


# JAVASCRIPT

## OBJECTS IN DEPTH



Object	Properties	Methods
	<p>car.name = Fiat</p> <p>car.model = 500</p> <p>car.weight = 850kg</p> <p>car.color = white</p>	<p>car.start()</p> <p>car.drive()</p> <p>car.brake()</p> <p>car.stop()</p>

```
const car = {type:"Fiat", model:"500", color:"white"};
```

# Object Prototype

```
const obj = {}  
  
console.log(obj.toString());
```

```
"[object Object]"
```

```
▼ {} ⓘ  
  ▾ [[Prototype]]: Object  
    ▶ constructor: f Object()  
    ▶ hasOwnProperty: f hasOwnProperty()  
    ▶ isPrototypeOf: f isPrototypeOf()  
    ▶ propertyIsEnumerable: f propertyIsEnumerable()  
    ▶ toLocaleString: f toLocaleString()  
    ▶ toString: f toString()  
    ▶ valueOf: f valueOf()  
    ▶ __defineGetter__: f __defineGetter__()  
    ▶ __defineSetter__: f __defineSetter__()  
    ▶ __lookupGetter__: f __lookupGetter__()  
    ▶ __lookupSetter__: f __lookupSetter__()  
    ▶ __proto__: (...)  
    ▶ get __proto__: f __proto__()  
    ▶ set __proto__: f __proto__()
```

Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.

# Object.create()

The **Object.create()** method creates a new object, using an existing object as the prototype of the newly created object.

```
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"
```

# Object.defineProperty()

The **Object.defineProperty()** method defines new or modifies existing properties directly on an object, returning the object.

```
1 const object1 = {};  
2  
3 Object.defineProperty(object1, {  
4   property1: {  
5     value: 42,  
6     writable: true  
7   },  
8   property2: {}  
9 });  
0  
1 console.log(object1.property1);  
2 // expected output: 42  
3
```

# Object.defineProperty()

The static method **Object.defineProperty()** defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

```
1 const object1 = {};  
2  
3 Object.defineProperty(object1, 'property1', {  
4   value: 42,  
5   writable: false  
6 });  
7  
8 object1.property1 = 77;  
9 // throws an error in strict mode  
10  
11 console.log(object1.property1);  
12 // expected output: 42  
13
```

# Object.defineProperty()

The **Object.getOwnPropertyDescriptor()** method returns an object describing the configuration of a specific property on a given object (that is, one directly present on an object and not in the object's prototype chain). The object returned is mutable but mutating it has no effect on the original property's configuration.

```
1 const object1 = {  
2   property1: 42  
3 };  
4  
5 const descriptor1 = Object.getOwnPropertyDescriptor(object1, 'property1');  
6  
7 console.log(descriptor1.configurable);  
8 // expected output: true  
9  
10 console.log(descriptor1.value);  
11 // expected output: 42  
12
```

# Object.getOwnPropertyNames()

The **Object.getOwnPropertyNames()** method returns an array of all properties (including non-enumerable properties except for those which use Symbol) found directly in a given object.

```
1 const object1 = {  
2   a: 1,  
3   b: 2,  
4   c: 3  
5 };  
6  
7 console.log(Object.getOwnPropertyNames(object1));  
8 // expected output: Array ["a", "b", "c"]  
9
```



# Object.getPrototypeOf()

The **Object.getPrototypeOf()** method returns the prototype (i.e. the value of the internal `[[Prototype]]` property) of the specified object.

```
1 const prototype1 = {};  
2 const object1 = Object.create(prototype1);  
3  
4 console.log(Object.getPrototypeOf(object1) === prototype1);  
5 // expected output: true  
6
```

# Object.hasOwn()

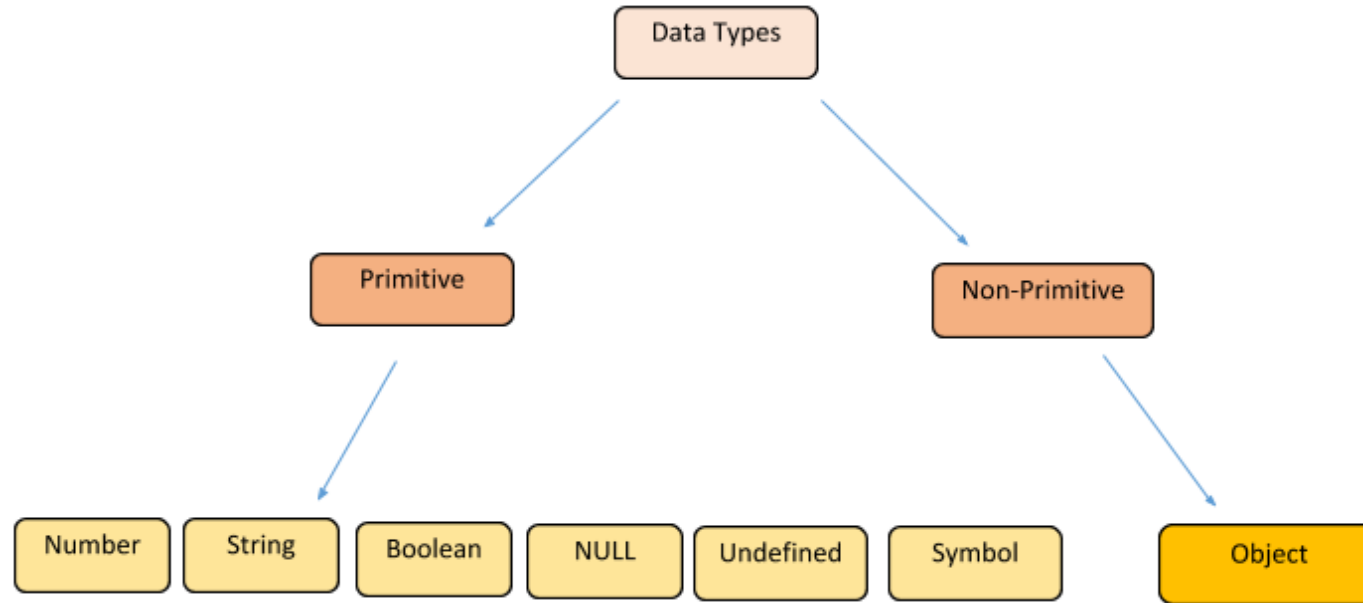
The **Object.hasOwn()** static method returns true if the specified object has the indicated property as its *own* property. If the property is inherited, or does not exist, the method returns false.

```
1 const object1 = {  
2   prop: 'exists'  
3 };  
4  
5 console.log(Object.hasOwn(object1, 'prop'));  
6 // expected output: true  
7  
8 console.log(Object.hasOwn(object1, 'toString'));  
9 // expected output: false  
10  
11 console.log(Object.hasOwn(object1, 'undeclaredPropertyValue'));  
12 // expected output: false  
13
```

# Tasks

1. Create a user object which prototype is from person object.
2. Set name property of user object not changeable.

# Data Types



# Map

Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

Methods and properties are:

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `map.get(key)` – returns the value by the key, `undefined` if `key` doesn't exist in map.
- `map.has(key)` – returns `true` if the `key` exists, `false` otherwise.
- `map.delete(key)` – removes the element (the key/value pair) by the key.
- `map.clear()` – removes everything from the map.
- `map.size` – returns the current element count.

```
1 let map = new Map();
2
3 map.set('1', 'str1'); // a string key
4 map.set(1, 'num1');   // a numeric key
5 map.set(true, 'bool1'); // a boolean key
6
7 // remember the regular Object? it would convert keys to string
8 // Map keeps the type, so these two are different:
9 alert( map.get(1) ); // 'num1'
10 alert( map.get('1') ); // 'str1'
11
12 alert( map.size ); // 3
```

# Weak Map

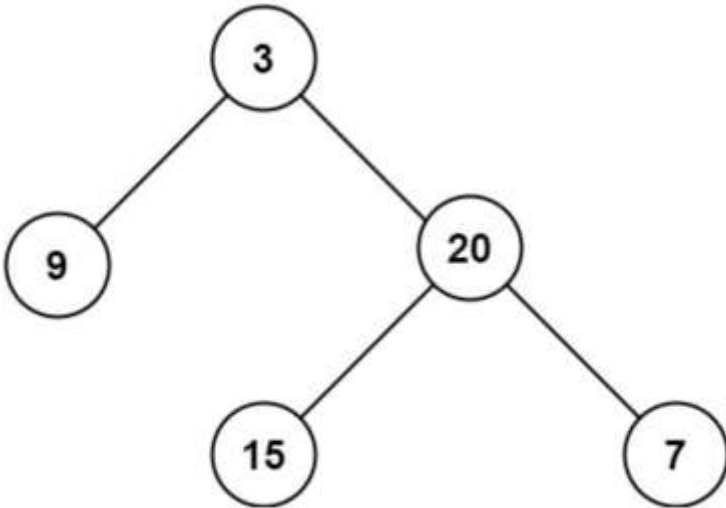
A **WeakMap** is a collection of key/value pairs whose keys must be objects, with values of any arbitrary [JavaScript type](#), and which does not create strong references to its keys. That is, an object's presence as a key in a **WeakMap** does not prevent the object from being garbage collected. Once an object used as a key has been collected, its corresponding values in any **WeakMap** become candidates for garbage collection as well — as long as they aren't strongly referred to elsewhere.

**WeakMap** allows associating data to objects in a way that doesn't prevent the key objects from being collected, even if the values reference the keys. However, a **WeakMap** doesn't allow observing the liveness of its keys, which is why it doesn't allow enumeration; if a **WeakMap** exposed any method to obtain a list of its keys, the list would depend on the state of garbage collection, introducing non-determinism. If you want to have a list of keys, you should use a [Map](#) rather than a **WeakMap**.

# Tasks

1. Write a JavaScript program to delete the **name** property from the following object.
2. Write a JavaScript program to get the length of a JavaScript object.
3. Create Dictionary class using Map()
4. **Display all the keys and values of a nested object.**
5. **Create an object which has a property 'userid' which can only be set once and will be a read only property**
6. A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

## 6.1 Construct the tree



Input: root = [3,9,20,null,null,15,7]  
Output: 3