Objects

22 July 2020 11:42

```
let user = new Object(); // "object constructor" syntax

let user = {};  // "object literal" syntax
```

Objects created with function constructor lets you have multiple instances of that object. Change made to one instance will not affect the other.
Object created using object literals are singletons. When a change is made to the object, it affects that object across the entire script.

To remove a property, we can use `delete` operator:

```
delete user.age;

let user = {

  name: "John",

  age: 30,

  "likes birds": true  // multiword property name must be quoted
};
```

When an object which is declared with const, we can only change the value of its properties but not the object itself.

When we print an object, then the object keys are automatically sorted in alphabetical or numerical order.

Computed properties:

```
let fruit = "apple";
let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};
let bag = {
    [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Property value shorthand:

```
{
  name, // same as name: name
  age,  // same as age: age
  // ...
  };
```

Unlike variables, we can use language-reserved words like “for”, “let”, “return” etc. as property names in object.

Other types are automatically converted to strings.
For instance, a number `0` becomes a string `"0"` when used as a property key:

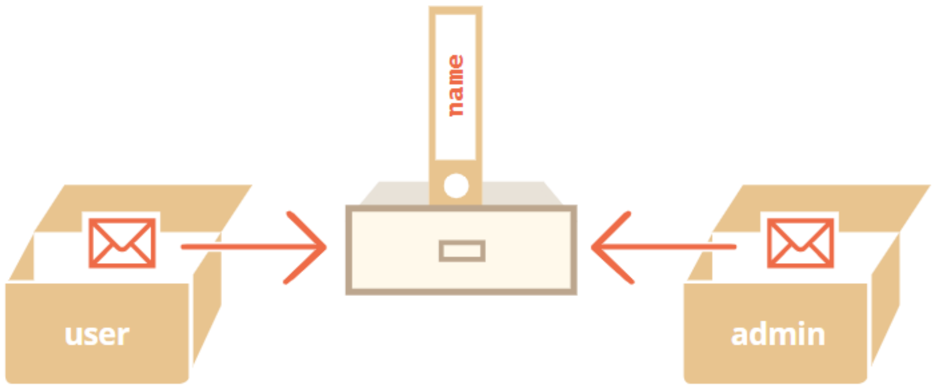
```
let obj = {
  0: "test" // same as "0": "test"
};
// both alerts access the same property (the number 0 is converted to string "0")
alert( obj["0"] ); // test
alert( obj[0] ); // test (same property)
```

Property existence test, “in” operator:
we can test the property existence in below 2 ways:

```
 1. user.noSuchProperty === undefined //there is an edge case when property exists with value undefined.
 2. "key" in object
```

Ordered like an object --> Better go through this link

Object copying, references:
A variable stores not the object itself, but its “address in memory”, in other words “a reference” to it.



Two objects are equal only if they are the same object.

```
let a = {};

let b = {};
// two independent objects

alert( a == b ); // false
```

Cloning and merging, Object.assign

```
Object.assign(dest, [src1, src2, src3...]) // Shallow copy
If the copied property name already exists, it gets overwritten:

let user = {
  name: "John",
  age: 30
};
let clone = Object.assign({}, user);
```

Nested cloning

```
let user = {
    name: "John",
    sizes: {
        height: 182,
        width: 50
    }
};
```

There's a standard algorithm for deep cloning and more complex cases, called the Structured cloning algorithm.

We can use recursion to implement it. Or, not to reinvent the wheel, take an existing implementation, for instance _.cloneDeep(obj) from the JavaScript library lodash.

https://flaviocopes.com/how-to-clone-javascript-object/
https://www.samanthaming.com/tidbits/70-3-ways-to-clone-objects/
Copying an object using spread operator is shallow.
JSON.stringify/parse only work with Number and String and Object literal without function or Symbol properties.

Garbage Collection(Reachability):
The basic garbage collection algorithm is called “mark-and-sweep”.
Internal algorithms

Some of the optimizations:

- **Generational collection** – objects are split into two sets: “new ones” and “old ones”. Many objects appear, do the job and die fast, they can be cleaned up aggressively. Those that survive for long enough, become “old” and are examined less often.
- **Incremental collection** – if there are many objects, and we try to walk and mark the whole object set at once, it take some time and introduce visible delays in the execution. So the engine tries to split the garbage collection pieces. Then the pieces are executed one by one, separately. That requires some extra bookkeeping between them tc track changes, but we have many tiny delays instead of a big one.
- **Idle-time collection** – the garbage collector tries to run only while the CPU is idle, to reduce the possible effe the execution.

Method shorthand

```
// these objects do the same

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
user = {
  sayHi() { // same as "sayHi: function()"
    alert("Hello");
  }
};
```

“this” in methods

To access the object, a method can use the this keyword.

```
let user = {
    name: "John",
    age: 30,
    sayHi() {
        // "this" is the "current object"

        alert(this.name);

    }
};
user.sayHi(); // John
```

“this” is not bound

In JavaScript, keyword `this` behaves unlike most other programming languages. It can be used in any function.

```
let user = { name: "John" };

let admin = { name: "Admin" };


function sayHi() {
  alert( this.name );
}


// use the same function in two objects

user.f = sayHi;

admin.f = sayHi;


// these calls have different this

// "this" inside the function is the object "before the dot"

user.f(); // John  (this == user)

admin.f(); // Admin  (this == admin)


admin['f']();
// Admin (dot or square brackets access the method – doesn't matter
```

If the function sayHi() is called without an object, then it throws an error.
In JavaScript `this` is “free”, its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is “before t

Arrow functions are special: they don’t have their “own” `this`. If we reference `this` from such a function, it’s taken from the outer “normal” function.

Constructor function

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with `"new"` operator

Technically, any function can be used as a constructor.

Return from constructors

If there is a `return` statement, then the rule is simple:

- If `return` is called with an object, then the object is returned instead of `this`.
- If `return` is called with a primitive, it’s ignored.

In other words, `return` with an object returns that object, in all other cases `this` is returned.

```
function SmallUser() {

 this.name = "John";
 return // <-- returns this
}
alert( new SmallUser().name );  // John
```

We can omit parentheses after new, if it has no arguments.

Optional chaining

Note: This is a recent addition to the language. Old browsers may need polyfills.
The optional chaining ?. stops the evaluation and returns undefined if the part before ?. is undefined or null.
Please note: the `?.` syntax makes optional the value before it, but not any further.

The optional chaining `?.` is not an operator, but a special syntax construct, that also works with functions and square brackets.

```
user1.admin?.();
user1?.[key];
delete user?.name; // delete user.name if user exists
```

We can use ?. for safe reading and deleting, but not writing. The optional chaining ?. has no use at the left side of an assignment.

```
// the idea of the code below is to write user.name, if user exists


user?.name = "John"; // Error, doesn't work, because it evaluates to undefined = "John"
```

Symbol type:
A “symbol” represents a unique identifier. A symbol value may be used as an identifier for object properties; this is the data type's primary purpose.

```
let id1 = Symbol("id");

let id2 = Symbol("id");


alert(id1 == id2); // false
```

Symbols don't auto-convert to a string: (use toString() function to convert it explicitly Or get symbol.description property to show the description only**)**

```
let id = Symbol("id");

alert(id); // TypeError: Cannot convert a Symbol value to a string

let id = Symbol("id");


let user = {

  name: "John",

  [id]: 123 // not "id": 123 // we need square brackets around it
};
```

Note: Symbolic properties do not participate in for..in loop. Object.keys(user) also ignores them. In contrast, Object.assign copies both string and symbol properties.

```
// read from the global registry

let id = Symbol.for("id"); // if the symbol did not exist, it is created


// read it again (maybe from another part of the code)

let idAgain = Symbol.for("id");


// the same symbol

alert( id === idAgain ); // true
```

For global symbols, not only Symbol.for(key) returns a symbol by name, but there's a reverse call: Symbol.keyFor(sym) that does the reverse: returns a name by a global symbol.
The Symbol.keyFor internally uses the global symbol registry to look up the key for the symbol. So it doesn't work fo non-global symbols. If the symbol is not global, it won't be able to find it and returns undefined.

Object to primitive conversion:

```
let user = {
  name: "John",

  money: 1000,
  [Symbol.toPrimitive](hint) {

    alert(`hint: ${hint}`);

    return hint == "string" ? `{name: "${this.name}"}` : this.money;

  }

};



// conversions demo:

alert(user); // hint: string -> {name: "John"}

alert(+user); // hint: number -> 1000

alert(user + 500); // hint: default -> 1500
```

By default, a plain object has following toString and valueOf methods:

- The toString method returns a string "[object Object]".
- The valueOf method returns the object itself.

```
let user = {

  name: "John",
  money: 1000,

  // for hint="string"
  toString() {

    return `{name: "${this.name}"}`;

  },

  // for hint="number" or "default"

  valueOf() {

    return this.money;

  }


};


alert(user); // toString -> {name: "John"}

alert(+user); // valueOf -> 1000

alert(user + 500); // valueOf -> 1500
```

Note: In the absence of Symbol.toPrimitive and valueOf, toString(if exists) will handle all primitive conversions.

Historical notes

For historical reasons, if toString or valueOf returns an object, there's no error, but such value is ignored (like i method didn't exist). That's because in ancient times there was no good "error" concept in JavaScript.

In contrast, Symbol.toPrimitive *must* return a primitive, otherwise there will be an error.

Property flags

Object properties, besides a **value**, have three special attributes (so-called "flags"):

- **writable** – if true, the value can be changed, otherwise it's read-only.
- **enumerable** – if true, then listed in loops, otherwise not listed.
- **configurable** – if true, the property can be deleted and these attributes can be modified, otherwise not.

By default, All of them are true.

The method Object.getOwnPropertyDescriptor allows to query the *full* information about a property.

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

To change the flags, we can use Object.defineProperty.

```
Object.defineProperty(obj, propertyName, descriptor)
```

If the property exists, defineProperty updates its flags. Otherwise, it creates the property with the given value anc flags; in that case, if a flag is not supplied, it is assumed false.

There's a method Object.defineProperties(obj, descriptors) that allows to define many properties at once.

```
Object.defineProperties(user, {

  name: { value: "John", writable: false },

  surname: { value: "Smith", writable: false },

  // ...

});
```

Getters and setters

```
let obj = {

  get propName() {

    // getter, the code executed on getting obj.propName

  },

  Set propName(value) {

    // setter, the code executed on setting obj.propName = value

  }

};
```

Please note that a property can be either an accessor (has get/set methods) or a data property (has a value), not bot

Descriptors for accessor properties are different from those for data properties.

An accessor descriptor may have:

- **get** – a function without arguments, that works when a property is read,
- **set** – a function with one argument, that is called when the property is set,
- **enumerable** – same as for data properties,
- **configurable** – same as for data properties.