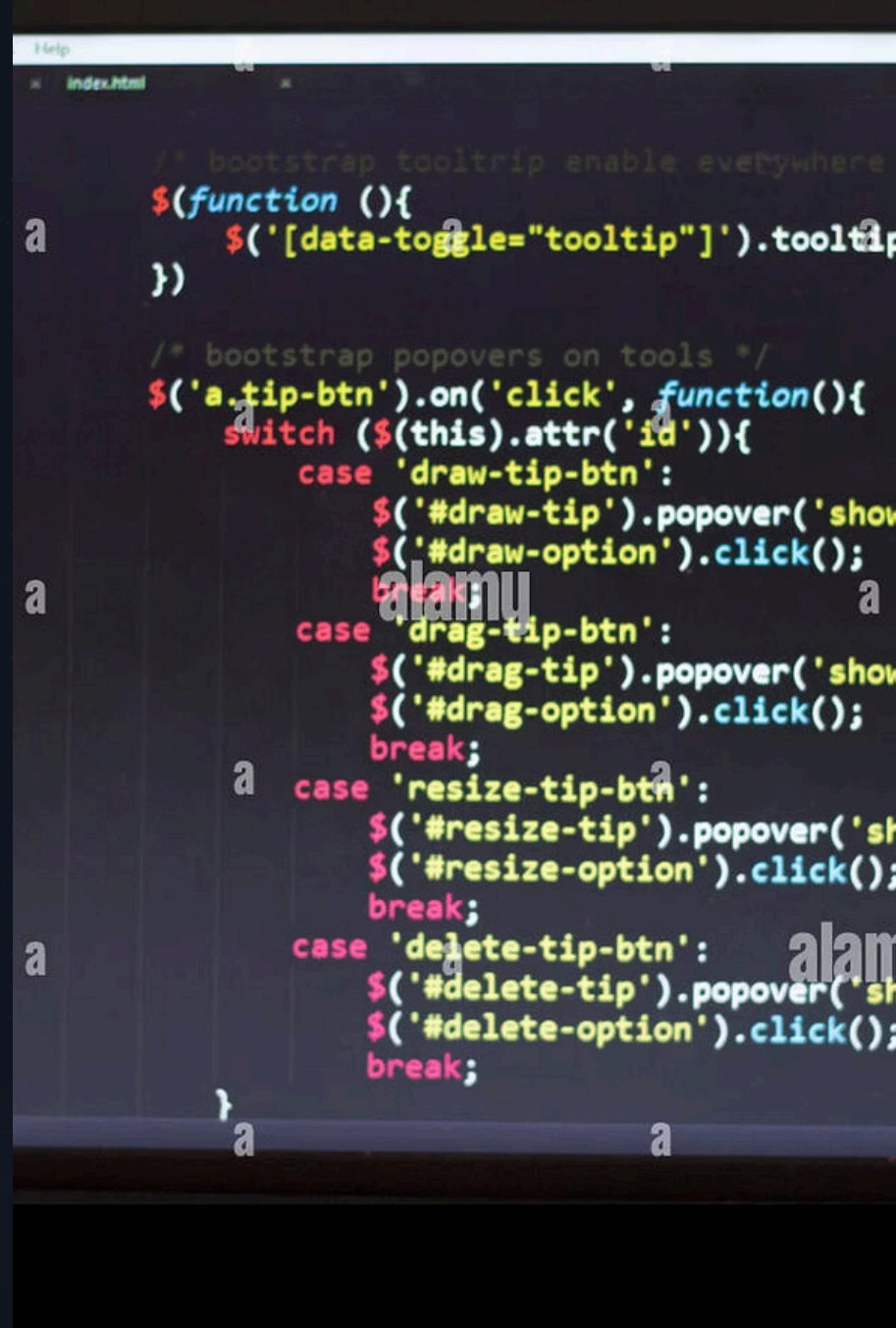
ООП в JavaScript

Об'єктно-орієнтоване програмування (ООП) — це парадигма програмування, яка базується на концепціях об'єктів та класів. В ООП програми організовані навколо об'єктів, які включають дані та методи для їх обробки. Розглянемо основні концепції ООП та різницю між різними парадигмами програмування в JavaScript.



Парадигми програмування

Процедурне програмування

Програма організована навколо послідовно виконуваних процедур або функцій. Дані та функції відокремлені.

```
function calculateSum(a, b) {
  return a + b;
}
const result = calculateSum(5, 3);
```

Функціональне програмування

Функції розглядаються як об'єкти першого класу. Працює з імутабельними даними.

```
const calculateSum = (a, b) => a + b;
console.log(calculateSum(5, 3));
```

Об'єктно-орієнтоване

Програма організована навколо об'єктів, які містять дані та методи для їх обробки.

```
class Calculator {
  calculateSum() {
    return this.a + this.b;
  }
}
```

У процедурному підході програма базується на послідовності функцій. У функціональному — на функціях та даних. У ООП — на об'єктах та класах.

ООП підхід в дії

Приклад класу

```
class Calculator {
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }

  calculateSum() {
    return this.a + this.b;
  }
}

const calculator = new Calculator(5, 3);
const result = calculator.calculateSum();
console.log(result); // 8
```

Ключові відмінності

- **Структура:** процедурний базується на функціях, функціональний — на чистих функціях, ООП — на об'єктах
- **Управління станом:** процедурний використовує глобальні змінні, функціональний — імутабельні дані, ООП — методи об'єктів
- **Переваги:** функціональне сприяє розділенню стану, ООП — управлінню складними структурами

Основні сутності ООП

JavaScript підтримує ООП за допомогою прототипної моделі. Об'єкти можуть мати властивості та методи, а їх структура визначається через прототипи або класи (синтаксис ES6).

Клас

Шаблон або опис для створення об'єктів. Визначає структуру та поведінку об'єкта, містить властивості та методи.

Екземпляр

Конкретний представник класу. Створюється на основі шаблону класу, має власні значення властивостей.

Інтерфейс

Визначає контракт — список методів, які має реалізувати клас. В JavaScript використовується "качина типізація".

Класи та екземпляри

Визначення класу

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is
${this.name}
    and I'm ${this.age} years old.`);
  }
}
```

Клас `Person` визначає структуру об'єкта з властивостями `name` та `age`, і методом `greet()`.

Створення екземплярів

```
const person1 = new
Person("Alice", 25);
const person2 = new Person("Bob",
30);

person1.greet();
// Hello, my name is Alice...
person2.greet();
// Hello, my name is Bob...
```

Кожен екземпляр має власні значення властивостей, але спадковує методи від класу.

Duck Typing в JavaScript

> "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"

У JavaScript немає класичних інтерфейсів через динамічну типізацію. Натомість використовується концепція **duck typing** — якщо об'єкт має необхідні властивості та методи певного типу, він може бути представлений як цей тип.

```
class Calculator {
  add(a, b) {
    return a + b;
  }

  subtract(a, b) {
    return a - b;
  }
}

const calc = new Calculator();
console.log(calc.add(5, 3));      // 8
console.log(calc.subtract(10, 4)); // 6
```

Об'єкт визначається не за його класом, а за його поведінкою та наявними методами.

Модифікатори доступу

Публічні

Доступні всюди — в класі, ззовні та в дочірніх класах

```
class Student {
  constructor(name) {
    this.name = name;
  }
}
```

Захищені (_)

Конвенція: доступні в класі та дочірніх класах

```
class BankAccount {
  constructor(balance) {
    this._balance = balance;
  }
}
```

Приватні (#)

Доступні тільки в самому класі

```
class Counter {
  #count = 0;

  #increment() {
    this.#count++;
  }
}
```

Конструктор

Спеціальний метод `constructor`, який викликається при створенні екземпляра для ініціалізації властивостей.

Властивості та методи

Властивості зберігають дані об'єкта. Методи — функції, які виконують дії над об'єктом.

Геттери та Сеттери

Ключові слова `get` і `set` дозволяють контролювати доступ до властивостей, виконувати додаткові дії при зчитуванні або записі значень. Це механізм інкапсуляції.

Геттер (get)

```
class Circle {
  constructor(radius) {
    this._radius = radius;
  }

  get radius() {
    return this._radius;
  }

  get area() {
    return Math.PI * this._radius ** 2;
  }
}

const circle = new Circle(5);
console.log(circle.radius); // 5
console.log(circle.area);   // 78.54
```

Сеттер (set)

```
class Temperature {
  constructor(celsius) {
    this._celsius = celsius;
  }

  set celsius(value) {
    if (value < -273.15) {
      console.log("Неможлива температура");
      return;
    }
    this._celsius = value;
  }

  get fahrenheit() {
    return this._celsius * 9/5 + 32;
  }
}
```

Геттери та сеттери зазвичай мають однакові імена і відображають ім'я захищеної властивості без спеціальних символів.

```
let sumFPChain = meetups.filter((m)=>{
    return m.isActive;
  })
  .map((m)=>{
```

Статичні методи та властивості

Статичні члени класу належать самому класу, а не його екземплярам. Використовуються для функціональності, не пов'язаної з конкретним об'єктом.

Статичні методи

Викликаються через клас, використовують ключове слово static

```
class MathHelper {
  static square(number) {
    return number * number;
  }
}

console.log(MathHelper.square(5)); // 25
```

Статичні властивості

Змінні або константи, що належать класу

```
class Configuration {
  static defaultLanguage = "en";
}

console.log(Configuration.defaultLanguage);
```

Практичне застосування

Допоміжні класи та утиліти

```
class Logger {
 static log(message) {
 console.log(`[LOG] ${message}`);
 }
}

Logger.log("Hello, world!");
```