# Розробка клієнтських сценаріїв з використанням JavaScript та бібліотеки jQuery

STEP computer ACADEMY

# Урок 2-2

Введення в ООП

## Contents

# Що таке ООП?

Об'єктно-орієнтоване програмування передбачає, що ви створюєте певний шаблон об'єкта, у якому описуєте його характеристики (певні змінні, або ключі об'єкта) і методи взаємодії з цим об'єктом (функції), моделюючи у коді поведінку реальних об'єктів. Тобто цей шаблон або фабрика, яка займається створенням об'єктів одного типу працює з кодом приблизно так само, як реальна фабрика штампує, наприклад, однотипні фломастери різних кольорів або випускає набагато складніші по своїй конструкції автомобілі. При об'єктно-орієнтованому підході у програмуванні кожен об'єкт повинен бути інтуїтивно зрозумілою сутністю, яка має методи і властивості, котрі чітко описують даний об'єкт.

Коли розглядають ООП у різних мовах програмування, то зазвичай говорять про створення об'єктів на основі класів. З 2015 року (стандарт ES6, або EcmaScript2015) класи з'явилися і в JavaScript, хоча і до цього часу в ньому можна було створювати об'єкти, які успадковували властивості від основного об'єкта-прототипу. І навіть з появою класів цей механізм не дуже змінився. Тобто ООП в JavaScript засноване на прототипах. Тепер необхідно розібратися, що таке прототип.

**Прототип** — це об'єкт, який містить дані про властивості та методи всіх об'єктів одного типу. Крім того, всі об'єкти є спадкоємцями класу Object, описаного в ядрі JS. За допомогою цього класу ви створювали літерал об'єкта. Тобто можна сказати, що прототипом будь-якого об'єкта у JS є клас Object.

Якщо порівнювати структуру взаємозв'язку об'єктів, побудовану на основі прототипів, з реальною сім'єю, то можна сказати, що в основі кожного роду (сім'ї) знаходяться 2 особи, які визначають риси (властивості або характеристики) майбутніх поколінь та варіанти їхньої поведінки або роботи. Наприклад, у сім'ї людей з темним волоссям і карими очима народжуватимуться діти (створюватимуться об'єкти) з такими ж характеристиками, хоча колір волосся можна змінити (призначити властивості нове значення). Крім того, в одній сім'ї можуть народжуватися діти з музичними здібностями або діти, які згодом стануть ювелірами або шевцями, або бізнесменами, бо в їхньому роді таку професію або сферу діяльності обирали всі. З погляду JavaScript, сфера діяльності — це вже метод, або функція, причому вона успадковується усіма об'єктами з одним прототипом автоматично, як музичні здібності в сім'ї.

Прототипи в JS — це можливість «передати у спадок» усім об'єктам одного типу певні характеристики таким чином, щоб кожен з об'єктів міг скористатися цими характеристиками у певній ситуації.

Приклад такої ситуації — це використання методів масиву. Наприклад, у нас є 2 групи студентів, які представлені у вигляді масивів. Нам потрібно перевести 2-х студентів з однієї групи до іншої. Зробимо це за допомогою методів масивів splice() та concat():

```
let group1 = ['Deniels', 'Jonhson', 'Overton',
              'Stufford', 'Templeton'],
group2 = ['Greenwood','Liner','Takerman'];
let students = group1.splice(2, 2);
console.log(students);
```

```
group2 = group2.concat(students);
console.log(group1, group2);
```

Результатом наших дій будуть 3 масиви в консолі: перший зі студентами, «видаленими» методом splice() з group1, а два інших — нові масиви group1 та group2.

Зараз нас більше цікавить той факт, що прототипом кожного масиву є об'єкт Array з набором методів, доступних кожному масиву (тобто об'єкту типу Array), але при цьому для кожного з наших масивів ми використовували тільки той метод, який був нам потрібен, виходячи з умов поставленого завдання.

А це означає, що прототип Array може зберігати певну кількість інформації, реалізовану у вигляді функцій (наприклад, на рисунку 2 це позначено у вигляді f concat()), але вони використовуються за потреби.


*Рисунок 1. (Початок)*

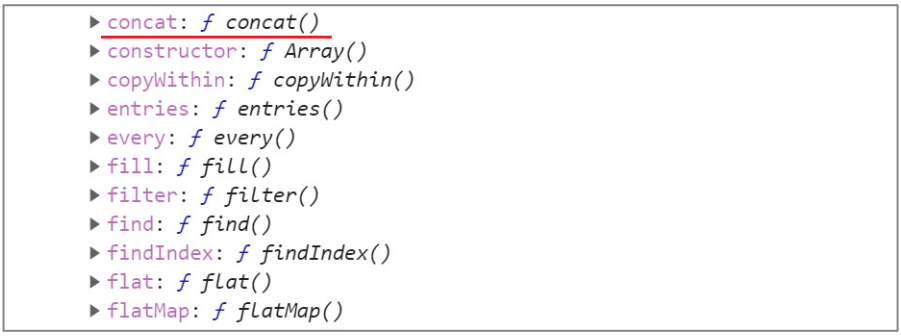


*Рисунок 2. (Закінчення)*

Повернемося до аналогії з сім'єю та музичними здібностями. Хтось у такій сім'ї може грати на роялі професійно або для себе, або співати (метод використовується), хтось нізащо не підійде до інструменту. Однак діти цих людей можуть знову-таки використати свій музичний хист або ніколи про нього не згадувати, але здібності, які притаманні цій родині, все одно зберігаються

Так і JavaScript — методи, реалізовані в прототипі, автоматично успадковуються, тобто передаються об'єкту такого типу, причому незалежно від того, чи використовуватиме об'єкт ці методи.

Тепер перейдемо до питання, як краще створювати об'єкти одного типу (або з одним прототипом) в JavaScript. Так от для того, щоб створити основу для прототипування об'єкта в JavaScript, зазвичай використовують функції-конструктори або класи.

# Функції-конструктори в JavaScript

На основі класів або функцій-конструкторів можна створити екземпляри об'єктів, з якими потім програміст працюватиме у коді.

Необхідність використання функцій-конструкторів, у порівнянні зі створенням літералів об'єктів, виникає тоді, коли об'єктів з однаковими властивостями та методами має бути не один-два, а декілька. Наприклад, нам потрібно створити групу студентів, у якій кожен студент — це окремий об'єкт з погляду маніпулювання ним у JavaScript.

Мінімально необхідною інформацією для таких об'єктів-студентів буде ім'я, прізвище та дата народження. Крім того, нам знадобляться два методи для кожного зі студентів: перший — для виведення в консоль інформації про нього, другий — для інформації про вік студента, що розраховується на основі поточної дати та дати його народження за допомогою методів об'єкта Date. Для початку створимо 2 таких студентів:

```
let diana = {
    firstname: 'Diana',
    lastname: 'Fenton',
    birthday: '07/22/1996',
    showInfo: function(){
        console.log('Student name: ' + this.firstname +
                    ' ' + this.lastname);
},
```

```
  showAge: function(){
    const deltaTime =  Date.now() -
                       Date.parse(this.birthday);
    const studentAge = Math.floor(deltaTime/
                       (365*24*60*60*1000));
    console.log(this.firstname + ' ' + this.lastname +
                ' is ' + studentAge + ' years old.');
  }
}
let luis = {
  firstname: 'Luis',
  lastname: 'Melitano',
  birthday: '02/06/2002',
  showInfo: function(){
    console.log('Student name: ' + this.firstname +
                ' ' + this.lastname);
  },
  showAge: function(){
    const deltaTime =  Date.now() -
                       Date.parse(this.birthday);
    const studentAge = Math.floor(deltaTime/
                       (365*24*60*60*1000));
    console.log(this.firstname + ' ' + this.lastname +
                ' is ' + studentAge +' years old.');
  }
}
console.log(diana);
diana.showInfo();
diana.showAge();
console.log(luis);
luis.showInfo();
luis.showAge();
```

На рисунку 3 можна побачити, що обидва об'єкти у нас мають властивості прототипу Object, а також 2 однакові функції.

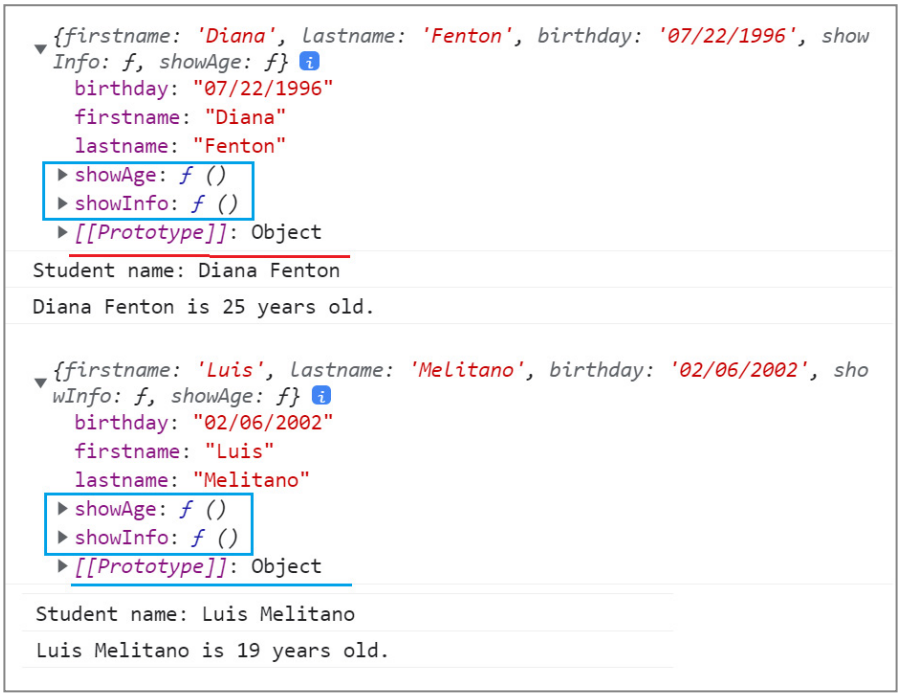


*Рисунок 2*

Якщо подивитися на масив коду, який нам потрібно дублювати та змінювати тільки для властивостей, то виникає питання: «Чи є можливість якось скоротити повторювані дії?». Звичайно, можна спробувати використати масиви та цикли, але це буде не найкращий варіант. Для вирішення цього завдання в JavaScript існують функції-конструктори та класи. Фактично, це два способи створити безліч однотипних об'єктів.

Оскільки функції-конструктори з'явилися в JavaScript значно раніше, ніж класи, давайте спочатку створимо таку функцію, яка буде відповідати за об'єкт, який містить потрібні нам дані про студента: його ім'я, прізви-

ще та дату народження, які ми задаватимемо відразу при створенні об'єкта за допомогою такої функції, передаючи їх у якості параметрів. У функції-конструкторі ці дані стануть властивостями нашого об'єкта.

Також, додаємо методи (функції) showInfo() та showAge(), які ми вже описували під час створення літералів об'єктів-студентів. Тобто ми візьмемо синтаксис літералу об'єкта і дещо змінимо в ньому відповідно до правил JavaScript та нашого завдання.

У результаті замінимо синтаксис, який застосовували в об'єктах-літералах, на аналогічний, але універсальний для одного типу об'єктів (Student у прикладі нижче). Спочатку нам потрібно буде визначити властивості об'єкта, перерахувавши їх за допомогою ключового слова this, наприклад: this.firstname = firstname. Це означає, що у властивість firstname даного об'єкта (this) ми запишемо значення, яке передається з аргументом firstname.

З методами-функціями аналогічна ситуація: ми описуємо метод this.showInfo() = function(){…}, записуючи до неї (функції) висновок у консоль даних про об'єкт-студент, екземпляр якого створюємо у вигляді змінної:

```
let michael = new Student('Michael', 'Dowson',
                          '11/23/2001');
```

Ключове слово this вказує на те, що метод викликається саме для цього екземпляра об'єкта Student, використовуючи саме його дані про ім'я та прізвище, а не дані якогось іншого об'єкта.

Функціям-конструкторам, на відміну від звичайних функцій, прийнято давати імена з великої літери, тому ми напишемо такий код:

```
function Student(firstname, lastname, birthday){
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.showInfo = function(){
        console.log('Student name: ' + this.firstname +
                    ' ' + this.lastname);
    }
    this.showAge = function(){
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
        const studentAge = Math.floor(deltaTime/
                          (365*24*60*60*1000));
        console.log(this.firstname + ' ' + this.lastname +
                    ' is ' + studentAge + ' years old.');
    }
}
let michael = new Student('Michael', 'Dowson',
                          '11/23/2001');
michael.showInfo();
michael.showAge();
let lisa = new Student('Lisa', 'Paltrow', '08/12/1998');
lisa.showInfo();
lisa.showAge();
console.log(michael, lisa);
```

Ви, напевно, поцікавитеся, що таке this всередині функції-конструктора? Це спеціальне ключове слово, яке посилається на поточний об'єкт, створений на основі даної функції-конструктора. Такими об'єктами в нашому коді є змінні michael та lisa, які створюються аналогічно

до масивів (let arr = new Array(3,4,78,9);) за допомогою ключового слова new.

Іншими словами this для michael вказує на дані студента '*Michael Dowson*' з датою народження '*11/23/2001*', а для lisa — на студентку '*Lisa Paltrow*', що народилася '*08/12/1998*'.

Саме тому в консолі ми побачимо такий текст під час викликів методів showInfo() та showAge().

```
Student name: Michael Dowson
Michael Dowson is 19 years old.
Student name: Lisa Paltrow
Lisa Paltrow is 22 years old.
```

*Рисунок 3*

Слід також уточнити, що michael та lisa є екземплярами об'єкта Student і всі властивості, та методи, характерні для цього об'єкта є у кожному з отриманих екземплярів.

В останньому рядку коду ми виводимо в консоль змінні michael та lisa, і при розкритті стрілок зліва від кожного об'єкта Student бачимо, які вони мають властивості та методи, а також бачимо, що прототипом для них є Object, а функція Student вказана як конструктор, звідки й пішла її назва.

**Відмінністю функції-конструктора** від звичайної функції є те, що **вона повертає створений об'єкт певного типу** (Student у нашому прикладі) без використання ключового слова return, тоді як звичайна функція повертає або те, що зазначено після return, або undefined, якщо ключове слово return не використовується.

Давайте для прикладу створимо ще одну функцію-конструктор Hotel, яка описуватиме певний готель в якійсь країні, причому нас буде цікавити крім назви готелю та його розташування, ще й кількість зайнятих та незайнятих у ньому номерів. Кількість зайнятих номерів ми передаватимемо у функцію-конструктор у якості параметра, як і загальну кількість номерів, а кількість вільних номерів та їх відсоток будемо отримувати за допомогою методів, тому що це нескладно розрахувати.

Відразу ж створимо три змінні-екземпляри Hotel і викличемо для них зазначені методи.

Код прикладу:

```
function Hotel (name, country, rooms, bookedRooms){
    this.name = name;
    this.country = country;
    this.rooms = rooms;
    this.bookedRooms = bookedRooms;
    this.availableRooms = function(){
        return this.rooms - this.bookedRooms;
    }
    this.availablePercent = function(){
        return Math.floor(this.availableRooms() /
                          this.rooms *100) +
                          '%';
    }
}

let antiqueRomanPalace = new Hotel('Antique Roman Palace',
                                   'Turkey', 270, 130),
sharmDreamsClub = new Hotel('Sharm Dreams Club',
                                   'Egypt', 320, 212),
miramarenHotel = new Hotel('Miramaren Hotel',
                                   'Greece', 70, 63);
```

```
console.log(antiqueRomanPalace.availableRooms(),
            antiqueRomanPalace.availablePercent());

console.log(sharmDreamsClub.availableRooms(),
            sharmDreamsClub.availablePercent());

console.log(miramarenHotel.availableRooms(),
            miramarenHotel.availablePercent()); //
```

У консоль буде виведено наступне:

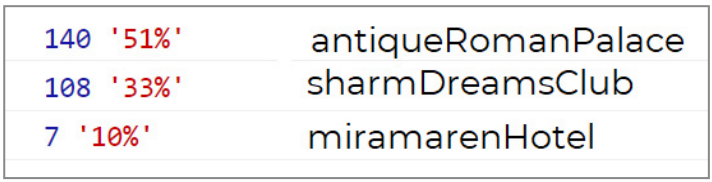


*Рисунок 4*

Отже, бачимо, що на основі переданих параметрів об'єктів типу Hotel, яких може бути величезна кількість, як і готелів по всьому світу, ми отримуємо розрахункові дані з методів.

Зверніть увагу, що в методі availablePercent() ми використовуємо метод availableRooms() для розрахунку вільних на цей час номерів, щоб не дублювати код.

Якщо інформація про готель оновиться, наприклад, кількість зайнятих номерів збільшиться, то повторний виклик методів дасть нам нову інформацію. Наприклад, в останньому готелі забронювали ще 4 номери:

```
miramarenHotel.bookedRooms += 4;
console.log(miramarenHotel.availableRooms(),
            miramarenHotel.availablePercent()); /
```

У консолі ми побачимо змінені дані у вигляді *3 '4%'*.

Як бачите, з властивостями об'єкта, створеного через функцію-конструктор, можна проводити й арифметичні дії, оскільки сама властивість має числовий тип даних.

Повернемося до прикладу зі студентами та ще раз розглянемо, про яку інформацію ми можемо дізнатися з консолі (рис. 6).


*Рисунок 5*

На цьому рисунку можна помітити, що методи showInfo() і showAge() належать кожному з об'єктів, тоді як у стандартних об'єктах типу Array, String, Date, розглянутих в рамках цього уроку, майже всі методи знаходяться в прототипі. Це можна переглянути у довідковій інформації на MDN.

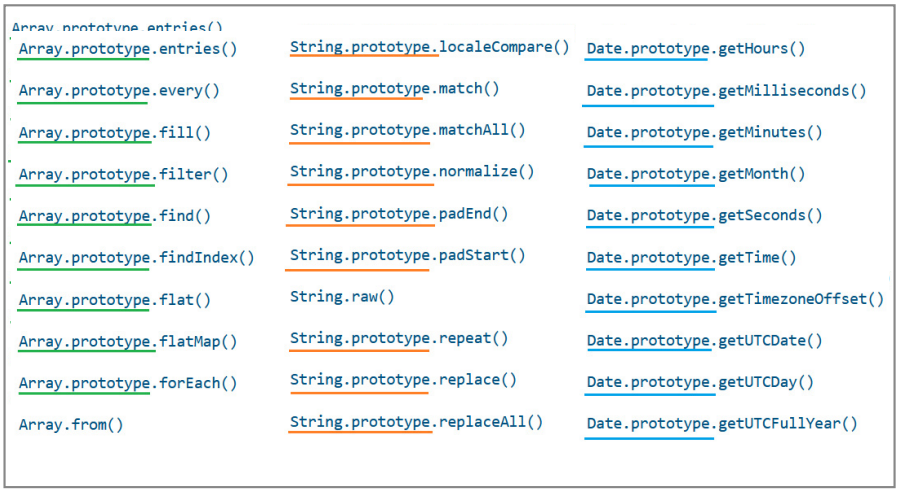


*Рисунок 6*

Чому це так? Річ у тому, що прототип — це носій важливої інформації про об'єкт, яким ви можете скористатися або можете залишити без уваги. Це все одно, що ген людини: потенційно носій гена може мати здібності до музики або спорту, але, якщо ці дані не розвивати, тобто не викликати функції, відповідальні за ці вміння, тоді ми ніколи не отримаємо з Майкла або Лізи ні музиканта, ні спортсмена.

Тому, в JavaScript всі методи прийнято записувати для прототипу. Використовувати їх чи ні для кожного об'єкта, залежатиме від завдання, у якому використовуються екземпляри об'єкта. Крім того, коли методи винесені в прототип, кожен об'єкт не «несе їх із собою», тому кількість оперативної пам'яті, що використовується, для скрипту дещо зменшується.

В коді функції-конструктора зараз буде дещо змінено:

```
function Student(firstname, lastname, birthday){
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
}

Student.prototype.showInfo = function(){
    console.log('Student name: ' + this.firstname +
                ' ' + this.lastname);
}

Student.prototype.showAge = function(){
    const deltaTime = Date.now() -
                      Date.parse(this.birthday);
    const studentAge = Math.floor(deltaTime/
                      (365*24*60*60*1000));
    console.log(this.firstname + ' ' + this.lastname +
                      ' is ' + studentAge +
                      ' years old.');
}

let michael = new Student('Michael', 'Dowson',
                          '11/23/2001');
michael.showInfo();
michael.showAge();
let lisa = new Student('Lisa', 'Paltrow', '08/12/1998');
lisa.showInfo();
lisa.showAge();
console.log(michael, lisa);
```

Стосовно виклику методів для змінних-екземплярів класу Student нічого не зміниться: вони також відображатимуть інформацію. Однак у консолі побачимо, що методи перемістилися у прототип об'єкта.


*Рисунок 7*

Для прикладу з готелями можна вивести методи в прототип, записавши так:

```
function Hotel (name, country, rooms, bookedRooms){
    this.name = name;
    this.country = country;
    this.rooms = rooms;
    this.bookedRooms = bookedRooms;
}
Hotel.prototype.availableRooms = function(){
    return this.rooms - this.bookedRooms;
}
Hotel.prototype.availablePercent = function(){
    return Math.floor(this.availableRooms() /
                      this.rooms *100) + '%';
}
```

```
antiqueRomanPalace.bookedRooms -=12;// было 130 из 270
console.log(antiqueRomanPalace.availableRooms(),
            antiqueRomanPalace.availablePercent());
```

На цей раз у консолі ми виявимо значення *152 '56%'*.

Як ми бачимо з підрахунків, методи, як і раніше, працюють коректно при зміні даних для однієї зі змінних.

# Класи в JavaScript

Синтаксис класу в JavaScript передбачає, що ви використовуєте ключове слово class, ім'я (назву) цього класу з великої літери та всередині нього описуєте головну функцію-конструктор, яка так і називається — constructor. У цій функції зазвичай ви вказуєте всі властивості класу, а методи описуєте, як інші функції, всередині класу.

Розглянемо на практиці, як ми можемо створити клас JavaScript.

Припустимо, нам необхідно реалізувати у вигляді класу код, аналогічний тому, що ми створювали у функції-конструкторі Student. Щоб не було конфлікту імен, назвемо наш клас Human і запишемо в ньому конструктор з двома методами.

```
class Human {
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
    }

    showInfo() {
        console.log(this.firstname + ' ' +
                    this.lastname);
    }

    showAge() {
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
```

```
        const age = Math.floor(deltaTime /
                          (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' +
                    this.lastname + ' is ' + age +
                    ' years old.');
    }
}

const john = new Human('John', 'Smith', '09-17-2003');

john.showInfo();

john.showAge();
```

Ми також створили екземпляр класу Human з ім'ям john і викликали для нього 2 реалізовані в нашому класі методи showInfo() та showAge(). В результаті роботи цих методів, у консоль буде виведено таку інформацію (на момент створення уроку):

```
John Smith
John Smith is 17 years old.
```

*Рисунок 8*

Як ми бачимо на рисунку 9, клас працює подібно до функції-конструктора. Давайте тепер виведемо в консоль інформацію про саму змінну john:

```
console.log(john);
```

Що ми бачимо у консолі? У прототипі нашого об'єкта знаходиться функція-конструктор, описана в класі, і два методи також одразу винесені в прототип (рис. 10).

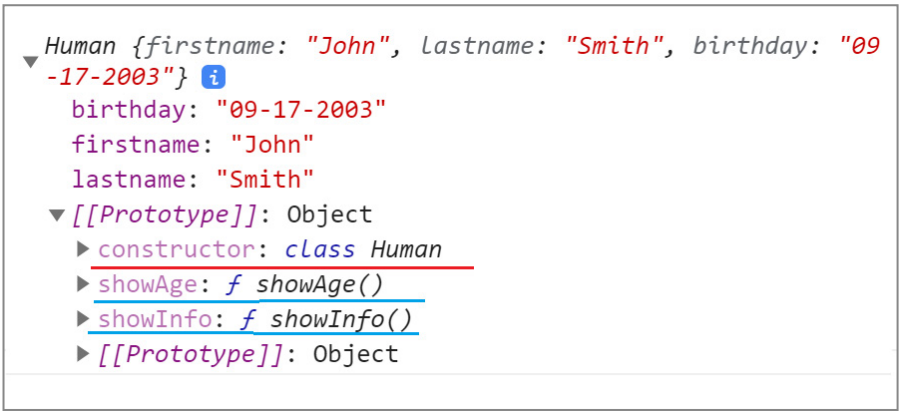


*Рисунок 9*

Давайте розглянемо ще один приклад створення класу Rectangle, який вирішуватиме споконвічну проблему школярів щодо знаходження площі та периметра прямокутника, для якого відомі його ширина та висота.

```
class Rectangle {
    constructor(width, height){
      this.width = width;
      this.height = height;
    }
    square() {
      return this.width*this.height;
    }
    perimeter(){
      return 2*(this.width+this.height);
    }
}
```

Створимо два екземпляри класу Rectangle і виведемо в консоль дані, що повертаються методами square() та perimeter().

```
let rect1 = new Rectangle(20, 30);
console.log(rect1.square(), rect1.perimeter()); //600 100
let rect2 = new Rectangle(78, 92);
console.log(rect2.square(), rect2.perimeter()); // 7176
                                                // 340
console.log(rect1, rect2);
```

І у другому класі ми знову бачимо, що методи перебувають у прототипі.

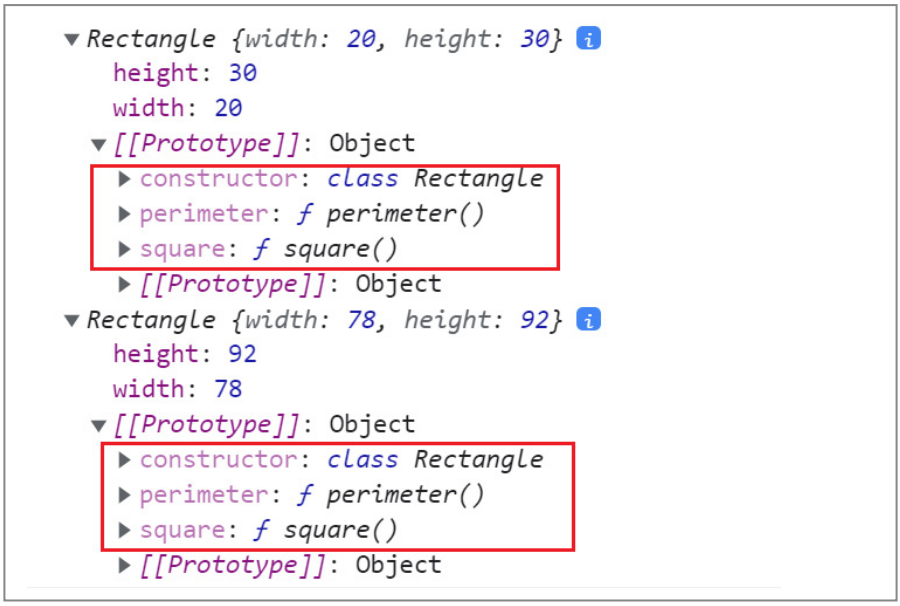


*Рисунок 10*

Можна зробити висновок, що синтаксис класу простіше, наочніше і зручніше, ніж функція-конструктор.

Ключове слово this, що використовується всередині класу, є внутрішнім «я» об'єкта. Як будь-яка людина, при запиті «Ваше ім'я» всередині, розуміє, що їй потрібно

ввести своє ім'я, а не ім'я сусіда, так і об'єкт при зверненні this.firstname розуміє, що йому необхідно взяти дані, записані при створенні об'єкта-екземпляра класу у відповідне поле. Для різних екземплярів класу, як і для різних людей, це значення буде відрізнятися, але при цьому кожен об'єкт має цілком конкретний запис цієї властивості та не зможе використовувати жодну іншу, аналогічно тому значенню, як у свідоцтві про народження зазначено те ім'я, яке батьки дали дитині після її появи на світ, а не довільне значення або те, що часто змінюється.

Що буде, якщо ми упустимо ключове слово this в синтаксисі класу, наприклад, так:

```
class Rectangle {
    constructor(width, height){
        width = width;
        height = height;
    }
    square() {
        return width*height;
    }
    perimeter(){
        return 2*(width+height);
    }
}
```

У цьому випадку ми побачимо помилку в консолі *Uncaught ReferenceError: width is not defined at Rectangle.square*, яка виникне всередині функції square при спробі використати неоголошену змінну width.

Якщо ми змінимо код і передаватимемо в функції square() і perimeter() параметри width і height, то все

одно отримаємо NaN як результат виклику цих методів, оскільки всередині класу ці параметри мають значення undefined.

```
class Rectangle {
    constructor(width, height){
        width = width;
        height = height;
    }
    square(width,height) {
        console.log(width, height);
        return width*height;
    }
    perimeter(width,height){
        return 2*(width+height);
    }
}

let rect1 = new Rectangle(20, 30);
console.log(rect1.square(), rect1.perimeter()); // NaN NaN
let rect2 = new Rectangle(78, 92);
console.log(rect2.square(), rect2.perimeter()); // NaN NaN
```

# Фундаментальні принципи ООП

У всіх мовах програмування вам доведеться зіткнутися із трьома фундаментальними принципами ООП — це успадкування, інкапсуляція та поліморфізм. Розглянемо докладніше, що це означає і як реалізовано в JavaScript, тим більше, що ми вже говорили про прототипи, як про механізм успадкування методів об'єкта в JS.

## 1. Інкапсуляція

Принцип інкапсуляції передбачає, що ми приховуємо деталі реалізації класу від доступу ззовні, роблячи змінні функції недоступними з екземплярів класу. Нагадаю, що екземпляри класу — це змінні (константи), створені на основі функції-конструктора або класу.

Всі властивості та методи, які ми створювали досі, прийнято називати публічними. Тому, ми можемо змінити їх будь-якої миті. Нагадаю, що ми створювали змінну john як екземпляр класу Human. Виведемо в консоль ті дані, які задали при ініціалізації змінної, а потім змінимо їх:

```
const john = new Human('John', 'Smith', '09-17-2003');
console.log(john.firstname, john.lastname);

john.firstname = 'Billy';
john.lastname = 'Thomas';

console.log(john.firstname, john.lastname);
```

Результат у консолі:


*Рисунок 11*

Однак, інколи виникає необхідність «сховати» від втручання будь-яку змінну або метод. Для цього є приватні змінні. Такі змінні прийнято оголошувати поза конструктором зі спеціальним символом #, а вже ініціалізувати безпосередньо в конструкторі.

**Примітка:** *на момент написання уроку ця можливість реалізована лише у браузері Google Chrome. У решті, ви можете зіткнутися з помилкою.*

Наприклад, у класі Human нам необхідна змінна #id, що вказує на ідентифікатор користувача у вигляді великого випадкового числа. Ми отримаємо її значення за допомогою методів об'єкта Math.

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
…
}

console.log(john);
```

Якщо вивести в консоль дані про об'єкт john, то ми побачимо цю змінну #id:

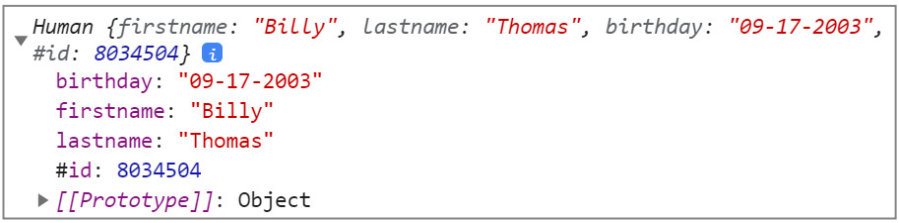


*Рисунок 12*

Однак, при спробі явного доступу до цієї змінної або втручання в неї, в консоль буде виведена помилка: *Uncaught SyntaxError: Private field '#id' must be declared in an enclosing class*

```
console.log(john.#id);
john.#id = 1;
```

Це показує, що дана змінна є приватною, тобто недоступною ззовні, але доступною лише межах класу, у якому вона оголошена. Однак, спосіб доступу до неї все ж таки є, і зробити це можна за допомогою спеціальних методів-аксесорів, які також називаються гетерами (від англ. *get*) і сетерами (від англ. *set*).

**Гетер** передбачає, що ми можемо отримати значення певної змінної, а **сетер** — що ми можемо встановити значення цієї змінної. Синтаксис у них такий:

```
get id(){
   return this.#id;
}
```

```
set id(value){
    this.#id = value;
}
```

Із запису помітно, що ключові слова get і set ми записуємо через пробіл від імені тієї властивості, яку хочемо зробити публічною для нашого класу, тобто доступною ззовні, але при цьому змінювати значення ми будемо для внутрішньої приватної (прихованої або інкапсульованої) змінної.

Тепер давайте запишемо клас Human повністю та викличемо методи-аксесори для змінної john:

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
    showInfo() {
        console.log(this.firstname + ' ' +
                    this.lastname);
    }
    showAge() {
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
        const age = Math.floor(deltaTime /
                    (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' +
                    this.lastname + ' is ' + age +
                    ' years old.');
    }
```

```
    get id(){
        return this.#id;
    }
    set id(value){
        this.#id = value;
    }
}

console.log(john.id);
john.id = 1;
console.log(john.id);
```

Зверніть увагу, що ми не викликаємо ні методу get id(), ні методу set id(). Натомість, ми отримуємо або встановлюємо шляхом привласнення значення 1 властивість id. Проте в консолі ми побачимо:

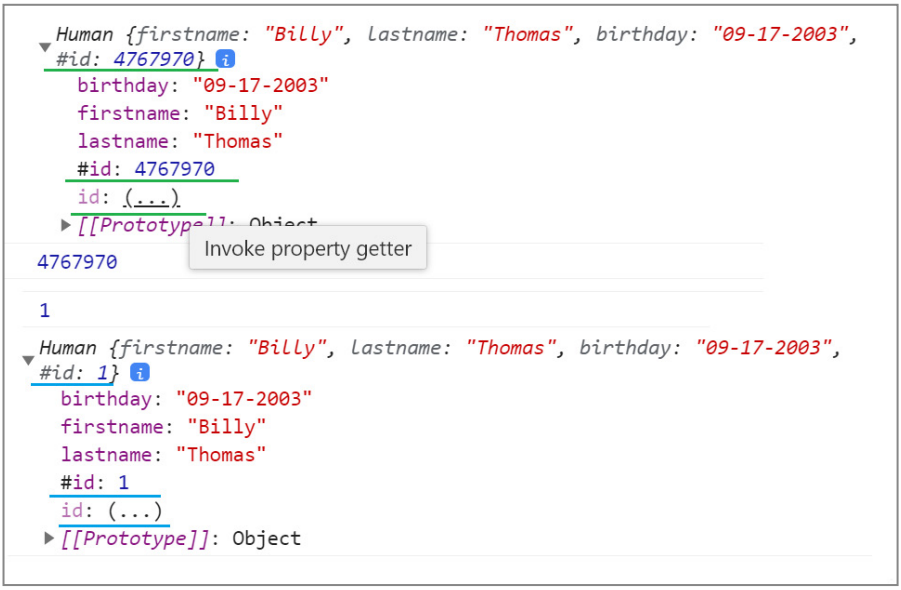


*Рисунок 13*

На рисунку 14 різними кольорами підкреслено значення приватної змінної #id до і після привласнення значення 1, а також гетер для властивості id.

## 2. Успадкування

Успадкування передбачає, що на основі одного класу можна створити інший, і в другому, дочірньому класі можна використовувати методи першого, батьківського класу. Це дуже зручно, тому що не потрібно створювати нові методи, що дублюють вже існуючі.

У прикладі з класом Human ми можемо використовувати принцип успадкування наступним чином. Ми створимо новий клас Teacher, в якому використовуватимемо ті ж дані для ініціалізації об'єкта: ім'я, прізвище, дату народження, але додамо ще масив предметів, які може викладати цей вчитель. Також, додамо ще один метод showSubjects(), який виводитиме інформацію про знання вчителя.

```
class Teacher extends Human{
    constructor(firstname, lastname, birthday,
                subjects = []){
        super(firstname, lastname, birthday );
        this.subjects = subjects;
    }

    showSubjects() {
        console.log(this.firstname +
                    ' ' +
                    this.lastname +
                    ' can teach you '+
                    this.subjects.join(', '));
```

```
    }
}

const kate = new Teacher('Kate', 'Lowdell', '07/15/1986',
                         ['biology', 'geography']);
```

Зверніть увагу на синтаксис: оголошуючи клас Teacher, ми записали ключове слово показуємо, що клас Teacher є спадкоємцем класу Human. Крім того, в конструктор ми передаємо не 3 параметри, а 4, але перші 3 з них використовуємо в ключовому слові super() для виклику конструктора батьківського класу Human.

Тепер найцікавіше. При створенні екземпляра класу Teacher з ім'ям kate ми отримуємо доступ не тільки до описаного у цьому класі методу showSubjects(), а й до всіх методів батьківського класу Human. Це можна побачити на рисунку 15, де при додаванні до імені екземпляра оператора-крапки, у текстовому редакторі з'явиться підказка у вигляді списку існуючих методів та властивостей.


*Рисунок 14*

Тобто наш екземпляр класу успадкував ці методи від іншого класу, оскільки вони пов'язані ключовим словом extends.

Якщо ми запишемо виклик всіх цих методів для об'єкта з ім'ям kate,

```
kate.showInfo();
    kate.showAge();
    kate.showSubjects();
```

то отримаємо наступні записи в консолі:

```
Kate Lowdell
Kate Lowdell is 35 years old.
Kate Lowdell can teach you biology, geography
```

*Рисунок 15*

Ускладнемо завдання і створимо ще один клас ITMentor, який тепер розширюватиме клас Teacher за тим самим принципом, використовуючи ключове слово extends:

```
class ITMentor extends Teacher{
    constructor(firstname, lastname, birthday,
                subjects = [], level){
        super(firstname, lastname, birthday, subjects);
        this.level = level;
    }

    showLevel(){
        console.log(this.firstname + ' ' +
                    this.lastname + ' has level '+
                    this.level);
    }
}
```

```
const andrew = new ITMentor("Andrew", "Phillipov",
                            '07/22/1986',
                            ['HTML', 'CSS', 'JavaScript',
                            'React', 'Angular'],
                            'Senior');
    andrew.showInfo();
    andrew.showAge();
    andrew.showSubjects();
    andrew.showLevel();
```

У цьому класі додалася ще одна властивість level, яка відповідає за рівень знань ITMentor та метод showLevel(), що виводить у консоль інформацію про нього. Викликаючи метод super() у конструкторі цього класу, ми звертаємося до класу Teacher, передаючи в нього необхідні параметри.

При створенні змінної-екземпляра класу ITMentor з ім'ям andrew ми автоматично отримуємо доступ не тільки до властивостей та методів цього класу, але й до властивостей та методів класів Teacher та Human. Якщо розкрити всі стрілочки, які з'являються при виведенні в консоль змінної andrew, то ми побачимо весь ланцюг прототипів, який показує нам, до якого класу належать методи.

З успадкуванням працює такий підхід — коли ми викликаємо будь-який метод або використовуємо якусь властивість, то спочатку рушій JavaScript буде шукати його в основному класі об'єкта, потім підніметься до батьківського класу, після того ще вище ланцюгом батьківських об'єктів аж до класу Object, поки не знайде відповідний метод або властивість. Це дуже зручно, хоча і передбачає, що ви повинні продумати, в якому об'єкті вони мають бути описані.


*Рисунок 16*

## 3. Поліморфізм

Поліморфізм можна перекласти як «безліч форм». Цей принцип допомагає проєктувати об'єкти таким чином, щоб вони могли спільно використовувати або перевизначати будь-яку поведінку. Найчастіше для перевизначення використовують методи батьківського класу. Звідси можна дійти висновку, що поліморфізм використовує успадкування.

Суть поліморфізму в ООП полягає в тому, що ви можете викликати той самий метод для різних об'єктів, але при цьому для кожного об'єкта цей метод спра-

цює по-своєму. Розглянемо це на прикладі методу showSubjects(), який з'явився у класі Teacher, але змінимо ми його у класі ITMentor.

У коді нижче наведено весь клас ITMentor зі зміненим методом showSubjects() і продубльовано створення константи andrew, а також знову викликаний метод showSubjects(), успадкований від класу Teacher.

```
class ITMentor extends Teacher {
    constructor(firstname, lastname, birthday,
                subjects = [], level){
        super(firstname, lastname, birthday, subjects);
        this.level = level;
    }

    showSubjects() {
        console.log('With '+this.firstname + ' ' +
                    this.lastname +
                    ' you can get such IT skills: '+
                    this.subjects.join(', '));

        document.write('<p>With '+this.firstname +
                    ' ' + this.lastname +
                    ' you can get such IT skills:
                        </p><ol><li>' +
                    this.subjects.join('<li>') +
                    '</ol>');
    }

    showLevel(){
        console.log(this.firstname + ' ' +
                    this.lastname + ' has level '+
                    this.level);
    }
}
```

```
const andrew = new ITMentor("Andrew", "Phillipov",
                            '07/22/1986', ['HTML',
                            'CSS', 'JavaScript',
                            'React', 'Angular'],
                            'Senior');
andrew.showSubjects();
```

Однак, тепер виведення інформації за допомогою showSubjects() змінилося — ми бачимо в консолі інший текст + текст із розміткою, який виведений у тіло документа (<body>) у вигляді абзацу та пронумерованого списку.

**console.log()**

```
With Andrew Phillipov you can get such IT skills: HTML,
CSS, JavaScript, React, Angular
```

**document.write()**

With Andrew Phillipov you can get such IT skills:

1. HTML
2. CSS
3. JavaScript
4. React
5. Angular

*Рисунок 17*

Це сталося внаслідок додавання в showSubjects() метода document.write(), який дозволяє вивести в тіло HTML-сторінки, з якою пов'язаний наш скрипт, будь-який текст, оформлений у теги за правилами HTML-розмітки.

Якщо ж ми викличемо метод для раніше оголошеної константи kate — екземпляра класу Teacher, то побачимо

текст, який формується методом showSubjects() для класу Teacher без будь-яких змін.

```
Kate Lowdell can teach you biology, geography
```

*Рисунок 18*

Це говорить про те, що JavaScript в першу чергу шукає властивість або метод у тому класі, екземпляром якого є створена змінна, а вже потім звертається по ланцюгу до кожного з батьківських класів, щоб знайти метод, викликаний для об'єкта, якщо він не знайдений для того класу, до якого належить цей об'єкт.

Давайте для демонстрації цього прийому змінимо в класі Human метод, який успадковується від головного JavaScript класу об'єктів Object і називається toString(). Цей метод відповідає за те, як буде виведено у рядковому вигляді екземпляр будь-якого об'єкта. Спочатку розглянемо, як будуть відображатися всі наші константи різних класів у тілі документа, якщо помістити їх у метод document.write():

```
document.write('Class Human: '+john + '<br>');
document.write('Class Teacher: '+kate + '<br>');
document.write('Class ITMentor: '+andrew + '<br>');
```

Результат на екрані:

```
Class Human: [object Object]
Class Teacher: [object Object]
Class ITMentor: [object Object]
```

Не дуже інформативно, чи не так? Тому додаємо в клас Human метод toString(), в якому виведемо ім'я та прізвище з властивостей кожного об'єкта, а також назву класу за допомогою властивості this.constructor.name:

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
    showInfo() {
        console.log(this.firstname + ' ' +
                    this.lastname);
    }
    showAge() {
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
        const age = Math.floor(deltaTime /
                    (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' +
                    this.lastname + ' is ' + age +
                    ' years old.');
    }
    get id(){
        return this.#id;
    }
    set id(value){
        this.#id = value;
    }
    toString(){
        return this.firstname + ' ' + this.lastname +
               ' is a '+ this.constructor.name;
    }
}
```

Тепер виклик методу document.write() дасть зовсім інший результат:

*Class Human: Billy Thomas is a Human*
*Class Teacher: Kate Lowdell is a Teacher*
*Class ITMentor: Andrew Phillipov is a ITMentor*

Зверніть увагу на те, що метод, доданий до батьківського класу Human, спрацював для всіх дочірніх класів цілком коректно.

## Оператор instanseof

Щоб перевірити, до якого класу належить об'єкт, створений за допомогою змінної, використовується оператор instanseof. Його синтаксис:

```
variableName instanseof Class
```

Цей оператор повертає true або false залежно від того, чи належить наш об'єкт до зазначеного класу. У коді нижче помітно, що екземпляри класу Teacher та ITMentor також є екземплярами батьківського класу Human, тоді як екземпляр класу Human не є екземпляром Teacher або ITMentor, як і екземпляром ніяк не пов'язаного з ним класу Array.

```
console.log(john instanceof Human); //true
console.log(kate instanceof Teacher); //true
console.log(kate instanceof Human); //true
console.log(andrew instanceof ITMentor); //true
console.log(andrew instanceof Teacher); //true
console.log(andrew instanceof Human); //true
console.log(john instanceof Teacher); //false
console.log(john instanceof ITMentor); //false
console.log(john instanceof Array); //false
```

# Розширення стандартних класів

На закінчення теми про принципи ООП хотілося б зупинитися на успадкуванні стандартних або вбудованих класів, описаних в ядрі JavaScript, на прикладі класу String. Ми створимо клас StringInfo і додамо до нього єдиний метод, який буде підраховувати кількість входжень будь-якої літери або слова в певний рядок. У цьому випадку ми можемо обійтися без виклику методу constructor:

```
class StringInfo extends String {
    calcLetter(letter) {
        let count = 0;
        let index = this.indexOf(letter);
        while(index != -1){
            count++;
            index = this.indexOf(letter,index+1);
        }
        return count;
    }
}

let myStr = new StringInfo("When the going gets tough,
                            the tough get going.");

console.log('g in "'+myStr +'" = '+
            myStr.calcLetter('g'));

console.log('going in "'+myStr +'" = '+
            myStr.calcLetter('going'));

console.log('"text" in "'+myStr +'" = '+
            myStr.calcLetter('text'));
```

У консолі ми побачимо таку інформацію:

```
"g" in "When the going gets tough, the tough get
        going." = 8
"going" in "When the going gets tough, the tough get
        going." = 2
"text" in "When the going gets tough, the tough get
        going." = 0
```

У коді методу, в дещо зміненому вигляді, ми використовували скрипт, який ви вже зустрічали в тексті уроку.

Ключове слово this у цьому класі вказує на той рядок, який був переданий у дужках при створенні екземпляра.

Майте на увазі, що не до всіх рядків можна застосувати метод calcLetter(). Якщо ми оголосимо звичайну рядкову змінну, яка буде екземпляром об'єкта String, а не StringInfo(), то виклик методу calcLetter() призведе до помилки:

```
let strHow = 'How do you do?';
console.log('"do" in "'+ strHow +'" = '+
            strHow.calcLetter('do'));
```

Помилка буде у вигляді повідомлення, що strHow.calcLetter не є функцією, оскільки в об'єкті String, до якого належить змінна strHow, такий метод не передбачений: *Uncaught TypeError: strHow.calcLetter is not a function*.

Тому потрібно розрізняти екземпляри об'єктів на основі створених вами класів та екземпляри батьківських класів, щоб не було плутанини. Класи, які розширюють стандартні або користувальницькі класи, успадковують властивості та методи своїх батьківських класів, але

класи-батьки не мають властивостей та методів дочірніх класів аналогічно тому, що ваша бабуся або прабабуся навряд чи змогли б користуватися сучасним комп'ютером або смартфоном, тому що в їх час такі девайси ще не були створені.

# Домашнє завдання

## Завдання 1

Створіть клас MyButton, який приймає 2 параметри у вигляді тексту (text) та класу кнопки (btnClass). Опишіть метод show(), який виводить методом document.write() екземпляр кнопки в тіло HTML-сторінки. Передбачте гетер і сетер, які дозволяють отримати та змінити властивість value кнопки, яка насправді змінює її властивість-text.

Опишіть у стилях кілька класів для сторінки, які дозволять створити різні екземпляри кнопок.

Виведіть кілька кнопок шляхом show(), для однієї з них змініть текст:


*Рисунок 19*

Створіть клас ColorButton, який успадковує клас MyButton, додавши до нього додатковий клас, який дозволяє змінювати колір фону та текст кнопки, додаючи до екземпляра ColorButton крім основного ще й додатковий клас. Наприклад, екземпляр ColorButton буде викликатись за такими параметрами:

```
let btnColor = new ColorButton("See more", "btn",
                                "btn-danger");
```

Кнопка, виведена за допомогою методу show(), матиме такий код:

```
<button type="button" class="btn btn-danger"> See more
</button>
```


# Урок 2-2.
# Введення в ООП