JavaScript – Object and its internal representation

## Objects:

Object is one of the datatypes in JavaScript (i.e. Non-Primitive/Reference datatype).

Generally, Objects are obvious of what they are in real world (e.g. – car, computer, camera,..) that have properties - described by Noun and Adjectives and actions possible described by verbs.

Technically, Objects are unordered collection of related data (either Primitive/reference type) in “Key: Value” pair.

Properties: Key: Value Pair

Key – Primitive datatype

Value – Any datatype (Number, Boolean, Objects, array of objects,..)

Methods – Actions possible/needed to be performed on objects

Example:

var person = {

firstName: "Sheldon",

lastName: "Cooper",

address: {

street: "24th Houston Street",

city: "Pasadena",

state:"California",

country: "US"

},

interests: ["String theory", "Comic Books", "Video games", "Sci-fi TV shows"],

contacts: [

{name: "Leonard", relation: "friend", phone: 8877887788, mailID: "leonard@bbt.com"},

{name: "Amy", relation: "Wife", phone: 4547328278, mailID: "amy@bbt.com"},

{name: "Raj", relation: "friend", phone: 6545654324, mailID: "raj@bbt.com"},

{name:"Missy", relation: "sister", phone: 6545678676, mailID: "missy@bbt.com"}

],

knockDoor : function() {return "Knock! Knock! Knock!" ;} ,

getFullName(){ return `${this.firstName} ${this.lastName}` }

}

All JavaScript values, except primitives, are objects.

**Mostly used:** Arrays are also a special type of object (i.e. List objects)

[(key/index:)value]. In JavaScript, whenever we modify an array, key names (i.e. index) are automatically updated.

Example:

var shoppingList = [“apple”,”orange”,”chips”,”choclates”,”mazza”];

## Internal Representation:

**Creating a JavaScript Object:**

Objects in JavaScript can be created in following ways:

* Using Object Literals // let person = {firstName:”John”, lastName:”Doe”}
* Using new keyword // const person = new Object();
* Using constructor
* Using Object.create()

**Properties:**

The Properties are defined as the variable attached to the object - defining the characteristics of an object. They can usually be changed, added, and deleted.

**Methods:**

Besides data, objects can have actions. The actions of objects are known as methods

**Adding/Modifying/Deleting Properties/Methods:**

We can add new properties/methods to an object once after its creation.

objectName.newPropertyName = value

objectName.newMethodName = function declaration

Example:

const dog ={};

console.log(dog); // {}

dog.name = "Scooby";

dog.bark = function(){ return "Woof Woof Woof"; }

console.log(dog); // { name: 'Scooby', bark: [Function] }

We can modify the properties using assignment operator(=).

dog.name **= “Lily”**;

console.log(dog); // { name: 'Lily' }

We can delete a property using **delete** operator. It deletes both value and property itself and returns Boolean. It has effect only on object properties, not in variables.

**delete** objectName.property

**delete** objectName.method

Example:

delete dog.name;

console.log(dog.name);// undefined

console.log(dog); // { bark: [Function] }

delete dog.bark;

console.log(dog) // {}

**Accessing Properties:**

Objects have properties that can be hierarchically organized. So they should be accessed accordingly to retrieve the exact value.

Properties can be accessed by the following ways:

* Using Member access (**.**)

objectName.property      // person.firstName

* Using Computed Member access (**[]**) – Sometimes if keys are numbers or strings with blank in it, it is impossible to access that property with ‘.’ , So we use this

objectName["string Property"]  // person[“first name”] => Bad Practice to use space between strings in key

objectName[KeyNumber]// person.contacts[0] or person[“contacts”][0]

objectName[expression]   // x = "cont"+"acts"; car[x]

**Accessing Methods:**

Methods of the object can be executed by invoking it with “()” at the end. If not, it will return the function definition

objectName.methodName()

console.log(person.knockDoor); //[Function: knockDoor]

console.log(person.knockDoor()) //Knock! Knock! Knock!

**Checking property exists in Object:**

We can use **in** operator, to check if a property exists in an object.

propertyName **in** objectName

The **in** operator returns true if the propertyName exists in the objectName.

console.log("firstName" in person); // true

console.log("profession" in person); //false

**The this value**:

We can use the **this** inside the method to access the object’s properties. It references the object that invoked the method.

console.log(person.getFullName()) // Sheldon Cooper

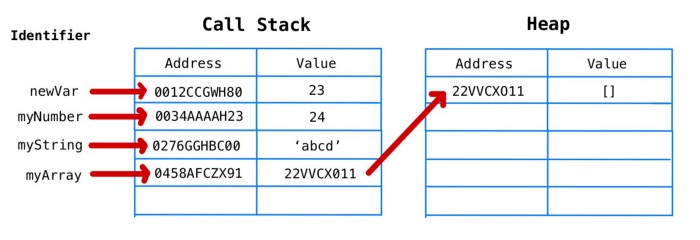
**Memory Model, Variable Assignment and declaration:**

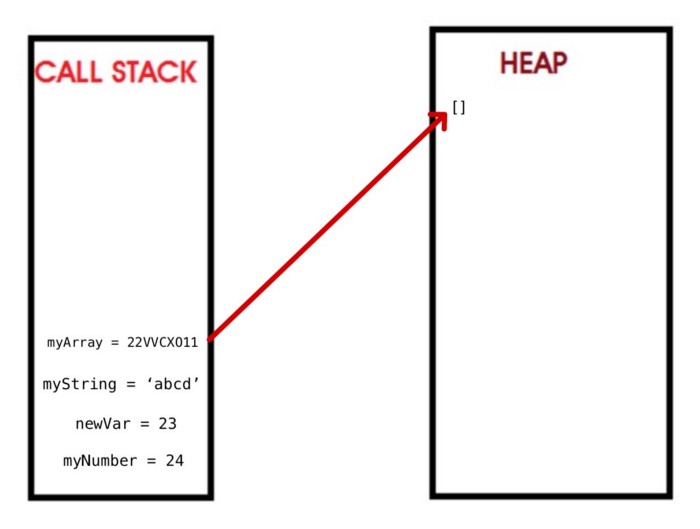
Non-primitive JS data types behave differently compared to primitive JS data types.

Primitive datatypes: Stored in Call Stack. Creates a unique identifier (name) for the variable; allocates an address in memory at runtime and stores the value in the address.

let newVar = 23;

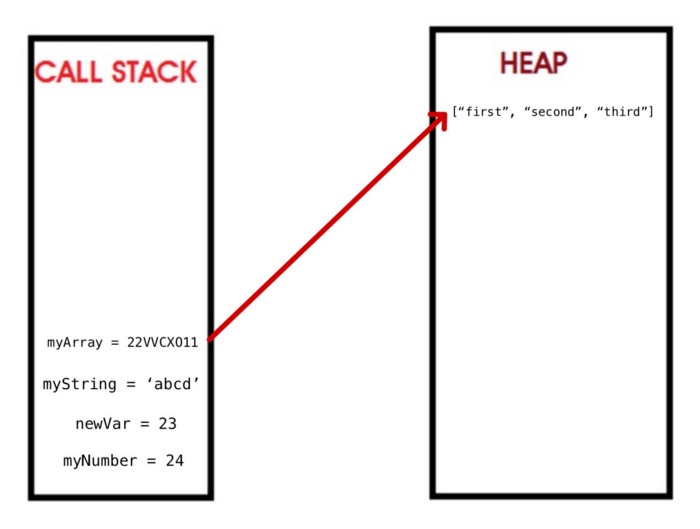
Non-Primitive/Reference datatype: Stored in Heap. Creates a unique identifier (name) for the variable; allocates an address in memory at runtime and stores the address of the heap which holds the value.

let myArray = [];



The key difference is that the heap can store unordered data that can grow dynamically—perfect for arrays and objects

myArray.push("first")  
myArray.push("second")  
myArray.push("third")  
myArray.push("fourth")  
myArray.pop()



## NOTES:

Unassigned properties of an object are [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) (and not [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)).

Primitive datatypes are immutable whereas objects are mutable.