

# Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery



# Урок № 6

## ECMAScript 6

### Содержание

<b>ECMAScript 6.....</b>	<b>4</b>
<b>Переменные .....</b>	<b>6</b>
Области видимости переменных.....	6
Всплытие .....	15
<b>Тип данных Symbol .....</b>	<b>24</b>
Глобальные символы.....	33
<b>Стрелочные функции .....</b>	<b>38</b>
Стрелочные функции как callback функции .....	42
This в стрелочных функциях.....	45
Стрелочные функции как методы объекта .....	57
<b>Классы.....</b>	<b>63</b>
Свойства-аксессоры .....	69
Статические свойства и методы класса.....	77
Наследование .....	80
<b>Модули.....</b>	<b>105</b>
Ключевое слово as.....	113

<b>Деструктуризация.....</b>	<b>119</b>
Деструктуризация объекта .....	119
Деструктуризация массива .....	127
<b>Аргументы по умолчанию.....</b>	<b>131</b>
<b>Строковые шаблоны.....</b>	<b>136</b>
<b>Домашнее задание EcmaScript 6 .....</b>	<b>140</b>
Задание 1. Разработка класса .....	140
Задание 2. Наследование классов.....	143
Задание 3. Деструктуризация .....	144
Задание 4. Параметры по умолчанию.....	145

# ECMAScript 6

С момента выхода JavaScript в 1995 году прошло много времени и язык приобрел новые возможности, некоторые из которых мы рассмотрим в этом уроке, а именно нововведения стандарта ECMAScript 6.

ECMAScript — это стандарт языка JavaScript разработанный ассоциацией ECMA, которая занимается стандартизацией информационных технологий.

Стандарт ECMAScript отвечает за синтаксис и важные составляющие языка, такие как типы данных, наследование, стандартные объекты, а именно [JSON](#), [Math](#), [Array](#) и многое другое, а также правила, которых должен придерживаться язык.

Изначально JavaScript не был стандартизирован. Первая стандартизация прошла в 1997 году, когда был принят стандарт ECMA-262. С полным описанием ECMA-262 можно ознакомиться [здесь](#).

Каждый новый стандарт кратко называют ES и указывают цифру обозначающую порядковый номер. Например, ES1 — первый стандарт, ES2 второй и т.д. Всего существует 10 стандартов каждый из которых вносит новые возможности.

Стандарты нужны для борьбы с хаосом и неразберихой. Так как JavaScript выполняется в разных браузерах, каждый из которых использует свой собственный движок для интерпретации страницы. Google Chrome использует движок Blink, Mozilla — Gecko, Safari — WebKit и т.д. Если бы спецификации не существовало, то каждый из движков

мог бы исполнять JavaScript по-разному и в каждом браузере выполнение одного JS кода приводило бы к разным результатам. Чтобы этого избежать JavaScript реализует спецификацию ECMAScript.

На данный момент наиболее новой спецификацией является ES10, которая была выпущена в июне 2019 года.

Однако наибольший вклад в развитие JS внес стандарт 2015 года — ES6. О нем и пойдет речь в этом уроке. Официальное название этого стандарта — ES2015 (когда-то было принято решение о том, чтобы именовать стандарты с указанием года выпуска, а не с его порядковым номером, но краткое название также осталось). Немного путает то, что стандарт имеет два названия, просто нужно запомнить, что есть официальное и краткое. Обычно используют краткое — ES6, однако на просторах интернета можно встретить и ES2015.

Теперь давайте приступим к детальному изучению нововведений ES6.

# Переменные

До появления ES6 переменные можно было объявить только с помощью ключевого слова **var**. Сейчас объявление переменных через **var** считается устаревшим, так как с ним связано множество проблем, которые мы разберем далее. На замену **var** пришли **let** и **const** в ES6.

Почему **var** считается устаревшим? Для того чтобы ответить на этот вопрос, нужно рассмотреть некоторые тонкости.

## Области видимости переменных

**Область видимости переменной** это — часть программы, в которой можно использовать эту переменную, т.е. область видимости определяет доступность переменной. Переменная может иметь **глобальную** или **локальную** видимость. Переменные объявленные в глобальной области видимости, а именно вне функции или блока, будут доступны в любой части программы.

Давайте рассмотрим глобальные переменные на небольшом примере:

```
var globalVar = 1;

function setGlobalVar () {
    globalVar = 2;
}

setGlobalVar();
console.log(globalVar);
```

Здесь создается переменная `globalVar` через оператор `var` и которой изначально присваивается значение `1`. После объявляется функция `setGlobalVar()`, изменяющая значение `globalVar` на `2`. Эта функция вызывается, а после в консоль выводится значение выше объявленной переменной. На экране появится значение `2`.

Наша переменная `globalVar` объявлена в глобальной области видимости, а это значит, что получить доступ к ней мы можем из любой части кода.

Однако это не совсем безопасно, так как из-за того, что глобальная переменная доступна везде ее значение легко случайно изменить, что может привести к нарушению работы кода. Также могут появиться проблемы с именованием переменных. Если проект достаточно большой или его разрабатывают несколько программистов, то практически невозможно запомнить название всех переменных, из-за чего можно случайно перезаписать одну из них, не заметив ошибку.

Ниже продемонстрировано, как это может произойти:

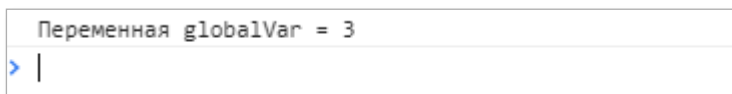
```
var globalVar = 1;
function setGlobalVar() {
    globalVar = 2;
}

setGlobalVar();
var globalVar = 3;

console.log(globalVar);
```

В начале создается глобальная переменная `globalVar` и ей присваивается значение `1`. Далее находится функция,

которая изменяет значение `globalVar` на 2, а еще ниже создается переменная с таким же именем, но с другим значением. После вывода значения переменной `globalVar` на экран мы увидим следующее:



```
Переменная globalVar = 3  
> |
```

*Рисунок 1*

В других языках программирования объявление двух переменных с одним и тем же именем вызовет синтаксическую ошибку. В JavaScript тоже не может быть двух или более переменных с одинаковым именем, но в отличие от других языков в JavaScript это не вызовет синтаксическую ошибку. Вместо этого создается только одна переменная, а при следующем объявлении будет перезаписано значение уже созданной переменной.

Из скриншота (рис. 1) видно, что значение переменной `globalVar` равно 3, это непредвиденное поведение, которое может нарушить работу кода. Сообщение об ошибке нет и заметить такую ошибку сложно, так как одинаковые переменные могут быть в разных файлах или в других библиотеках, а также другие программисты в вашей команде могут случайно объявить переменную с таким же именем. Проблему можно избежать, для этого нужно свести к минимуму использование глобальных переменных или начать использовать ключевое слово `let`, но о нем позже.

Также глобальные переменные создаются, когда при их объявлении не указывается ключевое слово `var`,



`let` или `const`, даже если они объявляются в функции или блоке.

Вот как это выглядит:

```
function welcome() {  
    greeting = "Hello";  
}  
  
welcome();  
console.log(greeting);
```

В примере есть функция `welcome()`, в которой объявляется переменная `greeting` без ключевого слова `var` или `let`. В этом случае переменная `greeting` будет находиться в глобальной области видимости и обратиться к ней можно даже за пределами функции, в которой она была объявлена.

Переменные объявленные в локальной области видимости, т.е. в теле функции или блока, будут доступны в той области блока или функции, в которой они были объявлены.

Вот как работают локальные переменные:

```
function printLocalVar() {  
    var localVar = 1;  
    console.log(localVar);  
}  
  
printLocalVar();  
console.log(localVar);
```

В функции `printLocalVar()` объявлена переменная `localVar`, которая является локальной. К ней можно

получить доступ из локальной области видимости, т.е. из функции, в которой объявлена переменная или из другой вложенной функции, но нельзя получить доступ извне. Функция `printLocalVar()` выводит значение переменной `localVar`, равное 1. В последней строке функция `console.log()` пытается получить доступ к переменной, находящейся в локальной области видимости функции `printLocalVar()` а не в глобальной области, что приводит к ошибке: «*Uncaught ReferenceError: localVar is not defined*».

Теперь со знанием, что такое область видимости, можем перейти к изучению ключевых слов `let` и `const`.

Для начала разберемся с `let`. Оператор `let`, как и `var`, используется для объявления переменной, но использование `let` позволяет избежать множества проблем, связанных с `var`.

Основное отличие `let` от `var` в области видимости. Переменные, которые объявлены с помощью `let` поддерживают локальную область видимости внутри блоков. Выше было сказано про локальную область видимости, что переменные объявленные внутри блока будут локальными. Это касается только переменных, объявленных с помощью `let`, так как ключевое слово `var` не поддерживает локальную область видимости в блоках.

Чтобы, это понять давайте рассмотрим пример:

```
function welcome() {  
    var greeting = "Hello";  
  
    if (true) {  
        var greeting = "Hi";  
    }  
}
```

```

        console.log(greeting);
    }
    console.log(greeting);
}
welcome();

```

Вначале объявлена переменная `greeting` которая хранит значение «Hello». Ниже в блоке условного оператора создается еще одна переменная, но с другим значением. Из-за того, что переменная объявлена с помощью `var` она не имеет блочной области видимости и две переменные с одним именем находятся в одной области видимости функции `welcome()`. Из-за того, что в одной области видимости не может быть двух переменных с одним именем то создается одна первая переменная со значением «Hello», а далее просто перезаписывается ее значение на «Hi». И на экране мы видим две строки «Hi».



Рисунок 2

Из скриншота (рис. 2) можно увидеть, что переменная `greeting` теперь хранит значение «Hi».

Давайте рассмотрим такой же пример только с использованием `let`:

```

function welcome() {
    let greeting = "Hello";
    if (true) {

```

```

        let greeting = "Hi";
        console.log(greeting);
    }
    console.log(greeting);
}
welcome();

```

В функции `welcome()` в начале объявляется переменная `greeting` со значением «`Hello`», после идет условный оператор, в его блоке объявляется другая переменная `greeting` со значением «`Hi`». На экране будет две строки сначала «`Hi`» а потом «`Hello`»:



Рисунок 3

Это происходит, потому что условный оператор `if` создает свою блочную область видимости для переменной. Поэтому эти две переменные хоть и называются одинаково, но по сути это две разные переменные в разных областях видимости. Когда происходит обращение к переменной используется та переменная, которая ближе к области видимости. Таким образом, когда в блоке `if` команда «`console.log(greeting);`» выводит на экран значение переменной `greeting`, берется та переменная, которая объявлена в блочной области видимости.

Для наглядности рассмотрим изображения, на которых видно, как располагаются области видимости при использовании `var` или `let`.

При использовании `var`:

```
function welcome() {  
    var greeting = "Hello";  
    if (true) {  
        var greeting = "Hi";  
        console.log(greeting);  
    }  
    console.log(greeting);  
}  
welcome();
```

Область видимости функции

Рисунок 4

На рисунке 4 показано, что при использовании ключевого слова `var` областью видимости для всех переменных является вся функция, без учета других блоков.

При использовании `let`:

```
function welcome(){  
    let greeting = "Hello";  
    if(true){  
        let greeting = "Hi";  
        console.log(greeting);  
    }  
    console.log(greeting);  
}  
welcome();
```

Область видимости функции

Блочная область видимости

Рисунок 5

Из рисунка 5 видно, что, когда переменные объявляются через **let**, область видимости может быть как функция, так и блоки, что дает возможность создавать переменные с одним именем в разных областях. Это также позволяет избежать ошибок со случайной перезаписью переменной.

Глобальные переменные, объявленные с помощью **let** работают также, но с особенностями переменных объявленных с помощью **let**. Это значит, что двух переменных с одним именем быть не может и если вы попытаетесь объявить две одинаковые переменные — генерируется ошибка. Это можно увидеть в этом примере:

```
let globalLet= 1;
function setGlobalVar() {
    globalLet = 2;
}

setGlobalVar();
let globalLet = 3;
console.log('Переменная globalLet = ${globalLet} ' );
```

Скриншот показывает работу примера выше:

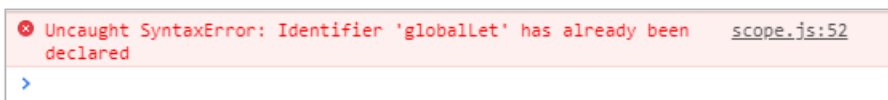


Рисунок 6

Из него видно, что при попытке объявить еще одну переменную **globalLet** возникает ошибка.

Следующей темой является всплытие переменных, которая необходима для изучения дальнейшего материала.

## Всплытие

Всплытие или как еще называют **hoisting** — это механизм, который поднимает переменные и объявление функций в начало своей области видимости.

Давайте рассмотрим пример:

```
console.log(greeting);  
var greeting = "Hello";  
console.log(greeting);
```

В первой строке выполняется функция **console.log()**, она выводит значение пока не созданной переменной **greeting**. Далее создается и инициализируется переменная **greeting** и ее значение еще раз выводится на экран.



undefined
Hello

*Рисунок 7*

На экране сначала появится значение «**undefined**», а после «**Hello**». Но почему «**undefined**», а не ошибка? Ведь мы в начале обращаемся к несуществующей переменной?

А все по тому, что в силу вступает механизм всплытия. Изначально интерпретатор JavaScript проходит по коду, находит все переменные и поднимает их в верх области видимости, при этом инициализирует их значение «**undefined**» по умолчанию. Поэтому при первом выводе на экран переменной **greeting**, она равна «**undefined**». Инициализация переменных не поднимается вместе с пе-

ременной в верх, а происходит там, где изначально была объявлена, т.е. `greeting` получит значение «`Hello`» во 2-й строке. В последней строке на экране появится значение уже инициализированной переменной.

Если учесть все особенности всплытия, пример выше будет интерпретирован так:

```
var greeting = undefined;  
console.log(greeting);  
greeting = "Hello"  
console.log(greeting);
```

Интерпретатор проходит по коду, находит переменную `greeting` и поднимает ее в верх, инициализируя и присваивая значение «`undefined`». Ниже уже происходит инициализация в том месте, где была объявлена переменная, а дальше происходит выполнение остального кода.

Использовать переменную до ее объявления не совсем логично и может вызвать проблемы, которые сложно заметить и исправить.

Вот одна из таких проблем:

```
function addOne() {  
    console.log(num + 1);  
    var num = 1;  
}
```

Есть функция `addOne()`, в которой сначала происходит обращение к переменной `num` и ее значение используется для вычисления суммы выражения. Из материала выше мы помним, что сначала происходит всплытие переменной в начало ее области видимости, при этом в перемен-



ную записывается значение `undefined`. В первой строчке значение `num` равно `undefined`, и к значению `undefined` происходит добавление числа. При попытке добавления числа к `undefined` возникает математическая ошибка, и результатом такого выражения будет значение `NaN`. Это можно увидеть на скриншоте:



Рисунок 8

А теперь давайте посмотрим, что будет если использовать переменную до ее создания, которая объявлена с помощью ключевого слова `let` на этом примере:

```
console.log(greeting);  
let greeting = "Hello"  
console.log(greeting);
```

В этом примере используется переменная `greeting`, которая была создана с помощью `let` вместо `var`. Можно подумать, что в первой строке, когда мы обращаемся к переменной, которая еще не была создана, ее значение будет `undefined` и на экране мы увидим именно его. Однако это не так и на экране мы увидим ошибку:

```
✖ ▶ Uncaught ReferenceError: Cannot access 'greeting' before initialization  
   at hosting.js:17
```

Рисунок 9

Переменные, которые объявлены с помощью `let`, все также поддерживают всплытие, но интерпретатор

не присваивает таким переменным `undefined`, а вместо этого они попадают во временную мертвую зону (*Temporal Dead Zone*). Переменная находится в мертвой зоне пока выполнение кода не дойдет до ее инициализации, до этого момента обращение к переменной приведет к ошибке.

Таким образом, при обращении к переменным до их инициализации программист будет уведомлен об ошибке, сможет легко найти ее и исправить.

Также `let` не позволяет объявить две переменные с одним именем и в одной области видимости, в отличие от `var`. Если это попытаться сделать, появится ошибка:

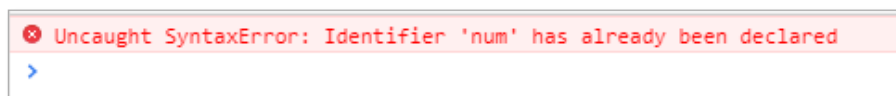


Рисунок 10

Объявляется глобальная переменная `globalLet` со значением `1` через оператор `let`. Работа с такой переменной происходит как с обычной глобальной переменной, она видна в других областях видимости, ее можно изменять. Однако при попытке объявить еще одну переменную с таким же именем, в 7 строке, произойдет ошибка, которую можно увидеть в консоли разработчика.

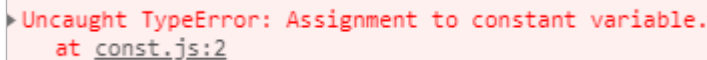
Настало время перейти к изучению ключевого слова `const`.

Ключевое слово `const` очень похоже на `let`, но есть одно большое отличие. Значения переменных, объявленных с помощью `const`, нельзя изменить. Такие переменные называются константами.

Здесь показано, что будет если попытаться изменить значение константы:

```
const PI = 3.14;  
PI = 3;
```

Когда мы пытаемся перезаписать значение константы происходит следующая ошибка:



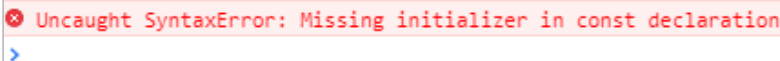
```
Uncaught TypeError: Assignment to constant variable.  
at const.js:2
```

Рисунок 11

Значение константы должно быть присвоено сразу после ее создания, иначе это приведет к ошибке, как в этом примере:

```
const PI;  
PI = 3.14;
```

Изначально создается константа `PI`, а уже после ей присваивается значение. Так использовать константу нельзя, поэтому произойдет ошибка:



```
Uncaught SyntaxError: Missing initializer in const declaration
```

Рисунок 12

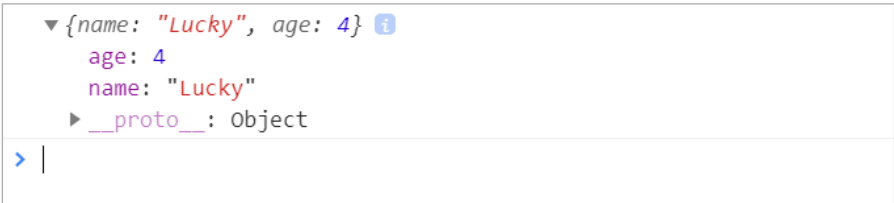
Раньше, до ES6, не было возможности объявить константу и поэтому была договоренность, что все переменные название которых написаны заглавными буквами являются константами и изменить их значения нельзя.

Однако сейчас эта договоренность не актуальна так как можно использовать ключевое слово `const` которое гарантирует, что значение не изменится.

Важно знать, что, если в константе находится объект, нельзя перезаписать только ссылку на этот объект, однако изменить сам объект можно. Вот как в этом примере:

```
let catLucky = {  
  name: "Lucky",  
  age: 3  
}  
  
const cat = catLucky;  
cat.age = 4;  
console.log(cat);
```

В константу `cat` присваивается объект `catLucky`. Это значит, что `cat` хранит ссылку на `catLucky`. После в объекте `catLucky` происходит изменение одного его из свойств. В результате при выводе константы `cat`, объект `catLucky`, на который ссылается константа будет иметь такой вид:



```
▼ {name: "Lucky", age: 4} ⓘ  
  age: 4  
  name: "Lucky"  
  ► ___proto___: Object
```

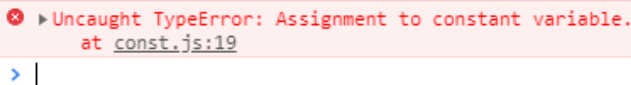
Рисунок 13

Также, можно добавлять и удалять свойства у объекта и это тоже не приведет к нарушению, главное, чтобы константа ссылалась на один и тот же объект.

Из-за того, что константу нельзя повторно инициализировать, этот код приведет к ошибке:

```
let catLucky = {  
  name: "Lucky",  
  age: 3  
}  
  
let catJessica = {  
  name: "Jessica",  
  age: 3  
}  
  
const cat = catLucky;  
cat = catJessica;  
  
console.log(cat);
```

Выше создается два объекта, сначала константа `cat` содержит в себе ссылку на объект `catLucky`, но в следующей строке происходит попытка присвоить ссылку на другой объект `catJessica`. В результате выполнения этого кода будет такая ошибка:



```
✖ Uncaught TypeError: Assignment to constant variable.  
  at const.js:19  
> |
```

Рисунок 14

То же происходит и с массивами. Когда константа ссылается на массив, добавлять и удалять элементы массива можно, нельзя перенаправлять константу на другой массив.

В этом примере можно увидеть, что добавление элементов в массив, на который ссылается константа не приведет к ошибке:

```
let catJessica = {
  name: "Jessica",
  age: 3
};

let dogScoobyDoo = {
  name: "Scooby Doo",
  age: 5
}

let arrPets = [catJessica, dogScoobyDoo];
const myPets = arrPets;

console.log(myPets);

let parrotPolly = {
  name: "Polly",
  age: 1
}

myPets.push(parrotPolly);
console.log(myPets);
```

Изначально есть два объекта `catJessica` и `dogScoobyDoo`, которые далее становятся элементами массива `arrPets`. После объявляется константа `myPets`, инициализируется ссылкой на массив `arrPets` и выводится на экран ее значение. Позже создается еще один объект `parrotPolly`, добавляется в массив, на который ссылается константа `myPets` и снова на экран выводится `myPets`.

```

▼ (2) [{...}, {...}] ⓘ
  ► 0: {name: "Jessica", age: 3}
  ► 1: {name: "Scooby Doo", age: 5}
  ► 2: {name: "Kesha", age: 1}
  length: 3
  ► __proto__: Array(0)

▼ (3) [{...}, {...}, {...}] ⓘ
  ► 0: {name: "Jessica", age: 3}
  ► 1: {name: "Scooby Doo", age: 5}
  ► 2: {name: "Kesha", age: 1}
  length: 3
  ► __proto__: Array(0)

```

Рисунок 15

Из скриншота видно, что при первом выводе на экран значением константы является массив, в котором находится два объекта. После происходит добавление элемента в массив, на который ссылается `myPets`. Потом значение константы `myPets` выводится на экран второй раз, и мы видим, что константа хранит ссылку на массив, в котором уже три элемента.

В этой таблице описаны различия между `var`, `let` и `const`:

Таблица 1

	<code>var</code>	<code>let</code>	<code>const</code>
Область видимости	Ограничена областью видимости функции	Ограничена областью видимости функции и блока	Ограничена областью видимости функции и блока
Значение при обращении к переменной, до ее объявления	<code>undefined</code>	<code>ReferenceError</code>	<code>ReferenceError</code>
Изменяемость	Может изменяться	Может изменяться	Не изменяема

# Тип данных Symbol

До появления ES6 в JavaScript было 5 примитивных типов данных, а именно: **number**, **string**, **boolean**, **null** и **undefined**. В стандарте ES6 появился новый примитивный тип — **Symbol**.

**Тип данных Symbol** или как его еще называют — символ — представляет собой уникальный идентификатор. Символы создаются с помощью функции **Symbol()**.

```
let symbol = Symbol();
```

**Symbol** создается без ключевого слова **new**, так как является примитивным.

**Примитивные типы** — это данные, такие как **34**, «**Hello**», **true** и другие, кроме объектов. Примитивные типы не имеют свойств и методов для работы. Также, такие значения являются неизменяемыми (иммутабельными).

При проверке на тип оператор **typeof** возвращает тип данных **symbol**.

При создании символа ему можно дать описание. Это необходимо для удобной отладки.

```
let name = Symbol("name");
```

Здесь создается символ с описанием «**name**».

Каждый символ является уникальным, даже если создать множество символов с одинаковым описанием это будут разные символы. Это видно из примера ниже:



```
let name1 = Symbol("name");  
let name2 = Symbol("name");  
  
console.log(name1 === name2);
```

Изначально создаются две переменные и им присваиваются значения типа **symbol** с одинаковым описанием. После эти переменные сравниваются на идентичность и в консоли можно увидеть, что значения переменных не являются одинаковыми:

A screenshot of a JavaScript console window. The text 'false' is displayed in blue, indicating a boolean result. Below the text is a blue arrow pointing to the right, which is a standard prompt for the next command.

Рисунок 16

Два символа с одинаковым описанием это два разных символа, потому что каждый символ уникален, их описания никак не влияют на уникальность.

В этом примере можно посмотреть, что появится в консоли, если попытаться получить значение переменной, которая содержит символ:

```
let valueId = Symbol("id");  
console.log(valueId);
```

На экране появится такой результат:

A screenshot of a JavaScript console window. The text 'Symbol(id)' is displayed in red, indicating a Symbol object. Below the text is a blue arrow pointing to the right, which is a standard prompt for the next command.

Рисунок 17

Получить значение, которое хранит `valueId` напрямую мы не можем. Вместо этого мы видим такую строку «`Symbol(id)`», здесь указан тип и описание символа.

Символы можно использовать в качестве ключей для свойств объекта. Они позволяют создавать скрытые свойства объекта, к которым нельзя обратиться обычным способом.

Давайте рассмотрим преимущества таких свойств на этом примере:

```
let user = {  
  login: "user1",  
  [Symbol("data")]: "This is important user data"  
}  
console.log(user.data);
```

Выше объявлен объект `user`, в котором есть два свойства. Свойство `login`, которое хранит логин пользователя и свойство `[Symbol("data")]` с описанием «`data`». Это свойство хранит важные данные об этом пользователе.

Для того, чтобы объявить ключ свойства символом необходимо:

1. Поставить квадратные скобки `[]`;
2. В квадратных скобках вызвать функцию `Symbol()`;
3. Функция `Symbol()` вернет уникальный результат который будет установлен как ключ для свойства.

Если вы попытаетесь узнать значение свойства `[Symbol("data")]` через такую запись «`user.data`» то в ответ мы увидим значение `undefined`, так как таким образом получить значение невозможно:

```
undefined
```

Рисунок 18

Также мы не увидим наличие такого свойства, если в консоль вывести все ключи через функции `Object.keys()` и `Object.getOwnPropertyNames()`

```
console.log(Object.keys(user));
console.log(Object.getOwnPropertyNames(user));
```

На экране появится такой результат:

```
▼ ["login"] 1
  0: "login"
  length: 1
  __proto__: Array(0)

▼ ["login"] 1
  0: "login"
  length: 1
  __proto__: Array(0)
```

Рисунок 19

Как видно из скриншота (рис. 19), обе функции возвращают только один ключ `login`, и при этом не отображают тот ключ, который содержит в себе значение символа.

Тоже происходит, если перебрать свойства объекта в цикле `for in`:

```
let user = {
  login: "user1",
  [Symbol("data")]: "This is important user data"
}
```

```
for (const key in user) {  
  console.log(key);  
}
```

При переборе свойств в цикле `for in` мы получаем тот-же результат, а именно то, что свойство символ не отображается.



*Рисунок 20*

Также ключи-символы не будут сериализованы в JSON при использовании `JSON.stringify()`.

Однако получить значение таких свойств все же можно. Как это сделать, показано в примере ниже:

```
let userData = Symbol("data");  
let user = {  
  login: "user1",  
  userData: "This is important user data"  
}  
  
console.log(user.userData);
```

Для получения доступа к свойствам-символа, нужно обратиться к нему по ссылке на символ. Для чего сначала создается переменная `userData`, которая хранит символ, после эта переменная указывается в качестве ключа для свойства объекта `user`. Далее происходит обращение к свойству-символу как к обычному свойству. На экране появится значение свойства:



This is important user data

Рисунок 21

Важно понимать, что `symbol` не создает скрытые поля и не обеспечивает безопасность в полном объеме. Скрытые свойства — это те свойства доступ, к которым можно получить только в самом объекте, и нельзя получить извне. К свойствам-символам все же можно получить доступ снаружи, хоть и сложнее чем к полностью открытым. Тип данных `symbol` необходим для обеспечения ошибкоустойчивости при работе со свойствами объекта — это можно увидеть из информации ниже.

Свойства символы приносят пользу при работе с сторонними библиотеками и фреймворками. Например, когда у нас есть объект из сторонней библиотеки, который мы используем и нам необходимо добавить к нему еще одно свойство. Однако напрямую делать это небезопасно, так как этот же объект используется в своей библиотеке, и добавленное свойство может нарушить ее работу. Например, когда свойства объекта перебираются в цикле `for in` и с каждым из свойств выполняются определенные действия. Выполнение этих действий со значением нового свойства может привести к ошибке. Для того чтобы этого не произошло, можно воспользоваться символами. Так как для того, чтобы обратиться к свойству-символу необходимо делать это по ссылке на символ, как это было в примере выше. Из-за того, что сторонняя библиотека не знает о символе, у нее нет возможности получить доступ к ней напрямую

или через цикл `for in` и функции `Object.keys()`, и `Object.GetOwnPropertyNames()`.

Также если добавлять свойство к стороннему объекту может возникнуть ситуация, что такое свойство уже существует, как например здесь:

```
let user = {  
    id: 1,  
    login: "user1",  
}  
  
console.log(user.id);  
  
user.id = 2;  
  
console.log(user.id);
```

Представим, что объект `user` это сторонний объект из другой библиотеки или фреймворка и нам необходимо к этому объекту добавить наш собственный `id`, который пригодится для нашей работы. У этого объекта уже есть свой `id`. Этот `id` может использоваться для внутренней работы библиотеки, в которой он объявлен. При выводе его на экран можно увидеть, что значение равно `1`. Предположим, что после происходит попытка добавления своего `id` со значением `2`, но так как свойство `id` у объекта `user` уже есть, происходит перезапись значения свойства. Теперь `id` объекта `user` равняется `2`.

<code>id = 1</code>
<code>id = 2</code>

Рисунок 22

Одним из решений этой проблемы является добавление нижнего подчеркивания или любого другого префикса в название свойства. В этом примере к ключу свойства добавляется нижнее подчеркивание:

```
let user = {  
  id: 1,  
  login: "user1",  
}  
  
user._id = 2;  
  
console.log(user.id);  
  
console.log(user._id);
```

К объекту `user` добавляется свойство `_id`. Оно не конфликтует с внутренним свойством `id` так как имеет другое имя. Это решает проблему конфликтов имен. Теперь на экране можно увидеть, что `id` и `_id` это разные свойства и имеют разные значения:



```
id = 1  
_id = 2
```

Рисунок 23

Однако, это решение не дает гарантий, что такого ключа для свойства нет, а также что внутренняя работа не будет подвержена ошибкам из-за нового свойства.

Использование символа в качестве ключа для свойства объекта гарантирует что такого же ключа нет, так как его значение уникально.

Вот как это можно сделать:

```
let id = Symbol("id")
let user = {
  id: 1,
  login: "user1",
}

user[id] = 2;
console.log('id = ${user.id}');
console.log('symbol id = ${user[id]}');
```

В примере к объекту `user` добавляется новое свойство-символ, с описанием «`id`». Обратите внимание что, добавлять свойство-символ нужно с помощью квадратных скобок, так как использование оператора точки приведет к изменению значения уже существующего `id`.

Теперь у объекта `user` есть два свойства `id`, первое `id`, которое было у него изначально и второе, которое мы добавили к нему. Так как символ возвращает уникальное значение, новое свойство гарантировано не будет конфликтовать уже с существующими свойствами этого объекта.

Символы как ключи для свойства объектов позволяют не беспокоиться о том, что добавление нового свойства к стороннему объекту может привести к ошибке.

Другим способом создания символа является вызов функции `Symbol.for()`. Главным отличием этого способа создания от создания через `Symbol()` заключается в том что, функция `Symbol.for()` сначала проверяет существует ли символ с таким же описанием, что и новый символ и если он существует, то возвращается этот символ, а если нет создается новый.



Это можно увидеть в этом примере:

```
let name = Symbol.for("name");  
let name2 = Symbol.for("name");  
  
console.log(name === name2);
```

В примере создаются две переменные, через функцию `Symbol.for()`, с одинаковым описанием «`name`». Далее в консоль выводится результат сравнения на идентичность двух переменных. Мы можем увидеть такой результат:

A screenshot of a console window showing the output of the code. The word "true" is displayed in a blue monospace font within a white rectangular box with a thin grey border.

*Рисунок 24*

Из этого скриншота видно, что обе переменные имеют одинаковое значение.

Так произошло, потому что при попытке создания второго символа с описанием «`name`» в глобальном реестре, был найден уже существующий символ с таким же описанием. Именно поэтому функция `Symbol.for()` вернула найденный символ в качестве результата.

Когда создается символ таким способом, он заносится в глобальный реестр символов и к нему можно получить доступ в любом месте программы.

## Глобальные символы

**Глобальный реестр** — это место куда заносятся символы, к которым можно получить доступ в любой части программы.

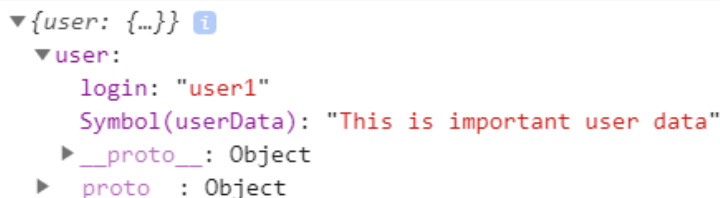
Символы, которые заносятся в глобальный реестр называются глобальными символами.

Например, когда есть один объект и нам нужно получить доступ к его свойству-символу. Если символ создан через `Symbol()` найти его невозможно так как обращаться к нему нужно через ссылку на символ. Когда нужно получать доступ к свойству-символу его необходимо создать через функцию `Symbol.for()`, для того чтобы к нему можно было обратиться.

Давайте рассмотрим это на примере:

```
let user = {  
  login: "user1",  
  [Symbol("userData")]: "This is important user data"  
};  
console.log(user);
```

Сначала объявлен объект `user`, который содержит свойство `login` и свойство-символ, обозначающее данные пользователя. Далее объект отображается на экране. Его значение можно увидеть открыв консоль разработчика:



```
▼ {user: {...}} ⓘ  
  ▼ user:  
    login: "user1"  
    Symbol(userData): "This is important user data"  
    ► __proto__: Object  
    ► __proto__: Object
```

Рисунок 25

Выше видно, что объект `user` имеет свойство-символ `userData`, но у нас нет возможности его использовать. Для

этого нужно изменить объект `user`, а именно свойство `userData`, создав его с помощью `Symbol.for()`. Это даст возможность найти свойство и использовать его.

Вот как теперь будет выглядеть пример:

```
let user = {
  login: "user1",
  [Symbol.for("userData ")]: "This is important
                             user data"
};

let userData = Symbol.for("userData");
console.log(user[userData]);
```

Свойство `userData` теперь создается с помощью функции `Symbol.for()`. Далее объявляется переменная `userData`, в которую функция `Symbol.for("userData")` возвращает ссылку на найденный символ из глобального регистра. Далее по этой ссылке происходит обращение к свойству `userData`. После на экран выводится свойство объекта через ссылку на символ. Обратите внимание на то, что обращение происходит через квадратные скобки.

На экране можно увидеть значение свойства `userData`:



```
This is important user data
```

Рисунок 26

Таким образом к символам, которые заносятся в глобальный реестр, можно получить доступ через переменную, которая содержит ссылку на символ, найденный в глобальном реестре, когда нам это необходимо.

Для глобальных символов есть метод `keyFor()`, который возвращает описание символа. Вот как это выглядит:

```
let symbol = Symbol.for("name");  
let symbol2 = Symbol("surname");  
  
console.log(Symbol.keyFor(symbol));  
console.log(Symbol.keyFor(symbol2));
```

В примере создаются два символа, первый через `Symbol.for("name")` — это значит, что символ будет глобальным, а второй создается через `Symbol("surname")`, этот символ не будет создан в глобальном реестре. При вызове `console.log()`, функция `Symbol.keyFor()` вернет описание для `symbol` и на экране появиться «name», так как символ глобальный и он будет найден в глобальном регистре. Следующий `console.log()` выводит `undefined`, потому что символ `symbol2` не является глобальным, так как он создан с помощью `Symbol()`. Поэтому использовать функцию `keyFor()` с не глобальными символами бессмысленно.

Значения символов из примера выше можно увидеть на этом скриншоте:

name
undefined

Рисунок 27

Есть еще один метод с помощью которого можно получить список всех символьных свойств — `Object.getPrototypeOfSymbols()`.

`Object.getOwnPropertySymbols()` — возвращает массив символьных свойств объекта, который был передан в качестве аргумента.

Пример как использовать функцию `getOwnPropertySymbols()`:

```
let user = {  
  login: "user1",  
  [Symbol("data")]: "This is important user data"  
}  
  
let propsSymbolUser =  
  Object.getOwnPropertySymbols(user);  
console.log(propsSymbolUser);
```

Объявлен объект `user`, который содержит символьное-свойство. Далее объявлена переменная `propsSymbolUser`, в ней будет храниться массив символьных свойств объекта `user`, которые вернет функция `Object.getOwnPropertySymbols()`. После, значение `propsSymbolUser` выводится на экран. Вот результат:

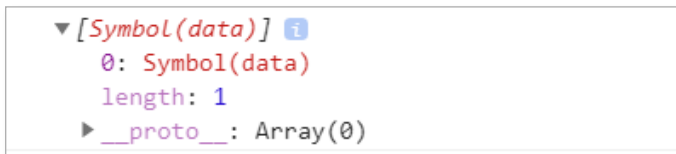


Рисунок 28

Переменная `propsSymbolUser` содержит массив из одного элемента, а именно свойства-символа.

# Стрелочные функции

ES6 принес в JavaScript новый способ написания функций, с помощью функций стрелок.

Функции-стрелки, также их называют **arrow function**, позволяют описывать функции более кратко из-за нового синтаксиса и обладают полезными особенностями при работе с **this**. Сначала поговорим про синтаксис.

Вот как выглядит базовый синтаксис стрелочной функции:

```
let greeting = (name) => { return 'Hello ${name}';  
  console.log(greeting("Alex"));
```

Для того чтобы создать функцию-стрелку, нужно указать список параметров в круглых скобках, далее следует стрелка, которая составлена из символов «=» и «>», после открываются фигурные скобки, в них находится логика функции. Функция в примере возвращает строку с приветствием.

Стрелочные функции создаются с помощью **function expression** (*функциональные выражения*). Это значит, что стрелочную функцию можно присвоить переменной и использовать ее как функцию. Таким образом, чтобы вызвать функцию-стрелку нужно обратиться к переменной, которой была присвоена функция, передав необходимые параметры. Также, стрелочные функции являются анонимными, т.е. такие функции не имеют своего имени.

Теперь переменная **greeting** содержит функцию-стрелку. Ниже в коде примера переменная **greeting** вызывается как

функция, передав в нее нужный параметр. В результате вызова функции на экране можно увидеть приветствие:



hello Alex

Рисунок 29

Функции-стрелки обладают компактным синтаксисом по сравнению с Function Declaration и Function Expression и сейчас мы это увидим:

```
function Add(num1, num2) {  
    let result = num1 + num2;  
    return result;  
}  
  
let Subtract = function(num1, num2) {  
    let result = num1 - num2;  
    return result;  
}  
  
console.log(add(1, 4));  
console.log(subtract(6, 3));
```

Выше объявляются две функции, первая Function Declaration, а вторая Function Expression. Обе функции занимают по 4 строки, хоть и являются достаточно простыми, и содержат немного логики. Они принимают два числа и выполняют с ними одну операцию. Результат работы двух функций появится на экран:



5



3

Рисунок 30

А точно ли нужно писать столько кода для небольшого набора действий? Однозначно нет, так как есть стрелочные функции. С их помощью код выглядит компактнее и занимает меньше места.

Вот как будет выглядеть пример выше, если использовать стрелочные функции:

```
let add = (num1, num2) => num1 + num2;
let subtract = (num1, num2) => num1 - num2;

console.log(add(1, 4));
console.log(subtract(6, 3));
```

Как видно из примера, обе функции занимают по одной строке, и выполняют такие же действия, что и функции из предыдущего примера. Результат:

5

3

*Рисунок 31*

Синтаксис стрелочных функций можно еще больше упростить.

Когда нужно передавать в функцию только один параметр, то круглые скобки можно не использовать. Вот так:

```
let squaring = num => { return num * num};
```

Если необходимо выполнить только одну операцию, допускается пропустить фигурные скобки и оператор **return** :

```
let squaring = num => num * num;
```



В теле этой функции выполняется одна операция умножения что позволит возвести число в степень. После завершения операции ее результат будет возвращен, как результат функции.

Если аргументы не нужны, нужно поставить круглые скобки, но ничего не передавать:

```
let sayHello = () => alert("Hello!");
```

Также, если использовать стрелочные функции для возврата объектного литерала необходимо использовать такой синтаксис:

```
let getPerson = (name, age) => ({ name: name, age: age });
```

В обычном синтаксисе, в фигурных скобках идет тело функции, в случае с литеральным объектом в фигурных скобках определяется объект. Для того чтобы интерпретатор понял, что здесь возвращается литеральный объект необходимо его обернуть в круглые скобки.

Функция выше принимает два аргумента, а именно имя и возврат человека, а после возвращает объект с свойствами `name` и `age`.

## Итоги

Стрелочные функции имеют компактный синтаксис по сравнению с функциями Function Declaration и Function Expression.

- **Function Declaration и Function Expression:** нужно всегда использовать полный синтаксис функции.

Так будет выглядеть Function Declaration функция:

```
function Add(num1, num2){
  let result = num1 + num2;
  return result;
}
```

### Стрелочные функции:

- Круглые скобки можно не использовать, если нет передаваемых параметров.
- Не нужны фигурные скобки и оператор `return`, если тело функции содержит одну операцию.

Так будет выглядеть стрелочная функция:

```
let add = (num1, num2) => num1 + num2;
```

Разобравшись с синтаксисом, пора узнать для чего используются функции-стрелки.

### Стрелочные функции как callback функции

Стрелочные функции часто используются в качестве callback-функций, например в методах массива: `map()`, `filter()`, `forEach` и т.д.

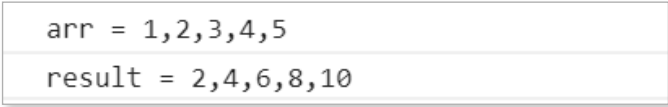
В этом примере можно увидеть, как использовать функцию-стрелку с методами массива.

```
let arr = [1, 2, 3, 4, 5];
console.log('arr = ${arr}');

let result = arr.map((el) => { return el * 2 });
console.log('result = ${result}');
```

Сначала создается массив `arr` и отображается на экране. Далее используется метод `map()` который позволяет вы-

звать переданную функцию для каждого элемента массива и возвращает новый массив из результатов вызовов этой функции. Массив `arr` вызывает функцию `map()`, в которую передается стрелочная функция. В этой функции каждый элемент массива умножается на `2` и возвращается из функции для формирования нового массива. После того, как метод `map()` пройдет по всем элементам массива `arr` и выполнит операцию умножения, он вернет новый массив, который будет присвоен в переменную `result`. Результат работы можно увидеть на этом скриншоте:



```
arr = 1,2,3,4,5  
result = 2,4,6,8,10
```

Рисунок 32

В первой строке показан исходный массив, а во второй новый массив.

Каждый элемент массива был передан в функцию-стрелку, с ним была выполнена операция умножения, а после результат этой операции возвращен из функции для формирования нового массива.

Так выглядит пример выше, если использовать Function Expression функции вместо стрелочных функций:

```
let arr = [1, 2, 3, 4, 5];  
console.log('arr = ${arr}');  
  
let result = arr.map(function(el) {  
    return el * 2;  
});  
console.log('result = ${result}');
```

На экране появится такой же результат:

```
arr = 1,2,3,4,5  
result = 2,4,6,8,10
```

*Рисунок 33*

Однако использование стрелочных функций является более удобным способом описать подобные действия.

Давайте рассмотрим еще один пример. Предположим, что мы хотим получить из массива все четные числа. Вот как это можно сделать при помощи функций-стрелок:

```
let arr = [1, 2, 3, 4, 5];  
console.log('arr = ${arr}');  
  
let result = arr.filter((el) => el % 2 === 0);  
console.log('result = ${result}');
```

Есть уже инициализированный числами массив, который отображается на экран. После, массив `arr` вызывает метод `filter()`, этот метод принимает стрелочную функцию. Метод `filter()` возвращает новый массив, в котором будет только те элементы, для которых передаваемая функция вернет `true`, с помощью необходимой проверки. Внутри функции-стрелки происходит проверка, является ли элемент массива четным с помощью оператора «остаток от деления». Если результат выражения вернет не ноль это значит, что число не является четным, иначе число четное. Таким способом в переменную `result` присваивается результат выполнения метода `filter()`, а после выводиться на экран. Результат выполнения кода:

```
arr = 1,2,3,4,5  
result = 2,4
```

Рисунок 34

В результате переменная **result** содержит все четные элементы исходного массива.

## This в стрелочных функциях

Очень важной особенностью стрелочных функций является то, что они не имеют **this**. Если быть точнее стрелочные функции берут **this** из внешнего контекста.

Для начала вспомним, что такое **this**?

Любая функция содержит в себе **this**, даже если она не является методом объекта. **this** — ссылка на объект, которая используется для доступа к данным объекта внутри функции.

Значение **this** определяется во время вызова функции и зависит от контекста. Под контекстом подразумевается то, как вызывается функция.

Разберемся со всем по порядку.

### Глобальный контекст

В глобальном контексте **this** ссылается на глобальный объект. Это значит, что при обращении к ключевому слову **this** в глобальной области видимости его значение будет объект **window**.

Это можно увидеть, если вывести на экран **this** из глобальной области видимости:

```
console.log(this);
```

В консоли разработчика появится объект `window`:

```

▼ Window {parent: Window, opener: null, top: Window, length: 0, frames: Wind
w, ...} ⓘ
  ▶ alert: f alert()
  ▶ applicationCache: ApplicationCache {status: 0, oncached: null, onchecking: ...
  ▶ atob: f atob()
  ▶ blur: f blur()
  ▶ btoa: f btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
  ▶ cancelIdleCallback: f cancelIdleCallback()
  ▶ captureEvents: f captureEvents()
  ▶ chrome: {loadTimes: f, csi: f, search: f}
  ▶ clearInterval: f clearInterval()
  ▶ clearTimeout: f clearTimeout()
  ▶ clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor...
  ▶ close: f close()
    closed: false
  ▶ confirm: f confirm()
  ▶ createImageBitmap: f createImageBitmap()
  ▶ crypto: Crypto {subtle: SubtleCrypto}
  ▶ customElements: CustomElementRegistry {}
    defaultStatus: ""
    defaultstatus: ""
    devicePixelRatio: 1
  ▶ document: document
  ▶ external: External {}
  ▶ fetch: f fetch()

```

Рисунок 35

Иными словами, контекст для `this` в глобальной области — это глобальный объект. В браузере глобальным объектом будет объект `window`, в *Node.js* глобальный объект — объект `global`.

### Контекст функции

Значение `this` в Function Declaration и Function Expression функциях зависит от того, как была вызвана функция, а не где была объявлена.


## Обычные функции

В глобальной области видимости `this` обычных функций будет ссылаться на глобальный объект.

Мы можем это проверить, создав функцию и отобразить на экран значение, которое хранится в `this`:

```
function printThis() {
    console.log(this);
}
printThis();
```

Результат будет таким:



```
Window {parent: Window, opener: null, top: Window, length: 0, frames: Wind
w, ...}
  ▶ alert: f alert()
  ▶ applicationCache: ApplicationCache {status: 0, oncached: null, onchecking: ...
  ▶ atob: f atob()
  ▶ blur: f blur()
  ▶ btoa: f btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
  ▶ cancelIdleCallback: f cancelIdleCallback()
  ▶ captureEvents: f captureEvents()
  ▶ chrome: {loadTimes: f, csi: f, search: f}
  ▶ clearInterval: f clearInterval()
  ▶ clearTimeout: f clearTimeout()
  ▶ clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor...
  ▶ close: f close()
    closed: false
  ▶ confirm: f confirm()
  ▶ createImageBitmap: f createImageBitmap()
  ▶ crypto: Crypto {subtle: SubtleCrypto}
  ▶ customElements: CustomElementRegistry {}
    defaultStatus: ""
    defaultstatus: ""
    devicePixelRatio: 1
  ▶ document: document
  ▶ external: External {}
  ▶ fetch: f fetch()
```

Рисунок 36

Так как `printThis()` это обычная функция и она вызывается не от какого либо объекта, то `this` внутри тела функции будет ссылаться на `windows`.

## Методы объекта

Когда вызывается метод объекта — это значит, что `this` будет ссылаться на объект, который вызвал функцию.

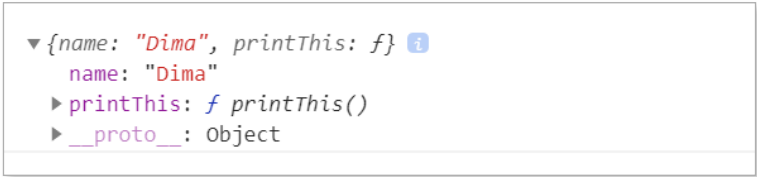
В следующем примере показано, что `this` метода `printThis()` ссылается на объект, который вызвал этот метод:

```
let person = {
  name: "Dima",
  printThis() {
    console.log(this);
  }
}

person.printThis();
```

В примере создан объект `person`. У этого объекта есть свойство `name` и метод `printThis()`. Ниже объект `person` вызывает метод `printThis()`, который выводит на экран `this` этого метода.

Такое значение `this` можно увидеть в консоли разработчика:



```
▼ {name: "Dima", printThis: f} ⓘ
  name: "Dima"
  ► printThis: f printThis()
  ► __proto__: Object
```

Рисунок 37



Контекстом для метода является объект, который вызвал этот метод. В этом случае это объект `person`, а значит, что `this` является ссылкой на `person`.

Очень важно знать, что `this` определяется именно от того, как был вызван метод, а не где он был объявлен.

Рассмотрим пример:

```
let personJames = {
  name: "James",
  printThis: function() {
    console.log(this);
  }
}

let personAnya = {
  name: "Anya"
}

personAnya.printThis = personJames.printThis;

personJames.printThis();
personAnya.printThis();
```

Объявляется объект `personJames` у него есть свойство `name` и метод `printThis()`. Метод `printThis()` отображает `this` метода с помощью `console.log()`. Так же, есть еще один объект `personAnya`, который содержит только свойство `name`. Представим, что теперь нужно использовать метод `printThis()` для объекта `personAnya`. Тогда мы присваиваем метод `printThis()` из объекта `personJames` в объект `personAnya`. После вызывается `printThis()` сначала у объекта `personJames` а потом у `personAnya`. Открыв консоль появится такой результат:

```

▼ {name: "James", printThis: f} ⓘ
  name: "James"
  ► printThis: f ()
  ► __proto__: Object

▼ {name: "Anya", printThis: f} ⓘ
  name: "Anya"
  ► printThis: f ()
  ► __proto__: Object

```

Рисунок 38

Из скриншота видно, что в первый вызов метода `printThis()` — `this` для этого метода была ссылка на объект `personDima`, а во второй раз — `this` являлся ссылкой на объект `personAnya`. На момент первого вызова `printThis()`, контекстом является объект `personDima`, при повторном вызове контекстом будет объект `personAnya`. Хотя метод `printThis()` изначально объявлялся в объекте `personDima`, но это не помешало вызвать его для объекта `personAnya`. Из этого следует вывод, что `this` определяется не от того в каком объекте был объявлен метод, а от того какой именно объект вызвал его.

Даже если функцию объявить как Function Declaration вне объекта, а позже добавить ее в объект то `this` определяется таким же образом:

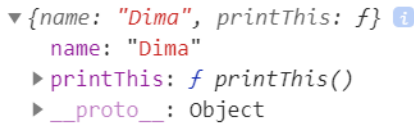
```

function printThis() {
  console.log(this);
}
let personDima = {
  name: "Dima",
}
personDima.printThis = printThis;
personDima.printThis();

```

В начале объявляется обычная Function Declaration функция `printThis()`. После объявляется объект `personDima`. Далее, в `personDima`, в свойство `printThis` устанавливается функция `printThis()`, которая была объявлена выше. А после `printThis()` вызывается от объекта `personDima`.

В результате можно увидеть:



```
▼ {name: "Dima", printThis: f} ⓘ
  name: "Dima"
  ► printThis: f printThis()
  ► __proto__: Object
```

Рисунок 39

Из скриншота понятно, что `this` метода `printThis()` является ссылка на объект `personDima`.

Теперь разобравшись с ключевым словом `this`, можно приступить к изучению особенностей работы с `this` в стрелочных функциях. Как уже было сказано, в Function Declaration и Function Expression функциях значение `this` это тот объект, который вызвал функцию. В функциях-стрелках `this` это `this` из кода, который содержит эту функцию.

Рассмотрим особенности использования `this` на примерах:

```
let studentIvan = {
  nameStudent: "Иван Иванов",
  disciplines: ["Программирование",
               "Машиностроение", "Английский"],
  showDisciplines : function() {
    this.disciplines.forEach(function(el) {
```

```

        console.log(`${this.nameStudent}
                    в университете изучает:
                    ${el}');
    })
}
}

studentIvan.showDisciplines();

```

В этом примере создается объект `studentIvan`. Объект `studentIvan` содержит свойство `nameStudent`, которое обозначает имя студента. А также массив `disciplines`, он содержит набор дисциплин. Метод `showDisciplines()` необходим для того, чтобы вывести на экран сообщение о том какие дисциплины посещает `studentIvan`. Внутри метода `showDisciplines()` перебираются все элементы массива `disciplines` функцией `forEach()`. Функция `forEach()`, в свою очередь, принимает callback-функцию, в нашем случае это обычная Function Expression функция, в которую передается итерируемый элемент массива. В передаваемой функции отображается строка о том, какие дисциплины посещает студент, обращаясь к свойству `nameStudent` через `this`, а также к элементу массива. После, вызывается метод `showDisciplines()`.

Результат можно увидеть на этом скриншоте:

undefined в университете изучает: Программирование
undefined в университете изучает: Машиностроение
undefined в университете изучает: Английский

Рисунок 40

Выше видно, что при отображении свойства `this.nameStudent` его значением является `undefined`. Это происходит из-за потери `this` внутри передаваемой callback-функции, так как она не вызывается от объекта `studentIvan`, а передается в `forEach()`, где и будет вызвана.

Пример можно изменить для наглядности, заменив `forEach()` на обычный цикл `for`. Такой код эквивалент коду из примера выше:

```
let studentIvan = {
  nameStudent: "Иван Иванов",
  disciplines: ["Программирование",
               "Машиностроение", "Английский"],
  showDisciplines : function(){
    let printDiscipline = function(el){
      console.log(`${this.nameStudent}
                  в университете изучает:
                  ${el}`);
    }

    for (let index = 0;
         index < this.disciplines.length;
         index++) {
      printDiscipline(this.disciplines[index]);
    }
  }
}

studentIvan.showDisciplines();
```

В `showDisciplines` объявляется Function Expression функция `printDiscipline()`, а также место метода `forEach()` используется цикл `for()`. В цикле вызывается функция `printDiscipline()` в нее передается элемент массива. В `print-`

`Discipline()` все также происходит обращение к свойству `nameStudent` через `this`, для отображения на экран.

Результат работы не изменился:

undefined в университете изучает: Программирование
undefined в университете изучает: Машиностроение
undefined в университете изучает: Английский

*Рисунок 41*

Значение свойства объекта `nameStudent` все также равно `undefined`. Однако теперь наглядно видно, что `printDiscipline()` вызывается не от объекта `studentIvan`. Как мы уже знаем, `this` зависит не от того, где объявлена функция, а от объекта, который вызвал функцию. В этом случае `printDiscipline()` вызывается как обычная функция, поэтому `this` не является ссылкой на `studentIvan`. Однако, эту проблему можно решить несколькими способами.

Первый способ — это использование переменной, которая сохраняет значение `this`. Вот как это выглядит:

```
let studentIvan = {
  nameStudent: "Иван Иванов",
  disciplines: ["Программирование",
               "Машиностроение", "Английский"],
  showDisciplines : function() {
    let that = this;
    this.disciplines.forEach(function(el) {
      console.log(`${that.nameStudent}
                  в университете изучает: ${el}`);
    })
  }
}
studentIvan.showDisciplines();
```

В функции `showDisciplines()` значение `this` сохраняется в переменную `that`. Теперь в теле callback-функции вместо `this` используется переменная `that`. Из-за замыкания функция имеет доступ к переменной `that`, в которой хранит ссылку на `this` объекта `studentIvan`.

Посмотрим на результат:

Иван Иванов в университете изучает: Программирование
Иван Иванов в университете изучает: Машиностроение
Иван Иванов в университете изучает: Английский

Рисунок 42

Теперь, когда в переменной `that` хранится `this` объекта `studentIvan` мы можем получить доступ к нужным свойствам и методам. Свойство `nameStudent` возвращает свое значение и сообщение в консоли разработчика выводится верно. Это видно по скриншоту выше.

Этот подход использовали ранее до появления ES6. С появлением стрелочных функций решать проблему с `this` стало проще. Давайте перепишем тот же пример, но с использованием стрелочных функций.

Вот как будет выглядеть измененный пример:

```
let studentIvan = {
  nameStudent: "Иван Иванов",
  disciplines: ["Программирование",
               "Машиностроение", "Английский"],
  showDisciplines : function() {
    this.disciplines.forEach((el) => {
      console.log(`${this.nameStudent}
                  в университете изучает:
                  ${el}`);
    });
  }
}
```

```

    })
  }
}

studentIvan.showDisciplines();

```

В функции `showDisciplines()` в `forEach()`, вместо функции Function Expression передаем стрелочную функцию. И все работает. Здесь можно убедиться в этом:

Иван Иванов в университете изучает: Программирование
Иван Иванов в университете изучает: Машиностроение
Иван Иванов в университете изучает: Английский

Рисунок 43

Результат верный, и сообщение отображается верно.

Это работает, потому что в стрелочных функциях нет собственного `this`. Он берется из той области, в которой функция-стрелка объявлена. Это наглядно показано на этом изображении:

```

let studentIvan = {
  nameStudent: "Иван Иванов",
  disciplines: ["Программирование", "Машиностроение", "Английский"],
  showDisciplines: function(){
    this.disciplines.forEach(
      (el) => {
        console.log(`${this.nameStudent} в университете изучает: ${el}`);
      }
    )
  }
}

studentIvan.showDisciplines();

```

Рисунок 44



Функция стрелка объявлена в методе `showDisciplines()`, соответственно `this` берется из этой функции, а точнее из внешнего контекста. **Внешний контекст** — это место, где объявлена функция, из него будет заимствоваться значение `this`. Из-за того, что `this` берется из внешнего контекста он сохраняется во вложенной функции.

## Стрелочные функции как методы объекта

Есть ситуация, в которой использование стрелочных-функций приводит к проблеме. Проблема появляется, когда стрелочная функция используется как метод объекта.

Рассмотрим это на примере:

```
let person = {
  myName: "Alex",
  greeting: () => {
    console.log('Hello, my name is
                ${this.myName}')
  }
}

person.greeting();
```

Есть объект `person`, он имеет свойство `myName` с значением «`Alex`», а также метод `greeting` этот метода объявлен, как стрелочная функция. Внутри метода `greeting` в консоль выводится приветствие и свойство `myName`.

Если зайти в консоль разработчика мы увидим такое сообщение:



```
Hello, my name is undefined
```

Рисунок 45

На экран выведется приветствие, но на месте значения свойства `myName` — `undefined`.

Так происходит потому, что, как мы уже знаем, стрелочные функции берут `this` из внешнего контекста. Таким образом, хоть стрелочная функция и является методом объекта `person`, значение `this` в ней будет не ссылка на этот объект, а внешний контекст.

На изображении показано, что `this` берется из окружающего контекста:

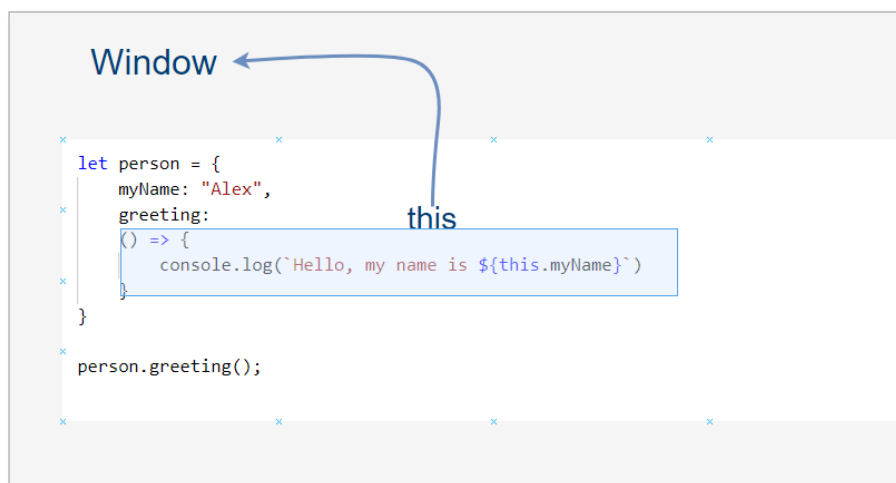


Рисунок 46

На рисунке видим, что `this` внутри стрелочной функции ссылается на `window`.

Так как `this` ссылается на `window`, при обращении к свойству `myName` — возвращается `undefined`, из-за того, что такого свойства в объекте нет.

Это проблема появилась из-за того, что методом объекта является стрелочная функция.

## Стрелочные функции и методы `call()`, `apply()` и `bind()`

Также со стрелочными функциями нельзя использовать методы: `call()`, `apply()` и `bind()`.

Например:

```
let person = {
  myName: "Alex",
  greeting: () => {
    console.log('Hello, my name is ${this.myName}')
  }
}

person.greeting();
```

Вначале создается объект `person` со свойством `myName` и методом `greeting()`, который является стрелочной функцией. Метод `greeting()` выводит сообщение с приветствием. Ниже объект `person` вызывает метод `greeting()` и на экране появится вот такое сообщение:



Hello, my name is undefined

Рисунок 47

Такая же проблема возникла в предыдущем примере, из-за того, что `this` для метода берется из внешнего контекста. Можно предположить, что `this` для стрелочной функции можно изменить путем его привязки через методы `bind()`, `apply()` и `call()`.

Давайте попробуем это сделать:

```
let person = {
  myName: "Alex",
```

```

    greeting: () => {
      console.log('Hello, my name is ${this.myName}')
    }
  }
  person.greeting.call(person);

```

Мы изменили вызов `greeting()`, используя метод `call()` и привязав `this` к объекту `person`, который передали в качестве параметра. Однако в консоли появляется такое-же сообщение:



```

Hello, my name is undefined

```

Рисунок 48

Как уже говорилось, `this` берется из внешнего контекста, однако изменить это никак нельзя, даже применив методы `bind()`, `apply()` и `call()`. Поэтому результат остался прежним.

### Стрелочные функции, как конструктор

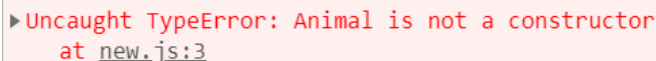
Использование функций-стрелок в качестве конструкторов не допускается.

```

let Animal = (name) => {name: name };
let cat = new Animal("Lucky");

```

Например, если попытаться выполнить этот код, в консоли появится такая ошибка:



```

Uncaught TypeError: Animal is not a constructor
at new.js:3

```

Рисунок 49

## Стрелочные функции и *arguments*

В функциях Function Declaration и Function Expression присутствует объект *arguments*, он хранит список аргументов, которые были переданы при вызове функции. Объект *arguments* представляется в виде псевдомассива.

Стрелочные функции не имеют своего псевдомассива *arguments*. Вместо этого *arguments* берется из внешнего контекста.

Например:

```
function greeting(name){  
  let printGreeting = () => {return  
                                'Hello ${arguments[0]}!'}  
  console.log(printGreeting());  
}  
  
greeting("Alex");
```

Функция *greeting()* будет выводить приветствие на экран. В нее передается имя, которое будет использоваться в стрелочной функции. В функции объявляется переменная *printGreeting*, она будет содержать функцию-стрелку, которая возвращает строку с приветствием. К передаваемому аргументу *name* в функции-стрелке, обращаемся через объект *arguments* по нулевому индексу, т.е. к первому аргументу. После вызываем стрелочную функцию через переменную *printGreeting*.

Такой результат можно увидеть, открыв консоль:



```
Hello Alex!
```

Рисунок 50

Объект `arguments` берется из внешнего контекста, т.е. из места, где объявлена функция. Таким образом можно обратиться к `arguments`.

# Классы

ES6 принес очень много нововведений. Среди них есть такое понятие как `class`.

Ключевое слово `class` — предоставляет простой и понятный способ объявления и создания объектов.

Прежде чем перейти к изучению конструкции `class`, давайте разберемся каким образом происходит создание объектов без использования классов.

Часто нужно создать множество объектов с похожей структурой, это может быть, например товары. Для таких целей использовались функции-конструкторы. **Функция конструктор** — это функция, которая вызывается с помощью ключевого слова `new` и инициализирует новый объект свойствами.

Давайте создадим функцию-конструктор, которая позволит создавать товары.

Вот как это выглядит:

```
function Product(name, price){
  this.name = name;
  this.price = price;

  this.printInfo = function(){
    console.log('Name: ${this.name},
               Price: ${this.price}');
  }
}

let phone = new Product("iPhone 10", 660);
```

```
console.log(phone.name);  
console.log(phone.price);
```

В примере объявлена функция-конструктор `Product()`. Внутри нее устанавливаются необходимые свойства, а именно: `name` — название товара, `price` — цена. Каждый товар может вывести информацию о себе, поэтому внутри функции-конструктора объявляется метод `printInfo()`. После создается объект `phone`, он содержит свойства и методы, которые были определены в функции-конструкторе `Product()`. На экран выводим значение свойств объекта — `name` и `price`. Результат вывода показан на рисунке 51.



Рисунок 51

Таким образом функции-конструкторы позволяют создавать множество однотипных объектов. До введения спецификации ECMAScript 2015 создать множество объектов можно было только таким способом. Однако, теперь создавать похожие по структуре объекты можно с помощью классов.

Введение классов в JS было сделано для того, чтобы объекты можно было создавать более очевидным способом. Также классы принесли дополнительные возможности, которые предназначены для упрощения работы с наследованием.



Давайте перепишем пример выше и вместо функции-конструктора будем использовать класс.

Для того чтобы объявить класс необходимо указать ключевое слово `class`, его имя и открыть фигурные скобки. Вот так:

```
class Product {  
}
```

Свойства для объекта устанавливаются в конструкторе. Конструктор определяется в теле класса с помощью ключевого слова `constructor`. Внутри конструктора определяются свойства этого класса. Например:

```
class Product {  
    constructor(name, price){  
        this.name = name;  
        this.price = price;  
    }  
}
```

Для того чтобы определить свойства в конструкторе необходимо использовать `this` перед ключом свойства.

Теперь все объекты, которые созданы с помощью класса `Product` будут иметь свойства `name` и `price`.

Методы, как и конструктор определяются в теле класса и выглядят иначе, по сравнению с методами в литеральных объектах:

```
class Product{  
    constructor(name, price){  
        this.name = name;  
    }  
}
```

```

        this.price = price;
    }

    printInfo = function(){
        console.log('Name: ${this.name},
                    Price: ${this.price}');
    }
}

```

Из примера видно, что для того чтобы объявить метод нужно просто указать имя метода, список передаваемых параметров в круглых скобках, а после тело функции.

Теперь, когда класс объявлен, нужно создать объект с его помощью. Заметьте, что никаких объектов еще не было создано, но теперь это возможно сделать.

Объекты с помощью классов создаются так же, как и при их создании, с помощью функций конструкторов:

```
let phone = new Product("iPhone 10", 660);
```

Объекты, которые были созданы с помощью классов называются — экземплярами класса. Когда создается экземпляр класса так же, как и с помощью функции конструктора происходят такие действия:

1. Вызывается функция `constructor()`.
2. Создается новый пустой объект.
3. В новый объект присваиваются все свойства, которые определены в методе `constructor()`. Ключевое слово `this` перед названием свойства используется для того, чтобы свойства принадлежали новому объекту.
4. Объект неявно возвращается из конструктора.

Как мы уже знаем классы это новая синтаксическая конструкция. Однако это не совсем так. В JavaScript класс — это функция.

Это можно проверить если вызывать оператор `typeof` для класса `Product`:

```
function
```

Рисунок 52

Когда объявляется класс, на самом деле, объявляется функция-конструктор. Иными словами, функция конструктор `Product`, которая была описана в начале раздела эквивалента классу `Product`. Однако, различия между ними все же есть, о них подробнее поговорим далее.

Классы можно объявить с помощью `class declaration` или `class expression`.

Синтаксис объявления классов как `class declaration` (*объявление класса*) мы уже видели. Для этого нужно использовать ключевое слово `class` и указать имя класса:

```
class Person{  
}
```

Изначально можно сделать вывод, что классы `class declaration` всплывают так же, как и функции объявленные через `function declaration`.

Давайте проверим это:

```
let phone = new Product("iPhone 10", 660);  
class Product{  
  constructor(name, price){
```

```

        this.name = name;
        this.price = price;
    }

    printInfo() {
        console.log('Name: ${this.name},
                    Price: ${this.price}');
    }
}

```

В этом примере, сначала создается объект класса **Product**, а после объявляется сам класс. Однако, если открыть консоль разработчика, можно увидеть такую ошибку:

```

► Uncaught ReferenceError: Cannot access 'Product'
  before initialization
    at class.js:1

```

Рисунок 53

Мы помним, что функция, объявленная с помощью **function declaration** может быть использована даже выше ее объявления из-за всплытия. Однако **class declaration** не обладает такой возможностью. Как видно из скриншота, использовать класс до его объявления нельзя, это приводит к ошибке.

Второй способ объявления класса — это **class expression** (*выражение класса*). Это значит, что класс можно присвоить в переменную. Пример:

```

let Product = class {
    constructor(name, price){
        this.name = name;
    }
}

```

```
        this.price = price;
    }

    printInfo() {
        console.log('Name: ${this.name},
                    Price: ${this.price}');
    }
};
```

Здесь создается анонимный класс, который потом мы присваиваем в переменную **Product**.

## Свойства-аксессоры

Значения свойств объекта могут часто меняться, например, товару в интернет-магазине нужно изменить цену в связи со скидкой.

Есть несколько способов изменить значение свойства. Первый способ это напрямую обратиться к свойству объекта и присвоить ему значение. Однако есть большая вероятность, что данные будут в некорректном виде. Например, свойству **price** можно присвоить значение меньше нуля или ноль. Это некорректно, так как цена товара должна быть больше нуля, и при дальнейшей работе это может привести к непредвиденным последствиям.

Не стоит обращаться напрямую к свойствам, которые могут нарушить работу всего кода, это небезопасно. К примеру, обращаться и изменять цену товара напрямую не лучший вариант. Если его значение случайно окажется ниже нуля, то при оформлении заказа подсчеты о стоимости всего заказа будут не верны. Поэтому есть договоренность между программистами, о том, что свойства

должны обозначаться нижним подчеркиванием в начале его имени. Такие свойства не нужно использовать напрямую. Для того, чтобы с ними работать используются специальные методы.

Ниже приведен код с использованием методов для установки значений свойств.

```
class Product{
  constructor(name, price){
    this._name = name;
    this._price = price;
  }
  setName(name){
    if(name.length !== 0 ){
      this._name = name;
    }
    else{
      console.log("This is not suitable data");
    }
  }
  setPrice(price) {
    if(price > 0 ){
      this._price = price;
    }
    else{
      console.log("This is not suitable data");
    }
  }
  printInfo(){
    console.log('Name: ${this._name},
                Price: ${this._price}');
  }
}

let phone = new Product("iPhone 10", 660);
phone.setPrice(500);
```

```
console.log(phone._price);  
phone.setPrice(0);  
console.log(phone._price);
```

В этом примере объявляется класс **Product**. В классе определяются два свойства **\_name** и **\_price**. Эти свойства обозначаются нижним подчеркиванием. Это нужно для того, чтобы программисты понимали, что использовать их напрямую нельзя. А работали с ними через методы. Для этого есть два метода **setName()** и **setPrice()**. Они принимают данные и проверяют их на корректность значений перед установкой в свойство. Метод **setName()** проверяет не является ли передаваемая строка пустой, и если строка не пустая, то она устанавливается в свойство **\_name**. В противном случае в консоль выводится сообщение, что передаваемые данные некорректные и значение не устанавливается. Метод **setPrice()** проверяет аргумент и если его значение корректно, то оно установится в свойство **\_price**.

Ниже в примере создается объект **phone**. И объект **phone** два раза вызывает метод **setPrice()**. В первый раз туда передается значение **500**, а во второй **0**. Значение свойства **\_price** выводится на экран, после каждого вызова **setPrice()**. Вот что появится в консоли разработчика в результате:

500
This is not suitable data
500

Рисунок 54

В первый вызов `setPrice()` установит значение `500` для свойства `_price`. Вызов функции `setPrice()` во второй раз принимает значение `0`. Это значение не пройдет проверку и не будет установлено в `_price`. Вместо этого выведет сообщение «This is not suitable data». Таким образом значение `_price` останется корректным.

Использование функций для установки значений свойств намного надежнее чем устанавливать значения напрямую. Однако, есть еще один более удобный способ устанавливать значения в свойства. Это свойства-аксессоры.

**Свойства-аксессоры** — это методы в классе которые позволяют обрабатывать данные перед установкой значений свойств или их возвратом. Свойства-аксессоры во внешнем коде выглядят как обычные свойства, но на самом деле являются функциями. Свойства-аксессоры бывают сеттерами и геттерами.

**Сеттеры** позволяют обработать данные перед их установкой в свойства. Для того чтобы установить сеттер нужно использовать ключевое слово `set` и указать имя. Имя можно указать похожее на свойство, которое будет использоваться внутри свойства-аксессора. Главное, чтобы имя свойства и имя аксессора не были идентичные. В сеттер передается один параметр, это те данные, которые должны быть установлены в свойство.

Давайте добавим сеттеры в класс `Product`:

```
class Product{
    constructor(name, price){
        this._name = name;
        this._price = price;
    }
}
```



```

set name(name){
    if(name.trim().length !== 0 ){
        this._name = name;
    }
    else{
        console.log("This is not suitable data");
    }
}
set price(price) {
    if(price > 0 ){
        this._price = price;
    }
    else{
        console.log("This is not suitable data");
    }
}
printInfo(){
    console.log('Name: ${this._name},
                Price: ${this._price}');
}
}

let phone = new Product("iPhone 10", 660);

phone.price = 550;
console.log(phone._price);

```

В класс добавляется два сеттера, **name** и **price**, логика проверки как и в предыдущем примере. Теперь, когда необходимо изменить значение для свойства объекта, используется сеттер. Свойства-аксессоры снаружи выглядят как обычные свойства. Это их преимущество перед методами. Свойства-аксессоры не нужно вызывать как методы объекта через круглые скобки, к ним обращаются как к обычным свойствам.

**Геттеры** возвращают значение свойств. Они объявляются ключевым словом `get()`.

Давайте объявим геттеры для свойств `_price` и `_name`.

```
class Product{
  constructor(name, price){
    this._name = name;
    this._price = price;
  }
  set name(name){
    if(name.length !== 0 ){
      this._name = name;
    }
    else{
      console.log("This is not suitable data");
    }
  }
  get name(){
    return this._name;
  }
  set price(price) {
    if(price > 0 ){
      this._price = price;
    }
    else{
      console.log("This is not suitable data");
    }
  }
  get price(){
    return this._price;
  }
  printInfo(){
    console.log('Name: ${_this.name},
                Price: ${_this.price}');
  }
}
```

```
let phone = new Product("iPhone 10", 660);  
  
console.log(phone.name);  
console.log(phone.price);
```

В класс **Product** были добавлены два геттера. Обратите внимание на их названия, они совпадают с названием сеттеров, но это не приведет к ошибкам. Когда происходит обращение к свойствам-аксессорам интерпретатор понимает какой именно нужно вызвать метод-аксессор по тому, как он был вызван. Если после свойства-аксессора идет оператор равенства, это сеттер, если отсутствует — геттер.

Теперь для того, чтобы получить значение свойств обращение происходит по геттеру, а не напрямую к свойствам объекта.

Также геттеры можно использовать для того, чтобы получить вычисляемое значение. Это нужно чтобы не хранить вычисляемые данные в объекте. Можно создать геттер, в котором будет находиться логика вычисления. Например, объект класса **Product** может вернуть свою стоимость в строковом представлении и со знаком доллара в конце. Для того, чтобы не хранить строку со стоимостью товара в объекте как свойство, можно объявить геттер. В геттере будет преобразовываться стоимость в строку и добавляться знак доллара в конец.

Вот как можно реализовать такой геттер:

```
class Product{  
  constructor(name, price){  
    this._name = name;
```

```

        this._price = price;
    }
    set name(name){
        if(name.lenght !== 0 ){
            this._name = name;
        }
        else{
            console.log("This is not suitable data");
        }
    }
    get name(){
        return this._name;
    }
    set price(price) {
        if(price > 0 ){
            this._price = price;
        }
        else{
            console.log("This is not suitable data");
        }
    }

    get price(){
        return this._price;
    }

    get stringPrice(){
        return +this._price + "$";
    }

    printInfo(){
        console.log('Name: ${this._name},
                    Price: ${this._price}');
    }
}

let phone = new Product("iPhone 10", 660);
console.log(phone.stringPrice);

```

В классе `Product` был объявлен геттер с именем `stringPrice`. Внутри геттера `stringPrice` происходит преобразование значения свойства `_price` в строку, к которой будет добавлен символ `$`. После вызывается свойство-аксессор `stringPrice`, который возвращает строку. Вот такой результат выполнения можно увидеть в консоли:

660\$

Рисунок 55

На экране появилось строковое значение стоимости товара.

## Статические свойства и методы класса

Статические свойства и методы являются свойствами и методами класса, а не объектов. Их можно вызывать только у класса, а не у экземпляра этого класса.

Статические свойства хранят данные, которые относятся не к конкретному объекту, а ко всем в целом. Они полезны для хранения какой-либо вспомогательной информации.

Статические свойства объявляются в теле класса, а не в конструкторе, в отличие от обычных свойств. Для их объявления используется ключевое слово `static` перед именем свойств.

Давайте посмотрим, как объявить статическое свойство:

```
class Product{
    constructor(name, price){
        Product.count++;
    }
}
```

```
        this._name = name;
        this._price = price;
    }
    static count = 0;
}

let phone = new Product("iPhone 10", 660);
let tv = new Product("TV", 400);

Product.printCount();
```

Внутри класса **Product** объявляется статическое свойство **count**. Свойство **count** будет хранить количество созданных объектов на основе этого класса. В конструкторе значение **count** увеличивается на один. Также, обратите внимание, что для того, чтобы обратиться к статическому свойству в классе необходимо указывать название класса и использовать точку. Каждый раз, когда создается новый объект вызывается конструктор и увеличивается значение статического свойства **count**. В примере создается два объекта **phone** и **tv**, а после на экран отображается значение **count** (рис. 56).

2

Рисунок 56

Так как было создано два объекта, конструктор был вызван два раза, а это значит, что значение **count** равно 2.

Статические методы принадлежат самому классу, а не объектам созданными на его основе. Они используются

для реализации вспомогательных методов при работе с классом. Для создания статического метода также используется ключевое слово **static**, а далее обычное объявление метода в классе.

Внутри статических методов можно обратиться к статическим свойствам, но нельзя обратиться к обычным.

Давайте, например, добавим статический метод для отображения количества созданных продуктов на экране.

```
class Product{
  constructor(name, price){
    Product.count++;
    this._name = name;
    this._price = price;
  }

  static count = 0;
  static printCount(){
    console.log(Product.count);
  }
}

let phone = new Product("iPhone 10", 660);
let tv = new Product("TV", 400);

Product.printCount();
```

Внутри класса **Product** объявляется статический метод **printCount()** с помощью ключевого слова **static**. В методе **printCount()** в консоль выводится значение статического свойства **count**. После создается два объекта и вызывается **printCount()**. Так как метод **printCount** статический это

значит, что он принадлежит самому классу и вызываться он должен не от объекта, а от класса. Результат работы статического метода:

2

*Рисунок 57*

## Наследование

Перед тем как изучить как наследоваться при помощи классов, давайте вспомним что вообще такое наследование и как оно реализуется.

Наследование в JavaScript реализуется через прототипы и называется прототипным наследованием.

**Прототипное наследование** — это наследование свойств и методов от своего прототипа.

Каждый объект имеет ссылку на свой объект-прототип, от которого он наследует свойства и методы. У объекта-прототипа также есть своя ссылка на свой объект-прототип и т.д. Таким образом, создается цепочка прототипов.

Когда мы обращаемся к какому-либо свойству или методу объекта, он сначала ищется в самом объекте и, если не был там найден, поиск продолжается в прототипе. Поиск будет продолжаться пока свойство или метод не будет найден или завершится цепочка прототипов.

Давайте выведем в консоль объект без свойств и методов и посмотрим на этот объект и его прототип:

```
let obj = {};  
console.log(obj);
```



Выше в консоли отображается полностью пустой объект. Вот что можно увидеть, открыв консоль разработчика:



```

▼ {} ⓘ
  ▼ __proto__:
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
  
```

Рисунок 58

В консоли видим, что у объекта есть свойство `__proto__`, хотя на экране отображался полностью пустой объект. Свойство `__proto__` это ссылка на объект-прототип, которая есть у каждого объекта. Даже абсолютно пустой объект имеет прототип. Если не устанавливать прототип явно, то по умолчанию прототипом объекта будет объект `Object`, а точнее его свойство `prototype`, о котором будет сказано позже. Как видно из скриншота (рис. 58) прототипом для `obj` является объект `Object`. А это значит, что все методы, которые находятся в `Object` были унаследованы.

Например, можно использовать метод `toString()` чтобы получить строковое представление объекта `obj`. Вот так:

```
let obj = {};  
console.log(obj.toString());
```

И на экране появится сообщение:

```
[object Object]
```

### Рисунок 59

Когда вызывается метод `toString()`, он сначала ищется в `obj`. Если его там нет, поиск продолжается в прототипе. Таким образом, хоть метод `toString()` был объявлен у объекта `Object` его также можно вызывать от `obj` из-за наследования.

Объекту можно установить прототип явно. Это даст возможность наследовать свойства и методы других объектов, а не только объекта `Object`.

Первый способ — это изменить значение ссылки `__proto__` уже созданному объекту напрямую. Однако, это не рекомендуется делать, так как это может сильно повлиять на производительность. Вместо этого воспользуйтесь вторым способом, а именно функцией-конструктором. Функции-конструкторы позволяют создавать множество объектов с одним прототипом для всех объектов. Давайте вспомним как реализуется наследование с помощью функций-конструкторов.

Так как функции-конструкторы позволяют создать множество объектов, использование прототипного наследования очень экономит память. Методы создаются не в каждом объекте, а хранятся только в одном объекте — прототипе.

У каждой функции есть свойство `prototype`, которое изначально ссылается на практически пустой объект. Давайте создадим функцию и посмотрим на ее свойство `prototype`:

```
function Bird() {  
  
}  
console.dir(Bird);
```

Выше была создана пустая функция `Bird`, а после ее вывели в консоль, с помощью функции `console.dir()`.  
Вот что появится на экране:



```
▼ f Bird() ⓘ  
  arguments: null  
  caller: null  
  length: 0  
  name: "Bird"  
  ► prototype: {constructor: f}  
  ► __proto__: f ()  
    [[FunctionLocation]]: inheritance.js:17  
    ► [[Scopes]]: Scopes[2]
```

Рисунок 60

Так как функции — это тоже объекты в консоли разработчика можно увидеть их внутренние свойства. Одним из них является свойство `prototype`. Если его раскрыть, то внутри можно увидеть два свойства. Первое свойство `constructor`, которое ссылается на эту же функцию-конструктор. А второе `__proto__` ссылается на `Object.prototype`.

На рисунке 61 можно увидеть, как устроено свойство `prototype` функции конструктора:

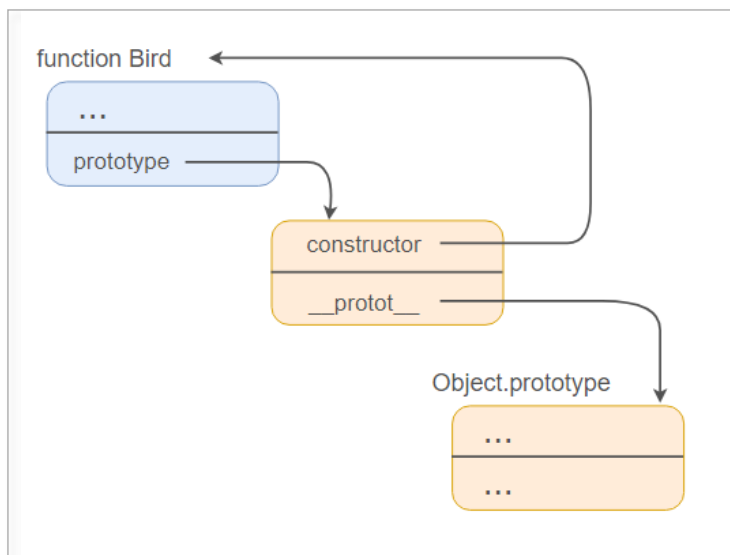


Рисунок 61

Свойство `prototype` функции `Bird` ссылается на объект. Этот объект имеет два свойства, `constructor` который ссылается на функцию `Bird` и ссылку `__proto__` она ссылается на `Object.prototype`. Зачем вообще нужно это свойство `prototype`?

Свойство `prototype` будет полезно, когда функция вызывается как функция-конструктор. Дело в том, что все объекты, которые были созданы через функции-конструкторы в качестве своей ссылки `__proto__` устанавливают объект, на который ссылается `prototype` функции конструктора.

Иными словами, свойство функции-конструктора `prototype` — это ссылка на объект, который будет являться прототипом для всех объектов созданных с помощью функции-конструктора.

Помните, три действия, которые выполняет функция-конструктор, когда вызывается с помощью ключевого слова **new**? На самом деле их больше:

1. Создается новый пустой объект.
2. В новый объект присваиваются все свойства, которые определены в методе **constructor()**.
3. В свойство **\_\_proto\_\_** нового объекта, устанавливается свойство **prototype** функции конструктора.
4. Объект неявно возвращается из конструктора.

В любом случае, создание объекта происходит с помощью функции конструктора, даже когда объекты создаются с помощью литерального синтаксиса. Например, вот так:

```
let obj = { }
```

На самом деле в коде вызывается функция-конструктор **Object()**, а создание объекта выше тоже самое что и

```
let obj = new Object()
```

Внутри функции-конструктора **Object()**, свойство **\_\_proto\_\_** устанавливается на **Object.prototype**. Поэтому **\_\_proto\_\_** всех объектов изначально ссылаются на **Object.prototype**.

Теперь стоит вспомнить как установить прототип для всех объектов, которые будут созданы через функцию-конструктор.

Например, определим функцию-конструктор для попугаев. А также объект, который будет прототипом для созданных птиц. Вот как это реализуется:

```

let bird = {
  fly() {
    console.log(`${this.name} is flying`);
  }
}

function Parrot(name, color){
  this.name = name;
  this.color = color;
}

```

Сначала был объявлен объект **bird**, этот объект станет прототипом для созданных объектов. Объект **bird** содержит в себе метод **fly()**. Ниже объявлена функция **Parrot**, она и будет функцией-конструктором для создаваемых объектов. В функцию передается **name** и **color**, которые инициализируют соответствующие свойства.

Далее укажем, что объект **bird** будет прототипом всех объектов созданных через функцию-конструктор **Parrot**.

Для этого необходимо указать, что свойство **prototype** функции **Parrot** будет ссылаться на объект **bird**. Это можно сделать с помощью такого синтаксиса:

```
Parrot.prototype = bird;
```

После, создадим объекты и посмотрим, какие свойства и методы они могут использовать:

```

let bird = {
  fly() {
    console.log(`${this.name} is flying`);
  }
}

```

```
function Parrot(name, color){
    this.name = name;
    this.color = color;
}
Parrot.prototype = bird;

let parrot1 = new Parrot("parrot1", "green and yellow");
let parrot2 = new Parrot("parrot2", "blue");

console.log(parrot1);
console.log(parrot2);
```

В пример добавились два новых объекта `parrot1` и `parrot2`. Они были созданы через функцию `Parrot()`. Внутри `Parrot()`, неявно в свойство `__proto__` нового объекта, будет установлено свойство `prototype` функции конструктора.

**Важное замечание:** в *prototype* была присвоена ссылка на *bird* до того, как создавались объекты. Если это не сделать до того, как будут созданы объекты, их `__proto__` будет ссылаться на почти пустой объект со свойствами `__proto__` и `constructor`, о котором говорилось ранее.

В конце примера на экране отображаются объекты `parrot1` и `parrot2`:



```
▼ Parrot {name: "parrot1", color: "green and yellow"} ⓘ
  color: "green and yellow"
  name: "parrot1"
  ► __proto__: Object

▼ Parrot {name: "parrot2", color: "blue"} ⓘ
  color: "blue"
  name: "parrot2"
  ► __proto__: Object
```

Рисунок 62

Выше видно, что `parrot1` и `parrot2` имеют собственные свойства `name` и `color`. Также, если открыть свойство `__proto__` можно увидеть метод `fly()`:



```

▼ Parrot {name: "parrot1", color: "green and yellow"} ⓘ
  color: "green and yellow"
  name: "parrot1"
  ▼ __proto__:
    ► fly: f fly()
    ► __proto__: Object
▼ Parrot {name: "parrot2", color: "blue"} ⓘ
  color: "blue"
  name: "parrot2"
  ▼ __proto__:
    ► fly: f fly()
    ► __proto__: Object

```

Рисунок 63

Так как прототипом для новых объектов является `bird`, они имеют возможность использовать метод `fly()`.

Давайте вызовем метод `fly()` от `parrot1` и `parrot2`:

```

let bird = {
  fly() {
    console.log(`${this.name} is flying`);
  }
}

function Parrot(name, color) {
  this.name = name;
  this.color = color;
}

Parrot.prototype = bird;
let parrot1 = new Parrot("parrot1", "green and yellow");

```



```
let parrot2 = new Parrot("parrot2", "blue");

parrot1.fly();
parrot2.fly();
```

Результат выполнения:

```
parrot1 is flying
parrot2 is flying
```

Рисунок 64

Давайте рассмотрим на изображении, на что ссылаются свойства **prototype** функции-конструктора и **\_\_proto\_\_** объектов:

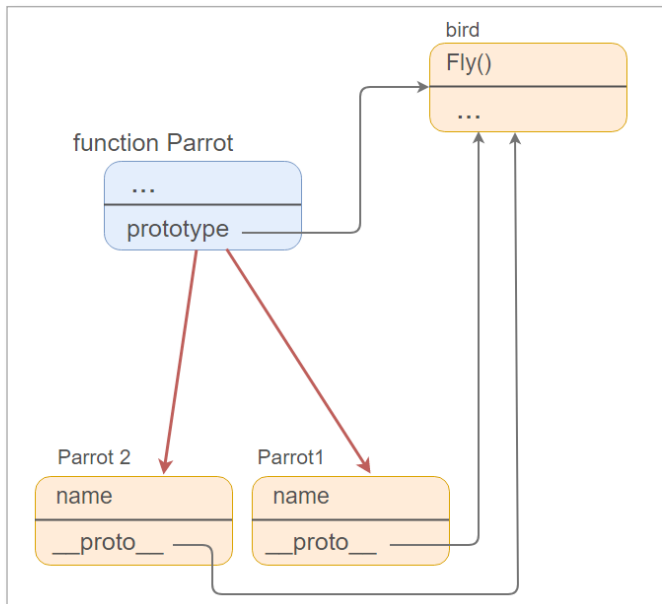


Рисунок 65

На рисунке 65 видно, что `prototype` функции ссылается на объект `bird`. Так как, объекты `parrot1` и `parro2` создаются функцией `Parrot()` (это обозначается красными стрелками), их свойство `__proto__` также ссылается на объект `bird`.

Теперь, когда мы разобрали наследование давайте узнаем, как его использовать в классах. Как мы помним, классы — это те же функции, а это значит, что и наследование реализуется по таким же правилам.

Давайте перепишем пример, который реализовали выше. В новом примере, вместо функции конструктора мы будем использовать классы. Начнем по порядку.

Объявим два класса `Bird` и `Parrot`, а после посмотрим их значение `__proto__` и `prototype`:

```
class Bird {
}

class Parrot{
}

console.dir(Bird);
console.dir(Parrot);
```

Выше создаются два пустых класса `Bird` и `Parrot`, которые никак не связаны.

На скриншоте (рис. 66) показано `__proto__` и `prototype` этих классов. Их `prototype` ссылается на разные объекты. Эти объекты имеют свойства `__proto__` и `constructor`. А их собственное свойство `__proto__` ссылается на `Function.prototype`. `Function.prototype` это прототип для всех функций. Так как классы — это функции их свойство `__proto__` ссылается на `Function.prototype`.

```

▼ class Bird ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Bird"
  ▼ prototype:
    ► constructor: class Bird
    ► __proto__: Object
    ► __proto__: f ()
    [[FunctionLocation]]: inheritance.js:43
    ► [[Scopes]]: Scopes[2]
▼ class Parrot ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Parrot"
  ▼ prototype:
    ► constructor: class Parrot
    ► __proto__: Object
    ► __proto__: f ()
    [[FunctionLocation]]: inheritance.js:49
    ► [[Scopes]]: Scopes[2]

```

Рисунок 66

Следующим шагом будет создание наследования между двумя классами. Это можно сделать, используя новое ключевое слово **extends**.

Для того чтобы наследоваться при определении класса, нужно указать **extends** и имя класса, свойства и методы которого будут наследоваться. Вот так:

```

class Parrot extends Bird{
}

```

Давайте рассмотрим, куда теперь ссылается **\_\_proto\_\_** и **prototype** при наследовании классов.

```

class Bird{
  fly(){
    console.log(`${this.name} is flying`);
  }
}

class Parrot extends Bird{
}

console.dir(Bird);
console.dir(Parrot);

```

Для этого, выше, в класс **Bird** добавили функцию **fly()**, а также теперь **Parrot** наследуется от класса **Bird**. Далее эти классы отображаются на экране. Класс **Bird** это базовый класс для **Parrot**, так принято называть класс от которого происходит наследование. А класс **Parrot** это дочерний класс, так как он был унаследован от **Bird**.

Сначала рассмотрим класс **Bird**:

```

▼ class Bird ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Bird"
  ▶ prototype: {constructor: f, fly: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: inheritance.js:55
  ▶ [[Scopes]]: Scopes[2]

```

Рисунок 67

В классе был объявлен метод **fly()**, но из скриншота видим, что конкретно в классе этот метод отсутствует. Однако, давайте посмотрим в свойство **prototype**:

```

▼ class Bird ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Bird"
  ▼ prototype:
    ▶ constructor: class Bird
    ▶ fly: f fly()
    ▶ __proto__: Object
    ▶ __proto__: f ()
    [[FunctionLocation]]: inheritance.js:55
    [[Scopes]]: Scopes[2]

```

Рисунок 68

Выше показано, что метод `fly()` был добавлен в объект на который ссылается свойство `prototype`. Так происходит всегда, когда метод объявляется в теле класса.

Мы также можем это сделать для функции-конструктора. Вот таким способом:

```

function Bird(){
}
Bird.prototype.fly = function() {
  console.log(`${this.name} is flying`);
}

console.dir(Bird);

```

В этом примере создается функция-конструктор `Bird`. После происходит обращение к свойству `prototype` и в объект, на который ссылается `prototype` добавляется новый метод. Однако, синтаксис классов упрощает эту манипуляцию. Классы позволяют просто определить метод прямо в теле класса, вследствие чего эти методы будут добавлены в `prototype`.

Вернемся к классам. Переходим к классу `Parrot`, напомним, этот класс был унаследован от класса `Bird` с помощью ключевого слова `extends`.

Изображение, на котором показан класс `Parrot`:

```
▼ class Parrot ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Parrot"
  ▶ prototype: Bird {constructor: f}
  ▶ __proto__: class Bird
    [[FunctionLocation]]: inheritance.js:68
    [[Scopes]]: Scopes[2]
```

Рисунок 69

Как и ожидается метода `fly()` в классе `Parrot` нет. Давайте откроем `prototype` и посмотрим есть ли у него метод `fly()`:

```
▼ class Parrot ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Parrot"
  ▼ prototype: Bird
    ▶ constructor: class Parrot
    ▶ __proto__: Object
    ▶ __proto__: class Bird
      [[FunctionLocation]]: inheritance.js:68
      ▶ [[Scopes]]: Scopes[2]
```

Рисунок 70

Метода `fly()` также нет в ссылке `prototype`. Однако, открыв `__proto__` свойства `prototype` можно обнаружить этот метод там:

```

▼ class Parrot ⓘ
  arguments: (...)
  caller: (...)
  length: 0
  name: "Parrot"
▼ prototype: Bird
  ▶ constructor: class Parrot
  ▼ __proto__:
    ▶ constructor: class Bird
    ▶ fly: f fly()
    ▶ __proto__: Object
  ▶ __proto__: class Bird
  [[FunctionLocation]]: inheritance.js:68
  ▶ [[Scopes]]: Scopes[2]

```

Рисунок 71

Это происходит из-за того, что ключевое слово **extends** устанавливает ссылку на объект, которая хранится в свойстве **prototype**, базового класса свойства **prototype**, **\_\_proto\_\_** дочернего класса.

Рассмотрим изображение на котором показана связь **prototype** базового класса и **prototype.\_\_proto\_\_** дочернего класса:

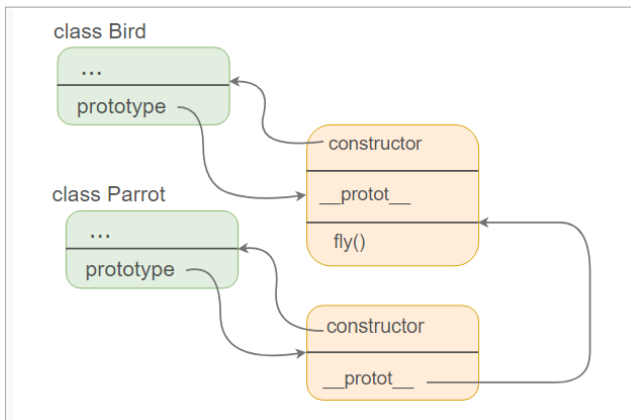


Рисунок 72

Давайте разберем, как ключевое слово `extends` устанавливает `prototype` и `__proto__` наследуемых классов.

Начнем с `Bird`, его свойство `prototype` ссылается на объект. Этот объект хранит такие свойства как `constructor`, он в свою очередь ссылается на тот же класс `Bird`. Ссылка `__proto__` ссылается на базовый класс для `Bird`. Также, в объекте на который ссылается `prototype` есть метод `fly()`. Так как `fly()` был объявлен в теле класса `Bird`, этот метод автоматически будет добавлен в объект, на который ссылается `prototype`.

Перейдем к классу `Parrot`. Его свойство `prototype` ссылается на другой объект, у которого также есть свойство `constructor` оно ссылается на класс `Parrot`. А свойство `__proto__` ссылается на объект, на который ссылается `prototype` класса `Bird` из-за наследования.

Теперь внесем еще пару изменений в класс `Parrot`. А именно, добавим в него свойства `name` и `color`, а также метод `talk()`. Метод `talk()` определяется в классе `Parrot`. Это было сделано из-за того, что не все птицы умеют разговаривать. Если бы мы объявили этот метод в `Bird`, он был бы доступен всем птицам, которые унаследовались от `Bird`. Такое поведение было бы неправильным.

Вот так теперь выглядит измененный класс `Parrot`:

```
class Bird{
  fly() {
    console.log(`${this.name} is flying`);
  }
}

class Parrot extends Bird{
  constructor(name, color){
```



```

        super();
        this.name = name;
        this.color = color;
    }
    talk(){
        console.log(`${this.name} is talking`);
    }
}

console.dir(Bird);
console.dir(Parrot);

```

В **Parrot** был добавлен конструктор, который принимает параметры и инициализирует свойства. Внутри конструктора первой строкой используется ключевое слово **super()**. Это ключевое слово используется как функция, которая вызывает конструктор базового класса. **super()** обязательно нужно использовать при наследовании и вызывать его нужно до первого обращения к **this**. Посмотрим, как теперь выглядит **Parrot**:

```

▼ class Parrot ⓘ
  arguments: (...)
  caller: (...)
  length: 2
  name: "Parrot"
  ▼ prototype: Bird
    ► constructor: class Parrot
    ► talk: f talk()
    ▼ __proto__:
      ► constructor: class Bird
      ► fly: f fly()
      ► __proto__: Object
    ► __proto__: class Bird
    [[FunctionLocation]]: inheritance.js:69
    ► [[Scopes]]: Scopes[2]

```

Рисунок 73

Как уже было сказано, все методы объявленные в теле класса будут добавлены в **prototype** этого класса. Таким образом, метод **talk()** находится в объекте на который ссылается **prototype**. Свойство **\_\_proto\_\_** свойства **prototype** ссылается на **Bird.prototype** из-за наследования.

Далее создадим несколько объектов класса **Parrot** и выведем их на экран, чтобы посмотреть на то, как устроено наследование:

```
class Bird{
  fly() {
    console.log(`${this.name} is flying`);
  }
}

class Parrot extends Bird{
  constructor(name, color){
    super();
    this.name = name;
    this.color = color;
  }
  talk(){
    console.log(`${this.name} is talking`);
  }
}

let parrot1 = new Parrot("parrot1", "green and yellow");
let parrot2 = new Parrot("parrot2", "blue");

console.log(parrot1);
console.log(parrot2);
```

На экране показаны два объекта **parrot1** и **parrot2**. Свойство **\_\_proto\_\_** этих двух объектов ссылается на **prototype** класса, с помощью которого они были созданы, т.е.

на `prototype` класса `Parrot`. Поэтому метод `talk()` находится в объекте на который ссылается `__proto__`. Также, если открыть свойство `__proto__` и обратиться к его свойству `__proto__` можно увидеть, что оно ссылается на `prototype` класса `Bird`. Поэтому объекты могут обращаться к методу `fly()`, который был объявлен в классе `Bird`.

```
▼ Parrot {name: "parrot1", color: "green and yellow"} ⓘ
  color: "green and yellow"
  name: "parrot1"
  ▼ __proto__: Bird
    ► constructor: class Parrot
    ► talk: f talk()
  ▼ __proto__:
    ► constructor: class Bird
    ► fly: f fly()
    ► __proto__: Object

▼ Parrot {name: "parrot2", color: "blue"} ⓘ
  color: "blue"
  name: "parrot2"
  ▼ __proto__: Bird
    ► constructor: class Parrot
    ► talk: f talk()
  ▼ __proto__:
    ► constructor: class Bird
    ► fly: f fly()
    ► __proto__: Object
```

Рисунок 74

Давайте рассмотрим это на рисунке 75 (см. ниже). Похожее изображение было выше, в нем мы разобрали связь между свойствами `prototype` двух классов. На этом изображение было добавлено создание объектов `parrot1` и `parrot2`, это обозначено красными стрелками. Как уже было сказано их свойства `__proto__` ссылаются на `prototype` класса `Parrot`.

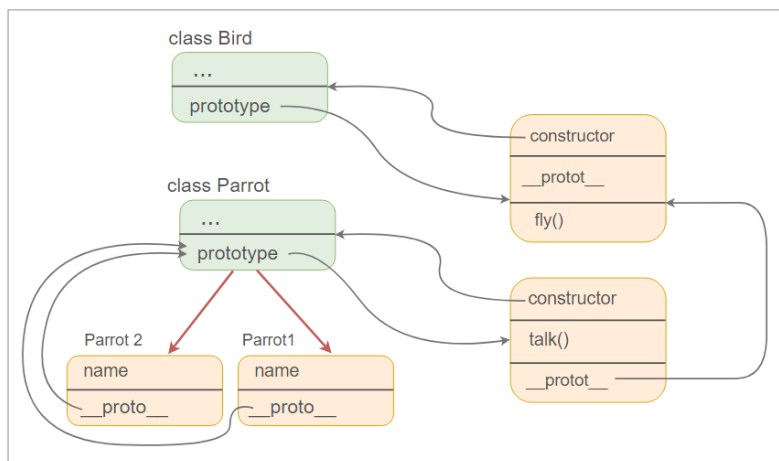


Рисунок 75

Свойство `prototype` класса `Parrot` ссылается на объект в котором есть методы `talk()`. А свойство `__proto__` этого объекта ссылается на другой объект из-за наследования от класса `Bird`. На этот же объект ссылается `prototype` класса `Bird`. Так как метод `fly()` был определен в классе `Bird`, фактически метод будет находится в `prototype` класса `Bird`.

Теперь давайте вызовем методы `talk()` и `fly()` созданных объектов:

```
class Bird{
  fly() {
    console.log(`${this.name} is flying`);
  }
}

class Parrot extends Bird{
  constructor(name, color){
    super();
    this.name = name;
  }
}
```

```

        this.color = color;
    }
    talk(){
        console.log(`${this.name} is talking`);
    }
}

let parrot1 = new Parrot("parrot1", "green and yellow");
let parrot2 = new Parrot("parrot2", "blue");

parrot1.fly();
parrot2.talk();

```

У объекта `parrot1` вызывается метод `fly()`, а у объекта `parrot2` метод `talk()`. Результат на экране:

parrot1 is flying
parrot2 is talking

*Рисунок 76*

Из этого изображения видно, что объекты класса `Parrot` могут обращаться к методам, которые находятся в их прототипе и выше.

Кроме того, что дочерние классы могут наследовать методы родительских классов, они могут их переопределять. Это позволяет давать им свою реализацию. Рассмотрим пример:

```

class Bird{
    fly(){
        console.log(`${this.name} is flying`);
    }
}

```

```


class Parrot extends Bird{
  constructor(name, color){
    super();
    this.name = name;
    this.color = color;
  }
  fly(){
    console.log('I am flying well');
  }

  talk(){
    console.log(`${this.name} is talking`);
  }
}

let parrot1 = new Parrot("parrot1", "green and yellow");
parrot1.fly();

```

В дочернем классе **Parrot** переопределен метод **fly()** базового класса. Теперь в **fly()** выводится на экран такая надпись: **'I am flying well'**. Посмотрим на результат:



I am flying well

*Рисунок 77*

На экране можно увидеть это сообщение вместо: **'parrot1 is flying'**.

### **Еще немного про *super()***

Ключевое слово **super** это метод, который вызывает конструктор базового класса. Вызов **super()** должен быть в начале конструктора, а точнее до первого использования ключевого слова **this**. Если использовать его после

первого обращения к `this` или вовсе его не использовать, будет сгенерирована вот такая ошибка:

```
► Uncaught ReferenceError: Must call super constructor in derived class before  
accessing 'this' or returning from derived constructor  
at new Parrot (inheritance.js:71)
```


### Рисунок 78

Также с помощью `super` можно вызывать методы базового класса. Например, в переопределенном методе `fly()` нужно сначала выполнить алгоритм метода `fly()` базового класса, а уже потом алгоритм переопределенного метода. Для этого, в переопределенном методе используется слово `super`, но без круглых скобок, так как нам не нужно вызывать конструктор базового класса. Вот как это выглядит:

```
class Bird{  
  fly() {  
    console.log(`${this.name} is flying`);  
  }  
}  
  
class Parrot extends Bird{  
  constructor(name, color){  
    super();  
    this.name = name;  
    this.color = color;  
  }  
  fly() {  
    super.fly();  
    console.log('I am flying well');  
  }  
  
  talk(){  
    console.log(`${this.name} is talking`);  
  }  
}
```

```
    }  
}  
  
let parrot1 = new Parrot("parrot1", "green and yellow");  
parrot1.fly();
```

В методе `fly()` класса `Parrot` сначала вызовется метод `fly()` класса `Bird`, а после выполняется дальнейшая логика. Ниже, в коде создается объект `parrot1` и вызывается `fly()`. Результат выполнения метода `fly()`:



```
parrot1 is flying  
I am flying well
```

*Рисунок 79*

Как видно из скриншота сначала вызывается и выполняется логика базового класса, а после и сам метод `fly()` дочернего класса.



# Модули

Со временем проекты, написанные на JavaScript становились все больше и программистам стало сложнее держать весь JavaScript код в одном файле. Долгое время JavaScript не поддерживал модули. Однако в спецификации ES-2015 была добавлена возможность их использования.

До появления модулей JavaScript код писали в одном файле. Однако, из-за того, что поддерживать код стало очень сложно его начали разбивать на разные файлы. Эти файлы подключались в HTML файл с помощью тега `script`. Хотя подключение JS файлов позволяет разбить один большой файл на несколько маленьких, но такой способ добавляет одну проблему. Рассмотрим ее на примере:

В примере используются два JavaScript файла *index.js* и *math.js* и один HTML — *index.html*.

Файл *index.html*:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
        content="width=device-width,
                initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
```

```
<script src="math.js"></script>
<script src="index.js"></script>

</html>
```

Файл *math.js*:

```
class Math{
  add(num1, num2){
    return num1 + num2;
  }

  minus(num1, num2){
    return num1 - num2;
  }
}
```

Файл *index.js*

```
let math = new Math();

console.log(math.add(1,2));
```

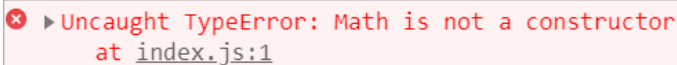
В файле *index.html* подключаются два скрипта. Первый это *math.js*, а второй *index.js*. В файле *math.js* объявляется класс **Math**, который содержит функции **add()** и **minus()**. А в *index.js* создается экземпляр класса **Math**, и вызывается функция **add()**. Результат можно увидеть на изображении:

3

Рисунок 80

Так можно реализовать разделение кода на файлы, однако, проблема заключается в том, что подключение JS файлов в HTML должно идти в правильном порядке. Из-за того, что код из одних файлов использует код из других файлов, образуются зависимости между ними.

В этом примере файл *index.js* зависит от *math.js*, а точнее от класса **Math**. В *index.html* сначала подключается файл *math.js*, а после файл *index.js*. Это позволяет в *index.js* использовать класс из *math.js*. Однако, что, если сначала подключить *index.js*, а после *math.js*? Если это сделать, то в консоли разработчика появится такая ошибка:

A screenshot of a browser's developer console showing an error. It features a red 'x' icon in a circle on the left. The text of the error is: 'Uncaught TypeError: Math is not a constructor' on the first line, and 'at index.js:1' on the second line. The background of the error box is light pink.

```
✖ ▶ Uncaught TypeError: Math is not a constructor  
   at index.js:1
```

Рисунок 81

Это происходит из-за того, что класс **Math** используется раньше, чем подключается файл, в котором он объявлен. Сначала будет подключен файл *index.js*. А это значит, что именно его код будет выполнен первым.

Внутри *index.js* интерпретатор встретит строку, где используется класс **Math** из файла *math.js*. Это и приводит к ошибке, так как файл *math.js* еще не был подключен, и соответственно класс **Math** не был создан.

Пример выше достаточно простой, но в больших проектах вряд ли будет всего 2 файла. Представьте, что файлов с JavaScript кодом будет несколько десятков. Тогда нужно учитывать все зависимости между ними и подключать их в правильном порядке. Для того, чтобы избежать зависимостей между файлами и избавиться от проблемы

с подключением скриптов в ES6 добавили модули. Они позволяют разбивать JS код на файлы и не думать в каком порядке их подключать.

Модули позволяют экспортировать определенные части, которые будут доступны только там, где были импортированы.

Модулем является один JavaScript файл, части которого можно экспортировать в другой модуль. Для того чтобы код из модуля можно было использовать в других модулях, его нужно экспортировать. Для этого есть ключевое слово **export**. Его указывают перед объявлением того, что нужно экспортировать. Вот так:

Модуль *math.js*:

```
export class Math{
  add(num1, num2){
    return num1 + num2;
  }

  minus(num1, num2){
    return num1 - num2;
  }
};
```

В примере ключевое слово **export** указывается перед тем, как объявить класс **Math**. Слово **export** значит, что переменные, функции и классы помеченные этим ключевым словом можно использовать в других модулях. Для того, чтобы использовать данные из других модулей есть **import**. С его помощью импортируются данные, которые были экспортированы из других модулей.

Давайте импортируем класс **Math** в модуль *index.js*:

## Модуль *index.js*:

```
import {Math} from "../math.js"
let math = new Math();
console.log(math.add(1,2));
```

Перед тем как использовать класс **Math** в другом модуле его необходимо импортировать. Для этого в первой строчке кода используется ключевое слово **import**. После него идут фигурные скобки, а в них название того, что нужно импортировать. Для того, чтобы указать откуда импортировать данные, используется ключевое слово **from**, а далее путь к модулю в кавычках.

Теперь, когда класс **Math** из модуля *math.js* импортирован в *index.js* можно посмотреть результат работы. Для этого *index.js* нужно подключить в *index.html*. Именно *index.js* будет главным файлом, в котором будет выполняться код.

**Важное замечание:** при подключении JavaScript файлов в HTML с помощью тегов *script* им нужно указывать специальный атрибут. Этот атрибут обозначает, что этот JS файл является модулем и в нем можно использовать ключевые слова **import** и **export**. Ниже показано как нужно подключить файл:

## HTML файл:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```

<meta name="viewport"
      content="width=device-width,
              initial-scale=1.0">
<title>Document</title>
</head>
<body>
</body>
<script type="module" src="index.js"></script>
</html>

```

Выше видно, что тегу `script`, который подключает *index.js* устанавливается атрибут `type` со значением `module`. Если для тега `script` не указать атрибут `type` со значением `module`, то при использовании ключевых слов `import` и `export` произойдет такая ошибка:

✖ Uncaught SyntaxError: Cannot use import statement outside a module

Рисунок 82

Также в HTML файле отсутствует подключение *math.js*. Так как данные из этого модуля экспортируются с помощью ключевого слова `export` подключение в HTML файле не нужно.

Ниже на изображении показан результат работы файла *index.js*:

3

Рисунок 83

Из скриншота понятно, что экспорт и импорт класса `Math` прошел успешно.

С помощью одного `export` можно экспортировать сразу несколько переменных, функций или классов. Например, в модуле *math.js* добавляется еще константа `PI`, которую тоже нужно экспортировать. В примере ниже мы можем увидеть, как это сделать:

Модуль *math.js*:

```
class Math{
  add(num1, num2){
    return num1 + num2;
  }

  minus(num1, num2){
    return num1 - num2;
  }
};

const PI = 3.14;

export {Math, PI};
```

В модуль была добавлена константа `PI`. Также слово `export` было удалено при объявлении класса `Math`, так как импортируется не только класс, а и константа. Импорт данных происходит в конце, когда все что нужно импортировать уже было объявлено. Для того чтобы экспортировать сразу множество каких-либо данных, нужно использовать `export`, а за ним перечислить все что нужно в фигурных скобках.

Теперь, для того чтобы в других модулях можно было использовать константу `PI` — ее необходимо импортировать. Для этого, просто, нужно добавить константу в фигурные скобки при импорте.

Вот так:

```
import {Math, PI} from "../math.js"

let math = new Math();

console.log(math.PI);
```

В примере теперь импортируется и класс **Math** и константа **PI**. Таким образом, можно экспортировать и импортировать набор функциональности.

Важной деталью является то, что каждый модуль обладает своей областью видимости. Это значит, что все переменные, функции и классы видны только в том модуле, в котором были объявлены. Если явно не экспортировать какие-либо данные, то получить к ним доступ из другого модуля не получится. Это очень полезно, так как мы имеем возможность экспортировать только то, что предназначено для внешнего использования, а внутренняя логика модуля будет закрыта.

В примере ниже экспортируется только класс **Math**, а константа **PI** остается закрыта для внешнего использования.

Модуль *math.js*:

```
class Math{
  add(num1, num2){
    return num1 + num2;
  }
  minus(num1, num2){
    return num1 - num2;
  }
};
```



```
const PI = 3.14;
export {Math};
```

Модуль *index.js*:

```
import {Math, PI} from "../math.js"
console.log(PI);
```

В модуле *index.js* происходит попытка импортировать и использовать константу **PI** из модуля *math.js*. Однако, при обращении **PI** происходит ошибка, так как в модуле *math.js* экспортируется только класс **Math**.

Такую ошибку можно увидеть в консоли:

✖ Uncaught SyntaxError: The requested module '../math.js' does not provide an export named 'PI'

Рисунок 84

## Ключевое слово **as**

Имена для импортированных данных можно изменить с помощью ключевого слова **as**. Для того чтобы изменить имя импортированных данных, нужно поставить фигурные скобки, указать имя того, что будет импортировано, затем ключевое слово **as**, а затем новое имя. Вот так:

Модуль *index.js*:

```
import {Math as Calculator} from "../math.js"
let calc = new Calculator();
console.log(calc.minus(3,2));
```

В модуле *index.js* импортируется класс **Math** из модуля *math.js*. Прямо в объявлении импорта изменяется имя для класса **Math**. Для этого после указания имени класса используется ключевое слово **as**, а далее идет новое имя, в нашем случае это **Calculator**. После создается новый объект класса **Math** с помощью имени **Calculator**. Далее выполняется метод **minus()**, а его результат выводится на экран. Вот результат:


 A screenshot of a terminal window with a light gray border. Inside the terminal, the number '1' is displayed in a blue monospace font.
 

1

Рисунок 85

Также есть возможность импортировать сразу все доступные данных из модуля. Это можно сделать с помощью записи **\* as**. Когда используется **\* as** все, что было доступно в импортируемом модуле будет представлено в виде объекта. Рассмотрим это на примере:

Модуль *math.js*:

```
class
{Math, PI}
```

Модуль *index.js*:

```
import * as Calculator from "./math.js"
console.log(Calculator.PI);
let math = new Calculator.Math();
console.log(math.minus(3,2));
```

В модуле *index.js* происходит импорт всех доступных данных из модуля *math.js*. В первой строке после ключевого слова **import** используется такая запись **\* as**. Эта

запись говорит о том, что из указанного модуля в объект импортируется все. Объект должен объявляться после `* as`. В нашем примере этим объектом является `Calculator`. После импорта происходит обращение к константе `PI` через объект `Calculator`. Аналогичным способом создается объект класса `Math()`, и вызывается функция `minus()`. Результат можно увидеть на экране:



1
3.14

Рисунок 86

Из примера видно, что запись `* as` позволяет импортировать сразу множество данных в отличие от одиночного импорта.

Импортировать сразу все возможные данные из модуля — это полезная возможность. Однако, как правило при разработке проекта, код стараются организовывать так, чтобы один модуль отвечал за одну функциональность. Например, модуль `User.js` будет отвечать только за класс `User`, а модуль `SayHello.js` отвечать за приветствие пользователя.

В такой ситуации есть `export default` (экспорт по умолчанию) позволяет экспортировать только что-то одно из всего модуля. То есть, он обозначает, что именно эти данные могут использоваться в модуле, в который они будут импортироваться.

Для того чтобы обозначить что будет экспортироваться по умолчанию используется ключевое слово `default`. Оно указывается сразу после ключевого слова `export`, а за ним идет имя того, что будет экспортироваться. В одном

модуле можно использовать **default** только один раз. Давайте рассмотрим.

Модуль *math.js*:

```
export default class Math{
  add(num1, num2){
    return num1 + num2;
  }
  minus(num1, num2){
    return num1 - num2;
  }
};

const PI = 3.14;
```

Модуль *index.js*:

```
import Math from "./math.js";

let math = new Math();
console.log(math.minus(3,2));
```

В *math.js* экспортируется по умолчанию класс **Math**. Перед объявлением класса указывается ключевое слово **export**, а за ним слово **default**. Также в модуле присутствует константа **PI**. Однако она не экспортируется из модуля. В модуле *math.js* импорт класса **Math** указывается, как и обычно, за исключением того, что фигурные скобки уже не нужны. Далее создается объект **math** и выполняется метод **minus()**. Результат на экране:

10

Рисунок 87

Также стоит сказать, что даже если в *math.js* перед объявлением **PI** указать слово **export**, то использовать константу в другом модуле не получится. Рассмотрим ниже:

Модуль *math.js*:

```
export default class Math{
  add(num1, num2){
    return num1 + num2;
  }

  minus(num1, num2){
    return num1 - num2;
  }
};

export const PI = 3.14;
```

Модуль *index.js*:

```
import Math from "./math.js";
import PI from "./math.js";

let math = new Math();
console.log(math.add(5, 5));
console.log(PI);
```

В примере, в модуле *math.js* экспортируется по умолчанию класс **Math**. А кроме него обычным способом экспортируется еще и константа **PI**. В *index.js* импортируется **Math** и **PI**. Они импортируются по отдельности, для этого есть такая запись:

```
import {Math, PI} from "./math.js";
```

вызовет ошибку в консоли.

Uncaught SyntaxError: The requested module './math.js' does not provide an export named 'Math'

Рисунок 88

В *index.js* происходит попытка отобразить значение **PI** на экране, но при выводе вместо «3.14» в консоли появится такое сообщение:

```
class Math{
  add(num1, num2){
    return num1 + num2;
  }

  minus(num1, num2){
    return num1 - num2;
  }
}
```

Рисунок 89

Как видно из скриншота выше на экран выводится все содержимое класса **Math**. Это происходит из-за того, что по умолчанию модуль *math.js* экспортирует класс **Math**. Когда указывается второй импорт, то в **PI** будет установлено именно то, что было экспортировано по умолчанию, в этом случае это класс **Math**.

Нельзя использовать два экспорта по умолчанию в одном модуле. Если указать **export default** для класса **Math** и для константы **PI**, то в консоли появится такая ошибка:

✖ Uncaught SyntaxError: Duplicate export of 'default'

Рисунок 90

# Деструктуризация

В стандарте ES6 появился новый синтаксис, который называют деструктуризацией, что позволяет разобрать массив или объект на переменные.

## Деструктуризация объекта

Деструктуризация удобна, когда в коде нужно часто обращаться к свойствам объекта. Рассмотрим это на примере:

```
let student = {  
  name: "Alan",  
  surname: "German",  
  age: 18,  
  group: "A1",  
  faculty: "Programming"  
};  
  
console.log(student.name);  
console.log(student.surname);  
console.log(student.age);
```

Мы видим, что создается объект `student` у которого множество свойств. Далее происходит обращение к трем свойствам, а именно к `name`, `surname` и `age`. Результат выполнения кода можно увидеть далее:

---

Alan

---

German

---

*Рисунок 91*

Этот код можно написать по-другому, используя деструктуризацию. В примере выше обращение к свойствам происходит с помощью записи `student`, а далее идет имя свойства. Эта запись абсолютно верная, но применив деструктуризацию мы можем больше ее не использовать.

Деструктуризация позволит разбить объект на набор переменных, что дает возможность не обращаться каждый раз к свойствам объекта с помощью точки, а работать с переменными. В дальнейшем вы часто будете использовать деструктуризацию, например при разработке React-приложений.

Посмотрим на синтаксис деструктуризации объекта:

```
let {property1, property2} = {property1: "value1",  
                             property2: "value2"};
```

Сначала нужно указать ключевое слово `let` (или `const`, если нужно чтобы значения не изменялись). После в фигурных скобках указывается набор переменных. Очень важно, чтобы имена переменных совпадали с именами свойств, которые нужно «достать» из объекта.

Таким образом, в переменную `property1` будет записано значение из свойства `property1`, а в переменную `property2` — значение `property2`. После того как объект был деструктуризирован, переменные будут хранить значение свойств объекта. В переменной `property1` хранится значение `value1`, а в `property2` — `value2`.

Рассмотрим деструктуризацию на примере:

```
let student = {  
  name: "Alan",
```



```
    surname: "German",  
    age: 18,  
    group: "A1",  
    faculty: "Programming"  
};  
  
let {name, surname, age} = student;  
  
console.log(name);  
console.log(surname);
```

В начале создаётся объект `student`. После идет деструктуризация объекта. В ней объявляются три переменные `name`, `surname` и `age`. В эти переменные будут присвоены значения свойств объекта `student` с таким же именем. После деструктуризации переменные `name`, `surname` и `age` можно использовать в коде. В примере на экран выводятся значения всех переменных:

Alan
German
18

Рисунок 92

Переменные `name`, `surname` и `age` хранят в себе значения полученные при деструктуризации объекта `student`.

Также при деструктуризации есть возможность дать другое название переменной, если нас не устраивает название свойства. Для того чтобы изменить имя переменной нужно поставить двоеточие и указать новое имя. Вот такими образом:

```
let student = {  
  name: "Alan",  
  surname: "German",  
  age: 18,  
  group: "A1",  
  faculty: "Programming"  
};  
  
let {name: firstName, surname: lastName, age} = student;  
  
console.log(firstName);  
console.log(lastName);
```

В примере при деструктуризации были изменены названия переменных `name` на `firstName` и `surname` на `lastName`. Далее `firstName` и `lastName` используются для вывода на экран:



Рисунок 93

Из скриншота видно, что свойство `name` сохраняется в переменную `firstName`, а свойство `surname` в `lastName`.

Если указанного свойства в объекте нет, то в переменную запишется значение `undefined`. Для того, чтобы избежать такой ситуации можно указать значение по умолчанию.

Когда значение по умолчанию используется, а указанного свойства нет, то будет установлено значение по умолчанию. Такая возможность приносит пользу, когда

точно не известно присутствует ли свойство в объекте. Вот как это выглядит:

```
let student = {  
  name: "Alan",  
  surname: "German",  
  group: "A1",  
  faculty: "Programming"  
};  
  
let {name, surname, age = 17} = student;  
console.log(age);
```

К примеру, у объекта `student` отсутствует свойство `age`. При деструктуризации переменной `age` через знак равно указывается значение по умолчанию равное `17`. Это значение будет установлено если в объекте `student` нет собственного свойства `age`. Так как у объекта `student` нет свойства `age` в переменную запишется значение по умолчанию. После `age` отображается на экран:



Рисунок 94

Переменная `age` равняется `17`, а это значит, что в нее было установлено значение по дефолту.

Переименование переменных и значение по умолчанию можно использовать вместе. Для этого, всего лишь, нужно объединить переименование и указать значение по умолчанию. Вот так:

```
let {age: yearsOld = 17 } = student;
```

Сначала указывается имя свойства, которое будет установлено в переменную, далее новое имя для переменной, а в конце значение по дефолту.

Пример использования такой записи:

```
let student = {  
  name: "Alan",  
  surname: "German",  
  group: "A1",  
  faculty: "Programming"  
};  
  
let {age: yearsOld = 17 } = student;  
  
console.log(yearsOld);
```

Выше из объекта `student` будет выбрано свойство `age`, которое будет записано в переменную `yearsOld`. Если свойства `age` нет в объекте, то будет установлено значение по умолчанию. После деструктуризации переменная `yearsOld` выводится на экран.

Результат выполнения примера:



17

Рисунок 95

Переменная получила значение `17` из-за того, что объект `student` не имел свойства `age`. А сама переменная называется `yearsOld`, так как была переименована при деструктуризации.

В деструктуризации есть возможность получить все оставшиеся значения объекта в одну переменную. До-

пустим мы хотим, чтобы значения свойств `name`, `surname`, `age` хранились в своих собственных переменных, а оставшиеся свойства `group` и `faculty` в одной переменной в виде объекта. Чтобы так сделать нужно использовать `rest` оператор. `Rest` оператор (еще называют остаточные параметры) — это специальный оператор, который обозначается троеточием. Этот оператор указывается перед названием переменной, в которую будет присваиваться все оставшиеся данные. Вот как это выглядит:

```
let {name, surname, age, ...otherInfo} = student;
```

Полный пример:

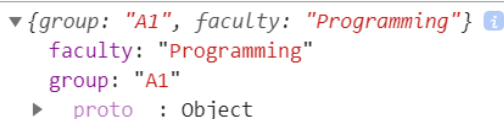
```
let student = {
  name: "Alan",
  surname: "German",
  age: 18,
  group: "A1",
  faculty: "Programming"
};

let {name, surname, age, ...otherInfo} = student;

console.log(otherInfo);
```

В этом примере в деструктуризации, кроме обычных переменных `name`, `surname` и `age` объявляется еще, и переменная `otherInfo` которая будет хранить все оставшиеся свойства объекта `student`. Это возможно благодаря оператору «...» который указывается перед объявлением переменной `otherInfo`. Переменная `otherInfo` будет содержать в себе объект, свойства которого будут оставшимися

свойствами `student`. После на экран отображается содержимое переменной `otherInfo`:



```
▼ {group: "A1", faculty: "Programming"} ⓘ
  faculty: "Programming"
  group: "A1"
  ► __proto__: Object
```

Рисунок 96

Из скриншота видно, что `otherInfo` это объект. Этот объект хранит свойства `group` и `faculty`.

Также деструктуризация позволяет получать свойства вложенных объектов. Разберемся на примере:

```
let student = {
  name: "Alan",
  surname: "German",
  age: 18,
  group: "A1",
  faculty: "Programming",
  items: {
    programming: 4.7,
    maths: 4.2,
    english: 4.5
  }
};

let {items: {programming, maths, english}} = student;

console.log(programming);
console.log(maths);
console.log(english);
```

Объекту `student` было добавлено новое свойство `items`. Это свойство является объектом, он хранит средний балл

всех предметов, которые посещает студент. Далее происходит деструктуризация. Для того, чтобы получить свойство объекта `items` нужно указать переменную с тем же именем. Далее идет двоеточие и в фигурных скобках перечисляются те свойства, которые нужно получить. Таким образом, в переменных `programming`, `maths` и `english` теперь хранятся данные свойств объекта `items`. Это можно проверить, отобразив их значение на экран. Результат таков:

4.7
4.2
4.5

Рисунок 97

Выше видно, что деструктуризация позволяет удобно получать значения из вложенного объекта.

## Деструктуризация массива

Деструктуризация позволяет разбивать на переменные не только объекты, а еще и массивы. Синтаксисы очень похожи за одним исключением. Когда деструктурируется массив вместо фигурных скобок используется квадратные.

Использование деструктуризации с массивами рассмотрим на примере:

```
let numbers = [1,2,3];  
let [one, two, three] = numbers;  
  
console.log(one);  
console.log(two);  
console.log(three);
```

В примере объявлен массив `numbers`, который содержит в себе числа. Далее происходит деструктуризация массива. В квадратных скобках описывается набор переменных, которые будут хранить в себе значения массива `numbers`. Значения будут записываться по порядку, т.е. первый элемент массива в первую переменную, второй во вторую и т.д. После того как массив был деструктурирован переменные `one`, `two` и `three` отображаются на экране. На изображении показан результат выполнения:

1
2
3

Рисунок 98

Ненужные элементы массива могут быть пропущены. Для этого нужно просто оставить место для переменной пустым. Вот таким образом:

```
let numbers = [1,2,3];  
let [one, , three] = numbers;  
  
console.log(one);  
console.log(three);
```

В примере при деструктуризации массива пропускается переменная `two`. На экран отображается только переменная `one` и `three`:

1
3


Рисунок 99



С массивами тоже можно использовать `rest` оператор. Например:

```
let numbers = [1,2,3,4,5];  
let [one, two, three, ...otherNumbers] = numbers;  
  
console.log(otherNumbers);
```

Теперь массив `numbers` хранит числа от 1 до 5. В деструктуризации объявляются 3 переменные, которые хранят соответствующие числа, а также одна переменная `otherNumbers`. Она будет хранить все оставшиеся числа, а точнее массив из оставшихся чисел. После значение `otherNumbers` выводится на экран:



```
▼ (2) [4, 5] ⓘ  
  0: 4  
  1: 5  
  length: 2  
  ► __proto__: Array(0)
```

Рисунок 100

Из скриншота видно, что `otherNumbers` содержит в себе массив из чисел 4 и 5. Если в массиве меньше элементов чем необходимо присвоить в переменные, то в переменные будет записано значение `undefined`. Чтобы так не происходило, можно использовать значение по умолчанию. Это также делается с помощью знака равно.

В этом примере используется значение по умолчанию:

```
let numbers = [1,2,];  
let [one, two, three = "three" ] = numbers;
```

```
console.log(one);  
console.log(two);  
console.log(three);
```

В примере объявляется массив **numbers**, он содержит в себе элементы **1** и **2**. На следующей строке в деструктуризации объявлено три переменных. Так как переменных больше чем элементов в массиве, то для переменной **three** будет установлено значение, которое было задано по умолчанию. Это можно проверить на это скриншоте:



1
2
three

*Рисунок 101*

При отображении на экране переменной **three** видно, что она содержит строчку «**three**». Именно эта строка является значением, которое было задано по умолчанию при деструктуризации.

# Аргументы по умолчанию

Как известно в JavaScript можно вызывать функцию и не передать ни единого параметра, даже если он необходим. Если параметр не был указан его значение будет **undefined**. Значение **undefined**, хоть и не считается ошибочным, но его появление может приводить к ошибкам. Рассмотрим на примере:

```
function add(num1, num2) {  
    return num1 + num2;  
}  
  
let sum = add(2)  
console.log(sum);
```

В примере объявляется функция **add()**. Она принимает два параметра, эти параметры суммируются и из функции возвращается результат выражения. Ниже вызывается функция **add()** и передается только один параметр. Результат функции записывается в переменную **sum**. После значение **sum** отображается на экране:

A screenshot of a web browser's developer console. It shows a single log entry with the value 'NaN' in blue text. The entry is enclosed in a light gray border with rounded corners.

Рисунок 102

На экране появилось значение **NaN**. А все, потому что, когда вызывалась функция **add()**, в нее был передан только один параметр. В свою очередь второй параметр

принял значение `undefined`. Результатом математических операций со значением `undefined` буде — `NaN`.

Для того чтобы избежать таких непредвиденных обстоятельств ранее использовался такой способ:

```
function add(num1, num2) {  
    num1 = num1 || 0;  
    num2 = num2 || 0;  
    return num1 + num2;  
}  
  
let summ = add(2)  
console.log(summ);
```

В этом примере объявляется функция `add()`. Однако, ее реализация была немного изменена. В начале функции происходит проверка каждого передаваемого параметра с помощью логического оператора `||` (или). Это позволяет узнать хранит ли параметр значение `undefined` и в случае, если хранит присвоить ему подходящее значение, в нашем случае это `0`. Если проверяемый параметр не имеет значения `undefined`, то будет установлено значение этого параметра. Параметр `num1` хранит значение `2`, поэтому при его проверке оператором `||` будет установлено само значение `num1`. Однако, функция `add()` была вызвана только с одним параметром, а это значит что `num2` имеет значение `undefined`, поэтому при его проверке в `num2` будет присвоено `0`. После того как отработала функция, ее результат выводится на экран:

2

Рисунок 103

На скриншоте отображается значение 2. Это потому, что при суммировании параметр `num1` был равен 2, а `num2` равен 0 из-за проверки в начале.

Этот способ использовался раньше, до появления ES6. Именно новый стандарт принес в JavaScript аргументы по умолчанию. Синтаксис аргументов по умолчанию очень прост. Для того чтобы указать значение по умолчанию нужно в объявлении функции сразу после имени параметра поставить знак равно и указать значение. Это значение будет установлено если этот параметр не был передан в функцию при ее вызове.

Вот как нужно устанавливать значение по умолчанию:

```
function add(num1 = 0, num2 = 0) {  
    return num1 + num2;  
}  
  
let summ = add(2)  
console.log(summ);
```

В объявлении функции `add()` каждому параметру указывается значение по умолчанию с помощью знака равно. Если параметр не был указан при вызове функции, то его значение будет не `undefined`, а то, что указано по умолчанию. Это видим на скриншоте:

A screenshot of a console window with a light gray border. Inside, the number '2' is displayed in a blue monospace font.

Рисунок 104

Так, как второй параметр не был указан, его значение стало равно 0, таким образом, при суммировании мы получили значение 2 а не NaN.

Напрашивается вопрос: можно ли указать второй параметр, а первый пропустить и оставить его по умолчанию? Нет, нельзя, так как значение по умолчанию применяется к последним параметрам, у которых отсутствует передаваемое значение.

В примере ниже можно это увидеть:

```
function test(a = 10, b = 20) {  
    console.log(a);  
    console.log(f);  
}  
  
test(1);
```

При объявлении функции `test()` для всех ее параметров указывается значение по умолчанию. Для первого параметра — `10`, а для второго — `12`. Внутри этой функции на экран выводятся значение параметров `a` и `b`. После функция `test()` вызывается, но только с один параметром. Вот что можно увидеть на экране:



1
12

Рисунок 105

Первым на экран отображается значение параметр `a`. Из скриншота видно, что `a` хранит значение — `1`. А значением для второго параметра является значение по умолчанию — `12`. Это из-за того, что, как мы уже говорили, значение по умолчанию применяется к последним параметрам, которые не имеют передаваемого значения. То

есть, интерпретатор увидел, что в функцию был передано одно значение и установил его в первый параметр, а так как передаваемых значений больше нет, последнему параметру было указано значение по умолчанию.

Однако, указать значение по умолчанию для первого параметра можно, указав ему значение `undefined` при вызове функции. Вот как в этом примере:

```
function test( a = 10, f = 12){  
    console.log(a);  
    console.log(f);  
}  
  
test(undefined, 1);
```

В этот раз при вызове функции `test()` первому параметру было явно указано значение `undefined`. А так как передаваемое значение равно `undefined`, то будет применено значение, которое было указано по умолчанию. Таким образом, параметр `a` — равен значению по умолчанию, а параметр `b` — передаваемому значению. Это видно на скриншоте:

10
1

Рисунок 106

# Строковые шаблоны

В ECMAScript 6 были добавлены строковые шаблоны. Это новый вид кавычек для строк, которые обозначаются обратными кавычками «` `». Строка помещается внутрь кавычек. Они позволяют записывать многострочный текст, а также использовать выражения и переменные внутри строк.

До появления строковых шаблонов для того, чтобы строка была многострочной использовался спецсимвол `\n`. Например:

```
let multilineString = "Это \n\  
Многострочный\n\  
Текст";  
  
console.log(multilineString);
```

Выше объявляется переменная `multilineString`, которая будет хранить многострочный текст. Для того, чтобы сделать текст многострочным используется запись `\n`. Эта запись указывается там, где нужен перевод на новую строку.

Результат выполнения кода:

```
Это  
Многострочный  
Текст
```

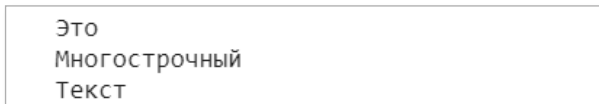
Рисунок 107



Теперь давайте рассмотрим, как выглядит многострочный текст в строковых шаблонах:

```
let multilineString = `Это  
Многострочный  
Текст`;  
  
console.log(multilineString);
```

Для того чтобы создать многострочный текст нужно указать обратные скобки «` `» (на клавиатуре расположены на знаке «~», в верхнем левом углу клавиатуры, комбинация клавиш — [Alt + 96](#) (96 на числовой клавиатуре)), и в них указать текст. А также там, где нужен перенос на новую строку поставить пробел. Результат на экране:



```
Это  
Многострочный  
Текст
```

*Рисунок 108*

Как видно из скриншота текст является многострочным. Мы смогли избавиться от всех спецсимволов `\n`.

Главное преимущество шаблонных строк перед обычными строками — это удобный синтаксис, позволяющий вставлять переменные и выражения прямо в строку.

Рассмотрим пример как в ES5 вставляли значение переменной в строку:

```
let myName = "Rita";  
let sayHello = "Hello " + myName + "!";  
console.log(sayHello);
```

В первой строке примера объявляется переменная `myName` со значением `Rita`. Далее объявляется переменная `sayHello`, которая будет содержать строку с приветствием. Для того, чтобы значение переменной `myName` было в строке происходит конкатенация строк с помощью оператора «+». Далее выводится значение переменной `sayHello`:



Hello Rita!

Рисунок 109

Конкатенация сработала, и мы увидели строку на экране со значением переменной `myName`.

Теперь давайте рассмотрим такой же пример, но с использованием шаблонных строк:

```
let myName = "Rita";  
let sayHello = `Hello ${myName}!`;   
console.log(sayHello);
```

В этом примере обратите внимание на шаблонную строку, а именно на ее синтаксис. Внутри обратных скобок идет текст, а далее указывается знак доллара и фигурные скобки. В фигурных скобках находится переменная значение которой должно быть установлено в строку. Далее переменная `sayHello` выводится на экран:



Hello Rita!

Рисунок 110

На скриншоте видно, что значение переменной `myName` находится прямо в строке. Этот способ выглядит

проще и намного удобнее, по сравнению со способом, который использовался ранее.

Таким же образом можно использовать выражения прямо в строке. На пример:

```
let cost = 15;  
let tax = 3;  
let printCost = 'Вы должны заплатить ${cost + tax}$';  
  
console.log(printCost);
```

В этом примере объявляется переменная `cost`, она хранит значение, которое должен заплатить покупатель. Ниже объявляется переменная `tax`, ее значение — это налог, который должен учитываться при оплате. Далее создается еще одна переменная `printCost`, она будет хранить строку, в которой будет указана конечная стоимость покупки с учетом налога. Для этого в фигурных скобках суммируются две переменные. Таким образом, мы получили строку с конечной стоимостью заказа.

Результат можно увидеть на экране:


A screenshot of a web browser window. The address bar shows a URL starting with 'http://'. The main content area displays the text 'Вы должны заплатить 18\$' in a simple black font. The browser's status bar at the bottom shows 'S'.

Рисунок 111

# Домашнее задание EcmaScript 6

## Задание 1. Разработка класса

Необходимо создать сайт: «Книга контактов»:

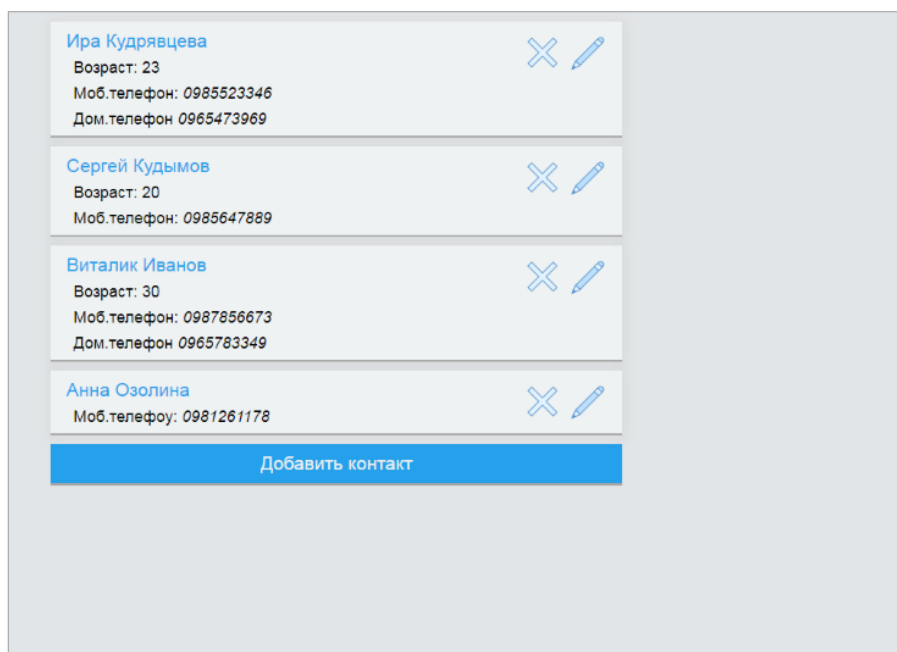


Рисунок 112

Для того чтобы разработать этот сайт нужно создать класс **ContactsBook**.

Класс **ContactsBook** должен содержать в себе список контактов. Для этого нужно создать класс **Contact**. Класс **Contact** будет состоять из таких свойств:

- Уникальный идентификатор;
- Имя;
- Фамилия;
- Возраст (необязательное поле);
- Номер мобильного телефона;
- Номер домашнего телефона (необязательное поле).

На сайте должна присутствовать возможность добавлять новые контакты.

При нажатии на кнопку «Добавить контакт» должно открыться модальное окно с формой добавления нового контакта:

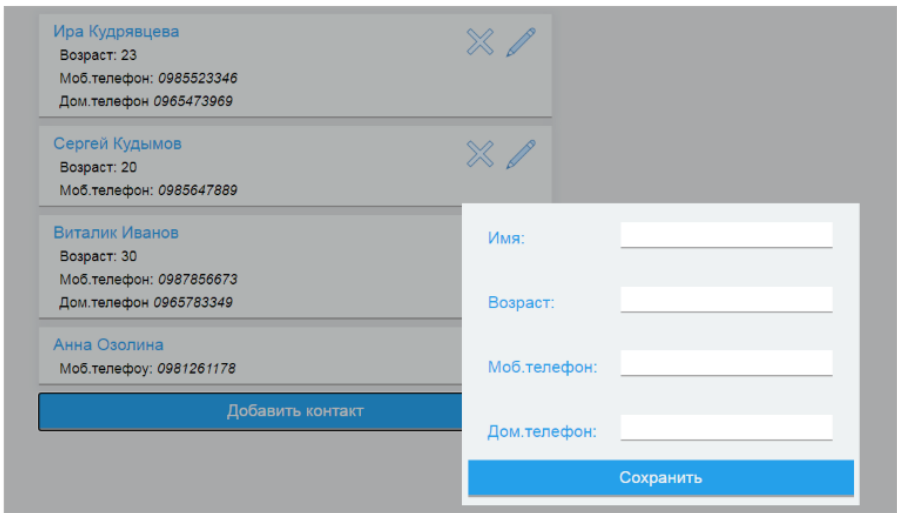


Рисунок 113

Так же нужно реализовать возможность удалять и редактировать контакты.

Нажав на кнопку ✕ должно открываться модальное окно с информацией о удаляемом контакте:

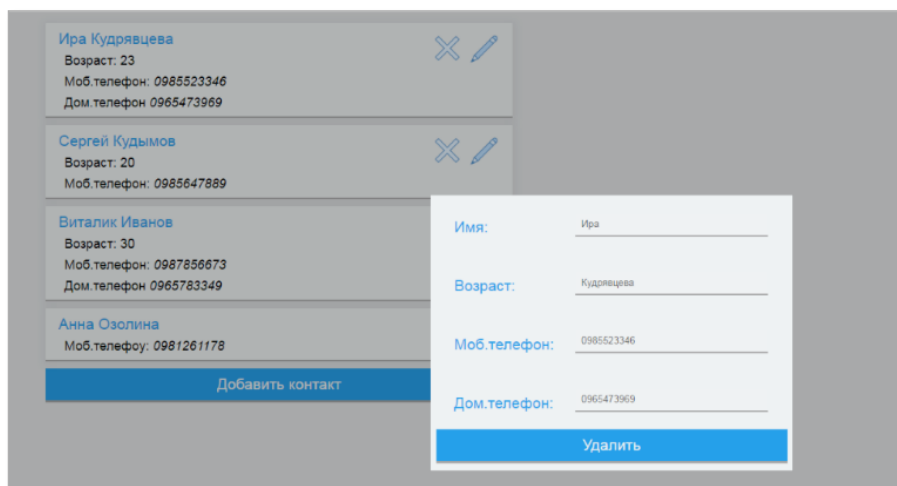



Рисунок 114

Чтобы отредактировать контакт нужно нажать на кнопку  после этого будет открыто модальное окно с информацией о контакте. Чтобы сохранить изменения нужно нажать на кнопку «Редактировать».

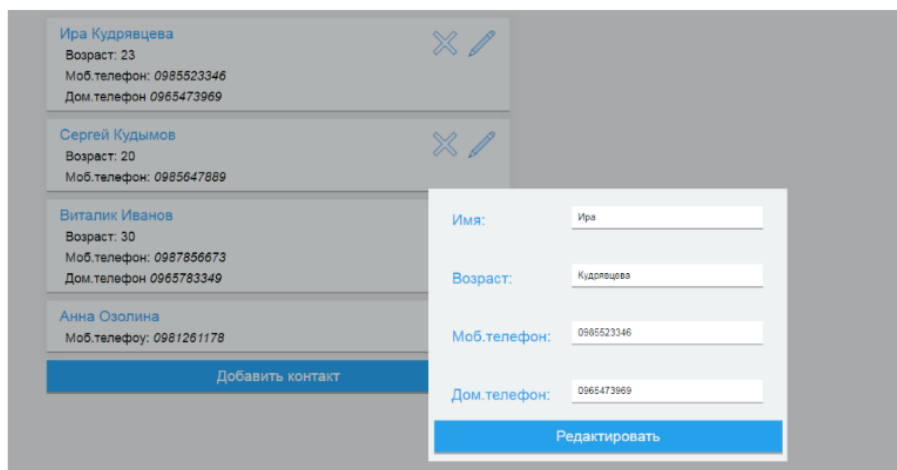


Рисунок 115

## Задание 2. Наследование классов

В этом задании нужно дополнить сайт «Книга контактов». Теперь контакты будут иметь не только номера телефонов, но еще ники в социальных сетях и мессенджерах. Для реализации задания вам нужно создать класс `ContactsAndSocialNetworks`. И унаследовать его от уже созданного класса `Contact`.

Кроме унаследованных полей класс `ContactsAndSocialNetworks` должен иметь такие поля:

- `NicknameTelegram`;
- `NicknameInstagram`;
- `NicknameFacebook`;
- `NicknameViber`;
- `NicknameLinkedin`;
- `NicknameTiktok`;

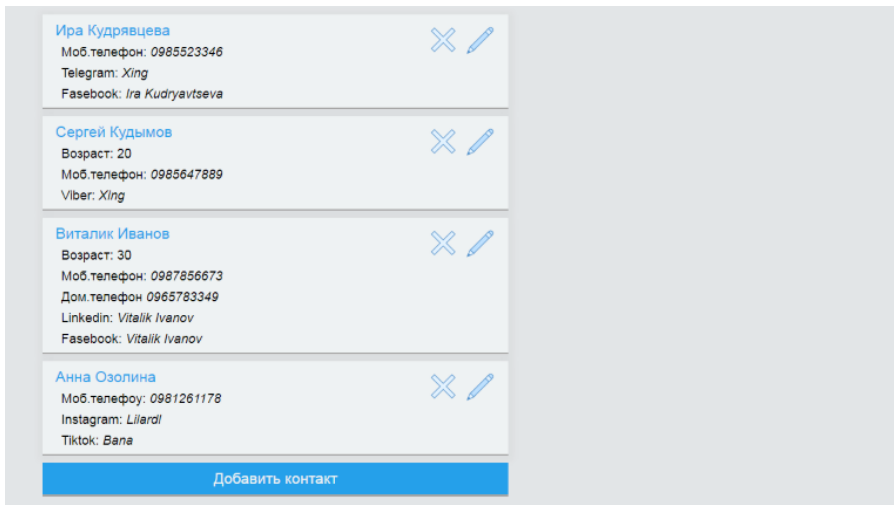


Рисунок 116

Эти поля могут быть необязательными, т.е. могут иметь значение `undefined`. Это значит, что если у контакта нет какой-либо соцсети, но есть другие, то отсутствующие соцсети можно оставить со значением `undefined`. Вот как должен теперь выглядеть сайт (рис. 116).

Кроме наследования в этом задании необходимо использовать модули. Да этого нужно создать три модуля: *ContactBook.js*, *Contacts.js* и *ContactsAndSocialNetworks.js*. В каждом из них нужно объявить соответствующие классы и импортировать те, которые будут необходимы.

### Задание 3. Деструктуризация

Дан объект:

```
let student = {  
  name: 'Ivan',  
  surname: 'Petrov',  
  age: 19,  
  subjects: ["English", "History", "Marketing"],  
  course: 3  
}
```

Необходимо выполнить такие задачи с помощью деструктуризации:

- Записать в переменные все поля объекта и вывести их на экран:
- Записать в переменные все поля объекта, кроме полей `age` и `course`.
- Записать в переменные поля `name`, `surname` и `age`. Так же, если какое-либо значение отсутствует, то в переменную должно записываться значение по умолчанию.



## Задание 4. Параметры по умолчанию

Создайте функцию, которая принимает массив чисел и число. Эта функция должна умножать каждый элемент массива на входящее число и выводить на экран. Нужно задать значение по умолчанию для массива и для числа.



## Урок 6. ECMAScript 6

© Татьяна Гетьман

© Компьютерная Академия «Шаг», [www.itstep.org](http://www.itstep.org).

Все права на охраняемые авторским правом фото-, аудио- и видео-произведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объем и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.