**Практична №4**
**Об’єкти**

***Завдання0:*** Опрацювати теоретичні відомості :
Пункти 4.1 - 4.6 за посиланням:
**https://learn.javascript.ru**

***Завдання1:*** Напишіть код, виконайте завдання кожного пункту окремим рядком:

1. Створіть пустий об’єкт **programmer**.
2. Додайте властивість **name** зі значенням Вашого імені .
3. Додайте властивість зі значенням свого прізвище.
4. Змініть значення властивості **name** на інше значення.
5. Видаліть властивість **name** з об’єкта.

***Завдання2:*** Напишіть функцію **isProperty (obj)**, яка вертає **true**, якщо у об'єкта немає властивостей, інакше **false**.

*Наприклад:*

```
let obj = {};
alert(isProperty (obj) ); // true
obj["la-la"] = "wow";
alert(isProperty (obj) ); // false
```

***Завдання3:***Створити об’єкт з оцінками за екзамен з розробки веб-застосувань,

*Наприклад*:

```
let ekzamen = {
  Vova: 5,
  Serg: 4,
  Sasha: 3
}
```

Напишіть код для знаходження середнього балу і збережіть результат в змінній serball. Якщо об'єкт пустий, то результат повинен бути 0.

***Завдання4:*** Створіть функцію **increasesNum(obj)**, яка збільшує всі числові властивості об'єкта **obj** на число, яке дорівнює Вашому дню народження.

*Наприклад*:

```
let clock = {
  hour: 14,
  minute: 52,
  title: "My clock"
};

increasesNum (menu);

// після виклику функції:
menu = {
  width: 20,
  height: 58,
  title: "My clock"
};
```

***Завдання5:***Створіть об'єкт калькулятор з трьома методами:
read () запитує два значення і зберігає їх як властивості об'єкта.
sum ()вертає сумму збережених значень.
mul ()перемножує збережені значения і вертає результат.

*Наприклад:*

```
let calculator = {
  // ... ваш код ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

***Методичні рекомендації:***

**Об’єкти** – це структури даних, які володіють станом (набір властивостей) та поведінкою (набір функцій – методів).

*1) Створення об’єкта:*

```
var car = {
  name: "Chevy",
  model:"Bel Air",
  year: 1957,
  color: "red",
  passengers: 2,
  convertible: false,
  mileage: 1021
}

console.log(car);
```

```
▶ {name: "Chevy", model: "Bel Air", year: 1957, color: "red", passengers: 2, …}
```

*2) Властивості можна додавати і видаляти:*

```
var car = {
  name: "Chevy",
  model:"Bel Air",
  year: 1957,
  mileage: 1021
}

delete car.mileage;
car.price = 10000;
console.log(car);
```

```
▶ {name: "Chevy", model: "Bel Air", year: 1957, price: 10000}
```

*3) Об’єкт з методом:*

```
var person = {
  firstName: "John",
  lastName: "Smith",
  age: 32,
  //Метод об'єкту - функція, визначена всередині об'єкта
  getFullName: function() {
    return `${this.firstName} ${this.lastName}`;
  }
};


console.log(`Hello, ${person.getFullName()}!`);
```

Hello, John Smith!

## *4) Перебір властивостей об'єкта в циклі*

```
for (property in person) {
  console.log(`${property}: ${person[property]};`)
}
```

```
Hello, John Smith!
firstName: John;
lastName: Smith;
age: 32;
getFullName: function() {
      return `${this.firstName} ${this.lastName}`;
    };
```

## *5) Передача об'єкта в функцію здійснюється за посиланням на об'єкт:*

```
//Об'єкт в функцію передається по посиланню
//Тобто параметр є вказівником на сам об'єкт
function loseWeight(dog, amount) {
  dog.weight -= amount;
}

var fido = {
  name: "Fido",
  weight: 40
}

loseWeight(fido, 5);
loseWeight(fido, 6);

console.log(fido);
```

## *6) Масиви об'єктів:*

```
var users = [
  {name: "John", age: 25},
  {name: "Alex", age: 30},
  {name: "Mary", age: 19}
];

//Вивести всі елементи в окремих рядках в форматі name: age
```

## *7) Конструктори об'єктів:*

**Конструктор** – функція, що повертає об'єкт та дозволяє створювати об'єкти з однаковим набором властивостей та методів:

```
function Dog(name, breed, weight) {
  this.name = name;
  this.breed = breed;
  this.weight = weight;
}

var fido = new Dog("Fido", "Mixed", 38);
console.log(fido);
```

```
▼ Dog {i}
    breed: "Mixed"
    name: "Fido"
    weight: 38
  ▶ __proto__: Object
```

## *8) Методи в конструкторах об'єктів:*

```
function Dog(name, breed, weight) {
  this.name = name;
  this.weight = weight;
  this.bark = function() {
    if (this.weight > 25) {
      console.log(this.name + " says Woof!");
    } else {
      console.log(this.name + " says Yip!");
    }
  }
}

var fido = new Dog("Fido", "Mixed", 38);
var fluffy = new Dog("Fluffy", "Poodle", 30);
var spot = new Dog("Spot", "Chihuahua", 10);

var dogs = [fido, fluffy, spot];

for (var i = 0; i < dogs.length; i++) {
  dogs[i].bark();
}
```

```
Fido says Woof!
Fluffy says Woof!
Spot says Yip!
```

## *9) Передача аргументів в конструктор як об'єктний літерал:*

```
function Car(params) {
  this.make = params.make;
  this.model = params.model;
  this.year = params.year;
}

var cadiParams = {
  make:"GM",
  model: "Cadillac",
  year: 1955
}

var cadi = new Car(cadiParams);
console.log(cadi);
```

*10) Екземпляри об'єктів:*

```
if (cadi instanceof Car) {
  console.log("Cadi is a Car");
}
```

**Методы объекта, "this"**

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

```
// Объект пользователя
let user = {
  name: "Джон",
  age: 30
};
```

И так же, как и в реальном мире, пользователь может *совершать действия*: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

**Примеры методов**

Для начала давайте научим нашего пользователя user здороваться:

```
let user = {
  name: "Джон",
  age: 30
};

user.sayHi = function() {
  alert("Привет!");
};

user.sayHi(); // Привет!
```

Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили её свойству user.sayHi нашего объекта.

Затем мы вызвали её. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют *методом* этого объекта.

Итак, мы получили метод sayHi объекта user.

Конечно, мы могли бы заранее объявить функцию и использовать её в качестве метода, примерно так:

```
let user = {
  // ...
};

// сначала объявляем
function sayHi() {
  alert("Привет!");
};

// затем добавляем в качестве метода
user.sayHi = sayHi;

user.sayHi(); // Привет!
```

**Объектно-ориентированное программирование**
Когда мы пишем наш код, используя объекты для представления сущностей реального мира, – это называется объектно-ориентированное программирование или сокращённо: «ООП».

ООП является большой предметной областью и интересной наукой само по себе. Как выбрать правильные сущности? Как организовать взаимодействие между ними? Это – создание архитектуры, и есть хорошие книги по этой теме, такие как «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» авторов Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес или «Объектно-ориентированный анализ и проектирование с примерами приложений» Гради Буча, а также ещё множество других книг.

## Сокращённая запись метода

Существует более короткий синтаксис для методов в литерале объекта:

```
// эти объекты делают одно и то же (одинаковые методы)

user = {
  sayHi: function() {
    alert("Привет");
  }
};
```

```
// сокращённая запись выглядит лучше, не так ли?
user = {
  sayHi() { // то же самое, что и "sayHi: function()"
    alert("Привет");
  }
};
```

Как было показано, мы можем пропустить ключевое слово "function" и просто написать sayHi().

Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов (что будет рассмотрено позже), но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

## Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user.

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.**

Значение this – это объект «перед точкой», который использовался для вызова метода.

Например:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {
    // this - это "текущий объект"
    alert(this.name);
  }

};

user.sayHi(); // Джон
```

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

```
let user = {
  name: "Джон",
```

```
  age: 30,

  sayHi() {
    alert(user.name); // используем переменную "user" вместо ключевого слова "this"
  }

};
```
…Но такой код будет ненадёжным. Если мы решим скопировать ссылку на объект user в другую переменную, например, admin = user, и перезапишем переменную user чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из admin.

Это показано ниже:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {
    alert( user.name ); // приведёт к ошибке
  }

};

let admin = user;
user = null; // обнулим переменную для наглядности, теперь она не хранит ссылку на объект.

admin.sayHi(); // Ошибка! Внутри sayHi() используется user, которая больше не ссылается на объект!
```
Если мы используем this.name вместо user.name внутри alert, тогда этот код будет работать.

## «this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции.

В этом коде нет синтаксической ошибки:

```
        function sayHi() {
  alert( this.name );
}
```
Значение this вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

```
let user = { name: "Джон" };
let admin = { name: "Админ" };
```

```
function sayHi() {
  alert( this.name );
}

// используем одну и ту же функцию в двух объектах
user.f = sayHi;
admin.f = sayHi;

// вызовы функции, приведённые ниже, имеют разное значение this
// "this" внутри функции является ссылкой на объект, который указан "перед точкой"
user.f(); // Джон  (this == user)
admin.f(); // Админ  (this == admin)

admin['f'](); // Админ (неважен способ доступа к методу - через точку или квадратные скобки)
```

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

**Вызов без объекта: this == undefined**

Мы даже можем вызвать функцию вовсе без использования объекта:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

В строгом режиме ("use strict") в таком коде значением this будет являться undefined. Если мы попытаемся получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет *глобальный объект* (window для браузера, мы вернёмся к этому позже в главе Глобальный объект). Это – исторически сложившееся поведение this, которое исправляется использованием строгого режима ("use strict").

Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

**Последствия свободного this**

Если вы до этого изучали другие языки программирования, тогда вы, скорее всего, привыкли к идее "фиксированного this" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод).

В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Эта идея вычисления this в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у

различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Здесь мы не будем судить о том, является ли это решение в языке хорошим или плохим. Мы должны понимать, как с этим работать, чтобы получать выгоды и избегать проблем.

## Внутренняя реализация: Ссылочный тип

**Продвинутая возможность языка**
Этот раздел объясняет сложную тему, чтобы лучше понимать некоторые запутанные случаи.

Если вы хотите продвигаться быстрее, его можно пропустить или отложить.
Некоторые хитрые способы вызова метода приводят к потере значения this, например:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); },
  bye() { alert("Пока"); }
};

user.hi(); // Джон (простой вызов метода работает хорошо)

// теперь давайте попробуем вызывать user.hi или user.bye
// в зависимости от имени пользователя user.name
(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!
```

В последней строчке кода используется условный оператор ?, который определяет, какой будет вызван метод (user.hi или user.bye) в зависимости от выполнения условия. В данном случае будет выбран user.hi.

Затем метод тут же вызывается с помощью скобок (). Но вызов не работает как положено!

Вы можете видеть, что при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим).

Так работает (доступ к методу объекта через точку):

```
user.hi();
```

Так уже не работает (вызываемый метод вычисляется):

```
(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!
```

Почему? Если мы хотим понять, почему так происходит, давайте разберёмся (заглянем под капот), как работает вызов методов (obj.method()).

Присмотревшись поближе, в выражении obj.method() можно заметить две операции:

1. Сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method).
2. Затем скобки () вызывают этот метод (исполняется код метода).

Итак, каким же образом информация о this передаётся из первой части во вторую?

Если мы поместим эти операции в отдельные строки, то значение this, естественно, будет потеряно:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); }
}

// разделим получение метода объекта и его вызов в разных строках
let hi = user.hi;
hi(); // Ошибка, потому что значением this является undefined
```

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

**Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого Reference Type.**

Этот ссылочный тип (Reference Type) является внутренним типом. Мы не можем явно использовать его, но он используется внутри языка.

Значение ссылочного типа – это «триплет»: комбинация из трёх значений (base, name, strict), где:

- base – это объект.
- name – это имя свойства объекта.
- strict – это режим исполнения. Является true, если действует строгий режим (use strict).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

```
// значение ссылочного типа (Reference Type)
(user, "hi", true)
```

Когда скобки () применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки . до вызывающих скобок ().

При любой другой операции, например, присваивании hi = user.hi, ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без this.

Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое). Позднее в этом учебнике мы изучим различные варианты решения проблемы потери значения this. Например, такие как func.bind().

## У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы используем this внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

```
let user = {
  firstName: "Илья",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Илья
```

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение this, а хотим брать его из внешнего контекста.