

# JavaScript 3

# JavaScript

## Методи і властивості

Всі значення в JavaScript, за винятком **null** і **undefined**, містять набір допоміжних функцій і значень, доступних «**через крапку**».

Такі функції називають «**методами**», а значення - «**властивостями**».

**властивість** **str.length**

У рядка є властивість **length**, що містить довжину:

```
alert ("Привіт, світ!". length); // 13
```

Можна записати рядок в змінну, а потім запросити її властивість:

```
var str = "Привіт, світ!";  
alert (str.length); // 13
```

# JavaScript

## Методи і властивості

### Метод `str.toUpperCase()`

Також у рядків є метод `toUpperCase()`, який повертає рядок у верхньому регістрі:

```
var hello = "Привіт, світ!";  
alert (hello.toUpperCase()); // "ПРИВІТ, СВІТ!"
```

# JavaScript

## Методи і властивості

для виклику методу обов'язково потрібні **круглі дужки**.

результат звернення до **toUpperCase** без дужок:

```
var hello = "Привіт";  
alert( hello.toUpperCase ); // function...
```



**Метод** - це вбудована команда, яку потрібно викликати для отримання значення. При зверненні без дужок ми отримуємо саму цю функцію. Як правило браузер виведе її якимось так: "function toUpperCase () {...}".

# JavaScript

## Числа

```
alert (1/0); // Infinity  
alert (12345/0); // Infinity
```

**Infinity** більше будь-якого числа.

Додавання до нескінченності не змінює її.

```
alert (Infinity > 1234567890); // true  
alert (Infinity + 5 == Infinity); // true
```

Нескінченність можна привласнити і в явному вигляді: `var x = Infinity`.

Буває і мінус нескінченність `-Infinity`:

```
alert (-1 / 0); // -Infinity
```

Нескінченність можна отримати також, якщо зробити дуже велике число, для якого кількість розрядів в двійковому поданні не поміщається у відповідну частину стандартного 64-бітного формату, наприклад:

```
alert (1e500); // Infinity
```

# JavaScript

## Числа

### NaN

Якщо математична операція не може бути здійснена, то повертається спеціальне значення NaN (Not-A-Number).

Наприклад, поділ  $0/0$  в математичному сенсі не визначено, тому його результат NaN:

```
alert (0/0); // NaN
```

Значення NaN використовується для позначення математичної помилки і має такі властивості:

Значення NaN - єдине, в своєму роді, яке не дорівнює нічому, включаючи себе.

```
if (NaN == NaN) alert ( "==" ); // Жоден виклик
```

```
if (NaN === NaN) alert ( "===" ); // Не спрацює
```

# JavaScript

## Числа

### NaN

Значення NaN можна перевірити функцією `isNaN (n)`, яка перетворює аргумент до числа і повертає `true`, якщо вийшло NaN, і `false` - для будь-якого іншого значення.

```
var n = 0/0;  
alert (isNaN (n)); // true  
alert (isNaN ( "12")); // False, рядок перетворився до числа 12
```

Будь-яка операція з NaN повертає NaN.

```
alert (NaN + 1); // NaN
```

Якщо аргумент `isNaN` - не число, то він автоматично перетворюється до числа.

*Ніякі математичні операції в JavaScript не можуть привести до помилки або «обрушити» програму. У гіршому випадку, результат буде NaN.*

# JavaScript

## Числа

### isFinite(n)

Якщо ми хочемо від відвідувача отримати число, то **Infinity** або **NaN** нам не підходять. Для того, щоб відрізнити «звичайні» числа від таких спеціальних значень, існує функція **isFinite**.

Функція **isFinite(n)** перетворює аргумент до числа і повертає **true**, якщо це не NaN / Infinity / -Infinity:

```
alert (isFinite (1)); // true  
alert (isFinite (Infinity)); // false  
alert (isFinite (NaN)); // false
```



# JavaScript

Для детального ознайомлення з інформацією про типи даних «ЧИСЛО» та «РЯДОК» перейдіть за посиланнями:

<https://learn.javascript.ru/number>

<https://learn.javascript.ru/string>

# JavaScript

## Об'єкти як асоціативні масиви

Об'єкти в JavaScript:

1. це **асоціативний масив**: структура, придатна для зберігання будь-яких даних.
2. **можливості для об'єктно-орієнтованого програмування.**

### створення об'єктів

Порожній об'єкт може бути створений одним з двох синтаксисів:

1. `o = new Object ();`
2. `o = {};` // Порожні фігурні дужки

Зазвичай користуються **синтаксисом 2**, тому що він коротший.

# JavaScript

## Операції з об'єктом

Об'єкт може містити в собі будь-які значення, які називаються **властивостями об'єкта**. Доступ до властивостей здійснюється по **імені властивості**.

Наприклад, створимо об'єкт **person** для зберігання інформації про людину:

```
var person = {}; // Поки порожній
```

Основні операції з об'єктами - це **створення, отримання та видалення властивостей**.

Для звернення до **властивостей** використовується запис «через крапку», виду **об'єкт. властивість**:

```
person.name = 'John';
```

```
person.age = 25;
```

Щоб прочитати їх - також звернемося через точку:

```
alert (person.name + ':' + person.age); // " John : 25"
```

Видалення здійснюється оператором delete:

# JavaScript

## Операції з об'єктом

Якщо потрібно перевірити, чи є в об'єкті властивість з певним ключем, то є оператор: "in".

синтаксис: "prop" in obj, причому ім'я властивості - у вигляді рядка:

```
if ("name" in person) {  
    alert ("Властивість name існує!");  
}
```

***Перший спосіб***

# JavaScript

## Операції з об'єктом

### *Другий спосіб*

Частіше використовується інший спосіб - порівняння значення з `undefined`.

В JavaScript можна звернутися до будь-якої властивості об'єкта, навіть якщо його немає. Помилки не буде.

Але якщо властивість не існує, то повернеться спеціальне значення `undefined`:

```
var person = {};  
alert (person.lalala); // Undefined, немає властивості з ключем lalala
```

Таким чином ми можемо легко перевірити існування властивості - отримавши його і порівнявши з `undefined`:

```
var person = {  
  name: "Вася"  
};  
alert (person.lalala === undefined); // True, властивості немає  
alert (person.name === undefined); // False, властивість є.
```

# JavaScript

## Операції з об'єктом

технічно можливо, що властивість є, а її значенням є **undefined**:

```
var obj = {};  
obj.test = undefined; // Додали властивість із значенням undefined  
// Перевіримо наявність властивостей test і відсутнього blabla  
alert (obj.test === undefined); // true  
alert (obj.blabla === undefined); // true
```

При цьому, при простому порівнянні наявності такої властивості не можна відрізнити від її відсутності.

А оператор **in** гарантує правильний результат:

```
var obj = {};  
obj.test = undefined;  
alert ( "test" in obj); // true  
alert ( "blabla" in obj); // false
```

Як правило, в коді не привласнюють **undefined**, щоб коректно працювали обидві перевірки. А в якості значення, що позначає невизначеність, використовується **null**.

# JavaScript

## Доступ через квадратні дужки

Існує альтернативний синтаксис роботи з властивостями, що використовує квадратні дужки **об'єкт [ 'властивість' ]**:

```
var person = {};  
person [ 'name' ] = 'Вася'; // Те ж що і person.name = 'Вася'
```

Записи **person [ 'name' ]** і **person.name** ідентичні, але квадратні дужки дозволяють використовувати в якості імені властивості будь-який рядок:

```
var person = {};  
person [ 'улюблений стиль музики' ] = 'Джаз';
```

Таке привласнення було б неможливо «через точку», так інтерпретатор після першого пропуску подумає, що властивість закінчилася, і далі видасть помилку:

```
person. улюблений стиль музики = 'Джаз'; // ??? помилка
```

В обох випадках, ім'я властивості має бути рядком. Якщо використано значення іншого типу - JavaScript призведе його до рядка автоматично.

# JavaScript

## Доступ до властивості через змінну

Квадратні дужки також дозволяють звернутися до властивості, ім'я якої зберігається в змінній:

```
var person = {};  
person.age = 25;  
var key = 'age';  
alert (person [key]); // Виведе 25 (person ['age'])
```

Взагалі, якщо ім'я властивості зберігається в змінній (`var key = "age"`), то єдиний спосіб до нього звернутися - це квадратні дужки `person [key]`.

Доступ через точку використовується, якщо ми на етапі написання програми вже знаємо назву властивості. А якщо воно буде визначено по ходу виконання, наприклад, введено відвідувачем і записано в змінну, то єдиний вибір - квадратні дужки.



# JavaScript

## Оголошення з властивостями

Наступні два фрагмента коду створюють однаковий об'єкт:

```
var menuSetup = {  
  width: 300,  
  height: 200,  
  title: "Menu"  
};
```

// Те ж саме, що:

```
var menuSetup = {};  
menuSetup.width = 300;  
menuSetup.height = 200;  
menuSetup.title = 'Menu';
```

# JavaScript

## Оголошення з властивостями

Як значення можна тут же вказати і інший об'єкт:

```
var user = {  
  name: "Таня",  
  age: 25,  
  size: {  
    top: 90,  
    middle: 60,  
    bottom: 90  
  }  
}
```

```
alert (user.name) // "Таня"
```

```
alert (user.size.top) // 90
```

# JavaScript

## Об'єкти: перебір властивостей

Для перебору всіх властивостей з об'єкта використовується цикл за властивостями `for..in`. Ця конструкція відрізняється від циклу `for (;;)`.

СИНТАКСИС:

```
for (key in obj) {  
    /* ... Робити щось з obj [key] ... */  
}
```

При цьому `for..in` послідовно перебере властивості об'єкта `obj`, ім'я кожної властивості буде записано в `key` і викликано тіло циклу.

Допоміжну змінну `key` можна оголосити прямо в циклі:

```
for (var key in menu) {  
    // ...  
}
```

# JavaScript

## Об'єкти: перебір властивостей

Приклад ітерації за властивостями:

```
var menu = {  
    width: 300,  
    height: 200,  
    title: "Menu"  
};  
  
for (var key in menu) {  
    // Цей код буде викликаний для кожної властивості об'єкта  
    // ..і виведе ім'я властивості і її значення  
  
    alert ("Ключ:" + key + "значення:" + menu [key]);  
}
```

# JavaScript

## Копіювання за посиланням

В змінній, якій присвоєно об'єкт, зберігається не сам об'єкт, а «адреса його місця в пам'яті», іншими словами - «посилання» на нього.

```
var user = {  
  name: "Вася"  
};
```

При копіюванні змінної з об'єктом - копіюється посилання, а об'єкт як і раніше залишається в єдиному екземплярі.

наприклад:

```
var user = {name: "Вася"}; // В змінній - посилання  
var admin = user; // скопіювати посилання
```

Отримали дві змінні, в яких знаходяться посилання на один і той же об'єкт

# JavaScript

## Копіювання за посиланням

Так як об'єкт всього один, то зміни через будь-яку змінну видно в інших змінних:

```
var user = {name: 'Вася'};
```

```
var admin = user;
```

```
admin.name = 'НеВася'; // Поміняли дані через admin
```

```
alert (user.name); // 'НеВася', зміни видно в user
```

# JavaScript

## клонування об'єктів

Якщо потрібно скопіювати об'єкт цілком, створити саме повну незалежну копію, «клон» об'єкта, для цього потрібно пройти по об'єкту, дістати дані і скопіювати на рівні примітивів:

```
var user = {
    name: "Вася",
    age: 30
};
var clone = {}; // Новий порожній об'єкт
// Скопіюємо в нього всі властивості user
for (var key in user) {
    clone [key] = user [key];
}
// Тепер clone - повністю незалежна копія
clone.name = «НеВася»; // Поміняли дані в clone
alert (user.name); // Як і раніше "Вася"
```

Якщо ж властивості об'єктів, в свою чергу, можуть зберігати посилання на інші об'єкти, то потрібно обійти такі підоб'єкти і теж клонувати їх. Це називають «глибоким» клонуванням.

# JavaScript

**Масив** - різновид об'єкта, який призначений для зберігання пронумерованих значень

Порожній масив:

```
var arr = [];
```

Масив fruits з трьома елементами:

```
var fruits = ["Яблуко", "Апельсин", "Слива"];
```

Елементи нумеруються, починаючи з нуля.

Щоб отримати потрібний елемент з масиву - вказується його номер в квадратних дужках:

```
var fruits = ["Яблуко", "Апельсин", "Слива"];  
alert (fruits [0]); // Яблуко  
alert (fruits [1]); // Апельсин  
alert (fruits [2]); // Слива
```

Елемент можна завжди замінити:

```
fruits [2] = 'Груша'; // Тепер ["Яблуко", "Апельсин", "Груша"]
```

Або додати:

```
fruits [3] = 'Лимон'; // Тепер ["Яблуко", "Апельсин", "Груша", "Лимон"]
```



# JavaScript

Загальна кількість об'єктів, що зберігаються в масиві, міститься в його властивості `length`:

```
var fruits = ["Яблуко", "Апельсин", "Груша"];  
alert (fruits.length); // 3
```

Через `alert` можна вивести і масив цілком.

При цьому його елементи будуть перераховані через кому:

```
var fruits = ["Яблуко", "Апельсин", "Груша"];  
alert (fruits); // Яблуко, Апельсин, Груша
```

У масиві може зберігатися будь-яке число елементів будь-якого типу.

В тому числі, рядки, числа, об'єкти:

```
// Мікс значень
```

```
var arr = [1, 'Ім'я', {name: 'Петя'}, true];
```

```
// Отримати об'єкт з масиву і тут же - його властивість
```

```
alert (arr [2] .name); // Петя
```

# JavaScript

## Методи pop/push, shift/unshift

Одні із застосувань масиву - це черга і стек.

Для того, щоб реалізовувати ці структури даних, і просто для більш зручної роботи з початком і кінцем масиву існують спеціальні методи

### кінець масиву

**pop** Видаляє останній елемент з масиву і повертає його:

```
var fruits = ["Яблуко", "Апельсин", "Груша"];  
alert (fruits.pop ()); // Видалили "Груша"  
alert (fruits); // Яблуко, Апельсин
```

**push** Додає елемент в кінець масиву:

```
var fruits = ["Яблуко", "Апельсин"];  
fruits.push ("Груша");  
alert (fruits); // Яблуко, Апельсин, Груша
```

# JavaScript

## Методи pop/push, shift/unshift

### початок масиву

**shift** Видаляє з масиву перший елемент і повертає його:

```
var fruits = ["Яблуко", "Апельсин", "Груша"];  
alert (fruits.shift ()); // Видалили Яблуко  
alert (fruits); // Апельсин, Груша
```

**unshift** Додає елемент в початок масиву:

```
var fruits = ["Апельсин", "Груша"];  
fruits.unshift ('Яблуко');  
alert (fruits); // Яблуко, Апельсин, Груша
```

Методи push і unshift можуть додавати відразу по кілька елементів:

```
var fruits = ["Яблуко"];  
fruits.push ("Апельсин", "Персик");  
fruits.unshift ("Ананас", "Лимон");  
// Результат: ["Ананас", "Лимон", "Яблуко", "Апельсин", "Персик"]  
alert (fruits);
```

# JavaScript

можна привласнювати в масив будь-які властивості.

```
var fruits = []; // Створити масив  
fruits [99999] = 5; // Привласнити властивість з будь-яким номером  
fruits.age = 25; // Призначити властивість зі строковим ім'ям
```

Але масиви для того і придумані в JavaScript, щоб зручно працювати саме з впорядкованими, нумерованими даними. Як правило, немає причин використовувати масив як звичайний об'єкт, хоча технічно це і можливо.

Якщо в масиві є пропущені індекси, то при виведенні в більшості браузерів з'являються «зайві» коми, наприклад:

```
var a = [];  
a [0] = 0;  
a [5] = 5;  
alert (a); // 0 ,,,, 5
```

Ці коми з'являються тому, що алгоритм виведення масиву йде від 0 до `arr.length` і виводить все через кому. Відсутність значень дає кілька ком поспіль.

# JavaScript

## перебір елементів

Для перебору елементів використовується цикл:

```
var arr = ["Яблуко", "Апельсин", "Груша"];

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

Не використовуйте `for..in` для масивів

# JavaScript

Вбудовані методи для роботи з масивом автоматично оновлюють його довжину `length` - не кількість елементів масиву, а **останній індекс + 1**.

```
var arr = [];  
arr [1000] = true;  
alert (arr.length); // тисячу один
```

*До речі, якщо у вас елементи масиву нумеруються випадково або з великими пропусками, то варто подумати про те, щоб використовувати звичайний об'єкт. Масиви призначені саме для роботи з безперервною впорядкованою колекцією елементів.*

`length` для укорочення масиву. При зменшенні `length` масив коротшає.

```
var arr = [1, 2, 3, 4, 5];  
arr.length = 2; // Вкоротити до 2 елементів  
alert (arr); // [1, 2]  
arr.length = 5; // Повернути length назад, як було  
alert (arr [3]); // Undefined: значення не повернулися
```

Найпростіший спосіб очистити масив - це `arr.length = 0`.

# JavaScript

## Створення викликом new Array

Існує ще один синтаксис для створення масиву:

```
var arr = new Array ( "Яблуко", "Груша", "і т.п.");
```

є одна особливість. Зазвичай `new Array (елементи, ...)` створює масив з даних елементів, але якщо у нього **один аргумент-число** `new Array (число)`, то він створює масив без елементів, але із заданою довжиною.

```
var arr = new Array (2, 3);
```

```
alert (arr [0]); // 2, створений масив [2, 3], все ок
```

```
arr = new Array (2); // Створить масив [2]?
```

```
alert (arr [0]); // Undefined! у нас масив без елементів, довжини 2
```

Одержаний масив поводиться так, ніби його елементи рівні **undefined**.

# JavaScript

## багатовимірні масиви

Масиви в JavaScript можуть містити як елементи інші масиви. Це можна використовувати для створення багатовимірних масивів, наприклад матриць:

```
var matrix = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
];  
alert (matrix [1] [1]); // Центральний елемент
```



# JavaScript

## метод split

метод `split (s)`, який дозволяє перетворити рядок в масив, розбивши його по роздільнику `s`. У прикладі таким роздільником є **рядок з коми і пробілу**.

```
var names = 'Маша, Петя, Марина, Василь';  
var arr = names.split ( ', ');  
for (var i = 0; i <arr.length; i ++)  
    alert ( 'Вам повідомлення' + arr [i]);  
}
```

У методу `split` є необов'язковий другий аргумент - обмеження на кількість елементів в масиві. Якщо їх більше, ніж вказано - залишок масиву буде відкинутий:

```
alert ( "a,b,c,d".split ( ', 2')); // a, b
```

Розбивка по буквах

Виклик `split` з нового рядка розіб'є по буквах:

```
var str = "ТЕСТ";  
alert (str.split ("")); // т, е, с, т
```

# JavaScript

## метод join

Виклик `arr.join (str)` робить в точності протилежне `split`. Він бере масив і склеює його в рядок, використовуючи `str` як роздільник.

```
var arr = ['Маша', 'Петя', 'Марина', 'Василь'];  
var str = arr.join(';');  
alert (str); // Маша; Петя; Марина; Василь
```

## `new Array + join = Повторення рядка`

Код для повторення рядка 3 рази:

```
alert (new Array (4) .join ("ля")); // ляляля
```

Як видно, `new Array (4)` робить масив без елементів довжини 4, який `join` об'єднує в рядок, вставляючи між його елементами рядок "ля".

В результаті, так як елементи порожні, виходить повторення рядка.

# JavaScript

## Видалення з масиву

Так як масиви є об'єктами, то для видалення ключа можна скористатися звичайним delete:

```
var arr = [ "Я", "йду", "додому"];  
delete arr [1]; // Значення з індексом 1 видалено  
// Тепер arr = [ "Я", undefined, "додому"];  
alert (arr [1]); // undefined
```

Утворилася «дірка».

Це тому, що оператор **delete** видаляє пару «**ключ-значення**». Це - все, що він робить.

Звичайно при видаленні з масиву потрібно, щоб елементи, які залишилися змістились і заповнили проміжок,.

# JavaScript

## Метод splice

Метод `splice` - вміє видаляти елементи, вставляти елементи, замінювати елементи - по черзі і одночасно.

СИНТАКСИС:

```
arr.splice (index [, deleteCount, elem1, ..., elemN])
```

Видалити `deleteCount` елементів, починаючи з номера `index`, а потім вставити `elem1, ..., elemN` на їх місце. Повертає масив з видалених елементів.

# JavaScript

## Метод splice

видалення:

```
var arr = [ "Я", "вивчаю", "JavaScript"];  
arr.splice (1, 1); // Починаючи з позиції 1, видалити 1 елемент  
alert (arr); // Залишилося [ "Я", "JavaScript"]
```

видалимо 3 елементи і вставимо інші на їх місце:

```
var arr = [ "Я", "зараз", "вивчаю", "JavaScript"];  
// Видалити 3 перших елемента і додати інші замість них  
arr.splice (0, 3, "Ми", "вивчаємо")  
alert (arr) // тепер [ "Ми", "вивчаємо", "JavaScript"]
```

Тут видно, що splice повертає масив з видалених елементів:

```
var arr = [ "Я", "зараз", "вивчаю", "JavaScript"];  
// Видалити 2 перших елемента  
var removed = arr.splice (0, 2);  
alert (removed); // "Я", "зараз" <- array of removed elements
```

# JavaScript

## Метод splice

Метод splice також може вставляти елементи без видалення, для цього достатньо встановити `deleteCount` в 0:

```
var arr = ["Я", "вивчаю", "JavaScript"];  
// З позиції 2 Видалити 0 Вставити "складну", "мову"  
arr.splice (2, 0, "складний", "мова");  
alert (arr); // "Я", "вивчаю", "складну", "мову", "JavaScript"
```

Допускається використання негативного номера позиції, яка в цьому випадку відраховується з кінця:

```
var arr = [1, 2, 5]  
// Починаючи з позиції індексом -1 (перед останнім елементом)  
// Видалити 0 елементів,  
// Потім вставити числа 3 і 4  
arr.splice (-1, 0, 3, 4);  
alert (arr); // Результат: 1,2,3,4,5
```

# JavaScript

## метод slice

Метод `slice (begin, end)` копіює ділянку масиву від `begin` до `end`, не включаючи `end`. Вихідний масив при цьому не змінюється.

```
var arr = ["Чому", "треба", "вчити", "JavaScript"];  
var arr2 = arr.slice (1, 3); // Елементи 1, 2 (не включаючи 3)  
alert (arr2); // потрібно вчити
```

Аргументи поведуться так само, як і в `рядковому slice`:

Якщо не вказати `end` - копіювання буде до кінця масиву:

```
var arr = ["Чому", "треба", "вчити", "JavaScript"];  
alert (arr.slice (1)); // Взяти всі елементи, починаючи з номера 1
```

Можна використовувати негативні індекси, вони відраховуються з кінця:

```
var arr2 = arr.slice (-2); // Копіювати від 2-го елемента з кінця і далі
```

Якщо взагалі не вказати аргументів - скопіюється весь масив:

```
var fullCopy = arr.slice ();
```

# JavaScript

## Сортування, метод sort (fn)

Метод sort () сортує масив *на місці*. наприклад:

```
var arr = [1, 2, 15];  
arr.sort ();  
alert (arr); // 1, 15, 2
```

за замовчуванням **sort** сортує, **перетворюючи елементи до рядка**.

Тому і порядок у них рядковий, адже "2"> "15".

Для вказівки свого порядку сортування в метод **arr.sort (fn)** потрібно передати функцію **fn** від двох елементів, яка вміє порівнювати їх.

```
function compareNumeric(a, b) {  
    if (a > b) return 1;  
    if (a < b) return -1;  
}  
var arr = [ 1, 2, 15 ];  
arr.sort(compareNumeric);  
alert(arr); // 1, 2, 15
```



# JavaScript

## Сортування, метод sort (fn)

```
function compareNumeric(a, b) {  
    if (a > b) return 1;  
    if (a < b) return -1;  
}  
var arr = [ 1, 2, 15 ];  
arr.sort(compareNumeric);  
alert(arr); // 1, 2, 15
```

Зверніть увагу, передаємо в `sort ()` саме саму функцію `compareNumeric`, без виклику через дужки.

Алгоритм сортування, вбудований в JavaScript, буде передавати їй для порівняння елементи масиву. Вона повинна повертати:

- Позитивне значення, якщо  $a > b$ ,

- Негативне значення, якщо  $a < b$ ,

- Якщо рівні - можна 0, але взагалі - не важливо, що повертати, їх взаємний порядок не має значення.

# JavaScript

## Сортування, метод sort (fn)

```
function compareNumeric(a, b) {  
    if (a > b) return 1;  
    if (a < b) return -1;  
}  
var arr = [ 1, 2, 15 ];  
arr.sort(compareNumeric);  
alert(arr); // 1, 2, 15
```

Функцію `compareNumeric` для порівняння елементів-чисел можна спростити до одного рядка.

```
function compareNumeric (a, b) {  
    return a - b;  
}
```

Ця функція цілком підходить для `sort`, так як повертає позитивне число, якщо  $a > b$ , негативне, якщо навпаки, і 0, якщо числа рівні.

# JavaScript

## Сортування, метод sort (fn)

### алгоритм сортування

У методі **sort**, всередині самого інтерпретатора JavaScript, реалізований універсальний алгоритм сортування. Як правило, це «швидке сортування», додатково оптимізовано для невеликих масивів.

Він вирішує, які пари елементів і коли порівнювати, щоб впорядкувати швидше. Ми даємо йому функцію - метод порівняння, далі він викликає її сам.

Ті значення, з якими **sort** викликає функцію порівняння, можна побачити, якщо вставити в неї **alert**:

```
[1, -2, 15, 2, 0, 8] .sort (function (a, b) {  
    alert (a + "<>" + b);  
});
```

# JavaScript

## reverse

Метод `arr.reverse ()` змінює порядок елементів в масиві на зворотний.

```
var arr = [1, 2, 3];  
arr.reverse ();  
  
alert (arr); // 3,2,1
```

# JavaScript

## concat

Метод `arr.concat (value1, value2, ... valueN)` створює новий масив, в який копіюються елементи з `arr`, а також `value1, value2, ... valueN`.

```
var arr = [1, 2];  
var newArr = arr.concat (3, 4);  
alert (newArr); // 1,2,3,4
```

У `concat` є одна особливість. Якщо аргумент `concat` - **МАСИВ**, то `concat` додає елементи з нього.

```
var arr = [1, 2];  
var newArr = arr.concat ([3, 4], 5); // Те ж саме, що arr.concat (3,4,5)  
alert (newArr); // 1,2,3,4,5
```

# JavaScript

## indexOf / lastIndexOf

Ці методи не підтримуються в IE8-.

Метод «`arr.indexOf (searchElement [, fromIndex])`» повертає номер елемента `searchElement` в масиві `arr` або `-1`, якщо його немає.

Пошук починається з номера `fromIndex`, якщо він вказаний. Якщо немає - з початку масиву.

Для пошуку використовується суворе порівняння `===`.

```
var arr = [1, 0, false];  
alert (arr.indexOf (0)); // 1  
alert (arr.indexOf (false)); // 2  
alert (arr.indexOf (null)); // -1
```

по синтаксису він повністю аналогічний методу `indexOf` для рядків.

Метод «`arr.lastIndexOf (searchElement [, fromIndex])`» шукає справа-наліво: з кінця масиву або з номера `fromIndex`, якщо він вказаний.

Методи `indexOf / lastIndexOf` здійснюють пошук перебором

Якщо потрібно перевірити, чи існує значення в масиві - його потрібно перебрати.

Тільки так. Внутрішня реалізація `indexOf / lastIndexOf` здійснює повний перебір, аналогічний циклу `for` по масиву. Чим довше масив, тим довше він буде працювати.

# JavaScript

## Колекція унікальних елементів

Розглянемо задачу - є колекція рядків, і потрібно швидко перевіряти: чи є в них якийсь елемент. Масив для цього не підходить через повільне `indexOf`. Але підходить об'єкту. Доступ до властивості об'єкта здійснюється дуже швидко, так що можна зробити всі елементи ключами об'єкта і перевіряти, чи є вже такий ключ.

Наприклад, організуємо таку перевірку для колекції рядків "div", "a" і "form":

```
var store = {}; // Об'єкт для колекції
var items = [ "div", "a", "form" ];
for (var i = 0; i < items.length; i++) {
  var key = items [i]; // Для кожного елемента створюємо властивість
  store [key] = true; // Значення тут не важливо
}
```

Тепер для перевірки, чи є ключ `key`, досить виконати `if (store [key])`. Якщо є - можна використовувати значення, якщо ні - додати.

Таке рішення працює тільки з рядками, але можна застосувати до будь-яких елементів, для яких можна обчислити рядковий «унікальний ключ».

# JavaScript

## Object.keys (obj)

властивості об'єкта можна перебрати в циклі for..in.

Якщо ми хочемо працювати з ними у вигляді масиву, то використовується метод **Object.keys (obj)**. Він підтримується всюди, крім IE8-:

```
var user = {  
    name: "Петя",  
    age: 30  
}  
  
var keys = Object.keys (user);  
  
alert (keys); // Name, age
```



# JavaScript

## forEach

Метод «`arr.forEach (callback [, thisArg])`» використовується для перебору масиву.

Він для кожного елемента масиву викликає функцію `callback`.

Цій функції передає три параметра `callback (item, i, arr)`:

`item` - черговий елемент масиву.

`i` - його номер.

`arr` - масив, який перебирається.

наприклад:

```
var arr = [ "Яблуко", "Апельсин", "Груша"];  
arr.forEach (function (item, i, arr) {  
    alert (i + ":" + item + "(масив:" + arr + ")");  
});
```

Метод `forEach` нічого не повертає, його використовують тільки для перебору, як більш «елегантний» варіант, ніж звичайний цикл `for`.

# JavaScript

## filter

Метод «`arr.filter (callback [, thisArg])`» використовується для фільтрації масиву через функцію.

Він створює новий масив, в який увійдуть тільки ті елементи `arr`, для яких виклик `callback (item, i, arr)` поверне `true`.

наприклад:

```
var arr = [1, -1, 2, -2, 3];  
var positiveArr = arr.filter (function (number) {  
    return number > 0;  
});  
alert (positiveArr); // 1,2,3
```

# JavaScript

## map

Метод «`arr.map (callback [, thisArg])`» використовується для трансформації масиву.

Він створює новий масив, який буде складатися з результатів виклику `callback (item, i, arr)` для кожного елемента `arr`.

наприклад:

```
var names = ['HTML', 'CSS', 'JavaScript'];  
var nameLengths = names.map (function (name) {  
    return name.length;  
});  
// Отримали масив з довжинами  
alert (nameLengths); // 4,3,10
```

# JavaScript

## every / some

Ці методи використовуються для перевірки масиву.

Метод «`arr.every (callback [, thisArg])`» повертає `true`, якщо виклик `callback` поверне `true` для кожного елемента `arr`.

Метод «`arr.some (callback [, thisArg])`» повертає `true`, якщо виклик `callback` поверне `true` для якого-небудь елемента `arr`.

```
var arr = [1, -1, 2, -2, 3];
```

```
function isPositive (number) {  
    return number > 0;  
}
```

```
alert (arr.every (isPositive)); // False, не всі позитивні
```

```
alert (arr.some (isPositive)); // True, є хоч одне позитивне
```

# JavaScript

## reduce/reduceRight

Метод «`arr.reduce (callback [, initialValue])`» використовується для послідовної обробки кожного елемента масиву із збереженням проміжного результату.

Метод `reduce` використовується для обчислення на основі масиву будь-якого єдиного значення, інакше говорять «для згортання масиву».

Він застосовує функцію `callback` по черзі до кожного елемента масиву зліва направо, зберігаючи при цьому проміжний результат.

Аргументи функції `callback (previousValue, currentItem, index, arr)`:

- `previousValue` - останній результат виклику функції, він же «проміжний результат».

- `currentItem` - поточний елемент масиву, елементи перебираються по черзі зліва-направо.

- `index` - номер поточного елемента.

- `arr` - оброблюваний масив.

Крім `callback`, методу можна передати «початкове значення» - аргумент `initialValue`. Якщо він є, то на першому виклику значення `previousValue` дорівнюватиме `initialValue`, а якщо у `reduce` немає другого аргументу, то воно дорівнює першому елементу масиву, а перебір починається з другого.

# JavaScript

## reduce/reduceRight

Наприклад, в якості «згортання» ми хочемо отримати суму всіх елементів масиву.

```
var arr = [1, 2, 3, 4, 5]
// Для кожного елемента масиву запустити функцію,
// Проміжний результат передавати першим аргументом далі
var result = arr.reduce (function (sum, current) {
    return sum + current;
}, 0);

alert (result); // 15
```

# JavaScript

## reduce/reduceRight

```
var arr = [1, 2, 3, 4, 5]
// Для кожного елемента масиву запустити функцію,
// Проміжний результат передавати першим аргументом далі
var result = arr.reduce (function (sum, current) {
    return sum + current;
}, 0);

alert (result); // 15
```

При першому запуску **sum** - початкове значення, з якого починаються обчислення, дорівнює **нулю** (другий аргумент **reduce**).

Спочатку анонімна функція викликається з цим початковим значенням і першим елементом масиву, результат запам'ятовується і передається в наступний виклик, вже з другим аргументом масиву, потім нове значення бере участь в обчисленнях з третім аргументом і так далі.

Потік обчислень виходить такий

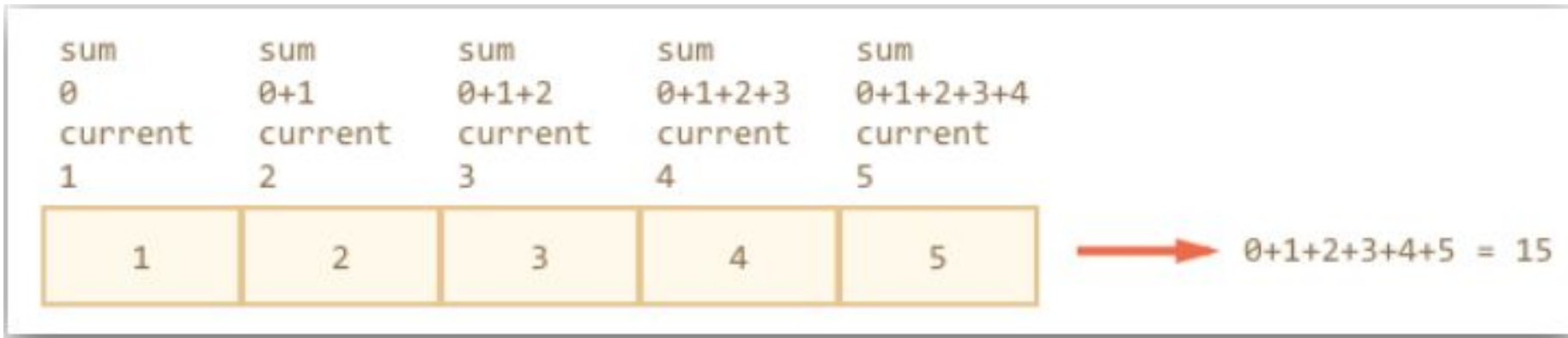
# JavaScript

## reduce/reduceRight

```
var arr = [1, 2, 3, 4, 5]
// Для кожного елемента масиву запустити функцію,
// Проміжний результат передавати першим аргументом далі
var result = arr.reduce(function (sum, current) {
    return sum + current;
}, 0);

alert (result); // 15
```

Потік обчислень виходить такий





# JavaScript

## reduce/reduceRight

повний набір аргументів функції для `reduce` включає в себе `function (sum, current, i, array)`, тобто номер поточного виклику `i` і весь масив `arr`, але тут в них немає потреби.

Подивимося, що буде, якщо не вказати `initialValue` у виклику `arr.reduce`:

```
var arr = [1, 2, 3, 4, 5]
// Прибрали 0 в кінці
var result = arr.reduce(function (sum, current) {
    return sum + current
});
alert (result); // 15
```

Результат - точно такий же, це тому, що при відсутності `initialValue` в якості першого значення береться перший елемент масиву, а перебір стартує з другого.

Таблиця обчислень буде така ж, за винятком першого рядка.

Метод `arr.reduceRight` працює аналогічно, але йде по масиву справа-наліво.