**Классы**

Базовый синтаксис для классов выглядит так:

class MyClass {

prop = value; // свойство

constructor(...) { // конструктор

// ...

}

method(...) {} // метод

get something(...) {} // геттер

set something(...) {} // сеттер

[Symbol.iterator]() {} // метод с вычисляемым именем (здесь - символом)

// ...

}

Затем использовать вызов new MyClass(). При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

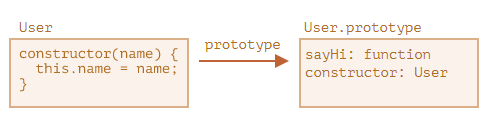
let user = new User("Иван");

user.sayHi();

Вот что на самом деле делает конструкция class User {...}:

* Создаёт функцию с именем User, которая становится результатом объявления класса. Код функции берётся из метода constructor (она будет пустой, если такого метода нет).
* Сохраняет все методы, такие как sayHi, в User.prototype.

При вызове метода объекта new User он будет взят из прототипа. Таким образом, объекты new User имеют доступ к методам класса. На картинке показан результат объявления class User:



То же самое без класса:

// перепишем класс User на чистых функциях

// 1. Создаём функцию constructor

function User(name) {

this.name = name;

}

// каждый прототип функции имеет свойство constructor по умолчанию,

// поэтому нам нет необходимости его создавать

// 2. Добавляем метод в прототип

User.prototype.sayHi = function() {

alert(this.name);

};

// Использование:

let user = new User("Иван");

user.sayHi();

Результат этого кода очень похож. Однако есть важные отличия.

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Пример Class Expression (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

Аналогично Named Function Expression, Class Expression может иметь имя, которое видно только внутри класса.

## Наследование классов

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}.

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

// Наследуем от Animal указывая "extends Animal"

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.hide(); // Белый кролик прячется!

Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение. Если мы определим свой метод в классе потомка, то он будет использоваться взамен родительского. Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. У классов есть ключевое слово "super" для таких случаев.

* super.method(...) вызывает родительский метод.
* super(...) вызывает родительский конструктор (работает только внутри нашего конструктора).
* У **стрелочных функций** нет super

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

stop() {

super.stop(); // вызываем родительский метод stop

this.hide(); // и затем hide

}

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!

Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор:

class Rabbit extends Animal {

// генерируется для классов-потомков, у которых нет своего конструктора

**constructor(...args) {**

**super(...args);**

**}**

}

В классах-потомках **конструктор обязан вызывать super**(...), и (!) делать это перед использованием this. Поэтому, если мы создаём собственный конструктор для потомка, мы должны вызвать super, в противном случае объект для this не будет создан, и мы получим ошибку.

## Статические свойства и методы

Мы также можем присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом static, например:

class User {

static staticMethod() {

alert(this === User);

}

}

User.staticMethod(); // true

Это фактически то же самое, что присвоить метод напрямую как свойство функции:

class User { }

User.staticMethod = function() {

alert(this === User);

}

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

class Article {

static publisher = "Илья Кантор";

}

alert( Article.publisher ); // Илья Кантор

Это то же самое, что и прямое присваивание Article:

Article.publisher = "Илья Кантор";

Эта возможность была добавлена в язык недавно.

Статические свойства и методы **наследуются**.

## Приватные и защищённые методы и свойства

В объектно-ориентированном программировании свойства и методы разделены на 2 группы:

* *Внутренний интерфейс* – методы и свойства, доступные из других методов класса, но не снаружи класса.
* *Внешний интерфейс* – методы и свойства, доступные снаружи класса.

В терминах ООП отделение внутреннего интерфейса от внешнего называется [**инкапсуляция**](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). В JavaScript есть два типа полей (свойств и методов) объекта:

* Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
* Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но **разрешён доступ для наследующих классов**). Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их **эмулируют**.

class CoffeeMachine {

**\_waterAmount** = 0;

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

get waterAmount() {

return this.\_waterAmount;

}

constructor(power) {

this.\_power = power;

}

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Свойство waterAmount сделали защищенным. Защищённые свойства обычно начинаются с префикса \_ - соглашение между программистами.

Чтобы сделать свойство **доступным только для чтения**, нужно создать только геттер, но не сеттер.

class CoffeeMachine {

// ...

constructor(power) {

this.\_power = power;

}

**get power**() {

return this.\_power;

}

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Здесь мы использовали синтаксис геттеров/сеттеров. Но в большинстве случаев использование функций get.../set... предпочтительнее:

class CoffeeMachine {

\_waterAmount = 0;

**setWaterAmount(value)** {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

**getWaterAmount()** {

return this.\_waterAmount;

}

}

new CoffeeMachine().setWaterAmount(100);

Защищённые методы **наследуются** в отличие от приватных свойств и методов. Приватные свойства и методы должны начинаться с #. Они доступны только внутри класса.

class CoffeeMachine {

#waterLimit = 200;

#checkWater(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

if (value > this.#waterLimit) throw new Error("Слишком много воды");

}

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

Приватные поля не конфликтуют с публичными. У нас может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

## Расширение встроенных классов

От встроенных классов, таких как Array, Map и других, тоже можно наследовать. Например, в этом примере PowerArray наследуется от встроенного Array:

// добавим один метод (можно более одного)

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);

alert(filteredArr); // 10, 50

alert(filteredArr.isEmpty()); // false

Обратите внимание на интересный момент: встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

Если бы мы хотели, чтобы методы map, filter и т. д. возвращали обычные массивы, мы могли бы вернуть Array в Symbol.species, вот так:

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

// встроенные методы массива будут использовать этот метод как конструктор

**static get [Symbol.species]() {**

**return Array;**

**}**

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

// filter создаст новый массив, используя arr.constructor[Symbol.species] как конструктор

let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array

alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function

## Проверка класса: "instanceof"

Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Также это работает с функциями-конструкторами и для встроенных классов.

let arr = [1, 2, 3];

alert( arr **instanceof** Array ); // true

alert( arr **instanceof** Object ); // true

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

## Примеси

Примесь – общий термин в объектно-ориентированном программировании: класс, который содержит в себе методы для других классов. Некоторые другие языки допускают множественное наследование. JavaScript не поддерживает множественное наследование, но с помощью примесей мы можем реализовать нечто похожее, скопировав методы в прототип.

// примесь

let sayHiMixin = {

sayHi() {

alert(`Привет, ${this.name}`);

},

sayBye() {

alert(`Пока, ${this.name}`);

}

};

// использование:

class User {

constructor(name) {

this.name = name;

}

}

// копируем методы

Object.assign(User.prototype, sayHiMixin);

// теперь User может сказать Привет

new User("Вася").sayHi(); // Привет, Вася!

Примеси могут наследовать друг друга.

let sayMixin = {

say(phrase) {

alert(phrase);

}

};

let sayHiMixin = {

**\_\_proto\_\_: sayMixin**, // (или мы можем использовать Object.create для задания прототипа)

sayHi() {

// вызываем метод родителя

super.say(`Привет, ${this.name}`); // (\*)

},

sayBye() {

super.say(`Пока, ${this.name}`); // (\*)

}

};

Мы можем использовать примеси для расширения функциональности классов, например, для обработки событий, как мы сделали это выше.

Давайте создадим примесь, которая позволит легко добавлять функциональность по работе с событиями любым классам/объектам.

* Примесь добавит метод .**trigger**(name, [data]) для генерации события. Аргумент name – это имя события, за которым могут следовать другие аргументы с данными для события.
* Также будет добавлен метод .**on**(name, handler), который назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем name, и получит данные из .trigger.
* …и метод .**off**(name, handler), который удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект user сможет сгенерировать событие "**login**" после входа пользователя в личный кабинет. А другой объект, к примеру, calendar сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь.

С примесями могут возникнуть конфликты, если они перезаписывают существующие методы класса. Стоит помнить об этом и быть внимательнее при выборе имён для методов примеси, чтобы их избежать.