JavaScript Objects in Depth, Part 1: The Fundamentals

dzone.com/articles/javascript-object-in-depth-part-1fundamental

JavaScript has two main data types: primitives and objects. The interesting thing here is that other than the primitive type everything else is an object, including functions. Therefore, if we are really interested in learning JavaScript, then understanding objects should be on the top in the list.

An object can be created with figure brackets (`{...}`) with an optional list of properties where a property is a key:value pair, where the key is a string and the value can be anything. By anything, we mean it can be a primitive or an object or even a function.

```
// Object in JavaScript

let employee = {
   name : "Employee 1",
   experience: 3,
   skills: ['Java', 'JavaScript', 'Data Structure & Algorithm'],
   address: {
      landmark: 'xyz',
      city: 'Hyderabad',
      pin: 500032
   },
   getExceptedHike : function(){
      // do some logic
      return "Contact to Manager";
   }
};
```

In the `employee` object, we can observe and conclude that the properties of an object are in the "key:value" form, separated with a comma (`,`) and the value can be anything.

Furthermore, we can create an empty object using one of two syntaxes:

```
// Creating an Object
let ob1 = new Object();   // "constuctor object syntax"
let ob2 = {};             //"literal object syntax"

// Go to console and verify it's type

typeof ob1;  // return: "object"
typeof 0b2;  // return: "object"
```

In the above code, we can verify the type of `ob1` and `ob2` using the `typeof` keyword, which returns the variable's type(s). The whole point here is that we can create an empty object using the above two syntaxes.

Thought: Are these two empty objects the same?

Comparison Operators in JavaScript

Before we compare these two objects, first we must understand how comparison operators work. Unlike other programming languages, JavaScript has both strict and type-converting comparisons.

1. Equality ('=='): The equality operator converts the operands if they are not of the same type, then applies a strict comparison. If both operands are objects, then JavaScript compares the internal references which are equal when operands refer to the same object in memory.

2. Strict Equality ('==='): This is also known as an identity operator. It returns true if the operands are strictly **equal** with **no type conversion.**

```
// Equality operator
console.log(1.00 == 1);        // true
console.log("1" == 1);// true
console.log(1 == 1);// true
console.log("a" == 1);         // false

// Strict Equality operator
console.log(1.00 === 1);       // true
console.log("1" === 1);       // false
console.log(1 === 1);          // true
console.log("A" === 1);        // false
```

Line 8 might be confusing for those who are coming from a different programming language such as Java, C++, etc. You might think the `typeof (1.00)` is float/double, so how does the strict equality operator return true? The fact is that JavaScript has only

number types, therefore 1 and 1.00 are both of type "number."

Now let's compare the two empty object and check the results.

```
let ob1 = new Object();
let ob2 = {};

if (ob1 === ob2) {
  console.log("Object are strictly equal");
} else if (ob1 == ob2) {
  console.log("Object are equal");
} else {
  console.log("Objects are not equal");
}

console:
Objects are not equal
```

Conclusion: Though both are the empty object, operands refer to a different object in memory. Hence they are not equal. TL;DR: No two empty objects are equal.

Accessing Object Properties in JavaScript

A property has a key (also known as a "name" or "identifier") before the colon (`:`) and the value to the right of it. We can access or set the property value using dot notation or Square Brackets. **Note:** We can use multiword property names (for example, ”year of birth”), but then they must be quoted and to access this property it must use square brackets (`[ ]`) notation. Check the below example for more clarification:

```
let emp = {
  name: "Nitesh Nandan",
  email: "abc@mail.com",
  "year of Birth": 1996
};

// Aceessing the property
console.log(emp.name);  // Nitesh Nandan
console.log(emp."year of birth"); // Error: Uncaught SyntaxError: Unexpected string
console.log(emp.year of birth);   // Error: Uncaught SyntaxError: Unexpected string

console.log(emp["year of birth"]); // 1996

// Setting the property value

emp.name = "John Martin";
emp["year of birth"] = 1980;
emp.city = "New York";   // Adding a new property city
console.log(emp.name);   // John Martin
console.log(emp.city);   // New York
console.log(emp.id);   // undefined
```

So this is how we access/set the property of an Object. In line 21, I have tried to access the property ("id") which is not a part of the object. If we try to access the property which is not present in the object it returns its value as undefined**.**

Now the question is, can we conclude if the value of some property is undefined? And, if so, then can we determine if the property itself is not defined? Let's check this example before we conclude:

```
let obj = {
   test: undefined
}

console.log(obj.test); // it's undefined, so - no such property??
```

Though line 5 returns undefined, technically speaking the property does exist. So the question remains the same: how can we check whether the property exists or not?

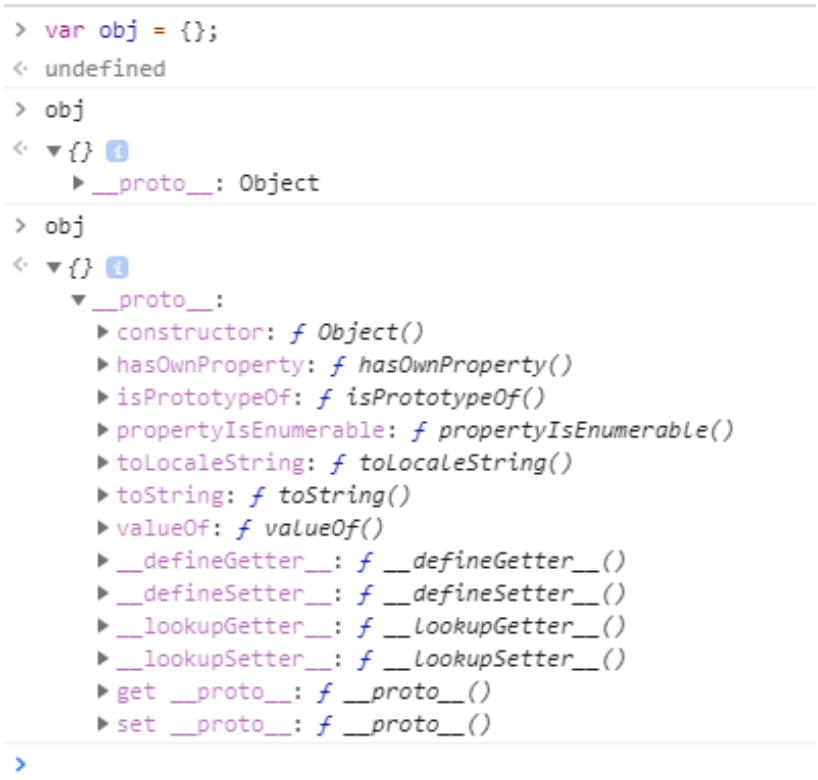
These are the ways to know whether the property exists or not:

```
console.log(obj.hasOwnProperty("test")); // true;
console.log("test" in obj);              // true;

console.log(obj.hasOwnProperty("name")); // false;
console.log("name" in obj);              // false;
```

So we can usethe `in` operator or the `hasOwnProperty` method to check for the existence of the property.

A situation like this very rarely happens because undefined values are usually not assigned. We mostly use null for "unknown" or "empty" values.

You might be wondering from where this `hasOwnProperty` came from. If you are familiar with the Java language then you might know that each Java class implicitly inherits a Java object class. Similarly, each object in JavaScript inherits (prototype) an object and that has some common methods; `hasOwnProperty` is defined there.



You can observe every object has a `__proto__` which has an object as its value and if we expand that object we can see the methods. The detailed explanation is out of the scope of this article.

We can use a for loop and the `in` operator to walk over all the keys of an object. The syntax is:

```
let emp = {
  name: "abc",
  email: "abc@mail.com",
  "year of Birth": 1996
};

for (key in emp) {
  console.log(key + ": " + emp[key]);
}

Console(output):
  name: abc
  email: abc@mail.com
  year of Birth: 1996
```

Getters and Setters (JavaScript Accessors)

In JavaScript objects, there are two kinds of properties:

1. Data properties: All the properties that we have been using up to now are data properties.
2. Accessor properties: These are functions that work on getting and setting a value but look like regular properties.

Let's look at an example for more clarification:

```
let user = {
  _userName: "user123",
  _age: 18,

  get userName() {
    console.log("userName getter method is invoked");
    return this._userName;
  },
  set userName(name) {
    console.log("userName setter method is invoked");
    this._userName = name;
  },
  get age() {
    console.log("age getter method is invoked");
    return this._age;
  },
  set age(age) {
    console.log("age setter method is invoked");
    this._age = age;
  }
};

console.log(user.userName);
// userName getter method is invoked
// user123

console.log(user.age);
// age getter method is invoked
// 18

user.userName = "abcUser";
// userName setter method is invoked

user.age = 22;
//age setter method is invoked


console.log(user.userName);
// userName getter method is invoked
// abcUser

console.log(user.age);
// age getter method is invoked
// 18
```

So what we find here is that the getter/setter method is implicitly invoked whenever we are accessing or setting its value. In JavaScript, there is no concept of private attributes, therefore, in the community, we followed a convention that the attribute name which starts with " `_` " is a private attribute and one should not access it directly.

Do Yourself: Take the above code and remove the first character from attribute name, i.e " `_` ", from all the attributes and execute. You will find something interesting; let me know in the comments.

We can add validation in the property with the help of accessor properties.

```
set userName(name) {
   if (name.length < 5) {
     console.log("Error: userName can't be less than five character");
     return;
   }
   this._userName = name;
 }

user.userName = "abc";
// Error: userName can't be less than five character
```

Object Cloning

Unlike the primitive data type, we can't copy an object. Copying an object creates one more reference to the same object. Let's check the example.

```
let ob1 = {
   name : "abc"
}
let ob2 = ob1;
console.log(ob2===ob1 && "Ob1 and Ob2 point to same object in memory.");
// Ob1 and Ob2 point to same object in memory.
```

So, by strictly comparing both objects, we have verified that both the references point to the same object in memory. In JavaScript, there is no built-in method for making a duplicate (exact clone) object. But if we really want that, then we need to create a new object and replicate the structure of the existing object by iterating over its properties and copying them on the primitive level one by one. The good thing is that we've already gone over how to iterate over object properties. You guessed it! We will be using the `in` operator.

```
let user = {
   name: "user1",
   age: "21"
}

let clone = {} // the new empty Object

// let's copy all user properties into it
for(let key in user){
   clone[key] = user[key];
}

console.log(user.name);  // user1
console.log(clone.name); // user1

console.log(user.age);   // 21
console.log(clone.age);  // 21

console.log(user!==clone && "Object Reference: user, clone points different object in memory." );
// Object Reference: user, clone points different object in memory.
```

We can also use the `Object.assign` method for the same purpose. It takes two arguments, `dest` and **src1, src2, . . . , srcN** (can be as many as needed) are objects, and returns an object.

`Object.assign(dest, src1, src2, . . ., srcN).`

It copies the properties of all object ***src1, . . ., srcN*** into `dest`.

```
let user = {
   name: "user1",
   age: "21"
}

let clone = Object.assign({}, user);

console.log(user.name);  // user1
console.log(clone.name); // user1

console.log(user.age);   // 21
console.log(clone.age);  // 21

console.log(user!==clone && "Object Reference: user, clone points different object in memory.");
// Object Reference: user, clone points different object in memory.
```

There are more ways to clone objects, which I will discuss in the next part of this tutorial.

This article covers the basic properties of JavaScript objects. In Part 2, I will go deeper into this concept. Let me know in the comment if you have any questions.

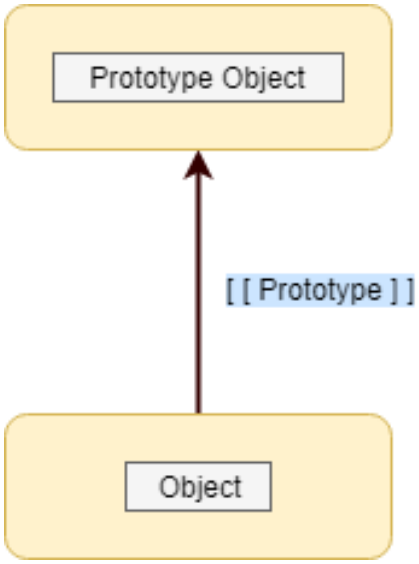
JavaScript Objects in Depth, Part 2: Inheritance

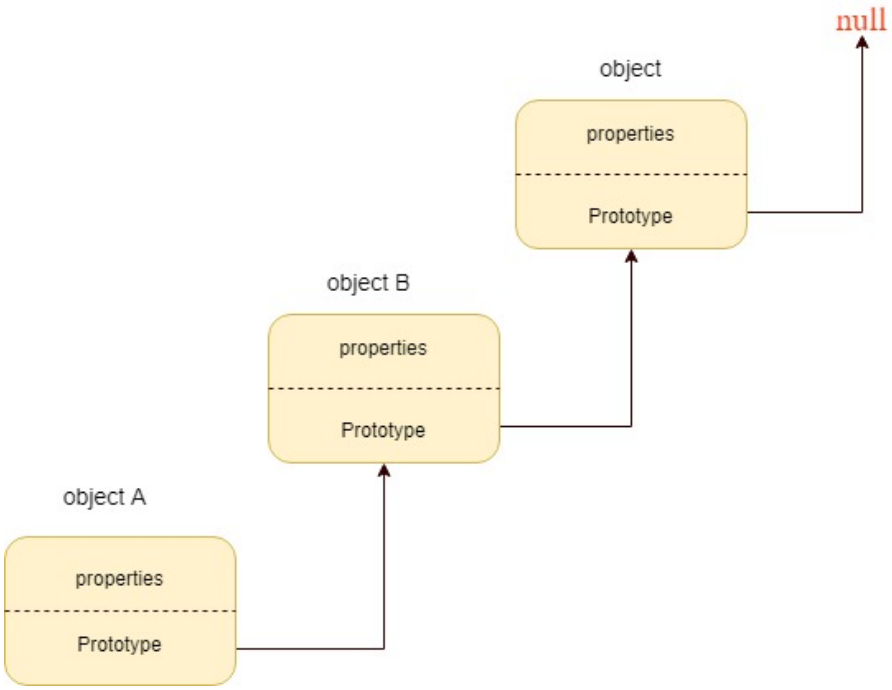
dzone.com/articles/javascript-objects-in-depth-part-2-inheritance

Inheritance is a key feature of Object-Oriented Programming. Like any other Object-Oriented Programming language, JavaScript also supports inheritance. JavaScript objects have a special hidden property `[[ Prototype ]]` that is either `null` or references another **object**, called "***a prototype***."

JavaScript Object Reference

Let's understand how this works. When we want to read a property from an object, and it's missing, JavaScript automatically tries to fetch it from its prototype object. In programming, this is called prototype inheritance. In prototype inheritance, the topmost object in the hierarchy chain has its prototype as `null`. Check the below diagram more clarity.





Prototype Chain

We have talked enough about the property `Prototype` but now the question is how can we access this special property? We have a getter/setter for Prototype, i.e. `__proto__` . Please note that `__proto__` is not the same as a prototype, it is just a getter/setter for it. We also have dedicated a getter/setter, `Object.getPrototypeOf/Object.setPrototypeOf` , that also gets/sets the prototype. Let's check the below code for more clarification.

```
let user = {
   name: "user1"
}

console.log(user.__proto__ === Object.getPrototypeOf(user) && "Both are same");
// Both are the same.
```

Inheritance

Let's write some code to inherit the property from a parent class, and try to understand how it's working. Please check the below example.

```
let employee = {
  works: true
}

let developer = {
   code : true
}

// Before Inheritance
console.log(developer.works);  //undefined
console.log(developer.code);   // true

developer.__proto__ = employee; // Inherting employee

console.log(developer.works);   // true
console.log(developer.code);    // true
```

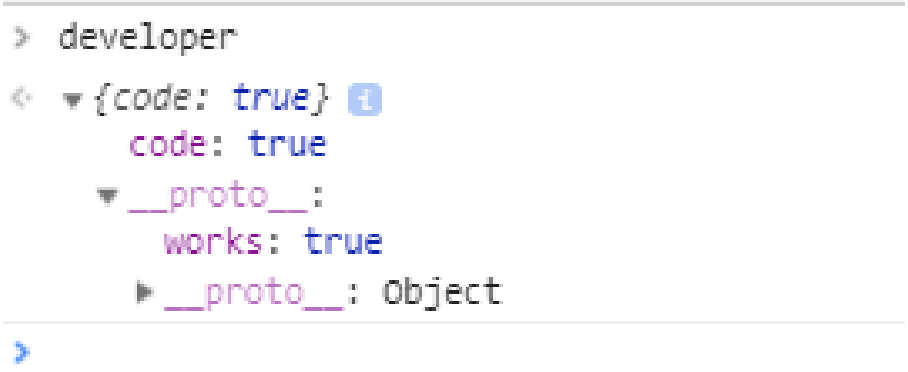
Initially, the `developer` object had only one attribute, i.e `code` ,but after setting its prototype to `employee` it inherits the `works` propertyfrom the **employee** object.

But, still, we have to confirm whether the `works` attributegets copied to the `developer` object or JavaScript internally refers to the `employee` object for the attribute. We will use `hasOwnPropery` for validation.

```
for(let key in developer){
   console.log("property name: " + key+",", "belongsToDeveloper: "+
developer.hasOwnProperty(key) );
}
```

```
Output (console):
  property name: code, belongsToDeveloper: true
  property name: works, belongsToDeveloper: false
```

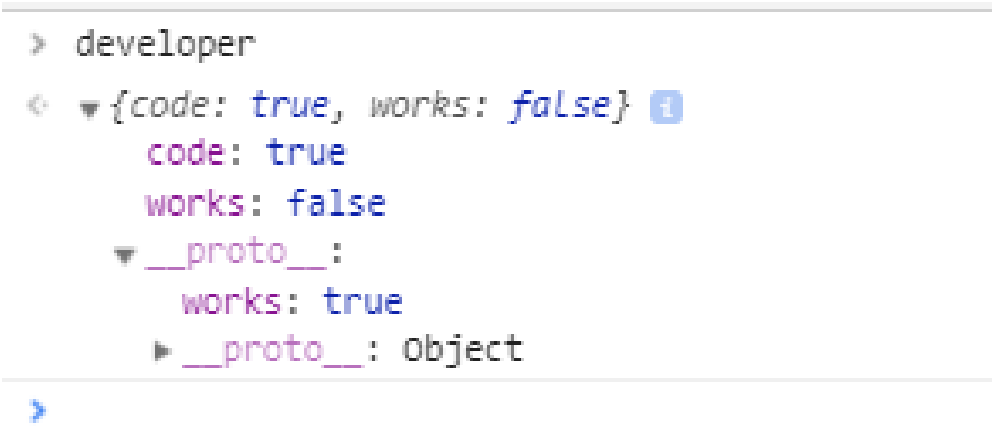
Also, we can cross-check from by looking into the**developer** object:



So as we were expecting, when we try to access the property works, it internally refers to the attribute of its `Prototype` . But still wondering what will happen if we try to write/delete the attribute **works** of the developer. Let's check.

```
let employee = {
  works: true
}

let developer = {
   code : true
}

developer.__proto__ = employee;
developer.works = false;

console.log(developer.works);   // false
console.log(developer.code);    // true

console.log(employee.works);    // true

for(let key in developer){
   console.log("atribute name: " + key+",", "belongsToDeveloper: "+
developer.hasOwnProperty(key) );
}

Output (Console):
 atribute name: code, belongsToDeveloper: true
 atribute name: works, belongsToDeveloper: true
```

Lines 12 and 15 confirms that now developer and employee both has its own property **works**. So we can conclude through this example is that the prototype is only used for **reading** purposes. Write/delete operation work directly with the Object. Let's verify the same in the console:



In the first part, we discussed how to clone an object using and operator and the `Object.assign` method. Do you think it will be fine to clone an objectt that has its prototype as another object the same way? Let me know in the comments!

Brain Teaser: Set the prototype of an object to some primitive type and let me know your thoughts in the comments.

Summary

Note: The references can't go in circles. JavaScript will throw an error if we try to assign `__proto__` in a circle. You may find it obvious, but still: there can be **only one** `[[Prototype]]` . An object may not inherit from two others.

This was a very brief introduction to inheritance in JavaScript. In Part 3, we will try to understand the keyword `this` and also look into the `constructor` function.