Lesson 12

План заняття

- ФП;
- ООП;
- Принципи ООП;
- Принцип успадкування через прототипи;
- Функії конструктори;
- Відмінність Prototype від __proto__;
- Присвоєння об'єкту прототипу вже створеному об'єкту;

Функціональне програмування

Функціональне програмування — це парадигма програмування, у якій процес обчислення сприймається як обчислення значень функцій у математичному розумінні останніх. Функціональне програмування передбачає обходитися обчисленням результатів функцій від вихідних даних і результатів інших функцій, і **не передбачає явного зберігання стану програми.**

Чисті функції

Чисті функії - відповідають таким вимогам:

- Фунція завжди повертає, при передачі їй одних і тих самих аргументів, той самий результат (такі функції також називають детермінованими).
- Така функція не має побічних ефектів (випадкові числа, HTTP-запити, надсилання листів, взаємодія з базою даних, взаємодія з глобальними змінними, читання та запис файлів, оперування поточним часом).

Серед сильних сторін чистих функцій можна назвати те що, що код, написаний із їх використанням, легше тестувати. Зокрема, не потрібно створювати якихось об'єктів-заглушок. Це дозволяє виконувати модульне тестування чистих функцій у різних контекстах:

- Якщо функцію передається параметр A — очікується повернення значення B.
- Якщо функцію передається параметр C — очікується повернення значення D.

Імутабельність

Імутабельність якоїсь сутності можна описати як те, що з часом вона не змінюється, або як неможливість змін цієї сутності. Якщо іммутабельний об'єкт намагаються змінити, зробити цього не вдасться. Натомість потрібно буде створити новий об'єкт, що містить нові значення.

Посилальна прозорість

Якщо функція незмінно повертає один і той же результат для одних і тих же переданих їй вхідних значень, вона має прозорість.

Чисті функції + іммутабельні дані = прозорість посилань

```
function square(n) {
  return n * n;
}
```

Функції як об'єкти першого класу

Ідея сприйняття функцій як об'єктів першого класу полягає в тому, що такі функції можна розглядати як значення та працювати з ними як із даними. При цьому можна виділити такі функції:

- Посилання на функції можна зберігати в константах та змінних і через них звертатися до функцій.
- Функції можна передавати іншим функціям як параметри.
- Можна повернути функції з інших функцій.

Функції вищого ладу

Говорячи про функції вищого порядку ми маємо на увазі функції, які характеризуються хоча б однією з таких особливостей:

- Функція приймає іншу функцію як аргумент (таких функцій може бути кілька).
- Функція повертає іншу функцію як результат своєї роботи.

Наприклод: filter(), map(), reduce() ...

ООП

Об'єктно-орієнтована ідеологія розроблялася як спроба пов'язати поведінку сутності з її даними та спроектувати об'єкти реального світу та бізнес-процесів у програмний код. Замислювалося, що такий код простіше читати і розуміти людиною, тому що людям властиво сприймати навколишній світ як безліч об'єктів, що взаємодіють між собою, що піддаються певній класифікації.

Кожен об'єкт має три складові: ідентичність (identity), стан (state) і поведінку (behaviour).

- Ідентичність (identity) - Це ще один термін рівності або однаковості значень. Коли ми говоримо “a і b ідентичні”, ми маємо на увазі “a і b вказують на одне і те ж значення”;
- Стан об'єкта - це набір всіх його полів та їх значень;
- Поведінка об'єкта – це набір всіх методів класу об'єкта;

Принципи ООП

- Абстрація;
- Інкапсуляція;
- Успадкування;
- Поліморфізм;

Успадкування через прототипи

Якщо ми хочемо збудувати новий об’єкт поверх того, що існує нам у цьому допоможе успадкування через прототипи.

Об’єкти мають спеціальну приховану властивість [[Prototype]], яка може приймати значення: або null, або мати посилання на інший об’єкт. Цей об’єкт називається "прототип". Коли ми зчитуємо якусь властивість об’єкта object, але її не має, JavaScript автоматично бере її з прототипу. В програмуванні це називається “успадкування через прототипи”.

Така властивість [[Prototype]] є внутрішньою та прихованою, але є декілька шляхів щоб її визначити.

Одним з них є використання спеціального імені __proto__

Успадкування через прототипи

Існує два обмеження:

1. Посилання через прототипи не може бути замкнено в кільце. JavaScript видасть помилку, якщо ми визначемо __proto__ в ланцюжку прототипів і замкнем його в кільце.
2. Значення __proto__ може бути, або посиланням на об’єкт, або null. Інші типи значень – ігноруються.

__proto__ є старим і давнім getter/setter для [[Prototype]]

Запам’ятайте!

Властивість __proto__ не є тою самою властивістю як внутрішня властивість [[Prototype]]. Це є getter/setter для [[Prototype]]

Згідно зі специфікацією мови, __proto__ повинно тільки підтримуватись в браузерах. Проте насправді, усі середовища, включаючи серверні, підтримують __proto__, а тому використовувати його можна досить безпечно.

Успадкування через прототипи

Операція по запису/видалення не застосовується на прототипах!

Прототипи можна використовувати тільки для зчитування властивостей.

Властивості ‘Accessor’ є винятком, оскільки присвоєння обробляється функцією встановлення (через ‘setter’)

Пам’ятайте!
Незалежно від того, де метод визначений: в об’єкті чи його прототипі, ключове слово this завжди вказує на об’єкт перед крапкою.

Коли успадковані методи викликаються на стороні об’єктів, що їх успадкували, вони можуть змінювати тільки свої стани а не стан того великого об’єкту.

Успадкування через прототипи

Цикл for..in також може проходитись по успадкованим властивостям.

Якщо нам потрібно виключити отримання успадкованих значень, використовуйте метод obj.hasOwnProperty(key), який повертає true якщо obj має тільки власні (не успадковані) властивості.

Якщо поля об'єкту позначені (стоять під прапорцем) як такі, що не рахуються (enumerable:false) то цикл for..in їх не зчитає.

Методи для прототипу

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

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

Object.create(proto, [descriptors]) – створює порожній об’єкт із заданим proto як [[Prototype]] і необов’язковими дескрипторами властивостей.

Ми можемо використати Object.create, щоб клонувати об’єкт

Трохи історії

- Властивість prototype функції-конструктора працювала з дуже давніх часів. Це найстаріший спосіб створення об’єктів із заданим прототипом.
- 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, щоб надати розробникам більше гнучкості.
- 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 Annex B стандарту як не обов’язкове для не-браузерних середовищ, але вважається свого роду застарілим.
- Пізніше, у 2022 році, було офіційно дозволено використовувати __proto__ в об’єктних літералах {...} (винесено з Annex B), але не як геттер/сеттер obj.__proto__ (ця можливість все ще в Annex B).

Function prototype

Як відомо, ми можемо створювати нові об’єкти за допомогою функції-конструктора. Оператор new автоматично створює приховану властивість [[Prototype]] для новоствореного об’єкта.

Вираз Function.prototype = obj дослівно означає наступне: коли new Function створено, його властивість [[Prototype]] посилається на об’єкт obj.

const some = new Function()

"prototype" це звичайна властивість, а [[Prototype]] означає some успадковує властивості від свого прототипа obj.

Function .prototype використовується тільки у разі використання функції-конструктора new Function

Властивість Function .prototype використовується коли буде викликано new Function, і створює властивість [[Prototype]] для нового об’єкта.

Типове значення

Кожна функція має властивість "prototype" навіть якщо ми цю властивість самі не прописуємо. Тобто вона існує за замовчуванням, або ця властивість є типовою.

В свою чергу, така типова властивість "prototype" представляє собою об’єкт, який має єдину властивість з назвою constructor, що зворотньо посилається на назву самої функції-конструктора.

```
function Animal() {}
/* властивість створена за замовчуванням
Animal.prototype = { constructor: Animal};
*/
```

Отже, якщо ми нічого не робимо з властивістю constructor то вона є доступна для всіх об’єктів rabbits через [[Prototype]]

constructor

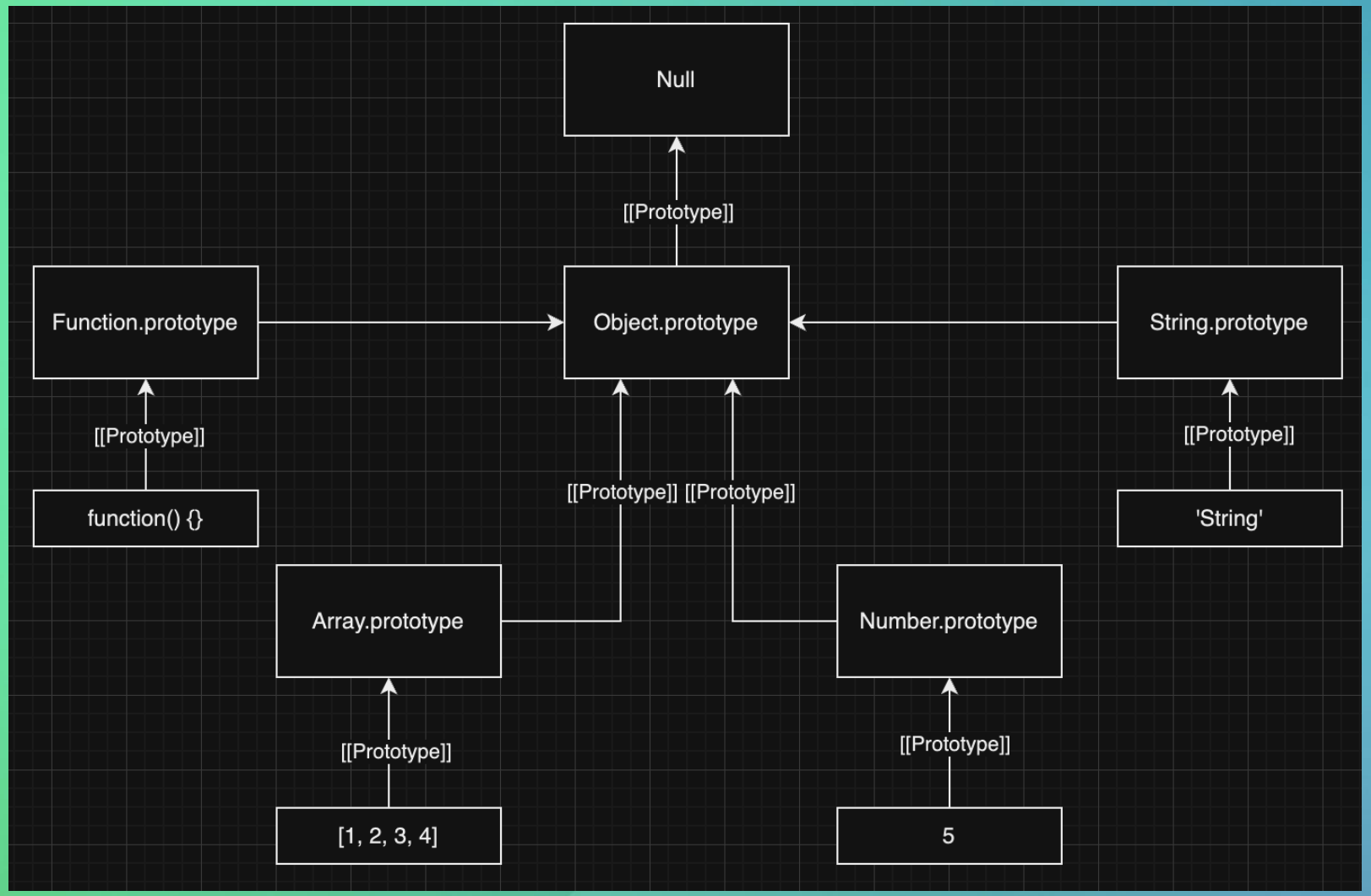
Ми можемо використовувати властивість constructor для створення нових об’єктів використовуючи той самий конструктор, який вже існує.

Це дуже практично у випадку наявності об’єкта, але не знаємо за допомогою якого саме конструктора той об’єкт був створений (для прикладу який був імпортований з якоїсь бібліотеки), а нам потрібно створити новий об’єкт по типу того, що вже існує.

JavaScript не забезпечує правильного значення "constructor". Воно існує за замовчуванням у властивостях "prototype" для функцій, але це все, що є. Те, що стається з "constructor" пізніше, цілковито залежить від нас самих.

Отже, щоб мати правильний "constructor" ми можемо чи додавати, чи видаляти властивості у дефолтному "prototype" замість того, щоб її цілковито заміняти

Вбудовані прототипи



Дякую за увагу