 'hello world';
str.reverse();
```

```
Array.prototype.duplicator = function(){
  return this.concat(this);
}

[1,2,3,4,5].duplicator();
```

```
var num = 10,
    name = "Addy Osmani",
    obj1 = {
      value: "first value"
    },
    obj2 = {
     value: "second value"
    },
    obj3 = obj2;

function change(num, name, obj1, obj2) {
    num = num * 10;
    name = "Paul Irish";
    obj1 = obj2;
    obj2.value = "new value";
}
```

```
change(num, name, obj1, obj2);

console.log(num);
console.log(name);
console.log(obj1.value);
console.log(obj2.value);
console.log(obj3.value);
```

```
const array = [1, 2, 3];
const obj = { ...array }; // { 0: 1, 1: 2, 2: 3 }
```

```
function myFunction(x, y, z) {}
const args = [0, 1, 2];
myFunction.apply(null, args);

function myFunction(x, y, z) {}
const args = [0, 1, 2];
myFunction(...args);
```

```
const arr = [1, 2, 3];
const arr2 = [...arr];
```

```
function myFunction(v, w, x, y, z) {}
const args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

```
const parts = ["shoulders", "knees"];
const lyrics = ["head", ...parts, "and", "toes"];
```

```
let arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

// Append all items from arr2 onto arr1
arr1 = arr1.concat(arr2);
```

```
let arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

arr1 = [...arr1, ...arr2];
```

```
const obj1 = { foo: "bar", x: 42 };
const obj2 = { foo: "baz", y: 13 };

const clonedObj = { ...obj1 };
// { foo: "bar", x: 42 }

const mergedObj = { ...obj1, ...obj2 };
// { foo: "baz", x: 42, y: 13 }
```

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(element => {
    return multiplier * element
  })
}

let arr = multiply(2, 1, 2, 3)
console.log(arr)
```