Зміст

[Коментарі в JavaScript 3](#_Toc176483327)

[Коментарі в JavaScript: синтаксис 3](#_Toc176483328)

[Чому написання коментарів надає сили 4](#_Toc176483329)

[Крапка з комою в JavaScript 4](#_Toc176483330)

[Автоматична вставка крапки з комою (ASI) 5](#_Toc176483331)

[Примітка щодо використання консолі в інструментах розробника у вашому браузері 5](#_Toc176483332)

[Якщо вам потрібно ввести кілька рядків коду, перш ніж запускати їх, не забудьте натиснути комбінацію клавіш SHIFT + ENTER, щоб перейти до наступного рядка. 6](#_Toc176483333)

[Вивести привітання в консоль 6](#_Toc176483334)

[Виведіть кілька слів у консоль 8](#_Toc176483335)

[1. Додаткові оператори 9](#_Toc176483336)

[Логічний оператор І в JavaScript:  && 9](#_Toc176483337)

[Логічний оператор АБО в JavaScript:  || 10](#_Toc176483338)

[Логічний оператор НЕ:  ! 11](#_Toc176483339)

[Оператор модуля:  % 12](#_Toc176483340)

[Оператор рівності  == 13](#_Toc176483341)

[Оператор суворої рівності  === 13](#_Toc176483342)

[Оператор нерівності  != 13](#_Toc176483343)

[Оператор суворої нерівності  !== 13](#_Toc176483344)

[2. Використання  операторів +  для рядків і чисел 14](#_Toc176483345)

[Об’єднання двох рядків за допомогою   оператора + 14](#_Toc176483346)

[Об’єднання рядків і чисел за допомогою   оператора + 14](#_Toc176483347)

[Оператор присвоювання додавання,  += 15](#_Toc176483348)

[Оператор присвоєння конкатенації,  += 15](#_Toc176483349)

[Пріоритет оператора та асоціативність 16](#_Toc176483350)

[Цикли та вкладені цикли 17](#_Toc176483351)

[Застосування циклів 19](#_Toc176483352)

[Деякі додаткові приклади 21](#_Toc176483353)

[Побудова та виклик функцій 22](#_Toc176483354)

[Об’єктні літерали та крапкова нотація 25](#_Toc176483355)

[Літерали об’єктів і крапкова нотація 25](#_Toc176483356)

[Об’єктні літерали та позначення в дужках 27](#_Toc176483357)

[Літерали об’єктів і позначення в дужках 27](#_Toc176483358)

[Масиви є об’єктами 29](#_Toc176483359)

[Математичні функції 30](#_Toc176483360)

[Числові константи 30](#_Toc176483361)

[Методи округлення 30](#_Toc176483362)

[Методи арифметики та числення 30](#_Toc176483363)

[Робота з рядками 31](#_Toc176483364)

[Методи об’єктів 32](#_Toc176483365)

[Синтаксичні, логічні помилки та помилки виконання 35](#_Toc176483366)

[ReferenceError 35](#_Toc176483367)

[SyntaxError 35](#_Toc176483368)

[TypeError 36](#_Toc176483369)

[RangeError 36](#_Toc176483370)

[Повернення значень із функцій 37](#_Toc176483371)

[Парадигма функціонального програмування 38](#_Toc176483372)

[Навчальні цілі 38](#_Toc176483373)

[Першокласні функції 40](#_Toc176483374)

[Чисті функції та побічні ефекти 42](#_Toc176483375)

[Принципи об'єктно-орієнтованого програмування 42](#_Toc176483376)

[Переваги ООП 42](#_Toc176483377)

[Принципи ООП 43](#_Toc176483378)

[Принципи ООП: Спадкування 43](#_Toc176483379)

[Принципи ООП: інкапсуляція 44](#_Toc176483380)

[Принципи ООП: абстракція 44](#_Toc176483381)

[Принципи ООП: поліморфізм 45](#_Toc176483382)

[Конструктори 47](#_Toc176483383)

[Створення класів 48](#_Toc176483384)

[Параметри за замовчуванням 59](#_Toc176483385)

[Розробка ОО програми 61](#_Toc176483386)

[Кодування   класу Animal 64](#_Toc176483387)

[Цикли for 68](#_Toc176483388)

[Ви дізнаєтесь, як концептуально працює цикл for of. 68](#_Toc176483389)

[Вбудовані методи 69](#_Toc176483390)

[Метод Object.keys  (). 69](#_Toc176483391)

[Метод Object.values (). 69](#_Toc176483392)

[Метод Object.entries (). 69](#_Toc176483393)

[Приклади 70](#_Toc176483394)

[Приклади літералів шаблонів 71](#_Toc176483395)

[Що таке літерали шаблону? 71](#_Toc176483396)

[Відмінності шаблону від звичайного рядка 72](#_Toc176483397)

[Приклади структур даних 73](#_Toc176483398)

[Робота з масивами в JavaScript 73](#_Toc176483399)

[Метод forEach (). 73](#_Toc176483400)

[Метод filter(). 74](#_Toc176483401)

[Метод карти​ 75](#_Toc176483402)

[Робота з об'єктами в JavaScript 75](#_Toc176483403)

[Робота з картами в JavaScript 76](#_Toc176483404)

[Робота з наборами в JavaScript 76](#_Toc176483405)

[Інші структури даних у JavaScript 77](#_Toc176483406)

[Використання Spread і Rest 77](#_Toc176483407)

[Об’єднання масивів, об’єктів за допомогою оператора rest 77](#_Toc176483408)

[Додайте нових членів до масивів без використання методу push(). 78](#_Toc176483409)

[Перетворення рядка в масив за допомогою оператора поширення Spread 78](#_Toc176483410)

[Копіювання об'єкта за допомогою оператора spread 78](#_Toc176483411)

**Коментарі в JavaScript**

Я вибрав коментарі як відправну точку з двох причин:

1. Їхній синтаксис - те, як написані коментарі, легко зрозуміти.
2. Написання коментарів може розширити ваші можливості як розробника.

Спочатку я поясню синтаксис, а після цього я обговорю, чому можливість писати коментарі є таким сильним.

**Коментарі в JavaScript: синтаксис**

У JavaScript є два різновиди коментарів:

1. Однорядкові коментарі
2. Багаторядкові коментарі

Однорядковий коментар створюється, коли ви додаєте один за одним два символи косої риски без пробілів.

// це коментар!

Усе, що слідує за однорядковим коментарем у JavaScript, браузер ігнорує.

Це означає, що, по суті, ви можете написати будь-який текст, код, символи, емодзі, що завгодно – і браузер проігнорує це.

Багаторядковий коментар, як свідчить його назва, охоплює кілька рядків коду та створюється за допомогою скісної риски та зірочки. Наприклад:

/\*

це

є

a

багаторядковий

коментар

\*/

Ви також можете використовувати синтаксис багаторядкового коментаря в одному рядку коду, як показано нижче:

/\* це багаторядковий коментар в одному рядку \* /

**Чому написання коментарів надає сили**

У цьому курсі передбачається, що ви ніколи не писали жодного рядка коду JavaScript.

Зважаючи на це припущення, розгляньте наслідки того, що ви щойно дізналися, тобто наслідки навчання написанню коментарів у JavaScript:

1. Тепер ви можете вільно висловлювати свої ідеї щодо будь-якого коду, який ви пишете.
2. Ви можете додати коментарі до будь-якого коду, який уже існує.
3. Ці коментарі можуть бути призначені для вас у майбутньому або для колег із вашої команди розробників.

Отже, коментарі дають змогу розширити можливості, оскільки вони полегшують спілкування з вами в майбутньому або з членами вашої команди, дозволяючи вам ставити запитання щодо коду, позначати код як «для виконання», «для покращення» або просто пояснювати, що даний фрагмент коду робить.

Крім того, ви навіть можете закоментувати деякий робочий код у файлі JavaScript, щоб запобігти його запуску.

По суті, коментарі дозволяють «вимкнути» фрагменти коду JavaScript.

На це може бути багато причин:

1. Спроба зрозуміти, як працює певний фрагмент коду.
2. Тестування різних рішень проблеми кодування - при цьому не потрібно видаляти існуючий код.
3. Налагодження – спроба точно визначити, чому ваш код несправний або поводиться непередбачувано.

**Крапка з комою в JavaScript**

У JavaScript крапка з комою -  **;**  використовується для чіткого відмежування частин коду від деяких інших частин коду.

**Автоматична вставка крапки з комою (ASI)**

Цікаво, що браузер має функцію, відому як «Автоматична вставка крапки з комою» — це означає, що він досить добре «заповнює пропуски» у випадку, якщо крапка з комою пропущена там, де вона повинна бути.

По суті, для розробників це означає, що здебільшого немає різниці, додано крапку з комою чи ні, оскільки браузер, швидше за все, це розбере.

Ось чому деякі розробники кажуть, що вам взагалі не варто морочитися з додаванням крапки з комою.

Однак інші розробники стверджують, що краще використовувати його скрізь, де це потрібно - для ясності.

Правда полягає в тому, що здебільшого ви можете думати про додавання крапки з комою в JavaScript як про необов’язковість – і певну стилістичну перевагу.

**Примітка щодо використання консолі в інструментах розробника у вашому браузері**

Як уже згадувалося раніше в цьому курсі, однією з причин такої популярності JavaScript є його доступність.

Щоб розпочати роботу з JavaScript, вам потрібен лише браузер. У цьому курсі я буду використовувати Google Chrome.

Після того, як ви встановили браузер і запустили його, клацніть правою кнопкою миші на поточній активній веб-сторінці та виберіть  команду **Перевірити**  в контекстному меню, що відкривається.

Це відкриє Інструменти розробника, а потім ви зможете клацнути вкладку «Консоль», щоб відкрити консоль, або, натиснувши клавішу  **ESC**  , ви вмикаєте та вимикаєте консоль незалежно від поточної активної панелі Інструментів розробника.

Ви можете ввести будь-яку команду JavaScript на консоль DevTools.

**Якщо вам потрібно ввести кілька рядків коду, перш ніж запускати їх, не забудьте натиснути комбінацію клавіш SHIFT + ENTER, щоб перейти до наступного рядка.**

Зверніть увагу на різницю між натисканням  клавіші **ENTER**  для запуску введеного вами коду JavaScript і натисканням  комбінації клавіш **SHIFT + ENTER**  для переходу до наступного рядка коду (замість запуску коду, який ви вже ввели).

Це все, що вам потрібно, щоб почати писати код JavaScript!

У наступних уроках не соромтеся дотримуватися будь-якого з двох способів:

1. Використання редактора коду VS і розширення Code Runner, як описано раніше
2. Використання самого браузера Chrome і запуск коду в консолі DevTools, як описано в цій статті

**Вивести привітання в консоль**

Тепер, коли ви знаєте, як перейти на вкладку «Консоль інструментів розробника», давайте використаємо її для запуску вашого першого фрагмента справжнього коду JavaScript.

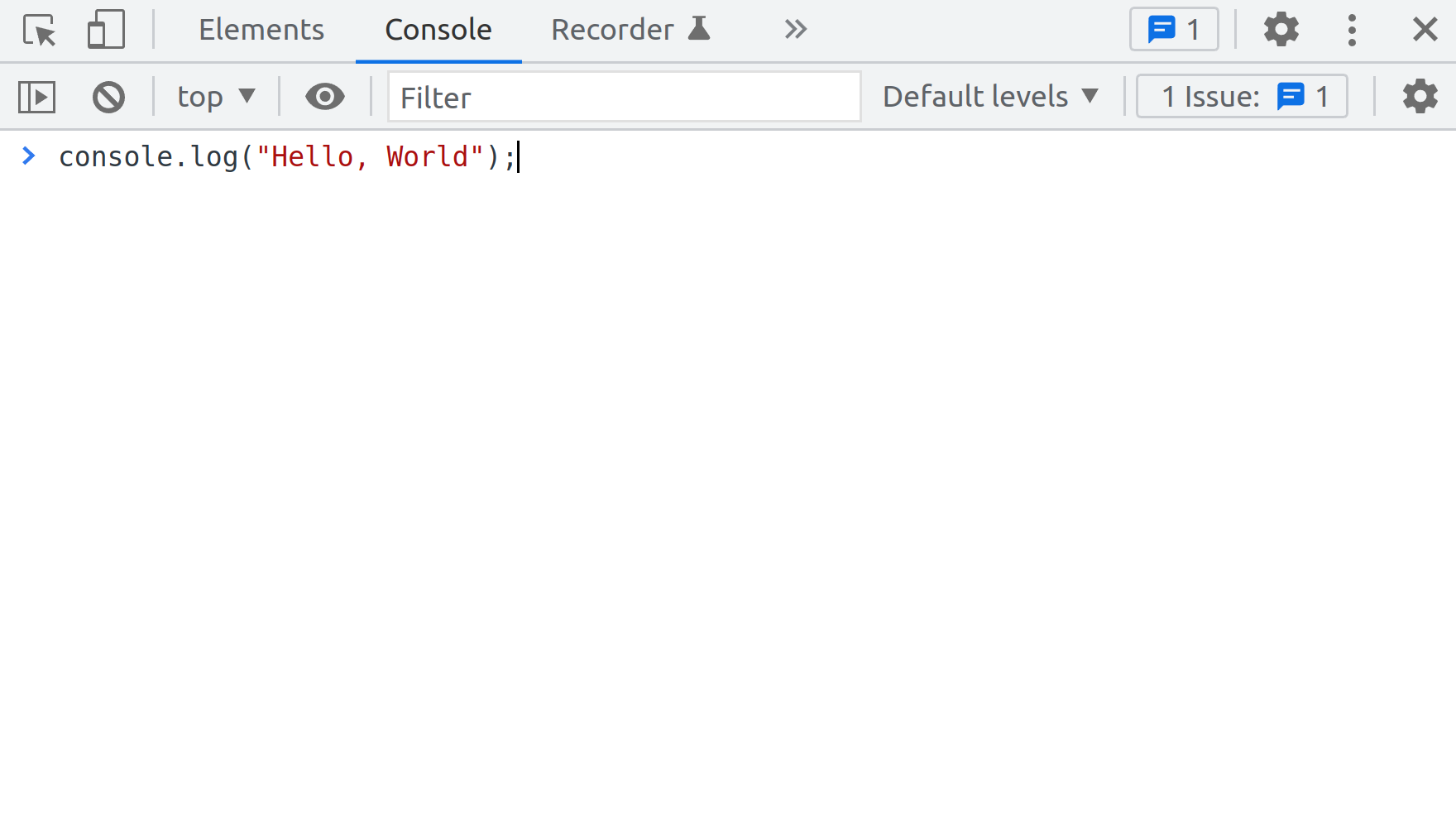
У Chrome, відкривши інструменти розробника, клацніть порожнє місце на вкладці консолі праворуч від синього  символу **>**  . Ви повинні побачити миготливу вертикальну лінію (також відому як «курсор»). Курсор означає, що ви можете вводити текст у консолі.

Якщо ви введете дійсний код JavaScript, він буде виконано, що означає: він буде оброблений і може призвести до певного результату.

Тепер ви використовуєте  функцію **console.log**  для виведення слів «Hello, World».

Для цього введіть у консоль таку команду:

console.log( "Привіт, світ" );



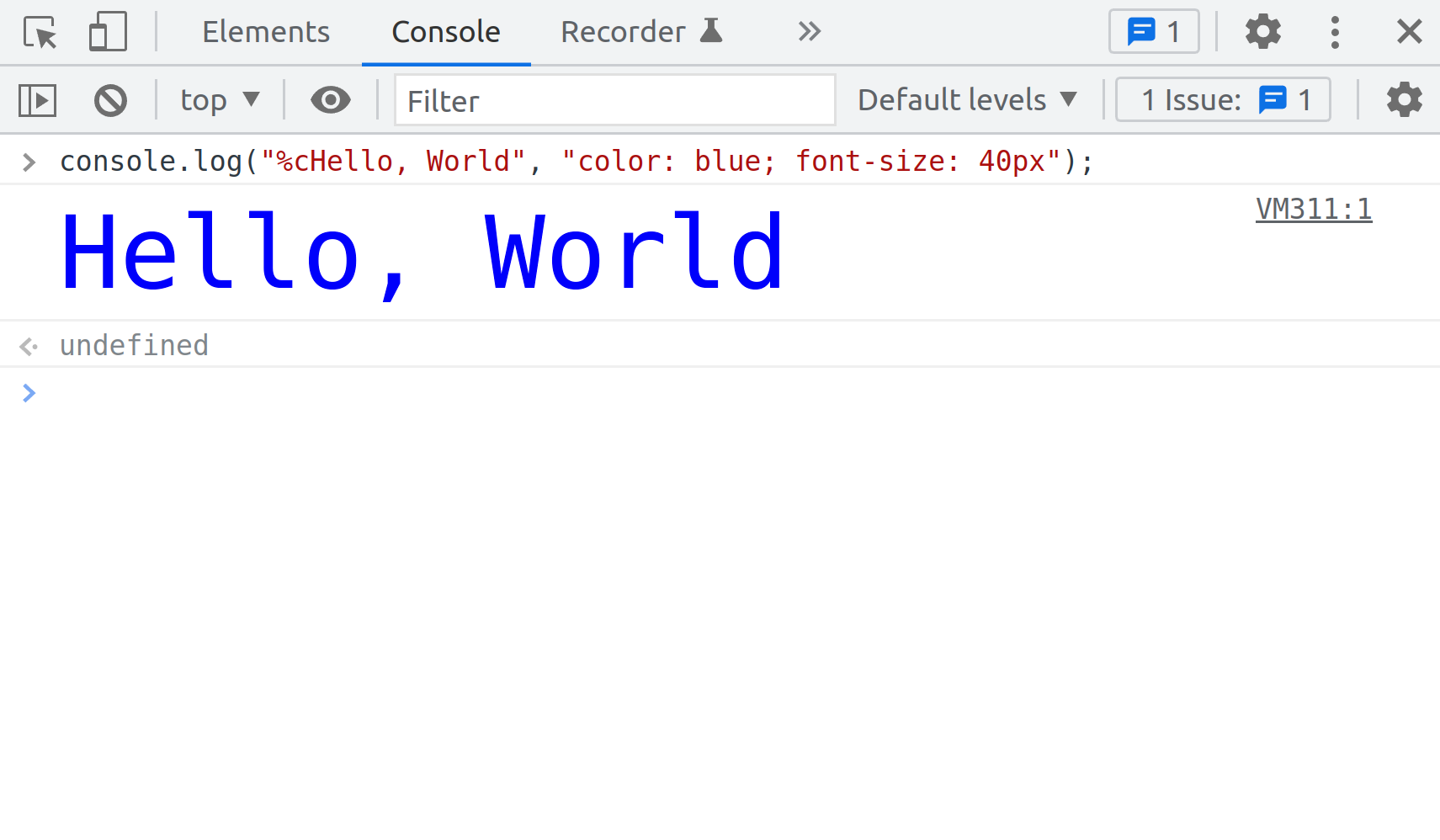
Якщо ви все зробили згідно з інструкцією, в консоль повинні вийти слова «Hello, World».

Ось ще одна, більш складна команда, щоб показати вам, що  команда **console.log**  має ряд хитрощів.

Наприклад, чи знаєте ви, що ви можете стилізувати вивід у консолі?

У цьому фрагменті коду є кілька доповнень: інший розмір шрифту та синій колір:

console.log( "%cHello, World" ,  "color: blue; font-size: 40px" );

журнал консолі з відсотком c авторів

Якщо ви скопіюєте та вставите цей фрагмент коду або, можливо, просто введете його в консоль, як тільки ви натиснете клавішу **ENTER** , щоб запустити його, ви отримаєте слова «Hello, World» виведені на консоль. Однак цього разу колір літер буде синім, а розмір шрифту – 40 пікселів. Отже, ви щойно навчилися гарному трюку з консоллю.

Якщо ви додасте  **%c**  відразу після  символу **"**  , ви зможете стилізувати вихідні дані консолі, додавши символ  **,**  після другого символу  **"** , а потім, всередині іншої пари  символів **"**  і  **"**  , використайте дійсний код CSS, щоб стилізувати слова, які ви хочете вивести на консоль.

Причиною, щоб показати вам цей маленький трюк, було, сподіваюся, мотивувати вас потренуватися в написанні різних слів у  команді **console.log**  і використати свої навички CSS, щоб змінити стиль цих слів у виведенні консолі. Таким чином, вам може бути цікаво практикувати цю щойно набуту навичку, а навчання та розвага завжди чудово поєднуються.

**Виведіть кілька слів у консоль**

Щоб вивести кілька слів на консоль, ви можете об’єднати їх за допомогою символу **+** , офіційно відомого як «оператор конкатенації», коли ми з’єднуємо фрагменти тексту, або «оператор додавання» для виконання математичної операції додавання двох чисел. .

Ось приклад об’єднання трьох окремих фрагментів тексту: **console.log("Hello " + "there, " + "World")** . Результатом цієї команди буде: *Hello there, World* .

Ось приклад виведення трьох окремих фрагментів тексту з використанням символу , замість цього:

**console.log("Привіт ", "тут, ", "Світ")**

Результатом цієї команди буде: *Hello there, World* .

**1. Додаткові оператори**

* Логічний оператор І:  **&&**
* Логічний оператор АБО:  **||**
* Логічний оператор НЕ:  **!**
* Оператор модуля:  **%**
* Оператор рівності: **==**
* Оператор суворої рівності: **===**
* Оператор нерівності: **!=**
* Оператор суворої нерівності: **!==**
* Оператор присвоювання додавання:  **+=**
* Оператор присвоєння конкатенації:  **+=**  (він такий же, як і попередній – про це пізніше)

**Логічний оператор І в JavaScript:  &&**

Логічний  **оператор AND**  , наприклад, використовується для підтвердження того, чи кілька порівнянь повернуть true.

У JavaScript цей оператор складається з двох символів амперсанда разом:  **&&** .

Припустімо, вам доручено розробити якийсь код, який перевірить, чи  змінна **currentTime**  знаходиться між  **9 ранку**  та  **5 вечора** . Код повинен мати console.log  **true**  , якщо  **currentTime > 9 і**  якщо  **currentTime < 17** .

Ось рішення:

var currentTime = 10;

console.log(currentTime > 9 && currentTime < 17);

Як працює цей код?

Спочатку в першому рядку я встановлюю  змінну **currentTime** і призначаю  їй  значення  **10 .**

Далі у другому рядку I console записує два порівняння:

**поточний час > 9**

**поточний час < 17**

Я також використовую  логічний оператор **&&**  , щоб об’єднати два порівняння.

По суті, мій код інтерпретується так:

console.log(10 > 9 && 10 < 17);

Порівняння  **10 > 9**  поверне  **true** .

Крім того, порівняння  **10 < 17**  поверне  **true** .

Це означає, що я можу далі переписати другий рядок свого рішення наступним чином:

console.log(true && true);

По суті, ось як працює мій код.

Тепер постає питання, яким буде результат  **console.log(true && true)** ?

Щоб зрозуміти відповідь, вам потрібно знати поведінку  логічного оператора **&&**  .

Логічний  оператор **&&**  повертає єдине значення: логічне значення  **true**  або  **false** відповідно до таких правил:

* Він повертає  **істину**  , якщо обидва значення праворуч і ліворуч оцінюються як  **істинні**
* Він повертає  **false**  у всіх інших випадках

Іншими словами:

**console.log(true && true)**  виведе:  **true**

**console.log(true && false)**  виведе:  **false**

**console.log(false && true)**  виведе:  **false**

**console.log(false && false)**  виведе:  **false**

**Логічний оператор АБО в JavaScript:  ||**

Логічний оператор АБО в JavaScript складається з двох вертикальних символів:  **||** .

Він використовується, коли ви хочете перевірити, чи принаймні одне з наведених порівнянь має значення  **true** .

Розглянемо наступне завдання: Вам потрібно написати програму на JavaScript, яка повертатиме  **true,**  якщо значення  змінної **currentTime**  не буде між  **9**  і  **17** . Інакше кажучи, ваш код має мати console.log  **true,**  якщо значення змінної  **currentTime**  менше  **9**  або більше  **17** .

Ось рішення:

var currentTime = 7;

console.log(currentTime < 9 || currentTime > 17);

У першому рядку коду я присвоюю   змінній  **currentTime число 7** .

У другому рядку я консольно реєструю результат перевірки того, чи  **currentTime < 9**  або  **currentTime > 17**  буде оцінено як  **true** .

Це те саме, що:

var currentTime = 7;

console.log(true || false);

Ось правила того, як  **||**  оператор обчислює задані значення:

**console.log(true || true)**  виведе:  **true**

**console.log(true || false)**  виведе:  **true**

**console.log(false || true)**  виведе:  **true**

**console.log(false || false)**  виведе:  **false**

Логічний оператор АБО завжди повертатиме  **істину** , за винятком випадків, коли обидві сторони оцінюють як  **хибність** . Іншими словами, щоб логічний оператор АБО повернув  **false** , результати обох порівнянь  **повинні**  повернути false.

Повертаючись до прикладу перевірки, чи  **currentTime < 9**  або  **currentTime > 17** , це має сенс: єдиний раз, коли ви отримаєте  **false,**  це коли значення, що зберігається в  змінній **currentTime**  більше  **9**  і менше  **17** .

**Логічний оператор НЕ:  !**

У JavaScript символом логічного оператора НЕ є знак оклику:  **!** .

Ви можете подумати про  **!**  оператор як перемикач, який перевертає оцінене логічне значення з  **істинного**  на  **хибне**  та з  **хибного**  на  **істинне** .

Наприклад, якщо я присвою логічне значення **true** змінній **petHungry** :

**var petHungry = true;**

...тоді я можу втішити те, що тварина більше не голодна, використовуючи **!** оператор, щоб перевернути логічне значення, що зберігається всередині змінної **petHungry** , ось так:

**console.log('Годування вихованця'); console.log("Тварина голодна: ", !petHungry); console.log(petHungry);**

Це результат наведеного вище коду:

Pet is hungry: true

Feeding the pet

Pet is hungry: false

true

Причиною зміни виведення в консолі є те, що ви перевернули значення, що зберігається всередині  змінної **petHungry**  , з  **true**  на  **false** .

Однак зауважте, що код у п’ятому рядку наведеного вище прикладу все ще виводить  **true**  – це пов’язано з тим, що я не перепризначив значення  змінної **petHungry**  .

Ось як я можу назавжди змінити значення, що зберігається в  змінній **petHungry,**  з  **true**  на  **false** :

var petHungry = true;

petHungry = !petHungry;

У цьому прикладі я спочатку присвоюю значення  **true**  новій змінній  **petHungry** . Потім у другому рядку я присвоюю протилежне значення,  **!true**  — читай: не вірно — існуючій  змінній **petHungry**  .

**Оператор модуля:  %**

Оператор модуля — ще один математичний оператор у JavaScript. Повертає залишок від ділення.

Щоб продемонструвати, як це працює, уявіть, що маленький ресторан із 4 стільцями на столі, а загалом 5 столів, раптом приймає 22 гостей.

Скільки гостей не зможуть сісти в ресторані?

Ви можете використовувати оператор модуля, щоб вирішити це.

console.log(22 % 5); // 2

Вихід дорівнює  **2** , тобто коли я ділю  **22**  на  **5** , я отримую  **4** , а залишок дорівнює  **2** , тобто є  **2**  людини, які не змогли отримати місце в цьому ресторані.

**Оператор рівності  ==**

Оператор рівності перевіряє, чи рівні два значення.

Наприклад, це порівняння повертає **true** : **5 == 5** . Дійсно, це правда, що 5 дорівнює 5.

Ось приклад оператора рівності, який повертає **false** : **5 == 6** . Дійсно, це правда, що 5 не дорівнює 6.

Крім того, навіть якщо одне з порівнюваних значень належить до числового типу, а інше — до рядкового, повернене значення все одно буде **істинним** : **5 == "5"** .

Це означає, що оператор рівності порівнює лише значення, але не типи.

**Оператор суворої рівності  ===**

Оператор суворої рівності порівнює як значення, так і типи даних.

За допомогою оператора суворої рівності порівняння **5 === 5** все одно повертає **true** . Значення на кожній стороні оператора суворої рівності мають однакові значення та однаковий тип. Однак порівняння **5 == "5"** тепер повертає **false** , оскільки значення рівні, але тип даних інший.

**Оператор нерівності  !=**

Оператор нерівності перевіряє, чи два значення не збігаються, але не перевіряє різницю в типах.

Наприклад, **5 != "5"** повертає false, оскільки невірно стверджувати, що число 5 не дорівнює числу 5, навіть якщо це інше число належить до рядкового типу даних.

**Оператор суворої нерівності  !==**

Щоб оператор суворої нерівності повертав **false** , порівнювані значення повинні мати однакові значення та однаковий тип даних.

Наприклад, **5 !== 5** повертає **false,** оскільки невірно говорити, що число 5 не має того самого значення та типу даних, що інше число 5.

Однак порівняння числа 5 і рядка 5 за допомогою оператора суворої нерівності повертає **true** .

console.log(5 !== "5")

**2. Використання  операторів +  для рядків і чисел**

**Об’єднання двох рядків за допомогою   оператора +**

Оператор  **+**  , коли використовується з числовим типом даних, додає ці значення разом.

Однак  оператор **+**  також використовується для об’єднання рядкових типів даних.

Наприклад:

"inter" + "net" // "internet"

"note" + "book" // "notebook"

Якщо  оператор **+**  використовується для об’єднання рядків, то його називають  оператором *конкатенації*  , і ви скажете, що він використовується для  *об’єднання*  рядків.

При використанні з числами  оператор **+**  є  **оператором додавання** , а при використанні з рядками  оператор **+**  є  **оператором конкатенації** .

**Об’єднання рядків і чисел за допомогою   оператора +**

Але що станеться, якщо об’єднати рядок і число за допомогою  оператора **+**  ?

Ось приклад:

365 + " days" // "365 days"

12 + " months" // "12 months"

Тут JavaScript намагається допомогти, перетворюючи числа на рядки, а потім  **об’єднуючи число та рядок разом** , у результаті чого отримується  **рядкове значення** .

Процес цього «прихованого» перетворення значень у JavaScript називається «примусом». JavaScript  *перетворює*  числове значення на рядкове значення, щоб він міг запускати  оператор **+**  для різних типів даних.

Процес примусу іноді може бути дещо несподіваним.

Розглянемо такий приклад:

1 + "2"

Яким буде результат  **1 + «2»** ?

Зверніть увагу, що значення  **1**  відноситься до типу даних числа, а значення  **«2»**  — до типу даних рядка, тому JavaScript приведе число  **1**  до рядка  **«1»** , а потім об’єднає його з рядком **"2"** ,  тому результатом є рядок  **"12"** .

**Оператор присвоювання додавання,  +=**

Оператор присвоювання додавання використовується, коли потрібно накопичити значення, що зберігаються у змінній.

Ось приклад: ви підраховуєте кількість понаднормових годин, відпрацьованих за тиждень.

Вам не потрібно вказувати тип роботи, ви просто хочете підрахувати загальну кількість годин.

Ви можете закодувати програму для його відстеження, наприклад:

var mon = 1;

var tue = 2;

var wed = 1;

var thu = 2;

var fri = 3;

console.log(mon + tue + wed + thu + fri); // 9

Ви можете спростити наведений вище код за допомогою оператора присвоювання додавання, як показано нижче:

var overtime = 1;

overtime += 2;

overtime += 1;

overtime += 2;

overtime += 3;

console.log(overtime); // 9

Використання оператора присвоювання додавання зменшує рядки вашого коду.

**Оператор присвоєння конкатенації,  +=**

Синтаксис цього оператора точно такий же, як і оператор присвоєння додавання. Різниця полягає в використовуваному типі даних:

var longString = "";

longString += "Once";

longString += " upon";

longString += " a";

longString += " time";

longString += "...";

console.log(longString); // "Once upon a time..."

**Пріоритет оператора та асоціативність**

Пріоритет операторів — це набір правил, які визначають, який оператор слід оцінювати першим.

Розглянемо такий приклад:

1 \* 2 + 3

Результатом коду вище є 5, оскільки оператор множення має пріоритет над оператором додавання.

Асоціативність операторів визначає, як працює пріоритет, коли код використовує оператори з однаковим пріоритетом.

Є два види:

* асоціативність зліва направо
* асоціативність справа наліво

Наприклад, оператор присвоювання є асоціативним справа наліво, тоді як оператор більшого є асоціативним зліва направо:

var num = 10; // the value on the right is assigned to the variable name on the left

5 > 4 > 3; // the 5 > 4 is evaluated first (to `true`), then true > 3 is evaluated to `false`, because the `true` value is coerced to `1`

**Умовні оператори**

Як **if else,** так і **switch** використовуються для визначення потоку виконання програми на основі того, чи були виконані деякі умови.

Ось чому їх іноді називають **операторами керування потоком** . Іншими словами, вони контролюють потік виконання вашого коду, так що деякий код можна пропустити, а інший — виконати.

В основі обох структур керування потоком лежить оцінка однієї чи кількох умов.

Як правило, **if else** краще підходить, якщо в умові є двійковий вибір.

Наприклад, простою англійською мовою:  *якщо сонячно, надіньте сонцезахисні окуляри. Інакше не* .

У цьому випадку використання оператора if є очевидним вибором.

Коли існує менша кількість можливих результатів перевірки правдивості, все одно можна використовувати оператор **if else** , наприклад:

if(light == "green") {

    console.log("Drive")

} else if (light == "orange") {

    console.log("Get ready")

} else if (light == "red") {

    console.log("Dont' drive")

} else {

    //this block will run if no condition matches

    console.log("The light is not green, orange, or red");

}

Однак, якщо існує багато можливих результатів, найкраще використовувати оператор switch, оскільки він менш багатослівний. Будучи легшим для читання, легше слідувати логіці, а отже, зменшити когнітивне навантаження від читання кількох умов.

Тим не менш, це не правило, встановлене в камені. Це просто стильовий вибір.

Щоб підкріпити цю думку, ось приклад попереднього оператора умови **if else** із використанням синтаксису switch:

//converting the previous if-else example with switch-case

switch(light) {

case 'green':

console.log("Drive");

break;

case 'orange':

console.log("Get ready");

break;

case 'red':

console.log("Don't drive");

break;

default:

//this block will run if no condition matches

console.log('The light is not green, orange, or red');

break;

}

**Цикли та вкладені цикли**

Припустімо, я хочу вивести настроювану таблицю множення.

Це ідеальний сценарій використання для вкладених циклів.

Змінна лічильника зовнішнього циклу діятиме як перше число, яке потрібно помножити, а змінна лічильника внутрішнього циклу діятиме як друге число, яке потрібно помножити.

Ось мій код:

//single loop

for (var firstNum = 0; firstNum < 2; firstNum++) {

    console.log(firstNum);

}

Результатом наведеного вище коду буде:

0 1

Це означає, що мій цикл for починається з 0 і зупиняється після 1.

Тепер я можу закодувати те, що пізніше стане внутрішнім циклом, чия змінна лічильника буде другим числом у цьому множенні:

//single loop

for (var secondNum = 0; secondNum < 10; secondNum++) {

    console.log(secondNum);

}

Цього разу результат:

0 1 2 3 4 5 6 7 8 9

Настав час поєднати першу і другу петлі:

//nested loops - one inside another

for (var firstNum = 0; firstNum < 2; firstNum++) {

    for (var secondNum = 0; secondNum < 10; secondNum++) {

        console.log(firstNum + ", " + secondNum);

    }

}

Тепер, коли я вкладаю другий цикл for у перший, і я консольно реєструю значення обох змінних лічильника під час виконання циклів, результат виглядає так:

0, 0 0, 1 0, 2 0, 3 0, 4 0, 5 0, 6 0, 7 0, 8 0, 9 1, 0 1, 1 1, 2 1, 3 1, 4 1, 5 1, 6 1, 7 1, 8 1, 9

Тепер, коли у мене є список усіх чисел, які будуть перемножені, отримати фактичний результат цього множення так само просто, як оновити  виклик **console.log()**  :

//nested loops - one inside another

for (var firstNum = 0; firstNum < 2; firstNum++) {

    for (var secondNum = 0; secondNum < 10; secondNum++) {

        console.log(firstNum + " times " + secondNum + " equals " + firstNum \* secondNum);

    }

}

Результат тепер:

0 помножити на 0 дорівнює 0 0 помножити 1 дорівнює 0 0 помножити 2 дорівнює 0 0 помножити 3 дорівнює 0 0 помножити 4 дорівнює 0 0 помножити 5 дорівнює 0 0 помножити 6 дорівнює 0 0 помножити на 7 дорівнює 0 0 помножити на 8 дорівнює 0 0 помножити на 9 дорівнює 0 1 помножити на 0 дорівнює 0 1 помножити на 1 дорівнює 1 1 на 2 дорівнює 2 1 на 3 дорівнює 3 1 на 4 дорівнює 4 1 на 5 дорівнює 5 1 на 6 дорівнює 6 1 на 7 дорівнює 7 1 на 8 дорівнює 8 1 на 9 дорівнює 9

Це створює дуже цікаві комбінації.

Наприклад, я можу створити власну таблицю ділення:

//nested loops - one inside another

for (var i = 100; i > 10; i = i - 10) {

    for (var j = 10; j > 4; j = j - 5) {

        console.log(i + " divided by " + j + " equals " + i / j);

    }

}

Ось результат вищезгаданого вкладеного циклу:

100 поділити на 10 дорівнює 10 100 поділити на 5 дорівнює 20 90 поділити на 10 дорівнює 9 90 поділити на 5 дорівнює 18 80 поділити на 10 дорівнює 8 80 поділити на 5 дорівнює 16 70 поділити на 10 дорівнює 7 70 поділити на 5 дорівнює 14 60 поділити на 10 дорівнює 6 60 поділити на 5 дорівнює 12 50 поділити на 10 дорівнює 5 50 поділити на 5 дорівнює 10 40 поділити на 10 дорівнює 4 40 поділити на 5 дорівнює 8 30 поділити на 10 дорівнює 3 30 поділити на 5 дорівнює 6 20 поділити на 10 дорівнює 2 20 поділити на 5 дорівнює 4

Не соромтеся спробувати деякі інші комбінації ітерацій вкладеного циклу та подивіться, який результат ви отримаєте.

**Застосування циклів**

У цьому читанні ми обговоримо на дуже високому рівні причини використання циклів у JavaScript.

Зауважте, що ми будемо підтримувати це обговорення на високому рівні, оскільки на даний момент ви все ще не розумієте кількох «частинок головоломки».

Ось чому ми не будемо занурюватися в деталі синтаксису та реалізації, а замість цього просто обговоримо, як і чому цикли використовуються в повсякденній роботі розробників JavaScript.

Розглянемо такий приклад: ви працюєте розробником в інтернет-магазині.

У магазині продаються кубики з літерами для малюків, і весь розділ сайту «Купити зараз» організовано за макетом, де кожен кубик у продажу відображається у вигляді простої картки із зображенням кубика, літери, якій він навчає, короткий опис і ціна.

Картки організовані в ряди, так що кожен ряд містить три картки - три різні літери.

Кожна картка є попереднім переглядом того конкретного куба з літерами, що продається, а також посиланням на цілу сторінку, присвячену наданню додаткової інформації про кубики, їх навчальну цінність і надання відвідувачам способу завершити процес оформлення замовлення.

Тепер ось коротке запитання: куди петлі вписуються в цю сітку карток, що демонструють кубики з літерами, що продаються?

Щоб зрозуміти, як це працює, дозвольте мені закодувати базовий прототип того, як це може працювати.

Оскільки у вас ще недостатньо знань, щоб відображати макети веб-сайтів у браузері за допомогою JavaScript, наразі мені доведеться задовольнитися використанням простого рядка та консолі.

Тим не менш, це має бути весела вправа.

var cubes = 'ABCDEFG';

//styling console output using CSS with a %c format specifier

for (var i = 0; i < cubes.length; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}

***Примітка: щоб застосувати стилі, спробуйте запустити цей фрагмент коду в консолі вашого браузера.***

Ось і все, за допомогою цього простого коду вихідні дані в консолі показують кожну літеру в окремому рядку, стилізованому як літерний кубик для малюків.

Сам код має бути здебільшого знайомим, за винятком  **cubes.length**  і  синтаксису **cubes[i]**  .

Не вдаючись у подробиці, ось обидва фрагменти коду пояснюються якомога простіше.

Cubes.length  **повертає**  число. Оскільки  **cubes**  — це рядок символів,  **cubes.length**  дає мені довжину рядка, збереженого в змінній.

Отже, це дає мені число 7, фактично роблячи мій цикл for таким:

var cubes = 'ABCDEFG';

//styling console output using CSS with a %c format specifier

for (var i = 0; i < 7; i++) {

    var styles = "font-size: 40px; border-radius: 10px; border: 1px solid blue; background: pink; color: purple";

    console.log("%c" + cubes[i], styles)

}



Другим новим фрагментом коду є  фрагмент **cubes[i]**  .

Це просто націлює кожну окрему букву в циклі на основі поточного значення змінної  **i**  .

Іншими словами,  **cubes[i]** , коли  **i**  дорівнює  **0** , це:  **A** .

Тоді  **cubes[i]** , коли  **i**  дорівнює  **1** , це:  **B** .

Це відбувається стільки циклів, скільки виконується мій цикл for - і це визначається значенням  **cubes.length**  .

Це також дуже універсально, оскільки, наприклад, якби я вирішив змінити довжину  рядка **кубів**  , мені не потрібно було б оновлювати умову  **i < cubes.length** , оскільки вона автоматично оновлюється, коли я змінюю довжину  **кубики**  рядок.

Існують інші способи зберігання даних у програмах JavaScript, про які ви не чули.

Але ми можемо використовувати той самий підхід із цими іншими видами даних, щоб досягти результатів, які, по суті, працюють за тим же принципом, що й щойно описаний.

Використання циклів — це суть підходу, який використовується для розробки різноманітних функціональних можливостей програмного забезпечення сьогодні.

**Деякі додаткові приклади**

Якщо я кодую поштовий клієнт, я отримаю деякі структуровані дані про електронні листи, які відображатимуться у папці «Вхідні», а потім використаю цикл, щоб фактично відобразити їх у гарно відформатованому вигляді.

Якщо я кодую сайт електронної комерції з продажу автомобілів, я отримаю джерело добре структурованих даних про кожну з машин, а потім переберу ці дані, щоб відобразити їх на екрані.

Якщо я кодую онлайн-календар, я буду циклічно переглядати дані, що містяться в кожному з днів, щоб відобразити добре відформатований календар.

Є багато, багато інших прикладів використання циклів у коді.

Використання циклів із даними, правильно відформатованими для певного завдання, є ключовим компонентом створення програмного забезпечення.

У наступних уроках ми дізнаємося про різні способи групування пов’язаних даних і відображення їх на екрані за допомогою JavaScript.

У поєднанні з тим, що ви вже дізналися про цикли, це дає вам навички для створення різних типів інтерфейсів користувача з повторюваною інформацією.

Ось деякі більш конкретні приклади:

* циклічний перегляд заголовків публікацій блогу в деяких структурованих даних і відображення заголовка кожної публікації блогу на домашній сторінці блогу
* переглядати публікації в соціальних мережах у деяких структурованих даних і відображати кожну публікацію в соціальних мережах на основі певних умов
* перегляд деяких структурованих даних про одяг, доступний для продажу в онлайн-магазині одягу, і відображення відповідних даних для кожного елемента одягу

Тепер ви розумієте важливість знання того, як працювати з циклами в JavaScript. На наступних уроках ми дізнаємося іншу відповідну інформацію, яка дозволить вам це зробити.

**Побудова та виклик функцій**

Давайте почнемо з того, що дамо назву нашій декларації функції:

function listArrayItems(arr) {

    // ... code to be added ...

}

Отже, я оголосив  функцію **listArrayItems**  і налаштував її на прийом єдиного параметра  **arr** ,  який означає масив.

Тепер мені потрібно буде закодувати цикл for для проходження циклу по масиву.

Як було сказано в попередніх уроках цього курсу, для циклу for потрібна така інформація:

1. початкове значення лічильника циклу як тимчасова змінна  **i**

2. умова виходу (максимальне значення змінної  **i** лічильника циклу , вище якого цикл більше не виконується)

3. як оновити значення  **i**  після кожного циклу

Ось інформація, яку я буду використовувати в цій декларації функції: 1. Початковий лічильник циклу буде  **0** . Причина встановлення значення нуля полягає в тому, що масиви також відраховуються від нуля.

Це означає, що я матиму однозначне відображення поточного значення змінної  **i**  в будь-який момент часу, що відповідає тій же позиції індексу будь-якого елемента в  масиві **arr**  2.

Умова виходу з циклу for полягає в тому, що значення  **i**  дорівнює або перевищує  **arr.length** .

Оскільки  **arr.length**  підраховує кількість елементів у масиві від одиниці, а елементи масиву індексуються від нуля, це фактично означає, що як тільки  **i**  дорівнює  **arr.length** , цикл завершиться, а будь-який інший код після нього завершиться. буде запущено.

Це практично означає, що умовою виходу для цього циклу for буде  **i < arr.length,**  що повертає  **false** .

Іншими словами, доки  **i < arr.length**  має значення true, цей цикл for продовжуватиме працювати. 3. Щоб переконатися, що жоден із елементів у  масиві **arr**  не пропускається, я маю збільшувати значення  **i**  на  **1**  після кожного циклу.

Тепер, коли я точно знаю, як повинен поводитися мій цикл for, я можу додати його до своєї   функції **listArrayItems() :**

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        // ... code pending here ...

    }

}

Тепер все, що мені залишилося вирішити, це те, як я хочу вивести кожен елемент із отриманого  масиву **arr**  .

Це може бути таким же простим, як консольний журнал індексу елемента масиву поточного значення  **i** :

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        console.log(arr[i]) //display the array item where the index is euqal to i

    }

}

Якщо я зараз викликаю  функцію **listArrayItems**  , я можу, наприклад, надати їй такий масив кольорів:

var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'pink'];

listArrayItems(colors); //display all items in the array at once

Результатом буде:

червоний

помаранчевий

жовтий

зелений

блакитний

фіолетовий

рожевий

Я можу оновити результат будь-яким способом. Наприклад, ось мої  елементи **arr**  з номером перед кожним елементом:

//function that takes an array as input and display all items of this array

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        console.log(i, arr[i])

    }

}

var colors = ['red', 'orange', 'yellow', 'green', 'blue', 'purple', 'pink'];

listArrayItems(кольори);

Тепер вихід наведеного вище коду буде таким:

0 'red'

1 'orange'

2 'yellow'

3 'green'

4 'blue'

5 'purple'

6 'pink'

Щоб почати відлік від одиниці замість нуля, я можу оновити свою декларацію функції наступним чином:

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        console.log(i+1, arr[i])

    }

}

Виклик наведеної вище оновленої декларації функції для мого  масиву **кольорів**  тепер призведе до наступного результату:

1 'red'

2 'orange'

3 'yellow'

4 'green'

5 'blue'

6 'purple'

7 'pink'

Я навіть можу додати одну або кілька умов, наприклад:

function listArrayItems(arr) {

    for (var i = 0; i < arr.length; i++) {

        if (arr[i] == 'red') {

            console.log(i\*100, "tomato!")

        } else {

            console.log(i\*100, arr[i])

        }

    }

}

Тепер я додаю потік керування прямо в мою функцію, залежно від того, чи відповідає конкретний елемент масиву певному значенню - у цьому випадку рядок  **"red"** .

Виклик моєї останньої оновленої версії  функції **listArrayItems**  для  масиву **кольорів**  тепер призведе до такого результату:

0 'tomato!'

100 'orange'

200 'yellow'

300 'green'

400 'blue'

500 'purple'

600 'pink'

**Об’єктні літерали та крапкова нотація**

**Літерали об’єктів і крапкова нотація**

Одним із найпоширеніших способів побудови об’єкта в JavaScript є використання синтаксису об’єктного літералу:  **{}** .

Щоб мати доступ до цього літералу об’єкта, дуже часто призначають його змінній, наприклад:

var user = {}; //create an object

Тепер об’єктний літерал призначається змінній  **user** , що означає, що об’єкт, до якого він прив’язаний, можна розширювати та маніпулювати безліччю способів.

Іноді можна відразу побудувати цілий об’єкт, використовуючи синтаксис літералу об’єкта, вказавши властивості об’єкта, розділені парами «ключ-значення», використовуючи синтаксис, який уже розглядався в попередньому пункті цього уроку.

Ось один з таких раніше побудованих об'єктів:

//creating an object with properties and their values

var assistantManager = {

    rangeTilesPerTurn: 3,

    socialSkills: 30,

    streetSmarts: 30,

    health: 40,

    specialAbility: "young and ambitious",

    greeting: "Let's make some money"

}

Принадність цього синтаксису в тому, що його легко читати.

По суті, він складається з двох кроків:

1. Оголошення нової змінної та присвоєння їй літералу об’єкта – іншими словами, це:  **var assistantManager = {}**

2. Присвоєння значень кожному з ключів об’єкта за допомогою оператора присвоювання  **=**

Зауважте, що за допомогою цього прикладу синтаксису дуже легко створити будь-який об’єкт у JavaScript.

Наприклад, ось  об’єкт **таблиці**  :

var table = {

    legs: 3,

    color: "brown",

    priceUSD: 100,

}

Щоб отримати доступ до  об’єкта **таблиці**  , я можу просто консольно зареєструвати весь об’єкт:

console.log(table);//display the object in the developer console

Поверненим значенням є весь  об’єкт **таблиці**  :

{legs: 3, color: 'brown', priceUSD: 100}

Крім того, я можу консольно реєструвати будь-яку окрему властивість, наприклад:

console.log(table.color); // 'brown'

Тепер, коли я маю цей «синтаксичний рецепт», я можу створити будь-який інший об’єкт подібним чином:

var house = {

    rooms: 3,

    color: "brown",

    priceUSD: 10000,

}

Альтернативний підхід до побудови об’єктів полягає в тому, щоб спочатку зберегти порожній літерал об’єкта до змінної, потім використовувати крапкову нотацію для оголошення нових властивостей на льоту та використовувати оператор присвоєння для додавання значень до цих властивостей; наприклад:

var house2 = {};

house2.rooms = 4;

house2.color = "pink";

house2.priceUSD = 12345;

Крім того, ніщо не заважає мені поєднувати обидва підходи. Наприклад:

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000}

house.windows = 10;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 10}

Ця гнучкість додатково означає, що я можу оновлювати вже наявні властивості, а не просто додавати нові:

house.windows = 11;

console.log(house); // {rooms: 3, color: "brown", priceUSD: 10000, windows: 11}

**Об’єктні літерали та позначення в дужках**

**Літерали об’єктів і позначення в дужках**

Існує альтернативний синтаксис позначення крапкою, який я використовував до цього моменту.

Цей альтернативний синтаксис відомий як  *позначення в дужках* .

Щоб зрозуміти, як це працює, найкраще скористатися прикладом, тому я знову пройду процес кодування  об’єкта **house2**  так само, як і з крапковою нотацією, тільки цього разу я використаю дужки позначення.

var  house2 = {};

house2[ "кімнати" ] =  4 ;

house2[ 'color' ]=  "pink" ;

house2[ "priceUSD" ] =  12345 ;

console.log(будинок2);  // {кімнат: 4, колір: 'рожевий', цінаUSD: 12345}

Зауважте, що використовуючи нотацію в дужках, я, по суті, просто загортаю ключ кожної властивості  **як рядок** всередині одинарних або подвійних лапок - як і зі звичайними рядками.

Потім я загортаю весь ключ властивості у відкриваючу та закриваючу квадратні дужки.

Ось, по суті, і все.

Я можу як отримувати доступ до властивостей об’єктів, так і оновлювати їх, використовуючи або крапкову нотацію, або дужку, або комбінацію обох, як у наступному прикладі:

var car = {};

car.color = "red";

car["color"] = "green";

car["speed"] = 200;

car.speed = 100;

console.log(car); // {color: "green", speed: 100}

На даний момент, мабуть, достатньо інформації про створення об'єкта.

Перш ніж я обговорюю тему масивів та об’єктів, дозвольте мені просто дати вам ще одну важливу інформацію про позначення в дужках.

За допомогою нотації в дужках я можу додавати пробіли в імена властивостей, наприклад:

car["number of doors"] = 4;

console.log(car); // {color: 'green', speed: 100, number of doors: 4}

Крім того, я можу додати числа (як рядковий тип даних) як ключі властивостей:

car["2022"] = 1901;

console.log(car); // {2022: 1901, color: 'green', speed: 100, number of doors: 4}

Однак робити це не рекомендується через очевидні причини, оскільки ключ властивості як рядок цифр насправді не передає багато корисної інформації.

Нарешті, є одна дуже корисна річ, яку має дужкова нотація, але недоступна в точковій нотації: вона може обчислювати вирази.

Щоб зрозуміти, що це означає, розглянемо наступний приклад:

var arrOfKeys = ['speed', 'altitude', 'color'];

var drone = {

    speed: 100,

    altitude: 200,

    color: "red"

}

for (var i = 0; i < arrOfKeys.length; i++) {

    console.log(drone[arrOfKeys[i]])

}

Наведений вище код призведе до такого результату:

100

200

червоний

Використовуючи той факт, що дужки можуть обчислювати вирази, я отримав доступ до   властивості  **arrOfKeys[i] об’єкта drone**  .

Це значення змінювалося в кожному циклі під час роботи циклу for.

Зокрема, під час першого запуску його було оцінено так:

* Значення  **i**  було  **0**
* Значення  **arrOfKeys[i]**  було  **arrOfKeys[0]** , що було  **«швидкістю»**
* Таким чином,  **drone[arrOfKeys[i]]**  було оцінено як  **drone["speed"],**  що дорівнює  **100**

Це дозволило мені пройти по кожному зі значень, що зберігаються всередині  об’єкта **дрона**  , на основі кожного з ключів його властивостей.

**Масиви є об’єктами**

У JavaScript масиви є об’єктами. Це означає, що масиви також мають деякі вбудовані властивості та методи.

Одними з найбільш часто використовуваних вбудованих методів для масивів є  методи **push()**  і  **pop()**  .

Щоб додати нові елементи до масиву, я можу використати  метод **push()**  :

var  fruits = [];

fruits.push( "яблуко" );  // ['яблуко']

fruits.push( 'груша' );  // ['яблуко', 'груша']

Щоб видалити останній елемент із масиву, я можу використати  метод **pop()**  :

fruits.pop();

console.log(fruits); // ['apple']

Пов’язуючись з деякими попередніми уроками в цьому курсі, тепер я можу побудувати функцію, яка приймає всі свої аргументи та вставляє їх у масив, наприклад:

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(один);

    arr.push(два);

    arr.push(три);

    console.log(arr);

}

Тепер я можу викликати  функцію **arrayBuilder()**  , наприклад, так:

arrayBuilder('apple', 'pear', 'plum'); // ['apple', 'pear', 'plum']

Навіть краще, мені не потрібно консольно реєструвати щойно створений масив.

Натомість я можу повернути його:

function arrayBuilder(one, two, three) {

    var arr = [];

    arr.push(один);

    arr.push(два);

    arr.push(три);

    return arr;

}

Крім того, я можу зберегти виклик цієї функції у змінній.

Я можу назвати його як завгодно, але цього разу я використаю назву:  **simpleArr** .

var simpleArr = arrayBuilder('apple', 'pear', 'plum');

І тепер я можу консольно реєструвати значення, що зберігаються в  **simpleArr** :

console.log(simpleArr); // ['apple','pear','plum']

**Математичні функції**

JavaScript має зручні вбудовані об’єкти. Одним із таких популярних вбудованих об’єктів є об’єкт Math.

**Числові константи**

Ось деякі з вбудованих числових констант, які існують в об’єкті Math:

* Число PI:  **Math.PI**  , що дорівнює приблизно 3,14159
* Стала Ейлера:  **Math.E**  , яка дорівнює приблизно 2,718
* Натуральний логарифм 2:  **Math.LN2** , який дорівнює приблизно 0,693

**Методи округлення**

До них належать:

* **Math.ceil()**  - округлює до найближчого цілого числа
* **Math.floor()**  - округляє до найближчого цілого числа
* **Math.round()**  - округляє до найближчого цілого числа, якщо десятковий дроб дорівнює  **0,5**  або більше; інакше округляє до найближчого цілого числа
* **Math.trunc()**  - обрізає десятковий дроб, залишаючи лише ціле число

**Методи арифметики та числення**

Ось невичерпний список деяких поширених методів арифметики та числення, які існують в  об’єкті **Math**  :

* **Math.pow(2,3)**  - обчислює число  **2**  у степені  **3** , результат  **8**
* **Math.sqrt(16)**  - обчислює квадратний корінь із  **16** , результат дорівнює  **4**
* **Math.cbrt(8)**  - знаходить кубічний корінь з  **8** , результат  **2**
* **Math.abs(-10)**  - повертає абсолютне значення, результат  **10**
* Логарифмічні методи:  **Math.log()** ,  **Math.log2()** ,  **Math.log10()**
* Повертає мінімальне та максимальне значення всіх вхідних даних:  **Math.min(9,8,7)**  повертає  **7** ,  **Math.max(9,8,7)**  повертає  **9** .
* Тригонометричні методи:  **Math.sin()** ,  **Math.cos()** ,  **Math.tan()** тощо.

**Робота з рядками**

Для всіх прикладів я буду використовувати одну або обидві наступні змінні:

var greet = "Hello, ";

var place = "World"

Зауважте, що незалежно від властивостей і методів рядків, які я демонструю в наступних прикладах, я міг би запустити їх безпосередньо на цих рядках, не зберігаючи їх у змінних, таких як ті, які я назвав  **greet**  і  **place** .

У деяких наступних прикладах, для ясності, замість використання назви змінної я використовуватиму сам рядок.

Усі рядки мають у своєму розпорядженні кілька вбудованих властивостей, але є одна властивість, яка дійсно корисна:  властивість **length**  , яка використовується так:

greet.length; // 7

Щоб прочитати кожен окремий символ за певним індексом у рядку, починаючи з нуля, я можу використати  метод **charAt()**  :

greet.charAt(0); // 'H'

Метод concat  **()**  об’єднує два рядки:

"Wo".concat("rl").concat("d"); // 'World'

indexOf   повертає розташування першої позиції, яка відповідає символу **:**

"ho-ho-ho".indexOf('h'); // 0

"ho-ho-ho".indexOf('o'); // 1

"ho-ho-ho".indexOf('-'); // 2

LastIndexOf   знаходить останній збіг, інакше він працює так само,  **як indexOf** .

Метод  **split**  розбиває рядок на масив підрядків:

"ho-ho-ho".split("-"); // ['ho', 'ho', 'ho']

Існують також деякі методи зміни регістру рядків. Наприклад:

greet.toUpperCase(); // "HELLO, "

greet.toLowerCase(); // "hello, "

Ось список усіх методів, розглянутих у цій шпаргалці:

* **charAt()**
* **concat()**
* **indexOf()**
* **lastIndexOf()**
* **split()**
* **toUpperCase()**
* **toLowerCase()**

Перейти до наступного пункту

**Методи об’єктів**

Можливо, ви вже знайомі з об’єктами в JavaScript.

У цьому відео ви дізнаєтеся, як проектувати об’єкти як комбінації даних і функцій.

Як ви вже могли знати, об’єкт складається з пар ключ-значення, відомих як властивості.

Ми можемо додавати нові пари ключ-значення до існуючих об’єктів за допомогою крапкової нотації та оператора присвоювання.

var car = {};

car.color = "red"; //update the value of a property of the car objject

Вони відомі як властивості та можуть приймати багато типів даних, у тому числі функції.

var car = {};

car.color = "red";

//add a method to the car object so that it can be called as car.turnkey()

car.turnKey = function() {

  console.log('engine running');

}

Якщо функція є властивістю об’єкта, її називають методом.

Це функція, до якої можна отримати доступ лише через об’єкт JavaScript, членом якого вона є. Наприклад, до методу log, який належить об’єкту консолі, можна отримати доступ лише через об’єкт консолі.

**console.log('Привіт, світ');**

Давайте розглянемо це далі. Я створю об’єкт за допомогою функції конструктора.

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

Спочатку я побудую новий об’єкт під буквальною назвою  **автомобіль** . Я вводжу  **var** , пробіл,  **car** , пробіл, знак рівності, пробіл, за якими йдуть фігурні дужки і, нарешті, крапка з комою.

Тепер я розширю  об’єкт **car**  , призначивши йому властивість під назвою  **mileage** .

Коли я перевіряю об’єкт, я можу підтвердити, що він містить  властивість **пробігу**  , встановлену на  **98765** .

Я хочу додати іншу властивість до  об’єкта **автомобіля**  . Цього разу я додам властивість під назвою  **color**  і встановлю для неї значення  **«red»** .

Я можу перевірити об’єкт ще раз, ввівши його назву в консоль браузера. Тож тепер, коли я вводжу  **console.log(car)** , я отримую об’єкт із двома властивостями:  властивістю **mileage**  , яка має значення  **98765** , і  властивістю **color**  , яка має значення  **«red»** .

Чудово, тепер я додав дві властивості до свого об’єкта.

Далі я хочу додати метод до мого  об’єкта **автомобіля**  . І цей метод, коли викликається, виведе деякий текст на консоль.

Отже, я знову додаю іншу властивість до мого  об’єкта **автомобіля**  . Зрештою, метод — це ще одна властивість об’єкта  **автомобіля**  . Це просто ще одне співвідношення ключ-значення, яке містить об’єкт car.

Унікальним є те, що значення, яке я йому призначаю, є функцією.

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

console.log(car);

Отже, я починаю з введення  **car**  dot  **turnTheKey** , дорівнює, а потім вводжу код своєї функції. Отже  , **функція** відкриває-закриває дужки. Потім дві фігурні дужки, де я розміщу свій код. Нарешті, у фігурних дужках я вводжу консольний журнал із крапками, а потім повідомлення  **«Двигун працює»** .

Тепер я можу знову перевірити свій  **автомобільний**  об’єкт, ввівши його назву в метод журналу консолі. Цього разу він показує, що  об’єкт **car**  містить три властивості; властивість колір   ,  **властивість пробігу та**  властивість  **turnTheKey**  .

Пам’ятайте, що всі пари ключ-значення в об’єкті називаються просто властивостями. Однак, якщо я хочу розрізнити властивості, які можна виконати, я називаю такі властивості методами.

Отже, тепер я хочу додати інший метод до  об’єкта **автомобіля**  . Я назву його  **lightsOn** .

Я знову вводжу  **car.lightsOn** , а потім додаю знак рівності, і знову, оскільки це метод, я призначаю його функції. Ця функція також матиме журнал консолі в своєму тілі, і я просто реєструю рядок із текстом  **«The lights is on»** .

//example of adding properties and methods to an object

var car = {};

car.mileage = 98765;

car.color = "red";

console.log(car);

car.turnTheKey = function() {

    console.log("The engine is running")

}

car.lightsOn = function() {

    console.log("The lights are on.")

}

console.log(car);

car.turnTheKey();

car.lightsOn()

Добре, тепер я додав чотири властивості до свого об’єкта. І два з них є методами.

Я вже переконався, що я отримую правильний  **пробіг**  і  **колір**  свого  об’єкта **автомобіля**  . Тепер я спробую виконати  методи **turnTheKey**  і  **lightsOn**  .

Спочатку я викличу  метод **turnTheKey**  .

Пам’ятайте, що до цього методу можна отримати доступ лише через  об’єкт **car**  , тому мені спочатку потрібно ввести назву об’єкта, який містить  метод **turnTheKey**  . Іншими словами, мені потрібно ввести слово  **car** , за яким стоїть крапка, а потім назва мого методу, яким є  **turnTheKey** .

Пам'ятайте, що ця властивість є методом. Отже, щоб запустити його, мені потрібно додати відкриваючу та закриваючу дужки, щоб механізм JavaScript міг обробити мій код JavaScript.

Зверніть увагу, що це призводить до того, що  рядок **«Двигун працює»**  реєструється на консолі.

Зараз я перевірю інший метод. Знову ж таки, мені потрібно отримати до нього доступ через  об’єкт **car**  , тому я вводжу  **car.lightsOn** і знову потрібно додати ці дужки, щоб викликати  метод **lightsOn**  . Я натискаю клавішу ENTER і помічаю текст, який відображається на консолі.

Успіх! Важливо пам’ятати, що коли механізм JavaScript виконує цей рядок коду, він знаходить  об’єкт **car**  у своїй пам’яті. Потім він знаходить  метод **lightsOn**  на  об’єкті **автомобіля**  . Він читає оголошення функції, збережене в цій властивості, і виконує його рядок за рядком.

Оскільки існує лише один рядок коду, механізм JavaScript реєструє рядок  **«The lights is on»**  на консолі.

**Синтаксичні, логічні помилки та помилки виконання**

Ось деякі з найпоширеніших помилок у JavaScript:

* ReferenceError
* SyntaxError
* TypeError
* RangeError

Є ще деякі помилки в JavaScript. Ці інші помилки включають:

* AggregateError
* Помилка
* InternalError
* URIError

Однак у цьому читанні я зосереджуся на помилках посилання, синтаксису, типу та діапазону.

**ReferenceError**

Помилка ReferenceError виникає, наприклад, коли хтось намагається використати змінні, які ніде не оголошені.

Прикладом може бути, скажімо, спроба консолі зареєструвати змінну, яка не існує:

console.log(ім’я користувача);

Якщо змінну з ім’ям  **користувача**  не було оголошено, наведений вище рядок коду призведе до такого результату:

Uncaught ReferenceError: username is not defined

**SyntaxError**

Будь-який недійсний код JavaScript спричинить SyntaxError.

Наприклад:

var a "there's no assignment operator here";

Наведений вище рядок коду видасть таку помилку:

Uncaught SyntaxError: Unexpected string

Існує цікаве застереження щодо SyntaxError у JavaScript: її не можна перехопити за допомогою  блоку **try-catch**  .

**TypeError**

Помилка TypeError виникає, наприклад, під час спроби запустити метод для непідтримуваного типу даних.

Простий приклад: спроба запустити  метод **pop()**  для рядка:

"hello".pop() // Uncaught TypeError: "hello".pop is not a function

Поведінка рядків, подібна до масиву, вже розглядалася в попередньому уроці цього курсу.

Однак, як можна підтвердити, запустивши наведений вище рядок коду, рядки не мають усіх доступних методів масиву, і спроба використати деякі з цих методів призведе до появи TypeError.

**RangeError**

Помилка RangeError виникає, коли ми надаємо значення функції, але це значення виходить за межі дозволеного діапазону допустимих вхідних значень.

Ось простий приклад перетворення звичайного  *числа за основою 10*  (число загальної десяткової системи) у  *число за основою 2*  (тобто двійкове число).

Наприклад:

(10).toString(2); // '1010'

Значення  **2**  , передане в  метод **toString()**  , схоже на те, що потрібно сказати JavaScript: «перетворити значення  **10**  системи числення з основою 10 на відповідну частину в системі числення з підставою 2».

JavaScript зобов’язує і «перекладає» «звичайне» число 10 на його двійковий аналог.

Крім використання системи числення з основою 2, я також можу використовувати систему числення з основою 8, ось так:

(10).toString(8); // 12

Я повертаю значення  **12** , яке є простим числом 10, записаним у системі числення з основою 8.

Однак, якщо я спробую використати неіснуючу систему числення, таку як уявна система числення  *Base 100* , оскільки ця система числення фактично не існує в JavaScript, я отримаю RangeError, оскільки неіснуюча   система  числення *Base 100* **не відповідає діапазон**  систем числення, які доступні для  методу **toString()**  :

(10).toString(100); // Uncaught RangeError: toString() radix argument must be between 2 and 36

У цій статті ви розглянули деякі з найпоширеніших помилок у JavaScript.

**Повернення значень із функцій**

Багато функцій за замовчуванням повертають значення  **undefined** .

Прикладом є  функція **console.log()**  .

Якщо я виконаю:

console.log('Hello');

... ось такий результат буде у консолі:

Hello

undefined

Оскільки  функція **console.log()**  створена таким чином, щоб не мати явно встановленого значення, що повертається, вона отримує стандартне значення, що повертається,  **undefined** .

Тепер я напишу власну реалізацію  **console.log()** , яка не повертає значення  **undefined** :

function consoleLog(val) {

    console.log(val)

    return val

}

Я використовую  функцію **console.log()**  у своїй власній   декларації функції **consoleLog** . І я вказую його, щоб повернути значення його аргументу.

Тепер, коли я запускаю свою спеціальну  функцію **consoleLog()**  :

consoleLog('Hello')

Я отримую такий результат:

Hello

'Hello'

Отже, значення виводиться на консоль, але воно також повертається.

Чому це корисно?

Це корисно, оскільки я можу використовувати значення, що повертаються з однієї функції всередині іншої функції.

Ось приклад.

Спочатку я закодую функцію, яка повертає подвійне число, яке вона отримала:

function doubleIt(num) {

    return num \* 2

}

Тепер я закодую іншу функцію, яка створює об’єкт із певним значенням:

function objectMaker(val) {

    return {

        prop: val

    }

}

Я можу викликати  функцію **objectMaker()**  із будь-яким значенням, яке мені подобається, наприклад:

objectMaker(20);

Поверненим значенням буде об’єкт з єдиним  ключем **prop**  , встановленим на  **20** :

{prop:20}

Тепер розглянемо цей код:

doubleIt(10).toString()

Наведений вище код повертає число  **20**  як рядок, тобто:  **"20"** .

Я навіть можу комбінувати виклики своїх спеціальних функцій наступним чином:

objectMaker( doubleIt(100) );

Тепер це поверне таке значення:

{prop: 200}

Що все це означає?

Це означає, що завдяки JavaScript, який дозволяє мені використовувати  ключове слово **return**  , як описано вище, я можу мати кілька викликів функцій, повертаючи дані та маніпулюючи значеннями, залежно від будь-якого завдання кодування, яке переді мною стоїть.

Можливість повертати власні значення є однією з основ, що робить можливим функціональне програмування.

**Парадигма функціонального програмування**

**Навчальні цілі**

* Вміти пояснити, що існує декілька парадигм програмування
* Вміти пояснити основну різницю між двома домінуючими парадигмами програмування: функціональним та об’єктно-орієнтованим програмуванням.
* Зрозумійте на дуже високому рівні, як працює парадигма функціонального програмування

«Насправді існує кілька стилів кодування, також відомих як  **парадигми** . Загальний стиль називається  **функціональним програмуванням** або скорочено FP.

У функціональному програмуванні ми використовуємо багато функцій і змінних.

function getTotal(a,b) {

    return a + b

}

var num1 = 2;

var num2 = 3;

var total = getTotal(num1, num2);

Під час написання коду FP ми зберігаємо дані та функціональні можливості окремо та передаємо дані у функції лише тоді, коли нам потрібно щось обчислити.

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

У функціональному програмуванні функції повертають нові значення, а потім використовують ці значення в іншому місці коду.

function getDistance(mph, h) {

    return mph \* h

}

var mph = 60;

var h = 2;

var distance = getDistance(mph, h);

console.log(distance); // <====== THIS HERE!

Іншим стилем є  **об’єктно-орієнтоване програмування (ООП)** . У цьому стилі ми групуємо дані та функції як властивості та методи всередині об’єктів.

Наприклад, якщо у мене є  об’єкт **virtualPet**  , я можу надати йому  властивість **sleepy**  і  метод **nap()**  :

var virtualPet = {

    sleepy: true,

    nap: function() {}

}

В ООП методи  **оновлюють властивості,**  що зберігаються в об’єкті, замість того, щоб генерувати нові значення, що повертаються.

Наприклад, якщо я перевіряю  властивість **sleepy**  об’єкта  **virtualPet**  , я можу підтвердити, що для неї встановлено значення  **true** .

Однак після того, як я запустив  метод **nap()**  для  об’єкта **virtualPet**  , чи   зміниться значення властивості **sleepy ?**

//creating an object

var virtualPet = {

    sleepy: true,

    nap: function() {

        this.sleepy = false

    }

}

console.log(virtualPet.sleepy) // true

virtualPet.nap()

console.log(virtualPet.sleepy) // false

ООП допомагає нам моделювати об’єкти реального життя. Це найкраще працює, коли групування властивостей і даних в об’єкті має логічний сенс, тобто властивості та методи «належать разом».

Зауважте, що тут не йдеться про глибоке обговорення ООП; натомість я просто хочу показати вам найпростіше пояснення того, що це таке і як це працює, щоб зробити найважливішу різницю між FP та ООП.

Підводячи підсумок, ми можемо сказати, що парадигма функціонального програмування працює, зберігаючи дані та функціональні можливості окремо. Його аналог, ООП, працює, зберігаючи дані та функції, згруповані в значущі об’єкти.

У функціональному програмуванні є ще багато концепцій та ідей.

Ось деякі з найважливіших:

* Першокласні функції
* Функція вищого порядку
* Чисті функції та побічні ефекти

У функціональному програмуванні існує багато інших концепцій і принципів, але поки що зупинимося на цих трьох.

**Першокласні функції**

Часто кажуть, що функції в JavaScript є «першокласними громадянами». Що це означає?

Це означає, що функція в JavaScript – це просто ще одне значення, яке ми можемо:

* перейти до інших функцій
* зберегти у змінній
* повернення з інших функцій

Іншими словами, функція в JavaScript — це просто значення — з цієї точки зору воно майже не відрізняється від рядка чи числа. Наприклад, у JavaScript цілком нормально передавати виклик функції іншій функції. Щоб пояснити, як це працює, розглянемо наступну програму.

function addTwoNums(a, b) {

    console.log(a + b)

}

function randomNum() {

    return Math.floor((Math.random() \* 10) + 1);

}

function specificNum() { return 42 };

var useRandom = true;

var getNumber;

if(useRandom) {

    getNumber = randomNum

} else {

    getNumber = specificNum

}

addTwoNums(getNumber(), getNumber())

Я запускаю програму за допомогою функції **addTwoNums()** , визначення якої я вже використовував раніше в різних варіантах. Причина, чому ця функція є повторюваним прикладом, полягає в тому, що вона настільки проста, що допомагає пояснити концепції, які інакше було б трохи важче зрозуміти. Далі я кодую функцію під назвою **randomNum()** , яка повертає випадкове число від 0 до 10. Потім я кодую іншу функцію під назвою **specificNum()** , яка повертає певне число, число 42. Потім я зберігаю змінну під назвою **useRandom** і я встановіть для нього логічне значення **true** . Я оголошую ще одну змінну під назвою **getNumber** . Ось де все стає цікавим. У наступних кількох рядках у мене є оператор if else. Умова if виконується, коли значення **useRandom** має значення **true** . Якщо це так, уся декларація функції **randomNum()** зберігається в змінній **getNumber** . В іншому випадку я зберігаю всю декларацію функції **specificNum() у змінній getNumber** . Іншими словами, залежно від того, що **useRandom** має значення **true** або **false** , змінній **getNumber** буде присвоєно або оголошення функції **randomNum() , або оголошення функції specificNum()** . З усім цим набором коду я можу викликати функцію **addTwoNums()** , передаючи їй виклик змінної **getNumber()** як перший і другий аргументи. **Це працює, тому що функції в JavaScript є справді першокласними громадянами, які можна призначати іменам змінних і передавати так само, як я передавав би рядок, число, об’єкт тощо.** Примітка: більшість коду всередині **randomNum( )** оголошення функції взято з попереднього уроку, а саме уроку, де обговорювався об’єкт Math у JavaScript. Це підводить мене до другої фундаментальної концепції функціонального програмування, яка є концепцією функцій вищого порядку.

**Функції вищого порядку**

Функція вищого порядку — це функція, яка має одну або обидві такі характеристики:

* Він приймає інші функції як аргументи
* Він повертає функції під час виклику

У JavaScript немає «спеціального способу» визначення функцій вищого порядку. Це просто особливість мови. Сама мова дозволяє мені передавати функцію іншій функції або повертати функцію з іншої функції. Продовжуючи попередній розділ, розглянемо наступний код, у якому я перевизначаю функцію **addTwoNums()** , щоб вона була функцією вищого порядку:

function addTwoNums(getNumber1, getNumber2) {

    console.log(getNumber1() + getNumber2());

}

Ви можете розглядати наведене вище оголошення функції **addTwoNums** як опис того, як вона матиме справу з входами **getNumber1** і **getNumber2** : як тільки вона отримає їх як аргументи, вона спробує викликати їх і об’єднати значення, повернуті цими викликами. Наприклад:

addTwoNums(specificNum, specificNum); // returned number is 84

addTwoNums(specificNum, randomNum); // returned number is 42 + some random number

**Чисті функції та побічні ефекти**

Іншою концепцією функціонального програмування є чисті функції.

Чиста функція повертає той самий результат, якщо їй надано однакові значення.

Прикладом чистої функції є  функція **addTwoNums()**  із попереднього розділу:

function addTwoNums(a, b) {

    console.log(a + b)

}

Ця функція завжди повертатиме той самий вихід на основі вхідних даних. Наприклад, якщо ми надаємо йому конкретне значення, скажімо,  **5** і  **6** :

addTwoNums(5,6); // 11

... результат завжди буде однаковим.

Ще одне правило, за яким функція вважається чистою, полягає в тому, що вона не повинна мати побічних ефектів. Побічний ефект — це будь-який випадок, коли функція вносить зміни поза собою.

Це включає:

* змінювати значення змінних поза самою функцією або навіть покладатися на зовнішні змінні
* виклик API браузера (навіть самої консолі!)
* виклик **Math.random()** - оскільки значення не можна достовірно повторити

Тема чистих і нечистих функцій може стати дещо складнішою.

Наразі достатньо знати, що ця концепція існує і що вона пов’язана з функціональним програмуванням.

**Принципи об'єктно-орієнтованого програмування**

У цій статті ви дізнаєтеся про переваги об’єктно-орієнтованого програмування (ООП) і принципи ООП.

**Переваги ООП**

Використання парадигми об’єктно-орієнтованого програмування (ООП) має багато переваг.

ООП допомагає розробникам імітувати зв’язок між об’єктами в реальному світі. У певному сенсі це допомагає вам міркувати про зв’язки між речами у вашому програмному забезпеченні, так само, як ви робили б це в реальному світі. Таким чином, ООП є ефективним підходом до пошуку рішень у коді, який ви пишете. OOP також:

* Дозволяє писати модульний код,
* Робить ваш код більш гнучким і
* Робить ваш код багаторазовим.

**Принципи ООП**

Чотири фундаментальні принципи ООП: успадкування, інкапсуляція, абстракція та поліморфізм. Ви по черзі дізнаєтеся про кожен із цих принципів. Про об’єкти слід пам’ятати, що вони існують в ієрархічній структурі. Це означає, що вихідним базовим або суперкласом для всього є клас Object, усі об’єкти походять від цього класу. Це дозволяє нам використовувати метод Object.create(). створювати або інстанціювати об’єкти наших класів.

class Animal { /\* ...class code here... \*/ }

var myDog = Object.create(Animal)

console.log (Animal)

Більш поширеним методом створення об’єктів із класів є використання ключового слова **new** . Коли використовується стандартний або порожній метод конструктора, JavaScript неявно викликає суперклас Object для створення екземпляра.

class Animal { /\* ...class code here... \*/ }

var myDog = new Animal()

console.log (Animal)

Ця концепція досліджується в наступному розділі про успадкування

**Принципи ООП: Спадкування**

Спадкування є однією з основ об'єктно-орієнтованого програмування.

По суті, це дуже проста концепція. Це працює так:

1. Існує базовий клас "речі".
2. Існує один або кілька підкласів "речей", які успадковують властивості базового класу (іноді їх також називають "суперкласом")
3. Можуть існувати деякі інші під-підкласи "речей", які успадковують ці класи в пункті 2.

Зверніть увагу, що кожен підклас успадковує свій суперклас. У свою чергу, підклас також може бути супер-класом, якщо існують класи, що успадковують цей підклас.

Все це може здатися трохи «комп’ютерним», тому ось більш практичний приклад:

1. Є базовий клас «Тварина».
2. Існує ще один клас, підклас, який успадковує «Тварина», і назва цього класу — «Птах».
3. Далі є ще один клас, успадкований від «Птахи», і це клас «Орел».

Таким чином, у наведеному вище прикладі я моделюю об’єкти з реального світу, будуючи зв’язки між Твариною, Птахом і Орлом. Кожен з них є окремим класом, тобто кожен з них є окремим кресленням для конкретних екземплярів об’єктів, які можна створювати за потреби.

Щоб налаштувати відношення успадкування між класами в JavaScript, я можу використати  ключове слово **extends**  , як у  **класі B extends A** .

Ось приклад ієрархії успадкування в JavaScript:

class Animal { /\* ...class code here... \*/ }

class Bird extends Animal { /\* ...class code here... \*/ }

class Eagle extends Bird { /\* ...class code here... \*/ }

**Принципи ООП: інкапсуляція**

Найпростіше кажучи, інкапсуляція має відношення до того, щоб зробити реалізацію коду «прихованою» від інших користувачів, у тому сенсі, що їм не потрібно знати, як працює мій код, щоб «споживати» код.

Наприклад, коли я запускаю такий код:

"abc".toUpperCase();

Мені насправді не потрібно турбуватися чи навіть витрачати час на роздуми про те, як  працює метод **toUpperCase()**  . Все, що я хочу, це використовувати його, оскільки я знаю, що він доступний для мене. Навіть якщо базовий синтаксис, тобто реалізація методу  **toUpperCase()**  , зміниться, якщо він не порушує мій код, мені не потрібно турбуватися про те, що він робить у фоновому режимі чи навіть як це робить це.

**Принципи ООП: абстракція**

Абстракція — це написання коду таким чином, щоб зробити його більш узагальненим.

Поняття інкапсуляції та абстракції часто неправильно розуміють, оскільки їхні відмінності можуть здаватися розмитими.

Допоможе подумати про це в таких термінах:

* Абстракція полягає в тому, щоб виділити  *концепцію*  того, що ви намагаєтеся зробити, а не мати справу з конкретним проявом цієї концепції.
* Інкапсуляція означає, що ви не маєте доступу до того, як певна реалізація працює внутрішньо, або вас не цікавить.

Хоча і інкапсуляція, і абстракція є важливими поняттями в ООП, для того, щоб по-справжньому заглибитися в цю тему, потрібно більше досвіду програмування загалом.

Наразі достатньо знати про їх існування в ООП.

**Принципи ООП: поліморфізм**

Поліморфізм - це слово, що походить від грецької мови і означає «множинні форми». Альтернативним перекладом може бути: «щось, що може приймати багато форм».

Отже, щоб зрозуміти, що таке поліморфізм, давайте розглянемо деякі реальні об’єкти.

* Двері мають дзвінок. Можна сказати, що дзвінок є властивістю дверного предмета. У цей дзвінок можна дзвонити. Коли б хтось подзвонив у двері? Очевидно, щоб хтось з’явився в дверях.
* А тепер розглянемо дзвінок на велосипеді. Велосипед має дзвінок. Можна сказати, що дзвін є властивістю предмета велосипеда. У цей дзвін теж можна було дзвонити. Однак причина, намір і результат того, що хтось дзвонить у дзвінок на велосипеді, не те саме, що дзвонить у двері.

Наведені вище концепції можна закодувати в JavaScript таким чином:

const bicycle = {

    bell: function() {

        return "Ring, ring! Watch out, please!"

    }

}

const door = {

    bell: function() {

        return "Ring, ring! Come here, please!"

    }

}

Отже, я можу отримати доступ до   методу  **bell() об’єкта bicycle**  , використовуючи такий синтаксис:

bicycle.bell(); // "Get away, please"

Я також можу отримати доступ до   методу  **bell() об’єкта двері**  , використовуючи цей синтаксис:

door.bell(); // "Come here, please"

На цьому етапі можна зробити висновок, що те саме ім’я методу може мати абсолютно протилежне призначення, залежно від того, для якого об’єкта воно використовується.

Тепер, щоб зробити цей код справді поліморфним, я додам ще одне оголошення функції:

function ringTheBell(thing) {

    console.log(thing.bell())

}

Тепер я оголосив  функцію **ringTheBell()**  . Він приймає  параметр **речі**  , який, як я очікую, буде об’єктом, а саме, або  об’єкт **велосипеда**  , або  об’єкт **двері**  .

Отже, якщо я викликаю  функцію **ringTheBell()**  і передам їй  **велосипед**  як єдиний аргумент, ось результат:

ringTheBell(bicycle); // Ring, ring! Watch out, please!

Однак, якщо я викликаю  функцію **ringTheBell()**  і передам їй  об’єкт **двері**  , я отримаю такий результат:

ringTheBell(door); // "Ring, ring! Come here, please!"

Тепер ви бачили приклад, коли одна й та сама функція дає різні результати  **залежно від контексту**  , у якому вона використовується.

Ось ще один приклад оператора конкатенації, який використовується під час виклику вбудованого  методу **concat()**  .

Якщо я використовую  метод **concat()**  для двох рядків, він поводиться так само, як якщо б я використовував  оператор **+**  .

"abc".concat("def"); // 'abcdef'

Я також можу використовувати  метод **concat()**  для двох масивів. Ось результат:

["abc"].concat(["def"]); // ['abc', 'def']

Розгляньте можливість використання  оператора **+**  для двох масивів з одним членом у кожному:

["abc"] + ["def"]; // ["abcdef"]

Це означає, що  метод **concat()**  демонструє поліморфну ​​поведінку, оскільки він поводиться по-різному залежно від контексту - у цьому випадку залежно від того, які типи даних я йому надаю.

Повторюємо, що поліморфізм корисний, оскільки він дозволяє розробникам створювати об’єкти, які можуть мати однакову функціональність, а саме функції з однаковою назвою, які поводяться однаково. Однак у той же час ви можете перевизначити деякі частини спільної функціональності або навіть повну функціональність у деяких інших частинах структури ООП.

Ось приклад поліморфізму з використанням класів у JavaScript:

class Bird {

    useWings() {

        console.log("Flying!")

    }

}

class Eagle extends Bird {

    useWings() {

        super.useWings()

        console.log("Barely flapping!")

    }

}

class Penguin extends Bird {

    useWings() {

        console.log("Diving!")

    }

}

var baldEagle = new Eagle();

var kingPenguin = new Penguin();

baldEagle.useWings(); // "Flying! Barely flapping!"

kingPenguin.useWings(); // "Diving!"

 Підкласи  Penguin   і  **Eagle обидва** успадковують  суперклас **Bird** . Підклас  **Eagle**  успадковує  метод **useWings()**  від  класу **Bird**  , але розширює його додатковим журналом консолі.  Підклас Penguin не успадковує  клас  **useWings() – замість цього**  він має власну реалізацію, хоча  сам клас **Penguin**  розширює  клас **Bird**  .

**Конструктори**

У JavaScript є ряд вбудованих типів об’єктів, наприклад:

**Math** ,  **Date** ,  **Object** ,  **Function** ,  **Boolean** ,  **Symbol** ,  **Array** ,  **Map** ,  **Set** ,  **Promise** ,  **JSON** тощо.

Ці об’єкти іноді називають «рідними об’єктами».

Функції-конструктори, які зазвичай називають просто «конструкторами», — це спеціальні функції, які дозволяють нам створювати екземпляри цих вбудованих нативних об’єктів. Усі конструктори написані з великої літери.

Щоб використовувати функцію-конструктор, я повинен додати перед нею оператор  **new** .

Наприклад, щоб створити новий екземпляр об’єкта  **Date**  , я можу запустити:  **new Date()** . Я повертаю поточну дату й час, наприклад:

**Чт, 3 лютого 2022 р. 11:24:08 GMT+0100 (Центральноєвропейський стандартний час)**

Однак не всі вбудовані об’єкти мають функцію конструктора. Прикладом такого типу об’єкта є вбудований  об’єкт **Math**  .

Запуск  **new Math()**  видає  **Uncaught TypeError** , повідомляючи нас, що  **Math не є конструктором** .

Таким чином, я можу зробити висновок, що деякі вбудовані об’єкти дійсно мають конструктори, коли вони служать певній меті: дозволити нам створити конкретний екземпляр конструктора даного об’єкта. Вбудований  об’єкт **Date**  ідеально підходить для наявності конструктора, тому що кожен новий екземпляр об’єкта date, який я створюю, повинен мати унікальні дані за визначенням, оскільки це матиме іншу позначку часу – він буде створений в інший момент часу.

Інші вбудовані об’єкти, які не мають конструкторів, наприклад  об’єкт **Math**  , не потребують конструктора. Це просто статичні об’єкти, до властивостей і методів яких можна отримати прямий доступ із самого вбудованого об’єкта. Іншими словами, немає сенсу створювати екземпляр вбудованого  об’єкта **Math**  , щоб мати можливість використовувати його функціональність.

Наприклад, якщо я хочу використовувати   метод  **pow об’єкта Math** для обчислення експоненційних значень,  для цього  не потрібно створювати екземпляр об’єкта  **Math** . Наприклад, щоб перевести число 2 у ступінь 5, я б виконав:

**Math.pow(2,5); // --> 32**

Немає потреби створювати екземпляр об’єкта  **Math**  , оскільки в екземплярі цього конкретного об’єкта не буде нічого, що потрібно зберігати.

Окрім функцій конструктора для вбудованих об’єктів, я також можу визначити власні функції конструктора.

Ось приклад:

function Icecream(flavor) {

    this.flavor = flavor;

    this.meltIt = function() {

        console.log(`The ${this.flavor} icecream has melted`);

    }

}

# 

# **Створення класів**

Тепер ви повинні знати, що успадкування в JavaScript базується на об’єкті-прототипі.

Усі об’єкти, створені з прототипу, мають однакову функціональність.

Коли вам потрібно закодувати складніші ООП-зв’язки, ви можете використати  ключове слово **class**  і його простий для розуміння та міркування синтаксис.

Уявіть, що вам потрібно закодувати  клас **Train**  .

Після того, як ви закодували цей клас, ви зможете використовувати ключове слово  **new**  для створення екземплярів об’єктів  класу **Train**  .

Однак наразі вам спочатку потрібно визначити  клас **Train**  , використовуючи такий синтаксис:

class Train {}

Отже, ви використовуєте  ключове слово **class**  , потім вказуєте ім’я свого класу з великої першої літери, а потім додаєте відкриваючу та закриваючу фігурні дужки.

Між фігурними дужками першим фрагментом коду, який потрібно визначити, є  **конструктор** :

class Train {

    constructor() {

    }

}

Конструктор   використовуватиметься для створення властивостей майбутнього екземпляра об’єкта  **класу Train**  .

Наразі припустімо, що кожен екземпляр об’єкта  класу **Train**  повинен мати лише дві властивості під час створення екземпляра:  **color** і  **lightsOn** .

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

}

Зверніть увагу на синтаксис конструктора. Конструктор — це спеціальна функція в моєму   класі **Train .**

Перш за все, зауважте, що немає  ключового слова **function**  . Також зауважте, що   для визначення цієї функції використовується **конструктор** ключового слова. Ви вказуєте  параметри функції **-конструктора**  всередині відкриваючої та закриваючої дужок, як і у звичайних функціях. Назви параметрів:  **color**  і  **lightsOn** .

Далі в  тілі функції- **конструктора**  ви призначили передане   значення параметра  **кольору this.color** , а передане   значення параметра  **lightsOn — this.lightsOn** .

Що означає це  **ключове**  слово?

**Це майбутній екземпляр об’єкта   класу Train** .

По суті, це весь код, який потрібно написати, щоб досягти двох речей:

1. Цей код дозволяє мені  **створювати нові екземпляри   класу Train** .
2. Кожен екземпляр об’єкта  класу **Train**  , який я створюю, матиме власні властивості  **color**  і  **lightsOn** .

Тепер, щоб створити новий екземпляр класу  **Train**  , мені потрібно використовувати наступний синтаксис:

new Train()

Усередині круглих дужок вам потрібно передати такі значення, як  **«red»** і  **false** , наприклад, що означає, що для  властивості **color**  встановлено значення  **«red»** , а  для властивості **lightsOn**  — значення  **false** .

І щоб мати можливість взаємодіяти з новим об’єктом, створеним таким чином, вам потрібно призначити його змінній.

Зібравши все разом, ось ваш перший потяг:

var myFirstTrain = new Train('red', false);

Як і будь-яка інша змінна, тепер ви можете, наприклад, консольно реєструвати  об’єкт **myFirstTrain**  :

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

Ви можете продовжити створення екземплярів  класу **Train**  . Навіть якщо ви надасте їм абсолютно однакові властивості, вони все одно будуть окремими об’єктами.

var mySecondTrain = new Train('blue', false);

var myThirdTrain = new Train('blue', false);

Однак це не все, що можуть запропонувати заняття.

Ви також можете додавати методи до класів, і ці методи будуть спільні для всіх майбутніх екземплярів об’єктів мого  класу **Train**  .

Наприклад:

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

 Тепер у вашому класі  **Train** є чотири методи  : **toggleLights()** ,  **lightsStatus()** ,   **getSelf()**  і  **getPrototype()** .

1. Метод  **toggleLights**  використовує логічний оператор not,  **!** . Цей оператор змінить значення, що зберігається у  властивості **lightsOn**  майбутнього об’єкта екземпляра  класу **Train**  ; отже  **!this.lightsOn** . А  оператор **=**  ліворуч означає, що його буде призначено  **this.lightsOn** , що означає, що він стане новим значенням  властивості **lightsOn**  для даного екземпляра об’єкта.
2. Метод lightsStatus  **()**  у  класі **Train**  просто повідомляє про поточний статус  змінної **lightsOn**  даного екземпляра об’єкта.
3. Метод  **getSelf()**  друкує властивості екземпляра об’єкта, який він викликав.
4. Консоль  **getPrototype()**  реєструє прототип екземпляра об’єкта  класу **Train**  . Прототип містить усі властивості, спільні для всіх екземплярів об’єктів класу  **Train**  . Щоб отримати прототип, ви використаєте вбудований метод JavaScript  **Object.getPrototypeOf()**  і передасте йому  **цей**  об’єкт, тобто екземпляр об’єкта, у якому цей метод викликається.

Тепер ви можете створити абсолютно новий поїзд, використовуючи цей оновлений   клас **поїздів :**

var train4 = new Train('red', false);

І тепер ви можете запустити кожен із його методів один за одним, щоб підтвердити їхню поведінку:

train4.toggleLights(); // undefined

train4.lightsStatus(); // Lights on? true

train4.getSelf(); // Train {color: 'red', lightsOn: true}

train4.getPrototype(); // {constructor: f, toggleLights: f, ligthsStatus: f, getSelf: f, getPrototype: f}

Результатом виклику  **toggleLights()**  є зміна true на false і навпаки для  властивості **lightsOn**  .

Результатом виклику  **lightsStatus()**  є консольне реєстрування значення властивості  **lightsOn**  .

Результатом виклику  **getSelf()**  є реєстрація консоллю всього екземпляра об’єкта, у якому   викликається метод **getSelf()** . У цьому випадку поверненим об’єктом є  об’єкт **train4**  . Зверніть увагу, що цей об’єкт повертається лише з властивостями («даними»), які були створені за допомогою   функції  **constructor() класу Train**  . Це тому, що всі методи  класу **Train**  не «живуть» на жодному з екземплярів об’єктів класу  **Train**  — натомість вони живуть у прототипі, як буде підтверджено в наступному параграфі.

Нарешті, результатом виклику  методу **getPrototype()**  є реєстрація в консолі всіх властивостей  **прототипу** . Коли  синтаксис **класу**  використовується в JavaScript, це призводить до того, що  **в прототипі зберігаються лише спільні методи** , тоді як  функція **constructor()**  налаштовує механізм для збереження значень екземпляра («даних») під час створення екземпляра об’єкта.

Таким чином, підводячи підсумок, синтаксис класу в JavaScript дозволяє нам чітко відокремити дані окремого об’єкта – які існують у самому екземплярі об’єкта – від функціональних можливостей (методів) спільного об’єкта, які існують у прототипі та є спільними для всіх екземплярів об’єкта.

Однак це ще не вся історія.

Можна реалізувати поліморфізм за допомогою класів у JavaScript, успадкувавши від базового класу, а потім перевизначивши успадковану поведінку. Щоб зрозуміти, як це працює, найкраще скористатися прикладом.

У наведеному нижче коді ви спостерігатимете, як кодується інший клас, який називається  **HighSpeedTrain** і успадковує  клас **Train**  .

Це робить  клас **Train**  базовим класом або суперкласом класу  **HighSpeedTrain**  . Іншими словами,  клас **HighSpeedTrain**  стає підкласом  класу **Train**  , оскільки він успадковує від нього.

Щоб успадкувати від одного класу новий підклас, JavaScript надає  ключове слово **extends**  , яке працює наступним чином:

class HighSpeedTrain extends Train {

}

Як і в наведеному вище прикладі, синтаксис підкласу узгоджується з визначенням базового класу в JavaScript. Єдиним доповненням тут є  ключове слово **extends**  та ім’я класу, від якого успадковується підклас.

Тепер ви можете описати, як  працює **HighSpeedTrain**  . Знову ж таки, ви можете почати з визначення функції конструктора:

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

}

Зверніть увагу на невелику різницю в синтаксисі в конструкторі  класу **HighSpeedTrain**  , а саме на використання  ключового слова **super**  .

У класах JavaScript  **super**  використовується для визначення того, яка властивість успадковується від суперкласу в підкласі.

У цьому випадку я вирішив успадкувати обидві властивості від   суперкласу  **Train** у  підкласі **HighSpeedTrain .**

Ці властивості —  **колір**  і  **lightsOn** .

Далі ви додаєте додаткові властивості класу HighSpeedTrain у його конструктор, а саме властивості пасажирів і highSpeedOn.

Далі в тілі конструктора ви використовуєте  ключове слово **super**  і передаєте успадковані  властивості **color**  і  **lightsOn**  , які надходять із  класу **Train**  . На наступних лініях ви призначаєте  **пасажирів**  this.passengers  ,  **а highSpeedOn —**  this.highSpeedOn  .

Зауважте, що на додаток до успадкованих властивостей, ви також  **автоматично успадковуєте**  всі методи, які існують у  прототипі **Train**  , а саме, методи  **toggleLights()** ,  **lightsStatus()** ,  **getSelf()** і  **getPrototype()**  .

Тепер давайте додамо інший метод, який буде специфічним для  класу **HighSpeedTrain**  :  метод **toggleHighSpeed()**  .

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

}

Крім того, уявіть, що ви зрозуміли, що вам не подобається, як  працює метод **toggleLights()**  із суперкласу, і ви хочете реалізувати його трохи по-іншому в підкласі. Ви можете додати його в  клас **HighSpeedTrain**  .

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLigths();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}

Отже, як ви перевизначили поведінку вихідного  методу **toggleLights()**  ?

У супер-класі  метод **toggleLights()**  був визначений наступним чином:

toggleLights() {

    this.lightsOn = !this.lightsOn;

}

Ви зрозуміли, що  метод **HighSpeedTrain**  має повторно використовувати існуючу поведінку вихідного  методу **toggleLights()**  , тому ви використали  синтаксис **super.toggleLights(),**  щоб успадкувати весь метод суперкласу.

**Далі ви також успадковуєте поведінку методу lightsStatus()** суперкласу   , оскільки ви розумієте, що хочете, щоб оновлений статус  властивості **lightsOn**  реєструвався на консолі щоразу, коли ви викликаєте  метод **toggleLights()**  у підпорядкованому клас.

Нарешті, ви також додаєте третій рядок у повторно реалізований  метод **toggleLights()**  , а саме:

console.log('Lights are 100% operational.');

Ви додали цей третій рядок, щоб показати, що я можу поєднати «позичений» код методу з суперкласу з вашим власним кодом у підкласі.

Тепер ви готові побудувати кілька об’єктів поїзда.

var train5 = new Train('blue', false);

var highSpeed1 = new HighSpeedTrain(200, false, 'green', false);

Ви створили  об’єкт **train5**  класу  **Train**  і встановили для нього  **колір «синій»,**  а   для  **lightsOn**  —  **false** .

Далі ви створили  об’єкт **highSpeed1**  для  класу **HighSpeedTrain**  , встановивши для  **пасажирів**  значення  **200** ,  для **highSpeedOn**  значення  **false** ,  **колір**  —  **«зелений»** і lightsOn — значення false.

Тепер ви можете перевірити поведінку  **train5** , викликавши, наприклад,  метод **toggleLights()**  , а потім  метод **lightsStatus()**  :

train5.toggleLights(); // undefined

train5.lightsStatus(); // Lights on? true

Ось повний готовий код для цього уроку:

class Train {

    constructor(color, lightsOn) {

        this.color = color;

        this.lightsOn = lightsOn;

    }

    toggleLights() {

        this.lightsOn = !this.lightsOn;

    }

    lightsStatus() {

        console.log('Lights on?', this.lightsOn);

    }

    getSelf() {

        console.log(this);

    }

    getPrototype() {

        var proto = Object.getPrototypeOf(this);

        console.log(proto);

    }

}

class HighSpeedTrain extends Train {

    constructor(passengers, highSpeedOn, color, lightsOn) {

        super(color, lightsOn);

        this.passengers = passengers;

        this.highSpeedOn = highSpeedOn;

    }

    toggleHighSpeed() {

        this.highSpeedOn = !this.highSpeedOn;

        console.log('High speed status:', this.highSpeedOn);

    }

    toggleLights() {

        super.toggleLights();

        super.lightsStatus();

        console.log('Lights are 100% operational.');

    }

}

var myFirstTrain = new Train('red', false);

console.log(myFirstTrain); // Train {color: 'red', lightsOn: false}

var mySecondTrain = new Train('blue', false);

Зверніть увагу на те, що  метод **toggleLights()**  поводиться інакше в  класі **HighSpeedTrain,**  ніж у  класі **Train**  .

Крім того, це допомагає візуалізувати те, що відбувається, отримавши прототип  поїздів **train5**  і  **highSpeed1** :

train5.getPrototype(); // {constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}

highSpeed1.getPrototype(); // Train {constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}

Повернуті значення в цьому випадку спочатку можуть здатися трохи складними для розуміння, але насправді це досить просто:

1. Об’єкт-прототип об’єкта  **train5**  було створено, коли ви визначили клас  **Train** . Ви можете отримати доступ до прототипу за допомогою  синтаксису **Train.prototype**  і повернути об’єкт прототипу.
2. Об’єктом-прототипом об’єкта  **highSpeed1**  є цей об’єкт: **{constructor: ƒ, toggleHighSpeed: ƒ, toggleLights: ƒ}** . У свою чергу, цей об’єкт має власний прототип, який можна знайти за допомогою такого синтаксису:  **HighSpeedTrain.prototype.\_\_proto\_\_** . Запуск цього коду повертає:  **{constructor: ƒ, toggleLights: ƒ, lightsStatus: ƒ, getSelf: ƒ, getPrototype: ƒ}** .

Прототипи здаються легкими для розуміння на певному рівні, але легко загубитися в складності. Це одна з причин, чому синтаксис класів у JavaScript покращує роботу розробника, полегшуючи міркування про зв’язки між класами. Однак, удосконалюючи свої навички, ви завжди повинні прагнути краще розуміти свої інструменти, зокрема прототипи. Зрештою, JavaScript — це лише інструмент, а ви зараз «зазирнули за шторку».

У цьому читанні ви дізналися саму суть того, як ООП із класами працює в JavaScript. Однак це ще не все.

На уроці з проектування об’єктно-орієнтованої програми ви дізнаєтеся ще кілька корисних понять. Здебільшого вони пов’язані з кодуванням ваших класів, щоб було ще легше створювати екземпляри об’єктів цих класів у JavaScript.

**Використання екземпляра класу як властивості конструктора іншого класу**

Розглянемо такий приклад:

class StationaryBike {

    constructor(position, gears) {

        this.position = position

        this.gears = gears

    }

}

class Treadmill {

    constructor(position, modes) {

        this.position = position

        this.modes = modes

    }

}

class Gym {

    constructor(openHrs, stationaryBikePos, treadmillPos) {

        this.openHrs = openHrs

        this.stationaryBike = new StationaryBike(stationaryBikePos, 8)

        this.treadmill = new Treadmill(treadmillPos, 5)

    }

}

var boxingGym = new Gym("7-22", "right corner", "left corner")

console.log(boxingGym.openHrs) //

console.log(boxingGym.stationaryBike) //

console.log(boxingGym.treadmill) //

У цьому прикладі визначено три класи: **StationaryBike** , **Treadmill** і **Gym** .

Клас **StationaryBike** закодовано таким чином, що його майбутній екземпляр об’єкта матиме властивості **position** та **gears** . Властивість **position** описує, де в тренажерному залі буде розміщено велотренажер, а властивість **gears** визначає кількість передач, яку має мати цей велотренажер.

Клас **Treadmill** також має позицію та іншу властивість, що називається **режимами** (як у «режимах вправи»).

Клас **Gym** має три параметри у своїй функції-конструкторі: **openHrs** , **stationaryBikePos** , **treadmillPos** .

Цей код дозволяє мені створити екземпляр нового екземпляра об’єкта класу **Gym** , а потім, перевіряючи його, я отримую таку інформацію:

* **властивість openHrs** дорівнює " **7-22"** (тобто з 7 ранку до 10 вечора)
* Властивість **stationaryBike** є об'єктом типу **StationaryBike** , що містить дві властивості: **position** і **gears**
* властивість **treadmill** є об'єктом типу **Treadmill** , що містить дві властивості: **положення** та **режими**

тиСкинути

Тепер я можу зробити скільки завгодно морозива:

function Icecream(flavor) {

    this.flavor = flavor;

    this.meltIt = function() {

        console.log(`The ${this.flavor} icecream has melted`);

    }

}

let kiwiIcecream = new Icecream("kiwi");

let appleIcecream = new Icecream("apple");

kiwiIcecream; // --> Icecream {flavor: 'kiwi', meltIt: ƒ}

appleIcecream; // --> Icecream {flavor: 'apple', meltIt: ƒ}

Я щойно створив два екземпляри об’єктів  типу **Icecream**  .

Найпоширенішим варіантом використання  **new**  є використання його з одним із вбудованих типів об’єктів.

Зауважте, що використання функцій конструктора для всіх вбудованих об’єктів іноді є не найкращим підходом.

Особливо це стосується конструкторів об’єктів примітивних типів, а саме:  **String** ,  **Number** і  **Boolean** .

Наприклад, за допомогою вбудованого  конструктора **String**  я можу створити нові рядки:

let  apple =  new String ( "яблуко" );

яблуко;  // --> рядок {'apple'}

**Змінна apple** є   об'єктом типу  **String** .

Давайте подивимося, чим  об’єкт **apple**  відрізняється від наступної  змінної **pear**  :

let pear = "pear";

pear; // --> "pear"

Змінна  **pear**  — це рядковий літерал, тобто примітивне значення Javascript.

Змінна  **pear**  , будучи примітивним значенням, завжди буде більш продуктивною, ніж  змінна **apple**  , яка є об’єктом.

Крім більшої продуктивності, через те, що кожен об’єкт у JavaScript унікальний, ви не можете порівняти об’єкт String з іншим об’єктом String, навіть якщо їхні значення ідентичні.

Іншими словами, якщо ви порівнюєте  **new String('plum') === new String('plum')** , ви отримаєте  **false** , тоді як  **"plum" === "plum"**  повертає true. Ви отримуєте  **false**  під час порівняння об’єктів, тому що порівнюються не значення, які ви передаєте конструктору, а місце розташування пам’яті, де зберігаються об’єкти.

Окрім невикористання конструкторів для побудови об’єктних версій примітивів, вам краще не використовувати конструктори під час побудови простих звичайних об’єктів.

Замість  **new Object** ви повинні дотримуватися синтаксису об’єктного літералу:  **{}** .

Об’єкт RegExp — ще один вбудований об’єкт у JavaScript. Він використовується для **зіставлення рядків за зразком** за допомогою так званих «регулярних виразів». Регулярні вирази існують у багатьох мовах, не лише в JavaScript.

У JavaScript ви можете створити екземпляр конструктора RegExp за допомогою **new RegExp** .

Крім того, ви можете використовувати літерал шаблону замість RegExp. Ось приклад використання **/d/** як літералу шаблону, переданого як аргумент методу **match** у рядку.

"abcd".match(/d/); // ['d', index: 3, input: 'abcd', groups: undefined]

"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]

Замість використання  конструкторів **Array** ,  **Function** і  **RegExp**  ви повинні використовувати їхні різновиди літералів масиву, літералів функцій і літералів шаблону:  **[]** ,  **() {}** і  **/()/** .

Однак, будуючи об'єкти інших вбудованих типів, ми можемо використовувати конструктор.

Ось кілька прикладів:

new Date();

new Error();

new Map();

new Promise();

new Set();

new WeakSet();

new WeakMap();

Наведений вище список не є вичерпним, але він лише для того, щоб дати вам уявлення про деякі функції конструктора, які ви можете використовувати.

**Параметри за замовчуванням**

Корисна функція ES6 дозволяє мені встановити параметр за замовчуванням у визначенні функції По-перше, .

Це означає, що я буду використовувати функцію ES6, яка дозволяє мені встановити параметр за замовчуванням у визначенні функції, що йде рука об руку з підходом до захисного кодування, але не вимагає майже жодних зусиль для реалізації.

Наприклад, розглянемо оголошення функції без параметрів за замовчуванням:

function noDefaultParams(number) {

    console.log('Result:', number \* number)

}

Очевидно, що  функція **noDefaultParams**  має повертати будь-яке отримане число  *в квадраті* .

Але що, якби я назвав це так:

noDefaultParams(); // Result: NaN

Через свою динамічну природу JavaScript не видає помилки, але повертає безглуздий результат.

Тепер розглянемо наступне вдосконалення, використовуючи параметри за замовчуванням:

function withDefaultParams(number = 10) {

    console.log('Result:', number \* number)

}

Параметри за замовчуванням дозволяють мені побудувати функцію, яка працюватиме зі значеннями аргументів за замовчуванням, навіть якщо я не передаю їй жодних аргументів, і водночас залишаюся достатньо гнучким, щоб дозволити мені передавати значення користувацьких аргументів і працювати з ними відповідно.

Тепер це дозволяє мені кодувати свої класи таким чином, щоб полегшити створення екземплярів об’єктів.

Розглянемо таке визначення класу:

class NoDefaultParams {

    constructor(num1, num2, num3, string1, bool1) {

        this.num1 = num1;

        this.num2 = num2;

        this.num3 = num3;

        this.string1 = string1;

        this.bool1 = bool1;

    }

    розрахувати() {

        if(this.bool1) {

            console.log(this.string1, this.num1 + this.num2 + this.num3);

            return;

        }

        return "The value of bool1 is incorrect"

    }

}

Тепер я створю екземпляр об’єкта  класу **NoDefaultParams**  і запусту   для нього метод **calculate()** . Очевидно, що  **bool1**  має бути встановлено на  **true**  під час виклику, щоб це працювало, але я встановлю його на false навмисно, щоб підкреслити те, що я маю на увазі.

var fail = new NoDefaultParams(1,2,3,false);

fail.calculate(); // 'The value of bool1 is incorrect'

Цей приклад може підкреслити причину, чому іноді з’являються дивні повідомлення про помилки під час використання певного програмного забезпечення – можливо, розробники просто не мали достатньо часу, щоб покращити його.

Однак тепер, коли ви знаєте про параметри за замовчуванням, цей приклад можна покращити наступним чином:

class WithDefaultParams {

    constructor(num1 = 1, num2 = 2, num3 = 3, string1 = "Result:", bool1 = true) {

        this.num1 = num1;

        this.num2 = num2;

        this.num3 = num3;

        this.string1 = string1;

        this.bool1 = bool1;

    }

    розрахувати() {

        if(this.bool1) {

            console.log(this.string1, this.num1 + this.num2 + this.num3);

            return;

        }

        return "The value of bool1 is incorrect"

    }

}

var better = new WithDefaultParams();

better.calculate(); // Result: 6

Цей підхід покращує роботу розробника з моїм кодом, оскільки мені більше не потрібно турбуватися про те, щоб надати  класу **WithDefaultParameters**  усі аргументи. Для швидких тестів це чудово, тому що мені більше не потрібно турбуватися про передачу відповідних аргументів.

Крім того, цей підхід справді блищить при побудові ієрархій успадкування за допомогою класів, оскільки він дає змогу надавати лише власні властивості в підкласі, приймаючи всі параметри за замовчуванням від конструктора суперкласу.

Підсумовуючи, у цьому читанні я розглянув наступне:

* Як підійти до розробки об’єктно-орієнтованої програми на JavaScript
* Роль   ключових слів **extends**  і  **super**
* Важливість використання параметрів за замовчуванням.

**Розробка об’єктно-орієнтованих програми**

У цьому читанні я покажу вам, як створювати класи в JavaScript, використовуючи всі концепції, які ви вивчили досі.

Зокрема, я готуюся створити таку ієрархію успадкування:

Тварина / \ Кіт Птах / \ \ Домашній Кіт Тигровий Папуга

Є два ключові слова, які є важливими для ООП із класами в JavaScript.

Ці ключові слова —  **extends**  і  **super** .

Ключове  слово **extends**  дозволяє мені успадковувати існуючий клас.

Базуючись на наведеній вище ієрархії, я можу закодувати  клас **Animal**  так:

class Animal {

    // ... class code here ...

}

Тоді я можу закодувати, наприклад,  підклас **Cat**  так:

class Cat extends Animal {

    // ... class code here ...

}

Ось як  ключове слово **extends**  використовується для встановлення зв’язків успадкування.

Ключове  слово **super**  дозволяє мені «позичити» функціональність у супер-класу в підкласі. Точну динаміку того, як це працює, буде розглянуто пізніше в цьому уроці.

Тепер я можу почати думати про те, як реалізувати свою ієрархію класів ООП.

Перш ніж почати, мені потрібно подумати про такі речі, як: \* Що має входити до базового класу  **Animal** ? Іншими словами, що всі підкласи успадкують від базового класу? \* Які конкретні властивості та методи відрізняють кожен клас від інших? \* Загалом, як мої заняття будуть пов’язані між собою?

Після того, як я все продумаю, я зможу будувати свої класи.

Отже, мій план такий:

1.  Конструктор класу **Animal**  матиме дві властивості:  **колір**  та  **енергію**

2.  Прототип класу **Animal**  матиме три методи:  **isActive()** ,  **sleep()** і  **getColor()** .

3.  Метод **isActive()**  під час кожного запуску знижуватиме значення  **енергії**  , поки воно не досягне  **0** . Метод  **isActive()**  також повідомляє про оновлене значення  **енергії** . Якщо  **енергія**  дорівнює нулю, тваринний об’єкт негайно перейде в сплячий режим, викликавши  метод **sleep()**  на основі зазначеної умови.

4.  Метод **getColor()**  просто реєструватиме консольне значення  властивості **color**  .

5.  Клас **Cat**  буде успадкований від  **Animal** з додатковими  властивостями **звуку** ,  **canJumpHigh** і  **canClimbTrees**  , характерними для  класу **Cat**  . Він також матиме власний  метод **makeSound()**  .

6.  Клас **Bird**  також буде успадкований від  **Animal** , але його особливі властивості будуть суттєво відрізнятися від  **Cat** . А саме,  клас **Bird**  матиме  властивості **sound**  і  **canFly**  , а  також метод **makeSound**  .

7.  Клас **HouseCat**  розширить  клас **Cat**  і матиме власний  **houseCatSound**  як спеціальну властивість. Крім того, він замінить  метод **makeSound()**  із  класу **Cat**  , але зробить це цікавим способом. Якщо  метод **makeSound()**  під час виклику отримує єдиний  аргумент **опції**  — встановлений на  **true** , тоді він запустить  **super.makeSound()**  — іншими словами, запустить код із батьківського класу ( **Cat** ) із додаванням запуску  **console.log(this.houseCatSound)** . По суті, це означає, що   метод  **makeSound() об’єкта екземпляра класу HouseCat**  матиме дві окремі поведінки залежно від того, передаємо ми його  **true**  чи  **false** .

8.  Клас **Tiger**  також буде успадкований від  **Cat** і матиме власну  властивість **tigerSound**  , тоді як решта поведінки буде майже такою ж, як у  класі **HouseCat**  .

9. Нарешті,  клас **Parrot**  розширить  клас **Bird**  за допомогою власної  властивості **canTalk**  і власного  методу **makeSound()** , який працює з двома умовними умовами: один, який перевіряє, чи  було передано   значення  **true makeSound**  під час виклику, а інший, що перевіряє значення, що зберігається у  властивості **this.canTalk**  .

Тепер, коли я повністю пояснив, як повинен працювати весь код у моїй ієрархії класів, я міг би почати його реалізацію, додавши всі вимоги як коментарі до структури коду.

На цьому етапі з усіма вимогами, записаними у вигляді коментарів, мій код має бути таким:

class Animal {

    // constructor: color, energy

    // isActive()

        // if energy > 0, energy -=20, console log energy

        // else if energy <= 0, sleep()

    // sleep()

        // energy += 20

        // console.log energy

}

class Cat extends Animal {

    // constructor: sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound()

        // console.log sound

}

class Bird extends Animal {

    // constructor: sound, canFly, color, energy

    // makeSound()

        // console.log sound

}

class HouseCat extends Cat {

    // constructor: houseCatSound, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // console.log(houseCatSound)

}

class Tiger extends Cat {

    // constructor: tigerSound, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // console.log(tigerSound)

}

class Parrot extends Bird {

    // constructor: canTalk, sound, canJumpHigh, canClimbTrees, color, energy

    // makeSound(option)

        // if (option)

            // super.makeSound()

        // if (canTalk)

            // console.log("talking!")

Тепер, коли я закодував свої вимоги в коментарях порожніх класів, я можу почати кодувати кожен клас відповідно до своїх специфікацій.

**Кодування   класу Animal**

Спочатку я закодую базовий  клас **Animal**  .

class Animal {

    constructor(color = 'yellow', energy = 100) {

        this.color = color;

        this.energy = energy;

    }

    isActive() {

        if(this.energy > 0) {

            this.energy -= 20;

            console.log('Energy is decreasing, currently at:', this.energy)

        } else if(this.energy == 0){

            this.sleep();

        }

    }

    сон() {

        this.energy += 20;

        console.log('Energy is increasing, currently at:', this.energy)

    }

    getColor() {

        console.log(this.color)

    }

}

Кожен тваринний об’єкт, яким би він не був, матиме спільні властивості  **кольору**  та  **енергії** .

Тепер я можу кодувати  класи **Cat**  і  **Bird**  :

class Cat extends Animal {

    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canClimbTrees = canClimbTrees;

        this.canJumpHigh = canJumpHigh;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

    makeSound() {

        console.log(this.sound);

    }

}

Примітка: якби я не використовував  ключове слово **super**  у наших підкласах, після виконання наведеного вище коду я отримав би таку помилку:  **Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' або повернення з похідного конструктора.**

І тепер я можу закодувати три класи, що залишилися:  **HouseCat** ,  **Tiger** і  **Parrot** .

class HouseCat extends Cat {

    constructor(houseCatSound = "meow", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.houseCatSound = houseCatSound;

    }

    makeSound(параметр) {

        if (option) {

            super.makeSound();

        }

        console.log(this.houseCatSound);

    }

}

class Tiger extends Cat {

    constructor(tigerSound = "Roar!", sound,canJumpHigh,canClimbTrees, color,energy) {

        super(sound,canJumpHigh,canClimbTrees, color,energy);

        this.tigerSound = tigerSound;

    }

    makeSound(параметр) {

        if (option) {

            super.makeSound();

        }

        console.log(this.tigerSound);

    }

}

class Parrot extends Bird {

    constructor(canTalk = false, sound,canFly, color,energy) {

        super(sound,canFly, color,energy);

        this.canTalk = canTalk;

    }

    makeSound(параметр) {

        if (option) {

            super.makeSound();

        }

        if (this.canTalk) {

            console.log("I'm a talking parrot!");

        }

    }

}

Тепер, коли ми налаштували всю цю структуру успадкування, ми можемо створювати різні тваринні об’єкти.

Наприклад, я можу побудувати двох папуг: один може говорити, а другий – ні.

var polly = new Parrot(true); // we're passing `true` to the constructor so that polly can talk

var fiji = new Parrot(false); // we're passing `false` to the constructor so that fiji can't talk

polly.makeSound(); // 'chirp', 'I'm a talking parrot!'

fiji.makeSound(); // 'chirp'

polly.color; // yellow

polly.energy; // 100

polly.isActive(); // Energy is decreasing, currently at: 80

var penguin = new Bird("shriek", false, "black and white", 200); // setting all the custom properties

penguin; // Bird {color: 'black and white', energy: 200, sound: 'shriek', canFly: false }

penguin.sound; // 'shriek'

penguin.canFly; // false

penguin.color; // 'black and white'

penguin.energy; // 200

penguin.isActive(); // Energy is decreasing, currently at: 180

Також я можу побудувати домашнього кота:

var leo = new HouseCat();

Тепер я можу  муркотіти **Лео**  :

// leo, no purring please:

leo.makeSound(false); // meow

// leo, both purr and meow now:

leo.makeSound(true); // purr, meow

Крім того, я можу побудувати тигра:

var cuddles = new Tiger();

Мій  **тигр**  вміє муркотіти й ревіти, або просто ревіти:

cuddles.makeSound(false); // Roar!

cuddels.makeSound(true); // purr, Roar!

Ось повний код із цього уроку для легшого копіювання:

class Animal {

    constructor(color = 'yellow', energy = 100) {

        this.color = color;

        this.energy = energy;

    }

    isActive() {

        if(this.energy > 0) {

            this.energy -= 20;

            console.log('Energy is decreasing, currently at:', this.energy)

        } else if(this.energy == 0){

            this.sleep();

        }

    }

    сон() {

        this.energy += 20;

        console.log('Energy is increasing, currently at:', this.energy)

    }

    getColor() {

        console.log(this.color)

    }

}

class Cat extends Animal {

    constructor(sound = 'purr', canJumpHigh = true, canClimbTrees = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canClimbTrees = canClimbTrees;

        this.canJumpHigh = canJumpHigh;

    }

    makeSound() {

        console.log(this.sound);

    }

}

class Bird extends Animal {

    constructor(sound = 'chirp', canFly = true, color, energy) {

        super(color, energy);

        this.sound = sound;

        this.canFly = canFly;

    }

**Цикли for**

Ви дізнаєтесь, як концептуально працює цикл for of.

Для початку важливо знати, що цикл for of не може працювати безпосередньо з об’єктом, оскільки  **об’єкт не можна повторювати** . Наприклад:

const car = {

    speed: 100,

    color: "blue"

}

for(prop of car) {

    console.log(prop)

}

Запуск наведеного вище фрагмента коду призведе до такої помилки:

Uncaught TypeError: car is not iterable

На відміну від об’єктів, масиви  *можна*  ітерувати. Наприклад:

const colors = ['red','orange','yellow']

for (var color of colors) {

    console.log(color);

}

Цього разу вихід виглядає наступним чином:

червоний

помаранчевий

жовтий

На щастя, ви можете використовувати той факт, що цикл for of можна запускати на масивах,  *щоб перебирати об’єкти* .

Але як?

Перш ніж ви зможете правильно відповісти на це запитання, вам спочатку потрібно переглянути три вбудовані методи:  **Object.keys()** ,  **Object.values()** і  **Object.entries()** .

**Вбудовані методи**

**Метод Object.keys  ().**

Метод  **Object.keys()**  отримує об’єкт як свій параметр. Пам’ятайте, що цей об’єкт є  **тим об’єктом, який ви хочете виконати в циклі** . Ще занадто рано пояснювати, як ви будете перебирати сам об'єкт; поки що зосередьтеся на поверненому масиві властивостей під час виклику  методу **Object.keys()**  .

Ось приклад запуску  методу **Object.keys()**  на абсолютно новому  об’єкті **car2**  :

const car2 = {

    speed: 200,

    color: "red"

}

console.log(Object.keys(car2)); // ['speed','color']

Отже, коли я запускаю  **Object.keys()**  і передаю йому мій   об’єкт  **car2 , повернуте значення є масивом рядків** , де кожен рядок є ключем властивостей, що містяться в моєму  об’єкті **car2**  .

**Метод Object.values ().**

Іншим корисним методом є  **Object.values()** :

const car3 = {

    speed: 300,

    color: "yellow"

}

console.log(Object.values(car3)); // [300, 'yellow']

**Метод Object.entries ().**

Нарешті, є ще один корисний метод,  **Object.entries()** , який повертає масив зі списком як ключів, так і значень.

const car4 = {

    speed: 400,

    color: 'magenta'

}

console.log(Object.entries(car4));

Під час виклику  методу **Object.entries()** повертається  наступне:

[ ['speed', 400], ['color', 'magenta'] ]

Цього разу значення, які повертаються, є масивами з 2 елементів, вкладеними в масив. Іншими словами, ви отримуєте масив масивів, де кожен елемент масиву має два члени, перший з яких є ключем властивості, а другий — значенням властивості.

По суті, це схоже на те, що ви перераховуєте всі властивості певного об’єкта, приблизно так:

[

    [propertyKey, propertyVal],

    [propertyKey, propertyVal],

    ...і т.д

]

Підводячи підсумок, ви дізналися, що ви можете перебирати масиви за допомогою  циклу **for of**  . Ви також дізналися, що ви можете витягувати ключі об’єкта, значення або те й інше за допомогою  синтаксису **Object.keys()** ,  **Object.values()** і  **Object.entries()**  .

**Приклади**

Тепер у вас є всі інгредієнти, необхідні для  **циклу по ключах і значенням властивостей будь-якого об’єкта** .

Ось дуже простий приклад того, як це зробити:

var clothingItem = {

    price: 50,

    color: 'beige',

    material: 'cotton',

    season: 'autumn'

}

for( const key of Object.keys(clothingItem) ) {

    console.log(key, ":", clothingItem[key])

}

Найважча частина для розуміння в цьому синтаксисі, ймовірно,  **clothingItem[key]** .

На щастя, це не надто важко зрозуміти, особливо тому, що ви вже розглянули цю концепцію раніше, коли вивчали,  **як отримати доступ до члена об’єкта за допомогою позначення в дужках** .

Згадайте, що ви також дізналися, як можна динамічно отримувати доступ до імені властивості.

Щоб повернутися до цієї концепції та показати практичну демонстрацію того, як це працює, давайте закодуємо оголошення функції, яка випадковим чином призначає або  **швидкість рядка, або колір**  рядка   імені змінної, а потім створимо об’єкт, який має лише два ключі:  ключ **швидкості**  та кольоровий  **ключ**  .

Після цього налаштування ви зможете динамічно отримувати доступ до однієї з цих властивостей абсолютно нового  об’єкта **дрона**  , використовуючи нотацію в дужках.

Ось код прикладу:

function testBracketsDynamicAccess() {

  var dynamicKey;

  if(Math.random() > 0.5) {

    dynamicKey = "speed";

   }else{

     dynamicKey = "color";

   }

    var drone = {

      speed: 15,

      color: "orange"

    }

    console.log(drone[dynamicKey]);

}

testBracketsDynamicAccess();

Цей приклад може здатися дещо заплутаним, але його мета полягає в тому, щоб продемонструвати той факт, що ви отримуєте одне чи інше значення з ключа об’єкта на основі рядка, який було призначено змінній dynamicKey  **і**  доступ до якого без проблем здійснюється за допомогою позначення в дужках.

Не соромтеся запустити  функцію **testBracketsDynamicAccess()**  кілька разів, і ви помітите, що іноді виводиться значення  **15** , а іноді воно  **помаранчеве** , хоча я завжди використовую  ключ **drone[dynamicKey]**  . Оскільки значення **dynamicKey**  заповнюється під час  виклику **Math.random()**  , іноді цей вираз обчислюється як  **drone["speed"]** , а в інших випадках цей вираз обчислюється як  **drone["color"]** .

**Приклади літералів шаблонів**

**Що таке літерали шаблону?**

Літерали шаблону — це альтернативний спосіб роботи з рядками, який був представлений у доповненні ES6 до мови JavaScript.

До ES6 єдиним способом створення рядків у JavaScript було розділення їх одинарними або подвійними лапками:

«Привіт, світ!»

"Привіт, світ!"

Поряд із попередніми способами побудови рядків, ES6 представив використання зворотних символів як розділювачів:

`Hello, World!`

Наведений вище фрагмент коду є прикладом рядка шаблону, який також відомий як літерал шаблону.

*Примітка. На більшості клавіатур знак зворотної позначки може розташовуватися над клавішею TAB, ліворуч від клавіші з цифрою 1.*

За допомогою літералів шаблону вираз можна вставити в  *заповнювач* . Покажчик місця заповнення представлено ${}, де все, що знаходиться у фігурних дужках, розглядається як JavaScript, а все, що знаходиться поза дужками, розглядається як рядок:

**Відмінності шаблону від звичайного рядка**

Існує кілька способів, якими рядок шаблону відрізняється від звичайного рядка.

* По-перше, це дозволяє  **змінну інтерполяцію** :

let greet = "Hello";

let place = "World";

console.log(`${greet} ${place} !`) //display both variables using template literals

Наведений вище журнал консолі виведе:

Hello World !

По суті, використання шаблонних літералів дозволяє програмістам вставляти змінні безпосередньо між зворотними галочками, без необхідності використовувати  оператор **+**  і одинарні або подвійні лапки для відокремлення рядкових літералів від змінних. Іншими словами, у ES5 наведений вище приклад потрібно було б записати так:

var greet = "Hello";

var place = "World";

console.log(greet + " " + place + "!"); //display both variables without using template literals

* Окрім інтерполяції змінних, рядки шаблону можуть охоплювати декілька рядків.

Наприклад, це ідеальний синтаксис:

`Hello,

World

!

`

Зверніть увагу, що це не можна зробити за допомогою  **рядкових літералів**  (тобто рядків, розділених одинарними або подвійними лапками):

"Hello,

World"

Наведений вище код під час виконання видає синтаксичну помилку.

Простіше кажучи, шаблонні літерали дозволяють багаторядкові рядки - те, що просто неможливо з рядковими літералами.

* Крім того, причина, чому можна інтерполювати змінні в літералах шаблону, полягає в тому, що цей синтаксис фактично дозволяє  **оцінювати вираз** .

Іншими словами, це:

//it's possible to perform arithmetic operation inside a template literal expression

console.log(`${1 + 1 + 1 + 1 + 1} stars!`)

У наведеному вище прикладі консольно запише такий рядок:  **5 зірок!** .

Це відкриває безліч можливостей. Наприклад, можна обчислити потрійний вираз у літералі шаблону.

Деякі додаткові варіанти використання літералів шаблону – це  **вкладені літерали шаблону**  та  **шаблони з тегами** . Однак вони дещо більш залучені і виходять за рамки цього читання.

**Приклади структур даних**

Основна увага буде зосереджена на роботі зі структурами даних Object, Array, Map і Set у JavaScript через низку прикладів.

**Робота з масивами в JavaScript**

Раніше ви розглянули багато концепцій, пов’язаних із роботою з масивами JavaScript.

Однак є ще кілька важливих тем, які можна розглянути, і одна з них, наприклад, робота з деякими вбудованими методами.

У цьому читанні фокус зосереджений на трьох конкретних методах, які існують для масивів:

1. **forEach**
2. **фільтр**
3. **карта**

Давайте розглянемо ці методи.

**Метод forEach ().**

Масиви в JavaScript оснащені зручним методом, який дозволяє циклічно переглядати кожного з їхніх членів.

Ось основний синтаксис:

const fruits = ['kiwi','mango','apple','pear'];

function appendIndex(fruit, index) {

    console.log(`${index}. ${fruit}`)

}

fruits.forEach(appendIndex);

Результат виконання наведеного вище коду такий:

0. kiwi

1. mango

2. apple

3. pear

Щоб пояснити синтаксис,  метод **forEach()**  приймає  **функцію, яка працюватиме з кожним елементом масиву** . Першим параметром цієї функції є сам поточний елемент масиву, а другим (необов’язковим) параметром є індекс.

Дуже часто функція, яку  має використовувати метод **forEach(),**  передається безпосередньо у виклик методу, ось так:

const veggies = ['onion', 'garlic', 'potato'];

veggies.forEach( function(veggie, index) {

    console.log(`${index}. ${veggie}`);

});

Це робить код більш компактним, але, можливо, дещо важчим для читання. Для покращення читабельності іноді використовуються функції стрілок. Ви можете дізнатися більше про функції стрілок у додатковому читанні.

**Метод filter().**

Ще один дуже корисний метод для масиву — це  метод **filter()**  . Він фільтрує ваші масиви  **на основі певного тесту** . Повертаються ті елементи масиву, які пройшли перевірку.

Ось приклад:

const nums = [0,10,20,30,40,50];

nums.filter( function(num) {

    return num > 20;

})

Ось повернуте значення масиву:

[30,40,50]

Подібно до  методу **forEach()** ,  метод **filter()**  також приймає функцію, і ця функція виконує певну роботу над кожним із елементів масиву.

**Метод карти​**

Нарешті, є дуже корисний  метод **карти**  .

Цей метод використовується для зіставлення кожного елемента масиву з елементом іншого масиву на основі будь-якої роботи, виконаної всередині функції, яка передається на карту як параметр.

Наприклад:

[0,10,20,30,40,50].map( function(num) {

    return num / 10

})

Повернене значення з коду вище:

[0,1,2,3,4,5]

Як уже обговорювалося, вибір належної структури даних впливає на сам код, який ви можете написати. Це пояснюється тим, що сама структура даних має деякі вбудовані функції, які спрощують виконання певних завдань або ускладнюють або навіть унеможливлюють без перетворення коду на правильну структуру даних.

Тепер, коли ви розглянули методи, давайте дослідимо, як працювати з різними вбудованими структурами даних у JavaScript.

**Робота з об'єктами в JavaScript**

Багато інформації про те, як працювати з об’єктами в JavaScript, вже розглянуто в цьому курсі.

Наведений нижче приклад демонструє, як використовувати структуру даних об’єкта для виконання конкретного завдання. Це завдання полягає в тому, щоб перетворити об’єкт на масив:

const result = [];

const drone = {

    speed: 100,

    color: 'yellow'

}

const droneKeys = Object.keys(drone);

droneKeys.forEach( function(key) {

    result.push(ключ, drone[ключ])

})

console.log(результат)

Це результат виконання наведеного вище коду:

['speed',100,'color','yellow']

Хоча це можливо і працює, необхідність зробити щось подібне може означати, що ви не вибрали правильну структуру даних для роботи у своєму коді.

З іншого боку, іноді вам не вдається вибрати структуру даних, з якою ви працюєте. Можливо, ці дані надходять від стороннього постачальника даних, і все, що ви можете зробити, це закодувати свою програму так, щоб вона споживала їх. Ви дізнаєтеся більше про обмін даними в Інтернеті, коли дізнаєтесь про JSON (нотація об’єктів JavaScript).

**Робота з картами в JavaScript**

Щоб створити нову карту, ви можете скористатися  конструктором **карти**  :

new Map();

Карта може бути дуже схожою на об’єкт у JS.

Однак він не має спадковості. Жодних прототипів! Це робить його корисним як сховище даних.

Наприклад:

let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.set(2, "The Runner-up");

bestBoxers.set(3, "The third place");

console.log(bestBoxers);

Ось результат консолі:

Map(3) {1 => 'The Champion', 2 => 'The Runner-up', 3 => 'The third place'}

Щоб отримати конкретне значення, потрібно використовувати  метод **get()**  . Наприклад:

bestBoxers.get(1); // 'The Champion'

**Робота з наборами в JavaScript**

Набір — це набір унікальних значень.

Щоб створити новий набір, ви можете скористатися  конструктором **Set**  :

new Set();

Конструктор  **Set**  може, наприклад, приймати масив.

Це означає, що ми можемо використовувати його для швидкого фільтрування масиву за унікальними членами.

const repetitiveFruits = ['apple','pear','apple','pear','plum', 'apple'];

const uniqueFruits = new Set(repetitiveFruits);

console.log(uniqueFruits);

Наведений вище код виводить на консоль наступне:

{'apple', 'pear', 'plum'}

**Інші структури даних у JavaScript**

Окрім вбудованих структур даних у JavaScript, можна створювати ненативні, власні структури даних.

Ці структури даних є вбудованими в деякі інші мови програмування або навіть ці інші мови програмування не підтримують їх нативно.

Деякі більш просунуті структури даних, які не були розглянуті, включають:

* Черги
* Зв’язані списки (однозв’язні та двозв’язані)
* дерева
* графіки

**Використання Spread і Rest**

У цій статті ви дізнаєтесь, як об’єднувати масиви та об’єкти за допомогою оператора rest. Ви також дізнаєтесь, як використовувати оператор spread, щоб:

* Додайте нових учасників до масивів без використання методу **push()** ,
* Перетворення рядка в масив і
* Скопіюйте об’єкт або масив в окремий об’єкт

**Згадайте, що методи push() і pop() використовуються для додавання та видалення елементів із кінця масиву.**

**Об’єднання масивів, об’єктів за допомогою оператора rest**

Використовуючи оператор spread, можна легко об’єднати масиви:

const fruits = ['apple', 'pear', 'plum']

const berries = ['blueberry', 'strawberry']

const fruitsAndBerries = [...fruits, ...berries] // concatenate

console.log(fruitsAndBerries); // outputs a single array

Ось результат:

['apple', 'pear', 'plum', 'blueberry', 'strawberry']

Також легко об’єднувати об’єкти:

const flying = { wings: 2 }

const car = { wheels: 4 }

const flyingCar = {...flying, ...car}

console.log(flyingCar) // {wings: 2, wheels: 4}

**Додайте нових членів до масивів без використання методу push().**

Ось як використовувати оператор поширення, щоб легко додати одного або кількох членів до існуючого масиву:

let veggies = ['onion', 'parsley'];

veggies = [...veggies, 'carrot', 'beetroot'];

console.log(овочі);

Ось результат:

['onion', 'parsley', 'carrot', 'beetroot']

**Перетворення рядка в масив за допомогою оператора поширення Spread**

Отримавши рядок, його легко розкласти на окремі елементи масиву:

const greeting = "Hello";

const arrayOfChars = [...greeting];

console.log(arrayOfChars); //  ['H', 'e', 'l', 'l', 'o']

**Копіювання об'єкта за допомогою оператора spread**

Ось як скопіювати об’єкт в повністю окремий об’єкт за допомогою оператора spread.

const car1 = {

    speed: 200,

    color: 'yellow'

}

const car 2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)

Вихід  **201, 200** .

Ви можете скопіювати масив в повністю окремий масив, також використовуючи оператор поширення, ось так:

const fruits1 = ['apples', 'pears']

const fruits2 = [...fruits1]

fruits1.pop()

console.log(fruits1, "not", fruits2)

Цього разу результат:

['apples'] 'not' ['apples','pears']

Зауважте, що оператор розповсюдження виконує лише поверхове копіювання вихідного масиву або об’єкта. Для отримання додаткової інформації про це зверніться до додаткової літератури.

Є багато інших трюків, які можна виконати за допомогою оператора поширення. Деякі з них дуже зручні, коли ви починаєте працювати з такою бібліотекою, як React.