**Лекція 9. Масив: методи перебору. Псевдомасив аргументів "arguments". Дата і час.**

1. Масив: перебираючі методи

2. Псевдомасив аргументів "arguments".

3.Дата і час.

**1.** **Масив: методи перебору**.

Сучасний стандарт JavaScript надає багато методів для «розумного» перебору масивів, які є в сучасних браузерах ...

... Ну а для їх підтримки в IE8- просто підключіть бібліотеку ES5-shim.

**forEach**

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

Він для кожного елемента масиву викликає функцію callback.

Цій функції він 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 + ")" );

});

Другий, необов'язковий аргумент forEach дозволяє вказати контекст this для callback. Ми обговоримо його в деталях трохи пізніше.

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

**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

**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 с довжинами

alert( nameLengths ); // 4,3,10

**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 елементу 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

**reduce / reduceRight**

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

Це один з найскладніших методів для роботи з масивами. Але його варто освоїти, тому що часом з його допомогою можна в кілька рядків вирішити задачу, яка інакше зажадала б в рази більше місця і часу.

Метод reduce  використовується для обчислення на основі масиву будь-якого єдиного значення, інакше кажуть «для згортки масиву». Трохи далі ми розберемо приклад для обчислення суми.

Він застосовує функцію callback по черзі до кожного елементу 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.

Найпростіше зрозуміти роботу методу reduce на прикладі.

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

Ось рішення в один рядок:

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);

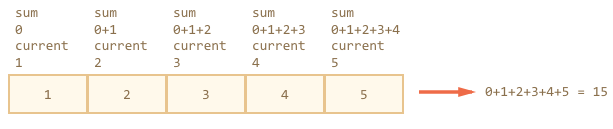
alert( result ); // 15

Розберемо, що в ньому відбувається.

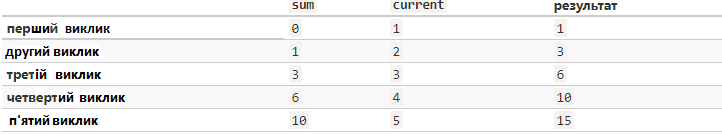
При першому запуску sum - початкове значення, з якого починаються обчислення, дорівнює нулю (другий аргумент reduce).

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



У вигляді таблиці де кожен рядок - виклик функції на черговому елементі масиву:



Як видно, результат попереднього виклику передається в перший аргумент наступного.

До речі, 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]

// забрали 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 вирахуванням першого рядка.

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

**Висновок:**

Ми розглянули методи:

* forEach - для перебору масиву.
* filter - для фільтрації масиву.
* every/some - для перевірки масиву.
* map - для трансформації масиву в масив.
* reduce/reduceRight - для проходу по масиву з обчисленням значення.

У багатьох ситуаціях їх використання дозволяє написати код коротше і зрозуміліше, ніж звичайний перебір через for.

**2. Псевдомасив аргументів "arguments".**

В JavaScript будь-яка функція може бути викликана з будь-якою кількістю аргументів.

Наприклад:

function go(a,b) {

alert("a="+a+", b="+b);

}

go(1); // a=1, b=undefined

go(1,2); // a=1, b=2

go(1,2,3); // a=1, b=2, третій аргумент не викличе помилку

**В JavaScript немає «перевантаження» функцій**

У деяких мовах програміст може створити дві функції з однаковим ім'ям, але різним набором аргументів, а при виклику інтерпретатор сам вибере потрібну:

function log(a) {

...

}

function log(a, b, c) {

...

}

log(a); // викликається перша функція

log(a, b, c); // викликається друга функція

Це називається «поліморфізмом функцій» або «перевантаженням функцій». В JavaScript нічого подібного немає.

**Може бути тільки одна функція з ім'ям log, яка викликається з будь-якими аргументами.**

А вже всередині вона може подивитися, з чим викликана і по-різному відпрацювати.

У прикладі вище другого оголошення log просто перевизначить перше.

**Доступ до «зайвих» аргументів**

Як отримати значення аргументів, яких немає в списку параметрів?

Доступ до них здійснюється через «псевдо-масив» arguments.

Він містить список аргументів за номерами: arguments[0], arguments[1] ..., а також властивість length.

Наприклад, виведемо список всіх аргументів:

function sayHi() {

for (var i = 0; i < arguments.length; i++) {

alert( "Привіт, " + arguments[i] );

}

}

sayHi("Вінні", "П’ятачок"); // 'Привіт, Вінні', 'Привіт, П’ятачок'

Всі параметри знаходяться в arguments, навіть якщо вони є в списку. Код вище спрацював би також, будь функція оголошена sayHi(a,b,c).

**Зв'язок між arguments і параметрами**

У старому стандарті JavaScript псевдо-масив arguments і змінні-параметри посилаються на одні і ті ж значення.

В результаті зміни arguments впливають на параметри і навпаки.

Наприклад:

function f(x) {

arguments[0] = 5; // змінює змінну x

alert( x ); // 5

}

f(1);

Навпаки:

function f(x) {

x = 5;

alert( arguments[0] ); // 5, обновленный x

}

f(1);

У сучасній редакції стандарту ця поведінка змінена. Аргументи відділені від локальних змінних:

function f(x) {

"use strict"; // для браузерів с підтримкою строгого режиму

arguments[0] = 5;

alert( x ); // не 5, а 1! Змінна "відв’язана" від arguments }

f(1);

**Якщо ви не використовуєте строгий режим, то щоб змінні не змінювалися «несподівано», рекомендується ніколи не змінювати arguments.**

**arguments - це не масив**

Часта помилка новачків - спроба застосувати методи Array до arguments. Це неможливо:

function sayHi() {

var a = arguments.shift(); // помилка! немає такого методу!

}

sayHi(1);

Справа в тому, що arguments - це не масив Array.

Насправді, це звичайний об'єкт, просто ключі числові і є length. На цьому схожість закінчується. Ніяких особливих методів у нього немає, і методи масивів він теж не підтримує.

Втім, ніхто не заважає зробити звичайний масив з arguments, наприклад так:

var args = [];

for (var i = 0; i < arguments.length; i++) {

args[i] = arguments[i];

}

Такі об'єкти іноді називають *«колекціями»* або *«псевдомасивами»*.

**Приклад: копіювання властивостей copy (dst, src1, src2 ...)**

Іноді постає завдання - скопіювати в існуючий об'єкт властивості з одного або декількох інших.

Напишемо для цього функцію copy. Вона буде працювати з будь-яким числом аргументів, завдяки використанню arguments.

Синтаксис:

**copy (dst, src1, src2 ...)**

Копіює властивості з об'єктів src1, src2,... в об'єкт dst. Повертає новий об'єкт.

Використання:

* Для об'єднання декількох об'єктів в один:

var vasya = {

age: 21,

name: 'Вася',

surname: 'Петренко'

};

var user = {

isAdmin: false,

isEmailConfirmed: true

};

var student = {

university: 'My university'

};

// додати до vasya властивість з user і student

copy(vasya, user, student);

alert( vasya.isAdmin ); // false

alert( vasya.university ); // My university

* Для створення копії об'єкта user:

// скопіює всі властивості в пустий об’єкт

var userClone = copy({}, user);

Такий «клон» об'єкта може стати в нагоді там, де ми хочемо змінювати його властивості, при цьому не чіпаючи вихідний об'єкт user.

**Аргументи за замовчуванням через ||**

Якщо функція викликана з меншою кількістю аргументів, ніж вказано, то відсутні аргументи вважаються рівними undefined.

Найчастіше в разі відсутності аргументу ми хочемо присвоїти йому деяке «стандартне» значення або, інакше кажучи, значення «за замовчуванням». Це можна зручно зробити за допомогою оператора логічне АБО ||.

Наприклад, функція showWarning, описана нижче, повинна показувати попередження. Для цього вона приймає ширину width, висоту height, заголовок title і вміст contents, але більша частина цих аргументів необов'язкова:

function showWarning(width, height, title, contents) {

width = width || 200; // якщо не вказана width, то width = 200

height = height || 100; // якщо нема height, то height = 100

title = title || "Попередження";

//...

}

Це відмінно працює в тих ситуаціях, коли «нормальне» значення параметра в логічному контексті відмінно від false. У коді вище, при передачі width = 0 або width = null, оператор АБО замінить його на значення за замовчуванням.

А що, якщо ми хочемо використовувати значення за замовчуванням тільки якщо width === undefined? У цьому випадку оператор АБО вже не підійде, потрібно поставити явну перевірку:

function showWarning(width, height, title, contents) {

if (width === undefined) width = 200;

if (height === undefined) height = 100;

if (title === undefined) title = "Попередження";

//...

}

**«Іменовані аргументи»**

Іменовані аргументи - альтернативна техніка роботи з аргументами, яка взагалі не використовує arguments.

Деякі мови програмування дозволяють передати параметри якось так: f(width=100, height=200), тобто по іменах, а що не передано, тих аргументів немає. Це дуже зручно в тих випадках, коли аргументів багато, складно запам'ятати їх порядок і більшість взагалі не треба передавати.

Така ситуація часто зустрічається в компонентах інтерфейсу. Наприклад, у «меню» може бути маса налаштувань відображення, які можна «підкрутити» але зазвичай потрібно передати всього один-два головних параметра, а решта візьмуться за замовчуванням.

В JavaScript для цих цілей використовується передача аргументів у вигляді об'єкта, а в його властивості ми передаємо параметри.

Виходить так:

function showWarning(options) {

var width = options.width || 200; // за замовчуванням

var height = options.height || 100;

var contents = options.contents || "Попередження";

// ...

}

Викликати таку функцію дуже легко. Досить передати об'єкт аргументів, вказавши в ньому тільки потрібні:

showWarning({

contents: "Ви викликали функцію" // і все зрозуміло!

});

Порівняємо це з передачею аргументів через список:

showWarning(null, null, "Попередження!");

// думки програміста "а що це за null, null на початку? потрібно дивитись опис функції"

Чи не правда, об'єкт - набагато простіше і зрозуміліше?

Ще один бонус крім гарного вигляду - можливість повторного використання об'єкта аргументів:

var opts = {

width: 400,

height: 200,

contents: "Текст"

};

showWarning(opts);

opts.contents = "Інший текст";

showWarning(opts); // викликати з новим текстом, без копіювання інших аргументів

Іменовані аргументи застосовуються в багатьох JavaScript-фреймворках.

**Висновок:**

* Повний список аргументів, з якими викликана функція, доступний через arguments.
* Це псевдомасив, тобто об'єкт, який схожий на масив, в ньому є нумеровані властивості і length, але методів масиву у нього немає.
* Для вказівки аргументів за замовчуванням, в тих випадках, коли вони заздалегідь не false, зручний оператор ||.

У тих випадках, коли можливих аргументів багато і, особливо, коли більшість їх мають значення за замовчуванням, замість роботи з arguments організовують передачу даних через об'єкт, який як правило називають options.

Можливий і гібридний підхід, при якому перший аргумент обов'язковий, а другий - options, який містить всілякі додаткові параметри:

function showMessage(text, options) {

// показати повідомлення text, налаштування показу вказані в options

}

**3.Дата і час.**

Для роботи з датою і часом в JavaScript використовуються об'єкти Date.

**Створення**

Для створення нового об'єкта типу Date використовується один з синтаксисів:

**new Date()**

Створює об'єкт Date з поточною датою і часом:

var now = new Date();

alert( now );

**new Date(milliseconds)**

Створює об'єкт Date, значення якого дорівнює кількості мілісекунд (1/1000 секунди), що пройшли з 1 січня 1970 року GMT + 0.

// 24 години після 01.01.1970 GMT+0

var Jan02\_1970 = new Date(3600 \* 24 \* 1000);

alert( Jan02\_1970 );

**new Date(datestring)**

Якщо єдиний аргумент - рядок, використовується виклик Date.parse (див. Далі) для читання дати з неї.

**new Date(year, month, date, hours, minutes, seconds, ms)**

Дату можна створити, використовуючи компоненти в місцевій часовій зоні. Для цього формату є обов'язковими тільки перші два аргументи. Відсутні параметри, починаючи з hours вважаються рівними нулю, а date - одиниці.

Зауважимо:

* Рік year повинен бути з 4 цифр.
* Відлік місяців month починається з нуля 0.

Наприклад:

new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 січня 2011, 00:00:00

new Date(2011, 0, 1); // те саме, години/секунди за замовчуванням дорівнює 0

Дата задана з точністю до мілісекунд:

var date = new Date(2011, 0, 1, 2, 3, 4, 567);

alert( date ); // 1.01.2011, 02:03:04.567

**Отримання компонентів дати**

Для доступу до компонентів дати-часу об'єкта Date використовуються наступні методи:

**getFullYear()**

Отримати рік (з 4 цифр)

**getMonth()**

Отримати місяць, від 0 до 11.

**getDate()**

Отримати число місяця, від 1 до 31.

**getHours(),getMinutes(), getSeconds(), getMilliseconds()**

Отримати відповідні компоненти.

Додатково можна отримати день тижня:

**getDay()**

Отримати номер дня в тижні. Тиждень в JavaScript починається з неділі, так що результат буде числом **від 0 (неділя) до 6 (субота)**.

**Всі методи, зазначені вище, повертають результат для місцевої тимчасової зони.**

Існують також UTC-варіанти цих методів, які повертають день, місяць, рік і т.п. для зони GMT + 0 (UTC): getUTCFullYear(), getUTCMonth(), getUTCDay(). Тобто, відразу після "get" вставляється "UTC".

Якщо ваше локальне час зрушено щодо UTC, то наступний код покаже різні години:

// поточна дата

var date = new Date();

// час в поточній часовій зоні

alert( date.getHours() );

// котра зараз година в Лондоне?

// година в зоні GMT+0

alert( date.getUTCHours() );

Крім описаних вище, існують два спеціальних методу без UTC-варіанту:

**getTime()**

Повертає число мілісекунд, що пройшли з 1 січня 1970 року GMT + 0, тобто того ж виду, який використовується в конструкторі new Date (milliseconds).

**getTimezoneOffset()**

Повертає різницю між місцевим і UTC-часом, в хвилинах.

alert( new Date().getTimezoneOffset() ); // Для GMT-1 виведе 60

Наступні методи дозволяють встановлювати компоненти дати і часу:

* setFullYear(year [, month, date])
* setMonth(month [, date])
* setDate(date)
* setHours(hour [, min, sec, ms])
* setMinutes(min [, sec, ms])
* setSeconds(sec [, ms])
* setMilliseconds(ms)
* setTime(milliseconds) (встановлює всю дату по мілісекунд з 01.01.1970 UTC)

Всі вони, крім setTime(), мають також UTC-варіантом, наприклад: setUTCHours().

Як видно, деякі методи можуть встановлювати кілька компонентів дати одночасно, зокрема, setHours. При цьому якщо якась компонента не вказана, вона не змінюється. наприклад:

var today = new Date;

today.setHours(0);

alert( today ); // сьогодні, але година змінена на 0

today.setHours(0, 0, 0, 0);

alert( today ); // сьогодні, рівно 00:00:00.

**Автовиправлення дати**

Автовиправлення - дуже зручне властивість об'єктів Date. Воно полягає в тому, що можна встановлювати явно некоректні компоненти (наприклад 32 січня), а об'єкт сам себе поправить.

var d = new Date(2013, 0, 32); // 32 січня 2013 ?!?

alert(d); // ... це 1 лютого 2013!

**Неправильні компоненти дати автоматично розподіляються по іншим.**

Наприклад, потрібно збільшити на 2 дня дату «28 лютого 2011». Може бути так, що це буде 2 березня, а може бути і 1 березня, якщо рік високосний. Але нам про все це думати не потрібно. Просто додаємо два дні. Решту зробить Date:

var d = new Date(2011, 1, 28);

d.setDate(d.getDate() + 2);

alert( d ); // 2 березня, 2011

Також це використовують для отримання дати, віддаленій від наявної на потрібний проміжок часу. Наприклад, отримаємо дату на 70 секунд велику поточної:

var d = new Date();

d.setSeconds(d.getSeconds() + 70);

alert( d ); // виведе коректну дату

Можна встановити і нульові, і навіть негативні компоненти. наприклад:

var d = new Date;

d.setDate(1); // поставити перше число місяця

alert( d );

d.setDate(0); // нульового числа немає, буде останнє число попереднього місяця

alert( d );

var d = new Date;

d.setDate(-1); // передостаннє число попереднього місяця

alert( d );

**Перетворення до числа, різниця дат**

Коли об'єкт Date використовується в числовому контексті, він перетвориться в кількість мілісекунд:

alert(+new Date) // +date те саме, що: +date.valueOf()

**Важливий побічний ефект: дати можна вичитати, результат віднімання об'єктів Date - їх тимчасова різниця, в мілісекундах.**

Це використовують для вимірювання часу:

var start = new Date; // засікли час

// щось зробити

for (var i = 0; i < 100000; i++) {

var doSomething = i \* i \* i;

}

var end = new Date; // кінець виміру

alert( "Цикл занял " + (end - start) + " ms" );

**Бенчмаркінг**

Припустимо, у нас є кілька варіантів вирішення задачі, кожен описаний функцією.

Як дізнатися, який швидше?

Для прикладу візьмемо дві функції, які бігають по масиву:

function walkIn(arr) {

for (var key in arr) arr[key]++

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

Щоб поміряти, яка з них швидше, не можна запустити один раз walkIn, один раз walkLength і заміряти різницю. Одноразовий запуск ненадійний, будь-яка міні-перешкода спотворить результат.

Для правильного бенчмаркінгу функція запускається багато разів, щоб сам тест зайняв чимало часу. Це зведе вплив перешкод до мінімуму. Складну процедуру можна запускати 100 раз, просту - 1000 разів ...

Поміряти, яка з функцій швидше:

var arr = [];

for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {

for (var key in arr) arr[key]++;

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

function bench(f) {

var date = new Date();

for (var i = 0; i < 10000; i++) f(arr);

return new Date() - date;

}

alert( 'Час walkIn: ' + bench(walkIn) + 'мс' );

alert( 'Час walkLength: ' + bench(walkLength) + 'мс' );

Тепер уявімо собі, що під час першого бенчмаркінгу bench (walkIn) комп'ютер щось робив паралельно важливе (раптом) і це займало ресурси, а під час другого - перестав. Реальна ситуація? Звичайно реальна, особливо на сучасних ОС, де багато процесів одночасно.

**Набагато більш надійні результати можна отримати, якщо весь пакет тестів прогнати багато разів.**

var arr = [];

for (var i = 0; i < 1000; i++) arr[i] = 0;

function walkIn(arr) {

for (var key in arr) arr[key]++;

}

function walkLength(arr) {

for (var i = 0; i < arr.length; i++) arr[i]++;

}

function bench(f) {

var date = new Date();

for (var i = 0; i < 1000; i++) f(arr);

return new Date() - date;

}

// bench для кожного тесту запустимо багато разів, чергуючи

var timeIn = 0,

timeLength = 0;

for (var i = 0; i < 100; i++) {

timeIn += bench(walkIn);

timeLength += bench(walkLength);

}

alert( 'Час walkIn: ' + timeIn + 'мс' );

alert( 'Час walkLength: ' + timeLength + 'мс' );

**Форматування і висновок дат**

У всіх браузерах, крім IE10-, підтримується новий стандарт Ecma 402, який додає спеціальні методи для форматування дат.

Це робиться викликом date.toLocaleString(локаль, опції), в якому можна задати багато налаштувань. Він дозволяє вказати, які параметри дати потрібно вивести, і ряд налаштувань виведення, після чого інтерпретатор сам сформує рядок.

Приклад з багатьма параметрами дати і російським, потім англійською (США) форматуванням:

var date = new Date(2014, 11, 31, 12, 30, 0);

var options = {

era: 'long',

year: 'numeric',

month: 'long',

day: 'numeric',

weekday: 'long',

timezone: 'UTC',

hour: 'numeric',

minute: 'numeric',

second: 'numeric'

};

alert( date.toLocaleString("ru", options) ); // середа, 31 грудня 2014 г. н.ч. 12:30:00

alert( date.toLocaleString("en-US", options) ); // Wednesday, December 31, 2014 Anno Domini 12:30:00 PM

**Методи виведення без локалізації:**

toString(), toDateString(), toTimeString() Повертають стандартне рядкове подання, не заданий жорстко в стандарті, а залежне від браузера. Єдина вимога до нього - читаність людиною. Метод toString повертає дату цілком, toDateString() і toTimeString() - тільки дату і час відповідно.

var d = new Date();

alert( d.toString() ); // вивід, схожий на 'Wed Jan 26 2011 16:40:50 GMT+0300'

toUTCString() Те ж саме, що toString(), але дата в зоні UTC.

toISOString() Повертає дату в форматі ISO Деталі формату будуть далі. Підтримується сучасними браузерами, що не підтримується IE8-.

var d = new Date();

alert( d.toISOString() ); // вивід, схожий на '2011-01-26T13:51:50.417Z'

Якщо хочеться мати велику гнучкість і кросбраузерну, то також можна скористатися спеціальною бібліотекою, наприклад Moment.JS або написати свою функцію форматування.

**Розбір рядка, Date.parse**

Всі сучасні браузери, включаючи IE9 +, розуміють дати в спрощеному форматі ISO 8601 Extended.

Цей формат виглядає так: YYYY-MM-DDTHH:mm:ss.sssZ, де:

* YYYY-MM-DD - дата в форматі рік-місяць-день.
* Звичайний символ T використовується як роздільник.
* HH:mm:ss.sss - час: годинник-хвилини-секунди-мілісекунди.
* Частина 'Z' позначає тимчасову зону - в форматі +-hh:mm, або символ Z, що позначає UTC. За стандартом її можна не вказувати, тоді UTC, але в Safari з цим помилка, так що краще вказувати завжди.

Також можливі укорочені варіанти, наприклад YYYY-MM-DD або YYYY-MM або навіть тільки YYYY.

Метод Date.parse(str) розбирає рядок str в такому форматі і повертає відповідне їй кількість мілісекунд. Якщо це неможливо, Date.parse повертає NaN.

Наприклад:

var msUTC = Date.parse('2012-01-26T13:51:50.417Z'); // зона UTC

alert( msUTC ); // 1327571510417 (число мілісекунд)

З таймзоною -07: 00 GMT:

var ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert( ms ); // 1327611110417 (число мілісекунд)

**Метод Date.now ()**

Метод Date.now() повертає дату відразу у вигляді мілісекунд.

Технічно, він аналогічний викликом +new Date(), але на відміну від нього не створює проміжний об'єкт дати, а тому - у багато разів швидше.

Його використання особливо рекомендується там, де продуктивність при роботі з датами критична. Зазвичай це не на веб-сторінках, а, наприклад, в розробці ігор на JavaScript.

**Висновок:**

* Дата і час представлені в JavaScript одним об'єктом: Date. Створити «тільки час» при цьому не можна, воно повинно бути з датою. Список методів Date ви можете знайти в довіднику Date або вище.
* Відлік місяців починається з нуля.
* Відлік днів тижня (для getDay()) теж починається з нуля (і цієї неділі).
* Об'єкт Date зручний тим, що автокорректіруется. Завдяки цьому легко зрушувати дати.
* При перетворенні до числа об'єкт Date дає кількість мілісекунд, що пройшли з 1 січня 1970 UTC. Побічний наслідок - дати можна вичитати, результатом буде різниця в мілісекундах.
* Для отримання поточної дати в мілісекундах краще використовувати Date.now(), щоб не створювати зайвий об'єкт Date (крім IE8-)
* Для бенчмаркінгу краще використовувати performance.now() (крім IE9-), він в 1000 разів точніше.