**Лекція 6. Робота з числами та рядками у JS.**

1. Числа.
2. Рядки.

1.Всі числа в JavaScript, як цілі так і дробові, мають тип Number і зберігаються в 64-бітному форматі IEEE-754, також відомому як «double precision».

Тут ми розглянемо різні тонкощі, пов'язані з роботою з числами в JavaScript.

**Способи запису**

В JavaScript можна записувати числа не тільки в десятковій, а й в шістнадцятковій (починається з 0x) системі числення:

alert( 0xFF ); // 255 в шістнадцяткова система

Також доступна запис в «науковому форматі» (ще говорять «запис з плаваючою точкою»), який виглядає як <число>e<кількість нулів>.

Наприклад, 1e3 – це 1 з 3 нулями, тобто 1000.

// ще приклад наукової форми: 3 з 5 нулями

alert( 3e5 ); // 300000

Якщо кількість нулів від’ємна, то число зсувається вправо за десяткову точку, так що виходить десятковий дріб:

// тут 3 зміщено 5 разів вправо, за десяткову точку.

alert( 3e-5 ); // 0.00003 <-- 5 нулів, включаючи початковий нуль

**Ділення на нуль, Infinity**

Уявіть, що ви збираєтеся створити нову мову ... Люди будуть називати її «JavaScript» (або «LiveScript» ... неважливо).

Що повинно відбуватися при спробі поділу на нуль?

Як правило, помилка в програмі ... У всякому разі, в більшості мов програмування це саме так.

Але творець JavaScript вирішив піти математично правильним шляхом. Адже чим менше дільник, тим більше результат. При розподілі на дуже-дуже маленьке число повинно вийти дуже велике. В математичному аналізі це описується через межі, і якщо мати на увазі межа, то в якості результату ділення на 0 ми отримуємо «нескінченність», яка позначається символом ∞ (в JavaScript Infinity).

alert( 1 / 0 ); // Infinity

alert( 12345 / 0 ); // Infinity

**Infinity - особливе чисельне значення, яке поводиться в точності як математична нескінченність ∞.**

* Infinity більше любого числа.
* Додавання до безкінечності 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 ну дуже велике число, для якого кількість розрядів в двійковому представленні не поміщаються у відповідну частину стандартного 64-бітного формату, наприклад:

alert( 1e500 ); // Infinity

**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.

Наступний код нічого не виведе:

if (NaN == NaN) alert( "==" ); // Ні один виклик

if (NaN === NaN) alert( "===" ); // не спрацює

* Значення NaN можна перевірити спеціальний метод isNaN(n), який 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 на звичайне числа 12

* Значення NaN «чіпляється». Будь-яка операція з NaN повертає NaN.

alert( NaN + 1 ); // NaN

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

**Цікавий спосіб перевірки на NaN**

Звідси випливає цікавий спосіб перевірки значення на NaN: можна перевірити значення на рівність самому собі, якщо не дорівнює - то NaN:

var n = 0 / 0;

if (n !== n) alert( 'n = NaN!' );

Це працює, але для наочності краще використовувати isNaN(n).

**Математичні операції в JS безпечні**

Ніякі математичні операції в JavaScript не можуть привести до помилки або «обрушити» програму.

У гіршому випадку результат буде NaN.

**isFinite(n)**

Отже, в JavaScript є звичайні числа і три спеціальних числових значення: NaN, Infinity і -Infinity.

Той факт, що вони, хоч і особливі, але числа, демонструється роботою оператора +:

var value = prompt("Введите Infinity", 'Infinity');

var number = +value;

alert( number ); // Infinity, плюс перетворив рядок "Infinity" до такого "числа"

Зазвичай 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

**Перетворення до числа**

Більшість арифметичних операцій і математичних функцій перетворять значення в число автоматично.

Для того щоб зробити це явно, зазвичай перед значенням ставлять унарний плюс '+':

var s = "12.34";

alert( +s ); // 12.34

При цьому, якщо рядок не є в точності числом, то результат буде NaN:

alert( +"12test" ); // NaN

Єдиний виняток - пробільні символи на початку і в кінці рядка, які ігноруються:

alert( +" -12" ); // -12

alert( +" \n34 \n" ); // 34, переклад рядка \ n є пробільним символом

alert( +"" ); // 0, порожній рядок стає нулем

alert( +"1 2" ); // NaN, пробіл посередині числа - помилка

Аналогічним чином відбувається перетворення і в інших математичних операторах і функціях:

alert( '12.34' / "-2" ); // -6.17

**М'яке перетворення: parseInt і parseFloat**

У світі HTML / CSS багато значення не є в точності числами. Наприклад метрики CSS: 10pt або -12px.

Оператор '+' для таких значень поверне NaN:

alert(+"12px") // NaN

Для зручного читання таких значень існує функція parseInt:

alert( parseInt('12px') ); // 12

**Функція parseInt і її аналог parseFloat перетворять рядок символ за символом, поки це можливо.**

При виникненні помилки повертається число, яке вийшло. Функція parseInt читає з рядка ціле число, а parseFloat - дробове.

alert( parseInt('12px') ) // 12, помилка на символі 'p'

alert( parseFloat('12.3.4') ) // 12.3, помилка на другій точці

Звичайно, існують ситуації, коли parseInt/parseFloat повертають NaN. Це відбувається при помилці на першому ж символі:

alert( parseInt('a123') ); // NaN

Функція parseInt також дозволяє вказати систему числення, тобто зчитувати числа, задані в шістнадцятковій і інших системах числення:

alert( parseInt('FF', 16) ); // 255

**Перевірка на число**

Для перевірки рядка на число можна використовувати функцію isNaN(str).

Вона перетворює рядок в число аналогічно +, а потім поверне true, якщо це NaN, тобто якщо перетворення не вдалося:

var x = prompt("Введіть значення", "-11.5");

if (isNaN(x)) {

alert( "Рядок перетворився в NaN. Не число" );

} else {

alert( "Число" );

}

Однак, у такої перевірки є дві особливості:

Порожній рядок і рядок з пробільних символів перетворюються до 0, тому вважаються числами.

Якщо застосувати таку перевірку не до рядка, то можуть бути сюрпризи, зокрема isNaN вважатиме числами значення false, true, null, так як вони хоча і не числа, але перетворюються до них.

alert( isNaN(null) ); // false - не NaN, тобто "число"

alert( isNaN("\n \n") ); // false - не NaN, тобто "число"

Якщо така поведінка допустима, то isNaN - прийнятний варіант.

Якщо ж потрібна дійсно точна перевірка на число, яка не вважає числом рядок з прогалин, логічні та спеціальні значення, а також відсікає Infinity - використовуйте наступну функцію isNumeric:

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}Розберемося, як вона працює. Почнемо справа.

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

Таким чином, права частина відсіє свідомо не-числа, але залишить такі значення як true/false/null і порожній рядок '', так як вони коректно перетворюються в числа.

Для їх перевірки потрібна ліва частина. Виклик parseFloat(true/false/null/'') поверне NaN для цих значень.

Так влаштована функція parseFloat: вона перетворює аргумент до рядку, тобто true/false/null стають "true"/"false"/"null", а потім зчитує з неї число, при цьому порожній рядок дає NaN.

В результаті відсівається все, крім рядків-чисел і простих чисел.

**toString (система числення)**

Як показано вище, числа можна записувати не тільки в 10-вій, а й в 16-вій системі. Але буває і протилежне завдання: отримати 16-кове подання числа. Для цього використовується метод toString(основа системи), наприклад:

var n = 255;

alert( n.toString(16) ); // ff

Зокрема, це використовують для роботи з значеннями кольору в браузері, виду #AABBCC.

Основа може бути будь-яким від 2 до 36.

Основа 2 буває корисно для налагодження побітових операцій:

var n = 4;

alert( n.toString(2) ); // 100

Основа 36 (за кількістю букв в англійському алфавіті - 26, разом з цифрами, яких 10) використовується для того, щоб «кодувати» число у вигляді буквено-цифровий рядки. У цій системі числення спочатку використовуються цифри, а потім букви від a до z:

var n = 1234567890;

alert( n.toString(36) ); // kf12oi

За допомогою такого кодування можна «вкоротити» довгий цифровий ідентифікатор, наприклад щоб видати його в якості URL.

**Округлення**

Одна з найбільш частих операцій з числом - округлення. В JavaScript існують цілих 3 функції для цього.

**Math.floor**

округлює вниз

**Math.ceil**

округлює вгору

**Math.round**

Округлює до найближчого цілого

alert( Math.floor(3.1) ); // 3

alert( Math.ceil(3.1) ); // 4

alert( Math.round(3.1) ); // 3

**Округлення до заданої точності**

Для округлення до потрібної цифри після коми можна помножити і поділити на 10 з потрібною кількістю нулів. Наприклад, округлимо 3.456 до 2-го знака після коми:

var n = 3.456;

alert( Math.round(n \* 100) / 100 ); // 3.456 -> 345.6 -> 346 -> 3.46

Таким чином можна округляти число і вгору і вниз.

**num.toFixed (precision)**

Існує також спеціальний метод num.toFixed(precision), який округлює число num до точності precision і повертає результат у вигляді рядка:

var n = 12.34;

alert( n.toFixed(1) ); // "12.3"

Округлення йде до найближчого значення, аналогічно Math.round:

var n = 12.36;

alert( n.toFixed(1) ); // "12.4"

Підсумковий рядок, при необхідності, доповнюється нулями до потрібної точності:

var n = 12.34;

alert( n.toFixed(5) ); // "12.34000", додані нулі до 5 знаків після коми

Якщо нам потрібно саме число, то ми можемо отримати його, застосувавши '+' до результату n.toFixed(..):

var n = 12.34;

alert( +n.toFixed(5) ); // 12.34

**Метод toFixed не еквівалентний Math.round!**

Наприклад, зробимо округлення до одного знака після коми з використанням двох способів: toFixed і Math.round з множенням і діленням:

var price = 6.35;

alert(+ price.toFixed(1) ); // 6.3

alert( Math.round(price \* 10) / 10 ); // 6.4

Як видно результат різний! Варіант округлення через Math.round вийшов більш коректним, так як за загальноприйнятими правилами 5 округляється вгору. А toFixed може округлити його як вгору, так і вниз. Чому? Скоро дізнаємося!

**Неточні обчислення**

Запустіть цей приклад:

alert( 0.1 + 0.2 == 0.3 );

Він вивів false. Результат дещо дивний, чи не так? Можливо, помилка в браузері? Поміняйте браузер, запустіть ще раз.

Добре, тепер ми можемо бути впевнені: 0.1 + 0.2 це не 0.3. Але тоді що ж це?

alert( 0.1 + 0.2 ); // 0.30000000000000004

Як бачите, сталася невелика обчислювальна помилка, результат складання 0.1 + 0.2 трохи більше, ніж 0.3.

alert( 0.1 + 0.2 > 0.3 ); // true

Вся справа в тому, що в стандарті IEEE 754 на число виділяється рівно 8 байт (= 64 біта), не більше і не менше.

Число 0.1 (одна десята) записується просто в десятковому форматі. Але в двійковій системі числення це нескінченна дріб, так як одиниця на десять в двійковій системі так просто не ділиться. Також нескінченної дробом є 0.2 (=2/10).

Двійкове значення нескінченних дробів зберігається тільки до певного знака, тому виникає неточність. Її навіть можна побачити:

alert( 0.1.toFixed(20) ); // 0.10000000000000000555

Коли ми складаємо 0.1 і 0.2, то дві неточності складаються, отримуємо незначну, але все ж помилку в обчисленнях.

Звичайно, це не означає, що точні обчислення для таких чисел неможливі. Вони можливі. І навіть необхідні.

Наприклад, є два способи скласти 0.1 і 0.2:

1. Зробити їх цілими, скласти, а потім поділити:

alert( (0.1 \* 10 + 0.2 \* 10) / 10 ); // 0.3

Це працює, так як числа 0.1\*10 = 1 і 0.2\*10 = 2 можуть бути точно представлені в двійковій системі.

2. Скласти, а потім округлити до розумного знака після коми. Округлення до 10-го знака зазвичай буває достатньо, щоб відсікти помилку обчислень:

var result = 0.1 + 0.2;

alert( +result.toFixed(1) ); // 0.3

**Цікавий приклад**

Число, яке росте саме по собі:

alert( 9999999999999999 ); // виведе 10000000000000000

Причина та ж - втрата точності.

З 64 біт, відведених на число, самі цифри числа займають до 52 біт, решта 11 біт зберігають позицію десяткового дробу і один біт - знак. Так що якщо 52 біт не вистачає на цифри, то при записі пропадуть молодші розряди.

Інтерпретатор не видасть помилку, але в результаті вийде «не зовсім те число», що ми і бачимо в прикладі вище. Як то кажуть: «як зміг, так записав».

Заради справедливості зауважимо, що в точності те ж саме відбувається в будь-якій іншій мові, де використовується формат IEEE 754, включаючи Java, C, PHP, Ruby, Perl.

**Інші математичні методи**

JavaScript надає базові тригонометричні і деякі інші функції для роботи з числами.

**Тригонометрія**

Вбудовані функції для тригонометричних обчислень:

**Math.acos(x)**

Повертає арккосинус x (в радіанах)

**Math.asin(x)**

Повертає арксинус x (в радіанах)

**Math.atan(x)**

Повертає арктангенс x (в радіанах)

**Math.atan2(y, x)**

Повертає кут до точки (y, x). Опис функції: Atan2.

**Math.sin(x)**

Обчислює синус x

**Math.cos(x)**

Обчислює косинус x

**Math.tan(x)**

Повертає тангенс x

**Функції загального призначення**

Різні корисні функції:

**Math.sqrt(x)**

Повертає квадратний корінь з x.

**Math.log(x)**

Повертає натуральний (за основою e) логарифм x.

**Math.pow(x, exp)**

Зводить число в ступінь, повертає xexp, наприклад Math.pow(2,3) = 8. Працює в тому числі з дробовими і від’ємними степенями, наприклад: Math.pow(4, -1/2) = 0.5.

**Math.abs(x)**

Повертає абсолютне значення числа

**Math.exp(x)**

Повертає ex, де e - основа натуральних логарифмів.

**Math.max(a, b, c...)**

Повертає найбільший зі списку аргументів

**Math.min(a, b, c...)**

Повертає найменший зі списку аргументів

**Math.random()**

Повертає псевдовипадкове число в інтервалі [0,1) - тобто між 0 (включно) і 1 (не включаючи).

**Форматування**

Для красивого виведення чисел у стандарті ECMA 402 є метод toLocaleString():

var number = 123456789;

alert( number.toLocaleString() ); // 123 456 789

Його підтримують всі сучасні браузери, крім IE10- (для яких потрібно підключити бібліотеку Intl.JS). Він також вміє форматувати валюту і відсотки.

**Висновки**

* Числа можуть бути записані в десятковій, шістнадцятковій системах, а також «науковим» способом.
* В JavaScript існує числове значення нескінченність Infinity.
* Помилка обчислень дає NaN.
* Арифметичні і математичні функції перетворять рядок в точності в число, ігноруючи початкові і кінцеві пробіли.
* Функції parseInt/parseFloat роблять числа з рядків, які починаються з числа.
* Є 3 способи округлення: Math.floor, Math.round, Math.ceil. Для округлення до потрібного знака використовуйте +n.toFixed(p) або трюк з множенням і діленням на 10p.
* Дробові числа дають помилку обчислень. При необхідності її можна відсікти округленням до потрібного знака.
* Випадкові числа від 0 до 1 генеруються за допомогою Math.random(), решта - перетворенням з них.

1. **Рядки**

В JavaScript будь-які текстові дані є рядками. Не існує окремого типу «символ», який є в ряді інших мов.

Внутрішнім форматом рядків, незалежно від кодування сторінки, є Юнікод (Unicode).

**Створення рядків**

Рядки створюються за допомогою подвійних або одинарних лапок:

var text = "мій рядок";

var anotherText = 'ще рядок';

var str = "012345";

В JavaScript немає різниці між подвійними і одинарними лапками.

**Спеціальні символи**

Рядки можуть містити спеціальні символи. Самий часто використовуваний з таких символів - це «новий рядок».

Він позначається як \n, наприклад:

alert( 'Привіт\nСвіт' ); // виведе "Світ" на новому рядку

| ***Спеціальні символи*** | |
| --- | --- |
| **Символ** | **Опис** |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage return |
| \t | Tab |
| \uNNNN | Символ в кодуванні Юнікод з шістнадцятковим кодом `NNNN`. Наприклад, `\ u00A9` - Юнікодние представлення символу копірайт © |

**Екранування спеціальних символів**

Якщо рядок в одинарних лапках, то внутрішні одинарні лапки всередині повинні бути екрановані, тобто забезпечені зворотним слешем \', ось так:

var str = 'I\'m a JavaScript programmer';

У подвійних лапках - екрануються внутрішні подвійні:

var str = "I'm a JavaScript \"programmer\" ";

alert( str ); // I'm a JavaScript "programmer"

Екранування служить виключно для правильного сприйняття рядки JavaScript. У пам'яті рядок буде містити сам символ без '\'. Ви можете побачити це, запустивши приклад вище.

Сам символ зворотного слеша '\' є службовим, тому завжди екранується, тобто пишеться як \\:

var str = ' символ \\ ';

alert( str ); // символ \

Заекранувати можна будь-який символ. Якщо він не спеціальний, то нічого не станеться:

alert( "\a" ); // a

// ідентично alert( "a" );

**Методи і властивості**

Тут ми розглянемо методи і властивості рядків, з деякими з яких ми знайомилися раніше.

**Довжина length**

Одне з найбільш частих дій з рядком - це отримання її довжини:

var str = "My\n"; // 3 символа. Третій - перенос рядка

alert( str.length ); // 3

**Доступ до символів**

Щоб отримати цей символ, натисніть виклик charAt(позиция). Перший символ має позицію 0:

var str = "jQuery";

alert( str.charAt(0) ); // "j"

В JavaScript **немає окремого типу «символ»**, так що charAt повертає рядок, що складається з обраного символу.

Також для доступу до символу можна використовувати квадратні дужки:

var str = "Я - сучасний браузер!";

alert( str[0] ); // "Я"

Різниця між цим способом і charAt полягає в тому, що якщо символу немає - charAt видає порожню рядок, а дужки - undefined:

alert( "".charAt(0) ); // пустий рядок

alert( "" [0] ); // undefined

Взагалі ж метод charAt існує з історичних причин, адже квадратні дужки - простіше і коротше.

**Виклик методу - завжди з дужками**

Зверніть увагу, str.length - це властивість рядка, а str.charAt(pos) - метод, тобто функція.

Звернення до методу завжди йде з дужками, а до властивості - без дужок.

**Зміна рядків**

Вміст рядка в JavaScript не можна змінювати. Не можна взяти символ посередині і замінити його. Як тільки рядок створена - вона така назавжди.

Можна лише створити цілком новий рядок і привласнити в змінну замість старої, наприклад:

var str = "рядок";

str = str[3] + str[4] + str[5];

alert( str ); // док

**Зміна регістра**

Методи toLowerCase() і toUpperCase() змінюють регістр рядка на нижній / верхній:

alert( "Интерфейс".toUpperCase() ); // ІНТЕРФЕЙС

Приклад нижче отримує перший символ і призводить його до нижнього регістру:

alert( "Інтерфейс" [0].toLowerCase() ); // 'І'

**Пошук підрядка**

Для пошуку підрядка є метод indexOf (підрядок [, початкова\_позіція]).

Він повертає позицію, на якій знаходиться підрядок або -1, якщо нічого не знайдено. наприклад:

var str = "Widget with id";

alert( str.indexOf("Widget") ); // 0, т.к. "Widget" знайдений прямо в початку str

alert( str.indexOf("id") ); // 1, т.к. "id" знайдений, починаючи з позиції 1

alert( str.indexOf("widget") ); // -1, не знайдено, так як пошук враховує регістр

Необов'язковий другий аргумент дозволяє шукати, починаючи з вказаної позиції. Наприклад, перший раз "id" з'являється на позиції 1. Щоб знайти його наступна поява - запустимо пошук з позиції 2:

var str = "Widget with id";

alert(str.indexOf("id", 2)) // 12, пошук розпочато з позиції 2

**Пошук всіх входжень**

Щоб знайти всі входження підрядка, потрібно запустити indexOf  в циклі. Як тільки отримуємо чергову позицію - починаємо наступний пошук.

Приклад такого циклу:

var str = " Віслюк Іа-Іа подивився на віадук"; // шукаємо в цьому рядку

var target = "Іа"; // ціль пошуку

var pos = 0;

while (true) {

var foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( foundPos ); // знайшли на цій позиції

pos = foundPos + 1; // продовжити пошук з наступною

}

Такий цикл починає пошук з позиції 0, потім знайшовши підстроку на позиції foundPos, наступний пошук продовжить з позиції pos = foundPos+1, і так далі, поки щось знаходить.

Втім, той же алгоритм можна записати і коротше:

var str = " Віслюк Іа-Іа подивився на віадук"; // шукаємо в цьому рядку

var target = "Іа"; // ціль пошуку

var pos = -1;

while ((pos = str.indexOf(target, pos + 1)) != -1) {

alert( pos );

}

**Взяття підрядка: substring, substr, slice**

В JavaScript існують цілих 3 (!) методи для взяття підрядка, з невеликими відмінностями між ними.

**substring(start [, end])**

Метод substring(start, end) повертає підрядок з позиції start до, але не включаючи end.

var str = "stringify";

alert(str.substring(0,1)); // "s", символи з позиції 0 по 1 не включаючи 1.

Якщо аргумент end відсутній, то йде до кінця рядка:

var str = "stringify";

alert(str.substring(2)); // ringify, символи з позиції 2 до кінця

**substr(start [, length])**

Перший аргумент має такий же зміст, як і в substring, а другий містить не кінцеву позицію, а кількість символів.

var str = "stringify";

str = str.substr(2,4); // ring, з 2-ї позиції 4 символа

alert(str)

Якщо другий аргумент немає - мається на увазі «до кінця рядка».

**slice(start [, end])**

Повертає частину рядка від позиції start до, але не включаючи, позиції end. Сенс параметрів - такий же як в substring.

**Негативні аргументи**

Різниця між substring і slice - в тому, як вони працюють з негативними і виходять за кордон рядки аргументами:

**substring(start, end)**

Негативні аргументи інтерпретуються як рівні нулю. Занадто великі значення усікаються до довжини рядка:

alert( "testme".substring(-2) ); // "testme", -2 стає 0

Крім того, якщо start > end, то аргументи міняються місцями, тобто повертається ділянку рядки між start і end:

alert( "testme".substring(4, -1) ); // "test"

// -1 стає 0 -> отримали substring(4, 0)

// 4 > 0, так що аргументи змінюються місцями -> substring(0, 4) = "test"

**slice**

Негативні значення відраховуються від кінця рядка:

alert( "testme".slice(-2) ); // "me", від 2 позиції с кінця

alert( "testme".slice(1, -1) ); // "estm", від 1 позиції до першої з кінця.

Це набагато зручніше, ніж дивна логіка substring.

Негативне значення першого параметра підтримується в substr у всіх браузерах, крім IE8-.

Якщо вибирати з цих трьох методів один, для використання в більшості ситуацій - то це буде slice: він і негативні аргументи підтримує і працює найбільш очевидно.

**Кодування Юнікод**

Як ми знаємо, символи порівнюються в алфавітному порядку 'А' < 'Б' < 'В' < ... < 'Я'.

Але є кілька дивних ситуацій ...

Чому буква 'а' маленька більше літери 'Я' великої?

alert( 'а' > 'Я' ); // true

Буква 'е' знаходиться в алфавіті між д і є: абвгдеєжз .... Але чому тоді 'е' більше 'я'?

alert( 'е' > 'я' ); // true

Щоб розібратися з цим, звернемося до внутрішнього уявлення рядків в JavaScript.

**Всі рядки мають внутрішню систему кодування Юнікод.**

Неважливо, якою мовою написана сторінка, чи знаходиться вона в windows-1251 або utf-8. Усередині JavaScript-інтерпретатора всі рядки приводяться до єдиного «Юнікодного» виду. Кожному символу відповідає свій код.

**String.fromCharCode (code)**

Повертає символ за кодом code:

alert( String.fromCharCode(1072) ); // 'а'

... І метод для отримання цифрового коду з символу:

**str.charCodeAt (pos)**

Повертає код символу на позиції pos. Відлік позиції починається з нуля.

alert( "абрикос".charCodeAt(0) ); // 1072, код 'а'

Тепер повернемося до прикладів вище. Чому порівняння 'е'>'я' і 'а'>'Я' дають такий дивний результат?

Справа в тому, що символи порівнюються не за алфавітом, а за кодом. У кого код більше - той і більше. У Юнікоді є багато різних символів. Кириличним буквах відповідає тільки невелика частина з них, докладніше - Кирилиця в Юнікоді.

Виведемо відрізок символів юникода з кодами від 1034 до 1113:

var str = '';

for (var i = 1034; i <= 1113; i++) {

str += String.fromCharCode(i);

}

alert( str );

Результат:

ЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюяѐёђѓєѕіїјљ

Ми можемо побачити з цього відрізку дві важливі речі:

**Малі літери йдуть після вживання великої, тому вони завжди більше.**

Зокрема 'а'(код 1072) > 'Я'(код 1071).

Те ж саме відбувається і в англійському алфавіті, там 'a' > 'Z'.

**Посимвольні порівняння**

Порівняння рядків працює лексикографічно, інакше кажучи, посимвольно.

Порівняння рядків s1 і s2 обробляється за наступним алгоритмом:

Порівнюються перші символи: s1[0] і s2[0]. Якщо вони різні, то порівнюємо їх і, в залежності від результату їх порівняння, повернеться true або false. Якщо ж вони однакові, то порівнюються другі символи s1[1] і s2[1]. Потім треті s1[2] і s2[2] і так далі, поки символом не будуть нарешті різними, і тоді який символ більше - той рядок і більше. Якщо ж в будь-якої рядку закінчилися символи, то вважаємо, що вона менше, а якщо закінчилися в обох - вони рівні.

Специфікація мови визначає цей алгоритм більш детально. Якщо ж говорити простими словами, сенс алгоритму в точності відповідає порядку, за яким імена заносяться в орфографічний словник.

"Вася" > "Ваня" // true, тому що початкові символи збігаються, а потім 'c'> 'н'

"Дома" > "До" // true, тому що початок збігається, але в 1-му рядку більше символів

**Висновки**

* Рядки в JavaScript мають внутрішню систему кодування Юнікод. При написанні рядка можна використовувати спеціальні символи, наприклад \n і вставляти Юнікодні символи за кодом.
* Ми познайомилися з властивістю length і методами charAt, toLowerCase/toUpperCase, substring/substr/slice (кращий slice). Є й інші методи, наприклад trim обрізає прогалини з початку і кінця рядка.
* Рядки порівнюються посимвольно. Тому якщо число отримано у вигляді рядка, то такі числа можуть порівнюватися некоректно, потрібно перетворити його до типу *number*.
* При порівнянні рядків слід мати на увазі, що букви порівнюються за їх кодами.