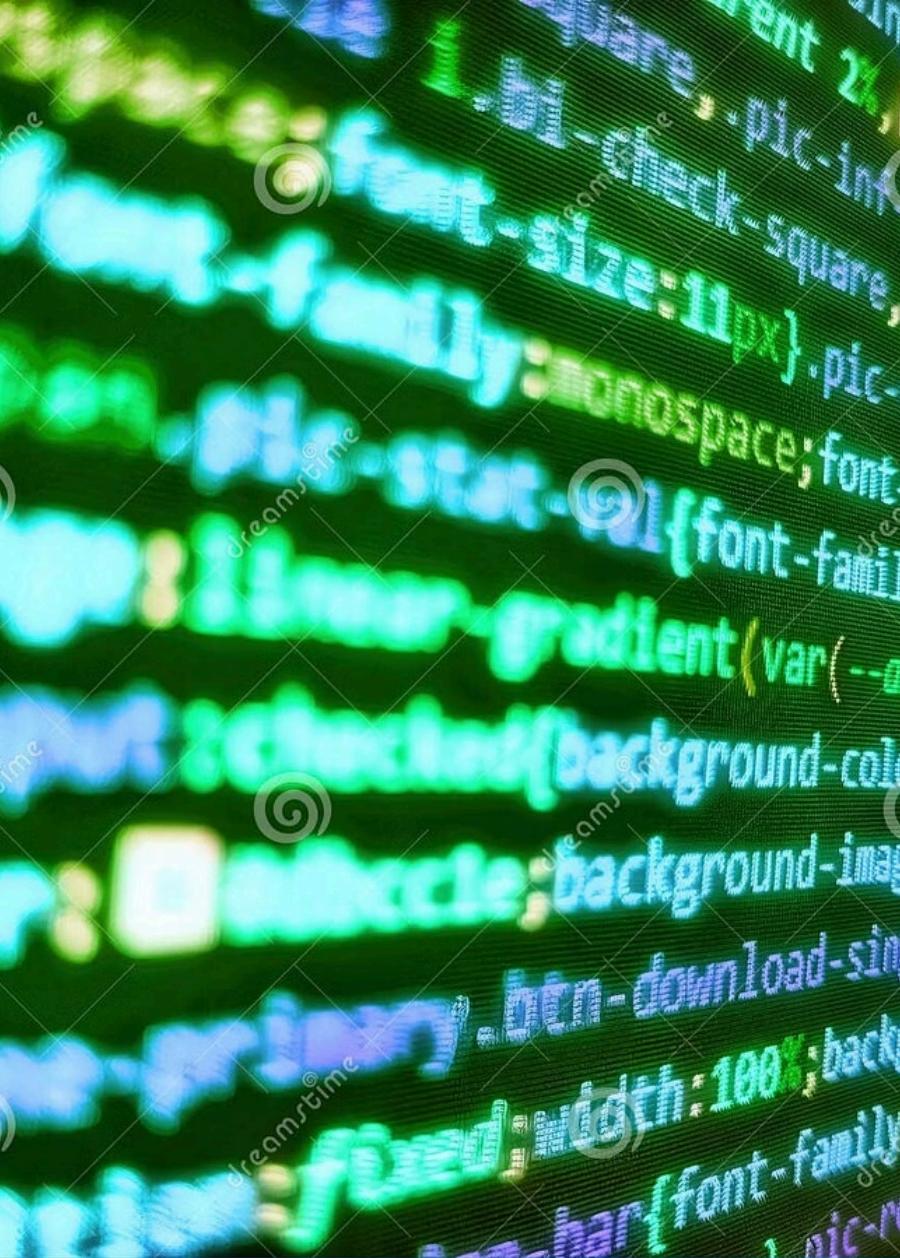


Реалізація принципів ООП в JavaScript

Глибокий огляд об'єктно-орієнтованого програмування та прототипного наслідування в JavaScript для розробників середнього рівня



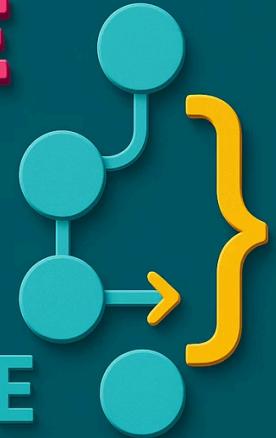
Наслідування через прототипи

Прототипна модель

Прототипне наслідування є ключовим аспектом ООП в JavaScript. На відміну від класичної моделі наслідування з інших мов, прототипи дозволяють об'єктам використовувати властивості та методи інших об'єктів через спеціальний механізм.

Кожен об'єкт в JavaScript має прототип (`__proto__`), який сам є об'єктом. Коли властивість чи метод не знаходяться у поточному об'єкті, JavaScript автоматично перевіряє прототип для їх пошуку.

PROTOTYPE CHAIN THE CORE OF JS INHERITANCE



```
// Створення об'єкта "прототипу"
const animalPrototype = {
  speak() {
    console.log(`${this.name} says ${this.sound}`);
  }
};

// Створення об'єкта з наслідуванням
const dog = {
  name: "Dog",
  sound: "Woof"
};
dog.__proto__ = animalPrototype;

dog.speak(); // Dog says Woof
```

Як це працює під капотом

01

Перевірка об'єкта

JavaScript спершу перевіряє, чи є властивість безпосередньо у самому об'єкті

02

Пошук у прототипі

Якщо не знайдено, JavaScript переходить до перевірки прототипу об'єкта

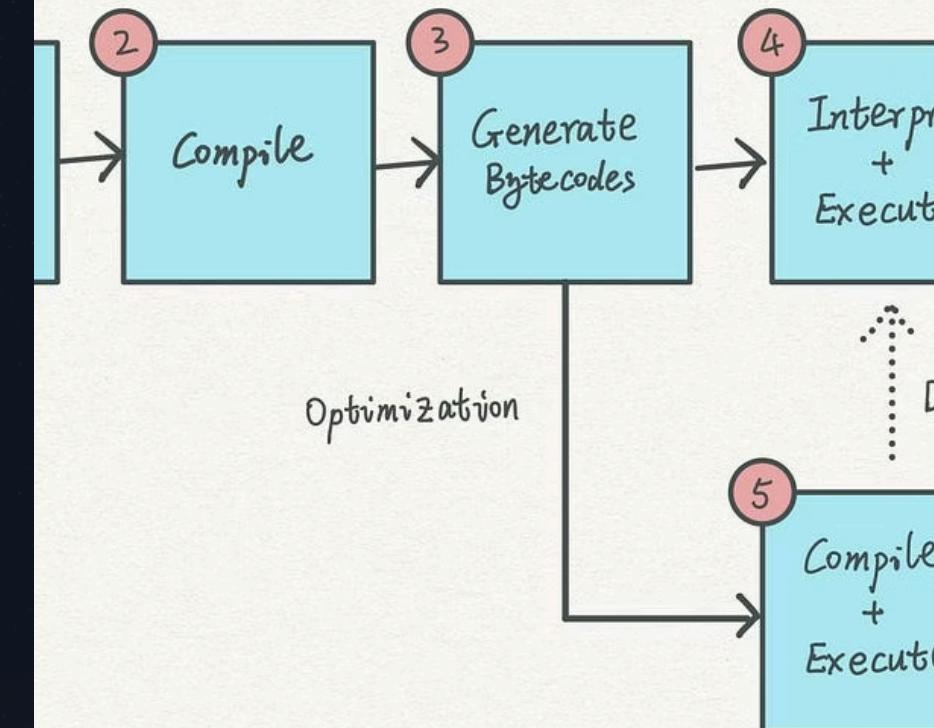
03

Ланцюжок прототипів

Процес продовжується вгору по ланцюжку, доки не буде знайдено властивість або досягнуто Object.prototype

У прикладі з `dog.speak()`, метод не знаходиться в об'єкті dog, тому JavaScript переходить до прототипу animalPrototype і знаходить там метод speak. Таким чином відбувається успадкування методу від прототипу.

ENGINE STEPS



Класи в ES6

ES6 вводить новий синтаксис для оголошення класів, але насправді вони використовують ту ж саму прототипну модель наслідування. Це синтаксичний цукор над існуючою системою прототипів.

Ключове слово `extends` використовується для наслідування, а `super` викликає конструктор батьківського класу.

```
class Animal {  
  constructor(name, sound) {  
    this.name = name;  
    this.sound = sound;  
  }  
  
  speak() {  
    console.log(`${this.name} says ${this.sound}`);  
  }  
}  
  
class Dog extends Animal {  
  constructor(name) {  
    super(name, "Woof");  
  }  
}  
  
const dog = new Dog("Dog");  
dog.speak(); // Dog says Woof
```

Daily Tip
04-04-2020

JavaScript ES6 Sets



```
let magic = new Set();  
magic.add('🧙‍♀️');  
magic.add('🧙‍♂️');  
magic.add('🔥');  
console.log(magic.size); // 3  
magic.add('🧙‍♀️');  
console.log(magic.size); // 3  
  
console.log(magic.has('🧙‍♀️')); // true  
console.log(magic.has('🔥')); // false  
magic.delete('🧙‍♀️');
```

Конструктор дочірнього класу

Ключове слово extends

Використовується для вказання батьківського класу, від якого буде успадковувати дочірній клас

Ключове слово super

Посилання на конструктор батьківського класу. Дозволяє викликати батьківський конструктор та ініціалізувати успадковані властивості

```
class Animal {  
    constructor(name) {  
        this.name = name;  
    }  
    speak() {  
        console.log(`${this.name} makes a sound.`);  
    }  
}  
  
class Dog extends Animal {  
    constructor(name, breed) {  
        super(name);  
        this.breed = breed;  
    }  
    speak() {  
        console.log(` ${this.name} barks. `);  
    }  
}  
  
const dog = new Dog("Buddy", "Golden Retriever");  
console.log(dog.breed); // Golden Retriever  
dog.speak(); // Buddy barks.
```

Методи дочірнього класу

Поліморфізм

Дочірній клас може мати власні методи з такими самими іменами як у батьківського класу. Методи з одинаковим ім'ям можуть виконувати різні дії залежно від класу.

Власні методи

Будь-який клас може мати власні методи і властивості, незалежно від того, чи він наслідується від іншого класу.

```
class Shape {  
    constructor(color) { this.color = color; }  
    draw() {  
        console.log(`Drawing a shape with ${this.color} color.`);  
    }  
}  
  
class Circle extends Shape {  
    constructor(color, radius) {  
        super(color);  
        this.radius = radius;  
    }  
    printInfo() {  
        console.log(`INFO: Radius: ${this.radius}, color: ${this.color}`);  
    }  
    draw() {  
        console.log(`Drawing a circle with ${this.color} color and radius ${this.radius}.`);  
    }  
}
```

- **Значення this:** В методі об'єкта this вказує на об'єкт. В методі класу this вказує на поточний екземпляр класу.

Поліморфізм



Один інтерфейс

Дозволяє використовувати один і той самий інтерфейс для об'єктів різних класів



Гнучка заміна

Можливість замінювати один об'єкт іншим для гнучкої взаємодії з різними типами даних

```
class Animal {  
    makeSound() {  
        console.log("Some generic sound");  
    }  
}
```

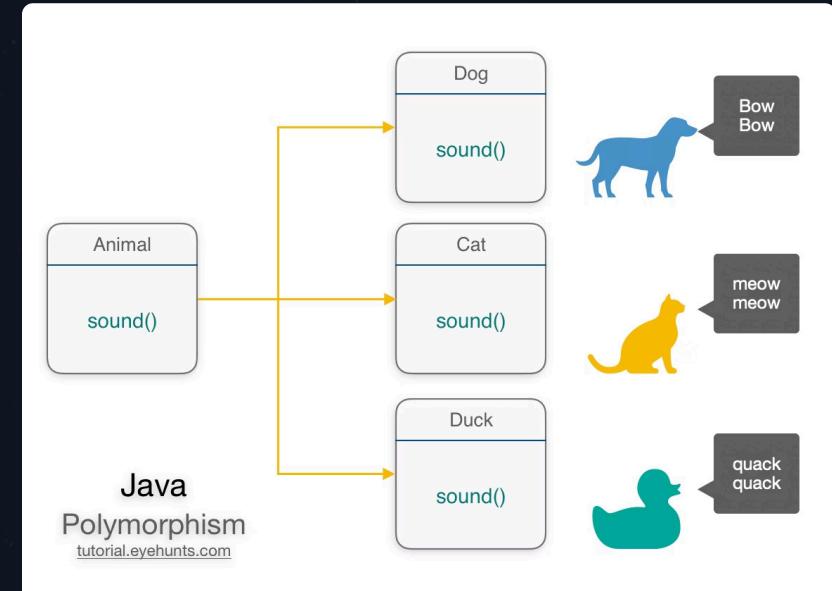
```
class Dog extends Animal {  
    makeSound() {  
        console.log("Woof woof!");  
    }  
}
```

```
class Cat extends Animal {  
    makeSound() {  
        console.log("Meow!");  
    }  
}
```

```
function animalSound(animal) {  
    animal.makeSound();  
}
```

```
const dog = new Dog();  
const cat = new Cat();
```

```
animalSound(dog); // Woof woof!  
animalSound(cat); // Meow!
```



Абстракція

Абстракція дозволяє створювати спрощений інтерфейс для взаємодії з об'єктами, приховуючи деталі реалізації. Це допомагає зосередитися на важливих аспектах та спростити взаємодію з об'єктами.



Приховання деталей

Внутрішня реалізація прихована від користувача класу



Фокус на важливому

Взаємодія через чіткий та зрозумілий інтерфейс



Спрощення

Складність системи стає керованою та зрозумілою

```
class Car {  
  constructor(make, model) {  
    this.make = make;  
    this.model = model;  
  }  
  
  drive() {  
    console.log(`${this.make} ${this.model} is driving.`);  
  }  
}  
  
const myCar = new Car("Toyota", "Camry");  
myCar.drive(); // Toyota Camry is driving.
```

Інкапсуляція



ENCAPSULATION

Encapsulation is a process of providing security & controlled access to the most important component of an object i.e., data members of an object.

security can be provided by two steps:

1. preventing direct access by declaring data members as private.
2. providing controlled access by using public setters and public getters.



Приховання внутрішнього стану

Інкапсуляція дозволяє приховувати деталі реалізації внутрішнього стану об'єкта від зовнішнього світу. Це забезпечує контроль доступу до внутрішніх деталей та гарантує абстракцію.

У JavaScript приватні поля позначаються символом `#` перед назвою.

```
class BankAccount {  
    #balance = 0;  
  
    deposit(amount) {  
        if (amount > 0) {  
            this.#balance += amount;  
        }  
    }  
  
    getBalance() {  
        return this.#balance;  
    }  
}  
  
const account = new BankAccount();  
account.deposit(100);  
console.log(account.getBalance()); // 100  
// console.log(account.#balance); // Error: приватне поле
```

Приватні поля недоступні ззовні класу, що забезпечує безпеку та цілісність даних.