

Язык сценариев **Javascript** и библиотека **jQuery**

Unit 2

Объект. Массивы.
Строки. Объект Date.
Объект Math.
Введение в ООП

Contents

Объекты	4
Что такое объект?	4
Удаление свойств	6
Проверка существования свойства внутри объекта.....	7
Ещё один способ создания объекта со свойствами.....	8
Просмотр всех свойств внутри объекта.....	9
Массивы	10
Что такое массивы?	10
Объект Array.....	10
Создание массива второй способ.....	13
Обращение к элементам массива	14
Свойства и методы массивов	16
Строки	24
Объект String	24
Свойства и методы String	26
Задержки и интервалы	31
Периодический вызов функций	31

Использование математических возможностей	37
Объект Math	37
Свойства и методы объекта Math	37
Случайные числа	39
Объект Date. Обработка даты и времени	41
Что такое ООП?	48
Функции-конструкторы в JavaScript	52
Классы в JavaScript.....	65
Фундаментальные принципы ООП	71
Инкапсуляция	71
Наследование.....	76
Полиморфизм.....	80
Оператор instanceof	85
Расширение стандартных классов.....	86
Домашнее задание	89
Задание 1.....	89
Задание 2.....	89
Задание 3.....	90
Задание 4.....	91

Объекты

Что такое объект?

Вы уже сталкивались с понятием объектов в ранее изученных языках программирования. Однако, лишний раз немного напомнить не повредит.

Объект — это некоторая конкретная реализация какой-то сущности. Например, у нас есть некоторая общая сущность яблоко. Объектом будет конкретное яблоко, лежащее перед нами. Или, например, сущность автомобиль, красный автомобиль марки AUDI, который мы видим перед собой в автосалоне, это объект. Ну и конечно сущности с точки зрения объектно-ориентированного программирования — это классы.

В JavaScript есть поддержка объектов и объектно-ориентированного программирования. Мы обязательно поговорим ещё об этом, но начнём мы немного с другой стороны. JavaScript даёт нам возможность посмотреть на объекты с двух сторон: первая — это объектно-ориентированный стиль, вторая — объекты, как ассоциативные массивы.

И начнем мы со второго подхода. Понятие ассоциативного массива вам уже встречалось ранее. Напомним, это массив, который состоит из пар «ключ-значение». Например, в качестве ключа может быть название стран, а в качестве значения название столиц этих стран. Главное ограничение на такой массив состоит в том, что ключи должны быть уникальными. Посмотрим, как данная концепция реализована в JavaScript.

Первый шаг для нас — создание объекта. Делается это следующим образом:

```
// создаем пустой объект
// мы используем ключевое слово new для создания объекта
var obj = new Object();
// второй вариант создания объекта
var obj2 = {};
```

Вы можете использовать тот или иной механизм в зависимости от своих предпочтений. Для добавления пар «ключ-значение» в объект можно использовать два подхода. Первый подход через стандартный синтаксис массива. В примере ниже мы создаём объекта студента и заполняем его свойствами.

```
// создаем пустой объект
var student=new Object();
// Добавляем свойство в объект используя обычный
// синтаксис массивов
// ["ключ"]
student["Name"] = "Vasya";
// вместо двойных кавычек можно использовать одинарные
student['Age'] = 23;
alert(student["Name"]);
alert(student['Age']);
```

При втором способе вы добавляете свойства в объект, как будто вы уже ранее указывали их.

```
// создаем пустой объект
var firm={};
// создаём свойство внутри объекта
firm.Name = "Star Inc";
```

```
firm.Address = 'Somewhere street 5';
alert(firm.Name);
alert(firm.Address);
```

Вы можете использовать любой из двух способов, но если у вас возникнет необходимость хранить в качестве ключа строку, содержащую с пробелами, то тогда второй синтаксис не подойдет.

```
// создаем пустой объект
var dog = {};
// ниже правильное использование синтаксиса
dog['Name of dog'] = 'Caesar';
alert(dog['Name of dog']);
// строка ниже это синтаксическая ошибка
// dog.Name Of dog = 'Pit';
```

Удаление свойств

Добавленные к объекту свойства можно удалить. Для этого используется **delete**. После удаления свойства, ключ и значение безвозвратно исчезнут.

```
// создаем пустой объект
var cat = {};
// задаём значение свойствам
// синтаксис через точку
cat.Name = "Vasiliy";
// синтаксис, как при работе с массивами
cat["Age"] = 2;

// отображаем значения
alert(cat.Name);
```

```
// можем обращаться в любом синтаксисе
alert(cat.Age);
alert(cat["Age"]);

// удаляем свойства
delete cat.Name;
delete cat["Age"];

// при попытке показа значения мы увидим значение
// undefined
alert(cat.Name);
alert(cat.Age);
```

Проверка существования свойства внутри объекта

Вы уже знаете, что при отображении несуществующего свойства отображается значение `undefined`. А как проверить существует ли свойство в коде?

Для этого можно использовать конструкцию `in`. Она возвращает `true`, если свойство есть внутри объекта.

```
// создали объект
var obj={};
obj.Name = "Oleg";

// проверяем есть ли свойство Age
// in вернет false, так как этого свойства нет
if("Age" in obj){
    alert("Exists");
}
else{
    alert("Not exists");
}
```

Ещё один способ создания объекта со свойствами

Кроме уже известных вам механизмов создания объектов есть ещё один путь. Вы же не думали, что их всего два? Третий вариант создания объекта позволяет сразу создать большое количество свойств внутри объекта. Покажем на примере:

```
// создали объект студента
var student = {
    name: "Daria",
    lastName: "Kislicina",
    age: 23
};

/*
    То же самое, что и выше
    var student = {};
    student.name = "Daria";
    student.lastName = "Kislicina";
    student.age = 23;
*/
alert(student.age);
```

Этот путь очень удобен, когда вам нужно создать описать объект сразу. При этом у вас есть возможность добавить свойство позже, когда оно вам понадобится. При создании объекта можно внутрь вставить другой объект.

```
// создали объект студента
var student = {
    name: "Daria",
    lastName: "Kislicina",
    age: 23,
    address: {
```

```
        street:"Tiraspolskaya 5",
        city:"Odessa",
        country:"Ukraine"
    }
};

alert(student.lastName);
alert(student.address.street);
alert(student.address.city);
```

Просмотр всех свойств внутри объекта

У вас существует возможность узнать все свойства объекта. Для этого необходимо использовать цикл с использованием конструкции `in`.

```
var rect={
    x:0,
    y:0,
    endX:10,
    endY:10
};

// в tempProperty будет попадать название свойства
// такое как x,y,endX,endY

for(var tempProperty in rect){
    // отображаем название свойства
    alert(tempProperty);
    // значение свойства
    alert(rect[tempProperty]);
}
```

Массивы

Что такое массивы?

Мы надеемся, что когда вы читаете данные строки этот вопрос не стоит для вас остро, так как вы уже знакомы с этим понятием из других языков программирования, изученных ранее. Однако, все равно сделаем шаг и назад и вспомним, что такое массив.

Массив — это структура данных, которая группирует набор некоторых значений под одним именем. Для доступа к конкретному значению используется индекс. Индексация в массивах в JavaScript начинается с нуля. Сразу обращаем ваше внимание, что в JavaScript в массиве могут храниться значения разных типов.

В JavaScript массив может быть создан двумя способами. Начнем с конструкции **Array**.

Объект Array

Первый способ создания массива состоит в использовании конструкции **new Array**.

Ниже приводим возможные формы создания:

```
// создаем пустой массив
var arrayName = new Array();

// создаем массив заданной длины
var arrayName= new Array(Number length);

// создаем массив и сразу инициализируем его значениями
var arrayName = new Array(element1, element2, ...
                           elementN);
```

Все три конструкции достаточно просты и не несут никаких сложностей для программиста. Рассмотрим пример записи значений в массив для каждой из форм.

```
// создали пустой массив
var arr = new Array();
// заполнили его значениями
arr[0] = 34;
arr[1] = 99;
arr[2] = 100;

// создали массив с длиной 3
var arr2 = new Array(3);

// заполнили его значениями
arr2[0] = 111;
arr2[1] = 56;
arr2[2] = 73;

// теперь длина массива увеличилась на 2 элемента
arr2[3] = 333;
arr2[4] = 999;

// создали массив и сразу записали в него три значения
var arr3 = new Array("music", "guitar", "apple");

// добавили ещё один элемент
arr3[3] = "lemon";
```

Обратите внимание массивы в JavaScript автоматически увеличиваются в размере, когда это необходимо. Это значит, что вам не придется задумываться о динамическом выделении памяти.

Для того, чтобы узнать длину массива нужно воспользоваться встроенным свойством массива под названием `length`.

```

var arr = new Array(10,20);

// отображаем значение нулевого элемента
// 10
alert(arr[0]);

// с помощью alert можно показывать весь массив сразу
// элементы массива будут перечислены через запятую
alert(arr);

// показываем длину массива
// 2
alert(arr.length);

```

При использовании свойства `length` нужно помнить об одном важном моменте. Свойство `length` — это не количество элементов массива, а значение последнего индекса **+1**.

```

var arr = new Array();

// реальных заполненных элементов один
arr[499] = 86;

// на экране 500
// length это последний индекс + 1
alert(arr.length);

// при обращении к неинициализированному элементу
// отобразится undefined
alert(arr[0]);

```

Вы уже знаете, что массивы растут в размере самостоятельно, когда это требуется кодом скрипта. А что делать в том случае, если вам нужно уменьшить размер массива?

ва? Ответ очевиден! Надо уменьшить значение `length` до нужной длины.

```
var arr = new Array(11, 74, 35);

// делаем размер массива равным 2
// значение 35 потеряно навсегда
arr.length = 2;

// на экране 11,74
alert(arr);

// а теперь размер массива 5, но
// заданы значения только двум элементам
// arr[0] = 11
// arr[1] = 74
arr.length = 5;
alert(arr);

// теперь размер массива 0
// все значения утеряны навсегда
arr.length = 0;
```

Создание массива второй способ

Первый способ создания массивов немного громоздкий. Второй способ привычней для нас. Рассмотрим его:

```
// создаём пустой массив
var arrayName = [];

// создаём массив с набором значений
var arrayName = [element1, element2, ..., elementN];
```

Интересным моментом является использование `[]` для создания массива. Раньше вы сталкивались с исполь-

зованием {} для той же цели. Теперь попробуем на примере новые конструкции.

```
// создаём пустой массив
var arr = [];

// записали в него 2 элемента
arr[0] = 11;
arr[1] = 12;
// 11,12
alert(arr);

// создаём массив с тремя элементами
var arr= [88,99,111];
// 88,99,11
alert(arr);

// создаём массив с тремя элементами
var cars = ["BMW", "Audi", "Toyota"];
// "BMW", "Audi", "Toyota"
alert(cars);
// 3
alert(cars.length);
```

По этому коду можно сделать простой вывод: такая форма создания массива удобней первого способа с [Array](#).

Обращение к элементам массива

Мы уже умеем обращаться к элементам массива используя индекс. Добавим в наш багаж пример перебора массива с помощью цикла.

```
var arr = [2,9,33,1];
var amt = 0;
// считаем в цикле сумму элементов массива
```

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

// результат 45
alert(amt);
```

И ещё пример. Отобразим содержимое массива с элементами разного типа.

```
var arr = [33,"sun",12,"planet"];
for(var i = 0;i<arr.length;i++){
    alert(arr[i]);
}
```

Теперь настало время создать двумерный массив.

```
// создали двумерный массив 2 строки 3 столбца
// 1 3 5
// 2 7 8
var arr = [
    [1,3,5],
    [2,7,8]
];

// 1
alert(arr[0][0]);
// 8
alert(arr[1][2]);
```

Обычные переменные в JavaScript передаются внутрь функции по значению. Это значит, что при передаче переменной в функцию создаётся копия переменной, если вы измените значение этой копии это никак не повлияет

на оригинальную переменную. Массивы же передаются в функцию по ссылке. Так происходит, потому что массив — это объект, а объекты всегда передаются внутрь функций по ссылке. Отсюда можно сделать вывод, что изменения значений элементов массива сохраняются при выходе из функции.

```
// функция записывает новое значение по указанному
// индексу
function SetValue(arr, index, newValue) {
    arr[index] = newValue;
}
var arr = [88,11,3];

// 88,11,3
alert(arr);
SetValue(arr,0,999);

// 999,11,3
alert(arr);
```

Свойства и методы массивов

У массивов в JavaScript есть большое количество встроенных методов. Мы познакомимся с некоторыми из них.

Начнем с методов для поиска. Это [indexOf](#) и [lastIndexOf](#). Первый ищет совпадение в массиве слева направо. Сигнатура метода:

```
name_of_array.indexOf(what_to_search[, fromIndex])
```

- `what_to_search` — значение, которое мы ищем в массиве;

- `fromIndex` — необязательный параметр, который используется для указания стартового индекса. Если мы не указываем его, то поиск начнется с нулевого индекса.

Если искомое значение найдено метод возвращает индекс найденного значения, если же значения нет в массиве, то метод вернет `-1`.

Продемонстрируем работу этого метода на примерах:

```
var arr = [1,45,-3,78,1];

// ищем значение 45
var index = arr.indexOf(45);

// на экране индекс 1
alert(index);

// ищем значение, которого нет в массиве
index = arr.indexOf(99);

// на экране -1, так как 99 нет в массиве
alert(index);
```

А теперь давайте посмотрим пример подсчета сколько раз некоторое искомое значение встречается в массиве.

```
var arr = [12,45,-3,82,12,78,12];

// счетчик для подсчета количества раз вхождения
// искомого значения в массив
// искать будем значение 12
var counter = 0;
var index = arr.indexOf(12);
while(index != -1){
    counter++;
```

```
//двигаемся дальше по массиву за счет изменения
//индекса на значение index+1
index = arr.indexOf(12, index+1);
}

//на экране 3
alert(counter);
```

Метод `lastIndexOf` работает по схожему индексу, но ищет справа налево. Это значит, что поиск начинается с последнего доступного индекса. Сигнатура метода:

```
name_of_array.indexOf(what_to_search[, fromIndex])
```

- `what_to_search` — значение, которое мы ищем в массиве;
- `fromIndex` — необязательный параметр, который используется для указания стартового индекса. Если мы не указываем его, то поиск начнется с самого последнего индекса.

Если искомое значение найдено метод возвращает индекс найденного значения, если же значения нет в массиве, то метод вернет `-1`.

Давайте на примере разберем отличия этого метода.

```
var arr = [12, 45, -3, 82, 12, 78, 12];
//ищем 12 справа налево
var index = arr.lastIndexOf(12);
//на экране 6
alert(index);
index = arr.lastIndexOf(77);
//на экране -1
alert(index);
```

А теперь посчитаем сколько раз встречается искомое значение в массиве при помощи `lastIndexOf`.

```
var arr = [12,45,-3,82,12,78,12];
// счетчик для подсчета количества раз вхождения
// искомого значения в массив
// искать будем значение 12
var counter = 0;
var index = arr.lastIndexOf(12);
while(index != -1){
    counter++;
    // мы проверяем на ноль
    // так как ниже начинаем с index-1
    // при 0 мы получим старт -1
    // для метода lastIndexOf отрицательный индекс
    // означает искать с конца массива
    if(index == 0)
        break;
    //двигаемся дальше по массиву за счет изменения
    // индекса на значение индекс-1
    index = arr.lastIndexOf(12, index-1);
}
// на экране 3
alert(counter);
```

При работе с данными очень важно иметь механизм их сортировки. И такой механизм представлен в виде метода `sort`, который имеет следующую сигнатуру:

```
name_of_array.sort([compareFunc])
```

- `what_to_search` — значение, которое мы ищем в массиве;
- `compareFunc` — необязательный параметр, имя пользовательской функции, которая будет использована

для сортировки массива. Если её не указывать данные в массиве будут отсортированы по правилам сортировки строк

Начнем в наших примерах с сортировки по умолчанию.

```
var arr = [10,1,3,33,6];
arr.sort();
// 1 10 3 33 6
alert(arr);
```

В результате сортировки по умолчанию **10** стоит перед **33**, так получилось из-за строковой сортировки. Если вам нужна числовая сортировка, то нужно будет реализовать функцию сортировки. Возможно вы уже сталкивались с таким решением в других языках программирования. Общая форма данной функции:

```
function name_of_function(name_of_var1, name_of_var2) {
    body_of_function
}
```

Вы можете назвать функцию любым именем. Функция должна принимать два параметра. Это две значения массива, которые в данный момент сравниваются алгоритмом сортировки. Предположим вы хотите сортировать элементы массива по возрастанию, тогда правила сортировки для вас такие,

- если **name_of_var1 > name_of_var2**, возвращаемое значение из функции положительное (обычно используют 1);

- если `name_of_var1 < name_of_var2`, возвращаемое значение из функции отрицательное (обычно используют `-1`);
- если равны, то обычно возвращают `0`.

Для сортировки по убыванию в первом случае возвращайте отрицательное значение, а во втором положительное.

Рассмотрим пример сортировки данных по возрастанию:

```
function compareFunc(a,b) {  
    if(a>b)  
        return 1;  
    else if(b>a)  
        return -1;  
    else  
        return 0;  
}  
var arr = [10,1,3,33,6];  
  
// 1 3 6 10 33  
arr.sort(compareFunc);
```

А теперь сортировка по убыванию

```
function compareFunc(a,b) {  
    if(a>b)  
        return -1;  
    else if(b>a)  
        return 1;  
    else  
        return 0;  
}  
var arr = [10,1,3,33,6];
```

```
// 33 10 6 3 1
arr.sort(compareFunc);
alert(arr);
```

Рассмотрим следующую задачу: у нас есть строка и нам нужно её конвертировать в массив, разбив на элементы массива на основе какого-то разделителя. Для того, чтобы это сделать используем метод строки под названием **split**. Его сигнатура:

```
name_of_string.split(separator)
```

- **separator** — сепаратор с его помощью мы разбиваем строку на элементы массива. В результате работы метода возвращается массив результата.

Этот метод можно использовать, например, при работе с текстом. Разобьем строку с помощью вызова **split**.

```
var str = "apple,onion,strawberry";
// разбиваем на основании ,
// в массиве будет три элемента apple onion strawberry
var arr = str.split(',');
// apple на экране
alert(arr[0]);
```

Обратите внимание, что **split** это строковый метод. У массива есть обратный метод **join**. Он используется для конвертирования массива в строку. Например:

```
var arr = ["bmw", "audi", "opel"];
// создаем строку
// в качестве разделительного символа между элементами
```

```
// массива указываем *
var str = arr.join(" * ");
// bmw*audi*opel
alert(str);
// если не указать разделитель, то будет использована,
var str2 = arr.join();
// bmw,audi,opel
alert(str2);
```

Мы привели лишь часть методов массива в JavaScript. С остальными вы можете ознакомиться тщательно изучив MSDN или документацию по [ссылке](#).

Строки

Объект String

В JavaScript не существует отдельных типов для строк и символов. Есть только строковый тип и если вам нужно работать с одним символом вы создаете строку с одним символом. Содержимое строки может быть заключено в двойные или одинарные кавычки. Для JavaScript между ними нет разницы. Вы можете выбрать любой стиль, хотя с нашей точки зрения использование двойных кавычек более привычно. Пример создания строк:

```
// используем двойные кавычки
var str = "Test string";
alert(str);

// используем одинарные кавычки
var str2 = 'New string';
alert(str2);
```

Внутренним форматом хранения строк в JavaScript является Unicode. И при этом совершенно неважно какая у вас кодировка вашей страницы. Вы можете включать в свою строку специальные символы. Например, escape-последовательности. Пример строк с такими символами:

```
// \t – табуляция
var str = "Sun\t is going \\down\\\";
// Sun is going \down\
alert(str);
```

```
// для вставки кавычек в строку надо использовать \"  
var str2 = "\"Yes\"";  
  
// на экране "Yes"  
alert(str2);
```

Для доступа к конкретному элементу строки необходимо использовать уже известные вам `[]`.

```
var str = "trees and fruits";  
  
// на экране trees and fruits  
alert(str);  
  
// на экране t  
alert(str[0]);  
  
// на экране s  
alert(str[4]);
```

Строки в JavaScript являются неизменяемыми объектами. Это значит, что после создания строки вы не можете изменить i-ый элемент строки (например, элемент по индексу `0` или `5`). Вы можете пересоздать строку только целиком!

```
var str = "trees and fruits";  
  
// на экране trees and fruits  
alert(str);  
str[0] = "Z";  
  
// на экране все равно t  
alert(str[0]);
```

```

str[4] = 'W'

// на экране s
alert(str[4]);
str = "Here is a car";

// на экране Here is a car
alert(str);

```

Для конкатенации строк используется оператор **+**.

```

var first = "boat";
var second = "river";
var result = first + " and " + second;

// boat and river
alert(result);

```

Свойства и методы String

У строки есть большое количество свойств и методов, которые очень полезны для решения той или иной задачи. Начнем с самого очевидного: свойство **length** используется для получения длины строки:

```

var str = "gold";

// 4
alert(str.length);

```

Вы не можете уменьшить или увеличить длину строки задав значение **length** явно. Это не приведет к желаемому результату.

Метод `charAt` является аналогом `[]` и используется для доступа к i -му элементу массива. Отличие между `charAt` и `[]` состоит в том, что при обращении к несуществующему индексу `charAt` возвращает пустую строку, а `[]` значение `undefined`.

```
var str = "gold";
// g
alert(str.charAt(0));

// пустая строка
alert(str.charAt(15));

// undefined
alert(str[15]);
```

Методы `toLowerCase` и `toUpperCase` изменяют регистр букв. Метод `toLowerCase` меняет регистр на нижний, а `toUpperCase` меняет регистр на верхний. Обратите внимание, что новое значение возвращается из методов, но при этом содержимое самой строки не меняется.

```
var str = "Football";
var newStr = str.toLowerCase();

// football
alert(newStr);

// Football
alert(str);
```

Ранее из урока вы уже узнали о методах `indexOf` и `lastIndexOf`, которые использовались для поиска по

массиву. У строки есть свои версии этих методов, которые ведут себя также, как и их аналоги в массиве. Пример поиска значения в строке:

```
var str = "earth and sun";
var index = str.indexOf("sun");

// значение индекса равно 10
alert(index);
index = str.indexOf("moon");

// значение индекса равно -1
// так как moon нет в строке
alert(index);
```

А теперь посчитаем сколько раз некоторое слово встречается в строке:

```
var str = "test it is test sun test no";
var counter = 0;
var wordToFind = "test";
var index = str.indexOf(wordToFind);
while(index != -1){
    counter++;
    index = str.indexOf(wordToFind, index+1);
}
// 3
alert(counter);
```

Схожий пример поиска вы уже видели в разделе о массивах.

Методы **substr**, **substring** используются для получения подстроки из строки.

Начнем с **substring(start, [end])**. Метод возвращает подстроку начиная с индекса **start**, но не включая ин-

декс `end`. Если `end` не указан, то возвращаем подстроку до конца строки.

```
var str = "Some value";
// end – необязательный параметр, который можно опустить
var newStr = str.substring(2);

// me value
alert(newStr);
newStr = str.substring(1,3);

// om
alert(newStr);
```

Метод `substr(start, [length])` работает немного по другому. Он возвращает подстроку начиная с `start`, при этом можно указать длину подстроки во втором параметре. Если же длина не указана, то возвращается подстрока до конца оригинальной строки.

```
var str = "Some value";
// length – необязательный параметр, который можно
// опустить
var newStr = str.substr(2);

// me value
alert(newStr);
newStr = str.substr(1,3);

// ome
alert(newStr);
```

Для сравнения строк с учетом локали используется метод `localCompare(compareValue)`. Если строки между

себой равны метод вернет **0**, если строка, которая вызывала метод больше строки в параметре метод вернет **1**, иначе **-1**. Принципы сравнения такие же, как у известной вам из языка С функции **strcmp**.

```
var str = "cheese";  
  
// попадем в if, так как строки равны между собой  
if(str.localeCompare("cheese")==0){  
    alert("Strings are equal! Cheese!");  
}  
else{  
    alert("Strings are not equal! Not cheese!");  
}  
  
str = "Fb";  
  
// первая строка больше, так как сравнение происходит  
// посимвольно до первого несовпадения  
// F больше по числовому коду чем f  
// поэтому первая строка больше  
if(str.localeCompare("fb")>0){  
    alert("First is greater");  
}  
else{  
    alert("Equal or less than second string");  
}
```

Задержки и интервалы

Периодический вызов функций

Иногда возникает необходимость вызвать функцию через определенное количество времени. Например, если вы реализуете онлайновую игру и дали игроку 60 секунд на решение пазла, вам нужно иметь механизм, который отсчитает эти 60 секунд, после чего игра сообщит пользователю, что время вышло. В обычной жизни такой механизм называется таймером. Вы можете ежедневно наблюдать работу таймера в микроволновой печи, когда производите разогрев чего-либо в ней.

Для реализации таймера в JavaScript существует набор функций. Начнем с функции установки таймера.

```
setTimeout (функция/код, задержка, аргумент1, аргумент2, ...)
```

Функция `setTimeout` используется для установки таймера. Функция/код — функция или код, который запустится через указанный промежуток времени.

Задержка — время задержки для таймера. После истечения времени будет запущена функция/код из первого аргумента. Задержка указывается в миллисекундах. Для справки: *1000 миллисекунд = 1 секунде*.

Аргумент1, аргумент2, ... — аргументы, которые можно передать в функцию, которая сработает, когда указанная задержка закончится.

Функция `setTimeout` возвращает идентификатор установленного таймера. Его можно использовать для того, чтобы отменить срабатывание таймера.

Начнем знакомство с таймером с самых простых примеров. В этом примере кода таймер сработает через секунду и будет вызвана функция `HelloWorld`, указанная в параметре.

```
function HelloWorld() {
    alert("Hello world!");
}

setTimeout(HelloWorld, 1000);
```

Попробуем передать аргументы в функцию:

```
function Sum(a,b) {
    alert(a+b);
}

setTimeout(Sum, 1000,1,2);
```

Мы передали значения `1` (попало в параметр `a`) и `2` (попало в параметр `b`). Функция посчитала сумму двух чисел и показала её в окне сообщения.

Когда мы описывали параметры для `setTimeout` мы говорили, что первый параметр может быть функцией или просто кодом. Например:

```
setTimeout("alert('hi')", 1000);
```

Здесь в качестве кода используется `alert`. Указывать код в таком виде не рекомендуется, так как это нечитабельно и минимизаторы кода при обработке такого фрагмента могут неправильно его интерпретировать. Альтернативой является использование безымянных (анонимных) функций.

```
setTimeout(function() {
    alert("Hi!");
}, 1000);
```

В примере выше мы создали анонимную функцию с одной строкой кода для отображения окна сообщения. Такие анонимные функции можно создавать и с параметрами.

```
setTimeout(
    function(a,b) {
        alert(a*b);
    },
    1000, 3, 7
);
```

После установки таймера возвращается идентификатор. Его можно использовать для отмены вызова функции.

```
var id = setTimeout(function() {
    alert("Boom!");
}, 3000
);
alert("id timer: "+id);
```

В этом примере мы отобразили идентификатор таймера. В следующем примере мы используем полученный идентификатор для отмены таймера. Функция отмены таймера `clearTimeout`.

```
clearTimeout(идентификатор_таймера)
var id = setTimeout(function() {
    alert("Boom!");
}, 50000
);
clearTimeout(id);
```

Мы установили таймер на 50 секунд, а потом отменили его. Естественно в результате наших действий работа таймера будет остановлена.

А что же делать, если нам необходимо срабатывание некоторой функции через определенные интервалы времени? Механизм, который мы разобрали выше позволяет создать функцию, которая будет вызвана единожды. Для решения нашей проблемы можно использовать два пути.

Первый способ это использование функции `setInterval`.

```
setInterval(функция/код, интервал_времени, аргумент1,
            аргумент2, ...)
```

Посмотрев на функцию можно легко сделать вывод о том, что она полностью схожа с `setTimeout`. Отличие только в принципе действия. В отличии от `setTimeout` функция/код, указанная в первом параметре будет регулярно вызываться через указанный интервал времени. Для остановки вызова функции нужно будет вызвать функцию `clearInterval`. Попробуем использовать интервальный механизм.

```
setInterval(
    function() {
        alert("Boom!");
    },
    2000);
```

Функция, указанная в первом параметре будет вызываться каждые две секунды. В этом коде мы не вызвали `clearInterval` поэтому окно сообщения с надписью `Boom` будет вызываться каждые две секунды, пока окно или

вкладка браузера не будет закрыта. Теперь добавим вызов `clearInterval`.

```
var id = setInterval(IntervalFunc, 2000);
var counter = 0;

function IntervalFunc() {
    if(counter == 3) {
        clearInterval(id);
        return;
    }
    counter++;
    alert("Boom");
}
```

Окно сообщения будет показано три раза, после чего мы остановим интервальную функцию. Альтернативой использованию `setInterval` является рекурсивный вызов `setTimer`. Например:

```
var id = setTimeout(TimeOutFunc, 2000);
var counter = 0;

function TimeOutFunc() {
    // если таймер сработал уже трижды
    // останавливаем процесс
    if (counter == 3) {
        clearTimeout(id);
        return;
    }
    counter++;
    alert("Boom Timer");

    // ставим заново таймер на две секунды
    id = setTimeout(TimeOutFunc, 2000);
}
```

Принцип использования таймера вместо интервалов прост: внутри таймерной функции мы заново ставим таймер. В нашем примере мы заново устанавливали таймер на две секунды внутри таймерной функции. Когда наш цикл работы исполнится три раза мы вызываем `clearTimeout` и выходим из таймерной функции. При установке нового таймера мы не обязаны указывать ту же задержку, что и при первом вызове. Например:

```
var id = setTimeout(TimeOutFunc, 2000);
var counter = 1;
function TimeOutFunc() {
    alert("Boom Timer");
    switch(counter) {
        case 1:
            id = setTimeout(TimeOutFunc, 5000);
            break;
        case 2:
            id = setTimeout(TimeOutFunc, 10000);
            break;
        case 3:
            clearTimeout(id);
            return;
    }
    counter++;
}
```

В нашем случае у каждого из таймеров своя задержка: 2, 5, 10 секунд.

Для создания интервальной функции можно использовать `setInterval` или `setTimeout`. Выбор за вами. Мы можем добавить лишь то, что `setTimeout` несколько гибче, так как у вас есть возможность каждый раз указывать новую длительность задержки.

Использование математических возможностей

Объект Math

При решении задач из области программирования вам могут понадобиться математические возможности, встроенные в JavaScript. Как же можно воспользоваться ими? Сложно ли это сделать? Ответ: нет. Для того, чтобы использовать математические возможности JavaScript вам необходимо обратиться к встроенному объекту [Math](#). Внутри этого объекта вы найдете целую мириаду функций для решения той или иной математической проблемы, стоящей перед вами. Давайте начнем знакомство с внутренним устройством этого объекта.

Свойства и методы объекта Math

Мы не будем рассматривать все свойства и методы объекта [Math](#). Начнем с некоторых свойств:

- [Math.PI](#) — возвращает число ПИ (приблизительно 3.14);
- [Math.E](#) — возвращает число Эйлера (приблизительно 2.718);
- [Math.SQRT2](#) — возвращает квадратный корень из двух (приблизительно 1.414);
- [Math.SQRT1_2](#) — возвращает квадратный корень из одной второй (приблизительно 0.707).

Пример использования:

```
alert(Math.PI);
alert(Math.E);
```

Теперь давайте рассмотрим некоторые методы:

- **Math.ceil(x)** — возвращает округление параметра **x** вверх до ближайшего целого;
- **Math.floor(x)** — возвращает округление параметра **x** вниз до ближайшего целого;
- **Math.round(x)** — возвращает округленное значение параметра **x** до ближайшего целого (если дробная часть больше или равна **0,5** тогда округление вверх, иначе вниз);
- **Math.pow(x, y)** — возвращает значение **x** возведенное в степень **y**;
- **Math.sqrt(x)** — возвращает квадратный корень из **x**;
- **Math.min()** и **Math.max** — возвращают соответственно минимум и максимум из переданных параметров;
- **Math.abs(x)** — возвращает модуль переданного параметра **x**;

Пример использования функций по округлению и усечению значений:

```
var res = Math.ceil(4.3);
// 5
alert(res);
res = Math.round(4.3);
// 4
alert(res);
```

```
res = Math.round(4.5);  
// 5  
alert(res);  
res = Math.floor(4.5);  
// 4  
alert(res);
```

Пример использования функций по поиску максимума и минимума:

```
var res = Math.min(1, 4, -1, -9, 20);  
// -9  
alert(res);  
res = Math.max(21, 3, 4, 5, 6, 7);  
// 21  
alert(res);
```

Мы уверены, что знакомство и использование других математических методов не составит для вас сложности.

Случайные числа

В жизни любого программиста наступает момент, когда приходится иметь дело со случайными числами. Например, вам необходимо создать программное обеспечение для лотереи или для игровых автоматов. В этом случае вам нужен будет механизм, который будет генерировать вам случайные числа. Для генерации псевдослучайных чисел в JavaScript используется метод `random` объекта `Math`.

- `Math.random()` — возвращает псевдослучайное число в диапазоне от `0` до `1`;

Пример использования:

```
var res = Math.random();  
// псевдослучайное число  
alert(res);  
res = Math.random();  
// псевдослучайное число  
alert(res);
```

Обратите внимание, что число псевдослучайное. Это означает, что число, которое будет возвращено будет сгенерировано алгоритмически. Обычно в основе алгоритмов, генерирующих случайные числа, лежит понятие начальной точки. По умолчанию обычно при первом вызове функции для генерации случайного числа в качестве начальной точки берут количество миллисекунд, прошедших с 1 января 1970 года. В некоторых языках вы можете изменить начальную точку. В JavaScript данная возможность скрыта. Каким образом можно сгенерировать псевдослучайное число в другом диапазоне, отличном от 0 и 1. Для решения этой задачи используется связка `random` и `floor`. Например:

```
// случайное значение от 0 до 9  
alert(Math.floor(Math.random()*10));  
// случайное значение от 0 до 11  
alert(Math.floor(Math.random()*11));  
// случайное значение от 1 до 10  
alert(Math.floor(Math.random()*10)+1);
```

Как видно из примера выше использование случайных чисел не несет большой сложности.

Объект Date. Обработка даты и времени

Объект **Date** в JavaScript используется для управления датой и временем. С помощью этого объекта можно получать данные о текущей дате, а также дате в прошлом или в будущем.

Создается экземпляр объекта **Date** с помощью ключевого слова **new**. Если в скобках вы не передаете данные, то переменная инициализируется текущими данными даты и времени с компьютера пользователя, загрузившего файл со скриптом.

```
const today = new Date();
console.log(today);
```

При выводе в консоль вы увидите дату в формате «День недели, месяц, день год часы:минуты:секунды часовской пояс по Гринвичу» примерно так:

```
Sat Aug 07 2021 20:22:50 GMT+0300
```

Рисунок 1

Получить текущую дату можно, написав

```
Date.now()
```

В этом случае вы получите, например, число **1628357359903** — это количество миллисекунд, прошед-

ших с 1 января 1970 г. Эта дата считается «точкой отсчета» для всех дат в JavaScript.

Также количество миллисекунд можно получить с помощью метода `getTime()` или `valueOf()`.

Если вас интересует какая-то определенная дата, то при создании переменной в скобках нужно указать ее в одном из строковых форматов:

```
let date1 = new Date("2021-05-17"); // год-месяц-день
console.log(date1);
let date2 = new Date("06/25/2021"); // месяц/день/год
console.log(date2);
const date3 = new Date('November 2, 1999 13:25:00');
console.log(date3);
const date4 = new Date('1999-11-02T13:25:00');
console.log(date4);
const date5 = new Date('02 November 1999 13:25:00');
console.log(date5);
```

Если вы укажете неверную дату, то JavaScript при попытке использовать дату вернет строку '*Invalid Date*'.

Вы также можете задавать дату в виде нескольких чисел через запятую:

```
const date6 = new Date(2022, 0, 12, 03, 45, 12, 500);
// год, месяц 0-11, день, часы, минуты, секунды,
// миллисекунды
console.log(date6);
```

Обратите внимание на то, что год нужно задавать в виде 4-х цифр, месяцы указываются от 0 (январь) до 11 (декабрь), день от 1 до 31 (если не УКАЗАН, то будет 1),

часы, минуты, секунды и миллисекунды можно не указывать, тогда вместо них будут подставлены 0.

Созданный любым из перечисленных методов объект **Date** содержит число миллисекунд, прошедших с 1 января 1970 г. Поэтому любые экземпляры объекта **Date** можно вычесть друг из друга, узнавая разницу между ними в миллисекундах, а затем переводить в часы, дни, годы и т.п., например:

```
console.log('Разница между датами в миллисекундах ',
            date2 - date1);
console.log('Разница между датами в днях ',
            Math.round((date2 - date1)/24/60/60/1000));
```

В консоли мы увидим такой результат для объявленных выше переменных и **date2**:

*Разница между датами в миллисекундах 3358800000
Разница между датами в днях 39*

Для любого из представленных способов вы можете узнать количество миллисекунд с 1 января 1970 года, использовав метод **Date.parse()**, указав в скобках дату в виде строки.

```
console.log(Date.parse('11/08/2025'));
```

Еще один способ создания даты подразумевает, что в скобках вы указываете количество миллисекунд:

```
const date7 = new Date(1728357351109);
// Tue Oct 08 2024 06:15:51
console.log(date7);
```

Разные способы создания дат позволяют манипулировать ими в различных задачах.

Например, нам нужно получить переменную с текущей датой, т.е. сегодняшнем днем, а также переменные с вчерашней и завтрашней датой. Проще всего это сделать на основе последнего способа создания экземпляра объекта [Date](#).

```
let today = new Date();
let yesterday = new Date(today - 24 * 60 * 60 * 1000);
let tomorrow = new Date(today + 24 * 60 * 60 * 1000);
console.log(yesterday, today, tomorrow);
```

Объект [Date](#) имеет ряд методов, с помощью которых вы можете получить доступ к отдельным данным из переменной-даты:

- [getFullYear\(\)](#) — возвращает год, состоящий из 4 цифр
- [getMonth\(\)](#) — возвращает номер месяца от 0 (январь) до 11 (декабрь).
- [getDate\(\)](#) — возвращает день месяца, от 1 до 31.
- [getHours\(\)](#), [getMinutes\(\)](#), [getSeconds\(\)](#), [getMilliseconds\(\)](#) — возвращают соответственно, часы, минуты, секунды или миллисекунды.
- [getDay\(\)](#) — возвращает день недели: **0** соответствует воскресенью, **1** — понедельнику, **2** — вторнику и так далее.

Типичная задача на использование одного из методов — это определение того, на какой день недели приходился день рождения пользователя.

```
let userdata = prompt("Enter your birthday in  
year-month-day format", "2002-08-14");  
let birthday = new Date(userdata) == 'Invalid Date' ?  
    new Date() : new Date(userdata);  
console.log(birthday);  
let days = ['sunday', 'monday', 'tuesday', 'wednesday',  
           'thursday', 'friday', 'saturday'];  
alert("You were born on " + days[birthday.getDay()]);
```

Методы объекта **Date** также позволяют установить дату и время. Для этого существуют такие методы:

- [setFullYear\(\)](#) — устанавливает год, состоящий из 4 цифр. Необязательными параметрами являются месяц и день месяца.
- [setMonth\(\)](#) — устанавливает номер месяц, от 0 (январь) до 11 (декабрь). Необязательным параметром является день месяца.
- [setDate\(\)](#) — устанавливает день месяца, от 1 до 31.
- [setHours\(\)](#),[setMinutes\(\)](#),[setSeconds\(\)](#),[setMilliseconds\(\)](#) — устанавливают соответственно, количество часов, минут, секунд или миллисекунд.
- [setTime\(\)](#) — устанавливает количество миллисекунд, прошедших с 1 января 1970 00:00:00 по UTC.

Все методы из списка выше возвращают результат для местной временной зоны, в которой часто присутствует «добавка» в виде GMT+0300 или т.п., что означает, что к времени нулевого (Гринвичского) меридиана прибавляется или отнимается некоторое количество часов. При создании экземпляров объекта **Date** нужно учиты-

вать, что поверхность земли разделена на 24 часовых пояса. В JavaScript же существует всего два часовых пояса:

- **Local Time** — часовой пояс, в котором находится ваш компьютер
- **UTC (Coordinated Universal Time)** — то же самое, что время по Гринвичу (GMT)

Поэтому в JavaScript существуют UTC-варианты методов, возвращающие день, месяц, год и т.п. для зоны GMT+0 (UTC), то есть Гринвичский меридиана: `getUTCFullYear()`, `getUTCMonth()`, `getUTCDay()`. То есть, сразу после «`get`» вставляется «`UTC`».

Кроме того, в объекте `Date` есть еще методы, позволяющие вывести дату в том виде, как она представлена на машине пользователя, то есть в локальном формате с учетом смещения по времени относительно Гринвичского меридиана:

- `toLocaleString()` — возвращает строку даты и времени в локальном формате, то есть в формате той временной зоны, в которой находится операционная система пользователя. Для русскоязычной зоны, например, формат даты будет таким: 21.09.2021, 10:39:47.
- `toLocaleDateString()` — возвращает строку даты в локальном формате, например, 21.09.2021.
- `toLocaleTimeString()` — возвращает строку времени в локальном формате, например 10:39:47.

В следующем примере мы зададим дату, а затем будем выяснить, в каких еще месяцах этот день выпадает на тот же день недели. В нашем примере это понедельник.

В том случае, если эта дата приходится на понедельник в другом месяце, мы выведем эту дату в консоль.

```
let d = new Date('02/07/2022');
let month = d.getMonth(), weekDay = d.getDay();
console.log(weekDay, d.toLocaleDateString());
for(let i=0; i<12; i++) {
    if(i == month) continue;
    d.setMonth(i);
    if(weekDay == d.getDay())
        console.log(d, d.toLocaleDateString());
}
```

В консоли мы увидим следующее:

Mon Mar 07 2022 00:00:00 GMT+0200 "07.03.2022"

Mon Nov 07 2022 00:00:00 GMT+0200 "07.11.2022"

Что такое ООП?

Объектно-ориентированное программирование подразумевает, что вы создаете некий шаблон объекта, в котором описываете его свойства (некие переменные, или ключи объекта) и методы взаимодействия с этим объектом (функции), как бы моделируя в коде поведение реальных объектов. То есть этот шаблон или фабрика по созданию объектов одного типа работает с кодом примерно так же, как реальная фабрика штампует, например, однотипные фломастеры разных цветов или выпускает намного более сложные по своей конструкции автомобили. При объектно-ориентированном подходе в программировании каждый объект должен представлять собой интуитивно понятную сущность, у которой есть методы и свойства, четко описывающие данный объект.

Когда рассматривают ООП в различных языках программирования, то обычно говорят о создании объектов на основе классов. С 2015 года (стандарт ES6, или EcmaScript2015) классы появились и в JavaScript, хотя и до этого времени в нем можно было создавать объекты, которые наследовали свойства от некого основного объекта-прототипа. И даже с появлением классов этот механизм не слишком-то изменился. То есть в JavaScript ООП основано на прототипах. Теперь необходимо разобраться, что же такое прототип.

Прототип — это объект, который содержит данные о свойствах и методах всех объектов одного типа. Кроме того, все объекты являются наследниками класса `Object`, описанного в ядре JS. С помощью этого класса вы со-

здавали литерал объекта. То есть можно сказать, что прототипом любого объекта в JS является класс [Object](#).

Если сравнивать структуру взаимосвязи объектов, построенную на основе прототипов, с реальной семьей, то можно сказать, что в основе каждого рода (семьи) находятся 2 человека, которые определяют черты (свойства или характеристики) будущих поколений и варианты их поведения или работы. Например, в семье людей с темными волосами и карими глазами будут рождаться дети (создаваться объекты) с такими же характеристиками, хотя цвет волос можно изменить (назначить свойству новое значение). Кроме того, в одной семье могут рождаться дети с музыкальными способностями или дети, которые впоследствии станут ювелирами или сапожниками, или бизнесменами, потому что в их роду такую профессию или сферу деятельности выбирали все. С точки зрения JavaScript сфера деятельности — это уже метод, или функция, причем она наследуется всеми объектами с одним прототипом автоматически, как и музыкальные способности в семье.

Прототипы в JS — это возможность «передать по наследству» всем объектам одного типа некие характеристики, причем таким образом, чтобы каждый из объектов мог воспользоваться этими характеристиками в определенной ситуации.

Пример такой ситуации — это использование методов массива. Например, у нас есть 2 группы студентов, представленные в виде массивов. Нам нужно перевести 2-х студентов из одной группы в другую. Сделаем это с помощью методов массивов `splice()` и `concat()`:

```

let group1 = ['Deniels', 'Jonhson', 'Overton',
             'Stufford', 'Templeton'],
group2 = ['Greenwood', 'Liner', 'Takerman'];
let students = group1.splice(2,2);
console.log(students);
group2 = group2.concat(students);
console.log(group1, group2);

```

Результатом наших действий будут 3 массива в консоли: первый со студентами, «удаленными» методом `splice()` из `group1`, а два других — новые массивы `group1` и `group2`. Интересует нас сейчас больше тот факт, что прототипом каждого из массивов является объект `Array` с набором методов, доступных каждому массиву (т.е. объекту типа `Array`), но при этом для каждого из наших массивов мы использовали только тот метод, который нужен был нам, исходя их условия поставленной задачи.

А это значит, что прототип `Array` может хранить некое количество информации, реализованных в виде функций (на рисунке 2 это обозначено в виде `f concat()`, например), но используемых по мере необходимости.

Вернусь к аналогии с семьей и музыкальными способностями. Кто-то в такой семье может играть на рояле профессионально или для себя, или петь (метод используется), кто-то ни за что не подойдет к инструменту. Однако дети этих людей могут опять-таки использовать свои музыкальные способности или никогда о них не вспоминать, но способности при этом никуда из семьи не денутся.

```

▶ (2) ['Overton', 'Stufford']

▼ (3) ['Deniels', 'Jonhson', 'Templeton'] ⓘ
  0: "Deniels"
  1: "Jonhson"
  2: "Templeton"
  length: 3
  ► [[Prototype]]: Array(0)

▼ (5) ['Greenwood', 'Liner', 'Takerman', 'Overton', 'Stufford'] ⓘ
  0: "Greenwood"
  1: "Liner"
  2: "Takerman"
  3: "Overton"
  4: "Stufford"
  length: 5
  ▼ [[Prototype]]: Array(0)
    ▶ at: f at()
    ▶ concat: f concat()
    ▶ constructor: f Array()
    ▶ copyWithin: f copyWithin()
    ▶ entries: f entries()
    ▶ every: f every()
    ▶ fill: f fill()
    ▶ filter: f filter()
    ▶ find: f find()
    ▶ findIndex: f findIndex()
    ▶ flat: f flat()
    ▶ flatMap: f flatMap()
  
```

Рисунок 2

Так и в JavaScript — методы, реализованные в прототипе, автоматически наследуются, т.е. передаются объекту такого типа, причем вне зависимости от того, будет ли объект эти методы использовать.

Теперь перейдем к вопросу о том, как лучше создавать объекты одного типа (или с одним прототипом) в JavaScript. Так вот для того чтобы создать основу для прототипирования объекта в JavaScript обычно используют функции-конструкторы или классы.

ФУНКЦИИ-КОНСТРУКТОРЫ В JavaScript

На основе классов или функций-конструкторов можно создать экземпляры объектов, с которыми потом программист будет работать в коде.

Необходимость использования функций-конструкторов по сравнению с созданием литералов объектов возникает тогда, когда объектов с одинаковыми свойствами и методами должно быть не один-два, а несколько. Например, нам нужно создать группу студентов, в которой каждый студент — это отдельный объект с точки зрения манипулирования им в JavaScript.

Минимально необходимой информацией для таких объектов-студентов будет имя, фамилия и дата рождения. Кроме того, нам понадобится два метода для каждого из студентов: первый для вывода в консоль информации о нем, второй — для информации о возрасте студента, рассчитываемой на основе текущей даты и даты его рождения с помощью методов объекта [Date](#). Для начала создадим 2-х таких студентов:

```
let diana = {
    firstname: 'Diana',
    lastname: 'Fenton',
    birthday: '07/22/1996',
    showInfo: function() {
        console.log('Student name: ' + this.firstname+
                   ' ' + this.lastname);
    },
}
```

```
showAge: function() {
    const deltaTime = Date.now() -
                      Date.parse(this.birthday);
    const studentAge = Math.floor(deltaTime/
        (365*24*60*60*1000));
    console.log(this.firstname+ ' '+this.lastname+
        ' is '+studentAge +' years old.');
}
}

let luis = {
    firstname: 'Luis',
    lastname: 'Melitano',
    birthday: '02/06/2002',
    showInfo: function(){
        console.log('Student name: '+this.firstname+
            ' '+this.lastname);
    },
    showAge: function(){
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
        const studentAge = Math.floor(deltaTime/
            (365*24*60*60*1000));
        console.log(this.firstname+ ' '+this.lastname+
            ' is '+studentAge +' years old.');
    }
}

console.log(diana);
diana.showInfo();
diana.showAge();
console.log(luis);
luis.showInfo();
luis.showAge();
```

На рисунке 3 видно, что оба объекта у нас имеют в качестве прототипа **Object**, а также 2 одинаковые функции.

```

▼ {firstname: 'Diana', lastname: 'Fenton', birthday: '07/22/1996', show
  Info: f, showAge: f} ⓘ
    birthday: "07/22/1996"
    firstname: "Diana"
    lastname: "Fenton"
  ▶ showAge: f ()
  ▶ showInfo: f ()
  ► [[Prototype]]: Object
Student name: Diana Fenton
Diana Fenton is 25 years old.

▼ {firstname: 'Luis', lastname: 'Melitano', birthday: '02/06/2002', sho
  wInfo: f, showAge: f} ⓘ
    birthday: "02/06/2002"
    firstname: "Luis"
    lastname: "Melitano"
  ▶ showAge: f ()
  ▶ showInfo: f ()
  ► [[Prototype]]: Object
Student name: Luis Melitano
Luis Melitano is 19 years old.

```

Рисунок 3

Если же посмотреть на количество кода, который нам нужно дублировать и изменять только для свойств, то возникает вопрос «А можно ли как-то сократить повторяющиеся действия?». Можно, конечно, попытаться использовать массивы и циклы, но это будет не лучший вариант. Для решения этой задачи в JavaScript существуют функции-конструкторы и классы. Фактически это два способа создать множество однотипных объектов.

Поскольку функции-конструкторы появились в JavaScript значительно раньше, чем классы, давайте сначала создадим такую функцию, которая будет отвечать за объект, который содержит нужные нам данные о студенте: его имя, фамилию и дату рождения, которые мы будем задавать сразу при создании объекта с помощью такой функции, передавая их в качестве параметров. В функции-конструкторе эти данные станут свойствами нашего объекта.

Также добавим методы (функции) `showInfo()` и `showAge()`, которые уже описывали при создании литералов объектов-студентов. То есть мы возьмем синтаксис литерала объекта и несколько его перепишем в соответствии с правилами JavaScript и нашей задачей.

В итоге мы заменим синтаксис, который применяли в объектах-литералах, на аналогичный, но универсальный для одного типа объектов (`Student` в примере ниже). Сначала нам нужно будет определить свойства объекта, перечислив их с помощью ключевого слова `this`, например: `this.firestname = firstname`. Это будет значить, что в свойство `firstname` данного объекта (`this`) мы запишем значение, передаваемое с аргументом `firstname`.

С методами-функциями аналогичная ситуация: мы описываем метод `this.showInfo() = function(){...}`, записывая в ней вывод в консоль данных о объекте-студенте, экземпляра которого создаем в виде переменной `let michael = new Student('Michael', 'Dowson', '11/23/2001');`. Ключевое слово `this` указывает на то, что метод вызывается именно для этого экземпляра объекта `Student`, используя именно его данные об имени и фамилии, а не данные какого-то другого объекта.

Функциям-конструкторам, в отличие от обычных функций, принято давать имена с большой буквы, поэтому мы напишем такой код:

```
function Student(firstname, lastname, birthday) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.showInfo = function() {
        console.log('Student name: '+this.firstname+
                   ' '+this.lastname);
    }
    this.showAge = function() {
        const deltaTime = Date.now() -
                          Date.parse(this.birthday);
        const studentAge = Math.floor(deltaTime/
                                         (365*24*60*60*1000));
        console.log(this.firstname+' '+this.lastname+
                   ' is '+studentAge +' years old.');
    }
}

let michael = new Student('Michael', 'Dowson',
                           '11/23/2001');
michael.showInfo();
michael.showAge();
let lisa = new Student('Lisa', 'Paltrow', '08/12/1998');
lisa.showInfo();
lisa.showAge();
console.log(michael, lisa);
```

Вы, вероятно, задаётесь вопросом, что такое `this` внутри функции-конструктора? Это специальное ключевое слово, которое ссылается на текущий объект, созданный

на основе данной функции-конструктора. Такими объектами в нашем коде являются переменные `michael` и `lisa`, которые создаются аналогично массивам (`let arr = new Array(3,4,78,9);`) с помощью ключевого слова `new`.

Иными словами `this` для `michael` указывает на данные студента '*Michael Dowson*' с датой рождения '*11/23/2001*', а для `lisa` — на студентку '*Lisa Paltrow*', родившуюся '*08/12/1998*'.

Именно поэтому в консоли мы увидим такой текст при вызове методов `showInfo()` и `showAge()`.

Student name: Michael Dowson
Michael Dowson is 19 years old.
Student name: Lisa Paltrow
Lisa Paltrow is 22 years old.

Рисунок 4

Следует также уточнить, что `michael` и `lisa` являются экземплярами объекта `Student` и все свойства, и методы, характерные для этого объекта есть в каждом из полученных экземпляров.

В последней строке кода мы выводим в консоль переменные `michael` и `lisa`, и при раскрытии стрелок слева от каждого объекта `Student` видим, какие они имеют свойства и методы, а также видим, что прототипом для них является `Object`, а функция `Student` указана, как конструктор, откуда и пошло ее название.

Отличием функции-конструктора от обычной функции является то, что она возвращает созданный объект

определенного типа ([Student](#) в нашем примере) без использования ключевого слова [return](#), тогда как обычная функция возвращает либо то, что указано после [return](#), либо [undefined](#), если ключевое слово [return](#) в ней не используется.

Давайте для примера создадим еще одну функцию-конструктор [Hotel](#), которая будет описывать некий отель в некой стране, причем нас будет интересовать помимо названия отеля и его месторасположения еще и количество занятых и незанятых в нем номеров. Количество занятых номеров мы будем передавать в функцию-конструктор в качестве параметра, как и общее количество номеров, а количество доступных номеров и их процент будем получать с помощью методов, т.к. это несложно рассчитать.

Сразу же создадим три переменные-экземпляры [Hotel](#) и вызовем для них указанные методы:

Код примера:

```
function Hotel (name, country, rooms, bookedRooms) {
    this.name = name;
    this.country = country;
    this.rooms = rooms;
    this.bookedRooms = bookedRooms;
    this.availableRooms = function() {
        return this.rooms - this.bookedRooms;
    }
    this.availablePercent = function() {
        return Math.floor(this.availableRooms() /
                           this.rooms *100) + '%';
    }
}
```

```
let antiqueRomanPalace = new Hotel('Antique Roman Palace',
                                    'Turkey', 270, 130),
sharmDreamsClub = new Hotel('Sharm Dreams Club',
                            'Egypt', 320, 212),
miramarenHotel = new Hotel('Miramaren Hotel',
                           'Greece', 70, 63);
console.log(antiqueRomanPalace.availableRooms(),
            antiqueRomanPalace.availablePercent());
console.log(sharmDreamsClub.availableRooms(),
            sharmDreamsClub.availablePercent());
console.log(miramarenHotel.availableRooms(),
            miramarenHotel.availablePercent()); //
```

В консоль будет выведено следующее:

140 '51%	antiqueRomanPalace
108 '33%	sharmDreamsClub
7 '10%	miramarenHotel

Рисунок 5

Итак, мы видим, что на основе переданных параметров объектов типа `Hotel`, которых может быть огромное количество, как и отелей по всему миру, мы получаем расчётные данные из методов.

Обратите внимание, что в методе `availablePercent()` мы используем метод `availableRooms()` для расчета доступных на данный момент номеров, чтобы не дублировать код.

Если информация по отелю обновится, например, количество занятых номеров увеличится, то повторный вызов методов даст нам уже новую информацию. Например, в последнем отеле забронировали еще 4 номера:

```
miramarenHotel.bookedRooms += 4;
console.log(miramarenHotel.availableRooms(),
            miramarenHotel.availablePercent()); /
```

В консоли мы увидим измененные данные в виде 3 '4%'.

Как вы видите, со свойствами объекта, созданного через функцию, конструктор, можно проводить и арифметические действия, поскольку само свойство имеет числовой тип данных.

Давайте вернемся, к примеру со студентами и еще раз рассмотрим, какую информацию мы можем почерпнуть из консоли.

```
▼ Student {firstname: "Michael", lastname: "Dowson", birthday: "11/2
  3/2001", showInfo: f, showAge: f} ⓘ
  birthday: "11/23/2001"
  firstname: "Michael"
  lastname: "Dowson"
  ▶ showAge: f ()
  ▶ showInfo: f ()
  ▶ [[Prototype]]: Object
    ▶ constructor: f Student(firstname, lastname, birthday)
    ▶ [[Prototype]]: Object
▼ Student {firstname: "Lisa", lastname: "Paltrow", birthday: "08/12/1
  998", showInfo: f, showAge: f} ⓘ
  birthday: "08/12/1998"
  firstname: "Lisa"
  lastname: "Paltrow"
  ▶ showAge: f ()
  ▶ showInfo: f ()
  ▶ [[Prototype]]: Object
    ▶ constructor: f Student(firstname, lastname, birthday)
    ▶ [[Prototype]]: Object
```

Рисунок 6

На этом скриншоте можно заметить, что методы `showInfo()` и `showAge()` принадлежат каждому из объектов, тогда как у стандартных объектов типа `Array`, `String`, `Date`, рассмотренных в рамках этого урока, почти все методы находятся в прототипе. Это можно посмотреть в справочной информации на MDN.

<code>Array.prototype.entries()</code>	<code>String.prototype.localeCompare()</code>	<code>Date.prototype.getHours()</code>
<code>Array.prototype.every()</code>	<code>String.prototype.match()</code>	<code>Date.prototype.getMilliseconds()</code>
<code>Array.prototype.fill()</code>	<code>String.prototype.matchAll()</code>	<code>Date.prototype.getMinutes()</code>
<code>Array.prototype.filter()</code>	<code>String.prototype.normalize()</code>	<code>Date.prototype.getMonth()</code>
<code>Array.prototype.find()</code>	<code>String.prototype.padEnd()</code>	<code>Date.prototype.getSeconds()</code>
<code>Array.prototype.findIndex()</code>	<code>String.prototype.padStart()</code>	<code>Date.prototype.getTime()</code>
<code>Array.prototype.flat()</code>	<code>String.raw()</code>	<code>Date.prototype.getTimezoneOffset()</code>
<code>Array.prototype.flatMap()</code>	<code>String.prototype.repeat()</code>	<code>Date.prototype.getUTCDate()</code>
<code>Array.prototype.forEach()</code>	<code>String.prototype.replace()</code>	<code>Date.prototype.getUTCDay()</code>
<code>Array.from()</code>	<code>String.prototype.replaceAll()</code>	<code>Date.prototype.getUTCFullYear()</code>

Рисунок 7

Почему это так? Дело в том, что прототип — это носитель важной информации об объекте, которой вы можете воспользоваться, а можете оставить без внимания. Это все равно, что ген человека: потенциально носитель гена может обладать способностями к музыке или спорту, но, если эти данные не развивать, т.е. не вызывать функции, ответственные за эти умения, то мы никогда не получим из Майкла или Лизы ни музыканта, ни спортсмена.

Поэтому в JavaScript все методы принято записывать для прототипа. Использовать их или нет для каждого объекта зависит от задачи, в которой используются экземпляры объекта. Кроме того, когда методы вынесены

в прототип, каждый объект не «несет их с собой», поэтому количество используемой оперативной памяти для скрипта несколько уменьшается.

Код функции-конструктора сейчас будет несколько видоизменен:

```
function Student(firstname, lastname, birthday) {  
    this.firstname = firstname;  
    this.lastname = lastname;  
    this.birthday = birthday;  
}  
  
Student.prototype.showInfo = function(){  
    console.log('Student name: ' + this.firstname +  
               ' ' + this.lastname);  
}  
  
Student.prototype.showAge = function(){  
    const deltaTime = Date.now() -  
                    Date.parse(this.birthday);  
    const studentAge = Math.floor(deltaTime /  
                                (365*24*60*60*1000));  
    console.log(this.firstname + ' ' + this.lastname +  
               ' is ' + studentAge + ' years old.');//  
}  
  
let michael = new Student('Michael', 'Dowson',  
                         '11/23/2001');  
michael.showInfo();  
michael.showAge();  
let lisa = new Student('Lisa', 'Paltrow', '08/12/1998');  
lisa.showInfo();  
lisa.showAge();  
console.log(michael, lisa);
```

В плане вызова методов для переменных-экземпляров класса `Student` ничего не изменится: они все также будут отображать информацию. Однако в консоли мы увидим, что методы «переехали» в прототип объекта.

```

▼ Student {firstname: "Michael", lastname: "Dowson", birthday: "11/2
3/2001"} ⓘ
  birthday: "11/23/2001"
  firstname: "Michael"
  lastname: "Dowson"
  ▾ [[Prototype]]: Object
    ► showAge: f ()
    ► showInfo: f ()
    ► constructor: f Student(firstname, lastname, birthday)
    ► [[Prototype]]: Object
▼ Student {firstname: "Lisa", lastname: "Paltrow", birthday: "08/12/1
998"} ⓘ
  birthday: "08/12/1998"
  firstname: "Lisa"
  lastname: "Paltrow"
  ▾ [[Prototype]]: Object
    ► showAge: f ()
    ► showInfo: f ()
    ► constructor: f Student(firstname, lastname, birthday)
    ► [[Prototype]]: Object
  
```

Рисунок 8

Для примера с отелями также можно вывести методы в прототип, записав так:

```

function Hotel (name, country, rooms, bookedRooms) {
  this.name = name;
  this.country = country;
  this.rooms = rooms;
  this.bookedRooms = bookedRooms;
}
  
```

```
Hotel.prototype.availableRooms = function() {  
    return this.rooms - this.bookedRooms;  
}  
Hotel.prototype.availablePercent = function() {  
    return Math.floor(this.availableRooms() /  
                      this.rooms *100) + '%';  
}  
  
antiqueRomanPalace.bookedRooms -=12;// было 130 из 270  
console.log(antiqueRomanPalace.availableRooms(),  
            antiqueRomanPalace.availablePercent());
```

На этот раз в консоли мы обнаружим значения 152 '56%'.

Как видно из подсчетов, методы по-прежнему работают корректно при изменении данных для одной из переменных.

Классы в JavaScript

Синтаксис класса в JavaScript подразумевает, что вы используете ключевое слово **class**, имя (название) этого класса с большой буквы и внутри него описываете главную функцию-конструктор, которая так и называется — **constructor**. В этой функции, как правило, вы указываете все свойства класса, а методы описываете, как другие функции внутри класса.

Давайте посмотрим на практике, как мы можем создать класс в JavaScript.

Допустим, нам нужно реализовать в виде класса код, аналогичный тому, который мы создавали в функции-конструкторе **Student**. Для того чтобы не было конфликта имен, назовем наш класс **Human** и запишем в нем конструктор и два метода.

```
class Human {  
    constructor(firstname, lastname, birthday) {  
        this.firstname = firstname;  
        this.lastname = lastname;  
        this.birthday = birthday;  
    }  
  
    showInfo() {  
        console.log(this.firstname + ' ' +  
                    this.lastname);  
    }  
  
    showAge() {  
        const deltaTime = Date.now() -  
                        Date.parse(this.birthday);  
    }  
}
```

```

        const age = Math.floor(deltaTime /
            (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' +
            this.lastname + ' is ' + age +
            ' years old.');
    }
}

const john = new Human('John', 'Smith', '09-17-2003');
john.showInfo();
john.showAge();

```

Мы также создали экземпляр класса `Human` с именем `john` и вызвали для него 2 реализованных в нашем классе метода `showInfo()` и `showAge()`. В результате работы этих методов в консоль будет выведена такая информация (на момент создания урока):

John Smith
John Smith is 17 years old.

Рисунок 9

Как можно увидеть на скриншоте класс работает подобно функции-конструктору. Давайте теперь выведем в консоль информацию о самой переменной `john`:

```
console.log(john);
```

Что мы видим в консоли? В прототипе нашего объекта находится функция-конструктор, описанная в классе, и два метода также сразу же вынесены в прототип.

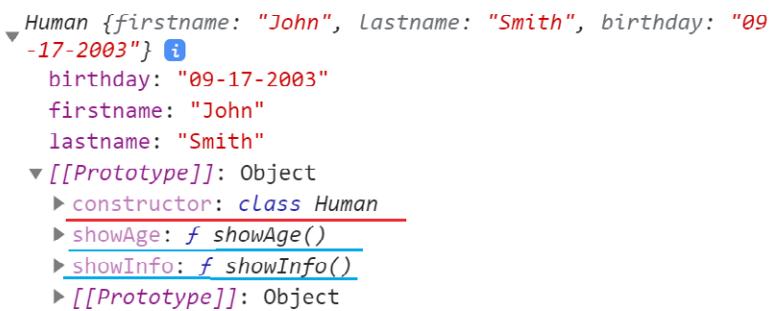


Рисунок 10

Давайте рассмотрим еще один пример создания класса `Rectangle`, который будет решать извечную проблему школьников по нахождению площади и периметра прямоугольника, для которого известны его ширина и высота.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }
    square() {
        return this.width*this.height;
    }
    perimeter() {
        return 2*(this.width+this.height);
    }
}
```

Создадим два экземпляра класса `Rectangle` и выведем в консоль данные, возвращаемые методами `square()` и `perimeter()`.

```

let rect1 = new Rectangle(20, 30);
console.log(rect1.square(), rect1.perimeter());
// 600 100

let rect2 = new Rectangle(78, 92);
console.log(rect2.square(), rect2.perimeter());
// 7176 340

console.log(rect1, rect2);

```

И во втором классе мы снова видим, что методы находятся в прототипе.

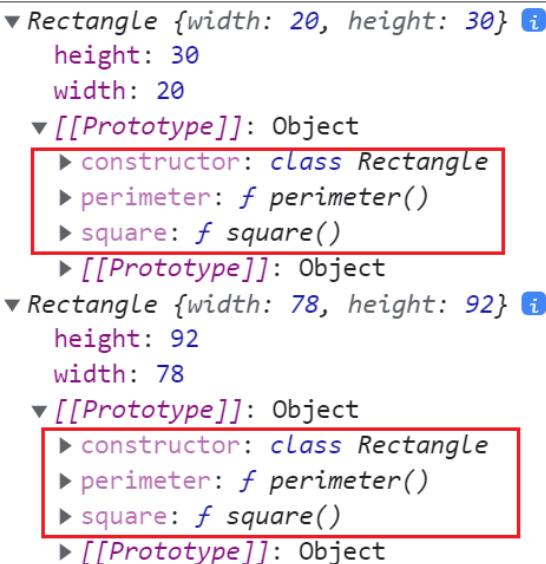


Рисунок 11

Можно уже сделать вывод, что синтаксис класса проще, наглядней и удобней, чем функция-конструктор.

Ключевое слово `this`, используемое внутри класса является как бы внутренним «я» объекта. Как любой чело-

век при запросе «Ваше имя» внутри понимает, что ему нужно ввести свое имя, а не имя соседа, например, так и объект при обращении `this.firstname` понимает, что ему нужно взять данные, записанные при создании объекта-экземпляра класса в соответствующее поле. Для разных экземпляров класса, как и для разных людей, это значение будет отличаться, но при этом каждый объект имеет вполне конкретную запись этого свойства и не сможет использовать никакую другую аналогично тому, что в свидетельстве о рождении указано то имя, которое дали родители ребенку после его появления на свет, а не произвольное и часто изменяемое.

Что будет, если мы упустим ключевое слово `this` в синтаксисе класса, например, так:

```
class Rectangle {  
    constructor(width, height){  
        width = width;  
        height = height;  
    }  
    square() {  
        return width*height;  
    }  
    perimeter() {  
        return 2*(width+height);  
    }  
}
```

В этом случае мы увидим ошибку в консоли *Uncaught ReferenceError: width is not defined at Rectangle.square*, которая возникнет внутри функции `square` при попытке использовать необъявленную переменную `width`.

Если мы изменим код и будем передавать в функции `square()` и `perimeter()` параметры `width` и `height`, то все равно получим `NaN` в качестве результата вызова этих методов, т.к. внутри класса эти параметры имеют значение `undefined`.

```
class Rectangle {  
    constructor(width, height){  
        width = width;  
        height = height;  
    }  
    square(width,height) {  
        console.log(width, height);  
        return width*height;  
    }  
    perimeter(width,height){  
        return 2*(width+height);  
    }  
}  
let rect1 = new Rectangle(20, 30);  
console.log(rect1.square(), rect1.perimeter());  
                                // NaN NaN  
let rect2 = new Rectangle(78, 92);  
console.log(rect2.square(), rect2.perimeter());  
                                // NaN NaN
```

Фундаментальные принципы ООП

Во всех языках программирования вы столкнетесь с тремя фундаментальными принципами ООП — это наследование, инкапсуляция и полиморфизм. Давайте разберем подробнее, что это значит, и как реализовано в JavaScript, тем более, что мы уже говорили о прототипах, как механизме наследования методов объекта в JS.

Инкапсуляция

Принцип инкапсуляции подразумевает, что мы скрываем детали реализации класса от доступа извне, делая переменные и функции недоступными из экземпляров класса. Напомню, что экземпляры класса — это переменные (константы), созданные на основе функции-конструктора или класса.

Все свойства и методы, которые мы создавали до сих пор, принято называть **публичными**. Поэтому мы можем изменить их в любой момент. Напомню, что мы создавали переменную `john` как экземпляр класса `Human`. Выведем в консоль те данные, которые задали при инициализации переменной, а потом поменяем их:

```
const john = new Human('John', 'Smith', '09-17-2003');
console.log(john.firstname, john.lastname);
john.firstname = 'Billy';
john.lastname = 'Thomas';
console.log(john.firstname, john.lastname);
```

Результат в консоли:

John Smith
Billy Thomas

Рисунок 12

Однако бывает необходимость «спрятать» от изменения какую-либо переменную или метод. Для этого существуют приватные переменные. Такие переменные принято объявлять вне конструктора с ведущим знаком `#`, а инициализировать в конструкторе.

Примечание: на момент написания урока эта возможность реализована только в браузере Google Chrome. Во всех остальных вы можете столкнуться с ошибкой.

Например, в классе `Human` нам необходима переменная `#id`, указывающая идентификатор пользователя в виде большого случайного числа. Мы получим ее значение с помощью методов объекта `Math`.

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
    ...
}
console.log(john);
```

Если вывести в консоль данные об объекте `john`, то мы увидим эту переменную `#id`:

```
▼ Human {firstname: "Billy", lastname: "Thomas", birthday: "09-17-2003",
  #id: 8034504} ⓘ
  birthday: "09-17-2003"
  firstname: "Billy"
  lastname: "Thomas"
  #id: 8034504
▶ [[Prototype]]: Object
```

Рисунок 13

Однако при попытке явного доступа к этой переменной или ее изменения, в консоль будет выведена ошибка: *Uncaught SyntaxError: Private field '#id' must be declared in an enclosing class*

```
console.log(john.#id);
john.#id = 1;
```

Это как раз показывает, что данная переменная является приватной, то есть недоступной извне, но доступной только в пределах класса, в котором она объявлена. Однако способ для доступа к ней все же имеется, и сделать это можно с помощью специальных методов-аксессоров, которые также называются геттерами (от англ. *get*) и сеттерами (от англ. *set*).

Геттер подразумевает, что мы можем получить значение некой переменной, а сеттер — что мы можем установить значение этой переменной. Синтаксис у них такой:

```
get id() {
  return this.#id;
}
```

```
set id(value) {
    this.#id = value;
}
```

Из записи видно, что ключевые слова `get` и `set` мы записываем через пробел от имени того свойства, которое мы хотим сделать публичным для нашего класса, т.е. доступным извне, но при этом менять значение мы будем для внутренней приватной (скрытой, или инкапсулированной) переменной.

Теперь давайте запишем класс `Human` полностью и вызовем методы-аксессоры для переменной `john`:

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
    showInfo() {
        console.log(this.firstname + ' ' +
    }
    showAge() {
        const deltaTime = Date.now() -
                        Date.parse(this.birthday);
        const age = Math.floor(deltaTime /
                               (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' +
                    this.lastname + ' is ' +
                    age + ' years old.');
    }
}
```

```

    get id() {
        return this.#id;
    }
    set id(value) {
        this.#id = value;
    }
}

console.log(john.id);
john.id = 1;
console.log(john.id);

```

Обратите внимание, что мы не вызываем ни метод `get id()`, ни метод `set id()`. Вместо этого мы получаем или устанавливаем путем присваивания значения `1` свойство `id`. Тем не менее в консоли мы увидим:

The screenshot shows the state of a `Human` object in a browser's developer tools. The object has properties: `firstname: "Billy"`, `lastname: "Thomas"`, `birthday: "09-17-2003"`, and `#id: 4767970`. A tooltip for the `#id` property says "Invoke property getter". When the `#id` property is changed to `1`, the object is updated, and the new state is shown below, where the `#id` value is now `1`.

```

▼ Human {firstname: "Billy", lastname: "Thomas", birthday: "09-17-2003",
#id: 4767970} ⓘ
  birthday: "09-17-2003"
  firstname: "Billy"
  lastname: "Thomas"
  #id: 4767970
  id: (...)

► [[Prototype]]: Object
4767970
  Invoke property getter

1

▼ Human {firstname: "Billy", lastname: "Thomas", birthday: "09-17-2003",
#id: 1} ⓘ
  birthday: "09-17-2003"
  firstname: "Billy"
  lastname: "Thomas"
  #id: 1
  id: (...)

► [[Prototype]]: Object

```

Рисунок 14

На рисунке 14 разным цветом подчеркнуты значения приватной переменной `#id` до и после присваивания значения `1`, а также геттер для свойства `id`.

Наследование

Наследование предполагает, что на основе одного класса можно создать другой, и во втором, дочернем классе, можно использовать методы первого, родительского класса. Это очень удобно, так как не нужно создавать новые методы, дублирующие уже созданные.

В примере с классом `Human` мы можем использовать принцип наследования следующим образом: мы создадим новый класс `Teacher`, в котором используем те же данные для инициализации объекта: имя, фамилию, дату рождения, но еще добавим массив предметов, который может читать этот учитель. Также добавим еще один метод `showSubjects()`, который будет выводить информацию о знаниях учителя.

```
class Teacher extends Human{
    constructor(firstname, lastname, birthday,
               subjects = []) {
        super(firstname, lastname, birthday);
        this.subjects = subjects;
    }

    showSubjects() {
        console.log(this.firstname + ' ' +
                    this.lastname + ' can teach you ' +
                    this.subjects.join(', '));
    }
}
```

```
const kate = new Teacher('Kate', 'Lowdell',
    '07/15/1986',
    ['biology', 'geography']);
```

Обратите внимание на синтаксис: объявляя класс `Teacher` мы записали ключевое слово `extends` (расширяет) и указали имя класса для расширения — `Human`. Таким образом мы показываем, что класс `Teacher` является наследником класса `Human`. Кроме того, в конструктор мы передаем не 3 параметра, а 4, но первые 3 из них используем в ключевом слове `super()` для вызова конструктора родительского класса `Human`.

Теперь самое интересное. При создании экземпляра класса `Teacher` с именем `kate` мы получаем доступ не только к описанному в этом классе методу `showSubjects()`, а и ко всем методам родительского класса `Human`. Это видно на рисунке 15 при добавлении оператора точка к имени экземпляра в текстовом редакторе в виде подсказок.

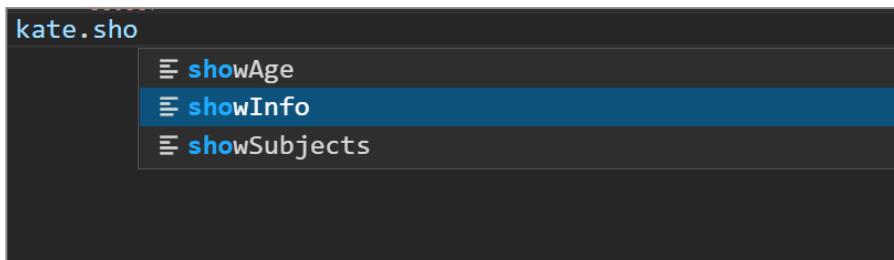


Рисунок 15

То есть наш экземпляр класса унаследовал эти методы от другого класса, так как они взаимосвязаны за счет ключевого слова `extends`.

Если мы запишем вызов всех этих методов для объекта с именем `katе`,

```
katе.showInfo();
katе.showAge();
katе.showSubjects();
```

то получим следующие записи в консоли:

Kate Lowdell
Kate Lowdell is 35 years old.
Kate Lowdell can teach you biology, geography

Рисунок 16

Давайте теперь усложним задачу и создадим еще один класс `ITMentor`, который будет расширять теперь уже класс `Teacher` по тому же принципу, используя ключевое слово `extends`:

```
class ITMentor extends Teacher{
    constructor(firstname, lastname, birthday,
               subjects = [], level){
        super(firstname, lastname, birthday, subjects);
        this.level = level;
    }

    showLevel(){
        console.log(this.firstname + ' ' +
                   this.lastname + ' has level ' +
                   this.level);
    }
}
```

```
const andrew = new ITMentor("Andrew", "Phillipov",
                            '07/22/1986', ['HTML',
                            'CSS', 'JavaScript',
                            'React', 'Angular'],
                            'Senior');

andrew.showInfo();
andrew.showAge();
andrew.showSubjects();
andrew.showLevel();
```

В этом классе у нас добавилось еще одно свойство `level`, отвечающее за уровень знаний `ITMentor` и метод `showLevel()`, выводящий в консоль информацию о нем. При вызове метода `super()` в конструкторе этого класса мы уже обращаемся к классу `Teacher`, передавая в него требуемые параметры.

При создании переменной-экземпляра класса `ITMentor` с именем `andrew` мы автоматически получаем доступ не только к свойствам и методам этого класса, но и к свойствам и методам классов `Teacher` и `Human`. Если раскрыть все стрелочки, которые появляются при выводе в консоль переменной `andrew`, то мы увидим всю цепочку прототипов, которая показывает нам, какому классу принадлежат какие методы.

С наследованием работает такой подход — когда мы вызываем какой-либо метод или используем некое свойство, то сначала движок JavaScript будет искать его в основном классе объекта, потом поднимется к родительскому классу, затем выше по цепочке родительских объектов вплоть до класса `Object`, пока не найдет соответствующий метод или свойство. Это очень удобно, хотя и предпола-

гает, что вы должны продумать, в каком из объектов они должны быть описаны.

```

▼ ITMentor {firstname: "Andrew", lastname: "Phillipov", birthday: "07/22/1986", subjects: Array(5),
  level: "Senior"} ⓘ
  birthday: "07/22/1986"
  firstname: "Andrew"
  lastname: "Phillipov"
  level: "Senior"
  ▶ subjects: (5) ["HTML", "CSS", "JavaScript", "React", "Angular"]
1 ▶ [[Prototype]]: Teacher
  ▶ constructor: class ITMentor
  ▶ showLevel(): f showLevel()
2 ▶ [[Prototype]]: Human
  ▶ constructor: class Teacher
  ▶ showSubjects(): f showSubjects()
3 ▶ [[Prototype]]: Object
  ▶ constructor: class Human
  ▶ showAge(): f showAge()
  ▶ showInfo(): f showInfo()
4 ▶ [[Prototype]]: Object
  ▶ 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__()

```

Рисунок 17

Полиморфизм

Полиморфизм можно перевести, как множество форм. Этот принцип помогает проектировать объекты таким образом, чтобы они могли совместно использовать или переопределять любое поведение. Чаще всего для переопределения используются методы родительского класса. Отсюда можно сделать вывод, что *полиморфизм* использует *наследование*.

Смысл полиморфизма в ООП заключается в том, что вы можете вызвать один и тот же метод для разных объ-

ектов, но при этом для каждого объекта этот метод сработает по-своему. Давайте рассмотрим это на примере метода `showSubjects()`, который появился в классе `Teacher`, но изменим мы его в классе `ITMentor`.

В коде ниже приведен весь класс `ITMentor` с измененным методом `showSubjects()` и продублировано создание константы `andrew`, а также заново вызван метод `showSubjects()`, унаследованный от класса `Teacher`.

```
class ITMentor extends Teacher {  
    constructor(firstname, lastname, birthday,  
               subjects = [], level){  
        super(firstname, lastname, birthday, subjects);  
        this.level = level;  
    }  
  
    showSubjects() {  
        console.log('With '+this.firstname + ' ' +  
                   this.lastname +  
                   ' you can get such IT skills: '+  
                   this.subjects.join(', '));  
        document.write('<p>With '+this.firstname +  
                      ' ' + this.lastname +  
                      ' you can get such IT skills:  
                      </p><ol><li>' +  
                      this.subjects.join('<li>') +  
                      '</ol>');  
    }  
  
    showLevel(){  
        console.log(this.firstname + ' ' +  
                   this.lastname + ' has level ' +  
                   this.level);  
    }  
}
```

```
const andrew = new ITMentor("Andrew", "Phillipov",
    '07/22/1986', ['HTML', 'CSS',
    'JavaScript', 'React', 'Angular'],
    'Senior');
andrew.showSubjects();
```

Однако теперь вывод информации с помощью `showSubjects()` изменился — мы видим в консоли другой текст + текст с разметкой у нас выведен в тело документа (`<body>`) в виде абзаца и нумерованного списка.

console.log()

With Andrew Phillipov you can get such IT skills: HTML, CSS, JavaScript, React, Angular

document.write()

With Andrew Phillipov you can get such IT skills:

1. HTML
2. CSS
3. JavaScript
4. React
5. Angular

Рисунок 18

Это произошло за счет того, что мы добавили в `showSubjects()` метод `document.write()`, который позволяет вывести в тело html-страницы, с которой связан наш скрипт, любой текст, оформленный в теги по правилам html-разметки.

Если же мы вызовем метод для ранее объявленной константы `kate` — экземпляра класса `Teacher`, то увидим

текст, формируемый методом `showSubjects()` для класса `Teacher` без каких-либо изменений.

Kate Lowdell can teach you biology, geography

Рисунок 19

Это говорит о том, что JavaScript в первую очередь ищет свойство или метод в том классе, экземпляром которого является созданная переменная, а уже потом обращается по цепочке к каждому из родительских классов, чтобы найти вызываемый для объекта метод, если он не найден для того класса, к которому принадлежит этот объект.

Давайте для демонстрации этого приема изменим в классе `Human` метод, который наследуется от главного в JavaScript класса объектов `Object` и называется `toString()`. Этот метод отвечает за то, как будет выведен в строковом виде экземпляр любого объекта. Для начала посмотрим, как отобразятся все наши константы разных классов в теле документа, если поместить их в метод `document.write()`:

```
document.write('Class Human: '+john + '<br>');
document.write('Class Teacher: '+kate + '<br>');
document.write('Class ITMentor: '+andrew + '<br>');
```

Результат на экране:

*Class Human: [object Object]
 Class Teacher: [object Object]
 Class ITMentor: [object Object]*

Не слишком информативно, не так ли? Поэтому добавляем в класс `Human` метод `toString()`, в котором выведем

имя и фамилию из свойств каждого объекта, а также название класса с помощью свойства `this.constructor.name`:

```
class Human {
    #id
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = birthday;
        this.#id = Math.floor(Math.random()*10e6);
    }
    showInfo() {
        console.log(this.firstname + ' ' +
                    this.lastname);
    }
    showAge() {
        const deltaTime = Date.now() -
                        Date.parse(this.birthday);
        const age = Math.floor(deltaTime /
                               (365 * 24 * 60 * 60 * 1000));
        console.log(this.firstname + ' ' + this.lastname +
                    ' is ' + age + ' years old.');
    }
    get id(){
        return this.#id;
    }
    set id(value){
        this.#id = value;
    }
    toString(){
        return this.firstname + ' ' + this.lastname +
               ' is a '+ this.constructor.name;
    }
}
```

Теперь вызов метода `document.write()` даст совсем другой результат:

Class Human: Billy Thomas is a Human

Class Teacher: Kate Lowdell is a Teacher

Class ITMentor: Andrew Phillipov is a ITMentor

Обратите внимание на то, что метод, добавленный в родительский класс **Human** сработал для всех дочерних классов вполне корректно.

Оператор instanceof

Для проверки того, к какому классу принадлежит объект, созданный с помощью переменной, используется оператор **instanceof**. Его синтаксис:

```
variableName instanceof Class
```

Этот оператор возвращает **true** или **false** в зависимости от того, принадлежит ли наш объект к указанному классу. В коде ниже видно, что экземпляры класса **Teacher** и **ITMentor** также являются экземплярами родительского класса **Human**, тогда как экземпляр класса **Human** не является экземпляром **Teacher** или **ITMentor**, как и экземпляром никак не связанного с ним класса **Array**.

```
console.log(john instanceof Human); //true
console.log(kate instanceof Teacher); //true
console.log(kate instanceof Human); //true
console.log(andrew instanceof ITMentor); //true
console.log(andrew instanceof Teacher); //true
console.log(andrew instanceof Human); //true
console.log(john instanceof Teacher); //false
console.log(john instanceof ITMentor); //false
console.log(john instanceof Array); //false
```

Расширение стандартных классов

В заключение темы о принципах ООП хотелось бы остановиться на наследовании стандартных, или встроенных, классов, описанных в ядре JavaScript, на примере класса `String`. Мы создадим класс `StringInfo` и добавим в него единственный метод, который будет подсчитывать нам количество вхождений какой-либо буквы или слова в некую строку. В этом случае мы можем обойтись без вызова метода `constructor`:

```
class StringInfo extends String {  
    calcLetter(letter) {  
        let count = 0;  
        let index = this.indexOf(letter);  
        while(index != -1){  
            count++;  
            index = this.indexOf(letter, index+1);  
        }  
        return count;  
    }  
  
    let myStr = new StringInfo("When the going gets  
                                tough, the tough get going.");  
    console.log('g in "'+myStr+'" = '+  
              myStr.calcLetter('g'));  
    console.log('going in "'+myStr+'" = '+  
              myStr.calcLetter('going'));  
    console.log('"text" in "'+myStr+'" = '+  
              myStr.calcLetter('text'));
```

В консоли мы увидим следующую информацию:

"g" in *"When the going gets tough, the tough get going."* = 8
"going" in *"When the going gets tough, the tough get going."* = 2
"text" in *"When the going gets tough, the tough get going."* = 0

В коде метода в несколько измененном виде мы использовали тот скрипт, который вы уже встречали в тексте урока.

Ключевое слово **this** в этом классе указывает на ту строку, которая была передана в скобках при создании экземпляра.

Имейте в виду, что не ко всем строкам можно применить метод **calcLetter()**. Если мы объявим обычную строковую переменную, которая будет экземпляром объекта **String**, а не **StringInfo()**, то вызов метода **calcLetter()** приведет к ошибке:

```
let strHow = 'How do you do?';
console.log('"do" in "'+ strHow +'" = '+
            strHow.calcLetter('do'));
```

Ошибка будет в виде сообщения, что **strHow.calcLetter** не является функцией, т.к. в объекте **String**, к которому принадлежит переменная **strHow**, такой метод не предусмотрен: *Uncaught TypeError: strHow.calcLetter is not a function.*

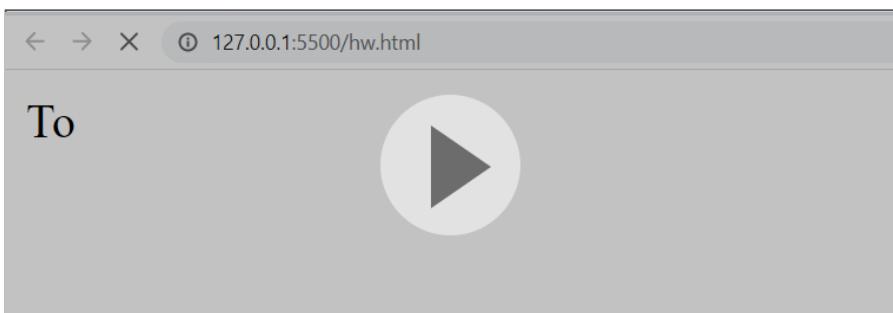
Поэтому нужно различать экземпляры объектов на основе созданных вами классов и экземпляры родительских классов, чтобы не было путаницы. Классы, которые расширяют стандартные или пользовательские классы, наследуют свойства и методы своих родительских классов.

сов, но классы-родители не имеют свойств и методов до-
черних классов аналогично тому, что ваша бабушка или
прабабушка вряд ли смогла управлять современным ком-
пьютером или смартфоном, т.к. в ее время эти девайсы
не были еще созданы.

Домашнее задание

Задание 1

С помощью методов `setTimeout()` или `setInterval()` выведите в тело документа методом `document.write()` первую строку из монолога Гамлете «To be, or not to be, that is the question...» так, чтобы буквы появлялись по одной через 200-300 миллисекунд, а затем с новой строки также методом `document.write()` нужно вывести «William Shakespeare, from "Hamlet"» (видео 1). Если видео не работает, посмотреть, как это выглядит, можно в файле `hamlet.gif` (файл прикреплен к данному pdf-файлу).



Video 1

Задание 2

Рассчитайте, сколько дней, часов, минут и секунд осталось до Нового года. Выведите эти значения красиво, используя метод `document.write()` с тегами `<p>` и `` и классами для них. Стили можно записать в отдельном css-файле. Если одно из чисел будет меньше 10, то его нужно вывести с ведущим 9, например, так:

52	10	09	04
days	hours	minutes	seconds

Рисунок 20

Сделайте свой скрипт универсальным, задав дату следующего Нового года относительно текущей даты с помощью методов объекта `Date`.

Задание 3

Создайте объект `list`, задайте для него: свойство `values`, содержащее массив похожих значений, например, каких-либо продуктов

Метод `printList()`, который сортирует все элементы массива `values` в алфавитном порядке и выводит их в виде нумерованного списка в тело документа методом `document.write()`, метод `add(product)`, который добавляет к `values` еще один элемент.

Выведите сначала массив начальных значений объекта `list` с помощью его метода `printList()`. Например, это будет список продуктов:

1. apple
2. ice cream
3. kiwi
4. potato
5. sour cream
6. tomato

Рисунок 21

Затем добавьте еще один какой-нибудь элемент с помощью метода `add()` и снова выведите все значений объекта `list` методом `printList()`.

1. apple
2. ice cream
3. kiwi
4. potato
5. pumpkin
6. sour cream
7. tomato

Рисунок 22

Затем замените все значения в свойстве `list.values` на другой массив и снова выведите его методом `printList()`. Например, так:

1. C#
2. HTML
3. JavaScript
4. PHP

Рисунок 23

Задание 4

Создайте класс `MyButton`, который принимает 2 параметра в виде текста (`text`) и класса кнопки (`btnClass`). Опишите в нем метод `show()`, который выводит методом `document.write()` экземпляр кнопки в тело html-страницы. Предусмотрите геттер и сеттер, которые позволяют получить и изменить свойство `value` кнопки, которое на самом деле изменяет ее свойство `text`.

Опишите в стилях для страницы несколько классов, которые позволяют создать разные экземпляры кнопок.

Выведите несколько кнопок методом `show()`, для одной из них поменяйте текст:



Рисунок 24

Создайте класс `ColorButton`, который наследует класс `MyButton`, добавив в него дополнительный класс, который позволяет менять цвет фона и текста кнопки, добавляя к экземпляру `ColorButton` помимо основного дополнительный класс. Например, экземпляр `ColorButton` будет вызываться с такими параметрами:

```
let btnColor = new ColorButton("See more", "btn",
                               "btn-danger");
```

Кнопка, выведенная с помощью метода `show()` будет иметь такой код:

```
<button type="button" class="btn btn-danger">
    See more
</button>
```

Домашнее задание