JavaScript: руководство по классам

В JavaScript используется модель прототипного наследования: каждый объект наследует поля (свойства) и методы объекта-прототипа.

Прототипное наследование может имитировать классическую модель наследования от классов. Для этого в ES6 было представлено ключевое слово class: синтаксический сахар для прототипного наследования.

 Мы научимся определять классы, их частные (приватные) и открытые (публичные) поля и методы, а также создавать экземпляры.

### **1. Определение: ключевое слово class**

Для определения класса используется ключевое слово class:

**class** **User** {

// тело класса

}

Такой синтаксис называется объявлением класса.

Класс может не иметь названия. С помощью выражения класса можно присвоить класс переменной:

**const** UserClass = **class** {

// тело класса

}

Классы можно экспортировать в виде модулей. Вот пример экспорта по умолчанию:

**export** **default** **class** **User** {

// тело класса

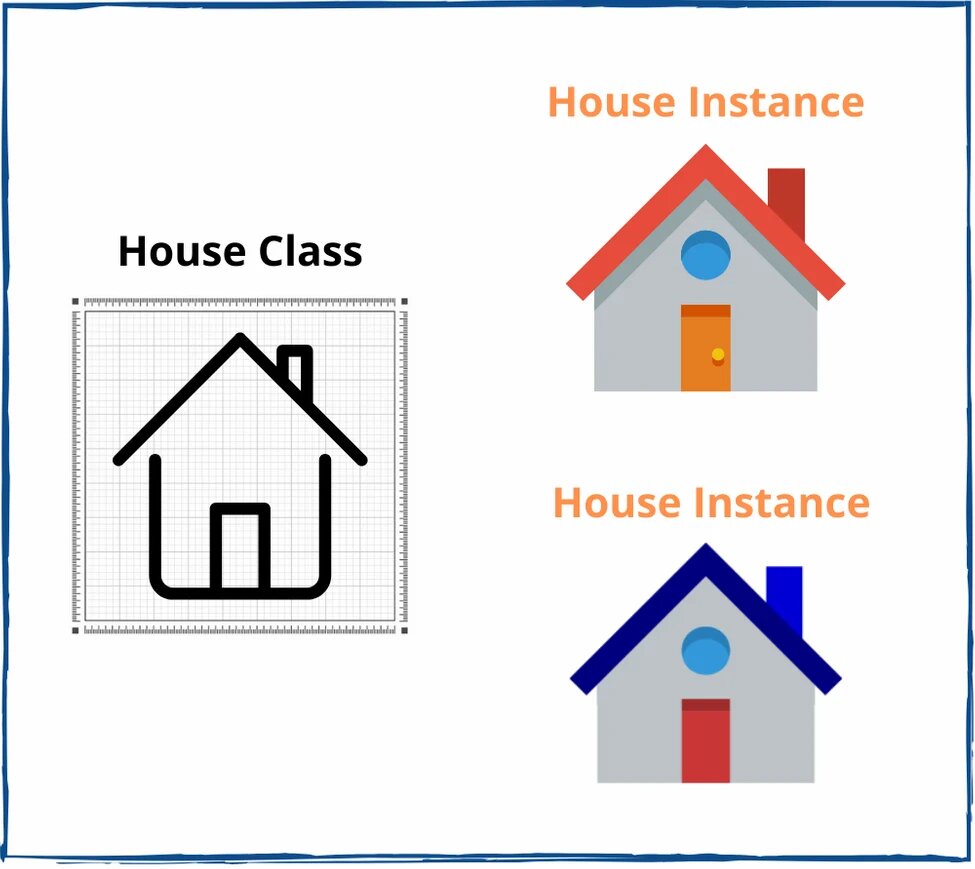
}

А вот пример именованного экспорта:

**export** **class** **User** {

// тело класса

}

Классы используются для создания экземпляров. Экземпляр — это объект, содержащий данные и логику класса.  
  
  
  
Экземпляры создаются с помощью оператора new: instance = new Class().  
  
Вот как создать экземпляр класса User:

**const** myUser = **new** User()

### **2. Инициализация: constructor()**

constructor(param1, param2, ...) — это специальный метод внутри класса, служащий для инициализации экземпляра. Это то место, где устанавливаются начальные значения полей экземпляра и осуществляется его настройка.  
  
В следующем примере конструктор устанавливает начальное значение поля name:

**class** **User** {

**constructor**(name) {

**this**.name = name

}

}

Конструктор принимает один параметр — name, который используется для установки начального значения поля this.name.  
  
this в конструкторе указывает на создаваемый экземпляр.  
  
Аргумент, используемый для создания экземпляра класса, становится параметром его конструктора:

**class** **User** {

**constructor**(name) {

name // Печорин

**this**.name = name

}

}

**const** user = **new** User('Печорин')

Параметр name внутри конструктора имеет значение 'Печорин'.  
  
Если не определить собственный конструктор, будет создан стандартный конструктор, представляющий собой пустую функцию, не влияющую на экземпляр.

### **3. Поля**

Поля класса — это переменные, содержащие определенную информацию. Поля могут быть разделены на две группы:

1. Поля экземпляров класса
2. Поля самого класса (статические)

Поля также имеют два уровня доступа:

1. Открытые (публичные): поля доступны как внутри класса, так и в экзмеплярах
2. Частные (приватные): поля доступны только внутри класса

### **3.1. Открытые поля экземпляров класса**

**class** **User** {

**constructor**(name) {

**this**.name = name

}

}

Выражение this.name = name создает поле экземпляра name и присваивает ему начальное значение.  
  
Доступ к этому полю можно получить с помощью аксессора свойства:

**const** user = **new** User('Печорин')

user.name // Печорин

В данном случае name — открытое поле, поскольку оно доступно за пределами класса User.  
  
При неявном создании полей внутри конструктора, сложно получить список всех полей. Для этого поля нужно извлекать из конструктора.  
  
Лучшим способом является явное определение полей класса. Неважно, что делает конструктор, экземпляр всегда имеет одинаковый набор полей.  
  
[Предложение по созданию полей класса](https://github.com/tc39/proposal-class-fields) позволяет определять поля внутри класса. Кроме того, здесь же можно присваивать полям начальные значения:

**class** **SomeClass** {

field1

field2 = 'Начальное значение'

// ...

}

Изменим код класса User, определив в нем открытое поле name:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

}

**const** user = **new** User('Печорин')

user.name // Печорин

Такие открытые поля являются очень наглядными, быстрый взгляд на класс позволяет понять структуру его данных.  
  
Более того, поле класса может быть инициализировано в момент определения:

**class** **User** {

name = 'Имярек'

**constructor**() {

// инициализация отсутствует

}

}

**const** user = **new** User()

user.name // Имярек

На доступ к открытым полям и их изменение нет ограничений. Читать и присваивать значения таким полям можно в конструкторе, методах и за пределами класса.

### **3.2. Частные поля экземпляров класса**

Инкапсуляция позволяет скрывать внутренние детали реализации класса. Тот, кто использует инкапсулированный класс, опирается на публичный интерфейс, не вдаваясь в подробности реализации класса.  
  
Такие классы проще обновлять при изменении деталей реализации.  
  
Хорошим способом скрыть детали является использование частных полей. Такие поля могут быть прочитаны и изменены только внутри класса, которому они принадлежат. За пределами класса частные поля недоступны.  
  
Для того, чтобы сделать поле частным, перед его названием следует поставить символ #, например, #myPrivateField. При обращении к такому полю всегда должен использоваться указанный префикс.  
  
Сделаем поле name частным:

**class** **User** {

#name

**constructor**(name) {

**this**.#name = name

}

getName() {

**return** **this**.#name

}

}

**const** user = **new** User('Печорин')

user.getName() // Печорин

user.#name // SyntaxError

#name — частное поле. Доступ к нему можно получить только внутри класса User. Это позволяет сделать метод getName().  
  
Однако, при попытке получить доступ к #name за пределами класса User будет выброшена синтаксическая ошибка: SyntaxError: Private field '#name' must be declared in an enclosing class.

### **3.3. Открытые статические поля**

В классе можно определить поля, принадлежащие самому классу: статические поля. Такие поля используются для создания констант, хранящих нужную классу информацию.  
  
Для создания статических полей используется ключевое слово static перед названием поля: static myStaticField.  
  
Добавим новое поле type для определения типа пользователя: администратора или обычного. Статические поля TYPE\_ADMIN и TYPE\_REGULAR — константы для каждого типа пользователей:

**class** **User** {

**static** TYPE\_ADMIN = 'admin'

**static** TYPE\_REGULAR = 'regular'

name

type

**constructor**(name, type) {

**this**.name = name

**this**.type = type

}

}

**const** admin = **new** User('Администратор сайта', User.TYPE\_ADMIN)

admin.type === User.TYPE\_ADMIN // true

Для доступа к статическим полям следует использовать название класса и название свойства: User.TYPE\_ADMIN и User.TYPE\_REGULAR.

### **3.4. Частные статические поля**

Иногда статические поля также являются частью внутренней реализации класса. Для инкапсуляции таких полей можно сделать их частными.  
  
Для этого следует перед названием поля поставить префикс #: static #myPrivateStaticFiled.  
  
Предположим, что мы хотим ограничить количество экземпляров класса User. Для сокрытия информации о количестве экземпляров можно создать частные статические поля:

**class** **User** {

**static** #MAX\_INSTANCES = 2

**static** #instances = 0

name

**constructor**(name) {

User.#instances++

**if** (User.#instances > User.#MAX\_INSTANCES) {

**throw** **new** Error('Невозможно создать экземпляр класса User')

}

**this**.name = name

}

}

**new** User('Печорин')

**new** User('Бэла')

**new** User('Грушницкий') // Невозможно создать экземпляр класса User

Статическое поле User.#MAX\_INSTANCES определяет допустимое количество экземпляров, а User.#instances — количество созданных экземпляров.  
  
Эти частные статические поля доступны только внутри класса User. Ничто из внешнего мира не может повлиять на ограничения: в этом заключается одно из преимуществ инкапсуляции.  
  
Прим. пер.: если ограничить количество экземпляров одним, получится интересная реализация шаблона проектирования «Одиночка» (Singleton).

### **4. Методы**

Поля содержат данные. Возможность изменять данные обеспечивается специальными функциями, являющимися частью класса: методами.  
  
JavaScript поддерживает как методы экземпляров класса, так и статические методы.

### **4.1. Методы экземпляров класса**

Методы экземпляра класса могут изменять его данные. Методы экземпляра могут вызывать другие методы экземпляра, а также статические методы.  
  
Например, определим метод getName(), возвращающий имя пользователя:

**class** **User** {

name = 'Имярек'

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**const** user = **new** User('Печорин')

user.getName() // Печорин

В методе класса, также как и в конструкторе, this указывает на создаваемый экземпляр. Используйте this для получения данных экземпляра: this.field, или для вызова методов: this.method().  
  
Добавим новый метод nameContains(str), принимающий один аргумент и вызывающий другой метод:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

nameContains(str) {

**return** **this**.getName().includes(str)

}

}

**const** user = **new** User('Печорин')

user.nameContains('Печорин') // true

user.nameContains('Грушницкий') // false

nameContains(str) — метод класса User, принимающий один аргумент. Он вызывает другой метод экземпляра getName() для получения имени пользователя.  
  
Метод также может быть частным. Для того, чтобы сделать метод частным следует использовать префикс #.  
  
Сделаем метод getName() частным:

**class** **User** {

#name

**constructor**(name) {

**this**.#name = name

}

#getName() {

**return** **this**.#name

}

nameContains(str) {

**return** **this**.#getName().includes(str)

}

}

**const** user = **new** User('Печорин')

user.nameContains('Печорин') // true

user.nameContains('Грушницкий') // false

user.#getName // SyntaxError

#getName() — частный метод. Внутри метода nameContains(str) мы вызываем его так: this.#getName().  
  
Будучи частным, метод #getName() не может быть вызван за пределами класса User.

### **4.2. Геттеры и сеттеры**

Геттеры и сеттеры — это аксессоры или вычисляемые свойства. Это методы, имитирующие поля, но позволяющие читать и записывать данные.  
  
Геттеры используются для получения данных, сеттеры — для их изменения.  
  
Для установки запрета на присвоение полю name пустой строки, обернем частное поле #nameValue в геттер и сеттер:

**class** **User** {

#nameValue

**constructor**(name) {

**this**.name = name

}

**get** name() {

**return** **this**.#nameValue

}

**set** name(name) {

**if** (name === '') {

**throw** **new** Error('Имя пользователя не может быть пустым')

}

**this**.#nameValue = name

}

}

**const** user = **new** User('Печорин')

user.name // вызывается геттер, Печорин

user.name = 'Бэла' // вызывается сеттер

user.name = '' // Имя пользователя не может быть пустым

### **4.3. Статические методы**

Статические методы — это функции, принадлежащие самому классу. Они определяют логику класса, а не его экземпляров.  
  
Для создания статического метода используется ключевое слово static перед названием метода: static myStaticMethod().  
  
При работе со статическими методами, следует помнить о двух простых правилах:

1. Статический метод имеет доступ к статическим полям
2. Он не имеет доступа к полям экземпляров

Создадим статический метод для проверки того, что пользователь с указанным именем уже создан:

**class** **User** {

**static** #takenNames = []

**static** isNameTaken(name) {

**return** User.#takenNames.includes(name)

}

name = 'Имярек'

**constructor**(name) {

**this**.name = name

User.#takenNames.push(name)

}

}

**const** user = **new** User('Печорин')

User.isNameTaken('Печорин') // true

User.isNameTaken('Грушницкий') // false

isNameTaken() — статический метод, использующий частное статическое поле User.#takenNames для определения использованных имен.  
  
Статические методы также могут быть частными: static #myPrivateStaticMethod(). Такие методы могут вызываться только внутри класса.

### **5. Наследование: extends**

Классы в JavaScript поддерживают наследование с помощью ключевого слова extends.  
  
В выражении class Child extends Parent { } класс Child наследует от класса Parent конструктор, поля и методы.  
  
Создадим дочерний класс ContentWriter, расширяющий родительский класс User:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**class** **ContentWriter** **extends** **User** {

posts = []

}

**const** writer = **new** ContentWriter('Лермонтов')

writer.name // Лермонтов

writer.getName() // Лермонтов

writer.posts // []

ContentWriter наследует от User конструктор, метод getName() и поле name. В самом ContentWriter определяется новое поле posts.  
  
Обратите внимание, что частные поля и методы родительского класса не наследуются дочерними классами.

###### **5.1. Родительский конструктор: super() в constructor()**

Для того, чтобы вызвать конструктор родительского класса в дочернем классе, следует использовать специальную функцию super(), доступную в конструкторе дочернего класса.  
  
Пусть конструктор ContentWriter вызывает родительский конструктор и инициализирует поле posts:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**class** **ContentWriter** **extends** **User** {

posts = []

**constructor**(name, posts) {

**super**(name)

**this**.posts = posts

}

}

**const** writer = **new** ContentWriter('Лермонтов', ['Герой нашего времени'])

writer.name // Лермонтов

writer.posts // ['Герой нашего времени']

super(name) в дочернем классе ContentWriter вызывает конструктор родительского класса User.  
  
Обратите внимание, что в дочернем конструкторе перед использованием ключевого слова this вызывается super(). Вызов super() «привязывает» родительский конструктор к экземпляру.

**class** **Child** **extends** **Parent** {

**constructor**(value1, value2) {

// не работает!

**this**.prop2 = value2

**super**(value1)

}

}

###### **5.2. Родительский экземпляр: super в методах**

Для того, чтобы получить доступ к родительскому методу внутри дочернего класса, следует использовать специальное сокращение super:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**class** **ContentWriter** **extends** **User** {

posts = []

**constructor**(name, posts) {

**super**(name)

**this**.posts = posts

}

getName() {

**const** name = **super**.getName()

**if** (name === '') {

**return** 'Имярек'

}

**return** name

}

}

**const** writer = **new** ContentWriter('', ['Герой нашего времени'])

writer.getName() // Имярек

getName() дочернего класса ContentWriter вызывает метод getName() родительского класса User.  
  
Это называется переопределением метода.  
  
Обратите внимание, что super можно использовать и для статических методов родительского класса.

### **6. Проверка типа объекта: instanceof**

Выражение object instanceof Class определяет, является ли объект экземпляром указанного класса.  
  
Рассмотрим пример:

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**const** user = **new** User('Печорин')

**const** obj = {}

user **instanceof** User // true

obj **instanceof** User // false

Оператор instanceof полиморфичен: он исследует всю цепочку классов.

**class** **User** {

name

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**class** **ContentWriter** **extends** **User** {

posts = []

**constructor**(name, posts) {

**super**(name)

**this**.posts = posts

}

}

**const** writer = **new** ContentWriter('Лермонтов', ['Герой нашего времени'])

writer **instanceof** ContentWriter // true

writer **instanceof** User // true

Что если нам нужно определить конкретный класс экземпляра? Для этого можно использовать свойство constructor:

writer.constructor === ContentWriter // true

writer.constructor === User // false

// или

writer.\_\_proto\_\_ === ContentWriter.prototype // true

writer.\_\_proto\_\_ === User.prototype // false

**7. Классы и прототипы**

Надо сказать, что синтаксис классов — это хорошая абстракция над прототипным наследованием. Для использования классов не нужно обращаться к прототипам.  
  
Однако, классы являются лишь надстройкой над прототипным наследованием. Любой класс — это функция, создающая экземпляр при вызове конструктора.  
  
Следущие два примера идентичны.  
  
Классы:

**class** **User** {

**constructor**(name) {

**this**.name = name

}

getName() {

**return** **this**.name

}

}

**const** user = **new** User('Печорин')

user.getName() // Печорин

user **instanceof** User // true

Прототипы:

**function** **User**(name) {

**this**.name = name

}

User.prototype.getName = **function** () {

**return** **this**.name

}

**const** user = **new** User('Печорин')

user.getName() // Печорин

user **instanceof** User // true

Поэтому для понимания классов требуется хорошее знание прототипного наследования.