Lesson - 17

Objects and Arrays

SkillUp by Dobrea Vladislav

Lesson Plan

- Objects
- Object methods
- Arrays
- Array methods

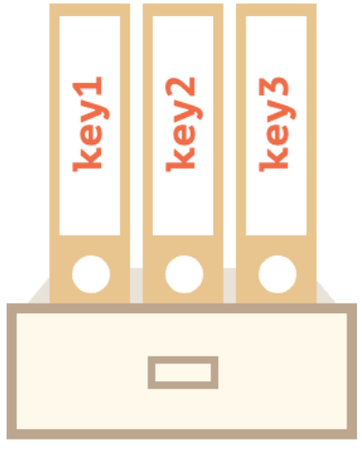
Objects

As we know from the chapter Data types, there are eight data types in JavaScript. Seven 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.

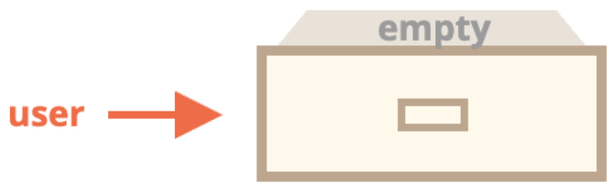
We can imagine an object as a cabinet with signed files. Every piece of data is stored in its file by the key. It’s easy to find a file by its name or add/remove a file.



Creating Objects

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

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



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

Object literals and properties

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

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

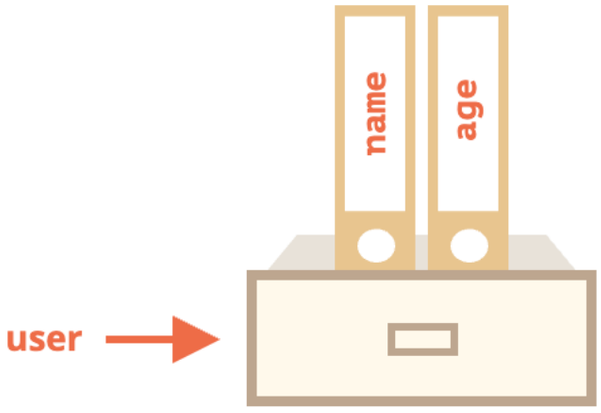
A property has a key (also known as “name” or “identifier”) before the colon `":"` and a value to the right of it.

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`.

Object literals and properties

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.

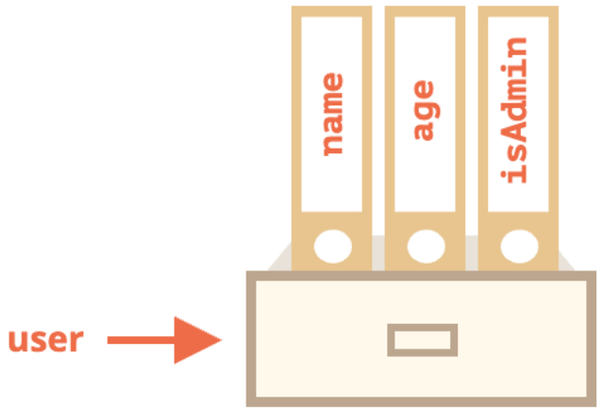
Property values are accessible using the dot notation:

```
// get property values of the object:
alert( user.name ); // John
alert( user.age ); // 30
```

Adding and removing properties

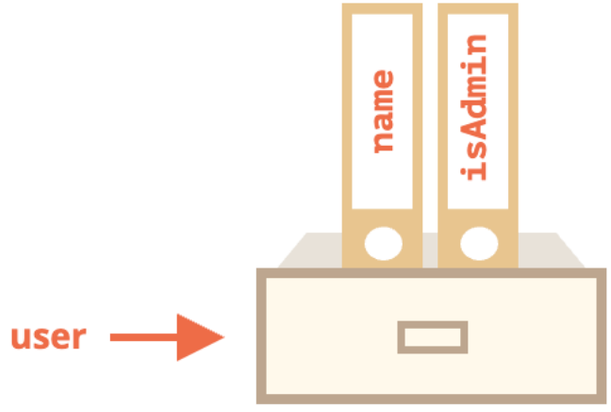
The value can be of any type. Let's add a boolean one:

```
user.isAdmin = true;
```



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

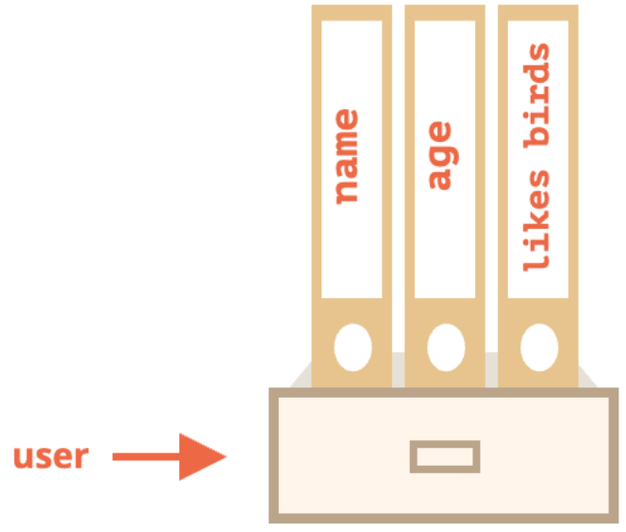
```
delete user.age;
```



Multiword properties

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

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



Last but not least

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

```
let user = {
  name: "John",
  age: 30,
}
```

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:

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

JavaScript doesn't understand that. It thinks that we address `user.likes`, and then gives a syntax error when comes across unexpected `birds`.

The dot requires the key to be a valid variable identifier. That implies: contains no spaces, doesn't start with a digit and doesn't include special characters (`$` and `_` are allowed).

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

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

// delete
delete user["likes birds"];
```

Square brackets

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:

```
let key = "likes birds";

// same as user["likes birds"] = true;
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.

For instance:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("What do you want to know about the user?", "name");

// access by variable
alert( user[key] ); // John (if enter "name")
```

Computed properties

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

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};

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}`.

Computed properties

We can use more complex expressions inside square brackets:

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

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.

Property value shorthand

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

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    //  ... other properties
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

Property value shorthand

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:

```
function makeUser(name, age) {
  return {
    name, // same as name: name
    age,  // same as age: age
    // ...
  };
}
```

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

```
let user = {
  name,  // same as name:name
  age: 30
};
```

Property name limitations

Property names (keys) must be either strings or symbols (a special type for identifiers, to be covered later).

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 name limitations

Reserved words are allowed as property names.

As we already know, 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:

```
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

We can use any string as a key, but there’s a special property named `__proto__` that gets special treatment for historical reasons.

For instance, we can’t set it to a non-object value:

```
let obj = {};
obj.__proto__ = 5; // assign a number
alert(obj.__proto__); // [object Object] - the value is an object, didn't work as intended
```

Property existence test ‘in’

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:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true means "no such property"
```

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

```
"key" in object

let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age exists
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.

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:

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

Example:

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // keys
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
}
```

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 and 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.

Integer properties

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 proper
alert( String(Math.trunc(Number("+49"))) ); // "49", not same "+49" ⇒ nc
alert( String(Math.trunc(Number("1.2"))) ); // "1", not same "1.2" ⇒ not
```

Integer properties

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”.

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

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

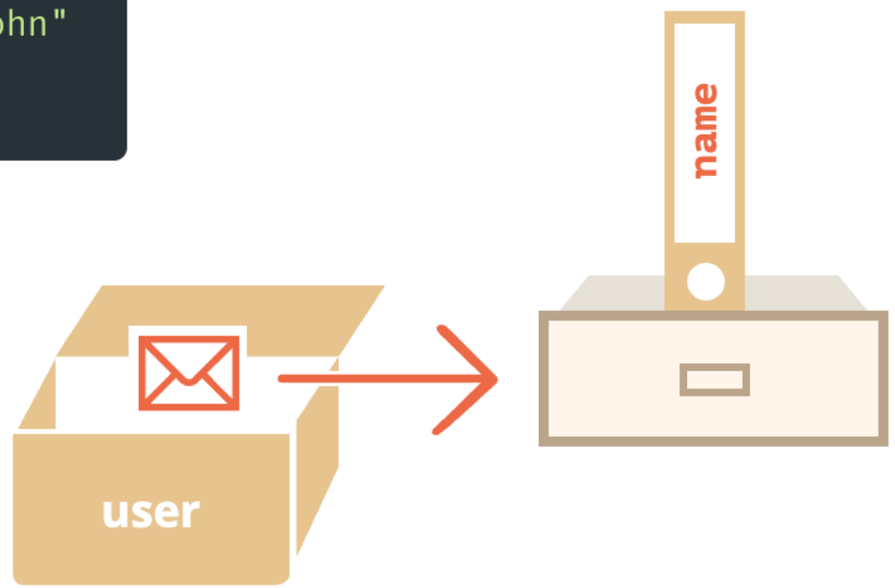
Copying by reference

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:

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



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.

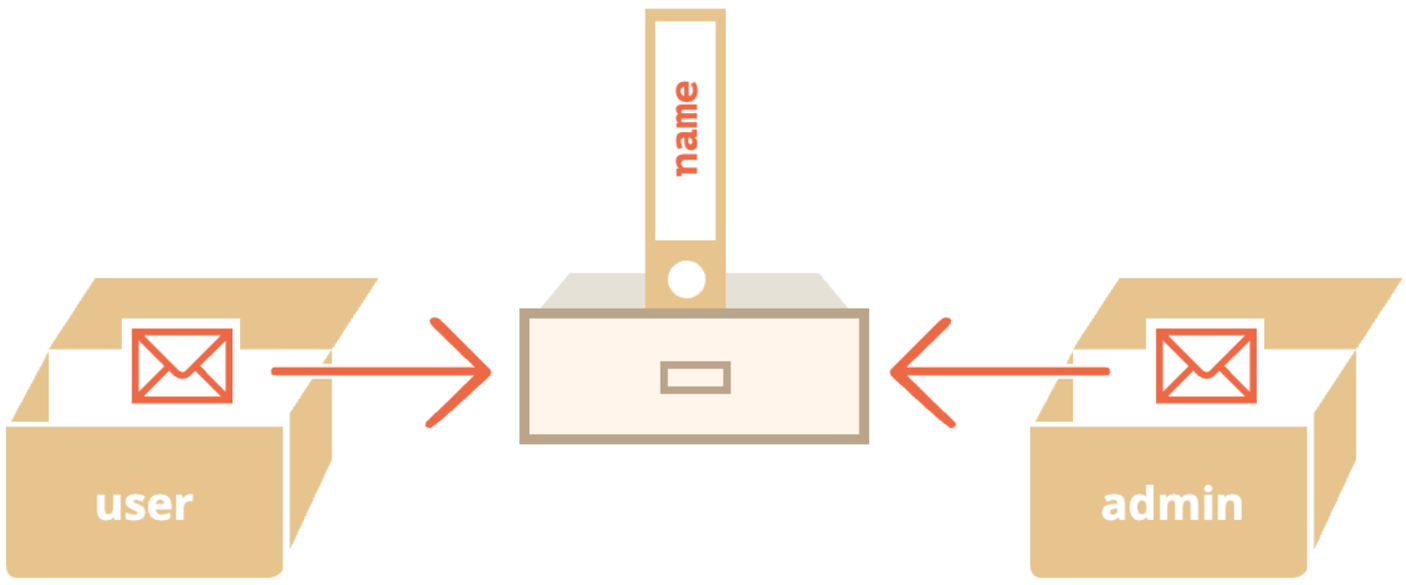
Copying by reference

For instance:

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

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:

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, if two variables reference the same object, they are equal:

```
let a = {};
let b = a; // copy the reference

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
```

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

```
let a = {};
let b = {}; // two independent objects

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

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

user.age = 25; // (*)

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 only 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:

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

// Error (can't reassign user)
user = {
  name: "Pete"
};
```

...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

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:

```
let user = {
  name: "John",
  age: 30
};

let clone = {}; // the new empty object

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

// now clone is a fully independent clone
clone.name = "Pete"; // changed the data in it

alert( user.name ); // still John in the original object
```

Object.assign()

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

The syntax is:

```
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:

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

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

// copies all properties from permissions1 and permissions2 into user
Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }
```

Object cloning

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:

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

alert( user.sizes.height ); // 182
```

Deep cloning

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:

```
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.

Summary

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.

Arrays

Objects allow you to store keyed collections of values. That's fine.

But quite often we find that we need an *ordered collection*, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.

There exists a special data structure named `Array`, to store ordered collections.

Declaration

There are two syntaxes for creating an empty array:

```
let arr = new Array();
let arr = [];
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
```

Array elements are numbered, starting with zero.

Accessing arrays

We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

...Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

The total count of the elements in the array is its `length`:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits.length ); // 3
```

Methods push/shift , shift/unshift

A queue is one of the most common uses of an array. In computer science, this means an ordered collection of elements which supports two operations:

- `push` appends an element to the end.
- `shift` get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st.

Arrays support both operations.

In practice we need it very often. For example, a queue of messages that need to be shown on-screen.

Methods push/shift , shift/unshift

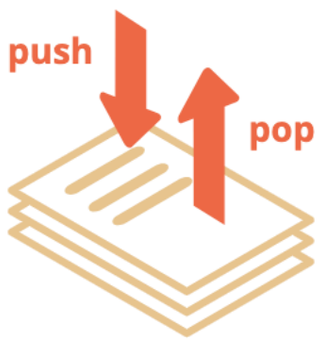
There’s another use case for arrays – the data structure named stack.

It supports two operations:

- `push` adds an element to the end.
- `pop` takes an element from the end.

So new elements are added or taken always from the “end”.

A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:



For stacks, the latest pushed item is received first, that’s also called LIFO (Last-In-First-Out) principle. For queues, we have FIFO (First-In-First-Out).

Arrays in JavaScript can work both as a queue and as a stack. They allow you to add/remove elements both to/from the beginning or the end.

In computer science the data structure that allows this, is called deque.

Methods that work with end of array

pop

Extracts the last element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.pop() ); // remove "Pear" and alert it

alert( fruits ); // Apple, Orange
```

push

Append the element to the end of the array:

```
let fruits = ["Apple", "Orange"];

fruits.push("Pear");

alert( fruits ); // Apple, Orange, Pear
```

The call `fruits.push(...)` is equal to `fruits[fruits.length] = ...`.

Methods that work with beginning of array

`shift`

Extracts the first element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];

alert( fruits.shift() ); // remove Apple and alert it

alert( fruits ); // Orange, Pear
```

`unshift`

Add the element to the beginning of the array:

```
let fruits = ["Orange", "Pear"];

fruits.unshift('Apple');

alert( fruits ); // Apple, Orange, Pear
```

Loops in array

One of the oldest ways to cycle array items is the `for` loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

But for arrays there is another form of loop, `for..of`:

```
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
```

The `for..of` doesn't give access to the number of the current element, just its value, but in most cases that's enough. And it's shorter.

A word about length

The `length` property automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but the greatest numeric index plus one.

For instance, a single element with a large index gives a big length:

```
let fruits = [];
fruits[123] = "Apple";

alert( fruits.length ); // 124
```

Note that we usually don't use arrays like that.

Another interesting thing about the `length` property is that it's writable.

If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. The process is irreversible, here's the example:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]

arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

Learning Resources

1. **Objects**
    1. https://javascript.info/object
    2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
2. **Arrays**
    1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
    2. https://javascript.info/array

Home Work pt1

1. Create a new project in GitHub lesson-16-hw
2. Create a index.html file
3. Create a script.js file and attach it to index.html
4. Let's try 5 array operations.

    1. Create an array `styles` with items "Jazz" and "Blues".
    2. Append "Rock-n-Roll" to the end.
    3. Replace the value in the middle by "Classics". Your code for finding the middle value should work for any arrays with odd length.
    4. Strip off the first value of the array and show it.
    5. Prepend `Rap` and `Reggae` to the array.

The array in the process:

```
Jazz, Blues
Jazz, Blues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

Home Work pt2

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

```
// before the call
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};

multiplyNumeric(menu);

// after the call
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

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.