

Type Conversion. Conditional
Statements. Conditional
Operator.
Loops. Functions. Callback

Зміст уроку

1. [Перетворення типів](#)
2. [Управління потоком виконання](#)
3. [Умовний \(Тернарний\) оператор](#)
4. [Оператор кома](#)
5. [Цикли](#)
6. [Функції](#)
7. [Callback функції](#)

Quiz

1. Який оператор використовується для збільшення значення змінної на одиницю?
 - a. --
 - b. ++
 - c. ==
 - d. !=
2. Який оператор повертає `true`, якщо обидва операнди не рівні один одному?
 - a. ===
 - b. <=
 - c. !=
 - d. >=
3. Яка функція об'єкта `Math` використовується для обчислення найбільшого значення з набору чисел?
 - a. `Math.min()`
 - b. `Math.max()`
 - c. `Math.abs()`
 - d. `Math.sqrt()`
4. Який метод рядка в JavaScript використовується для перетворення всіх символів рядка на великі літери?
 - a. `toLowerCase()`
 - b. `toUpperCase()`
 - c. `split()`
 - d. `substring()`
5. Який логічний оператор представляє логічне "І"?
 - a. `&&`
 - b. `||`
 - c. `!`
 - d. `??`
6. Яке значення JavaScript використовується для представлення відсутності будь-якого значення?
 - a. `0`
 - b. `NaN`
 - c. `null`
 - d. `undefined`
7. Що поверне вираз `typeof undefined`?
 - a. `"null"`
 - b. `"undefined"`
 - c. `"object"`
 - d. `"boolean"`
8. Який метод використовується для розбиття рядка на масив підрядків?
 - a. `join()`
 - b. `split()`
 - c. `slice()`
 - d. `splice()`
9. Який оператор використовується для строгого порівняння без перетворення типу?
 - a. `=`
 - b. `==`
 - c. `===`
 - d. `!=`
10. Яка функція повертає індекс першого входження підрядка в рядок, або `-1`, якщо підрядок не знайдено?
 - a. `indexOf()`
 - b. `search()`
 - c. `findIndex()`
 - d. `locate()`

Перетворення типів

Перетворення типів у JavaScript - це процес зміни типу даних з одного у інший, наприклад, з рядка в число або з числа в булеве значення.

Ми використовуємо перетворення типів для обробки та маніпуляції даними різних типів, щоб вони відповідали очікуваному формату операцій або функцій.

Явне перетворення типів

З використанням конструкторів: JavaScript надає вбудовані конструктори, такі як **Number()**, **String()**, і **Boolean()**, які можна використовувати для явного перетворення типів.

```
var strToNum = Number("456"); // "456" стає числом 456
```

```
var boolToStr = String(true); // true стає рядком "true"
```

```
var numToBool = Boolean(0); // 0 стає булевим false, оскільки 0 вважається "falsy" значенням
```

За допомогою метода **toString() та функцій **parseInt()**, **parseFloat()**:**

```
var num = 12345.6789
```

```
var numToStr = num.toString() // Конвертує число в рядок "12345.6789"
```

```
var strToInt = parseInt('100px') // Конвертує рядок в ціле число 100, ігноруючи нечислові символи
```

```
var strToFloat = parseFloat('3.14someText') // Конвертує рядок в число з плаваючою комою 3.14
```

Неявне перетворення типів

JavaScript також виконує неявне перетворення типів, коли це необхідно для операцій.

```
var result = '3' + 2 // JavaScript перетворює число 2 в рядок і конкатенує його з "3". результат "32"
```

```
var comparison = '5' == 5 // JavaScript перетворює рядок "5" в число 5 перед порівнянням, результат true
```


Швидке перетворення типів

Для швидкого перетворення типів існують короткі вирази:

```
var strToNumQuick = +'42' // Швидке перетворення рядка "42" в число за  
допомогою унарного оператора +
```

```
var numToStrQuick = 123 + '' // Швидке перетворення числа 123 в рядок за  
допомогою конкатенації з порожнім рядком
```

```
var boolToNum = +true // Швидке перетворення булевого true в число 1
```

Перетворення та порівняння

Під час порівняння значень різних типів JavaScript може виконувати неявне перетворення типів, що може призвести до неочікуваних результатів.

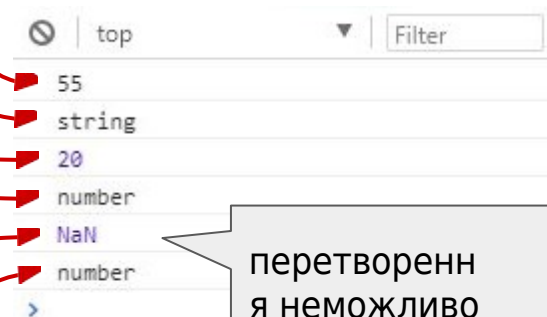
```
var compare = 0 == false // true, оскільки 0 і false є "falsy" значеннями
```

Автоматична конвертація типів:

```
console.log(5 + '5');  
console.log(typeof(5 + '5'));
```

```
console.log(5 * '4');  
console.log(typeof(5 * '4'));
```

```
console.log(5 * 'text');  
console.log(typeof(5 * 'text'));
```



Оператор порівняння з наведенням типів працює дуже неоднозначно, а іноді – непередбачувано:

```
console.log('5' == 5);  
console.log('0' == false);  
console.log(0 == false);  
console.log('5' == true);  
console.log('' == false);  
console.log(null == false);  
console.log(null == true);  
console.log(undefined == false);  
console.log(undefined == true);  
console.log(undefined == null);
```

Code	Result	File
<code>console.log('5' == 5);</code>	true	main.js:2
<code>console.log('0' == false);</code>	true	main.js:3
<code>console.log(0 == false);</code>	true	main.js:4
<code>console.log('5' == true);</code>	false	main.js:5
<code>console.log('' == false);</code>	true	main.js:6
<code>console.log(null == false);</code>	false	main.js:7
<code>console.log(null == true);</code>	false	main.js:8
<code>console.log(undefined == false);</code>	false	main.js:9
<code>console.log(undefined == true);</code>	false	main.js:10
<code>console.log(undefined == null);</code>	true	main.js:11

На практиці ми завжди будемо використовувати оператор суворо порівняння `===`.

За допомогою конструктора **Number ()** можна перевести будь-яке значення
в

ЧИСЛОВИЙ ТИП:

```
console.log(Number('12345'))           // 12345  
console.log(typeof Number('12345'))    // number
```

За допомогою конструктора **String()** можна перевести будь-яке значення в

рядковий тип:

```
console.log(String(12345))           // '12345'  
console.log(typeof String(12345))    // string
```

Для переведення в **логічний тип** можна використовувати конструктор

Boolean():

```
console.log(Boolean(1))           // true
console.log(typeof Boolean(1))    // boolean
```

Для перетворення типів є короткі форми запису.

Наприклад, для перетворення числа на логічний тип можна використовувати такий запис:

```
console.log(!!5) // true  
console.log(!!0) // false
```

! – простий **оператор логічного заперечення**.

Одного оператора логічного заперечення вже достатньо, щоб перевести число в логічний тип, але ми отримаємо інвертоване значення, тому щоб отримати правильне значення – використовується другий оператор.

Швидко перетворити значення в рядок можна таким чином:

```
console.log(12345 + '') // '12345'  
console.log(typeof (12345 + '')) // string
```

Швидке перетворення рядка в число виглядає так:

```
console.log(+ '12345') // 12345  
console.log(typeof + '12345') // number
```

Конвертація до рядка	Конвертація до числа	Конвертація до логічного значення
<code>String(123); // "123"</code>	<code>Number('123'); // 123</code>	<code>Boolean(''); // false</code>
<code>String(-12.3); // "-12.3"</code>	<code>Number('123.4'); // 123.4</code>	<code>Boolean('string'); // true</code>
<code>String(null); // "null"</code>	<code>Number('123,4'); // NaN</code>	<code>Boolean('false'); // true</code>
<code>String(undefined); // "undefined"</code>	<code>Number(''); // 0</code>	<code>Boolean(0); // false</code>
<code>String(true); // "true"</code>	<code>Number(null); // 0</code>	<code>Boolean(42); // true</code>
<code>String(false); // "false"</code>	<code>Number(undefined); // NaN</code>	<code>Boolean(-42); // true</code>
<code>String(function(){}); // "function() {}"</code>	<code>Number(true); // 1</code>	<code>Boolean(NaN); // false</code>
<code>String({}); // "[object Object]"</code>	<code>Number(false); // 0</code>	<code>Boolean(null); // false</code>
<code>String({ key: 42 }); // "[object Object]"</code>	<code>Number(function () {}); // NaN</code>	<code>Boolean(undefined); // false</code>
<code>String([]); // ""</code>	<code>Number({}); // NaN</code>	<code>Boolean(function () {}); // true</code>
<code>String([1, 2]); // "1,2"</code>	<code>Number([1, 2]); // NaN</code>	<code>Boolean([1, 2]); // true</code>

Конвертація до рядка	Конвертація до числа	Конвертація до логічного значення
<code>123 + ''; // "123"</code>	<code>+ '123'; // 123</code>	<code>!!''; // false</code>
<code>-12.3 + ''; // "-12.3"</code>	<code>+ '123.4'; // 123.4</code>	<code>!!'string'; // true</code>
<code>null + ''; // "null"</code>	<code>+ '123,4'; // NaN</code>	<code>!!'false'; // true</code>
<code>undefined + ''; // "undefined"</code>	<code>+ ''; // 0</code>	<code>!!0; // false</code>
<code>true + ''; // "true"</code>	<code>+null; // 0</code>	<code>!!42; // true</code>
<code>false + ''; // "false"</code>	<code>+undefined; // NaN</code>	<code>!!-42; // true</code>
<code>(function () {}) + ''; // "function() {}"</code>	<code>+true; // 1</code>	<code>!!NaN; // false</code>
<code>({}) + ''; // "[object Object]"</code>	<code>+false; // 0</code>	<code>!!null; // false</code>
<code>({ key: 42 }) + ''; // "[object Object]"</code>	<code>+function () {}; // NaN</code>	<code>!!undefined; // false</code>
<code>[] + ''; // ""</code>	<code>+{}; // NaN</code>	<code>!!function () {}; // true</code>
<code>[1, 2] + ''; // "1,2"</code>	<code>+ [1, 2]; // NaN</code>	<code>!![1, 2]; // true</code>

Ще один спосіб перетворення числа в рядок – використання методу

toString():

```
var number = 255
```

```
console.log(number.toString()) // '255'
```

```
console.log(typeof number.toString()) // string
```

Як параметр метод приймає основу системи числення:

```
console.log(number.toString(2)) // '11111111'
```

```
console.log(number.toString(16)) // 'ff'
```

```
console.log(number.toString(10)) // '255'
```

Цей метод також працює з логічними та іншими типами даних:

```
console.log(false.toString())           // 'false'  
console.log(typeof false.toString()) // string
```

Для перетворення рядка на число є дві глобальні функції :

parseInt() і **parseFloat()**.

```
console.log(parseInt('255')) // 255
```

```
console.log(parseInt('255.12345')) // 255
```

```
console.log(parseFloat('123.4567')) // 123.4567
```

Як другий параметр функції приймають основу системи числення. Це необов'язковий параметр, але його вказівка є гарною звичкою.

```
console.log(parseInt('255', 10)) // 255
console.log(parseInt('0x1F', 16)) // 31
console.log(parseInt('111', 2)) // 7
console.log(parseInt('10.265', 10)) // 10
console.log(parseFloat('10.265')) // 10.265
```

При використанні даних функцій рядок крім цифр може містити будь-які символи, які ігноруватимуться.

```
console.log(parseInt('14px', 10)) // 14  
console.log(parseInt('3.75rem')) // 3  
console.log(parseFloat('1.25rem')) // 1.25
```


При перетворенні на рядок наступних значень:

```
console.log(String(Infinity))           // 'Infinity'  
console.log(typeof String(Infinity))    // string
```

```
console.log(String(NaN))                 // 'NaN'  
console.log(typeof String(NaN))          // string
```

ми отримаємо відповідні
рядки.

Будь-які значення крім перших п'яти при перетворенні на логічний тип повернуть істину:

```
console.log(!!'')           // empty string    → false
console.log(!!NaN)          // NaN              → false
console.log(!!0)            // число 0          → false
console.log(!!null)         // null             → false
console.log(!!undefined)    // undefined        → false
```

```
console.log(!!'123')        // non-empty string → true
console.log(!!123)          // число, не 0      → true
console.log(!!obj)          // об'єкт             → true
```

При перетворенні рядка на число в ньому (рядку) може бути будь-яка кількість порожніх символів, які будуть проігноровані:

```
console.log(+' 4') // 4
console.log(+'   4') // 4
console.log(+' 4 ') // 4
console.log(+' 0xF ') // 15
```

але в рядку не може бути інших символів:

```
console.log(+' f 4 ') // NaN
console.log(+' 4 g ') // NaN
```

Якщо все ж таки потрібно перетворити на число рядок, що містить крім цифр і пробілів інші символи, потрібно використовувати функції

parseInt() або **parseFloat()**:

```
console.log(parseInt(' 4 f————')) // 4
```

```
console.log(parseInt(' f 4 ')) // NaN
```

```
console.log(parseFloat(' 4 g————')) // 4
```

```
console.log(parseFloat(' g 4 ')) // NaN
```

Булеві значення також можна перетворювати на числа і отримати при цьому

0 або **1**:

```
console.log(+true) // 1
```

```
console.log(+false) // 0
```

Той факт, що JavaScript легко і автоматично перетворює значення з одного типу в інший, наприклад при використанні якихось операторів - вважається мінусом мови, оскільки це може призвести до помилок, які важко спіймати.

В даному випадку допоможе **хороше розуміння роботи перетворення типів JavaScript і відмова від оператора рівності з перетворенням типів.**

Управління потоком виконання

Управління потоком виконання програми в JavaScript здійснюється за допомогою умовних конструкцій, таких як **if**, **if...else**, **if...else if...else**, та **switch...case**.

Ці конструкції дозволяють програмі виконувати різні блоки коду в залежності від заданих умов.

if {}

Конструкція **if** використовується для виконання коду, якщо задана умова істинна.

```
if (condition) {  
    // Код, що виконується, якщо умова істинна  
}
```

Приклад

```
var age = 20  
if (age >= 18) {  
    console.log('Ви можете голосувати.')}
```

if ... else

if else додає альтернативну дію, яка виконується, якщо умова хибна.

```
if (condition) {  
    // Кол. що виконується, якщо умова істинна  
} else {  
    // Кол. що виконується, якщо умова хибна  
}
```

Приклад

```
var score = 75
if (score >= 50) {
  console.log('Ви прошли тест.')
} else {
  console.log('Ви не прошли тест.')
}
```

if ... else if ... else

Цей варіант використовується для створення кількох умовних гілок.

```
if (condition1) {  
    // Кол. що виконується, якщо condition1 істинна  
} else if (condition2) {  
    // Кол. що виконується, якщо condition2 істинна  
} else {  
    // Кол. що виконується, якщо обидві умови хибні  
}
```

Приклад

```
var score = 85
if (score >= 90) {
  console.log('Відмінно')
} else if (score >= 75) {
  console.log('Добре')
} else {
  console.log('Задовільно')
}
```

Використовуючи кілька інструкцій **else if** можна, перебирати значення змінної, виконуючи певні інструкції :

```
var cityName = 'Bezlyudivka',
    status
if (cityName === 'Kharkiv')
  { status = 'City'
} else if (cityName === 'Andriivka')
  { status = 'Village'
} else if (cityName === 'Bezlyudivka')
  { status = 'Township'
}
console.log(cityName + ' ' + status) // Bezlyudivka Township
```

switch ... case

switch використовується для виконання різних дій на основі різних умов.

```
switch (expression)
{
    case value1:
        // Виконується, якщо expression дорівнює value1
        break
    case value2:
        // Виконується, якщо expression дорівнює value2
        break
    default:
        // Виконується, якщо жоден з вищезазначених випадків не відбувся
}
```


Приклад

```
var day = 2
switch (day) {
  case 1:
    console.log('Понеділок')
    break
  case 2:
    console.log('Вівторок')
    break
  case 3:
    console.log('Середа')
    break
  default:
    console.log('Інший день тижня')
}
```

Перепишемо попередній приклад **if** за допомогою оператора

```
case  
var cityName = 'Kharkiv',  
    status;
```

```
switch (cityName) {  
    case 'Kharkiv':  
        status = 'City';  
    case 'Andriivka':  
        status = 'Village';  
    case 'Bezlyudivka':  
        status = 'Township';  
}
```

Неправильне
рішення...

Вивелося не те, що було потрібне. Щоб цього уникнути, наприкінці кожного **case** крім останнього ми будемо використовувати інструкцію **break** (інструкція миттєво виходу).

```
console.log(cityName + ' ' + status); // 'Kharkiv Township'
```

```
var cityName = 'Nyrkiv',
    status

switch (cityName) {
  case 'Kharkiv':
    status = 'City'
    break
  case 'Andriivka':
    status = 'Village'
    break
  case 'Bezlyudivka':
    status = 'Township'
    break
  default:
    status = 'somewhere in Ukraine'
}
```

Правильне
рішення...

Ще найкраще
рішення...

Відмінне
рішення!

```
console.log(cityName + ' ' + status) // Nyrkiv somewhere in UA
```

Умовний (Тернарний) оператор

?

Умовний (тернарний) оператор у JavaScript є компактною альтернативою стандартній умовній конструкції **if...else**.

Він названий "тернарним", оскільки **включає три частини: умову, вираз, що виконується, якщо умова істинна, і вираз, що виконується, якщо умова хибна.**

Синтаксис тернарного оператора виглядає так:

умова ? виразЯкщо : виразІнікше

Параметри:

умова – вираз, що набуває значення **true** чи **false**.

виразЯкщо, виразІнікше – вирази, значення яких можуть належати будь-якому типу.

Перевіряється **умова**, потім якщо вона вірна – повертається значення **виразЯкщо**, якщо неправильно – значення **виразІнікше**.

Задання змінно в залежності від

ВМОВИ:

```
var age = 20
```

```
var canVote = age >= 18 ? 'так' : 'ні'
```

```
console.log('Чи може особа голосувати? ' + canVote)
```

```
// Виведе: "Чи може особа голосувати? так"
```

Використання тернарного оператора для виведення
повідомлень:

```
var score = 75
```

```
score >= 50 ? console.log('Ви пройшли') : console.log('Ви не пройшли')
```


Вкладені тернарні оператори для більш складних

VMOB:

```
var marks = 85;
```

```
var grade = marks >= 90 ? 'A' :
```

```
marks >= 80 ? 'B' :
```

```
marks >= 70 ? 'C' :
```

```
marks >= 60 ? 'D' : 'F';
```

```
console.log("Оцінка: " + grade); // Виведе: "Оцінка: B"
```

Оператор кома

,

Оператор **кома** є бінарним оператором, виконує кожен із його операндів (зліва направо) і повертає значення останнього операнда.

Оператор кома в JavaScript дозволяє нам виконати кілька виразів, розділених комою, в одному операторі. В результаті виконання такої конструкції повертається значення останнього виразу.

Синтаксис:

вираз1, вираз2, вираз3

Оператор кома використовується для:

- **Мінімізації коду** шляхом виконання декількох операцій в одному місці без необхідності розділення х на декілька рядків.
- **Ініціалізації** або **оновлення** кількох змінних в циклі.

```
// Ініціалізація кількох змінних:
```

```
var a = 1, b = 2, c = 3; // Встановлює значення для a, b, і c
```

```
// Використання в циклі for:
```

```
for (var i = 0, j = 10; i <= 10; i++, j--) {
```

```
    console.log(i, j); // Виводить пари чисел: 0 10, 1 9, ..., 10 0  
}
```

```
// Оцінка кількох виразів:
```

```
var x, y, z;
```

```
x = (y = 5, z = 10); // y стає 5, z стає 10, і x стає 10
```

// Виклик кількох функцій в одному рядку:

```
getData(), processData(), displayData();
```

// Виконання коду з побічними ефектами перед поверненням значення:

```
function doSomething() {
```

```
    return console.log('Повідомлення'). 5;
```

```
}
```

Цикли

Цикли використовуються для **повторення блоку коду** кілька разів. Кожен тип циклу має своє призначення та сценарій використання.

JavaScript має такі типи циклів:

- **for**
- **for..in**
- **for..of**
- **while**
- **do..while**

... і оператори для роботи з циклами: **label**, **break**, **continue**

for

Синтаксис:

for (**ініціалізація**; **умова**; **крок**) **тіло циклу**

ініціалізація – привласнюємо початкове значення

умова ^{змінно} – якщо це вираз істинно – цикл виконуватиметься,
якщо хибно – тіло циклу виконуватися нічого
очікувати.

крок – (інкремент, декремент) оновлюється значення
лічильника.

тіло циклу – код; *{ ... }* не є обов'язковими для однієї інструкції.

Однократне виконання тіла циклу називається **ітерацією**. Тобто, якщо тіло циклу було виконано *10 разів*, то пройшло **10 ітерацій**.

Найпростіший цикл **for** можна записати в такий спосіб:

```
for ( ; ; ) ;
```

Тут записаний нескінченний цикл, який нічого не робить.

Такий цикл краще не запускати, оскільки він призведе до зависання браузера.

Оголосити змінні можна у циклі:

```
for (var i = 0;;);
```

або поза циклом:

```
var i;  
for (i = 0;;);
```

Обидва записи ідентичні, але перший – лаконічніший.
Про область видимості змінних ми говоритимемо
пізніше.

```
for (var i = 0; i < 10; i++)  
    { console.log(i);  
}
```

⊘	top	▼	Filter	Default levels ▼	⚙
0				main.js:4	
1				main.js:4	
2				main.js:4	
3				main.js:4	
4				main.js:4	
5				main.js:4	
6				main.js:4	
7				main.js:4	
8				main.js:4	
9				main.js:4	
>					

- i = 0** – ініціалізація виконується один раз на початку циклу.
- i < 10** – умова виконується на кожній ітерації перед виконанням тіла циклу.
- i++** – інкремент виконується на кожній ітерації після виконання тіла циклу.

```
for (var i = 10; i > 0; i--) {  
    console.log(i);  
}
```

10	main.js:4
9	main.js:4
8	main.js:4
7	main.js:4
6	main.js:4
5	main.js:4
4	main.js:4
3	main.js:4
2	main.js:4
1	main.js:4
>	

Вважається, що такий цикл працює швидше.

0 при перетворенні на логічний тип повертає **false**, тому можна викинути інкремент (3-й вираз), а умову записати як **i--**. При цьому **i** лежить в іншому діапазоні:

```
for (var i = 10; i--;) {  
    console.log(i);  
}
```

9	main.js:4
8	main.js:4
7	main.js:4
6	main.js:4
5	main.js:4
4	main.js:4
3	main.js:4
2	main.js:4
1	main.js:4
0	main.js:4

for..of

for..of - це цикл, який дозволяє пройтися по всіх елементах ітерованого об'єкта, наприклад, масиву або рядка.

Він використовується для того, щоб зробити код, що перебирає колекції, читабельнішим і легшим для написання.

Коли використовується **for..of**, можна отримати безпосередній доступ до кожного елемента в колекції, не турбуючись про створення лічильника або доступ через індекси.

Приклад використання

for..of:

```
var fruits = ['яблуко', 'банан', 'киві']  
for (let fruit of fruits) {  
    console.log(fruit)  
}
```

for..in

for..in - це цикл, який використовується для ітерації по ключах об'єкта, тобто він дозволяє перебрати всі перелічувані властивості об'єкта, включаючи властивості, успадковані через ланцюжок прототипів.

Цей цикл зазвичай використовується, коли тобі потрібно працювати з властивостями об'єктів.

Приклад використання

for..in:

```
var user = {  
  name: 'Олекса',  
  age: 28,  
  profession: 'розробник'  
}  
for (var key in user) {  
  console.log(key + ': ' + user[key])  
}
```

while

Цикл **while** має наступний синтаксис:

while (**вираз**) **тело** **цикла**

Тут все трохи простіше. На кожній ітерації ми перевіряємо вираз істинності. У разі істинності виразу – виконується тіло циклу.

```
var i = 0;
while (i < 10) {
  console.log(i);
  i++;
}
```



0	main.js:4
1	main.js:4
2	main.js:4
3	main.js:4
4	main.js:4
5	main.js:4
6	main.js:4
7	main.js:4
8	main.js:4
9	main.js:4
>	

Інкремент можна записати будь-де, наприклад, для стислості – так:

```
while (i < 10)
{ console.log(i++)
};
```



0	main.js:4
1	main.js:4
2	main.js:4
3	main.js:4
4	main.js:4
5	main.js:4
6	main.js:4
7	main.js:4
8	main.js:4
9	main.js:4
>	

У випадку, як і з циклом **for**, ми можемо реалізувати декрементний лічильник:

```
while (i--)  
{ console.log(i  
);  
}
```



⊘	top	▼	Filter	Default levels ▼	⚙
9					main.js:4
8					main.js:4
7					main.js:4
6					main.js:4
5					main.js:4
4					main.js:4
3					main.js:4
2					main.js:4
1					main.js:4
0					main.js:4
>					

do .. while

Цикл **do..while** має наступний синтаксис:

do тіло циклу while (вираз)

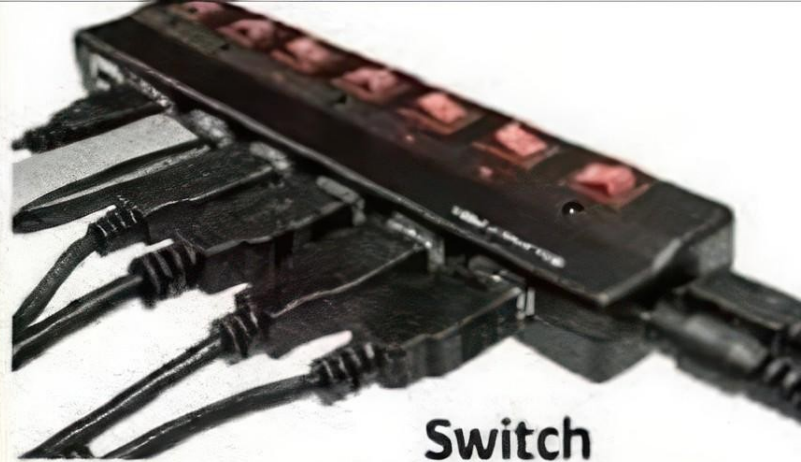
Він застосовується набагато рідше, ніж цикл **while** та його єдина відмінність від циклу **while** полягає в тому, що **умова перевіряється після виконання тіла циклу**.

Таким чином, тіло буде виконано мінімум один раз незалежно від істинності виразу.

```
var i = 0;  
do {  
    console.log(i++);  
} while (i < 10);
```



top	Filter	Default levels	
0			main.js:4
1			main.js:4
2			main.js:4
3			main.js:4
4			main.js:4
5			main.js:4
6			main.js:4
7			main.js:4
8			main.js:4
9			main.js:4
>			



for + break

```
for (let i = 0; i < 10; i++) {  
  if (i === 5) {  
    break // Припиняє виконання циклу, коли i дорівнює 5  
  }  
  console.log(i) // Виведе числа від 0 до 4  
}
```


for + break + continue

```
for (let i = 0; i < 10; i++)  
  { if (i === 5) {  
    break // Припиняє виконання циклу, коли i дорівнює 5  
  }  
  if (i % 2 === 0) {  
    // Якщо i є парним числом  
    continue // Пропускає поточну ітерацію i переходить до наступної  
  }  
  console.log(i) // Виведе лише непарні числа до 5 (1, 3)  
}
```

while + break

```
let i = 0
while (i < 10) {
  if (i === 5) {
    break // Припиняє цикл, коли i дорівнює 5
  }
  console.log(i) // Виведе числа від 0 до 4
  i++ // Не забуваємо збільшити лічильник
}
```

while + break + continue

```
let j = 0;
while (j < 10)
{
  if (j === 5)
  {
    break; // Припиняє цикл, коли j дорівнює 5
  }

  j++; // Збільшуємо лічильник на початку, тому що continue пропускає все після нього в циклі
  if (j % 2 === 0) {
    continue; // Пропускає поточну ітерацію, якщо j є парним числом
  }

  console.log(j); // Виведе лише непарні числа до 5 (1, 3)
}
```

Функці

Функція в програмуванні — це відокремлений блок програмного коду, який розробляється для виконання певно задачі або обчислення.

Вона може приймати дані на вході (аргументи), обробляти їх та повертати результат.

Функції створюються для того, щоб код, який виконує певну операцію, міг легко викликатися з різних місць програми без необхідності його дублювання.

Оголошення функції

```
function ідентифікатор (аргументи)  
    { інструкції  
      return вираз  
    }
```

ідентифікатор – ім'я функції .

аргументи – список змінних, які передаються всередину функції .

return – повертає значення з функції .

Function declaration statement (Оголошення функції)

Оголошення функції (function declaration statement):

```
function sayHello(name) {  
    return 'Hello ' + name  
}
```

```
console.log(sayHello('students')) // 'Hello students'
```

Оскільки функція повертає рядок, можна відразу викликати методи **String**:

```
console.log(sayHello('students').toUpperCase()) // 'HELLO STUDENTS'
```

Function definition expression (Функціональний вираз)

Функціональний вираз (function definition expression):

```
var sayHello = function (name) {  
    return 'Hello ' + name  
}; // ставимо: оскільки це звичайна інструкція з ініціалізацією  
console.log(sayHello('students').toUpperCase());
```

Функція `function() {...}` називається **анонімною**.
виду

Якщо ми **передаємо в функцію більше аргументів**, ніж очікується – помилки не відбудеться, просто всередині функції цьому аргументу не буде надано жодне ім'я.

```
var sayHello = function (name)
{ return 'Hello ' +      name;
};
console.log(sayHello('students', 123).toUpperCase());
```

Але це не означає, що ми не зможемо використати цей аргумент усередині функції .

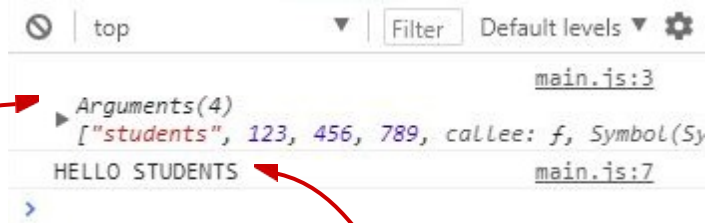
Якщо ми передаємо у функцію меншу кількість аргументів, ніж очікується – помилки також не станеться, при цьому всім неініціалізованим аргументам надається значення **undefined**.

```
var sayHello = function (name)
{ return 'Hello ' +      name;
};
console.log(sayHello().toUpperCase()); // HELLO UNDEFINED
```

arguments

Щоб отримати всі передані аргументи всередині функції, ми можемо використовувати об'єкт **arguments**.

```
var sayHello = function (name)
{ console.log(arguments);
  return 'Hello ' + name;
};
console.log(sayHello('students', 123, 456, 789).toUpperCase());
```



Об'єкт **arguments** поводитьься як масив, виводиться в консоль як масив, можна звертатися до окремих аргументів за допомогою індексів:

```
console.log(arguments[1]);    // 123  
console.log(arguments[2]);    // 456
```

... тим не менш, це **об'єкт**.

Перша корисна властивість об'єкта **arguments** – довжина, **length**:

```
console.log(arguments.length);    // 4
```

... за допомогою нього ми можемо, наприклад, перебрати всі передані аргументи в циклі та виконати будь-які дії з ними.

Callback функці

Callback функції в програмуванні — це функції , які передаються як аргументи іншим функціям.

Ми використовуємо **callback**-функції для того, щоб забезпечити можливість виклику певного коду після завершення виконання іншого коду, що дозволяє нам створювати асинхронний код, наприклад, при роботі з запитами до сервера або обробці подій.

```
function mySandwich (param1, param2, callback)  
{...}
```

```
mySandwich(param1, param2,  
callback);
```



```
function mySandwich (param1, param2, callback)
{ alert('Починаємо сти бутерброд.\n\n\
      Параметри: ' + param1 + ', ' +
      param2); callback();
}

mySandwich(param1, param2, callback);
```

```
function mySandwich (param1, param2, callback)
{ alert('Починаємо сти бутерброд.\n\n\
      Параметри: ' + param1 + ', ' +
      param2); callback();
}


mySandwich('шинка', 'сир', function()
{ alert('Закінчуємо сти
      бутерброд.');
```

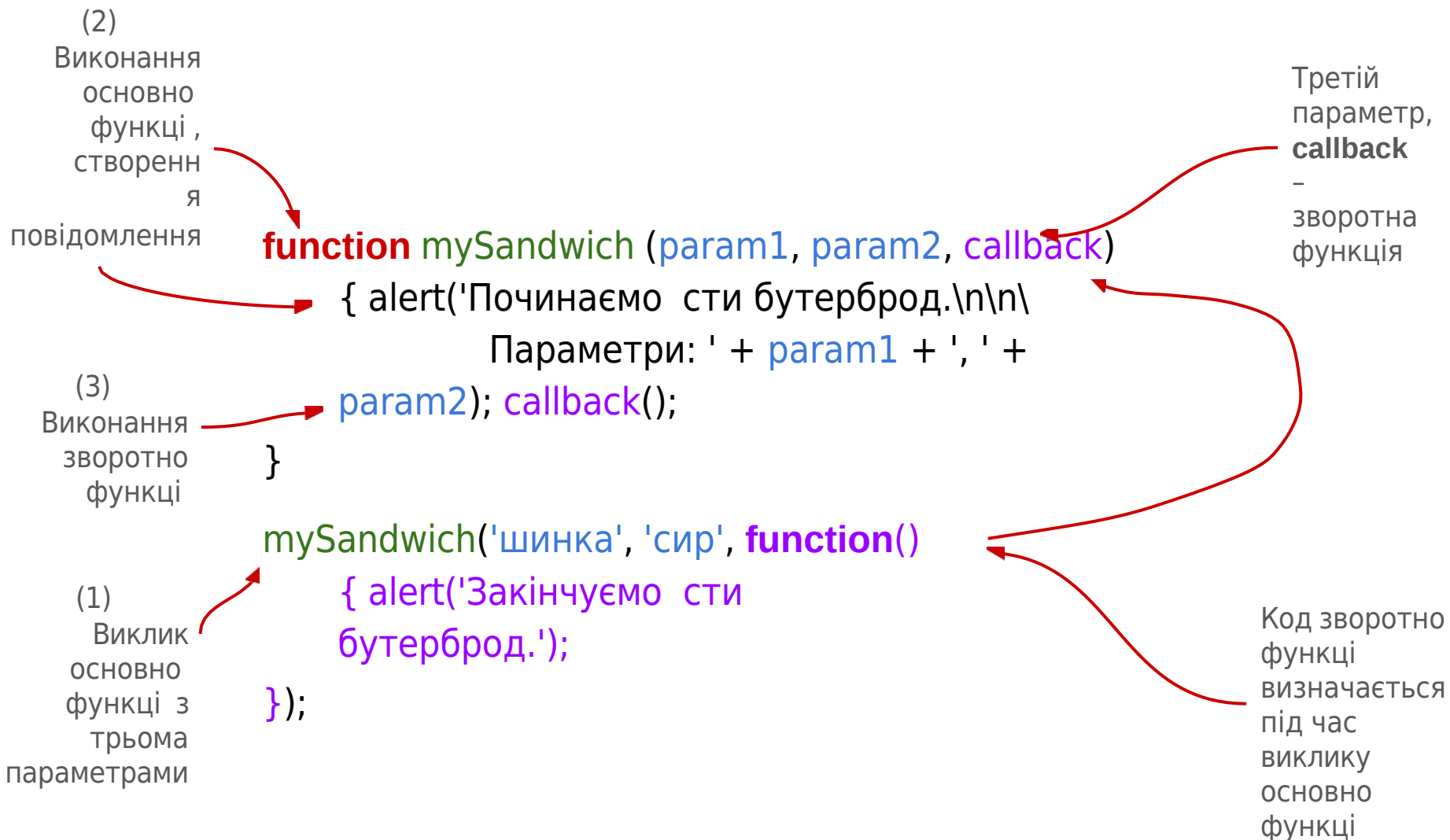
```
function mySandwich (param1, param2, callback)
{
    alert('Починаємо сти бутерброд.\n\n\
        Параметри: ' + param1 + ', ' +
        param2); callback();
}
```

```
mySandwich('шинка', 'сир', function()
{
    alert('Закінчуємо сти
    бутерброд.');
```

```
});
```

(1)
Виклик
основно
функції з
трьома
параметрами





Тут ми маємо функцію **mySandwich**, яка приймає три параметри.

Третій параметр – зворотна функція. Коли основна функція виконується, вона створює повідомлення з виведенням переданих параметрів. Потім виконується зворотна функція.

Зверніть увагу на параметр **callback**. Коли виконується функція повороту, потрібно використовувати дужки.

Сам код зворотної функції визначається третьому аргументі при виклику основної функції. Поворотна функція просто виводить повідомлення у тому, що вона виконалася. Тобто як аргумент функції може виступати функція.

ще приклади...

```
var func = function (callback)  
{...};
```

```
func(callback)  
;
```

ще приклади...

```
var func = function (callback)
    { var name = 'students';
      callback(name);
    };

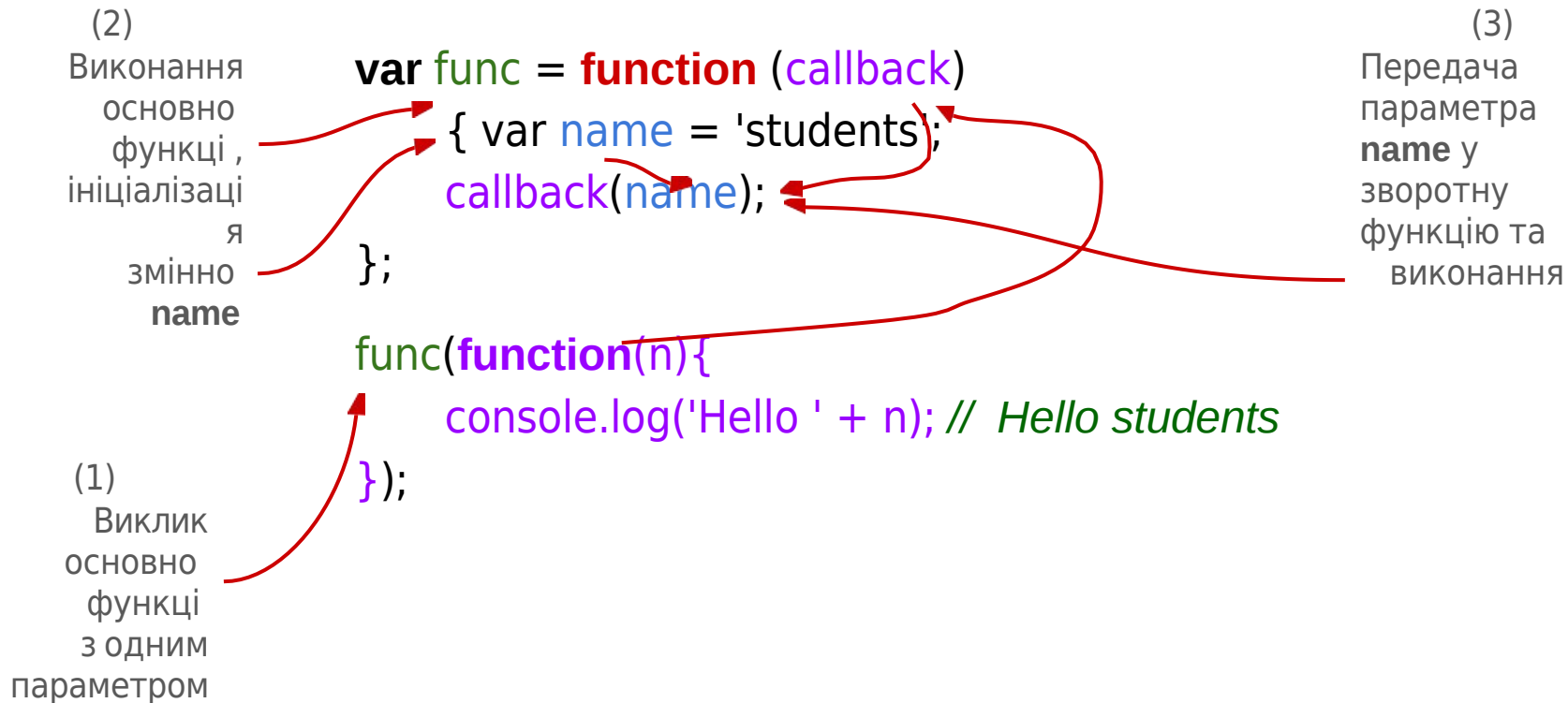
func(callback);
```

ще приклади...

```
var func = function (callback)
    { var name = 'students';
      callback(name);
    };

func(function(n){
    console.log('Hello ' + n); // Hello students
});
```


ще приклади...



// визначаємо функцію з аргументом callback

```
function someFunction (arg1, arg2, callback)  
{...}
```

// викликаємо функцію

```
someFunction(arg1, arg2,  
callback);
```

```
function someFunction (arg1, arg2, callback) {
```

```
    var myNumber = Math.ceil(Math.random() * (arg1 - arg2) + arg2);
```

```
    // тепер все готово і ми викликаємо callback, куди передаємо наш результат
```

```
    callback(myNumber);
```

```
}
```

```
someFunction(arg1, arg2, callback);
```