**Классы** обеспечивают объектно-ориентированное программирование — одну из наиболее важных концепций разработки. Данный подход значительно снизил сложность приложений и повысил удобство их обслуживания.

**Инкапсуляция (от лат. in capsule — в оболочке)** — это заключение данных и функциональности в оболочку. В объектно-ориентированном программировании в роли оболочки выступают классы: они не только собирают переменные и методы в одном месте, но и защищают их от вмешательства извне (сокрытие).

**Методы позволяют** контролировать обращение к данными и предотвратить их удаление или некорректное изменение. Например, можно запретить присваивать полю «возраст» объекта «Пользователь» число большее 130. Другими словами, это такая «защита от дурака» в программировании.

Классы — это на самом деле просто особая функция.

**Под классом в JavaScript** мы будем понимать шаблон, на основании которого создаются объекты.

ООП — очень важная парадигма программирования, в которой код структурируется в виде объектов, что приводит к более удобному обслуживанию и многократному использованию кода.

**Резюме**

Концепция ООП, позволяет структурировать код таким образом, чтобы объекты были основными участниками логики.

Классы — это оболочки объектов. Мы можем легко задать шаблон для объекта и создать экземпляр, используя ключевое слово new.

**Как создать класс:**

Используем ключевое слово class, чтобы сказать JavaScript, что хотим создать класс. Далее мы присваиваем имя классу. По общепринятым правилам имена классов начинаются с заглавной буквы.

**Классы в JavaScript** инкапсулируют данные и функции, которые являются частью этого класса. Если вы создадите класс, с его помощью вы сможете позже работать над объектами, используя следующий синтаксис:

**class ClassName** {

constructor(prop1, prop2) {

this.prop1 = prop1;

this.prop2 = prop2;

}

}

let obj = new ClassName("arg1", "arg2");

obj.prop1 // “arg1”

obj.prop2 // “arg2”

**Описание:**

Код определяет класс с именем ClassName, объявляет переменную obj и делает ее новым экземпляром объекта. Приводятся два аргумента, они будут использоваться конструктором для инициализации свойств.

**Пример:**

Пример с собакой также можно реализовать с использованием синтаксиса класса.

class Dog {

constructor(dogName, weight, color, breed) {

this.dogName = dogName;

this.weight = weight;

this.color = color;

this.breed = breed;

}

}

let dog = new Dog("JavaScript", 2.4, "brown", "chihuahua");

**Вывод в консоль свойства:**

console.log(dog.dogName, "is a", dog.breed, "and weighs", dog.weight, "kg.");

**Результат:**

JavaScript is a chihuahua and weighs 2.4 kg.

**Классы, по сути, являются схемами для объектов**. Если нужно создать 20 записей собак, придется писать гораздо меньше кода, когда у нас есть класс dog.

**Рассмотрим различные элементы классов.**

**Метод constructor** используется для инициализации объектов с помощью классов. В классе может быть только один метод constructor. Он содержит свойства, которые будут установлены при инициализации класса.

class Person {

constructor(firstname, lastname) {

this.firstname = firstname;

this.lastname = lastname;

}

}

Создание нового объекта на основе класса Person:

let p = new Person("Maaike", "van Putten");

Именно слово new дает понять JavaScript, что требуется искать специальную функцию конструктора в классе Person и создавать новый объект. Конструктор вызывается и возвращает экземпляр объекта person с заданными свойствами. Этот объект сохраняется в переменной p.

**В методе constructor вы можете задать значения по умолчанию.**

сonstructor(firstname, lastname = "Doe") {

this.firstname = firstname;

this.lastname = lastname;

}

**Методы**

Функции:

В классе можно указывать функции. Благодаря этому наш объект может начать выполнять действия, используя собственные свойства, — например, выводить имя на экран. Функции в классе называются методами. Для определения таких методов не используется ключевое слово function. Мы сразу начинаем писать имя:

class Person {

constructor(firstname, lastname) {

this.firstname = firstname;

this.lastname = lastname;

}

greet() {

console.log("Hi there! I'm", this.firstname);

}

}

**Вызвать метод greet** для объекта Person можно следующим образом:

let p = new Person("Maaike", "van Putten");

**p.greet(); // Hi there! I'm Maaike**

**Методы, как и функции, могут принимать параметры и возвращать результаты:**

class Person {

constructor(firstname, lastname) {

this.firstname = firstname;

this.lastname = lastname;

}

greet() {

console.log("Hi there!");

}

compliment(name, object) {

return "That's a wonderful " + object + ", " + name;

}

}

let compliment = p.compliment("Harry", "hat");

**console.log(compliment) // That's a wonderful hat, Harry**

**Свойства**

Свойства (иногда называемые полями) содержат данные класса.

Прямой доступ к свойствам лучше не предоставлять. Мы хотим, чтобы класс контролировал значения свойств, по нескольким причинам — например, нам необходимо убедиться, что свойство имеет требуемое значение. Допустим, вы хотите подтвердить, что возраст пользователя больше 18 лет. Можно достичь этого, сделав невозможным прямой доступ к свойству за пределами класса.

Вот как добавить свойства, недоступные извне. Ставим перед ними символ #:

class Person {

#firstname;

#lastname;

constructor(firstname, lastname) {

this.#firstname = firstname;

this.#lastname = lastname;

}

}

В данный момент к свойствам firstname и lastname доступа извне нет. Если попробуем ввести:

let p = new Person("Maria", "Saga");

console.log(p.firstname);

то получим: **undefined**

Это очень простой пример проверки. На данный момент мы вообще не можем получить доступ к свойству извне класса после конструктора. Оно доступно только изнутри класса. **Вот тут-то и вступают в игру геттеры и сеттеры.**

**Геттеры и сеттеры**

Геттеры и сеттеры — это особые свойства, которые можно использовать для получения данных из класса и записи полей данных в классе.  
Мы называем их аксессорами (accessors). Геттеры и сеттеры выглядят как функции, потому что после них ставят скобки (()), но функциями они не являются!

**Геттер** используется для получения свойства. Поэтому он не берет никаких параметров, а просто возвращает свойство.

**Сеттер** действует наоборот: принимает параметр, присваивает новое значение свойству и ничего не возвращает. **Сеттер** может включать в себя больше логики, например проводить какую-то проверку, как мы увидим ниже.

class Person {

#firstname;

#lastname;

constructor(firstname, lastname) {

this.#firstname = firstname;

this.#lastname = lastname;

}

get firstname() {

return this.#firstname;

}

set firstname(firstname) {

this.#firstname = firstname;

}

get lastname() {

return this.#lastname;

}

set lastname(lastname) {

this.#lastname = lastname;

}

}

**Геттер** можно использовать вне объекта, как если бы он был свойством. Свойства больше не доступны напрямую извне класса, но к ним можно получить доступ через геттер, возвращающий значение, и через сеттер, присваива­ющий значение. Вот как можно это реализовать вне экземпляра класса:

let p = new Person("Maria", "Saga");

**console.log(p.firstname); // Maria**

Мы создали новый объект Person с именем Maria и фамилией Saga. На выходе ото­бражается имя, что стало возможным только благодаря наличию у нас средства до­ступа — геттера. Мы также можем задать значение чему-то другому, используя сет­тер. Давайте обновим значение имени так, что теперь это будет не Maria, а Adnane:

**p.firstname = "Adnane";**

На данный момент в сеттере не происходит ничего особенного. Мы могли бы выполнить такую же проверку, как и в предыдущем конструкторе, например:

set firstname(firstname) {

if(firstname.startsWith("M")){

this.#firstname = firstname;

} else {

this.#firstname = "M" + firstname;

}

}

Теперь код проверит, начинается ли firstname с буквы M. Если да, то значение firstname обновится до значения параметра. Если нет, код добавит M перед параметром.

**Наследование** — одна из ключевых концепций ООП. Согласно ей классы могут иметь дочерние классы, которые наследуют свойства и методы родительского класса.

**Например,** создаем класс с названием Vehicle, в котором укажете некоторые общие свойства и методы объектов. Далее создаем дочерние классы, основанные на классе Vehicle, например boat, car, bicycle и motorcycle.

**class Vehicle** {

constructor(color, currentSpeed, maxSpeed) {

this.color = color;

this.currentSpeed = currentSpeed;

this.maxSpeed = maxSpeed;

}

move() {

console.log("moving at", this.currentSpeed);

}

accelerate(amount) {

this.currentSpeed += amount;

}

}

**class Motorcycle extends Vehicle** {

constructor(color, currentSpeed, maxSpeed, fuel) {

super(color, currentSpeed, maxSpeed);

this.fuel = fuel;

}

doWheelie() {

console.log("Driving on one wheel!");

}

}

С помощью **ключевого слова extends** мы указываем, что определенный класс является дочерним по отношению к другому. В данном случае Motorcycle — дочерний класс Vehicle. Это значит, что у нас будет доступ к свойствам и методам класса Vehicle в классе Motorcycle.

**Слово super** в конструкторе вызывает конструктор родителя — в данном случае конструктор Vehicle. Это гарантирует, что поля от родителя также заданы и что методы доступны без необходимости делать что-либо еще: они наследуются авто­матически. Вызов super() не является обязательным, но вы должны использовать это ключевое слово при нахождении в классе, который наследуется от другого класса — иначе получите ReferenceError.

В классе Motorcycle у нас есть доступ к полям класса Vehicle, поэтому следующая конструкция будет работать:

let motor = new Motorcycle("Black", 0, 250, "gasoline");

console.log(motor.color);

motor.accelerate(50);

motor.move();

**В результате вы увидите:**

Black

moving at 50

**Если в родительском классе есть геттеры и сеттеры**, они также наследуются дочер­ним классом. Следовательно, мы могли бы влиять на то, какие свойства могут быть извлечены и изменены (и как) за пределами нашего класса. Обычно это хорошая практика.

**Прототипы**

— это механизмы JavaScript, которые делают возможным существова­ние объектов.

Если при создании класса никакие детали специально не уточняются, объекты на­следуются от прототипа Object.prototype. Его (класс) можно считать базовым объектом, который всегда находится на вершине дерева наследования, а значит, всегда присутствует в наших объектах.

Для всех классов доступно свойство prototype, и оно всегда называется «прото­тип». Вызвать его можно так:

**ClassName.prototype**

Давайте приведем пример того, как добавить функцию в класс, используя свойство prototype. Для этого мы будем использовать класс Person:

class Person {

constructor(firstname, lastname) {

this.firstname = firstname;

this.lastname = lastname;

}

greet() {

console.log("Hi there!");

}

}

И вот как добавить функцию в этот класс, используя prototype:

**Person.prototype.introduce** = function () {

console.log("Hi, I'm", this.firstname);

};

**Свойство prototype** содержит в себе все методы и свойства объекта. Таким образом, добавление функции в prototype — это добавление функции в класс. Можно использовать prototype, чтобы добавлять свойства или методы объекту (как мы сделали выше с помощью функции introduce). Данный способ применим и для свойств:

**Person.prototype.favoriteColor = "green";**

После их можно вызывать из экземпляра класса Person:

let p = new Person("Maria", "Saga");

console.log(p.favoriteColor);

p.introduce();

**Мы получим следующий результат:**

green

Hi, I'm Maria

**Следовательно, методы и свойства, определенные через prototype, точно такие же, как если бы они были определены в классе.**

Прототипы не следует применять, если у вас есть контроль над кодом класса, и вы хотите изменить его навсегда, — в таком случае просто измените класс. Однако с помощью прототипов вы можете расширить существующие объекты, в том числе с применением условий. Важно также знать, что встроенные объекты JavaScript имеют прототипы и наследуют от Object.prototype. Не меняйте данный прототип, иначе это повлияет на работу JavaScript.