📅 09 Apr 2019

Objects

As we know from the chapter Data types, there are seven data types in JavaScript. Six of them are called "primitive", because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language. So we must understand them first before going in-depth anywhere else.

An object can be created with figure brackets `{…}` with an optional list of *properties*. A property is a "key: value" pair, where `key` is a string (also called a "property name"), and `value` can be anything.

We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It's easy to find a file by its name or add/remove a file.

An empty object ("empty cabinet") can be created using one of two syntaxes:

```
1  let user = new Object(); // "object constructor" syntax
2  let user = {};  // "object literal" syntax
```


Usually, the figure brackets `{...}` are used. That declaration is called an *object literal*.

Literals and properties

We can immediately put some properties into `{...}` as "key: value" pairs:

```
1  let user = {      // an object
2    name: "John",  // by key "name" store value "John"
3    age: 30        // by key "age" store value 30
4  };
```

A property has a key (also known as "name" or "identifier") before the colon `":"` and a value to the right of it.

In the `user` object, there are two properties:

1. The first property has the name `"name"` and the value `"John"`.
2. The second one has the name `"age"` and the value `30`.

The resulting `user` object can be imagined as a cabinet with two signed files labeled "name" and "age".
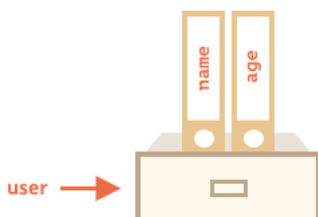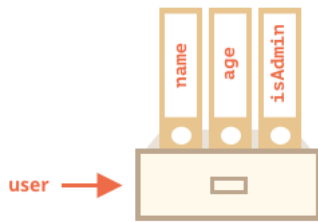

We can add, remove and read files from it any time.

Property values are accessible using the dot notation:

```
1  // get fields of the object:
2  alert( user.name ); // John
3  alert( user.age ); // 30
```
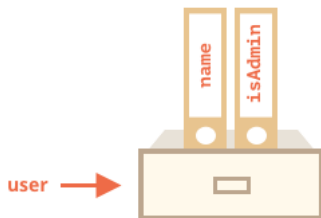
The value can be of any type. Let's add a boolean one:
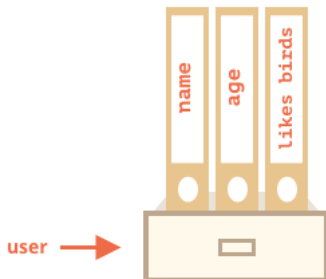
```
1  user.isAdmin = true;
```


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

```
1  delete user.age;
```


We can also use multiword property names, but then they must be quoted:

```
1  let user = {
2    name: "John",
3    age: 30,
4    "likes birds": true  // multiword property name must be quoted
5  };
```


The last property in the list may end with a comma:

```
1  let user = {
2    name: "John",
3    age: 30,
4  }
```

That is called a "trailing" or "hanging" comma. Makes it easier to add/remove/move around properties, because all lines become alike.

Square brackets

For multiword properties, the dot access doesn't work:

```
1  // this would give a syntax error
2  user.likes birds = true
```

That's because the dot requires the key to be a valid variable identifier. That is: no spaces and other limitations.

There's an alternative "square bracket notation" that works with any string:

```
1  let user = {};
2
3  // set
4  user["likes birds"] = true;
5
6  // get
7  alert(user["likes birds"]); // true
8
9  // delete
10 delete user["likes birds"];
```

Now everything is fine. Please note that the string inside the brackets is properly quoted (any type of quotes will do).

Square brackets also provide a way to obtain the property name as the result of any expression – as opposed to a literal string – like from a variable as follows:

```
1  let key = "likes birds";
2
3  // same as user["likes birds"] = true;
4  user[key] = true;
```

Here, the variable `key` may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility. The dot notation cannot be used in a similar way.

For instance:

```
1  let user = {
2    name: "John",
3    age: 30
4  };
5
6  let key = prompt("What do you want to know about the user?", "name");
7
8  // access by variable
9  alert( user[key] ); // John (if enter "name")
```

Computed properties

We can use square brackets in an object literal. That's called *computed properties*.

For instance:

```
1  let fruit = prompt("Which fruit to buy?", "apple");
2
3  let bag = {
4    [fruit]: 5, // the name of the property is taken from the variable fruit
5  };
6
7  alert( bag.apple ); // 5 if fruit="apple"
```

The meaning of a computed property is simple: `[fruit]` means that the property name should be taken from `fruit`.

So, if a visitor enters `"apple"`, `bag` will become `{apple: 5}`.

Essentially, that works the same as:

```
1  let fruit = prompt("Which fruit to buy?", "apple");
2  let bag = {};
3
4  // take property name from the fruit variable
5  bag[fruit] = 5;
```

…But looks nicer.

We can use more complex expressions inside square brackets:

```
1  let fruit = 'apple';
2  let bag = {
3    [fruit + 'Computers']: 5 // bag.appleComputers = 5
4  };
```

Square brackets are much more powerful than the dot notation. They allow any property names and variables. But they are also more cumbersome to write.

So most of the time, when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

> **ℹ️ Reserved words are allowed as property names**
>
> A variable cannot have a name equal to one of language-reserved words like "for", "let", "return" etc.
>
> But for an object property, there's no such restriction. Any name is fine:
>
> ```
> 1  let obj = {
> 2    for: 1,
> 3    let: 2,
> 4    return: 3
> 5  };
> 6
> 7  alert( obj.for + obj.let + obj.return );  // 6
> ```
>
> Basically, any name is allowed, but there's a special one: `"__proto__"` that gets special treatment for historical reasons. For instance, we can't set it to a non-object value:
>
> ```
> 1  let obj = {};
> 2  obj.__proto__ = 5;
> 3  alert(obj.__proto__); // [object Object], didn't work as intended
> ```
>
> As we see from the code, the assignment to a primitive `5` is ignored.
>
> That can become a source of bugs and even vulnerabilities if we intend to store arbitrary key-value pairs in an object, and allow a visitor to specify the keys.
>
> In that case the visitor may choose "**proto**" as the key, and the assignment logic will be ruined (as shown above).
>
> There is a way to make objects treat `__proto__` as a regular property, which we'll cover later, but first we need to know more about objects. There's also another data structure Map, that we'll learn in the chapter Map, Set, WeakMap and WeakSet, which supports arbitrary keys.

Property value shorthand

In real code we often use existing variables as values for property names.

For instance:

```
1   function makeUser(name, age) {
2     return {
3       name: name,
4       age: age
5       // ...other properties
6     };
7   }
8
9   let user = makeUser("John", 30);
10  alert(user.name); // John
```

In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there's a special *property value shorthand* to make it shorter.

Instead of `name:name` we can just write `name`, like this:

```
1   function makeUser(name, age) {
2     return {
3       name, // same as name: name
4       age   // same as age: age
5       // ...
6     };
7   }
```

We can use both normal properties and shorthands in the same object:

```
1   let user = {
2     name,  // same as name:name
3     age: 30
4   };
```

Existence check

A notable objects feature is that it's possible to access any property. There will be no error if the property doesn't exist! Accessing a non-existing property just returns `undefined`. It provides a very common way to test whether the property exists – to get it and compare vs undefined:

```
1  let user = {};
2
3  alert( user.noSuchProperty === undefined ); // true means "no such property"
```

There also exists a special operator `"in"` to check for the existence of a property.

The syntax is:

```
1  "key" in object
```

For instance:

```
1  let user = { name: "John", age: 30 };
2
3  alert( "age" in user ); // true, user.age exists
4  alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Please note that on the left side of `in` there must be a *property name*. That's usually a quoted string.

If we omit quotes, that would mean a variable containing the actual name will be tested. For instance:

```
1  let user = { age: 30 };
2
3  let key = "age";
4  alert( key in user ); // true, takes the name from key and checks for such property
```

> ℹ **Using "in" for properties that store `undefined`**
>
> Usually, the strict comparison `"=== undefined"` check works fine. But there's a special case when it fails, but `"in"` works correctly.
>
> It's when an object property exists, but stores `undefined`:
>
> ```
> 1  let obj = {
> 2    test: undefined
> 3  };
> 4
> 5  alert( obj.test ); // it's undefined, so - no such property?
> 6
> 7  alert( "test" in obj ); // true, the property does exist!
> ```
>
> In the code above, the property `obj.test` technically exists. So the `in` operator works right.
>
> Situations like this happen very rarely, because `undefined` is usually not assigned. We mostly use `null` for "unknown" or "empty" values. So the `in` operator is an exotic guest in the code.

The "for…in" loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`. This is a completely different thing from the `for(;;)` construct that we studied before.

The syntax:

```
1  for (key in object) {
2    // executes the body for each key among object properties
3  }
```

For instance, let's output all properties of `user`:

```
1  let user = {
2    name: "John",
3    age: 30,
4    isAdmin: true
5  };
6
7  for (let key in user) {
8    // keys
9    alert( key );  // name, age, isAdmin
10   // values for the keys
11   alert( user[key] ); // John, 30, true
12 }
```

Note that all "for" constructs allow us to declare the looping variable inside the loop, like `let key` here.

Also, we could use another variable name here instead of `key`. For instance, `"for (let prop in obj)"` is also widely used.

Ordered like an object

Are objects ordered? In other words, if we loop over an object, do we get all properties in the same order they were added? Can we rely on this?

The short answer is: "ordered in a special fashion": integer properties are sorted, others appear in creation order. The details follow.

As an example, let's consider an object with the phone codes:

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "Great Britain",
  // ..,
  "1": "USA"
};

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

The object may be used to suggest a list of options to the user. If we're making a site mainly for German audience then we probably want `49` to be the first.

But if we run the code, we see a totally different picture:

- USA (1) goes first
- then Switzerland (41) and so on.

The phone codes go in the ascending sorted order, because they are integers. So we see `1, 41, 44, 49`.

> ℹ️ **Integer properties? What's that?**
>
> The "integer property" term here means a string that can be converted to-and-from an integer without a change.
>
> So, "49" is an integer property name, because when it's transformed to an integer number and back, it's still the same. But "+49" and "1.2" are not:
>
> ```
> // Math.trunc is a built-in function that removes the decimal part
> alert( String(Math.trunc(Number("49"))) ); // "49", same, integer property
> alert( String(Math.trunc(Number("+49"))) ); // "49", not same "+49" ⇒ not integer prop
> alert( String(Math.trunc(Number("1.2"))) ); // "1", not same "1.2" ⇒ not integer prope
> ```

…On the other hand, if the keys are non-integer, then they are listed in the creation order, for instance:

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // add one more

// non-integer properties are listed in the creation order
for (let prop in user) {
  alert( prop ); // name, surname, age
}
```

So, to fix the issue with the phone codes, we can "cheat" by making the codes non-integer. Adding a plus `"+"` sign before each code is enough.

Like this:

```
let codes = {
  "+49": "Germany",
  "+41": "Switzerland",
  "+44": "Great Britain",
  // ..,
  "+1": "USA"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Now it works as intended.

Copying by reference

One of the fundamental differences of objects vs primitives is that they are stored and copied "by reference".

Primitive values: strings, numbers, booleans – are assigned/copied "as a whole value".

For instance:

```
1  let message = "Hello!";
2  let phrase = message;
```

As a result we have two independent variables, each one is storing the string `"Hello!"`.
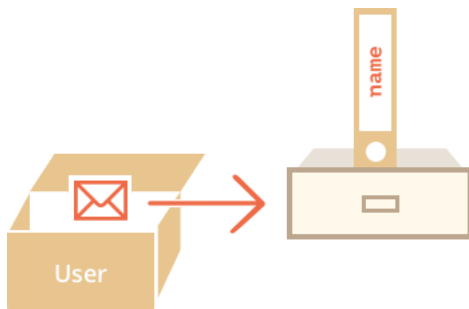

Objects are not like that.

A variable stores not the object itself, but its "address in memory", in other words "a reference" to it.

Here's the picture for the object:

```
1  let user = {
2    name: "John"
3  };
```


Here, the object is stored somewhere in memory. And the variable `user` has a "reference" to it.
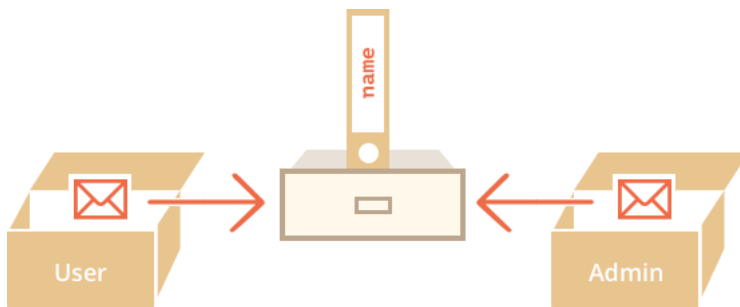
When an object variable is copied – the reference is copied, the object is not duplicated.

If we imagine an object as a cabinet, then a variable is a key to it. Copying a variable duplicates the key, but not the cabinet itself.

For instance:

```
1  let user = { name: "John" };
2
3  let admin = user; // copy the reference
```

Now we have two variables, each one with the reference to the same object:

We can use any variable to access the cabinet and modify its contents:

```
1  let user = { name: 'John' };
2
3  let admin = user;
4
```

```
5   admin.name = 'Pete'; // changed by the "admin" reference
6
7   alert(user.name); // 'Pete', changes are seen from the "user" reference
```

The example above demonstrates that there is only one object. As if we had a cabinet with two keys and used one of them (`admin`) to get into it. Then, if we later use the other key (`user`) we would see changes.

Comparison by reference

The equality `==` and strict equality `===` operators for objects work exactly the same.

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

For instance, two variables reference the same object, they are equal:

```
1   let a = {};
2   let b = a; // copy the reference
3
4   alert( a == b ); // true, both variables reference the same object
5   alert( a === b ); // true
```

And here two independent objects are not equal, even though both are empty:

```
1   let a = {};
2   let b = {}; // two independent objects
3
4   alert( a == b ); // false
```

For comparisons like `obj1 > obj2` or for a comparison against a primitive `obj == 5`, objects are converted to primitives. We'll study how object conversions work very soon, but to tell the truth, such comparisons are necessary very rarely and usually are a result of a coding mistake.

Const object

An object declared as `const` *can* be changed.

For instance:

```
1   const user = {
2     name: "John"
3   };
4
5   user.age = 25; // (*)
6
7   alert(user.age); // 25
```

It might seem that the line `(*)` would cause an error, but no, there's totally no problem. That's because `const` fixes the value of `user` itself. And here `user` stores the reference to the same object all the time. The line `(*)` goes *inside* the object, it doesn't reassign `user`.

The `const` would give an error if we try to set `user` to something else, for instance:

```
1   const user = {
2     name: "John"
3   };
4
5   // Error (can't reassign user)
6   user = {
7     name: "Pete"
8   };
```

…But what if we want to make constant object properties? So that `user.age = 25` would give an error. That's possible too. We'll cover it in the chapter Property flags and descriptors.

Cloning and merging, Object.assign

So, copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object? Create an independent copy, a clone?

That's also doable, but a little bit more difficult, because there's no built-in method for that in JavaScript. Actually, that's rarely needed. Copying by reference is good most of the time.

But if we really want that, then we need to create a new object and replicate the structure of the existing one by iterating over its properties and copying them on the primitive level.

Like this:

```
1  let user = {
2    name: "John",
3    age: 30
4  };
5
6  let clone = {}; // the new empty object
7
8  // let's copy all user properties into it
9  for (let key in user) {
10   clone[key] = user[key];
11 }
12
13 // now clone is a fully independent clone
14 clone.name = "Pete"; // changed the data in it
15
16 alert( user.name ); // still John in the original object
```

Also we can use the method Object.assign for that.

The syntax is:

```
1  Object.assign(dest, [src1, src2, src3...])
```

- Arguments `dest`, and `src1, ..., srcN` (can be as many as needed) are objects.
- It copies the properties of all objects `src1, ..., srcN` into `dest`. In other words, properties of all arguments starting from the 2nd are copied into the 1st. Then it returns `dest`.

For instance, we can use it to merge several objects into one:

```
1  let user = { name: "John" };
2
3  let permissions1 = { canView: true };
4  let permissions2 = { canEdit: true };
5
6  // copies all properties from permissions1 and permissions2 into user
7  Object.assign(user, permissions1, permissions2);
8
9  // now user = { name: "John", canView: true, canEdit: true }
```

If the receiving object (`user`) already has the same named property, it will be overwritten:

```
1  let user = { name: "John" };
2
3  // overwrite name, add isAdmin
4  Object.assign(user, { name: "Pete", isAdmin: true });
5
6  // now user = { name: "Pete", isAdmin: true }
```

We also can use `Object.assign` to replace the loop for simple cloning:

```
1  let user = {
2    name: "John",
3    age: 30
4  };
5
6  let clone = Object.assign({}, user);
```

It copies all properties of `user` into the empty object and returns it. Actually, the same as the loop, but shorter.

Until now we assumed that all properties of `user` are primitive. But properties can be references to other objects. What to do with them?

Like this:

```
1  let user = {
2    name: "John",
3    sizes: {
4      height: 182,
5      width: 50
6    }
7  };
8
9  alert( user.sizes.height ); // 182
```

Now it's not enough to copy `clone.sizes = user.sizes`, because the `user.sizes` is an object, it will be copied by reference. So `clone` and `user` will share the same sizes:

Like this:

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

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

alert( user.sizes === clone.sizes ); // true, same object

// user and clone share sizes
user.sizes.width++;       // change a property from one place
alert(clone.sizes.width); // 51, see the result from the other one
```

To fix that, we should use the cloning loop that examines each value of `user[key]` and, if it's an object, then replicate its structure as well. That is called a "deep cloning".

There's a standard algorithm for deep cloning that handles the case above and more complex cases, called the Structured cloning algorithm. In order not to reinvent the wheel, we can use a working implementation of it from the JavaScript library lodash, the method is called _.cloneDeep(obj).

Summary

Objects are associative arrays with several special features.

They store properties (key-value pairs), where:

- Property keys must be strings or symbols (usually strings).
- Values can be of any type.

To access a property, we can use:

- The dot notation: `obj.property`.
- Square brackets notation `obj["property"]`. Square brackets allow to take the key from a variable, like `obj[varWithKey]`.

Additional operators:

- To delete a property: `delete obj.prop`.
- To check if a property with the given key exists: `"key" in obj`.
- To iterate over an object: `for (let key in obj)` loop.

Objects are assigned and copied by reference. In other words, a variable stores not the "object value", but a "reference" (address in memory) for the value. So copying such a variable or passing it as a function argument copies that reference, not the object. All operations via copied references (like adding/removing properties) are performed on the same single object.

To make a "real copy" (a clone) we can use `Object.assign` or _.cloneDeep(obj).

What we've studied in this chapter is called a "plain object", or just `Object`.

There are many other kinds of objects in JavaScript:

- `Array` to store ordered data collections,
- `Date` to store the information about the date and time,
- `Error` to store the information about an error.
- …And so on.

They have their special features that we'll study later. Sometimes people say something like "Array type" or "Date type", but formally they are not types of their own, but belong to a single "object" data type. And they extend it in various ways.

Objects in JavaScript are very powerful. Here we've just scratched the surface of a topic that is really huge. We'll be closely working with objects and learning more about them in further parts of the tutorial.

✅ Tasks

Hello, object ⤴

importance: 5

Write the code, one line for each action:

1. Create an empty object `user`.
2. Add the property `name` with the value `John`.

3. Add the property `surname` with the value `Smith`.
4. Change the value of the `name` to `Pete`.
5. Remove the property `name` from the object.

Check for emptiness ☝

importance: 5

Write the function `isEmpty(obj)` which returns `true` if the object has no properties, `false` otherwise.

Should work like that:

```
let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false
```

Open a sandbox with tests.

Constant objects? ☝

importance: 5

Is it possible to change an object declared with `const`? What do you think?

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

// does it work?
user.name = "Pete";
```

Sum object properties ☝

importance: 5

We have an object storing salaries of our team:

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
}
```

Write the code to sum all salaries and store in the variable `sum`. Should be `390` in the example above.

If `salaries` is empty, then the result must be `0`.

Multiply numeric properties by 2 ☝

importance: 3

Create a function `multiplyNumeric(obj)` that multiplies all numeric properties of `obj` by `2`.

For instance:

```
// before the call
let menu = {
  width: 200,
```

```
 4    height: 300,
 5    title: "My menu"
 6  };
 7
 8  multiplyNumeric(menu);
 9
10  // after the call
11  menu = {
12    width: 400,
13    height: 600,
14    title: "My menu"
15  };
```

Please note that `multiplyNumeric` does not need to return anything. It should modify the object in-place.

P.S. Use `typeof` to check for a number here.

[Open a sandbox with tests.](#)

(solution)

Share 🐦 𝐟   🔗 Tutorial map

💬 Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)
- If you can't understand something in the article – please elaborate.