

Operators. Math. String. Boolean.
Null.
Undefined. Object

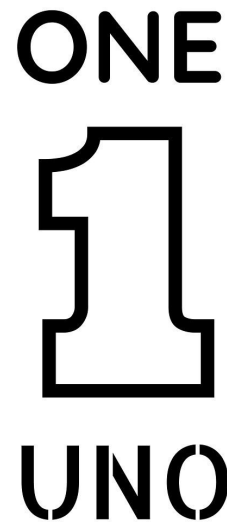
Зміст уроку

1. [Унарні оператори](#)
2. [Префіксний та постфіксний унарні оператори](#)
3. [Бінарні оператори](#)
4. [Присвоєння з операцією](#)
5. [Оператори відносини](#)
6. [Об'єкт Math](#)
7. [String](#)
8. [Методи рядків](#)
9. [Boolean](#)
10. [Логічні оператори](#)
11. [Null, Undefined](#)
12. [Object](#)

Унарні оператори

Унарні оператори в JavaScript - це оператори, які працюють лише з одним операндом.

Вони використовуються для зміни стану або значення цього єдиного операнда.



Унарний оператор мінус (-) - змінює знак числа на протилежний.

```
var c = 3  
console.log(-c) // Виведе: -3
```

Унарний оператор плюс (+) - спробує перетворити операнд на число, якщо це можливо (детально розглянемо далі).

```
var e = '5'  
console.log(+e) // Виведе: 5  
var f = 'hello'  
console.log(+f) // Виведе: NaN, оскільки "hello" не може бути перетворено на число
```

Оператор декрементації (- -) - автоматично зменшує значення змінної на одиницю.

```
var b = 5  
console.log(--b) // Виведе: 4
```

Оператор інкрементації (+ +) - автоматично збільшує значення змінної на одиницю.

```
var a = 5  
console.log(++a) // Виведе: 6
```

Оператор **typeof** - визначає тип даних операнда.

```
var d = true  
console.log(typeof d) // Виведе: "boolean"
```

Префіксний та постфіксний
унарні оператори

Префіксний інкремент та декремент

Префіксний інкремент (**`++varName`**) або декремент (**`--varName`**) збільшує або зменшує значення змінної на одиницю перед тим, як повернути значення. Це означає, що спочатку відбувається модифікація значення, а потім воно використовується у виразі.

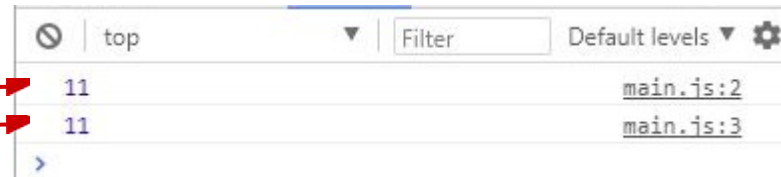
```
var num1 = 5  
console.log(++num1) // 6  
console.log(--num1) // 5
```

Постфіксний інкремент та декремент

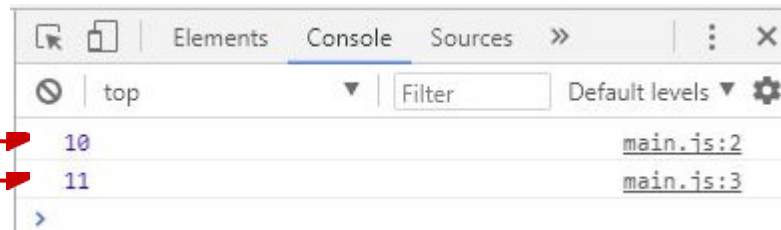
Постфіксний інкремент (**varName++**) або декремент (**varName--**) спочатку повертає поточне значення змінної, а потім збільшує або зменшує його на одиницю. Це означає, що використання значення відбувається до його модифікації.

```
var num2 = 5  
console.log(num2++) // 5  
console.log(num2) // 6  
console.log(num2--) // 6  
console.log(num2) // 5
```

```
var i = 10;  
console.log(++i);  
console.log(i);
```



```
var i = 10;  
console.log(i++);  
console.log(i);
```



Бінарні оператори

Бінарні оператори в JavaScript - це оператори, що вимагають два операнди для виконання операції .

Вони застосовуються для виконання арифметичних обчислень, порівняння значень, логічних операцій та для присвоєння значень.

Бінарні оператори дозволяють нам виконувати різноманітні операції між змінними та літералами, що є основою логіки багатьох програмних алгоритмів.

Арифметичні оператори (наприклад, **+**, **-**, *****, **/**) використовуються для виконання математичних обчислень між числами.

```
var sum = 10 + 5 // 15
```

```
var difference = 10 - 5 // 5
```

```
var product = 10 * 5 // 50
```

```
var quotient = 10 / 5 // 2
```

Оператори порівняння (наприклад, `==`, `!=`, `>`, `<`) використовуються для порівняння двох значень і повертають булеве значення (**true** або **false**) в залежності від результату порівняння.

```
var isEqual = 10 == 5 // false
var isNotEqual = 10 != 5 // true
var isGreater = 10 > 5 // true
var isLess = 10 < 5 // false
```

Логічні оператори (**&&**, **||**) використовуються для виконання логічних операцій між двома булевими значеннями або виразами.

```
var andOperation = true && false // false
```

```
var orOperation = true || false // true
```

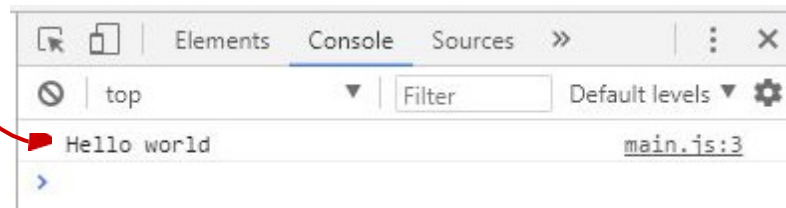

Оператор присвоєння (=) використовується для присвоєння значення правого операнда змінній, що є лівим операндом.

```
var number = 10 // Змінній number присвоєно значення 10
```

Додавання	$3 + 5$	<code>console.log(3 + 5);</code>	8
Віднімання	$8 - 4$	<code>console.log(8 - 4);</code>	4
Множення	$3 * 6$	<code>console.log(3 * 6);</code>	18
Поділ	$8 / 3$	<code>console.log(8 / 3);</code>	2.666665
Залишок від ділення	$8 \% 3$	<code>console.log(8 \% 3);</code>	2

Оператор **+** означає **конкатенацію** (склеювання), якщо одним із його операндів є рядок.

```
var string = 'Hello';  
console.log(string + ' world');
```



Присвоєння з операцією

Присвоєння з операцією в JavaScript є способом спрощення коду, який дозволяє нам комбінувати арифметичні, бітові або логічні операції з присвоєнням значення змінній.

Використовуючи ці оператори, ми можемо зменшити кількість необхідного коду для виконання операцій та присвоєння результату цих операцій змінним. Це робить код більш лаконічним та покращує його читабельність.

Оператор **присвоєння з додаванням** (**+=**) - додає правий операнд до лівого та присвоює результат лівому операнду.

```
var a = 10
```

```
a += 5 // Еквівалентно a = a + 5; Тепер a дорівнює 15
```

Оператор **присвоєння з відніманням** (**`-=`**) - віднімає правий операнд з лівого та присвоює результат лівому операнду.

```
var b = 10
```

```
b -= 5 // Еквівалентно b = b - 5; Тепер b дорівнює 5
```

Оператор **присвоєння з множенням** (**$\ast =$**) - множить лівий операнд на правий та присвоює результат лівому операнду.

```
var c = 10
```

```
c *= 5 // Еквівалентно c = c * 5; Тепер c дорівнює 50
```


Оператор **присвоєння з діленням** (**/=**) - ділить лівий операнд на правий та присвоює результат лівому операнду.

```
var d = 10
```

```
d /= 5 // Еквівалентно d = d / 5; Тепер d дорівнює 2
```

Оператор **присвоєння з остачею від ділення** (**$\% =$**) - знаходить остачу від ділення лівого операнда на правий та присвоює результат лівому операнду.

```
var e = 10
```

```
e %= 3 // Еквівалентно e = e % 3; Тепер e дорівнює 1
```

```
var number = 100
```

```
number = number + 20 // Збільшуємо number на 20, тепер number дорівнює 120
```

```
number = number - 20 // Зменшуємо number на 20, тепер number дорівнює 100
```

```
number = number * 2 // Множимо number на 2, тепер number дорівнює 200
```

```
number = number / 2 // Ділимо number на 2, тепер number дорівнює 100
```

```
number = number % 3 // Остача від ділення number на 3, тепер number дорівнює 1
```

```
var number = 100
```

```
// number = number + 20
```

```
number += 20 // Збільшуємо number на 20, тепер number дорівнює 120
```

```
// number = number - 20
```

```
number -= 20 // Зменшуємо number на 20, тепер number дорівнює 100
```

```
// number = number * 2
```

```
number *= 2 // Множимо number на 2, тепер number дорівнює 200
```

```
// number = number / 2
```

```
number /= 2 // Ділимо number на 2, тепер number дорівнює 100
```

```
// number = number % 3
```

```
number %= 3 // Знаходимо остачу від ділення number на 3, тепер number дорівнює 1
```



Менше символів і
красивіший код з
тим самим
результатом.

Оператори відносин

Оператори відносин у JavaScript використовуються для порівняння двох значень чи виразів, результатом чого є булеве значення: **true** (істина) або **false** (хибність).

Ці оператори дозволяють нам виконувати логічні порівняння, що є фундаментальною частиною умовних конструкцій та рішень, які приймаються під час виконання програми.

Основні оператори відносин

- | | |
|---------------------------------|---|
| == (рівно) | - перевіряє, чи рівні операнди за значенням. |
| != (не рівно) | - перевіряє, чи не рівні операнди за значенням. |
| === (строого рівно) | - перевіряє, чи рівні операнди за значенням та типом. |
| !== (строого не рівно) | - перевіряє, чи не рівні операнди за значенням та типом. |
| > (більше) | - перевіряє, чи лівий операнд більший за правий. |
| < (менше) | - перевіряє, чи лівий операнд менший за правий. |
| >= (більше або рівно) | - перевіряє, чи лівий операнд більший або рівний правому. |
| <= (менше або рівно) | - перевіряє, чи лівий операнд менший або рівний правому. |

Оператори відносини повертають значення логічного типу **true** або **false**: **false** – вираз хибно, **true** – вираз істинно.

7 > 9	console.log(7 > 9);	false
7 < 9	console.log(7 < 9);	true
9 >= 9	console.log(9 >= 9);	true
8 <= 10	console.log(8 <= 10);	true
10 === 10	console.log(10 === 10);	true
10 !== 10	console.log(10 !== 10);	false

- `=` оператор присвоєння.
- `==` оператор порівняння з наведенням
- `===` типів. оператор суворого порівняння.

Наприклад:

```
console.log(10 == '10');    // true  
console.log(10 === '10');  // false
```

На практиці краще не використовувати оператор порівняння на рівність з наведенням типів `==`, краще використовувати оператор порівняння `===`.

Об'єкт Math

Math у JavaScript - це вбудований об'єкт, що надає доступ до математичних констант і функцій.

Не потребує створення екземпляра, тому що всі властивості та методи **Math** статичні.

Властивості об'єкта **Math** містять **значення математичних констант**, що часто використовуються, наприклад:

- **Math.E**
- **Math.LN10**
- **Math.LN2**
- **Math.LOG10E**
- **Math.LOG2E**
- **Math.PI**
- **Math.SQRT1_2**
- **Math.SQRT2**

```
// Math.PI - число π. відношення довжини кола до його діаметру.
```

```
let circleRadius = 10;
```

```
let circumference = 2 * Math.PI * circleRadius;
```

```
console.log(circumference); // Обчислення довжини кола
```

```
// Math.E - основа натуральних логарифмів, приблизно дорівнює 2.718.
```

```
let n = Math.E;
```

```
console.log(n); // Виводить значення константи e
```

// Math.SQRT2 - квадратний корінь з 2, приблизно дорівнює 1.414.

```
let sqrtTwo = Math.SQRT2;
```

```
console.log(sqrtTwo); // Виводить квадратний корінь з 2
```

// Math.SQRT1_2 - квадратний корінь з 1/2, приблизно дорівнює 0.707.

```
let sqrtHalf = Math.SQRT1_2;
```

```
console.log(sqrtHalf); // Виводить квадратний корінь з ½
```

// Math.LN2 - натуральний логарифм числа 2, приблизно дорівнює 0.693.

```
let ln2 = Math.LN2;
```

```
console.log(ln2); // Виводить натуральний логарифм числа 2
```

// Math.LN10 - натуральний логарифм числа 10, приблизно дорівнює 2.303.

```
let ln10 = Math.LN10;
```

```
console.log(ln10); // Виводить натуральний логарифм числа 10
```

// Math.LOG2E - логарифм числа e за основою 2, приблизно дорівнює 1.442.

```
let log2e = Math.LOG2E;
```

```
console.log(log2e); // Виводить логарифм числа e за основою 2
```

// Math.LOG10E - логарифм числа e за основою 10, приблизно дорівнює 0.434.

```
let log10e = Math.LOG10E;
```

```
console.log(log10e); // Виводить логарифм числа e за основою 10
```

Math.round() у JavaScript - це метод, який округлює число до найближчого цілого.

```
let number = 5.49;
```

```
let rounded = Math.round(number);
```

```
console.log(rounded); // Виведе 5, оскільки дробова частина  
менше 0.5, і число заокруглюється вниз
```

```
number = 5.5;
```

```
rounded = Math.round(number);
```

```
console.log(rounded); // Виведе 6, оскільки дробова частина  
дорівнює 0.5 або більше, і число заокруглюється вгору
```


Math.floor() - це метод у JavaScript, який заокруглює число вниз до найближчого цілого.

```
let number = 5.95;
```

```
let roundedDown = Math.floor(number);
```

```
console.log(roundedDown); // Виведе 5, оскільки 5.95  
заокруглюється до найближчого меншого цілого числа
```

Math.ceil() - це метод, який заокруглює число вгору до найближчого цілого.

```
let number = 4.3;
```

```
let roundedUp = Math.ceil(number);
```

```
console.log(roundedUp); // Виведе 5, оскільки 4.3
```

заокруглюється вгору до найближчого цілого числа

Math.trunc() - це метод у JavaScript, який ми часто використовуємо у наших застосунках для видалення дробової частини числа.

```
let number = 6.84;
```

```
let truncatedNumber = Math.trunc(number);
```

```
console.log(truncatedNumber); // Виведе 6, оскільки дробова  
частина числа 6.84 видаляється
```

Math.floor() – округляє вниз

Math.ceil() – округляє вверх

Math.round() – округляє до найближчого
цілого

Math.trunc() – відкидає дробову частину

Number	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Math.random() - це метод у JavaScript, який ми часто використовуємо у наших застосунках для генерації випадкового числа між 0 (включно) та 1 (не включно).

```
let randomNum = Math.random();  
console.log(randomNum); // Виведе випадкове число між 0 та 1
```

```
let randomNum0to10 = Math.floor(Math.random() * 11);  
console.log(randomNum0to10); // Виведе випадкове число від 0 до 10
```

```
let randomNum0to100 = Math.floor(Math.random() * 101);  
console.log(randomNum0to100); // Виведе випадкове число від 0 до 100
```

Math.pow() - це метод у JavaScript, який ми використовуємо в наших застосунках для піднесення числа до певного ступеня.

```
let base = 3;
```

```
let exponent = 4;
```

```
let result = Math.pow(base, exponent);
```

```
console.log(result); // Виведе 81, оскільки 3 піднесене до  
4-го ступеня дорівнює 81
```

~~Math.pow~~ → **

У сучасному JavaScript краще використовувати оператор ****** для піднесення до степеня, оскільки він є простішим і читабельнішим варіантом у порівнянні з методом **Math.pow()**. Ось кілька причин:

1. Читабельність: Оператор ****** виглядає більш інтуїтивно та нагадує стандартну математичну нотацію, що полегшує читання коду.

```
let result = 2 ** 3 // Легко зрозуміти, що це 2 в степені 3
```

2. Сучасний стандарт: Оператор ****** було введено в ECMAScript 2016 (ES7), тому це більш сучасний синтаксис, який підтримується всіма сучасними браузерами.

3. Скорочений синтаксис: Використання оператора ****** дозволяє уникнути додаткових викликів функції **Math.pow()**, що спрощує код.

Методи об'єкту Math

- Math.abs()
- Math.acos()
- Math.acosh()
- Math.asin()
- Math.asinh()
- Math.atan()
- Math.atan2()
- Math.atanh()
- Math.cbrt()
- Math.ceil()
- Math.clz32()
- Math.cos()
- Math.cosh()
- Math.exp()
- Math.expm1()
- Math.floor()
- Math.fround()
- Math.hypot()
- Math.imul()
- Math.log()
- Math.log1p()
- Math.log10()
- Math.log2()
- Math.max()
- Math.min()
- Math.pow()
- Math.random()
- Math.round()
- Math.sign()
- Math.sin()
- Math.sinh()
- Math.sqrt()
- Math.tan()
- Math.tanh()
- Math.toSource()
- Math.trunc()

String

У JavaScript, **String** є примітивним типом даних, який використовується для представлення тексту.

Ми використовуємо рядки для зберігання та маніпуляції текстом, таким як імена користувачів, повідомлення, тексти на веб-сторінках та багато іншого.

Рядки можна створювати за допомогою **одинарних**, *подвійних* або **зворотних лапок**, що дозволяє включати в текст змінні та вирази.

Створення рядків

```
var greeting = 'Привіт, світе!'  
console.log(greeting) // Виведе: 'Привіт, світе!'
```

Об'єднання рядків

```
var firstName = 'Іван'  
var lastName = 'Іваненко'  
var fullName = firstName + ' ' + lastName  
console.log(fullName) // Виведе: 'Іван Іваненко'
```

Шаблонний літерал

Шаблонний літерал у JavaScript дозволяє створювати рядки, які можуть включати змінні та вирази, використовуючи зворотні лапки (```). Ми можемо вставляти значення всередину рядка за допомогою синтаксису **`${вираз}`** і таким чином робити рядки динамічними та зручними для читання.

Використання шаблонних літералів для вставки змінних та виразів

```
var firstName = 'Іван'  
var lastName = 'Іваненко'  
var fullName = firstName + ' ' + lastName  
var age = 25  
var introduction = `Мене звати ${fullName} і мені ${age} років.`  
console.log(introduction) // Виведе: 'Мене звати Іван Іваненко і мені 25 років.'
```

Зауважте, що для шаблонних літералів використовуються зворотні лапки.

Доступ до символів у рядку

```
var greeting = 'Привіт, світе!'
var letter = greeting[8] // Отримуємо дев'ятий символ з рядка greeting
console.log(letter) // Виведе: с
```

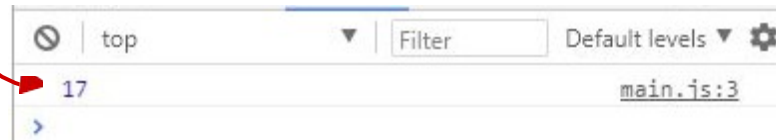
Довжина рядка

```
var greeting = 'Привіт, світе!'
```

```
var length = greeting.length
```

```
console.log(length) // 14 - кількість символів у рядку 'Привіт, світе!'
```

```
console.log('Some "new" string'.length)
```



Такий запис призведе до помилки:

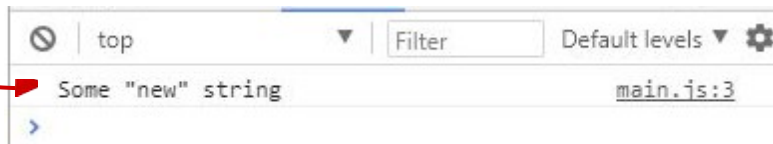
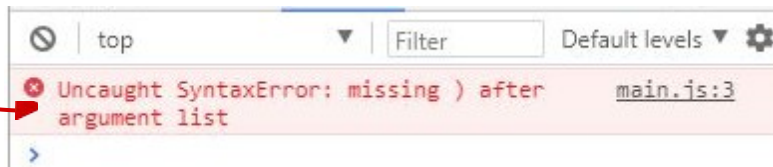
```
console.log("Some "new" string");
```

Подвійні лапки
- запис не за
кодстайлом

Рішення проблеми:

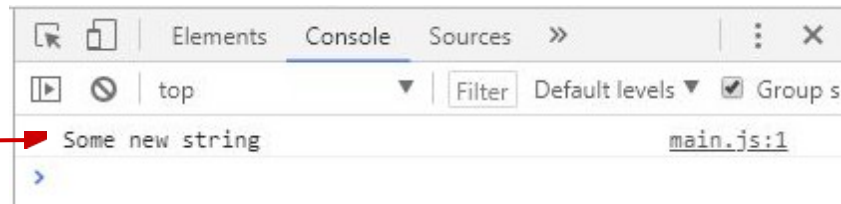
```
console.log('Some "new" string');
```

Про екранування (\) поговоримо пізніше.



Рядок можна записати на кількох рядках, для цього використовується символ `\` після якого не повинно бути нічого, в тому числі і символу пробіл, інакше може бути помилка:

```
console.log('Some \
new \
string');
```

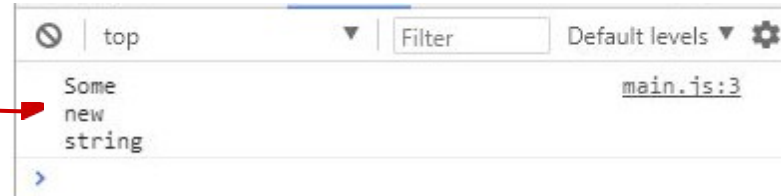


При цьому саме перенесення рядка не є частиною рядка.

Якщо ви хочете, щоб перенесення було частиною рядка, потрібно використовувати послідовність, що управляє. **\n**:

```
console.log('Some \nnew \nstring');
```

n – скорочення від **new line**, новий рядок.



Також є керуюча конструкція для табуляції :

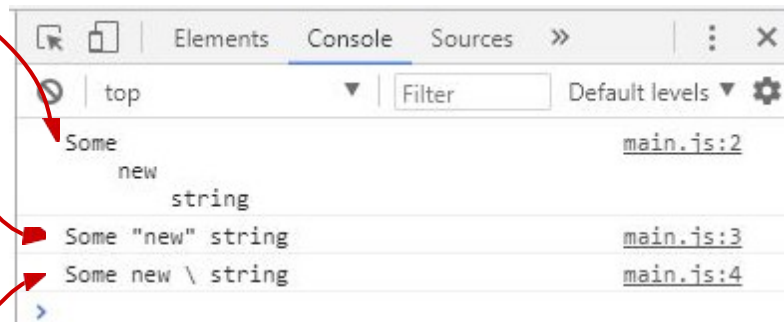
```
console.log('Some \n\tnew \n\t\tstring');
```

Для лапок – екранування:

```
console.log("Some \"new\" string");
```

Зворотний слеш \:

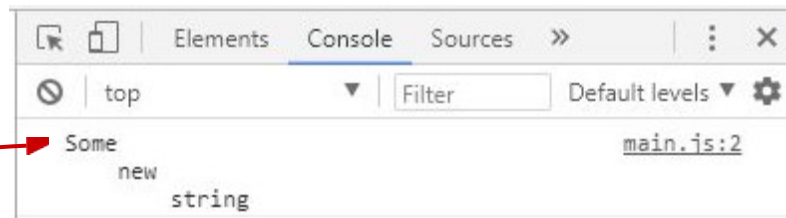
```
console.log("Some new \\ string");
```



Шаблонні рядки в JavaScript дозволяють створювати багаторядкові строки більш зручно, без необхідності використання символу зворотного слеша (`\`) на кінці кожного рядка.

Шаблонні рядки використовують зворотні лапки (```) для оголошення рядка, і всередині них можна використовувати реальні переноси рядків

```
console.log(`Some  
new  
string`);
```




Методи рядків

Методи рядків у JavaScript - це вбудовані функції , які ми використовуємо для маніпуляції рядками, наприклад, для пошуку підрядків, заміни фрагментів тексту, перетворення регістру та багато іншого.

Вони дозволяють нам з легкістю обробляти та модифікувати текстові дані.

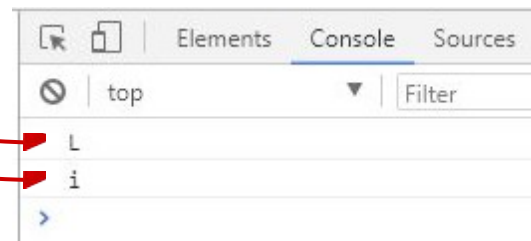
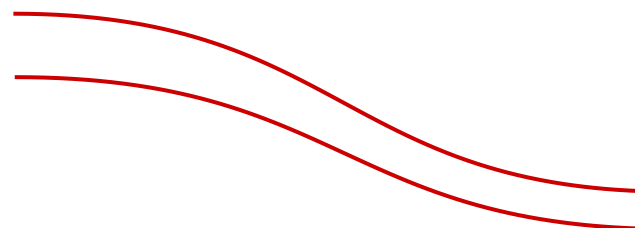
Метод **charAt()** повертає символ, який стоїть у рядку під певним індексом.



```
var string = 'Lorem ipsum dolor sit amet.';
console.log(string.charAt(0)); // L
console.log(string.charAt(6)); // i
```


Починаючи з **ECMAScript 5**, який підтримується всіма сучасними браузерами, замість методу **charAt()** можна використовувати роботу з рядком як з масивом.

```
var string = 'Lorem ipsum dolor sit amet,  
                consecetur adipiscing.';  
console.log(string[0]);  
console.log(string[6]);
```



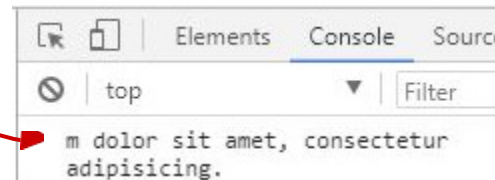
Властивість **length** повертає довжину рядка.

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipisicing.';  
console.log(string.length);    // 52  
console.log(string.charAt(string.length - 1));  
console.log(string[string.length - 1]);  
// останній символ у рядку
```

Метод **substring()** повертає підстроку вихідного рядка:

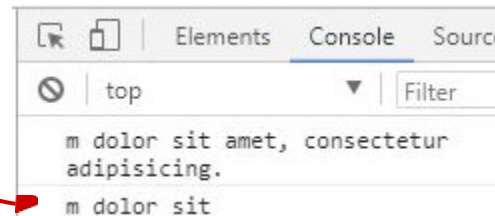
```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.substring(10));
```

Якщо вказано **один параметр** –
повертається підрядок із зазначено позиції
та до кінця вихідного рядка.



Якщо вказано два параметри, то повертається підрядок з **і до** вказано позиці :

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.substring(10, 21));
```



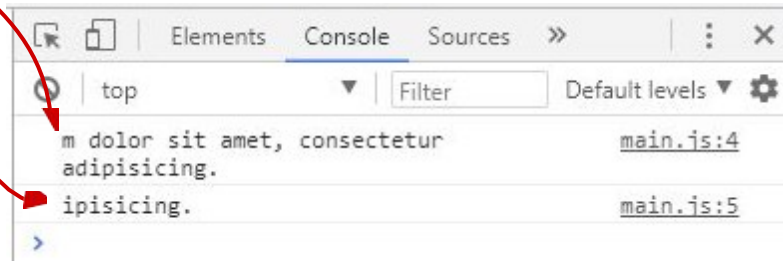
Коли ви використовуєте всі ці методи, потрібно пам'ятати, що **рядок** – **не змінюваний тип** (immutable type) у мові JavaScript і тому вони ніяк **не змінюють вихідний рядок**, вони просто **повертають новий рядок**.

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.substring(10, 21));  
console.log(string);
```



Метод `slice()` працює подібним чином, але здатний набувати негативних значень:

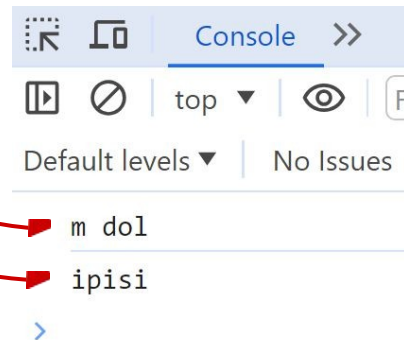
```
var string;  
string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.slice(10));  
  
string = 'Lorem ipsum dolor sit amet, consectetur  
adipiscing.';  
console.log(string.slice(-10));
```



```
var string;
```

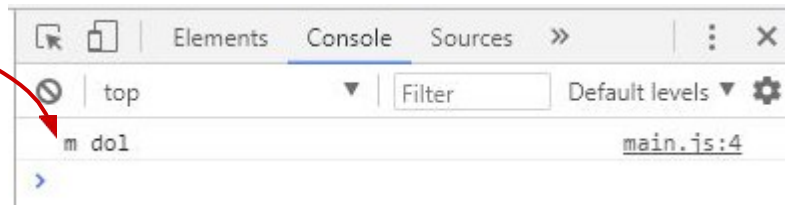
```
string = 'lorem ipsum dolor sit amet, consectetur adipiscing.' :  
console.log(string.slice(10, 15));
```

```
string = 'Lorem ipsum dolor sit amet, consectetur ad ipisicing.';  
console.log(string.slice(-10, -5));
```




Метод **substr()** дозволяє взяти певну кількість символів, починаючи із зазначеного:

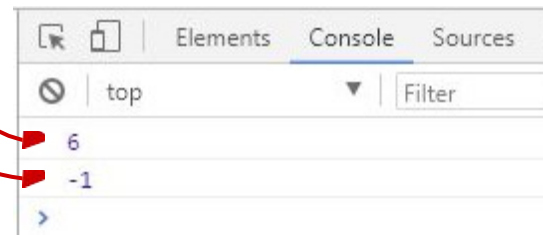
```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.substr(10, 5));
```



Для пошуку підрядки у рядку є метод **indexOf()**, який приймає підрядок і повертає позицію підрядка, або **-1** – якщо підрядок не знайдено.



```
var string = 'Lorem ipsum dolor sit amet, consectetur  
adipisicing.';  
console.log(string.indexOf('ips'));  
console.log(string.indexOf('not'));
```

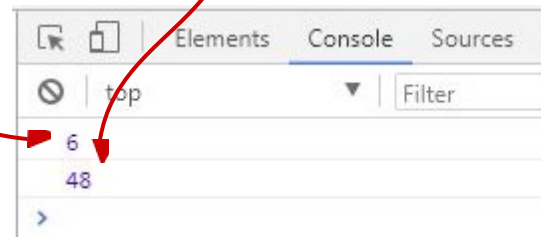
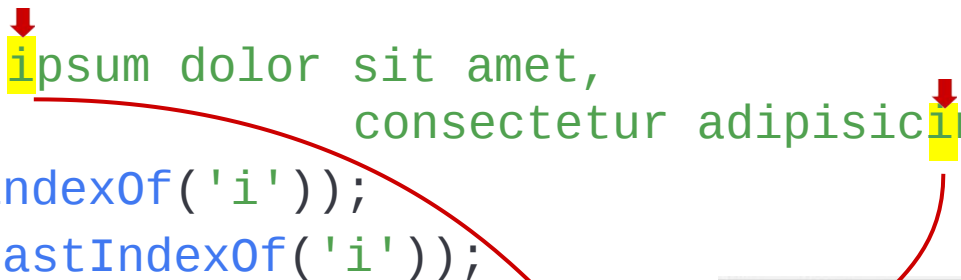


Метод **indexOf()** у JavaScript може приймати два параметри: перший — це підрядок, який потрібно знайти, а другий — позиція в рядку, з якої потрібно почати пошук.

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.indexOf('i', 10));  
// починає пошук з 10-ї позиції  
// поверне 19
```

Метод **lastIndexOf()** буде виконувати пошук підрядки з кінця.

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.indexOf('i'));  
console.log(string.lastIndexOf('i'));
```



Метод `replace()` дозволяє знайти та замінити підрядок.

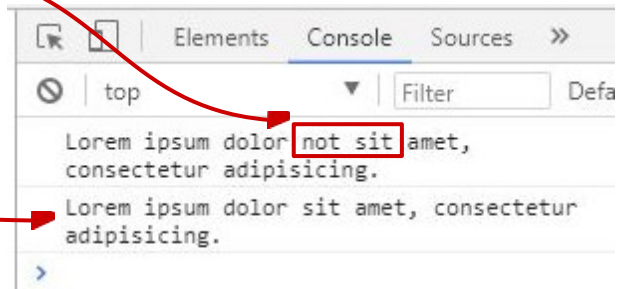
```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.replace('sit', 'not sit'));
```

При цьому вихідний рядок не змінюється:

```
console.log(string);
```

Щоб змінити значення, необхідно виконати:

```
string = string.replace('sit', 'not sit');
```



Метод **split()** розбиває рядок на масив, використовуючи роздільник.

```
var string = 'Lorem ipsum dolor sit amet,  
consectetur adipiscing.';  
console.log(string.split(' '));
```

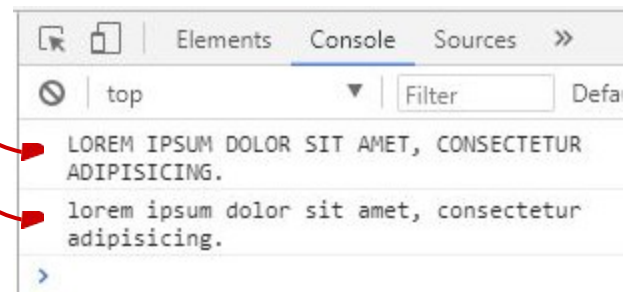
У цьому прикладі
використаний пробіл як
роздільник.

В отриманому масиві його елементами є слова.



Методи приведення до **верхнього регістру** – **toUpperCase()** і до **нижнього регістру** – **toLowerCase()**.

```
console.log(string.toUpperCase());  
console.log(string.toLowerCase());
```



Метод **includes()** використовується для перевірки, чи містить рядок певний підрядок. Метод includes() повертає true, якщо підрядок знайдено в рядку, і false в іншому випадку. Він чутливий до регістру, тому 'a' і 'A' розглядаються як різні символи.

```
var string = 'Lorem ipsum dolor sit amet.';
console.log(string.includes('ipsum')); // поверне true
console.log(string.includes('IPSUM')); // поверне false,
    оскільки 'IPSUM' відрізняється за регістром
console.log(string.includes('hello')); // поверне false,
    оскільки 'hello' не зустрічається в рядку
```

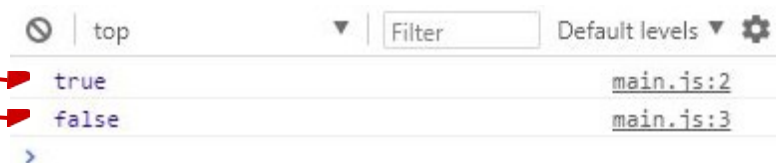
Boolean

Булевий чи **логічний** – це ще один простий тип даних у JavaScript. Логічний тип може набувати одне з двох значення: **істина** або **хибність**. Для запису в мові є зарезервовані слова: **true** і **false**.

Зазвичай логічні значення є результатом операції відносини, наприклад, порівняння на рівність:

```
console.log(5 === 5);
```

```
console.log(5 === 6);
```



Абсолютно будь-яке значення в мові JavaScript може бути перетворено на логічне.

Для цього перетворення використовується конструктор булевого типу **Boolean**.

```
console.log(Boolean(5));
```

Після перетворення справжнє значення прийматимуть усі значення, крім перерахованих: **undefined**, **null**, **0**, **NaN**, **""**:

```
console.log(Boolean(undefined));
```

```
console.log(Boolean(null));
```

```
console.log(Boolean(0));
```

```
console.log(Boolean(NaN));
```

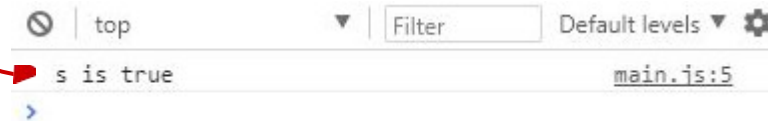
```
console.log(Boolean(' '));
```



Скрізь, де інтерпретатор JavaScript очікує отримати логічне значення, будь-які значення автоматично перетворюватимуться на логічні.

Наприклад:

```
var s = 'Some text';  
if (s) {  
    console.log('s is true');  
}
```



Логічні оператори

Логічні оператори в JavaScript використовуються для виконання операцій з логічними значеннями (**true** і **false**) та управління потоком виконання програми на основі умов.

Вони дозволяють нам комбінувати, інвертувати або перевіряти істинність виразів та умов.

Логічні оператори включають:

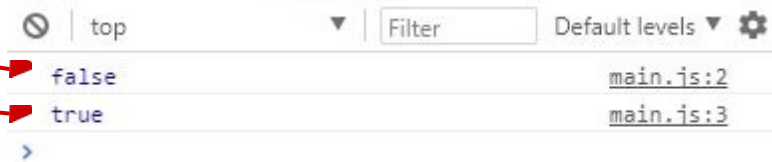
- **&&** (логічне "І") використовується для перевірки, чи істинні обидва операнди. Якщо обидва істинні, результат буде **true**; інакше - **false**.
- **||** (логічне "АБО") перевіряє, чи істинний хоча б один з операндів. Якщо хоча б один істинний, результат буде **true**.
- **!** (логічне НЕ) інвертує булеве значення: перетворює **true** в **false** та навпаки.
- **??** (оператор нульового злиття) повертає лівий операнд, якщо він **не null** і **не undefined**; інакше повертає правий.

&& (логічне "І")

Оператор логічного "І" записується за допомогою двох амперсандів **&&** і повертає істину тільки в тому випадку, коли обидва його операнди істинні.

```
console.log(true && false);
```

```
console.log(true && true);
```



|| (логічне "АБО")

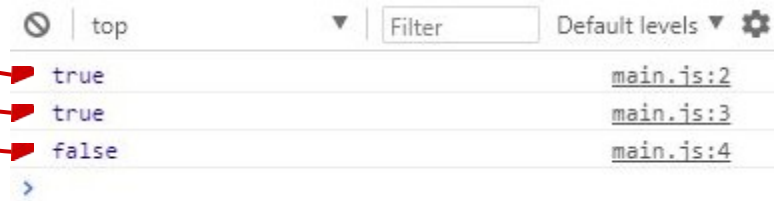
Оператор логічного "**Або**" записується за допомогою двох вертикальних ліній

|| і цей оператор повертає істину в тому випадку, коли хоча б один з операндів є істинним (**true**).

```
console.log(true || false);
```

```
console.log(true || true);
```

```
console.log(false || false);
```



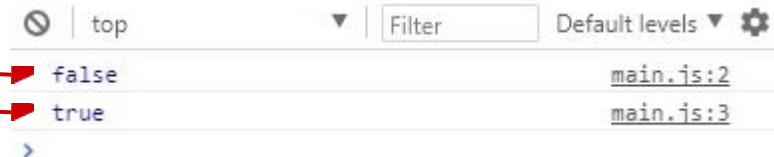
! (логічне НЕ)

Унарний **оператор логічного заперечення** записується за допомогою символу **!**

Якщо єдиний його операнд істинний (**true**) – він повертає хибне значення (**false**), і навпаки.

```
console.log(!true);
```

```
console.log(!false);
```



Для операторів **І (&&)** та **Або (||)** є цікаві способи застосування.

Наприклад:

вираз1 && вираз2

У випадку оператора логічного 'І' (&&), другий операнд виразу буде обчислений лише тоді, коли перший операнд є істинним (**true**).

Це означає, що якщо перший операнд хибний (**false**), то другий операнд ігнорується, оскільки весь вираз вже гарантовано буде хибним.

```
var a = 0,  
    b = 0,  
    isTrue = true,  
    isFalse = false;
```

```
isTrue && (a = 5);
```

```
console.log(a);
```

```
isFalse && (b = 5);
```

```
console.log(b);
```

top Filter Default levels

5	main.js:8
0	main.js:11

Наприклад:

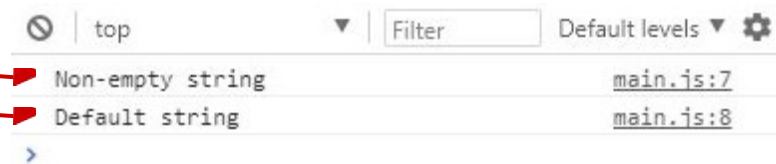
вираз1 || вираз2

З іншого боку, при використанні оператора логічного 'Або' (||), другий операнд перевіряється лише в тому випадку, якщо перший операнд є хибним (**false**).

Якщо перший операнд є істинним (**true**), другий операнд не враховується, бо весь вираз вже визначений як істинний."

```
var someString = 'Non-empty string',  
    emptyString = '',  
    newString1 = someString || 'Default string',  
    newString2 = emptyString || 'Default string';
```

```
console.log(newString1);  
console.log(newString2);
```



Оператор нульового злиття

Оператор **нульового злиття** – **??** це логічний оператор, який повертає значення правого операнда коли значення лівого операнда дорівнює **null** або **undefined**, інакше буде повернуто значення лівого операнда.

На відміну від **логічного Або (||)**, ліва частина оператора обчислюється та повертається навіть якщо його результат після приведення до логічного типу виявляється помилковим, але не є **null** або **undefined**.

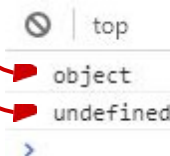
Null, Undefined

null та **undefined** – це два спеціальні типи даних у JavaScript, які означають відсутність значення.

null – це спеціальне значення, що означає 'нічого' або 'порожнє місце для значення'.

undefined – це спеціальне значення, що використовується, коли **змінна** була оголошена, але **не отримала жодного значення**, включаючи **null**.

```
console.log(typeof null);  
console.log(typeof undefined);
```



top
object
undefined
>

Цю проблему ми обговорювали на попередньому уроці

У JavaScript, **null** є спеціальним значенням, яке представляє відсутність будь-якого значення або об'єкта.

null використовується для:

- Явного вказівки на відсутність значення змінної або об'єкта.
- Очищення посилань на об'єкти, щоб допомогти зі зборкою сміття (габідж колекцією).
- Відмінності від **undefined**, яке вказує на те, що змінна була оголошена, але не ініціалізована жодним значенням.

Ситуації , коли зустрічається undefined

Значення неініціалізованої змінної:

```
var temp;  
console.log(temp);           // undefined
```

1

Звернення до неіснуючої властивості об'єкта:

```
var obj = {};  
console.log(obj.property);  // undefined
```

2

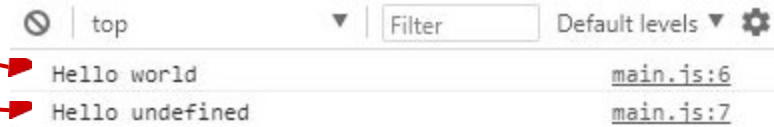
Звернення до неіснуючого елементу масиву:

```
var a = [1, 2, 3, 4];  
console.log(a[4]);          // undefined
```

3

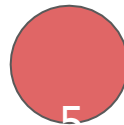
```
function test(string)
{ return 'Hello ' +
      string;
}
console.log(test('world'));
console.log(test());
```

4



Якщо функція не повертає будь-яке значення, значення буде **undefined**:

```
function empty(string) {}  
console.log(empty());
```



При порівнянні на рівність з приведенням типів **null** і **undefined** рівні:

```
console.log(null == undefined); // true
```

При звичайному порівнянні вони не рівні, оскільки мають різні типи:

```
console.log(null === undefined); // false
```

positive value



1



0



negative value



Infinity



NaN



null



undefined



Object

Об'єкт у JavaScript - це колекція властивостей, які представляють пари ключ- значення.

Кожна властивість об'єкта може бути як **значенням даних**, так і **функцією**, що в такому випадку називається методом об'єкта.

Об'єкти в JavaScript є гнучкими і можуть зберігати різноманітні типи даних, забезпечуючи потужний спосіб організації функціоналу та даних.

```
obj = {  
    key1: 123,           // властивість  
    key2: 'string',     // властивість  
    key3: function () {}, // метод  
    key4: function () {} // метод  
};
```

У загальному
вигляді

об'єкт виглядає так:

{

ключ: значення,

ключ: значення,

ключ: значення

}

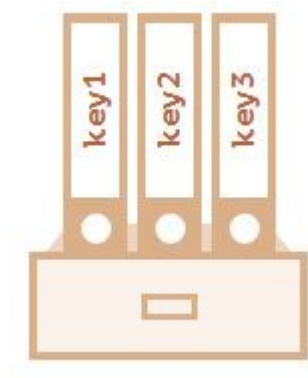
Об'єкт – це набір
властивостей, представлений
парою

ключ: значення.

ключ: значення поділяється
двокрапкою і перераховується
через кому.
Таких пар може бути скільки
завгодно.

Ми можемо уявити об'єкт як коробку з підписаними теками.

Кожен елемент даних зберігається у своїй теці, на якій написаний ключ.
За ключем легко знайти теку, видалити або додати до неї щось нове.



Порожній об'єкт ("порожній ящик")

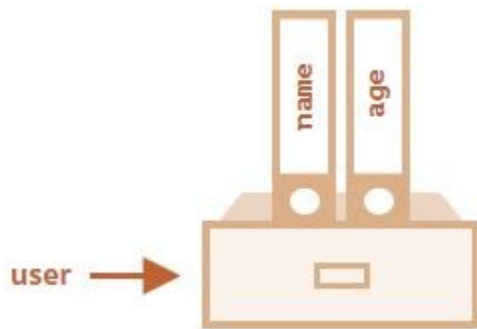
```
var user = {}; // синтаксис "літерал об'єкта"
```



Користуючись літеральним синтаксисом `{...}`, ми одразу можемо розмістити в об'єкті кілька властивостей у вигляді пар **ключ :**

значення:

```
var user = {  
  // об'єкт  
  name: 'John',  
  age: 30  
};
```



Об'єктний літерал

Наприклад:

```
{  
  name: 'John',  
  age: 25,  
  gender: 'male'  
}
```

Таким чином, ми записали об'єкт – це людина. Те, що ми написали, з погляду синтаксису називається **об'єктним літералом**.

Ми можемо привласнити об'єкт змінно та звертатися до окремих властивостей цього об'єкту..

```
var person =  
    { name:  
      'John',  
    age: 25,  
    gender: 'male'  
  }                                     // Jhon  
console.log(person.name);
```

Властивості об'єкта можна також називати полями.

Для звернення до полів об'єктів ми використовуємо вираз, який так і називається – **виразом звернення** або **вираз доступу** (property access expression) і має два синтаксиси в мові JavaScript.

Перший випадок ми використали раніше, його синтаксис такий:

вираз.ідентифікатор – **dot notation**

У другому випадку синтаксис наступний:

вираз[вираз] – **bracket notation**


Для звернення до полів можна використовувати будь-який варіант, але вони мають відмінності.

Завдяки тому, що у квадратні дужки ми можемо підставляти будь-який вираз, отже і формувати цю властивість можна динамічно.

```
console.log(person[ 'name' ] );
```

За допомогою цих же виразів ми можемо додавати та змінювати властивості об'єкта.

```
var person =  
    { name:  
      'Jhon',  
      age: 25,  
      gender: 'male'  
    };  
person.age = 30;  
person.userID = 100;  
console.log(person);
```



```
▶ {name: "Jhon", age: 30, gender: "male", userID: 100}
```

Як бачимо, властивість
можна додавати на льоту
під час виконання
скрипту.

Значенням будь-якої властивості може бути функція і таку властивість називають **методом**.

```
var person =  
    { name:  
      'Jhon',  
    age: 25,  
    gender: 'male',  
    sayHi: function() {  
        return 'Hello world!';  
    }; }  
console.log(person.sayHi());
```



new Object()

Об'єктний літерал – не єдиний спосіб створення об'єктів.

Інший спосіб полягає у використанні функції-конструктора **Object** та оператора **new**.

```
var person = new Object();  
person.property = 'value';
```

Як бачимо, з об'єктним літералом працювати з об'єктом простіше і синтаксис виходить коротшим.

create()

Ще один спосіб створення об'єктів – статичний метод **create()** класу **Object**, який приймає першим параметром об'єкт, що є прототипом нового об'єкта:

```
var person = Object.create(null);  
person.property = 'value';
```

Якщо ми **не хочемо**, щоб об'єкт успадковував якісь властивості, то можемо передати як параметр – **null**.

Тепер спробуємо передати якийсь об'єкт:

```
var person = Object.create({x: 10, y: 20});  
person.property = 'value';
```

```
console.log(person.x);
```

```
console.log(person.y);
```

```
console.log(person);
```

```
console.log(person.hasOwnProperty('x'));
```

```
console.log(person.hasOwnProperty('y'));
```



Ми бачимо у нового об'єкта властивості, але це не його власні властивості, а успадковані властивості від прототипу, в чому легко можемо переконатися.

Але якщо ми додамо таку властивість до об'єкту:

```
var person = Object.create({x: 10, y: 20});  
person.x = 15;
```

```
console.log(person);  
console.log(person.hasOwnProperty('x'));
```



Тепер метод **hasOwnProperty()** поверне **true** і ми побачимо рідну властивість об'єкта, а не успадковане.

delete

Для видалення властивостей об'єкта існує оператор **delete**. Це звичайний унарний оператор.

Синтаксис:

delete вираз

Оператором **delete** можна видалити лише рідні властивості об'єкта. Успадковані властивості можна видалити лише у прототипу.

Наприклад:

```
var person = Object.create({x: 10, y: 20});  
person.x = 15;
```

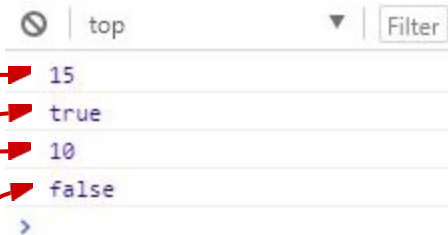
```
console.log(person.x);
```

```
console.log(person.hasOwnProperty('x'));
```

```
delete person.x;
```

```
console.log(person.x);
```

```
console.log(person.hasOwnProperty('x'));
```



in

Для перевірки наявності якості в об'єкті є бінарний оператор **in**.

```
var person = Object.create({x: 10, y: 20});
```

```
console.log('x' in person);
```

```
console.log('any' in person);
```



Цьому оператору все одно, успадкована властивість чи рідна.

Object.freeze()

Метод **Object.freeze()** в JavaScript використовується для заморожування об'єкта, що перешкоджає додавання нових властивостей до об'єкта, видалення старих властивостей з об'єкта, а також зміну значень існуючих властивостей або їхню конфігурацію. Заморожені об'єкти також є незмінними, тобто значення їх властивостей не можна змінити.

```
const obj = { prop: 42 };
```

```
Object.freeze(obj);
```

```
obj.prop = 33; // TypeError (в режимі суворості)
```

```
console.log(obj.prop); // Очікуваний вивід: 42
```

Object.assign()

Метод **Object.assign()** в JavaScript використовується для копіювання значень всіх власних перелічуваних властивостей з одного або більше об'єктів джерела в цільовий об'єкт. Він повертає цільовий об'єкт.

```
const target = { a: 1, b: 2 };
```

```
const source = { b: 4, c: 5 };
```

```
Object.assign(target, source);
```

```
console.log(target); // Object { a: 1, b: 4, c: 5 }
```

Глибоке копіювання об'єкта

Для глибокого копіювання об'єктів в JavaScript, ви можете використовувати метод **JSON.parse()** разом з **JSON.stringify()**. Цей підхід працює шляхом перетворення цільового об'єкта в рядок **JSON** за допомогою **JSON.stringify()**, а потім перетворення цього рядка назад в новий об'єкт за допомогою **JSON.parse()**.

```
const obj = { a: 1, b: { c: 2 } };  
const deepCopy = JSON.parse(JSON.stringify(obj));  
console.log(deepCopy); // Очікуваний вивід: { a: 1, b: { c: 2 } }  
deepCopy.b.c = 3;  
console.log(obj.b.c); // Очікуваний вивід: 2
```