**Створюємо перший скрипт**

Створення першого скрипта на JavaScript можна реалізувати за декілька простих кроків. Ось приклад простого скрипта, який виводить повідомлення "Привіт, світ!" у консоль браузера:

**1.Створіть HTML-файл**: Відкрийте текстовий редактор і створіть новий файл з розширенням .html. Назвіть його, наприклад, index.html.

**2.Додайте базову структуру HTML**: Введіть або скопіюйте наступний код у ваш файл:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Мій перший JavaScript</title>

</head>

<body>

<h1>Вітаю з першим скриптом на JavaScript!</h1>

<script>

// Ваш JavaScript код тут

</script>

</body>

</html>

3.Додайте JavaScript код: В середині тегу <script> додайте JavaScript код. Для виведення повідомлення "Привіт, світ!" використайте console.log():

<script>

console.log('Привіт, світ!');

</script>

Повний код сторінки тепер виглядатиме так:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Мій перший JavaScript</title>

</head>

<body>

<h1>Вітаю з першим скриптом на JavaScript!</h1>

<script>

console.log('Привіт, світ!');

</script>

</body>

</html>

4.Запустіть ваш HTML-файл: Відкрийте файл index.html у вашому веб-браузері. Щоб побачити результат виконання JavaScript коду, відкрийте інструменти розробника у вашому браузері (зазвичай це можна зробити натиснувши F12 або правою кнопкою миші та обравши "Переглянути код") і перейдіть до вкладки "Console".

Ви побачите ваше повідомлення "Привіт, світ!" у консолі браузера. Це підтверджує, що ваш JavaScript код працює і виконується браузером.

## Способи підключення скриптів

Підключення JavaScript скриптів до HTML-документа може відбуватися декількома способами, і кожен з них має свої особливості та призначення.

### Пряме вбудовування коду

Код JavaScript може бути вбудованим безпосередньо у HTML-документ між тегами <script> і </script>. Це зручно для невеликих скриптів.

<script>

alert("Привіт, світ!");

</script>

### Зовнішній файл

JavaScript також може бути розміщений у зовнішньому файлі і підключений через атрибут src тега <script>. Це сприяє кращій організації коду та його повторному використанню.

<script src="path/to/your-script.js"></script>

**Важливо**: коли використовується src, вміст між тегами <script> ігнорується.

### Атрибут async

Атрибут async дозволяє браузеру завантажувати зовнішній скрипт асинхронно. Це означає, що скрипт буде завантажуватися паралельно з HTML-документом і виконаний як тільки буде завантажений, не блокуючи аналіз документа.

<script async src="path/to/your-script.js"></script>

### Атрибут defer

Атрибут defer також дозволяє асинхронне завантаження скриптів, але з однією ключовою відмінністю: скрипти з defer не виконуються до повного завантаження документа і виконуються в порядку їх розміщення у документі.

<script defer src="path/to/your-script.js"></script>

**Приклад**: якщо у вас є скрипт, який маніпулює DOM-елементами, використання defer гарантує, що скрипт виконається тільки після того, як увесь HTML-документ буде завантажений і аналізований, тобто всі елементи будуть доступні для маніпуляції.

Вибір між async і defer залежить від конкретних потреб вашого скрипта та веб-сторінки.

## Інструкції та Блок інструкцій

### Інструкції

Інструкції в JavaScript - це вирази або команди, які виконують певні дії у наших скриптах. Ми використовуємо інструкції для того, щоб визначати змінні, керувати потоком виконання програми за допомогою умовних виразів, циклів, викликати функції тощо. Інструкції дозволяють нам створювати інтерактивні веб-сайти, обробляти дані, взаємодіяти з користувачем та багато іншого.

Приклад простої інструкції, яка створює змінну та присвоює їй значення:

var message = "Привіт, світ!";

У цьому прикладі ми використовуємо ключове слово var для оголошення змінної з іменем message. Потім ми присвоюємо їй рядкове значення "Привіт, світ!" за допомогою оператора присвоєння =. Ця інструкція дозволяє зберегти рядкове значення у змінній, щоб ми могли використовувати або змінювати його в майбутньому.

### Блок інструкцій

Блок інструкцій у JavaScript - це група однієї або декількох інструкцій, об'єднаних разом всередині фігурних дужок {}. Ми використовуємо блоки інструкцій для організації коду в логічні секції, які можна використовувати в умовних конструкціях, циклах, функціях та інших структурах управління потоком виконання. Блоки дозволяють нам групувати кілька дій разом, щоб вони виконувалися як єдине ціле під певними умовами або в рамках певних конструкцій.

Приклад використання блоку інструкцій у циклі for:

for (var i = 0; i < 5; i++) {

console.log("Значення змінної i зараз є: " + i);

}

У цьому прикладі ми оголошуємо змінну i за допомогою var і встановлюємо її початкове значення на 0. Потім ми використовуємо цикл for для того, щоб виконати блок інструкцій, який виводить поточне значення i у консоль, доки i менше 5. Блок інструкцій всередині циклу дозволяє нам виконати декілька команд для кожної ітерації циклу.

Блоки інструкцій є фундаментальною концепцією в JavaScript, яка надає структуру та гнучкість управлінню потоком виконання програм. Вони дозволяють згрупувати великі частини коду для кращої читабельності та підтримки.

## Вирази, Оператори та Типи операторів

### Вирази

Вирази в JavaScript - це фрагменти коду, які можуть бути оцінені та перетворені в значення. Ми використовуємо вирази для виконання обчислень, присвоєння значень змінним, виклику функцій, та операцій з об'єктами та іншими типами даних. Вирази слугують основними будівельними блоками для програм на JavaScript, дозволяючи створювати динамічні інструкції, які можуть адаптуватися до даних, з якими працює програма.

Ось приклад виразу, який обчислює суму двох чисел:

var sum = 10 + 5;

У цьому прикладі 10 + 5 є виразом, який обчислюється до значення 15, і це значення присвоюється змінній sum за допомогою оператора присвоєння =.

Також вираз може бути використаний для визначення умови в умовному операторі, наприклад:

if (sum > 20) {

console.log("Sum is greater than 20.");

} else {

console.log("Sum is not greater than 20.");

}

У цьому випадку sum > 20 є виразом, який перевіряє, чи змінна sum більша за 20. Вираз повертає булеве значення (true або false), яке використовується умовним оператором if для визначення, який блок коду виконати.

Вирази в JavaScript є вкрай гнучкими та потужними, надаючи можливість виконувати широкий спектр обчислень та маніпуляцій з даними всередині програм.

### Оператори

Оператори в JavaScript - це символи або слова, які використовуються для виконання операцій на одному або декількох операндах (значеннях або змінних) і повернення результату. Вони дозволяють нам виконувати математичні обчислення, порівняння значень, логічні операції, присвоєння значень, доступ до властивостей об'єктів та інше. Оператори є фундаментальними для створення виразів у JavaScript, що дозволяє нам маніпулювати даними та керувати потоком виконання програм.

Ось кілька прикладів операторів і як ми їх використовуємо:

### Математичні оператори

var sum = 10 + 5; // Оператор додавання '+'

var difference = 10 - 5; // Оператор віднімання '-'

var product = 10 \* 5; // Оператор множення '\*'

var quotient = 10 / 5; // Оператор ділення '/'

У цих прикладах ми використовуємо математичні оператори для виконання основних арифметичних обчислень і присвоюємо результати змінним.

### Оператори порівняння

var isEqual = (10 == 5); // Перевіряємо рівність, повертає false

var isNotEqual = (10 != 5); // Перевіряємо нерівність, повертає true

Оператори порівняння використовуються для порівняння двох значень, що дозволяє нам виконувати різні дії на основі результату порівняння.

### Логічні оператори

var and = (true && false); // Логічне 'І', повертає false

var or = (true || false); // Логічне 'АБО', повертає true

var not = !(true); // Логічне 'НЕ', повертає false

Логічні оператори використовуються для виконання логічних операцій, що є основою для створення умовних виразів та керування потоком програми.

Оператори в JavaScript грають ключову роль у створенні динамічних і гнучких програм, надаючи розробникам потужні інструменти для маніпуляції даними та логіки програми.

Error

### Типи операторів

Оператори в JavaScript поділяються на унарні, бінарні та тернарний, залежно від кількості операндів, з якими вони працюють.

### Унарні оператори

Унарні оператори приймають лише один операнд. Вони використовуються для виконання операцій, які потребують тільки одне значення, наприклад, зміну знака числа на протилежний або інкрементацію/декрементацію значення змінної на одиницю.

Приклади унарних операторів:

* Інкремент ++ (збільшує значення на одиницю)
* Декремент -- (зменшує значення на одиницю)
* Унарний мінус `` (змінює знак числа)
* Унарний плюс + (спроба перетворити операнд на число)

### Бінарні оператори

Бінарні оператори вимагають два операнди для виконання операцій. Вони використовуються для обчислень, порівнянь, логічних операцій та інших типів маніпуляцій з даними.

Приклади бінарних операторів:

* Арифметичні оператори (наприклад, +, , , /)
* Оператори порівняння (наприклад, ==, !=, >, <)
* Логічні оператори (наприклад, &&, ||)

### Тернарний оператор

Тернарний оператор в JavaScript - це єдиний оператор, який працює з трьома операндами. Він часто використовується як короткий спосіб для умовного виразу. Тернарний оператор дозволяє вибрати одне з двох значень залежно від умови, що робить код більш концентрованим та читабельним.

Формат тернарного оператора:

умова ? вираз1 : вираз2;

Якщо умова істинна, вибирається вираз1; якщо умова хибна - вираз2.

Оператори бувають різних типів, щоб надати гнучкість у виразах та дозволити розробникам виконувати широкий спектр операцій з даними, від простих математичних обчислень до складних логічних умовних конструкцій.

## Змінні

У JavaScript, змінні є контейнерами для зберігання даних, які можуть змінюватися з часом. Ми використовуємо змінні для того, щоб маніпулювати даними, зберігати результати обчислень та керувати потоком виконання програми за допомогою умовних конструкцій та циклів.

### Оголошення змінних

Оголошення змінної відбувається, коли ми вказуємо інтерпретатору JavaScript про намір використовувати змінну. При цьому можемо не присвоювати їй відразу якесь значення.

Наприклад:

var message;

### Ініціалізація змінних

Ініціалізація змінної відбувається, коли ми вперше присвоюємо їй значення. Це може бути зроблено одразу під час оголошення або пізніше в коді:

message = 'Hello';

або одразу під час оголошення:

var message = 'Hello!';

Ми можемо оголосити кілька змінних одним рядком, використовуючи одне оголошення var, розділяючи їх комами:

var user = 'John', age = 25, message = 'Hello';

Такий підхід дозволяє нам створювати чистіший і більш організований код, особливо коли потрібно оголосити кілька змінних, які будуть використовуватися разом.

### let та const

Хоча в цьому прикладі ми фокусуємося на var, важливо знати, що в JavaScript також існують let та const для оголошення змінних. Вони надають більше контролю над областю видимості змінної та незмінністю, відповідно, і будуть детальніше розглянуті пізніше.

Використання змінних є основою програмування на JavaScript, дозволяючи нам зберігати та маніпулювати даними в процесі виконання програм.

### Назва змінних

Конвенції назви змінних та нотації, такі як camelCase, є важливими аспектами програмування, оскільки вони допомагають забезпечити читабельність та однорідність коду в межах проєкту. Ці конвенції сприяють кращому розумінню коду та співпраці між розробниками.

**Що це таке?**

* **Конвенції назви змінних** – це домовленості про те, як називати змінні. Вони можуть включати правила про використання малих та великих літер, цифр, підкреслень тощо.
* **Нотація camelCase** – це спосіб письма ідентифікаторів, в якому кожне нове слово починається з великої літери, а перше слово з маленької. Вона названа так через свою схожість з горбами верблюда.

**Навіщо використовується?**

Конвенції та нотації використовуються для того, щоб код був більш організованим і легким для розуміння. Вони допомагають уникнути плутанини у великих проєктах, де багато розробників працюють над одним кодом. Це також полегшує виявлення змінних, функцій, класів тощо.

**Розглянемо приклад використання camelCase** у мові програмування JavaScript:

var userName = "JohnDoe"; // Змінна для зберігання імені користувача

var userAge = 30; // Змінна для зберігання віку користувача

var isUserLoggedIn = true; // Змінна для перевірки, чи ввійшов користувач у систему

function updateUserProfile(name, age) {

userName = name;

userAge = age;

}

updateUserProfile("JaneDoe", 28);

У цьому прикладі ми використовуємо camelCase для назв змінних та функцій (userName, userAge, isUserLoggedIn, updateUserProfile). Це дозволяє легко розрізняти між окремими словами в ідентифікаторах, зберігаючи при цьому код чистим і читабельним.

## Типи даних

У JavaScript існує 8 (вісім) основних типів даних, які поділяються на примітиви та об'єкти.

### Примітиви

Примітивні типи даних - це базові типи, які представляють одне значення:

1. **String**: Текст.
2. **Number**: Числа.
3. **BigInt**: Великі цілі числа.
4. **Boolean**: **true** або **false**.
5. **Undefined**: Невизначене значення.
6. **Null**: Відсутність значення.
7. **Symbol**: Унікальний ідентифікатор.

### Об'єкти

1. **Object** - це складний тип даних, який представляє колекції даних або більш складні структури. Об'єкти можуть містити примітивні типи даних, інші об'єкти або функції як свої властивості.

У JavaScript, примітиви та об'єкти взаємодіють між собою, дозволяючи розробникам ефективно працювати з різними типами даних і структурами даних для створення функціональних і гнучких програм.

### Примітиви (прості типи)

Примітиви, або прості типи, в JavaScript - це основні типи даних, які не є об'єктами і не мають методів. Вони представляють одиничні значення і вважаються незмінними, тобто їх значення не можна змінити після створення. Існує сім примітивних типів даних:

1. **String** - для текстових даних, наприклад: 'Hello, World!'.
2. **Number** - для числових даних, включаючи цілі числа та числа з плаваючою точкою, наприклад: 42, 3.14.
3. **BigInt** - для представлення цілих чисел, які перевищують обмеження типу Number, наприклад: 9007199254740991n.
4. **Boolean** - для логічних значень, може бути true або false.
5. **Undefined** - тип для змінних, яким не було присвоєно значення, вказує на відсутність визначеного значення.
6. **Null** - використовується для представлення відсутності будь-якого значення.
7. **Symbol** - введений у ECMAScript 2015, представляє унікальні та неповторні ідентифікатори.

Примітиви важливі тому, що вони дозволяють розробникам працювати з основними блоками інформації в програмі. Хоча вони самі по собі прості, примітиви можуть бути використані для побудови складніших структур даних і логіки програм.

### Об'єкти (об'єктні типи)

Об'єктні типи - це складні структури даних, які використовуються для зберігання колекцій даних та більш складних взаємозв'язків. Вони дозволяють нам організовувати та маніпулювати даними в більш структурованому та ефективному форматі.

* **Об'єкти**: Слугують для зберігання властивостей та методів у форматі ключ-значення. Використовуються для представлення реальних об'єктів, конфігурацій, опцій тощо.

var person = {name: "John", age: 30};

* **Масиви**: Спеціалізований тип об'єктів для зберігання списків та колекцій значень у впорядкованому вигляді. Використовуються для обробки послідовностей даних.

var numbers = [1, 2, 3, 4, 5];

* **Функції**: Функції в JavaScript є першокласними об'єктами, що означає, що вони можуть мати властивості та методи, але самі по собі не є об'єктами в традиційному розумінні. Функції можуть бути викликані, приймати аргументи та повертати значення. Вони використовуються для виконання коду, реалізації логіки програми та маніпуляцій з даними.

function greet(name) {

return "Hello " + name + "!";

}

* **Регулярні вирази**: Об'єкти, які представляють шаблони пошуку та заміни у тексті. Використовуються для валідації, пошуку та обробки тексту.

var regex = /hello/;

* **Об'єкт помилки**: Спеціалізовані об'єкти, які представляють помилки, що виникають під час виконання програми. Використовуються для обробки та повідомлення про помилки.

var error = new Error("Something went wrong");

* **Колекції**:

**Maps**: Колекції ключ-значення, де ключі можуть бути будь-якого типу. Використовуються для ефективного зберігання та пошуку даних.

var map = new Map();

map.set("key", "value");

**Sets**: Колекції унікальних значень будь-якого типу. Використовуються для зберігання унікальних елементів.

var set = new Set([1, 2, 3, 4, 5]);

**WeakMaps** та **WeakSets**: Варіанти Maps та Sets, які не перешкоджають зборці сміття для їх ключів/елементів. Використовуються для оптимізації пам'яті та управління залежностями, які можуть бути легко видалені.

var weakmap = new WeakMap();

var obj = {};

weakmap.set(obj, "value");

Кожен з цих типів даних виконує унікальну роль у розробці програм на JavaScript, дозволяючи розробникам ефективно структурувати та маніпулювати даними в різних сценаріях.

## Змінні та незмінні типи & Оператор присвоєння

У JavaScript значення діляться на змінні та незмінні типи. Незмінні типи включають примітивні типи, такі як числа, рядки, булеві значення, null і undefined, а також символи (Symbol). Ці типи називаються незмінними, оскільки після їх створення значення не можуть бути змінені.

З іншого боку, об'єктні типи, які включають об'єкти, масиви і функції, є змінними. Це означає, що їхній вміст або стан може бути модифікований після створення.

### Навіщо використовується?

Ця розбіжність між змінними та незмінними типами дозволяє розробникам мати гнучкість у роботі з даними, забезпечуючи можливість змінювати складні структури даних на льоту, а також зберігати незмінність простих типів для забезпечення передбачуваної поведінки та оптимізації виконання.

### Приклади коду

**Змінення об'єкта**

var obj = {};

obj.name = 'New name';

console.log(obj); // { name: 'New name' }

У цьому прикладі ми створили порожній об'єкт і додали в нього властивість name. Об'єкт змінився, демонструючи його змінність.

**Змінення масиву**

var array = [1, 2, 3];

array[1] = 54321;

console.log(array); // [1, 54321, 3]

Тут ми модифікуємо масив, змінюючи другий елемент. Масив як змінний тип дозволяє змінювати свій вміст.

**Спроба змінити рядок**

var myString = 'Lorem ipsum dolor sit amet.';

console.log(myString); // 'Lorem ipsum dolor sit amet.'

console.log(myString.toUpperCase()); // 'LOREM IPSUM DOLOR SIT AMET.'

console.log(myString); // 'Lorem ipsum dolor sit amet.'

У цьому випадку, хоча метод toUpperCase() повертає рядок у верхньому регістрі, вихідний рядок залишається незмінним. Це підтверджує незмінність рядків у JavaScript.

### Оператор присвоєння

Оператор присвоєння в JavaScript використовується для встановлення значення змінної. Він представлений знаком рівності (=). Основна мета цього оператора - присвоїти праву сторону виразу (значення або результат виразу) змінній, яка знаходиться з лівої сторони.

Побічний ефект у програмуванні - це будь-яка зміна стану програми, що відбувається в результаті виконання інструкції, окрім її основного призначення. У контексті оператора присвоєння, побічний ефект полягає в модифікації значення змінної. Це означає, що кожне використання оператора присвоєння змінює стан програми, присвоюючи нове значення змінній.

### Приклад коду

var a = 5; // Присвоєння змінній 'a' значення 5 - це побічний ефект, оскільки змінюється стан змінної 'a'

var b = a; // Присвоєння змінній 'b' значення змінної 'a' також є побічним ефектом

b += 2; // Оператор присвоєння з додаванням. Значення 'b' збільшується на 2, що також є побічним ефектом

У цьому прикладі ми бачимо три випадки використання оператора присвоєння:

1. Присвоєння простого значення змінній a.
2. Копіювання значення з однієї змінної (a) до іншої (b).
3. Використання складеного оператора присвоєння (+=), який збільшує значення змінної b на 2.

У кожному з цих випадків відбувається побічний ефект у вигляді зміни значення змінної, що впливає на стан програми. Оператор присвоєння є фундаментальним в програмуванні, оскільки він дозволяє управляти даними, зберігати та змінювати стани в межах програми.

## Number/BigInt

### Number

У JavaScript тип Number використовується для представлення як цілих, так і дробових чисел. Він підтримує стандарт IEEE 754 для подвійної точності, що дозволяє використовувати його для широкого спектру числових обчислень.

### Діапазон безпечних значень

Безпечний діапазон значень для типу Number в JavaScript знаходиться між -(2^53 - 1) та 2^53 - 1, включно. Це обмеження визначено через точність подання числа в бінарній формі. Значення за межами цього діапазону можуть втрачати точність при обчисленнях або порівнянні.

### Числовий літерал

Числовий літерал - це пряме представлення числа в коді. Воно може бути представлене у десятковій, шістнадцятковій, вісімковій або двійковій формі.

### Конструктор Number

Конструктор Number дозволяє створювати об'єкт-обгортку для примітивного числового типу. Це може бути корисним для використання числових методів або перетворення рядків на числа.

### Найпопулярніші методи числа

* toFixed(n) - форматує число, залишаючи n цифр після десяткового роздільника.
* toPrecision(n) - форматує число, залишаючи n значущих цифр.
* toString(base) - перетворює число в рядок, використовуючи задану основу (наприклад, двійкову, десяткову, шістнадцяткову).

### Приклад коду

var safeMax = Number.MAX\_SAFE\_INTEGER; // Найбільше безпечне ціле число

var safeMin = Number.MIN\_SAFE\_INTEGER; // Найменше безпечне ціле число

console.log(safeMax, safeMin); // 9007199254740991, -9007199254740991

var num = 12345.6789;

console.log(num.toFixed(2)); // '12345.68' - округлення до двох знаків після коми

console.log(num.toPrecision(6)); // '12345.7' - шість значущих цифр

console.log(num.toString(16)); // '3039.ad2f1a9fbe76c' - перетворення в шістнадцяткову систему

var fromString = Number("123.45");

console.log(fromString); // 123.45 - перетворення рядка на число

У цьому прикладі ми демонструємо роботу з діапазоном безпечних значень, використання числових методів для форматування та перетворення, а також створення числа з рядка за допомогою конструктора Number. Ці елементи є фундаментальними для роботи з числами у JavaScript і забезпечують гнучкість при обробці числових даних.

### BigInt

BigInt - це вбудований об'єкт у JavaScript, який надає можливість працювати з цілими числами довільної величини. Він був доданий до мови для того, щоб допомогти розробникам безпечно обробляти дуже великі числа, значення яких перевищують обмеження Number.MAX\_SAFE\_INTEGER.

### Навіщо використовується?

BigInt використовується для точних обчислень з великими цілими числами, які не можуть бути надійно представлені за допомогою типу Number через обмеження на максимальне безпечне ціле число. Він особливо корисний у фінансових обчисленнях, криптографії, обробці великих ідентифікаторів та інших областях, де потрібна висока точність.

### Приклад коду

// Створення BigInt через літерал

const bigInt1 = 1234567890123456789012345678901234567890n;

// Створення BigInt через конструктор

const bigInt2 = BigInt("1234567890123456789012345678901234567890");

// Операції з BigInt

const sum = bigInt1 + bigInt2;

console.log(sum.toString()); // Виведення: "2469135780246913578024691357802469135780"

// Порівняння BigInt з Number (пряме порівняння не можливе)

const smallNumber = 10;

console.log(bigInt1 > BigInt(smallNumber)); // true

// Ділення BigInt віддає цілу частину результату, ігноруючи дробову частину

const divisionResult = bigInt1 / 123n;

console.log(divisionResult); // Виведення: 1003717479311935481721869918699186991869918n

У цьому прикладі ми бачимо створення чисел BigInt як за допомогою літералу (n на кінці числа), так і за допомогою конструктора BigInt. Ми демонструємо основні операції з BigInt, включаючи арифметичні операції та порівняння з типом Number. Зауважимо, що при діленні BigInt результат завжди буде округлений до найближчого меншого цілого числа, ігноруючи дробову частину, що відрізняє поведінку BigInt від Number у JavaScript.