

```
9
10 import { lightTheme, darkTheme } from './';
11 import useLocalStorage from '../hooks/useLocalStorage';
12
13 import NavBar from '../components/NavBar';
14
15 function App({ Component, pageProps }) {
16   const [currentTheme, setCurrentTheme] = useState(
17     localStorage.getItem('theme') || 'light'
18   );
19   useEffect(effect) => {
20     const jssStyles = document.querySelector('#jss-stylesheet');
21     if (jssStyles) {
22       jssStyles.parentElement.remove();
23     }
24   }, [deps]);
25
26   return (
27     <>
28       <Head>
29         <title>ECU-DEV</title>
30         <meta name="viewport" content="width=device-width, initial-scale=1" />
31       </Head>
32       <ThemeProvider theme={currentTheme}>
33         <ApolloProvider client={apolloClient}>
34           <CssBaseline />
35           <Container>
36             <Component {...pageProps} />
37           </Container>
38         </ApolloProvider>
39       </ThemeProvider>
40     </>
41   );
42 }
```

# Прототипне успадкування в JavaScript

Глибоке занурення у механізми прототипного успадкування та його практичне застосування в сучасній розробці

# Вбудовані класи JavaScript

## Що таке вбудовані класи?

Вбудовані об'єкти (вбудовані класи) в JavaScript — це заздалегідь визначені об'єкти, що надаються мовою для роботи з різними типами даних. Ці класи включають **Array**, **Object**, **String**, **Number**, **Boolean**, **Date** та багато інших.

Кожен з них надає набір методів та властивостей, специфічних для відповідного типу даних, що робить їх потужними інструментами для обробки даних у різних сценаріях програмування.

## Створення екземплярів

За допомогою кожного з цих вбудованих класів ми явним чи неявним чином створюємо їх екземпляри (інстанси). Наприклад:

```
[1, 2, 3] === new Array(1, 2, 3)
```

Кожного разу, коли ми створюємо новий інстанс будь-якого типу (окрім `null` і `undefined`, які не мають відповідного класу), ми утворюємо зв'язок між інстансом та класом, а точніше — з його прототипом.

LOREM IPSUM

## Vector

Lore ipsum dolor sit amet, consectetur  
adipiscing elit, sed do eiusmod tempor  
cididunt ut labore et dolore magna aliqua. Ut  
enim ad minim veniam, quis nostrud  
ercitation ullamco laboris nisi ut aliquip ex ea  
ommodo consequat. Duis aute irure dolor in  
reprehenderit in voluptate velit esse cillum  
dolore eu fugiat nulla pariatur. Excepteur sint  
ccaecat cupidatat non proident, sunt in culpa  
qui officia deserunt mollit anim id est laborum.



ID 110325044 © Kog

# Що таке прототип?

## Механізм успадкування

Прототип — це механізм, який дозволяє об'єктам спадковувати властивості та методи від інших об'єктів

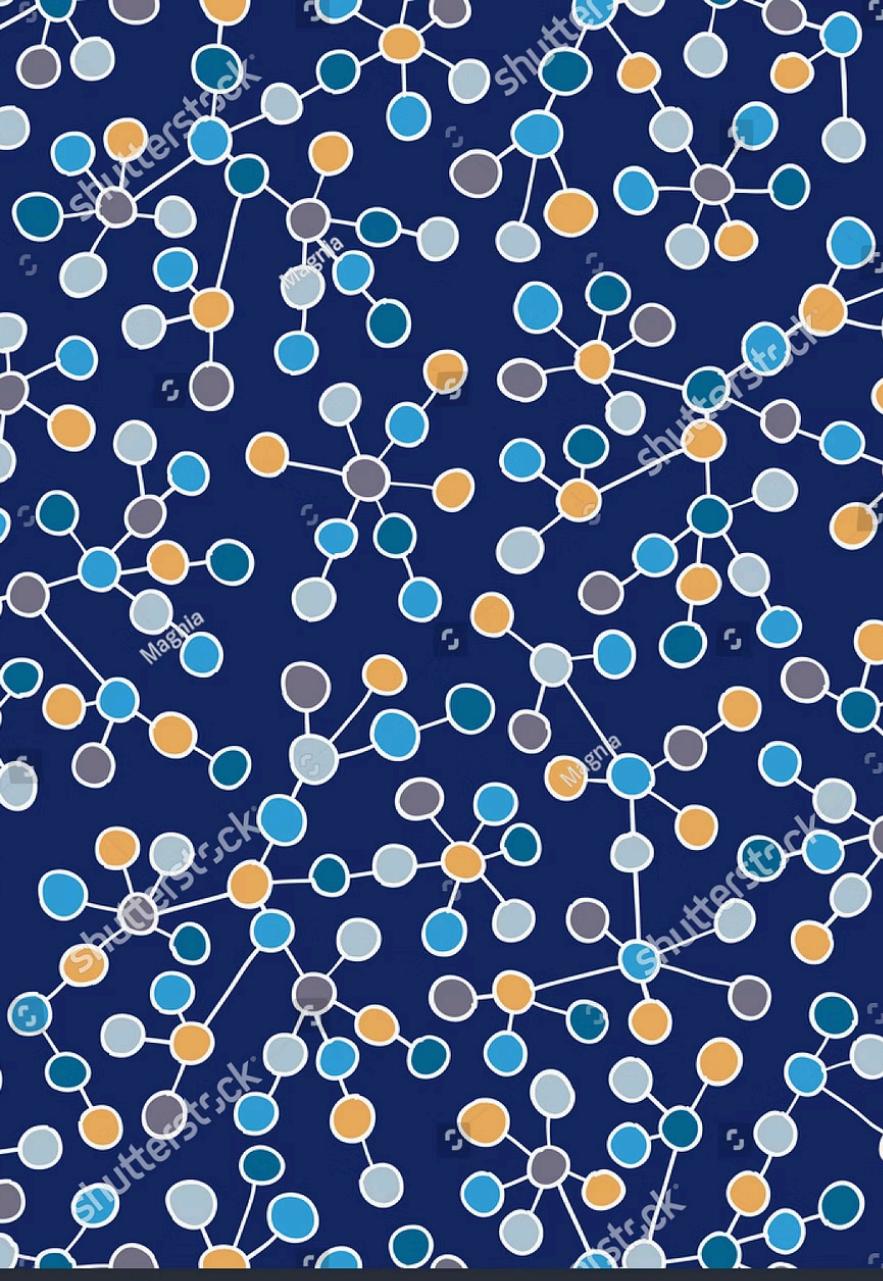
## Посилання на об'єкт

Кожен об'єкт в JavaScript може мати посилання на інший об'єкт, який вважається його «прототипом»

## Спільне використання коду

Це дозволяє об'єктам спільно використовувати код та функціональність, що робить програми більш ефективними

Іншими словами, прототипне успадкування — це фундаментальна концепція JavaScript, яка робить мову об'єктно-орієнтованою та дозволяє створювати більш елегантний та підтримуваний код.



# Ланцюжок прототипів

## Властивість `prototype`

Кожен з вбудованих класів — це функція, яка містить властивість `prototype`. Ця властивість є об'єктом і містить у собі всі вбудовані методи класу.

Наприклад, `Array.prototype` містить методи `map`, `filter`, `reduce` та інші.

## Властивість `__proto__`

Кожен об'єкт містить властивість `__proto__`, яка містить посилання на свій `prototype`. Таким чином утворюється ланцюжок прототипів.

Завдяки цьому ланцюжку ми отримуємо доступ до властивостей та методів, які зберігаються у прототипі.

- **Приклад:** Масив `[1, 2, 3]` має метод `map` з `Array.prototype`. Але механізми читання та запису властивостей дещо відрізняються.

# Механізми [[Get]] та [[Set]]



## Механізм [[Get]]

Коли JavaScript виконує операцію `obj.property`, він спочатку перевіряє, чи є така властивість в самому об'єкті `obj`.

Якщо властивість існує — її значення повертається одразу.

## Пошук у прототипах

Якщо властивість не існує в самому об'єкті, JavaScript йде ланцюжком прототипів, починаючи з поточного об'єкта.

Пошук продовжується, доки властивість не буде знайдена або не досягнуто кінця ланцюжка.

## Результат пошуку

Як тільки властивість знайдена в одному з об'єктів прототипу — її значення повертається.

Якщо властивості немає — повертається `undefined`.

## Механізм [[Set]]

Механізм [[Set]] працює інакше. На відміну від [[Get]], якщо необхідної властивості у цільовому об'єкті не буде знайдено, то вона буде **автоматично створена** у самому об'єкті, а не в прототипі.

Якщо властивість вже існує в об'єкті — вона буде оновлена новим значенням.



# Shadowing — затінення властивостей

Property shadowing (затінення властивості) — це ситуація, коли об'єкт має властивість з тим самим ім'ям, що й властивість об'єкту-прототипа.

01

## Властивість у прототипі

Спочатку метод або властивість існує в об'єкті-прототипі

02

## Створення однайменної властивості

У дочірньому об'єкті створюється властивість з таким самим ім'ям

03

## Затінення прототипу

Властивість внутрішнього об'єкта «затіняє» властивість прототипу, роблячи її недоступною для прямого доступу

```
const user = {  
  toString() {  
    return '[object User]';  
  }  
};
```

У цьому прикладі власний метод `toString()` затіняє стандартний метод `toString()` з `Object.prototype`.

# Переваги прототипного успадкування

Прототипне успадкування — це не просто особливість JavaScript, а потужний інструмент, який надає розробникам численні переваги.



## Простота та гнучкість

Прототипне успадкування надає простий та гнучкий спосіб створення та організації об'єктів. Ви можете створювати об'єкти, додавати методи та властивості до прототипів, і успадковувати їх у міру необхідності без складних конструкцій.



## Ефективне використання пам'яті

У прототипному успадкуванні методи та властивості поділяються між усіма об'єктами, що успадковують від одного прототипу. Це дозволяє заощаджувати пам'ять, оскільки методи не дублюються для кожного об'єкта окремо.



## Відкритість для розширення

Прототипне успадкування дозволяє додавати нові методи та властивості до об'єктів без зміни вихідного коду. Це забезпечує відкритість для розширення та легку підтримку змінних вимог проєкту.



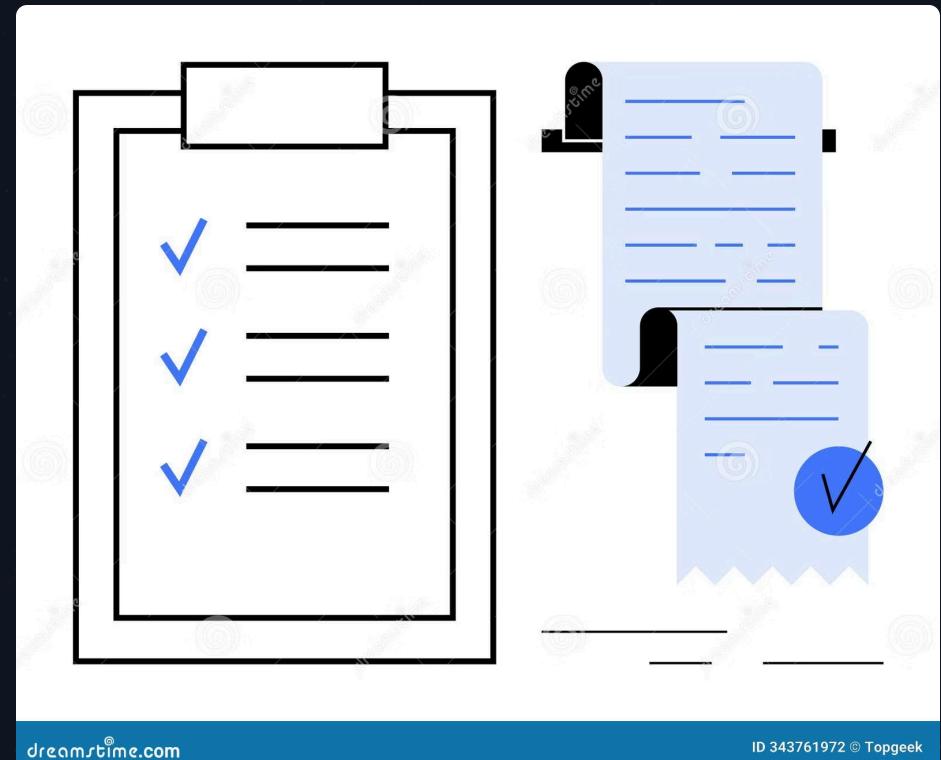
# Способи перевірки прототипу

## Метод `hasOwnProperty`

Метод `hasOwnProperty` — це вбудований метод об'єктів, який дозволяє перевірити, чи містить об'єкт **власну** (не успадковану) властивість із заданим ім'ям.

Цей метод приймає рядок, що представляє ім'я властивості, і повертає:

- `true` — якщо об'єкт містить власну властивість із цим ім'ям
- `false` — якщо властивість успадкована або відсутня



dreamstime.com

ID 343761972 © Topgeek

```
const person = {  
  name: 'John',  
  age: 42  
};  
  
console.log(person.hasOwnProperty('age'));  
// true  
  
console.log(person.hasOwnProperty('toString'));  
// false
```

У прикладі вище властивість `age` є власною властивістю об'єкта `person`, тому метод повертає `true`. Властивість `toString` успадкована від `Object.prototype`, тому повертається `false`.

# Використання hasOwnProperty з циклом for...in

Метод `hasOwnProperty` особливо корисний для виключення успадкованих властивостей при переборі властивостей об'єкта з використанням циклу `for...in`.

```
const user = {
  login() {
    return true;
  }
}

const person = {
  name: 'John',
  age: 42,
  __proto__: user
};

for (const item in person) {
  if (person.hasOwnProperty(item)) {
    console.log('власна властивість - ', item, ':', person[item]);
  } else {
    console.log('властивість прототипу - ', item, ':', person[item]);
  }
}
```

## Без `hasOwnProperty`

Цикл `for...in` перебирає **всі** властивості об'єкта, включаючи успадковані від прототипу

## З `hasOwnProperty`

Ми можемо відфільтрувати лише власні властивості об'єкта, ігноруючи успадковані



# Object.hasOwn — сучасна альтернатива



Метод `Object.hasOwn()` — це сучасніша альтернатива `hasOwnProperty`. Він повертає `true`, якщо вказана властивість є власною властивістю об'єкта (не успадкованою), навіть якщо значення властивості дорівнює `null` або `undefined`.

```
const user = {
  login() {
    return true;
  }
}

const person = {
  name: 'John',
  age: 42,
  __proto__: user
};

console.log(Object.hasOwn(person, 'name'));
// true

console.log(Object.hasOwn(person, 'login'));
// false
```

## Переваги `Object.hasOwn`

- Працює для об'єктів без прототипу
- Працює з об'єктами, що перевизначили `hasOwnProperty`
- Більш інтуїтивно зрозумілий синтаксис

## Відмінність від оператора `in`

На відміну від оператора `in`, цей метод **не перевіряє** зазначену властивість в ланцюжку прототипів об'єкта — тільки власні властивості.

**Рекомендація:** Використовуйте `Object.hasOwn()` замість `Object.hasOwnProperty()` у новому коді для кращої надійності та читабельності.

# Object.getOwnPropertyNames

Метод **Object.getOwnPropertyNames()** повертає масив з іменами всіх власних (не успадкованих) перерахованих властивостей об'єкта.

## Як працює метод

Цей метод приймає об'єкт як аргумент і повертає масив рядків, які містять імена власних властивостей цього об'єкта.

Метод *не включає* властивості з прототипу — тільки ті, що належать безпосередньо об'єкту.

## Приклад використання

```
const user = {
  login() {
    return true;
  }
};

const person = {
  name: 'John',
  age: 42,
  __proto__: user
};

console.log(
  Object.getOwnPropertyNames(person)
);
// ['name', 'age']
```

Як бачите, метод `login` з прототипу `user` не потрапив до результату, оскільки він не є власною властивістю об'єкта `person`.

D	E	F	G	H	I
	Last Name	First Name	Home Room	Period	Teacher
1	Adams	Aaliyah	Adams		Fraley
3	Addington	Aiden	Addington		Seaver
2	Addington	Arthur	Addington		Ladd
3	Addington	Dalton	Addington		White
5	Addington	Oliver	Addington		Blackwell
	Addington	Susan			
	Addington	Dalton			
1	Akers	Olivia	Akers		Turner
1	Alvarez	Luis	Alvarez		White
2	Antonio	Elizabeth	Antonio		Blackwell
2	Apel	Kaylee	Apel		Culbert
5	Archer	Ace	Archer		Turner
4	Archer	Harlen	Archer		Gilmer
5	Barber	Kaedyn	Barber		Blackwell
6	Barber	Khylin	Barber		Hill
6	Barber	Marion	Barber		Ladd
4	Barnett	Destini	Barnett		Hamm
2	Barnett	Hayley	Barnett		Seaver
2	Barnett	Jordan	Barnett		Addison
3	Barnett	Zachary	Barnett		Hill
5	Barrientos	Gabriella	Barrientos		Turner
6	Barton	Damian	Barton		Ladd
6	Beckham	Kaylee	Beckham		Fraley
6	Berry	Braxton	Berry		Heads
2	Berry	Gavin	Berry		Hill
1	Berry	Ryan	Berry		Fraley
3	Bingham	Riley	Bingham		Meadow
	Blackwell	Kellie			
5	Blackwell	Rylie	Blackwell		Gilmer
1	Bledsoe	Aleigha	Bledsoe		Culbert
3	Boggs	Dylan	Boggs		Carter
4	Bowen	Alley	Bowen		Meadow
6	Brandon	Kyrie	Brandon		Blackwell
5	Brickey	Dakota	Brickey		Culbert
1	Broadwater	Cori	Broadwater		White
2	Broadwater	Jonathan	Broadwater		Warren
2	Brown	Danielle	Brown		Hill
5	Bruke	Brendan	Bruke		Heads
4	Burchfield	Sebastian	Burchfield		Gilmer
4	Burke	Isabella	Burke		Addison

# Object.getOwnPropertySymbols

Метод `Object.getOwnPropertySymbols()` дозволяє отримати масив всіх власних символічних (не успадкованих) властивостей об'єкта.

## Що таке символільні властивості?

Символьні властивості є унікальними та не перераховуються, що означає, що їх не буде видно при звичайному переборі властивостей об'єкта з використанням циклу `for...in` та `Object.getOwnPropertyNames()`.

## Коли використовувати?

Символи корисні, коли потрібно додати приватні або унікальні властивості до об'єкта, які не конфліктують з іншими властивостями та не видимі при стандартному переборі.

```
const symbolA = Symbol('Prototype symbol');
const symbolB = Symbol('Own symbol');

const user = {
  [symbolA]: 'Prototype symbol',
  login() { return true; }
}

const person = {
  [symbolB]: 'Own symbol',
  name: 'John',
  age: 42,
  __proto__: user
};

console.log(Object.getOwnPropertyNames(user));
// ['login']

console.log(Object.getOwnPropertyNames(person));
// ['name', 'age']

console.log(Object.getOwnPropertySymbols(user));
// [Symbol(Prototype symbol)]

console.log(Object.getOwnPropertySymbols(person));
// [Symbol(Own symbol)]
```

# Object.getPrototypeOf

## Безпечний доступ до прототипу

Метод `Object.getPrototypeOf()` повертає прототип (тобто внутрішню властивість `[[Prototype]]`) зазначеного об'єкта.

Цей метод є рекомендованим способом отримання прототипу об'єкта, оскільки він безпечніший за використання властивості `_proto_`.



```
const user = {  
  login() {  
    return true;  
  }  
}  
  
const person = {  
  name: 'John',  
  age: 42,  
  _proto_: user  
};  
  
console.log(Object.getPrototypeOf(person));  
// user
```

- **Важливо:** Хоча властивість `_proto_` можна використовувати для отримання прототипу об'єкта, її використання **не рекомендується** через можливі проблеми сумісності та безпеки. Саме для вирішення цих проблем і був запроваджений метод `Object.getPrototypeOf()`.

# isPrototypeOf — перевірка зв'язку прототипів

Метод `isPrototypeOf()` — це вбудований метод, який дозволяє перевірити, чи є об'єкт прототипом для іншого об'єкта. Цей метод повертає `true`, якщо об'єкт є прототипом іншого об'єкта, і `false` в іншому випадку.

01

## Викликаємо метод на прототипі

Метод викликається на об'єкті, який ми хочемо перевірити як прототип

02

## Передаємо цільовий об'єкт

В якості аргументу передається об'єкт, для якого перевіряємо зв'язок

03

## Отримуємо результат

Метод повертає булеве значення про наявність зв'язку в ланцюжку прототипів

## Синтаксис та приклад використання:

```
const user = {  
  login() {  
    return true;  
  }  
};  
  
const person = {  
  name: 'John',  
  age: 42,  
  __proto__: user  
};  
  
console.log(user.isPrototypeOf(person));  
// true  
  
console.log(person.isPrototypeOf(user));  
// false
```

Метод `isPrototypeOf()` корисний, коли потрібно програмно перевірити відносини успадкування між об'єктами, особливо в складних ієрархіях прототипів.