Java Script

NEXT

План

конструктори

внутрішні методи

прототипи

Якість коду

GENIUS space

Конструктори

Звичайний синтаксис літерал об'єкта {} дозволяє створити один об'єкт. Але найчастіше потрібно створити багато однотипних об'єктів, динамічно, під час виконання програми. Для цього використовують функції конструктори, викликаючи їх за допомогою спеціального оператора new.

Конструктор це звичайна функція до якої застосували оператор new. Це призводить до створення нового об'єкта та виклику функції у контексті цього об'єкта.

Функція-конструктор
Технічно, функції-конструктори – це звичайні функції. Однак є дві загальні домовленості:

1. Ім’я функції-конструктора повинно починатися з великої літери.
2. Функції-конструктори повинні виконуватися лише з оператором "new".

GENIUS space

Наприклад:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Джек");

alert(user.name); // Джек
alert(user.isAdmin); // false
```

Коли функція виконується з new, відбуваються наступні кроки:

Створюється новий порожній об’єкт, якому присвоюється this.
Виконується тіло функції. Зазвичай воно модифікує this, додає до нього нові властивості.
Повертається значення this.

GENIUS space

Будь-яка функція, крім стрілочної, може бути використана як конструктор, тобто викликана оператором new. При виклику стрілки через New буде помилка.

Щоб відрізнити конструктор від звичайної функції, конструктори прийнято називати з великої літери, а саму назву відображати тип створюваного об'єкта. Ми вже працювали з конструкторами, вбудованими в мову: Object, Array, Number, String та інші.

Використання конструкторів зручно при створенні безлічі об'єктів з одним набором властивостей, що мають різні значення. Тому таку функцію і називають конструктором - вона призначена для конструювання об'єктів за заздалегідь підготовленим шаблоном.

GENIUS space

Внутрішні методи [[Call]] та [[Construct]]

Функції викликаються використовуючи два різні внутрішні методи: [[Call]] і [[Construct]].

Коли функція викликається без new, виконується метод [[Call]], який виконує тіло функції так, як описано в коді.

Коли функція викликається з new, виконується метод [[Construct]], який відповідає за створення нового об'єкта і виконання тіла функції з цим, що посилається на цей об'єкт.

Не всі функції мають внутрішній метод [[Construct]], і тому всі функції можуть бути викликані через new. Стрілецькі функції не мають методу [[Construct]] і, тому, не можуть бути використані як конструктори.

GENIUS space

Створення методів у конструкторі
Використання конструкторів для створення об’єктів дає велику гнучкість. Конструктор може мати параметри, які визначають, як побудувати об’єкт і що в нього помістити.

Звичайно, ми можемо додати до this не лише властивості, але й методи.

У наведеному нижче прикладі,
new User(name) створює об’єкт із заданим name та методом sayHi:

```
function User(name) {
this.name = name;
this.sayHi = function() {alert( "Моє ім’я: " + this.name );
};
}

let john = new User("Джон"); john.sayHi(); // Моє ім’я: Джон
```

GENIUS space

Object.prototype - прототип об'єкта Object.

prototype властивість об'єкта Object яка повертає прототип об'єкта.

Прототип - це звичайний об'єкт, який ділиться своєю поведінкою з іншими об'єктами - тобто стає батьком об'єкта. Прототип використовується в основному для наслідування.

Майже всі об'єкти в JavaScript є екземплярами Object і успадковують властивості, методи від Object.prototype. Хоча ці властивості можуть бути перевизначені.

Зміни в Object.prototype об'єкта розглядаються на всіх об'єктах через прототип ланцюжка, якщо властивості і методи не будуть перезаписані далі по ланцюжку прототипів.

Object.prototype знаходиться на вершині ланцюжка прототипів.

GENIUS space

Всі об'єкти в JavaScript є нащадками Object; всі об'єкти успадковують методи і властивості з прототипу об'єкта Object.prototype, хоча вони і можуть бути перевизначені. Наприклад, прототипи інших конструкторів скасовують властивість constructor і надають свої власні методи toString().

- Властивості:
- Object.constructor - Визначає функцію, яка створює прототип об'єкта.
- Object.__ proto__ - Вказує на об'єкт, який використовувався в якості прототипу при інстанціруванні об'єкта.
- Object.__noSuchMethod__ - дозволяє визначити функцію, виконуються при виклику в якості методу не певного члена об'єкта.
- Object.__ count__ - використовувалося для повернення кількості перерахованих властивостей, певних безпосередньо на призначеному для користувача об'єкті, але було видалено.
- Object.__ parent__ -використовувалося для вказівки контексту об'єкта, але було видалено.

GENIUS space

Методи:

Object.__defineGetter__() - асоціює функцію з властивістю, яка, при доступі до нього, виконує цю функцію і повертає її значення, що повертається.

Object.__defineSetter__() - асоціює функцію з властивістю, яке, при його установці, виконує цю функцію, змінює властивість.

Object.__ lookupGetter__() - повертає функцію, пов'язану із зазначеним властивістю методом __defineGetter__.

Object.__lookupSetter__() - Повертає функцію, пов'язану із зазначеним властивістю методом __defineSetter__.

Object.hasOwnProperty() - чи містить вказане властивість безпосередньо об'єкт, або він успадкував його по ланцюжку прототипів.

GENIUS space

Object.isPrototypeOf() - чи перебуває зазначений об'єкт в ланцюжку прототипів об'єкта, на якому був викликаний даний метод.

Object.propertyIsEnumerable() - повертає логічне значення, яке вказує, чи встановлений внутрішній атрибут ECMAScript DontEnum.

Object.toSource() - повертає рядок, що містить вихідний код об'єкта.

Object.toLocaleString() - повертає рядкове представлення об'єкту згідно локалі.

Object.toString() - повертає строкове представлення об'єкту.

Object.unwatch() - видаляє точку спостереження (watchpoint) зі властивості об'єкта.

Object.valueOf() - повертає значення примітиву зазначеного об'єкта.

Object.watch() - додає точку спостереження (watchpoint) до властивості об'єкта.

GENIUS space

Коротка історія
Якщо порахувати всі способи керування властивістю [[Prototype]], їх буде багато!
Багато способів робити одне й те ж саме!

Чому?
Так склалося історично.
Властивість "prototype" функції-конструктора реалізована з самих давніх часів.
Пізніше, в 2012 році, метод Object.create став стандартом. Це дало можливість створювати об’єкти з певним прототипом, проте не дозволяло отримувати або встановлювати його. Тому браузери реалізували не стандартний аксесор __proto__, що дозволяв користувачу отримувати та встановлювати прототип в будь-який час.
Ще пізніше, в 2015 році, методи Object.setPrototypeOf та Object.getPrototypeOf були додані до стандарту, для того, щоб виконувати аналогічну функціональність як і __proto__. Оскільки __proto__ було широко реалізовано, воно згадується в Додатку B стандарту як не обов’язкове для не-браузерних середовищ, але вважається свого роду застарілим.
Таким чином зараз ми маємо всі ці способи для роботи з прототипом.

GENIUS space

Методи прототипів, об’єкти без __proto__
Властивість __proto__ вважається застарілою (підтримується браузером відповідно до стандарту).

Сучасні методи:

Object.create(proto, [descriptors]) – створює пустий об’єкт з властивістю [[Prototype]], що посилається на переданий об’єкт proto, та необов’язковими для передачі дескрипторами властивостей descriptors.

Object.getPrototypeOf(obj) – повертає значення [[Prototype]] об’єкту obj.

Object.setPrototypeOf(obj, proto) – встановлює значення [[Prototype]] об’єкту obj рівне proto.

Ці методи необхідно використовувати на відміну від __proto__.

GENIUS space

"Прості" об’єкти

Як ми знаємо, об’єкти можуть використовуватися як асоціативні масиви для зберігання пар ключ/значення.

...Проте якщо ми спробуємо зберегти створені користувачем ключі в ньому (наприклад, словник з користувацьким вводом), ми можемо спостерігати цікавий збій: усі ключі працюють правильно окрім "__proto__".

Розгляньте приклад:

```
let obj = {};

let key = prompt("Введіть ключ", "__proto__");
obj[key] = "певне значення";

alert(obj[key]); // [object Object], не "певне значення"!
```

GENIUS space

В цьому випадку, якщо користувач введе __proto__, присвоєння проігнорується!

Це не повинно дивувати нас. Властивість __proto__ є особливою: вона має бути або об’єктом або null. Рядок не може стати прототипом.
Проте ми не намагалися реалізувати таку поведінку. Ми хотіли зберегти пари ключ/значення і при цьому ключ з назвою "__proto__" не зберігся. Тому це помилка!

В цьому прикладі наслідки не такі жахливі. Однак в інших випадках ми можемо встановлювати об’єктні значення і тоді прототип може дійсно змінитись. В результаті, виконання коду піде неправильним та неочікуваним шляхом.
Найгірше в цьому – зазвичай розробники не задумаються над тим, що така ситуація взагалі можлива. Це робить подібні помилки важкими для виявлення і перетворюються їх у вразливості, особливо коли JavaScript використовується на стороні серверу.

Неочікувані речі можуть траплятися при встановлені значення для властивості toString, яка за замовчуванням є функцією, та для інших вбудованих методів.

GENIUS space

Як ми можемо уникнути цієї проблеми?

В першу чергу, для зберігання ми можемо просто використовувати Map замість простих об’єктів, тоді все працює правильно.

Проте Object може також послужити нам, оскільки творці мови задумувались над цією проблемою вже дуже давно.

__proto__ не є властивістю об’єкту, але є аксесором Object.prototype

Таким чином, якщо obj.__proto__ зчитується або встановлюється, відповідний гетер/сетер викликається з прототипу та отримує/встановлює [[Prototype]].

Як було сказано на початку цього розділу: __proto__ це спосіб доступу до [[Prototype]], але не є самим [[Prototype]].

GENIUS space

Тепер, коли ми хочемо використати об’єкт як асоціативний масив та уникнути таких проблем, ми можемо зробити це за допомогою невеликої хитрості:

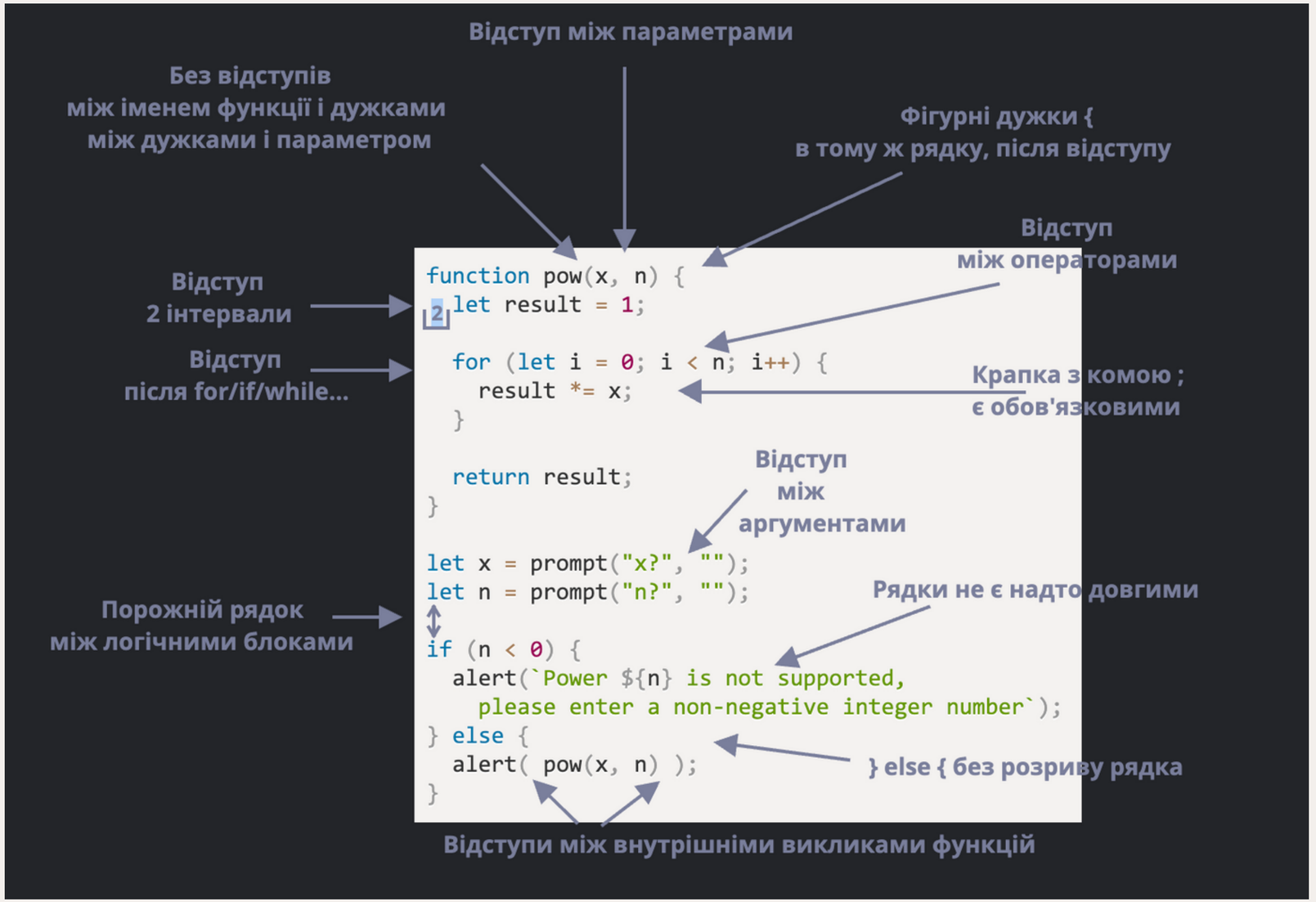
```
let obj = Object.create(null);

let key = prompt("Введіть ключ", "__proto__");
obj[key] = "певне значення";

alert(obj[key]); // "певне значення"
```

Object.create(null) створює пустий об’єкт без прототипу ([[Prototype]] дорівнює null):

Таким чином, відсутні наслідувані гетер/сетер для __proto__. Тепер ми працюємо зі звичайною властивістю, тому приклад вище працює правильно.
Ми можемо називати такі об’єкти “простими” або “чистим словниковим” об’єктом, тому що вони навіть простіше ніж звичайні об’єкти {...}.

GENIUS space

Якість коду


Відступ між параметрами
Без відступів
між іменем функції і дужками
між дужками і параметром
Фігурні дужки {
в тому ж рядку, після відступу
Відступ
між операторами
Відступ
2 інтервали
Відступ
після for/if/while...
Крапка з комою ;
є обов'язковими
Відступ
між
аргументами
Порожній рядок
між логічними блоками
Рядки не є надто довгими
} else { без розриву рядка
Відступи між внутрішніми викликами функцій
function pow(x, n) {
let result = 1;
for (let i = 0; i < n; i++) {
result *= x;
}
return result;
}
let x = prompt("x?", "");
let n = prompt("n?", "");
if (n < 0) {
alert(`Power ${n} is not supported,
please enter a non-negative integer number`);
} else {
alert(pow(x, n));
}

- Використовуйте зрозумілі імена змінних та функцій Код набагато легше читати, коли його написанні використовують зрозумілі, описові імена функцій і змінних.
- Пишіть короткі функції, які вирішують одне завдання Функції легше підтримувати, вони стають набагато зрозумілішими, читальнішими, якщо вони спрямовані на вирішення лише одного завдання.
- Пам'ятайте про принцип DRY Одна з перших ідей, яку намагаються донести до того, хто хоче стати програмістом, звучить так: не повторюйся (Don't Repeat Yourself, DRY). Якщо ви помічаєте у своїх проектах фрагменти, що повторюються — використовуйте такі програмні конструкції, які дозволять скоротити повтори одного і того ж коду.
- Використовуйте ідеї інкапсуляції та модульності Групуйте пов'язані змінні та функції для того, щоб зробити ваш код зрозумілішим та покращити його з точки зору його повторного використання.
- Документуйте код Пишіть хорошу документацію до свого коду - тоді той, хто зіткнеться з ним у майбутньому, зрозуміє, що і чому в цьому коді робиться.

GENIUS space

Подумайте про використання правил Сенді Метц
Сенді Метц чудово програмує на Ruby, робить цікаві доповіді та пише книги. Вона сформулювала чотири правила написання чистого коду об'єктно-орієнтованих мовами. Ось вони.

Класи не повинні бути довшими за 100 рядків коду.

Методи та функції не повинні бути довшими за 5 рядків коду.

Методам слід передавати трохи більше 4 параметрів.

Контролери можуть ініціалізувати лише один об'єкт.

https://www.youtube.com/watch?v=npOGOmkxuio

GENIUS space

Додаткові матеріали

- https://uk.javascript.info/constructor-new
- http://xn--80adth0aefm3i.xn--j1amh/object.prototype
- https://uk.javascript.info/prototype-methods
- https://uk.javascript.info/comments
- https://uk.javascript.info/coding-style
- http://xn--80adth0aefm3i.xn--j1amh/%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80
- https://yehudakatz.com/2011/08/12/understanding-prototypes-in-javascript/
- https://medium.com/@happymishra66/inheritance-in-javascript-21d2b82ffa6f
- https://google.github.io/styleguide/jsguide.html
- https://github.com/airbnb/javascript
- https://www.youtube.com/watch?v=npOGOmkxuio

GENIUS space

Домашнє завдання

----]----

Створіть функцію-конструктор Calculator, який створює об’єкти з трьома методами:

read() запитує два значення за допомогою prompt і запам’ятовує їх у властивостях об’єкта.
sum() повертає суму цих властивостей.
mul() повертає результат множення даних властивостей.

GENIUS space