

Expressions and Operators.
Variables. Primitives. Objects.
Mutable and Immutable
Types.
Assignment Operator. Numbers.
BigInt

Зміст уроку

1. [Вступ](#)
2. [Що таке JavaScript?](#)
3. [Що таке ECMAScript?](#)
4. [Еволюція ECMAScript](#)
5. [Призначення JavaScript](#)
6. [Що вміє JavaScript?](#)
7. [JavaScript для Web](#)
8. [JavaScript для Node.js](#)
9. [Що НЕ вміє JavaScript?](#)
10. [У чому унікальність JavaScript?](#)
11. [Сучасний підручник з JavaScript](#)
12. [Створюємо перший скрипт](#)
13. [Способи підключення скриптів](#)
14. [Інструкції](#)
15. [Вирази](#)
16. [Оператори](#)
17. [Змінні](#)
18. [Типи даних](#)
19. [Примітиви \(прості типи\)](#)
20. [Об'єкти \(об'єктні типи\)](#)
21. [Змінні та незмінні типи](#)
22. [Оператор присвоєння](#)
23. [Number](#)
24. [BigInt](#)

ВСТП

Що таке JavaScript?

JavaScript - це універсальна мова програмування, що активно використовується у веб-розробці та є однією з найпопулярніших мов у світі.

JavaScript входить до тріади мов, якими має володіти будь-який веб-розробник:

- **HTML**
- **CSS**
- **JavaScript**

Що таке ECMAScript?

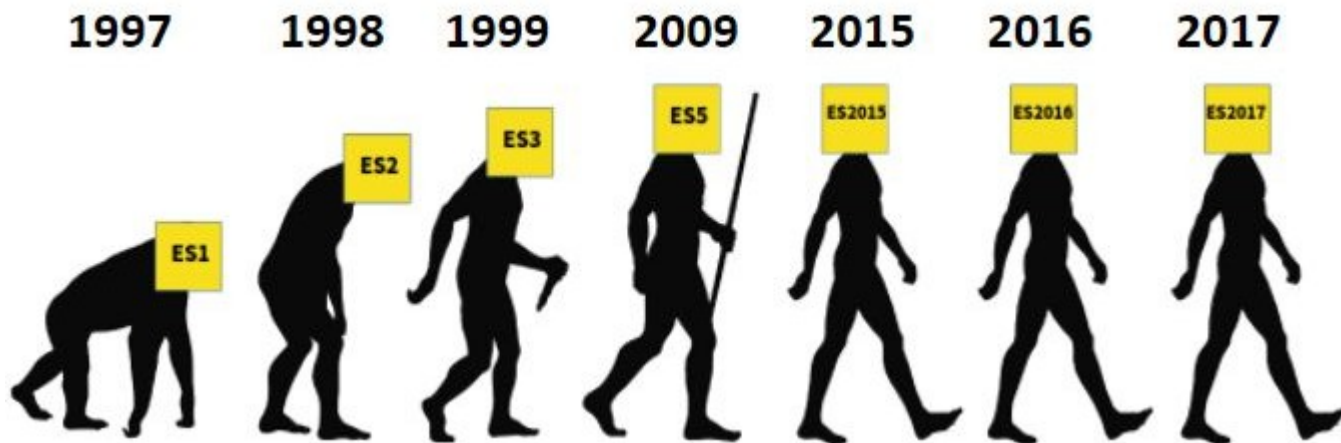
JavaScript – це скриптова мова загального призначення, що відповідає специфікації **ECMAScript**, це діалект мови ECMAScript.

ECMAScript – це специфікація, на якій базується мова **JavaScript**.

ECMAScript містить правила, відомості та рекомендації, які повинні дотримуватися скриптовою мовою, щоб вона вважалася сумісною з ECMAScript. Актуальна версія ECMAScript 6 (ES6).

JavaScript-движок - це програма або інтерпретатор, здатний розуміти та виконувати JavaScript-код.

Еволюція ECMAScript



ES1 (1997): Перше видання стандарту, створене для стандартизації JavaScript.

ES2 (1998): Внесла дрібні правки для відповідності ISO/IEC стандарту.

ES3 (1999): Додавало регулярні вирази, кращу обробку помилок, нові формати чисел.

ES4: Пропозиція була відкладена і ніколи не прийнята як стандарт.

ES5 (2009): Додавало строгий режим ('use strict'), JSON підтримку, нові методи для масивів.

ES6 (2015), також відомий як **ECMAScript 2015:** Класи, модулі, стрілочні функції, проміси, шаблонні літерали.

ES7 (2016), або **ECMAScript 2016:** Включав метод `Array.prototype.includes` та експоненційний оператор `(*)`.

ES8 (2017), або **ECMAScript 2017:** Асинхронні функції (`async/await`), метод `Object.values()`, `Object.entries()`.

ES9 (2018), або **ECMAScript 2018:** Оператори розпакування/збирання в об'єктах, асинхронна ітерація, регулярні вирази, поліпшення.

ES10 (2019), або **ECMAScript 2019:** Метод `Array.prototype.flat()`, `Array.prototype.flatMap()`,

`Object.fromEntries()`. **ES11 (2020),** або **ECMAScript 2020:** `BigInt`, динамічний імпорт, `nullish coalescing`

оператор, `Optional Chaining`. **ES12 (2021),** або **ECMAScript 2021:** Логічний оператор присвоєння,

роздільники цифр у числових літералах. **ES13 (2022),** або **ECMAScript 2022:** Методи класів, нові методи

масивів, постійні регулярні вирази.

ES14 (2023), або **ECMAScript 2023.**

Призначення JavaScript

JavaScript призначений для створення інтерактивних та динамічних веб- сайтів, дозволяючи розробникам маніпулювати вмістом і поведінкою веб- сторінок у реальному часі.

Код написаний на цій мові називають **скриптами**, які є звичайним текстом.

У веб-браузері скрипти підключаються безпосередньо до HTML і можуть виконуватися відразу після завантаження сторінки або у відповідь на ді користувача.

Процес виконання скриптів називається **інтерпретацією**.

Що вміє JavaScript?

JavaScript є універсальною мовою, що дозволяє виконувати широкий спектр завдань у різних областях програмування.

JavaScript для Web

В контексті веб-розробки, JavaScript використовується для додавання інтерактивності та динаміки до веб-сайтів. Основні можливості включають:

1. **Маніпуляція DOM:** Зміна структури, стилів або вмісту веб-сторінок динамічно.
2. **Обробка подій:** Відстеження користувацьких дій, таких як кліки мишею, натискання клавіш, прокрутка сторінок тощо.
3. **Валідація форм:** Перевірка даних форм на клієнтській стороні перед їх відправленням на сервер.
4. **Асинхронні запити:** Використання AJAX та Fetch API для асинхронного отримання даних з сервера без перезавантаження сторінки.
5. **Взаємодія з веб-API:** Використання різних API, доступних у сучасних браузерах, наприклад, Geolocation API, Web Storage API, Canvas API для графіки.
6. **Анімація:** Створення плавних анімацій елементів на сторінці.

JavaScript для Node.js

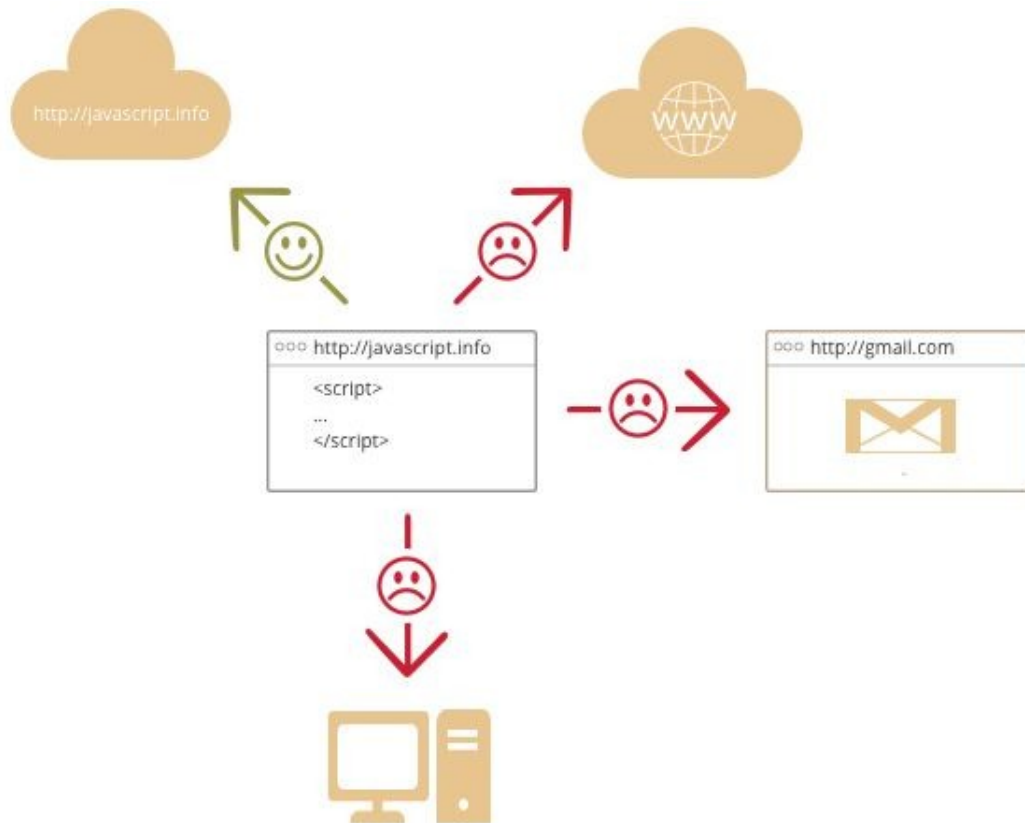
Node.js дозволяє використовувати JavaScript для розробки серверного програмного забезпечення, розширюючи можливості JavaScript за межі веб-браузерів:

1. **Розробка серверів:** Створення веб-серверів, RESTful API та реалізація серверної логіки.
2. **Робота з файловою системою:** Читання, запис, модифікація файлів та каталогів на сервері.
3. **Взаємодія з базами даних:** Підключення та робота з різними базами даних, такими як MongoDB, PostgreSQL, MySQL.
4. **Мережеве програмування:** Створення HTTP-серверів і клієнтів, веб-сокетів для реалізації двостороннього зв'язку в реальному часі.
5. **Модульна розробка:** Використання NPM для управління залежностями та спільної роботи з великою кількістю модулів та пакетів.
6. **Потокова обробка даних:** Робота з потоками для ефективного обробки великих об'ємів даних.
7. **Розробка інструментів командного рядка:** Створення утиліт командного рядка для автоматизації завдань і процесів розробки.

Використання JavaScript через Node.js розширює можливості мови на серверну сторону, дозволяючи розробникам створювати повноцінні веб-додатки, що включають як клієнтську, так і серверну логіку, використовуючи єдину мову програмування.

Що НЕ вміє JavaScript?

Більшість можливостей JavaScript у браузері обмежено поточним вікном та сторінкою.

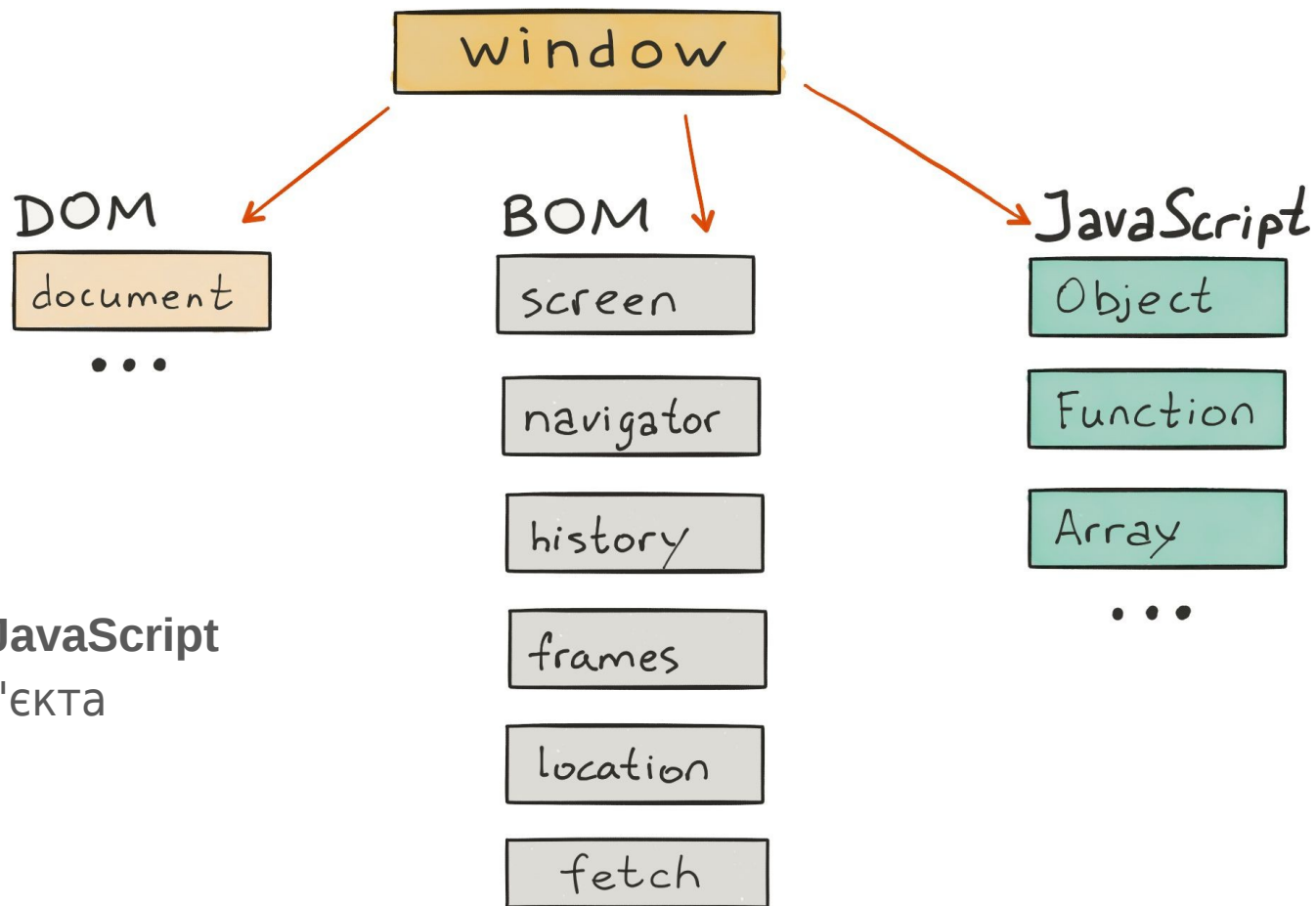


Незважаючи на свою універсальність та потужність, існують **обмеження** у тому, що **JavaScript не може робити**, особливо з міркувань безпеки та обмежень виконання в браузері:

1. Прямий доступ до файлової системи клієнта
2. Виконання довільних програм на комп'ютері користувача
3. Блокування основного потоку на тривалий час
4. Прямий доступ до апаратного забезпечення
5. Читання або запис інформації в базу даних безпосередньо
6. Виконання міжсайтового скриптингу (Cross-Site Scripting, XSS)
7. Взаємодія з іншими вкладками або програмами без явного дозволу користувача

У чому унікальність JavaScript?

1. Універсальність
2. Широка підтримка
3. Гнучкість
4. Велика екосистема
5. Спільнота
6. Інноваційність
7. Продуктивність



DOM, BOM, JavaScript
в ієрархії об'єкта
window

Сучасний підручник з JavaScript

Підручник із сучасного JavaScript: прості, але докладні пояснення з прикладами та завданнями, включаючи: замикання, DOM та події, об'єктно-орієнтоване програмування тощо.

<https://uk.javascript.info/>

Створюємо перший скрипт

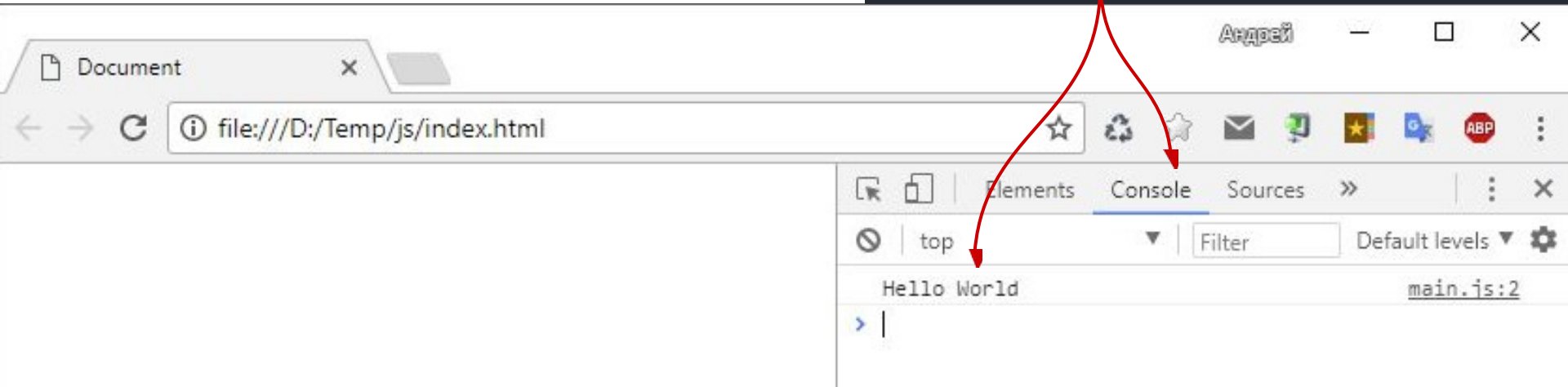
Створимо два файли: **index.html** и **main.js**, потім підключимо скрипт:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <script src="main.js"></script>
</body>
</html>
```

Виведемо що-небудь із скрипта в
консоль:

```
console.log('Hello World');
```

```
< > index.html x | main.js x
1
2 console.log('Hello World');
```



console.log

console.log - це метод **log** об'єкта **console**, який використовується для виведення повідомлень у консоль застосунку.

Це корисний інструмент для дебагінгу, оскільки дозволяє перевіряти значення змінних, стан застосунку або просто виводити інформаційні повідомлення під час розробки.

Приклад використання **console.log**:

```
console.log('Привіт, світ!');
```

alert

alert - це функція, що використовується для відображення модального діалогового вікна з повідомленням у браузері.

Це простий спосіб виводити інформацію користувачу вашого застосунку.

Коли викликається **alert**, виконання коду у застосунку тимчасово призупиняється, доки користувач не натисне кнопку "ОК" у діалоговому вікні.

Приклад використання **alert**:

```
alert( 'Ласкаво просимо!' );
```

prompt

prompt - це функція, яка відображає діалогове вікно з текстовим полем для введення користувачем даних у твоєму застосунку.

Ця функція зазвичай використовується для запиту у користувача ввести якусь інформацію.

Приклад використання **prompt**:

```
var name = prompt('Як тебе звати?', 'Введи своє ім'я тут');
```

confirm

confirm - це функція, яка відображає модальне діалогове вікно з повідомленням та двома кнопками: "ОК" та "Скасувати".

Ця функція використовується для отримання підтвердження від користувача перед виконанням певної дії. Вона повертає логічне значення **true**, якщо користувач натисне "ОК", і **false**, якщо вибере "Скасувати".

Приклад використання **confirm**:

```
var confirmation = confirm('Ти впевнений, що хочеш видалити свій профіль?');
```

Способи підключення скриптів

Оскільки зазначений **src**, то
внутрішня частина тегу ігнорується.

```
<script src="file.js">  
  alert(1);  
</script>
```



REJECT

```
<script src="file.js"></script>  
<script>  
  alert(1);  
</script>
```

Потрібно вибрати: чи **script** йде з
src, чи містить код. Тег вище слід
розбити на два: один – із **src**,
інший – із кодом, ось так...



ACCEPT

Атрибут async

Скрипт виконується повністю асинхронно. Тобто при виявленні

asvnc

```
<script async src="..."></script>
```

браузер не зупиняє обробку сторінки, а спокійно працює далі. Коли скрипт буде завантажено – він виконається.

Атрибут defer

Скрипт також виконується асинхронно, не змушує чекати на сторінку, але є дві відмінності від **async**.

Перше – браузер гарантує, що відносний порядок скриптів з **defer** буде збережено.

У такому коді (з **defer**) першим спрацює завжди **1.js**, а скрипт **2.js**, навіть якщо завантажився раніше, його чекатиме.

```
<script src="1.js" defer></script>  
<script src="2.js" defer></script>
```

У такому коді (з **async**) першим спрацює той скрипт, який раніше завантажиться:

```
<script src="1.js" async></script>  
<script src="2.js" async></script>
```

Інструкці

Інструкції в JavaScript - це вирази або команди, які виконують певні дії у наших скриптах.

```
var message = 'Привіт, світ!'
```

Усі програми на мові JavaScript складаються з **інструкцій** (statement):

інструкція; інструкція; інструкція;

Інструкції виконуються по черзі.

Усі інструкції , крім складових, відокремлюються **;**

Оскільки **порожні рядки ігноруються** інтерпретатором, зазвичай інструкції пишуться на новому рядку:

інструкція;

інструкція;

інструкція;

Складові інструкції

Для об'єднання кількох інструкцій у складову інструкцію використовується **блок інструкцій** (block statement) і він пишеться так:

```
{  
    інструкція1;  
    інструкція2;  
    інструкція3;  
}
```

Використовується в місцях, де парсер очікує побачити одну інструкцію, але ви хочете використовувати кілька інструкцій.

Блок інструкцій

Блок інструкцій у JavaScript - це група однієї або декількох інструкцій, об'єднаних разом всередині фігурних дужок **{}**.

Приклад використання блоку інструкцій у циклі **for**:

```
for (var i = 0; i < 5; i++) {  
    console.log('Значення змінної i зараз є: ' + i)  
}
```

Порожні інструкції (empty statement)

Порожні інструкції (**;** **;** **;** **;** **;** **;**) зазвичай використовуються там, де парсер очікує побачити якусь інструкцію, але ми не хочемо, щоб щось виконувалося. Другий випадок:

;(function(){...})(); – це функція, що самовикликається, розберемо пізніше

Перед тим як викладати сайт у продакшн, вважається гарною звичкою стискати та конкатенувати всі скрипти в один файл, що може спричинити помилки...

скрипт**;(function){}());**скрипт

Т

добре

скрипт**(function){}());**скрипт

Т

помилка

↓
скрипт**;;(function){}());**скрипт

добре

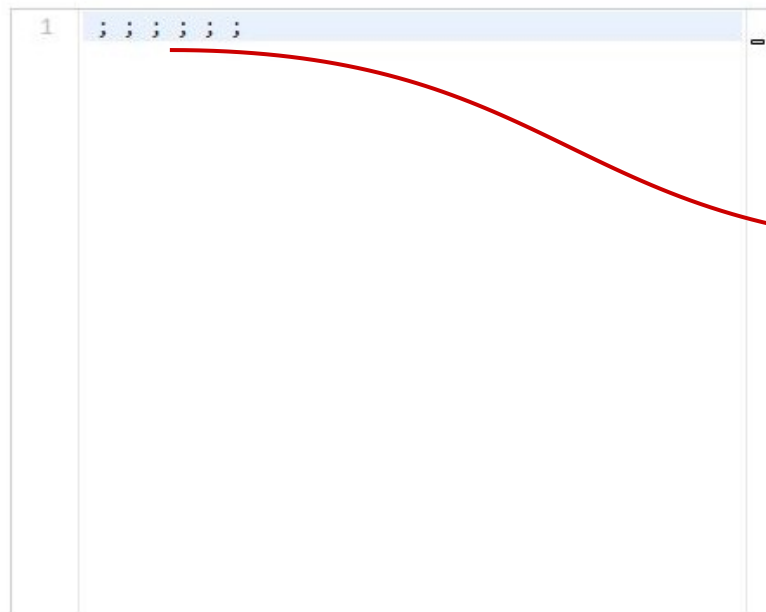
порожня інструкція ні на що не впливає

В Інтернеті існує ресурс, який емулює парсер JavaScript: Esprima.

Цей ресурс може створити синтаксичне дерево для нашого JavaScript- скрипта.

<http://esprima.org/demo/parse.html>

Parser produces the (beautiful) syntax tree



No error

Syntax node location info (start, end):

- ☐ Index-based range
- ☐ Line and column-based
- ☐ Attach comments

SyntaxTreeTokens

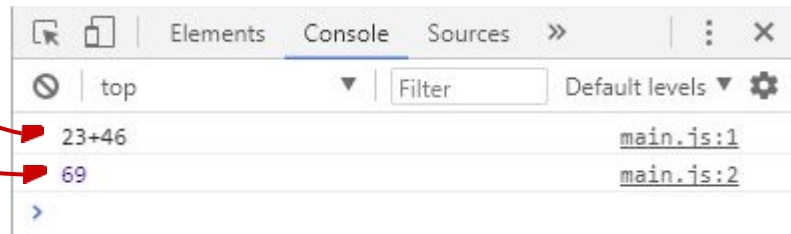
```
{
  "type": "Program",
  "body": [
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    },
    {
      "type": "EmptyStatement"
    }
  ],
  "sourceType": "script"
}
```

Вирази

Якщо порівнювати мову JavaScript із природною мовою, то **інструкції** – це фраза, **вирази** – результат висловлено фрази. Якщо **інструкції у програмі просто виконуються**, то **вирази завжди повертають якісь значення**.

Вираз це фрагмент коду, результатом якого є певна величина.

Якщо інтерпретатор бачить **вираз**, він обчислює його **значення** та замінює вираз **значенням**.



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.

Ось приклад виразу, який обчислює суму двох чисел:

```
var sum = 10 + 5
```

Вирази бувають **простими** та **складними**.

Прості вирази - це такі, які не містять в собі інших виразів. До простих виразів відносять:

- **ідентифікатори** (назви змінних);
- **літерали** (дані, які з'являються безпосередньо в програмі, наприклад, числа, рядки);
- деякі прості **слова** (наприклад, ключове слово **this**).

Всі ці елементи можуть використовуватися як вирази-інструкції (expression statements).

Parser produces the (beautiful) syntax tree

```
1 identifier
2 12345
3 this
```

Інструкції
висловлювання

Syntax

Tree

Tokens

Expand All Collapse All

- Program body [3]
 - ExpressionStatement
 - expression
 - Identifier
 - name: identifier
 - ExpressionStatement
 - expression
 - Literal
 - value: 12345
 - raw: 12345
 - ExpressionStatement
 - expression
 - ThisExpression

No error

Оператори

Оператори в JavaScript - це символи або слова, які використовуються для виконання операцій на одному або декількох операндах (значеннях або змінних) і повернення результату.

// 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 '/'

// Оператори порівняння

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

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

// 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

Типи операторів

Для того, щоб поєднати кілька простих виразів в одне складне, можна використовувати оператори.

Оператори бувають:

- унарні
- бінарні
- **тернарний** (три аргументи)

Розрізняються вони кількістю операндів.

Parser produces the (beautiful) syntax tree

```
1 10 + 20;
```

Бінарний
оператор

Лівий
літерал

Правий літерал

No error

Syntax

Tree

Tokens

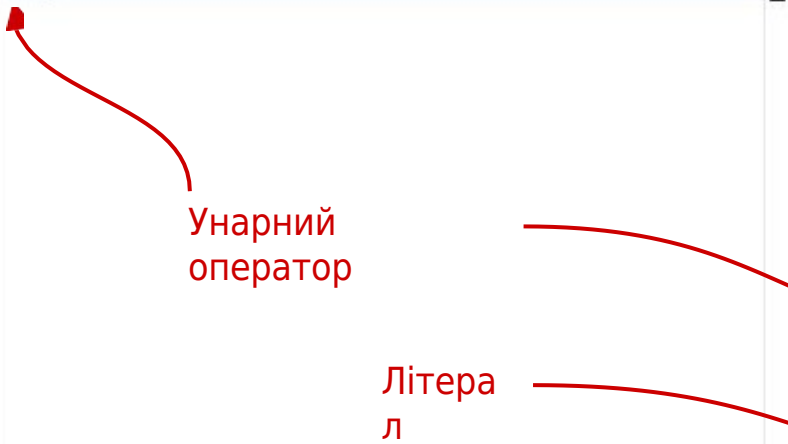
Expand All

Collapse All

- Program body [1]
 - ExpressionStatement
 - expression
 - BinaryExpression
 - operator: +
 - left
 - Literal
 - value: 10
 - raw: 10
 - right
 - Literal
 - value: 20
 - raw: 20

Parser produces the (beautiful) syntax tree

```
1 +20;
```



No error

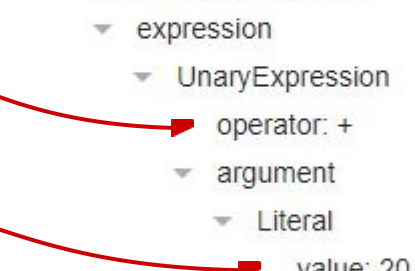
Syntax

Tree

Tokens

Expand All

Collapse All

- ▼ Program body [1]
 - ▼ ExpressionStatement
 - ▼ expression
 - ▼ UnaryExpression
 - operator: +
 - ▼ argument
 - ▼ Literal
 - value: 20
 - raw: 20
 - prefix: true
- 

Для того, щоб інструкція була корисною, в ній має бути хоча б один вираз із так званим **побічним ефектом** (side effect).

Наприклад:

10 + 20; це правильний вираз, помилки не буде,
інтерпретатор обчислить значення виразу і
відразу його забуде.

Якщо присво ти результат виразу змінній, наприклад:

```
num = 10 + 20;
```

цей вираз вже змінює щось поза нашого скрипта, оскільки це значення записується в оперативну пам'ять комп'ютера.

Тому можна сказати, що **оператор присвоювання – це оператор із побічним ефектом.**

Оператор із побічним ефектом — це оператор, який змінює стан програми або змінно під час свого виконання.

Щоб цей скрипт працював, попередньо необхідно **оголосити змінну**:

```
var num;
```

```
num = 10 + 20;
```

Якщо в інструкції немає жодного виразу з побічним ефектом, то в такій інструкції немає сенсу.

Змінні

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.

Для оголошення (створення змінно) використовується ключове слово

```
var:  
var message;
```

Інструкція з ключовим словом var називається **інструкцією оголошення** (declaration statement).

Після оголошення можна записати в змінну дані:

```
message = 'Hello';
```

```
var message;  
message = 'Hello';
```



Ці дані будуть збережені у відповідній області пам'яті і надалі доступні при зверненні на ім'я.

```
console.log(message); // виведе вміст змінної
```

Для стислості можна **поєднати** оголошення змінної та запис даних:

```
var message = 'Hello!';
```

Можна **оголосити кілька змінних** через кому:

```
var user, age, message;
```

а також **ініціалізувати** змінні:

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


Ініціалізацією називається створення первинних значень змінних.

Після того, як змінні оголошено і їм присвоєно значення, ми можемо звертатися до цих змінних будь-де в нашому скрипті.

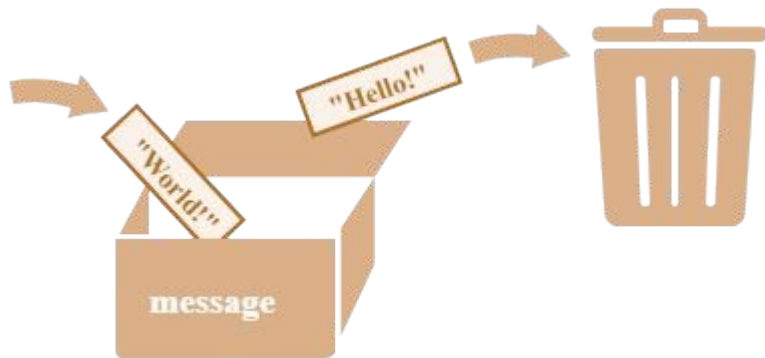
Ми можемо покласти будь-яке значення в коробку. Ми також можемо змінити його стільки разів, скільки захочемо:

```
var message;
```

```
message = "Hello!";
```

```
message = "World!"; // значение изменено
```

```
console.log(message);
```



Ідентифікатори в мові JavaScript можуть починатися з літери, символу

`_` або символу `$`:

```
var variable, _variable, $variable;
```

і не можуть починатися з цифри.

Однак далі на ім'я можна використовувати цифри:

```
var variable1, variable2, variable3;
```

Зарезервовані ключові слова

JavaScript (стандарт ECMA-262) визначає набір ключових слів (keywords) і зарезервованих слів для вирішення спеціалізованих завдань, таких як вказівка початку або кінця керуючої інструкції або виконання специфічної операції.

Ключові слова не можна використовувати як ідентифікатори та імена властивостей. Ось деякі з ключових слів: **break, case, catch, continue, debugger, default, delete, do, else, finally, for, function, if, in, instanceof, new, return, switch, this, throw, try, typeof, var, void, while, with.**

[Детальніше...](#)

Регістр ідентифікаторів

Мова JavaScript чутлива до регістру, тому, наприклад: **myvar** і **myVar** – це різні ідентифікатори.

Про це потрібно пам'ятати, коли ви даєте імена змінним та функціям.

Назва

ЗМІННИХ

Змінні починаються з маленько літери і далі всі йдуть у нижньому регістрі, окрім тих літер, які починають нові слова.

Цей стиль називається «[Верблюжа Нотація](#)» або **CamelCase** та прийнятий у мові JavaScript (також називається **lowerCamelCase**).

Наприклад, у мові PHP прийнято інший стиль: в іменах змінних слова пишуться в нижньому регістрі і розділяються символи

підкреслення:

newVariable = 345; – JavaScript

new_variable = 345; – PHP

Типи даних

JavaScript визначає 8 базових типів

1. String
2. Number
3. BigInt
4. Boolean
5. Undefined
6. Null
7. Symbol

8. Object

Примітив
и

Об'єкт
и

Примітиви (прості типи)

Сім простих типів даних

- числа
- `BigInt` (ES2020)
- рядки
- логічні значення
- тип `null`
- тип `undefined`
- `Symbol` (ES6 / ES2015)

```
var myNumber = 25;  
    myBigInt = 25000000000000n,  
    myString = 'Some string',  
    myBool   = true,  
    myNull   = null,  
    myUndef  = undefined,  
    mySym    = Symbol('name');
```

Числові та **рядкові** типи – звичні та зрозумілі нам.

BigInt – це спеціальний числовий тип, який дає змогу працювати з цілими числами довільно довжини.

Логічний (булевий) тип може приймати одне з двох значень: **true** чи **false**. Обидва типи **null** та **undefined** означає відсутність значення.

Тип даних **Symbol** – новий примітивний тип даних, що служить для створення унікальних ідентифікаторів.

У цій інструкції всі ініціалізують значення, крім **undefined** та **Symbol**, називаються літералами відповідного типу:

```
var myNumber = 25,  
    myBigInt = 25000000000000n,  
    myString = 'Some string',  
    myBool   = true,  
    myNull   = null,  
    myUndef  = undefined,  
    mySym    = Symbol('name');
```

числовий літерал

BigInt літерал

рядковий літерал

логічний літерал

літерал **null**

undefined – ідентифікатор

СИМВОЛ – єдиний тип у якого немає літералу

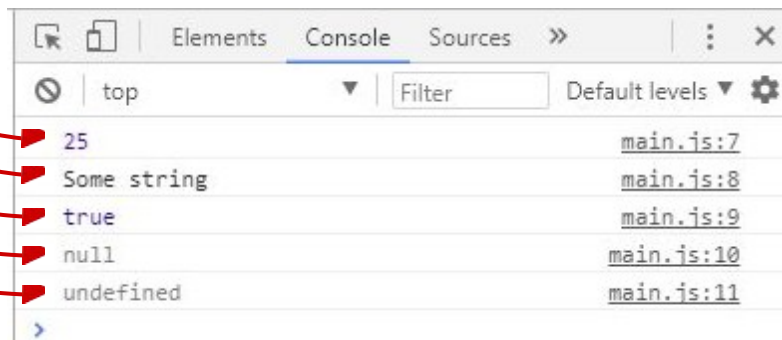
Використання примітивів

Після того як змінну оголошено і значення присвоєно, можна використовувати будь-де в нашому коді.

```
var message = 'Hello';  
alert(message);  
console.log(message);
```

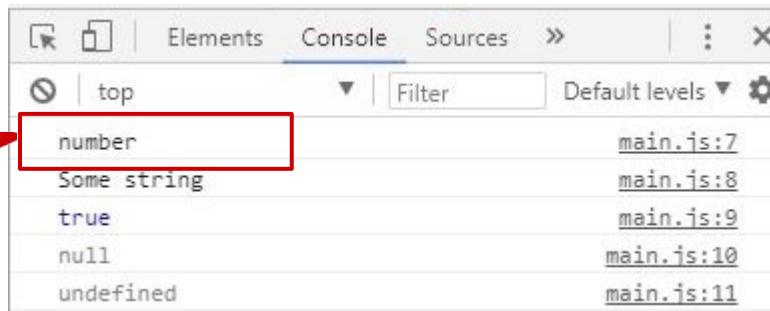
Наприклад, можна вивести значення змінних у консоль:

```
var myNumber = 25;  
var myString = 'Some string';  
var myBool = true;  
var myNull = null;  
var myUndef = undefined;  
console.log(myNumber);  
console.log(myString);  
console.log(myBool);  
console.log(myNull);  
console.log(myUndef);
```



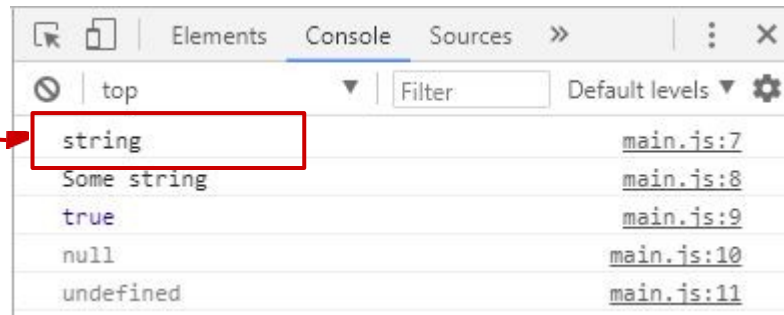
Для визначення типу вираження у JavaScript є унарний оператор **typeof**:

```
console.log(typeof myNumber);
```



Сам оператор повертає рядок:

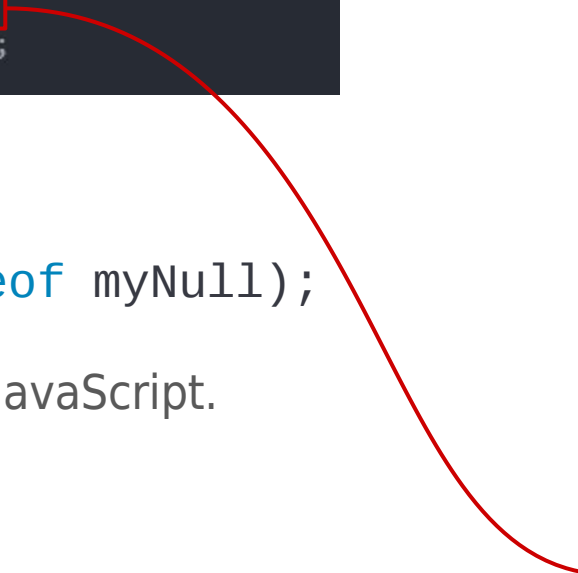
```
console.log(typeof typeof myNumber);
```



```
index.html x main.js
1
2 var myNumber = 25,
3   myString = 'Some string',
4   myBool = true,
5   myNull = null,
6   myUndef = undefined;
7
8 console.log(typeof myNumber);
9 console.log(typeof myString);
10 console.log(typeof myBool);
11 console.log(typeof myNull);
12 console.log(typeof myUndef);
13
```

`console.log(typeof myNull);`

Це відома помилка JavaScript.



top	
number	main.js:7
string	main.js:8
boolean	main.js:9
object	main.js:10
undefined	main.js:11

Об'єкти (об'єктні типи)

До **об'єктних типів даних** (об'єктів) відносяться будь-які значення, які не належать до семи простих типів:

- **об'єкти**
- **масиви**
- **функції**
- **регулярні вирази ***
- **об'єкт помилки**
- **колекції: Maps, Sets, WeakMaps, WeakSets**

```
var object = {name: 'Name'},  
    array  = [1, 2, 3],  
    func   = function () {},  
    regex  = /w+/g,  
    error  = new Error();  
    map    = new Map(),  
    set    = new Set();
```

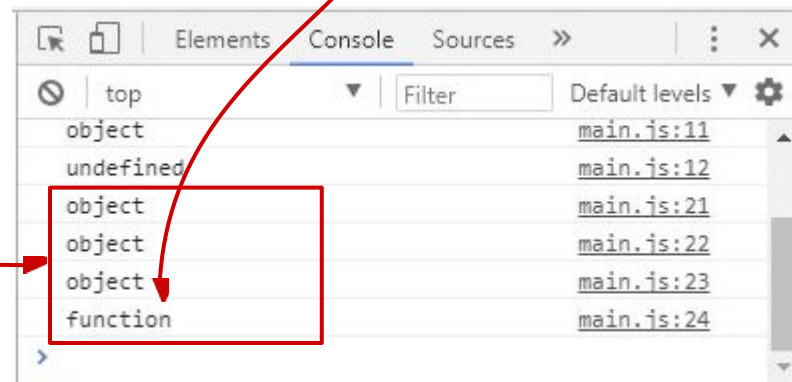
**регулярні вирази* – це шаблони, які використовуються для зіставлення послідовностей символів у рядках.

```

1
2 var myNumber = 25,
3     myString = 'Some string',
4     myBool = true,
5     myNull = null,
6     myUndef = undefined;
7
8 console.log(typeof myNumber);
9 console.log(typeof myString);
10 console.log(typeof myBool);
11 console.log(typeof myNull);
12 console.log(typeof myUndef);
13
14 // объектные типы
15
16 var obj = {name: "Somename"},
17     array = [1, 2, 3],
18     regexp = /w+/g,
19     func = function(){};
20
21 console.log(typeof obj);
22 console.log(typeof array);
23 console.log(typeof regexp);
24 console.log(typeof func);

```

Це лише одна особливість оператора **typeof**, про яку потрібно знати.



JavaScript – мова з динамічною типізацією, це означає, що **тип змінної визначається автоматично** інтерпретатором залежно від значення, що присвоєно їй.

JavaScript має **автоматичну конвертацію** (перетворення) **типів**.

Наприклад, якщо інтерпретатор в якомусь місці програми очікує побачити значення логічного типу, будь-яке підставлене туди значення буде автоматично приведено до логічного типу.

```
var res = 'string1';      console.log(res);    // 'string1'
res = 1;                  console.log(res);    // 1
res = res + 2;            console.log(res);    // 3
res = res + 'string2';    console.log(res);    // '3string2'
//      3      + '3string2'    =>      3 + NaN => NaN
//      '3'    + '3string2'    =>      '3string2'
```

Докладніше розглянемо
пізніше.

Змінні та незмінні типи

Значення в мові JavaScript можна розділити на **змінні** (**mutable**) та **незмінні** (**immutable**).

Усі **примітивні типи** в JavaScript є **незмінними**.

Це означає, що їх значення не можуть бути змінені після створення.

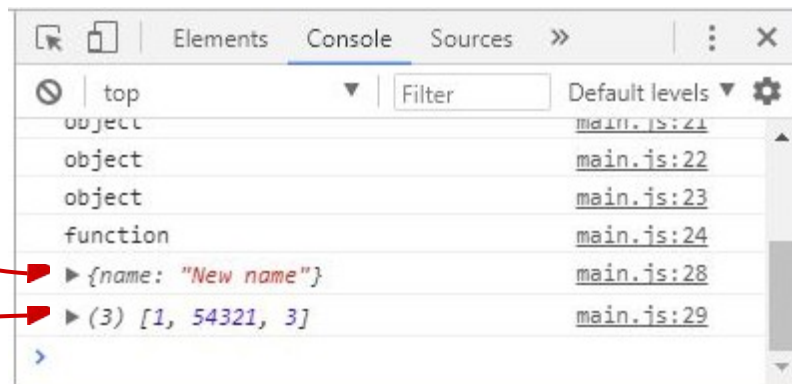
Об'єктні типи, які включають в себе об'єкти, масиви і функції, є **змінними**, оскільки їх стан або вміст може бути змінений.

Спробуємо змінити об'єкт:

```
var obj = {};  
obj.name = 'New name';  
console.log(obj);
```

та масив:

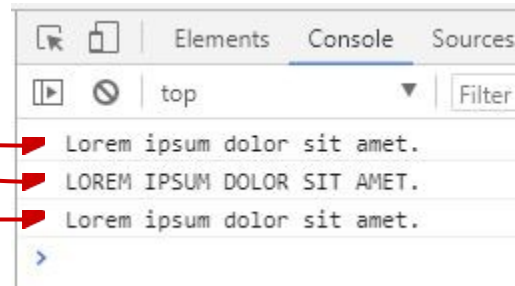
```
var array = [1, 2, 3];  
array[1] = 54321;  
console.log(array);
```



Таким чином, ми змінили об'єкт та масив.

Спробуємо змінити рядок, використовуючи метод, який перетворює його до верхнього регістру:

```
var myString = 'Lorem ipsum dolor sit amet.';  
console.log(myString);  
console.log(myString.toUpperCase());  
console.log(myString);
```



Як бачимо, вихідний рядок залишається незмінним, а повертається нам інший рядок.

Висновки

- **Об'єкти** за замовчуванням є **змінними**. Їх можна **модифікувати**.
- **Об'єкти** мають унікальні ідентифікатори і **порівнюються за посиланням**, а не значенням.
- Змінні містять посилання на об'єкти.
- **Примітивні типи** (строки, числа, булеві) є **незмінними**. Їх не можна безпосередньо модифікувати.
- **Примітиви порівнюються за значенням**. У них немає унікальних ідентифікаторів.

Оператор присвоєння

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

Побічний ефект у програмуванні - це будь-яка зміна стану програми, що відбувається в результаті виконання інструкції , окрім 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 = b + 2 // Оператор присвоєння з додаванням. Значення 'b'  
          збільшується на 2, що також є побічним ефектом
```

Number

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

Діапазон безпечних значень

Безпечний діапазон значень для типу **Number** в JavaScript знаходиться між **-(2⁵³ - 1)** та **2⁵³ - 1**, включно.

```
> Number.MAX_SAFE_INTEGER
```

```
< 9007199254740991
```


Числовий літерал

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

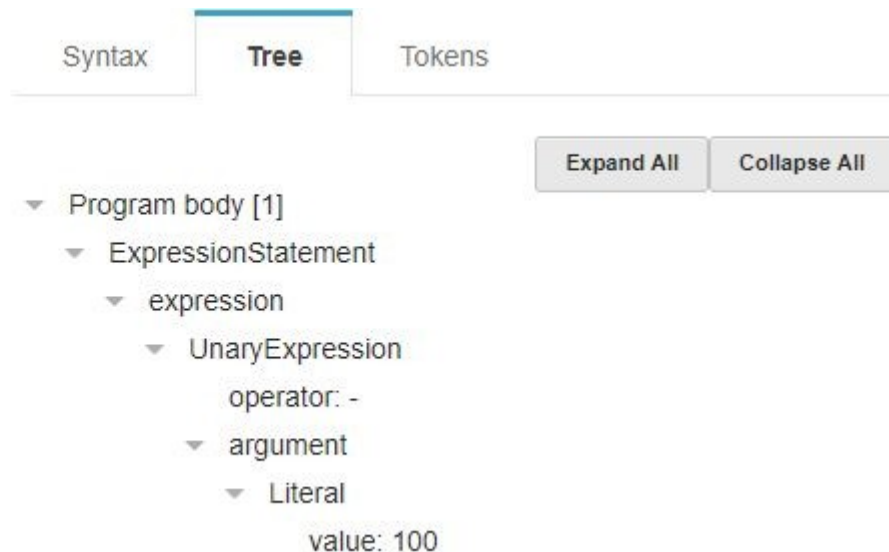
Конструктор Number

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

Негативних числових літералів немає.

Якщо ми запишемо якесь негативне число, то побачимо, що **знак мінус** поводить ся як **унарний оператор**.

```
1 -100
```



Крім десяткових цілих літералів, **JavaScript розпізнає шістнадцяткові значення.**

Шістнадцяткові літерали починаються з послідовності символів **0x**, за якою слідує рядок шістнадцяткових цифр. Шістнадцяткова цифра – це одна з цифр від **0** до **9** або літер від **A** до **F**, що мають значення від 10 до 15:

```
var a = 255;  
var b = 0xFF; // Число 255 у шістнадцятковій системі числення
```

Восьмирічні літерали в сучасному JavaScript повинні починатися з **0o** або **0O**:

```
var c = 0o377; // Число 255 у восьмирічній системі числення.
```

Літерали дійсних чисел можуть також представлятися в експоненційній нотації: дійсне число, за яким слідує буква е (або Е), а потім необов'язковий знак плюс або мінус і ціла експонента.

Така форма запису означає речовинне число, помножене на 10 ступеня, що визначається значенням експоненти:

```
var a = 16.75;
```

```
var b = 2e4;           // 2 * 10^4 = 20 000
```

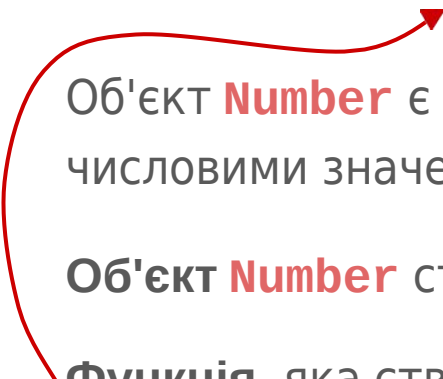
У мові JavaScript для простих типів є об'єкти обгортки (wrapper objects), які

надають ширші можливості для роботи з цими типами.

Для того, щоб **створити** такий об'єкт, ми будемо використовувати конструктор **Number ()**:

```
var N = new Number(5000);
```

```
var N = new Number(5000);
```



Об'єкт **Number** є обгорткою, що дозволяє нам працювати з числовими значеннями.

Об'єкт **Number** створюється через конструктор **Number ()**.

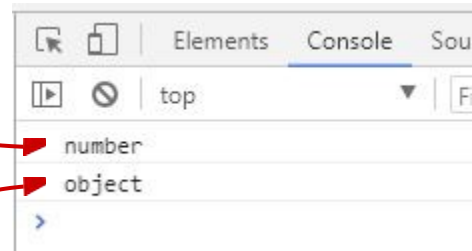
Функція, яка створює нові об'єкти, називається **конструктором** і прийнято писати **з великої літери**.

Якщо ми присвоюємо змінній якесь число, то така змінна матиме **числовий тип**:

```
var n = 5000;  
console.log(typeof n); // number
```

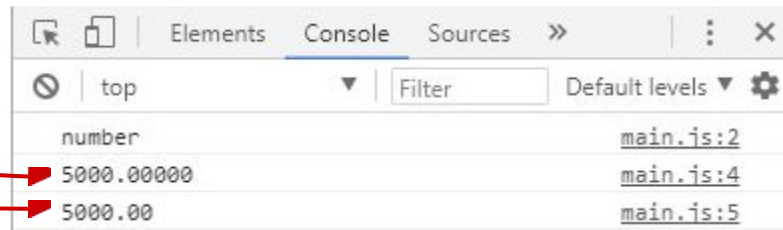
За допомогою конструктора **Number()** ми створюємо об'єкт і така змінна буде **об'єктом**:

```
var N = new Number(5000);  
console.log(typeof N); // object
```



Наприклад, за допомогою методу **toFixed()** ми можемо вивести встановлену кількість **знаків після коми**:

```
var N = new Number(5000);  
console.log(N.toFixed(5));  
console.log(N.toFixed(2));
```



Той самий метод можна викликати не тільки в об'єкта, а й у **змінної простого типу** і навіть у **числового літералу**:

```
var n = 3000;  
console.log(n.toFixed(3));  
console.log(2.toFixed(4));  
console.log(2..toFixed(4));
```



Методи **можна викликати тільки в об'єктів**, але ми переконалися, що можемо викликати їх у простих змінних і навіть у літералів.

Це можливо тому, що інтерпретатор створює відповідні **об'єкти обгортки «на льоту»**.

Наприклад, коли інтерпретатор обчислює значення виразу **n.toFixed(3)**, він бере значення змінної **n**, створює об'єкт **Number** з цим значенням, потім викликає метод **toFixed()** цього об'єкта та значення, яке повертає цей метод і буде значенням всього виразу.

Сам об'єкт **Number** знищується відразу, як інтерпретатор обчислить значення висловлювання.

Саме з цієї причини **прості значення можуть поводитися як об'єкти**, а не тому, що вони є об'єктами.

Саме тому конструктор **Number()** використовується досить рідко.

Крім методу **toFixed()** у об'єкта **Number** є метод **toExponential()** для переведення в експоненціальну форму, який як параметр приймає кількість цифр після точки:

```
var n = 3000;  
console.log(n.toExponential(4));    // 3.0000e+3
```

і метод **toPrecision()** для виведення з певною точністю:

```
var n = 15.89;  
console.log(n.toPrecision(3));      // 15.9
```

BigInt

BigInt - це новий тип даних в JavaScript, який був введений у стандарті ECMAScript 2019. Він призначений для роботи з цілими числами, які виходять за межі діапазону, що підтримується типом **Number**.

На відміну від типу **Number**, який використовує формат з плаваючою комою для зберігання чисел, **BigInt** використовує цілочисельне представлення. Це дозволяє представляти числа з точністю до довільно довжини.

// Приклад, коли тип Number не впорається і потрібно використовувати BigInt:

```
const maxNumber = Number.MAX_SAFE_INTEGER;  
console.log(maxNumber); // 9007199254740991
```

```
const bigNumber = BigInt(maxNumber);  
console.log(bigNumber); // 9007199254740991n
```

```
let moreMaxNumber = maxNumber + 1;  
moreMaxNumber = moreMaxNumber + 1;  
console.log(moreMaxNumber); // 9007199254740993 => 9007199254740992
```

```
let biggerNumber = bigNumber + 1n;  
biggerNumber = biggerNumber + 1n;  
console.log(biggerNumber); // 9007199254740993n
```