# Javascript data structures **What is a data structure?**

In computer science, a data structure is ****a format to organize, manage and store data**** in a way that allows ****efficient access and modification****.

More precisely, a data structure is a ****collection of**** ****data**** ****values****, the ****relationships**** among them, and the functions or ****operations**** that can be applied to that data.  
  
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Object/fromEntries

Arrays:  
An ****array**** is a collection of items stored at contiguous memory locations.

Each item can be accessed through its ****index**** (position) number. Arrays always start at index 0, so in an array of 4 elements we could access the 3rd element using the index number 2.

const arr = ['a', 'b', 'c', 'd']

console.log(arr[2]) // c

let b = ['a','b','c']

console.log('check c',a[2])

check c c  
  
let arrVal = [1,2,3];

undefined

arr.indexOf(3)

-1

arrVal.indexOf(2)

1

arrVal.indexOf(1)

0  
arrVal.push(4)

4

arrVal

(4) [1, 2, 3, 4]  
  
const indexOfTwo = arrVal.indexOf(2);

indexOfTwo

1

console.log(arrVal[indexOfTwo -1]);  
1

console.log(arrVal[indexOfTwo+1])

3

// A collection of the values 1, 2 and 3const arr = [1, 2, 3]

// Each value is related to one another, in the sense that each is indexed in a position of the arrayconst indexOfTwo = arr.indexOf(2)

console.log(arr[indexOfTwo-1]) // 1

console.log(arr[indexOfTwo+1]) // 3

// We can perform many operations on the array, like pushing new values into it

arr.push(4)

console.log(arr) // [1,2,3,4]

In JavaScript that's not the case, as we can store ****values of any type**** in the same array and the ****length**** of it can be ****dynamic**** (it can grow or shrink as much as necessary).

const arr = ['store', 1, 'whatever', 2, 'you want', 3]

Any data type can be stored in an array, and that includes arrays too. An array that has other arrays within itself is called a ****multidimensional array****.

const arr = [

[1,2,3],

[4,5,6],

[7,8,9],]

Checking Unique Array with IDs and Removed duplicate:  
let checkUniqueIds = [...new Set([1,22,33,33,33,33,444,444,555,55,22,2,3,4,4,4,444,55,])]  
console.log(checkUniqueIds)

(9) [1, 22, 33, 444, 555, 55, 2, 3, 4]  
  
storing data inside array in different ways:  
let arrDynamic = ['saikumar',1,'vijaykumar',2,'vinaykumar',3]

arrDynamic

1. *(6) ['saikumar', 1, 'vijaykumar', 2, 'vinaykumar', 3]*

let arrMultiDimensions = []

let arrMultiDimensions = [[1,2,3],[4,5,6],[7,8,9]]

arrMultiDimensions

1. *(3) [Array(3), Array(3), Array(3)]*
   1. **0**: (3) [1, 2, 3]
   2. **1**: (3) [4, 5, 6]
   3. **2**: (3) [7, 8, 9]
   4. **length**: 3

const arr = ['a', 'b', 'c', 'd']

console.log(arr[2]) // c

const arr = ['a', 'b', 'c', 'd']

console.log(arr.length) // 4

# **Objects (hash tables)**

In JavaScript, an ****object**** is a collection of ****key-value pairs****. This data structure is also called ****map****, ****dictionary**** or ****hash-table**** in other programming languages.

A typical JS object looks like this:

const obj = {

prop1: "I'm",

prop2: "an",

prop3: "object"  
}

To access properties you can use two different syntaxes, either object.property or object["property"]. To access methods we call object.method().  
const obj = {

prop1: "Hello!",

prop3: function() {console.log("I'm a property dude!")}}

console.log(obj.prop1) // "Hello!"

console.log(obj["prop1"]) // "Hello!"

obj.prop3() // "I'm a property dude!"  
  
obj.prop4 = 125

obj["prop5"] = "The new prop on the block"

obj.prop6 = () => console.log("yet another example")

console.log(obj.prop4) // 125

console.log(obj["prop5"]) // "The new prop on the block"

obj.prop6() // "yet another example"

test:

console.log(obj.prop1)

VM215:1 hellow  
console.log(obj['prop1'])

VM324:1 hellow

console.log(obj['prop2'])

VM447:1 ƒ (){

console.log(" this is saikumar! ...");

}

undefined

obj.prop2()

VM135:4 this is saikumar! …

let obj = {

prop1:"hellow",

prop2:function () {

conosle.log(" This is saikumar!.......");

}

}

undefined

obj

{prop1: 'hellow', prop2: ƒ}  
prop1: "hellow"  
prop2: ƒ ()[[Prototype]]: Objectconstructor: ƒ Object()hasOwnProperty: ƒ hasOwnProperty()isPrototypeOf: ƒ isPrototypeOf()propertyIsEnumerable: ƒ propertyIsEnumerable()toLocaleString: ƒ toLocaleString()toString: ƒ toString()valueOf: ƒ valueOf()\_\_defineGetter\_\_: ƒ \_\_defineGetter\_\_()\_\_defineSetter\_\_: ƒ \_\_defineSetter\_\_()\_\_lookupGetter\_\_: ƒ \_\_lookupGetter\_\_()\_\_lookupSetter\_\_: ƒ \_\_lookupSetter\_\_()\_\_proto\_\_: (...)get \_\_proto\_\_: ƒ \_\_proto\_\_()set \_\_proto\_\_: ƒ \_\_proto\_\_()

obj.prop3 = 'Sai';

'Sai'

obj['prop4'] = 'Vijay kumar pusam';

'Vijay kumar pusam'

obj.prop5 =() => { console.log('Hellow , this is vinay kumar ....')};

() => { console.log('Hellow , this is vinay kumar ....')}

obj.prop5();

1. VM2131:1 Hellow , this is vinay kumar .…   
     
   finally:  
   *{prop1: 'hellow', prop3: 'Sai', prop4: 'Vijay kumar pusam', prop2: ƒ, prop5: ƒ}*
   1. **prop1**: "hellow"
   2. **prop2**: *ƒ ()*
   3. **prop3**: "Sai"
   4. **prop4**: "Vijay kumar pusam"
   5. **prop5**: *() => { console.log('Hellow , this is vinay kumar ....')}*

Syntax:  
new Object()

new Object(value)

Object()

Object(value)

Object()

{}

Object('sai')

String {'sai'}

0: "s"

1: "a"

2: "i"

length: 3[[Prototype]]: String[[PrimitiveValue]]: "sai"

new Object()

{}

new Object('saikumar')

String {'saikumar'}

### [Creating a new Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Object" \l "creating_a_new_object)

const o = new Object();

o.foo = 42;

console.log(o);// { foo: 42 }  
let objectval = new Object();

undefined

objectval.Name = 'saikumar';

'saikumar'

console.log('OBJECT :',objectval)

VM322:1 OBJECT : {Name: 'saikumar'}

Types:  
const o = new Object();

Const o = new Object(undefined)

let o = new Object(undefined);

Undefined

Cosole.log(0) // {}

o.Name = 'sai';

'sai'

Console.log(o)

{Name: 'sai'}  
  
let o = new Object(null)

undefined

o

{}

o.Name = 'saikumar pusam'

'saikumar pusam'

o

{Name: 'saikumar pusam'}

### [btaining wrapper objects for BigInt and Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/Object" \l "obtaining_wrapper_objects_for_bigint_and_symbol)

const numberObj = new Number(1);

console.log(typeof numberObj); // "object"

const numberObj = new Number(1,2,3);

Number{

1

}

const bigIntObj = BigInt(1n);

bigIntObj // 1n

typeof bigIntObj // 'bigint'

bigInt : BigInt {1n}

typeof bigINTobject // 'object'

const bigintObj = Object(1n);

console.log(typeof bigintObj); // "object"

const symbolObj = Object(Symbol("foo"));

console.log(typeof symbolObj); // "object"

let symbolObj = Object(Symbol('saikumar'));

Undefined

symbolObj

Symbol {Symbol(saikumar), description: 'saikumar'}description: "saikumar"[[Prototype]]: Symbol[[PrimitiveValue]]: Symbol(saikumar)

symbolObj.description

'saikumar'

console.log(typeof symbolObj);

VM966:1 object

#### Object.assign():

let target = {a:1,b:3,c:3};

let source = {b:2,c:4}

let assignObj = Object.assign(target,source);

assignObj

{a: 1, b: 2, c: 4}  
  
console.log(target)

VM1865:1 {a: 1, b: 2, c: 4}

console.log(assignObj === target)

VM2021:1 true  
  
const o1 = { a: 1 };

const o2 = { b: 2 };

const o3 = { c: 3 };

const obj = Object.assign(o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

console.log(o1); // { a: 1, b: 2, c: 3 }, target object itself is changed.

const o1 = { a: 1, b: 1, c: 1 };

const o2 = { b: 2, c: 2 };

const o3 = { c: 3 };

const obj = Object.assign({}, o1, o2, o3);

console.log(obj); // { a: 1, b: 2, c: 3 }

Test:

let o1 = {a:1};

let o2 = {b:2}

let o3 = {c:3}

Object.assign(o1,o2,o3)

{a: 1, b: 2, c: 3}

Object.assign({},o1,o2,o3)

{a: 1, b: 2, c: 3}

o1

{a: 1, b: 2, c: 3}

let ob1 = {a:1,b:3,c:5};

let ob2 = {b:2,c:3}

let ob3 = {c:10}

undefined

Object.assign({},ob1,ob2,ob3)

{a: 1, b: 2, c: 10}

ob1

{a: 1, b: 3, c: 5}

#### Object.create()

const person = {

isHuman: false,

printIntroduction: function () {

console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);

},

};

const me = Object.create(person);

me.name = 'Matthew'; // "name" is a property set on "me", but not on "person"

me.isHuman = true; // Inherited properties can be overwritten

me.printIntroduction();

// Expected output: "My name is Matthew. Am I human? true"  
  
  
const a = {

isHuman:false,

printFunc:function () {

console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);

}

}

const M = Object.create(a);

M.name = 'saikuamr';

M.isHuman = true;

M.printFunc();

VM4894:4 My name is saikuamr. Am I human? True

#### Object.defineProperties()

const obj = {};

Object.defineProperties(obj, {

property1: {

value: true,

writable: true,

},

property2: {

value: "Hello",

writable: false,

},

// etc. etc.

});

const obj4 = {};

Object.defineProperties(obj4,{

property1: {

value:true,

},

property2: {

value:"SAI KUMAR"

}

});

{property1: true, property2: 'SAI KUMAR'}

#### Object.defineProperty()

const object1 = {};

Object.defineProperty(object1, 'property1', {

value: 42,

writable: false,

});

object1.property1 = 77;

// Throws an error in strict mode

console.log(object1.property1);

// Expected output: 42  
  
onst def = {};

Object.defineProperty(def,'Student',{

value:'Saikumar',

writable:true

});

{Student: 'Saikumar'}

def.Student = 'Pusam';

'Pusam'

def

{Student: 'Pusam'}

#### Object.entries()

const object1 = {

a: 'somestring',

b: 42,

};

for (const [key, value] of Object.entries(object1)) {

console.log(`${key}: ${value}`);

}

// Expected output:

// "a: somestring"

// "b: 42"

const obj = {

a:"saikumar",

b:29

}

for(const[key,value] of Object.entries(obj)){

console.log(`${key} : ${value}`);

}

a : saikumar

b : 29  
  
Object.entries(obj)

1. *(2) [Array(2), Array(2)]*
   1. **0**: (2) ['a', 'saikumar']
   2. **1**: (2) ['b', 29]
   3. **length**: 2
   4. [[Prototype]]: Array(0)

console.log(Object.entries(obj))

VM6429:1

1. *(2) [Array(2), Array(2)]*
   1. **0**: (2) ['a', 'saikumar']
   2. **1**: (2) ['b', 29]

Object.entries(obj)

1. *(2) [Array(2), Array(2)]*
   1. **0**: (2) ['a', 'saikumar']
   2. **1**: (2) ['b', 29]
   3. **length**: 2
   4. [[Prototype]]: Array(0)

console.log(Object.entries(obj))

VM6429:1

1. *(2) [Array(2), Array(2)]*
   1. **0**: (2) ['a', 'saikumar']
   2. **1**: (2) ['b', 29]

// Strings have indices as enumerable own properties

console.log(Object.entries("foo")); // [ ['0', 'f'], ['1', 'o'], ['2', 'o'] ]

// Other primitives except undefined and null have no own properties

console.log(Object.entries(100)); // []

const obj = { foo: "bar", baz: 42 };

const map = new Map(Object.entries(obj));

console.log(map); // Map(2) {"foo" => "bar", "baz" => 42}

// Using for...of loop

const obj = { a: 5, b: 7, c: 9 };

for (const [key, value] of Object.entries(obj)) {

console.log(`${key} ${value}`); // "a 5", "b 7", "c 9"

}

// Using array methods

Object.entries(obj).forEach(([key, value]) => {

console.log(`${key} ${value}`); // "a 5", "b 7", "c 9"

});

obj

{a: 'saikumar', b: 29}

Object.entries(obj).forEach(([key,value]) => {

console.log(`${key} : ${value}`);

})

VM6612:2 a : saikumar

VM6612:2 b : 29

let a = [];

let b = [];

Object.entries(obj).forEach(([key,value]) => {

a.push(key);

b.push(value);

console.log(a,b)

})

keys : ['a']0: "a"1: "b"length: 2[[Prototype]]: Array(0)

VM6982:7 values ['saikumar']0: "saikumar"1: 29length: 2[[Prototype]]: Array(0)

VM6982:6 keys : (2) ['a', 'b']0: "a"1: "b"length: 2[[Prototype]]: Array(0)

VM6982:7 values (2) ['saikumar', 29]0: "saikumar"1: 29length: 2[[Prototype]]: Array(0)  
  
  
obj

{a: 'saikumar', b: 29}

const map = new Map(Object.entries(obj))

console.log("map :",map)

// {'a' => 'saikumar', 'b' => 29}

let objNew = { name:"sai kumar", age:"29", DOB:"09-11-1994" } console.log(Object.entries(objNew));

1. *(3) [Array(2), Array(2), Array(2)]*
   1. **0**: (2) ['name', 'sai kumar']
   2. **1**: (2) ['age', '29']
   3. **2**: (2) ['DOB', '09-11-1994']

let randowKey = {100:"hundred",4:"four",2:"two",10:"ten"} Object.entries(randowKey)

1. *(4) [Array(2), Array(2), Array(2), Array(2)]*
   1. **0**: (2) ['2', 'two']
   2. **1**: (2) ['4', 'four']
   3. **2**: (2) ['10', 'ten']
   4. **3**: (2) ['100', 'hundred']

#### Object.freeze() we cant add or remove,update from the object when object is freezed. let freezObj = {

#### Name:"saikumar",

#### Age:29

#### }

#### let fObj = Object.freeze(freezObj);

#### undefined

#### fObj.Name='vijay';

#### 'vijay'

#### fObj

#### {Name: 'saikumar', Age: 29} Object.freeze(new Uint8Array(0)); // No elements

#### // Uint8Array []

#### Object.freeze(new Uint8Array(1)); // Has elements

#### // TypeError: Cannot freeze array buffer views with elements

#### Object.freeze(new DataView(new ArrayBuffer(32))); // No elements

#### // DataView {}

#### Object.freeze(new Float64Array(new ArrayBuffer(64), 63, 0)); // No elements

#### // Float64Array []

#### Object.freeze(new Float64Array(new ArrayBuffer(64), 32, 2)); // Has elements

#### // TypeError: Cannot freeze array buffer views with elements const obj = {

#### prop: 42,

#### };

#### Object.freeze(obj);

#### obj.prop = 33;

#### // Throws an error in strict mode

#### console.log(obj.prop);

#### // Expected output: 42

# Object.fromEntries() const entries = new Map([

# ['foo', 'bar'],

# ['baz', 42],

# ]);

# const obj = Object.fromEntries(entries);

# console.log(obj);

# // Expected output: Object { foo: "bar", baz: 42 }

# let mapObj = new Map([

# 

# ['name','sai'],

# ['age',29] const arr = [

# ["0", "a"],

# ["1", "b"],

# ["2", "c"],

# ];

# const obj = Object.fromEntries(arr);

# console.log(obj); // { 0: "a", 1: "b", 2: "c" } let arrObj = [

# ['Name','saikumar'],

# ['Age',25]

# ]

# Object.fromEntries(arrObj);

# {Name: 'saikumar', Age: 25}

# ]);

# undefined

# Object.fromEntries(mapObj);

# {name: 'sai', age: 29}

### [Using Object.groupBy()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/groupBy" \l "using_object.groupby)

First we define an array containing objects representing an inventory of different foodstuffs. Each food has a type and a quantity.

const inventory = [

{ name: "asparagus", type: "vegetables", quantity: 5 },

{ name: "bananas", type: "fruit", quantity: 0 },

{ name: "goat", type: "meat", quantity: 23 },

{ name: "cherries", type: "fruit", quantity: 5 },

{ name: "fish", type: "meat", quantity: 22 },

];

Object.groupBy(inventory,({type}) => type );  
result: const result = Object.groupBy(inventory, ({ type }) => type);

/\* Result is:

{

vegetables: [

{ name: 'asparagus', type: 'vegetables', quantity: 5 },

],

fruit: [

{ name: "bananas", type: "fruit", quantity: 0 },

{ name: "cherries", type: "fruit", quantity: 5 }

],

meat: [

{ name: "goat", type: "meat", quantity: 23 },

{ name: "fish", type: "meat", quantity: 22 }

]

}

\*/  
  
const inventory = [ { name: "asparagus", type: "vegetables", quantity: 5 }, { name: "bananas", type: "fruit", quantity: 0 }, { name: "goat", type: "meat", quantity: 23 }, { name: "cherries", type: "fruit", quantity: 5 }, { name: "fish", type: "meat", quantity: 22 }, ];

Object.groupBy(inventory,({quantity}) => quantity > 5 ? "ok" :"restock" );

1. *{restock: Array(3), ok: Array(2)}*
   1. **ok**: Array(2)
      1. **0**: {name: 'goat', type: 'meat', quantity: 23}
      2. **1**: {name: 'fish', type: 'meat', quantity: 22}
      3. **length**: 2
      4. [[Prototype]]: Array(0)
   2. **restock**: Array(3)
      1. **0**: {name: 'asparagus', type: 'vegetables', quantity: 5}
      2. **1**: {name: 'bananas', type: 'fruit', quantity: 0}
      3. **2**: {name: 'cherries', type: 'fruit', quantity: 5}
      4. **length**: 3

const inventory = [

{ name: "asparagus", type: "vegetables", quantity: 5 },

{ name: "bananas", type: "fruit", quantity: 0 },

{ name: "goat", type: "meat", quantity: 23 },

{ name: "cherries", type: "fruit", quantity: 5 },

{ name: "fish", type: "meat", quantity: 22 },

];

function callBack({quantity}) {

return quantity > 5 ? 'ok': 'restock';

}

1. Object.groupBy(inventory,callBack);  
    *{restock: Array(3), ok: Array(2)}*
   1. **ok**: Array(2)
      1. **0**: {name: 'goat', type: 'meat', quantity: 23}
      2. **1**: {name: 'fish', type: 'meat', quantity: 22}
      3. **length**: 2
      4. [[Prototype]]: Array(0)
   2. **restock**: Array(3)
      1. **0**: {name: 'asparagus', type: 'vegetables', quantity: 5}
      2. **1**: {name: 'bananas', type: 'fruit', quantity: 0}
      3. **2**: {name: 'cherries', type: 'fruit', quantity: 5}
      4. **length**: 3

#### Object.isFrozen() Its checking wheather object is freeze or not. let obj = {a:1,b:2}

#### Console.log(Object.isFrozen(obj)) // false Object.freeze(obj) console.log(Object.isFrozen(obj)) // true

#### Object.isSealed()

#### const object1 = {

#### property1: 42,

#### };

#### console.log(Object.isSealed(object1));

#### // Expected output: false

#### Object.seal(object1);

#### console.log(Object.isSealed(object1));

#### // Expected output: true

#### let obj = {name:'saikumar',age:29}

#### Object.isSealed(obj)

#### false

#### Object.seal(obj);

#### {name: 'saikumar', age: 29}

#### Object.isSealed(obj)

#### True Object.keys()

let obj = {name:'saikumar',age:29}

Object.keys(obj);

(2) ['name', 'age']  
  
  
const object1 = {

property1: 42,

};

#### Object.seal()

Object.seal(object1);

object1.property1 = 33;

console.log(object1.property1);

// Expected output: 33

delete object1.property1; // Cannot delete when sealed

console.log(object1.property1);

// Expected output: 33

let obj = {

name:'saikumar',age:25,designation:'React js'

}

Object.seal(obj)

{name: 'saikumar', age: 25, designation: 'React js'}age: 25designation: "React js"name: "saikumar"[[Prototype]]: Object

obj.age=29

29

obj

{name: 'saikumar', age: 29, designation: 'React js'}

delete obj.name;

False // can not delete when object is seal

obj

{name: 'saikumar', age: 29, designation: 'React js'}

#### let obj = {

#### name:'saikumar',age:25,designation:'React js'

#### } Object.values(obj)

#### ['saikumar', 29, 'React js']

Javascript methods :

a = 100;

console.log(a)

// 100  
  
onsole.log([] == false) // 0 == 0 not checking datatype

true

console.log([] === false) checking datatype ,value 0 === 0

false  
  
let a = [1,2,3,4];

let str ='1,2,3,4';

console.log(a == str) // here values or equal

// true  
  
console.log(a === str)   
// false  
  
checking block level var,let, const  
let v = 10;

var b = 20;

const c = 30;

function CheckBlockLevel() {

var b = 40;

let v = 20;

const c = 60;

console.log('var b =', b);

console.log('const c =',c);

console.log('v =',v);

}   
output:

CheckBlockLevel();

var b = 40

const c = 60

# v = 20 Function.prototype.apply():

let a= [1,2,4,5,6,7];  
Math.max.apply(null,a);  
// output: 7

let a= [1,2,4,5,6,7];  
Math.min.apply(null,a);  
// output: 1  
  
  
let numbers = [1,3,4,5,8,10];

Math.min.apply(null,numbers);

1

Math.max.apply(null,numbers);

10

let el = [2,6,9];

numbers.push.apply(numbers,el);

9

console.info(numbers)

(9) [1, 3, 4, 5, 8, 10, 2, 6, 9]  
  
const array = ["a", "b"];

const elements = [0, 1, 2];

array.push.apply(array, elements);

console.info(array); // ["a", "b", 0, 1, 2]  
  
const array = ["a", "b"];

const elements = [0, 1, 2];

array.push(…elements);

console.info(array); // ["a", "b", 0, 1, 2]  
  
  
let arr = [1,5,8,7];

let el = ['s','a','i'];

arr.push(...el);

7

console.info(arr)

 [1, 5, 8, 7, 's', 'a', 'i']

#### Function.bind() const module = {

#### x: 42,

#### getX: function () {

#### return this.x;

#### },

#### };

#### const unboundGetX = module.getX;

#### console.log(unboundGetX()); // The function gets invoked at the global scope

#### // Expected output: undefined

#### const boundGetX = unboundGetX.bind(module);

#### console.log(boundGetX());

#### // Expected output: 42 let val = {

#### name:'sai',

#### age:29,

#### GetAge: function () {

#### return this.age;

#### }

#### }

#### const getOBJ = val.GetAge;

#### console.log(getOBJ()); // undefined const d = getOBJ.bind(val); console.log(d()) // 29

#### 

#### Function.call() function Product(name, price) {

#### this.name = name;

#### this.price = price;

#### }

#### function Food(name, price) {

#### Product.call(this, name, price);

#### this.category = 'food';

#### }

#### console.log(new Food('cheese', 5).name);

#### // Expected output: "cheese" function Product(name,price) {

#### this.name= name;

#### this.price = price;

#### }

#### function Food(name,price){

#### Product.call(this,name,price); // using function.call(); this.category= 'Food sections'

#### }

#### console.log(new Food('cheese',19));

#### // Output: Food {name: 'cheese', price: 19, category: 'Food sections'}

function greet() {

console.log(this.animal, "typically sleep between", this.sleepDuration);

}

const obj = {

animal: "cats",

sleepDuration: "12 and 16 hours",

};

greet.call(obj); // cats typically sleep between 12 and 16 hours

function GetCat(){

console.log(this.animal,'sleep in ',this.description);

}

let obj = {

animal:'cats',

description:'between 12 and 16 hours daily'

}

GetCat.call(obj);

VM369:2 cats sleep in between 12 and 16 hours daily