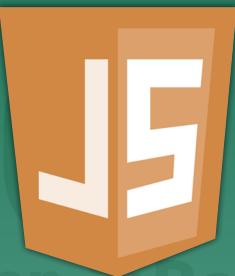


# Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery



# Unit 5

## Детальное о функциях

## Contents

<b>Объект arguments .....</b>	<b>3</b>
Цель и задачи объекта.....	3
Свойство length.....	5
Особенности функций в JavaScript .....	10
<b>Область видимости переменной .....</b>	<b>14</b>
Поднятие объявлений.....	24
Различия деклараций var, let и const .....	28
<b>Рекурсия.....</b>	<b>31</b>
Задание для самостоятельной работы .....	38

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

# Объект arguments

Одним из главных преимуществ функций является возможность многократного использования одного и того же кода для различных значений (аргументов), а также объединение этого кода под логичным и понятным именем. В ряде практических задач возникает необходимость работы не только с различными значениями аргументов, но и с разным их количеством.

Например, мы хотим написать функцию «`max`», находящую максимальный элемент среди переданных аргументов. Хотелось бы, чтобы функция могла работать с произвольным набором данных, то есть, чтобы в одном месте программы можно было записать «`max(x1, x2)`», а в другом — «`max(x1, x2, x3, x4)`». В JavaScript такая возможность уже реализована, и любая функция может принять произвольное количество аргументов.

## Цель и задачи объекта

При вызове функции все переданные в нее аргументы попадают в специальный объект «`arguments`», доступный для использования в теле функции. Рассмотрим его формирование на следующем примере. Создадим функцию «`logArguments`», задача которой будет вывод в консоль значения принятого в теле функции объекта «`arguments`». Введите в консоль браузера определение функции и нажмите «`Enter`»:

```
function logArguments () {  
    console.log(arguments);  
}
```

Затем несколько раз вызовем эту функцию, указав различное количество аргументов. Вводим в консоль, последовательно нажимая «Enter»

```
logArguments(1,2,3)
logArguments("text")
```

Во-первых, убеждаемся в том, что вызывается одна и та же функция, независимо от количества аргументов и их типа данных. Обратим внимание, что при объявлении функции мы вообще не указывали ожидаемых параметров.

Во-вторых, проанализируем состав данных, отображенных в консоли. В появившихся результатах нажимаем на треугольный символ слева от объекта, раскрывая дополнительные детали (см. рис. 1).

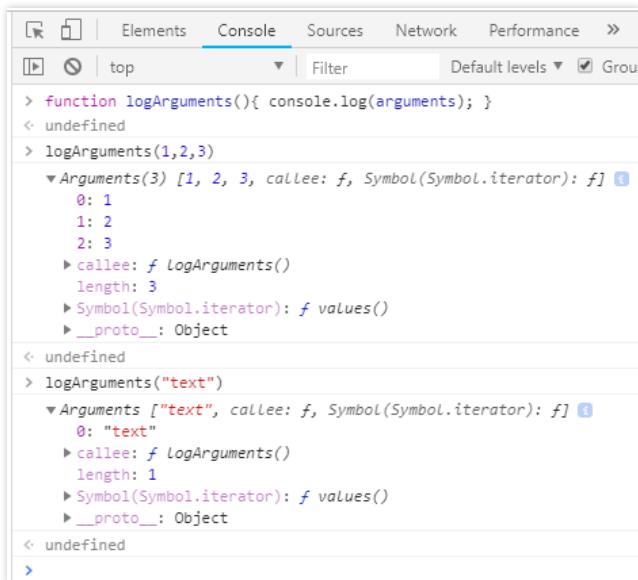


Рисунок 1

Как видно, в первом случае в объекте «`arguments`» наблюдаются три переданных числа `1`, `2` и `3`, идущие под индексами `0`, `1` и `2` соответственно. Во втором случае в объекте присутствует строка «`text`» с индексом `0`. По указанным индексам в функции можно получить данные обо всех переданных аргументах, независимо от их количества. Например, выражение «`arguments[0]`» будет отвечать за первый аргумент, переданный в функцию: число `1` в первом вызове или строку «`text`» — во втором. Выражение «`arguments[1]`» обеспечит доступ ко второму аргументу и так далее.

### Свойство `length`

Кроме значений переданных аргументов в объекте «`arguments`» присутствуют и другие данные (см. рис. 42). Для практического использования интерес представляет свойство «`length`», отвечающее за количество переданных в функцию аргументов.

Свойство «`length`» устанавливается каждый раз с новым вызовом функции и позволяет последовательно перебрать значения аргументов, например, при помощи цикла-счетчика «`for`».

Модифицируем определение функции «`logArguments`» для вывода переданных в функцию аргументов, только без отображения остальных полей объекта «`arguments`».

```
function logArguments() {  
    for(i=0; i<arguments.length; i++)  
        console.log("argument"+(i+1)+" =  
                    "+arguments[i])  
}
```

В теле функции реализуем цикл-счетчик по индексам аргументов от «0» до граничного значения «`arguments.length`». Для вывода в консоль формируем строку из надписи «`argument`», к которой добавляется номер аргумента «`i+1`» (добавляем 1 для того чтобы нумерация в консоли началась со значения 1, а не 0), затем к строке дописывается знак равенства «`+=`» и само значение аргумента с индексом «`i`».

Введите в консоль или скопируйте новое определение функции, после чего нажмите «`Enter`». Убедитесь в том, что переопределение функции проходит успешно и не вызывает ошибок. Проверим, заменилась ли функция на новую. Повторно вводим команды вызова функции, последовательно нажимая «`Enter`»

```
logArguments(1,2,3)
logArguments("text")
```

```
> function logArguments(){
    for(i=0; i<arguments.length; i++)
        console.log("argument"+(i+1)+" = "+arguments[i])
}
< undefined
> logArguments(1,2,3)
argument1 = 1
argument2 = 2
argument3 = 3
< undefined
> logArguments("text")
argument1 = text
< undefined
>
```

Рисунок 2

Наблюдаем новые результаты вызова функции, что свидетельствует о выполнении нового тела, то есть о замене старого определения функции на новое (рис. 2).

Подобным образом в любой функции можно узнать как об общем количестве переданных аргументов, так и об их значениях. Причем, собранные в одном объекте значения аргументов дают возможность использовать один общий цикл вместо отдельных имен для каждого параметра.

Следует отметить, что объект «`arguments`» существует параллельно с формальными параметрами функции. То есть можно использовать как имена параметров, так и свойства объекта «`arguments`», в зависимости от удобства или типа задачи. Для иллюстрации двух возможностей доступа к аргументам, заменим определение функции, добавив формальный параметр «`x`» в определении функции и вывод его значения в ее теле «`console.log("x = "+x);`»:

```
function logArguments(x) {  
    console.log("x = "+x);  
    for(i=0; i<arguments.length; i++)  
        console.log("argument"+(i+1)+" = "+arguments[i])  
}
```

Повторим запросы на вызов функции и убедимся в возможности одновременного доступа к первому аргументу как по имени «`x`», так и через свойство «`arguments[0]`» (см. рис. 3).

Последний вызов функции совершается без передачи аргументов. В таком случае цикл не выполняется ни разу, и вывода нумерованных аргументов нет. Формальный же

параметр «`x`» приобретает значение «`undefined`». То есть перед его использованием в выражениях всё же желательно предусмотреть проверку на пустоту.

The screenshot shows the Google Chrome DevTools interface with the 'Console' tab selected. The console output displays the execution of a JavaScript function named `logArguments`. The function logs the value of `x` and then iterates through the arguments array, logging each argument's index and value. The output shows various calls to `logArguments` with different arguments, including numbers and strings, and demonstrates how `x` is `undefined` when no argument is provided.

```

function logArguments(x){
    console.log("x = "+x);
    for(i=0; i<arguments.length; i++)
        console.log("argument"+(i+1)+" = "+arguments[i])
}
< undefined
> logArguments(1,2,3)
x = 1
argument1 = 1
argument2 = 2
argument3 = 3
< undefined
> logArguments("text")
x = text
argument1 = text
< undefined
> logArguments()
x = undefined
< undefined
>

```

Рисунок 3

В то же время обработка аргументов в цикле таких проверок не требует, т.к. используемое свойство «`arguments.length`» является неким предохранителем и при нулевом значении просто не запускает цикл. Выбор способа работы с параметрами или аргументами дает программисту дополнительную свободу.

В качестве дополнительного примера реализуем функцию, определяющую максимальное значение из переданных аргументов

## Объект arguments

```
function max() {
    if(arguments.length == 0) return undefined;
    ret = arguments[0];
    for(i=1;i<arguments.length;i++)
        if(arguments[i]>ret)
            ret = arguments[i];
    return ret;
}
```

В первой строке проверяем, есть ли у нас вообще аргументы. Если размер объекта равен нулю (`arguments.length == 0`) возвращаем значение «`undefined`». Обратите внимание, что команда «`return`» завершит работу функции, если условие выполнится. Это позволяет нам не писать «`else`» перед последующими командами.

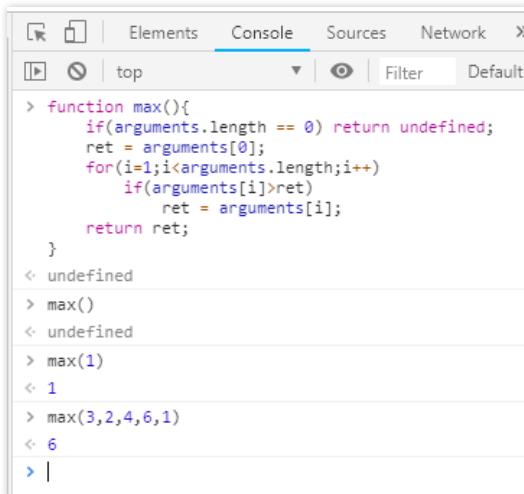


Рисунок 4

Далее применяет стандартный алгоритм поиска максимума: в переменную «`ret`» записываем значение первого

аргумента. Затем циклически сравниваем это значение с каждым из аргументов и, если значение аргумента окажется большим, чем хранимое, то заменяем это хранимое значение на новое. По завершению цикла возвращаем полученный в переменной «`ret`» результат.

Введите или скопируйте определение функции в консоль браузера. Попробуйте ее вызвать с различными значениями аргумента (рис. 4).

**Найдите ошибку.** Функция должна определять среднее арифметическое от переданных аргументов (их сумму, деленную на их количество). Правильным ли является следующее определение функции?

```
function mean() {
    sum = 0;
    for(i=0;i<arguments.length;i++)
        sum += arguments[i];
    return sum / arguments.length;
}
```

(В алгоритме работы ошибок нет, но в функции не анализируется случай ее вызова без аргументов. В таком случае значение `arguments.length` равно нулю и в последней строке тела функции происходит деление на ноль. В начале функции следует добавить проверку, аналогичную предыдущему примеру: «`if(arguments.length == 0) return undefined;`»).

## Особенности функций в JavaScript

С технической точки зрения, задача по реализации функций, в которые может передаваться произвольное

количество аргументов, является достаточно сложной. Далеко не во всех языках программирования предусмотрена возможность вызывать одну и ту же функцию, используя при вызове различное количество параметров.

Обычно, в таких языках функции с различным количеством параметров считаются разными функциями и должны быть созданы отдельно каждая из них для своего набора аргументов. При этом они вполне легально могут иметь одно и то же имя — отличаться они как раз и будут количеством принимаемых параметров. Такая ситуация называется «параметрическим полиморфизмом» и характерна для языков типа C++, C#, Java и т.п.

В JavaScript применяется объект «`arguments`», и любая функция может принять произвольное количество аргументов, независимо от того, сколько их было указано при объявлении функции. Это значительно упрощает разработку и сокращает количество кода, поскольку не требует переопределения разных функций для разных наборов данных.

Обратная сторона этой возможности заключается в том, что в JavaScript все функции, имеющие одинаковые имена, даже при разном наборе параметров, являются по сути одним и тем же объектом. Попытка объявить функцию с именем уже существующей функции, но с указанием другого числа параметров приведет к переопределению функции — существующая функция будет полностью заменена на новую.

Самым опасным при этом для программиста является то, что переопределение функции не приводит к ошибкам выполнения программы. Объясняется это

тем, что в объектно-ориентированном языке JavaScript функции являются разновидностью объектов и мало чем отличаются от переменных. Так же, как одной переменной можно присвоить новое значение и это не является ошибкой, одной функции можно вполне легально задать новое тело.

Программисты, имевшие опыт работы с полиморфными языками, привыкли, что попытка повторного объявления функции должна привести к исключительной ситуации, останавливающей работу программы. Поэтому для функции можно выбрать любое имя, а если выбор будет неудачным, произойдет ошибка, подсказывающая, какое имя уже занято. В JavaScript такого не происходит, требуя от программиста повышенного внимания при разработке своих функций.

Более того, нельзя использовать одинаковые имена для переменных и функций. Точнее, прямого запрета нет, но если объявить функцию с именем уже существующей переменной, то переменная исчезнет и появится функция. Данные из переменной будут потеряны безвозвратно. И наоборот, если ввести переменную, совпадающую по имени с функцией, то она «перекроет» собой эту функцию и функция станет недоступной. Причем всё это произойдет без программных ошибок, предупреждений или уведомлений.

Следующий код функции иллюстрирует описанные особенности.

```
function parity(x) {  
    if(x % 2 == 0)
```

```
    parity = "even""";  
else  
    parity = "odd";  
}
```

После объявления функции она будет доступна под именем «**parity**». Однако, после первого вызова функции с любым аргументом, имя «**parity**» будет переопределено телом функции как переменная и, вместо функции, она будет хранить строку «**even**» или «**odd**» в зависимости от переданного значения «**x**».

В JavaScript следует крайне внимательно следить за выбором имен переменных и функций!

# Область видимости переменной

Одной из особенностей составления программ, отличающейся для разных языков программирования, является понятие «области видимости» переменных, ее границ и возможностей. Областью видимости переменной называют участок программы, в котором возможен доступ к данной переменной по ее имени.

Следует отметить, что JavaScript в аспекте видимости переменных довольно сильно отличается от многих подобных языков программирования, поэтому постарайтесь обратить повышенное внимание к данному разделу, особенно, если Вы параллельно изучаете или изучали раньше другие языки программирования.

Рассмотрим следующий пример. Мы создаем функцию «`logX`», которая будет отображать в консоли значение переменной «`x`»

```
function logX() {  
    console.log(x);  
}
```

Поскольку мы неоднократно использовали переменную с именем «`x`» в предыдущих упражнениях, обновите вкладку браузера, в которой Вы работаете с консолью или откройте новую вкладку и вызовите новую консоль. Затем введите определение приведенной выше функции и нажмите «`Enter`».

Попробуйте вызвать функцию, набрав «`logX()`» и нажав «`Enter`». В результате ее выполнения должна появиться ошибка о неопределенной переменной «`x`» (см. рис. 46). Это указывает на первую особенность, приводящую иногда к ошибкам начинающих программистов.

Ранее мы говорили, что любая неопределенная переменная имеет тип «`undefined`» и обращение к любому имени не должно приводить к ошибке, даже если это имя нигде ранее не определялось. Ошибка возникает из-за подмены понятий типа переменной и ее значения. Действительно, тип неопределенной переменной является «`undefined`», но для получения этого типа необходимо указать оператор «`typeof`» (см. раздел 13). Также не приводит к ошибке присвоение значения неопределенной переменной (например, `x=1`), т.к. переменная будет автоматически создана в момент первого присвоения. К ошибке приводит обращение к значению неопределенной переменной — так называемая попытка ее чтения (операцией `console.log(x);`).

Для того чтобы устраниТЬ описанную ошибку, определим переменную «`x`», присвоив ей значение 1. Введем в консоли «`x=1`» и нажмем «`Enter`». После чего снова вызовем функцию «`logX()`». Как результат увидим значение «`1`» (см. рис. 5).

Поменяем значение переменной «`x`». Введем «`x=2`» и нажмем «`Enter`». Затем «`logX()`» и снова «`Enter`». Теперь результатом работы функции будет `2`.

Как следует из приведенного примера, функция «`logX()`» имеет доступ к переменной, описанной вне этой функции. Напомним, что функция является отдельной программой, в принципе, не зависимой от других функций и основной

программы. И наличие доступа в функциях к ресурсам основной программы — возможность не очевидная.

The screenshot shows the 'Console' tab of a browser's developer tools. It displays the following interaction:

```

> function logX(){console.log(x);}
< undefined
> logX()
    ● > Uncaught ReferenceError: x is not defined
        at logX (<anonymous>:1:29)
        at <anonymous>:1:1
> x=1
< 1
> logX()
    1
< undefined
> x=2
< 2
> logX()
    2
< undefined
>

```

The last line shows a red error message: "Uncaught ReferenceError: x is not defined". This occurs because the variable 'x' is only defined in the global scope (as shown by the first 'x=1' command), but is being used within the function 'logX()' which is defined in a local scope.

Рисунок 5

С точки зрения тела функции, эта переменная является глобальной, то есть объявленной во внешнем блоке кода, вызвавшем функцию, но остающейся доступной в текущем блоке (в функции). Обеспечивает такую видимость переменных специальный объект, называемый глобальным. В JavaScript роль глобального объекта играет объект «`window`». Об этом частично было рассказано в разделе 7 текущего урока — все переменные, объявляемые без принадлежности к какому-либо объекту, формально принадлежат объекту «`window`». Этому же объекту принадлежать переменные, описанные в консоли браузера.

При попытке доступа к глобальной переменной из любого места программы происходит запрос именно

к этому объекту. Все переменные (и другие свойства) объекта «`window`» формируют так называемую «глобальную область видимости» (ГОВ), являющуюся доступной во всех программных инструкциях, независимо от их вложенности друг в друга (имеется в виду, что одна функция может вызывать другую функцию и так далее по цепочке).

С другой стороны, объект «`window`» формируется отдельно для разных вкладок (или окон) браузера, что делает невозможным в одной вкладке использование переменных, объявленных в программном коде другой вкладки. Понятие глобального объекта распространяется только на одну вкладку. У разных вкладок — разные ГОВ.

Следует обратить внимание на то, что изменение, вносимые в глобальные переменные любым из программных блоков, сразу же отразятся и в других блоках, использующих эту переменную. Это происходит потому, что в разных блоках используется одна и та же глобальная программная переменная. Эффект, связанный с изменением значений глобальных переменных в результате работы функции носит название «побочного действия функции». В некоторых случаях это используется для обмена данными между различными функциями, но может носить и нежелательный характер.

Для иллюстрации эффекта побочного действия используем ранее созданный файл с функцией `incAndLog` (*исходный код доступен в папке Sources — файл js1\_9.html*) и внесем в него изменения — исключим параметр «`x`» из объявления функции (вместо объявления «`function incAndLog(x)`» запишем «`function incAndLog()`»). Остальной код остается без изменений:

```

<!doctype html>
<html>
    <head>
    </head>

    <body>
        <p id="Log"></p>
        <script>
            function incAndLog() {
                x = x+1;
                alert("inc x = " + x);
                Log.innerHTML += "<br>inc x = " + x;
            }
            x = 2;
            Log.innerHTML = "x = " + x;
            incAndLog(x);
            Log.innerHTML = "<br>x = " + x;
        </script>
    </body>
</html>

```

Исключение параметра из объявления функции нарушит механизм создания копии аргумента при вызове функции. Переменная «`x`» в ее теле теперь будет ссылаться не на свой параметр (как ранее), а на глобальную переменную «`x`» (а точнее, «`window.x`»).

В таком случае, во время работы функции инструкция «`x = x+1`» изменит значение глобальной переменной — той же, что используется в самом блоке `<script>`. После выхода из функции и возврата в блок `<script>` переменная будет иметь новое значение.

Сохраните файл и откройте его в браузере. Точно так же, как и в предыдущем случае появится сообщение «`inc`

`x = 3`», что свидетельствует о правильном чтении глобальной переменной в теле функции. После закрытия окна сообщения на странице появятся надписи, последняя из которых (`x=3`) выводится после вызова функции и свидетельствует о внесении изменений в переменную «`x`».

← → C

```
x = 2
inc x = 3
x = 3
```

*Рисунок 6*

Обратите внимание, что в предыдущем примере с этой функцией глобальная переменная оставалась равной `2`, то есть не менялась при работе функции. Это обеспечивалось введением параметра, из-за чего создавалась копия переменной.

Побочный эффект, связанный с влиянием на глобальные переменные может использоваться для обмена большими данными, для которых создание параметров-копий потребует значительного количества дополнительной памяти и времени на копирование. С другой стороны, побочный эффект может носить нежелательные последствия, если он возник из-за невнимательности программиста.

**Рассмотрим следующий пример.** Мы хотим несколько раз при помощи цикла запустить функцию «`logArguments`», описанную в предыдущем разделе, с передачей ей новых значений аргументов. Повторим определение функции:

```
function logArguments() {
    for(i=0; i<arguments.length; i++)
        console.log("argument"+(i+1)+" =
                    "+arguments[i])
}
```

Ведите или скопируйте в консоль это определение и нажмите «Enter», убедитесь в отсутствие ошибок.

Далее организуем цикл, три раза вызывающий эту функцию с разными значениями аргументов:

```
for(i=0;i<3;i++)
    logArguments(i, i+3, 2*i, i*i)
```

Ведите или скопируйте в консоль этот цикл и нажмите «Enter». В результате мы увидим только один запуск функции вместо ожидаемых трех (см. рис. 7).

```
function logArguments(){
    for(i=0; i<arguments.length; i++)
        console.log("argument"+(i+1)+" =
                    "+arguments[i])
}
< undefined
> for(i=0;i<3;i++)
    logArguments(i, i+3, 2*i, i*i)
argument1 = 0
argument2 = 3
argument3 = 0
< undefined
>
```

Рисунок 7

Объясняется это тем, что для организации цикла в функции «`logArguments`» используется такая же переменная «`i`», как и для цикла, который обеспечивает повторный вызов этой функции в консоли. После первого вызова функции переменная во внутреннем цикле пройдет значения, согласно количеству переданных аргументов (`0, 1, 2, 3`) и после очередного увеличения (до `4`) нарушит цикловое условие и приведет к завершению функции. То есть после вызова функции переменная «`i`» будет иметь значение `4`.

Повторная итерация цикла в консоли проверит цикловое условие и сочтет его ложным, т.к. текущее значение «`i`» больше предельного для цикла значения (`3`), в результате чего цикл будет остановлен. Проблема заключается в использовании одной переменной для разных задач и иллюстрирует негативные последствия от побочного действия функции.

Для предотвращения подобных негативных явлений в функциях могут быть описаны дополнительные переменные, не влияющие на глобальную область видимости. Такие переменные называют локальными, а код, в котором к ним возможен доступ — локальной областью видимости (ЛОВ). Для того чтобы создать локальную переменную используется ключевое слово `«var»` перед именем переменной. Начиная со стандарта `«ES6»` также можно использовать `«let»`. Инструкции `«var»` и `«let»` могут использоваться сами по себе (для декларации локальных переменных), а также в составе оператора присваивания:

```
var a;  
var b = 1;  
let c;  
let d = 2;
```

Локальные переменные имеют приоритет перед глобальными. Другими словами, если локальная и глобальная переменные имеют одинаковые имена, то в данном блоке будет использована локальная переменная. К глобальной переменной остается возможность обратиться через глобальный объект «[window](#)», указав после точки требуемое имя переменной (например, [window.x](#)).

Введение локальных переменных не означает, что доступ к глобальным переменным прекращается — если интерпретатор не находит в данном блоке локальной переменной с указанным именем, то автоматически будет использоваться глобальная переменная. Каждая переменная, которая предполагается как локальная, должна быть объявлена с применением инструкции «[var](#)» или «[let](#)».

Для того чтобы устраниТЬ эффект побочного действия функции в рассмотренном выше примере, в определении функции «[logArguments](#)» нужно указать, что переменная «[i](#)» должна быть локальной, то есть добавить декларацию «[var i](#)» в начале функции, либо добавить «[var](#)» перед «[i](#)» непосредственно в операторе цикла:

```
for (var i=0; i<arguments.length; i++)
```

В таком случае в теле функции будет использоваться локальная переменная «[i](#)», тогда как в консоли (за пределами функции) — глобальная. Хотя эти переменные име-

ют одинаковые имена, на самом деле они принадлежат различным объектам (различным областям видимости) и не влияют друг на друга.

Введите новое определение функции «`logArguments`» с добавлением инструкции «`var`» в консоли и повторите вызов цикла ее запуска (повтор команд в консоли и просмотр их истории возможен при помощи стрелок вверх и вниз на клавиатуре). Убедитесь, что с введением локальной переменной запуск функции происходит трижды, в полном соответствии с первым циклом. То есть побочное действие функции устранено.

Приведем еще один пример, иллюстрирующий положительное использование эффекта побочного действия функций. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке *Sources* — файл *js1\_10.html*).

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <p id="Log"></p>
    <script>
      function prepareStr() {
        str = "This string was prepared by
              function";
      }
      prepareStr();
      Log.innerHTML = "str = " + str;
    </script>
  </body>
</html>
```

В теле html документа создан пустой абзац «`<p id="Log"></p>`», предназначенный для последующего вывода в него данных.

Функция «`prepareStr`», описанная в данном примере, выполняет единственное действие — присваивает переменной «`str`» значение «`This string was prepared by function`». Поскольку функция не имеет параметров и перед именем переменной «`str`» не применяются модификаторы «`var`» или «`let`», обращение совершается к глобальной области видимости.

В блоке `<script>` эта функция вызывается «`prepareStr();`» после чего в абзац выводится сообщение, состоящее из надписи «`str =`» к которому добавляется содержание переменной «`str`» (`Log.innerHTML = "str = " + str;`).

Сохраните файл и откройте его при помощи браузера. Убедитесь, что на странице появляется надпись «`str = This string was prepared by function`».

Обратите внимание, что в самом блоке `<script>` переменная с именем «`str`» не создавалась, но использовалась. Эта переменная была создана в результате выполнения тела функции за счет того, что обращение велось к глобальной области видимости. Подобным образом побочное действие функций можно использовать для подготовки определенных данных перед их выводом или обработкой.

## Поднятие объявлений

Отличительной особенностью JavaScript, по сравнению со многими другими языками программирования, является принцип поднятия объявлений (англ. *hoisting*).

Заключается он в том, что перед выполнением программы все объявления функций и переменных «поднимаются» в начало программы или, точнее, в начало своей области видимости. И только затем код выполняется.

Поднятие объявлений позволяет использовать функции и переменные в коде до их фактического объявления. С одной стороны, это предоставляет удобство для описания всех дополнительных функций в конце программы, не загромождая основной алгоритм. С другой стороны, это может запутать программистов, привыкших к другим языкам, где подобного механизма нет.

**Рассмотрим пример:** создадим функцию, в которой дважды выводится содержимое переменной «`x`» — до и после её объявления

```
function showHoisting() {  
    console.log("x before declaration: "+x);  
    var x = 2;  
    console.log("x after declaration: "+x);  
}
```

Ведите или скопируйте в консоль это определение и нажмите «`Enter`», убедитесь в отсутствие ошибок.

Затем вызовите эту функцию, набрав «`showHoisting()`» в консоли. Ранее мы видели, что попытка доступа к переменной до ее объявления приводит к программной ошибке. Однако в данном случае мы наблюдаем значение «`undefined`», выведенное без ошибки (см. рис. 49). Полнотью соответствует ожиданиям второй вывод со значением «`x`» равным `2`, полученным после объявления переменной инструкцией «`var x = 2`».

Повторим вызов функции, только перед этим создадим глобальную переменную «`x`» со значением `1` (введем в консоли `x=1`). По логике, до объявления «`var x`» в теле функции под переменной «`x`» должна пониматься ее глобальная тезка. Но повторный вызов функции все так же дает «`undefined`» вместо ожидаемой единицы.

```

    Elements   Console   Sources
    top
    > function showHoisting(){
        console.log("x before var: "+x);
        var x = 2;
        console.log("x after var: "+x);
    }
    < undefined
    > showHoisting()
    x before var: undefined
    x after var: 2
    < undefined
    > x=1
    < 1
    > showHoisting()
    x before var: undefined
    x after var: 2
    < undefined
    >
  
```

Рисунок 8

Полученные результаты иллюстрируют эффект, связанный с работой поднятия объявлений. Благодаря ему инструкция «`var x`» перед выполнением функции переносится в самое начало функции, и первая операция «`console.log("x before declaration: "+x);`» на самом деле становится второй.

Как следствие, получаем две особенности. Во-первых, перемещение инструкции «`var x`» создает локальную пе-

переменную «`x`» с пока еще не заданным значением («`undefined`»), что предотвращает ошибку чтения неопределенной переменной. Во-вторых, локальная переменная «перекрывает» собой глобальную сразу после запуска функции, а не после команды «`var x=2`».

Обратите внимание, что в начало области видимости поднимается только объявление («`var x`»), но не присваивание. Значение 2 переменная «`x`» приобретает в том месте программы, в котором указана эта операция.

Поднятие объявлений и обработка их перед началом выполнения кода позволяет повторно использовать инструкцию «`var`» с тем же именем переменной без появления ошибки повторного объявления. Это также является необычным для многих популярных языков программирования и может служить причиной ошибок и недопонимания.

Если программист в другом языке пишет большую программу и сам уже забыл, какие переменные он использовал, то для него случайное повторное объявление переменной с тем же именем, что и раньше, приведет к ошибке, напомнив о том, что это имя уже занято. В JavaScript подобной ошибки не возникнет, повторное объявление будет проигнорировано, и в операции будет использоваться та же переменная, что была объявлена ранее. Это может послужить источником неправильной работы программы, если программист надеется, что объявляет новую переменную, а отсутствие ошибки воспринимается как некий предохранитель от повторного объявления.

Повторим еще раз: в JavaScript требуется повышенное внимание к выбору имен переменных и функций. Реко-

мендации по именованию разработаны не только ради красоты и читаемости кода, но и для предотвращения ряда ошибок.

## Различия деклараций `var`, `let` и `const`

Очередной особенностью JavaScript, по сравнению с другими языками программирования, является слабое структурирование областей видимости. Во многих языках каждый блок кода, заключенный в группирующий оператор, является собственной областью видимости. Переменные, описанные в разных блоках, при этом не взаимодействуют между собой, даже если имеют одинаковые имена.

В JavaScript отдельные области видимости создаются только внутри функций и действуют в пределах их тел. Другие блоки, циклы или условные операторы внутри одной функции используют ЛОВ самой функции. А вследствие поднятия объявлений, все декларации из всех блоков поднимаются в начало функции.

Например, объявленная в цикле переменная становится доступной и вне тела цикла. Причем доступной как после цикла, так и до него — с самого начала функции (или всей программы, если цикл описан в ней). Для программистов C++, C#, да и большинства подобных языков это кажется парадоксальным.

```
// here i does exist
for(var i=0; i<arguments.length; i++) { // here i
                                         // exists normally
}
// and here i still exists
```

Для того чтобы уменьшить количество ошибок, связанных с наличием у программиста предыдущего опыта, и сделать JavaScript более дружественным для таких программистов начиная со стандарта «ES6» (ES-2015) было введено ключевое слово «`let`». Переменная, объявленная с этим декларатором, становится доступной (локальной) только для того блока, в котором она описана (как это принято в других языках программирования).

```
// here i does not exist
for(let i=0; i<arguments.length; i++) { // here i exists
}
// and here i does not exist
```

Также переменная, объявленная оператором «`let`», не может быть повторно определена ни оператором «`let`», ни оператором «`var`» (в пределах данного блока). Если переменная была ранее создана при помощи «`var`», то попытка ее повторного создания с применением «`let`» приведет к ошибке. Это поведение возвращает программистам «предохранитель» повторного объявления переменных, не работающий для декларатора «`var`».

Упредить ошибки, связанные со случайным изменением значения переменной, можно при помощи ее объявления с ключевым словом «`const`» (также начиная со стандарта «ES6»). Объявленные с этим ключевым словом переменные:

- Являются видимыми только в «своем» блоке.
- Не могут быть повторно объявлены другим оператором («`var`», «`let`» или «`const`») в дальнейшем коде данного блока.

- Приведут к ошибке объявления, если в данном блоке ранее были объявлены другим оператором («`var`», «`let`» или «`const`») переменные с таким же именем.
- Не могут менять свое значение повторным присваиванием (возможно только одно присваивание значения переменной).

Деклараторы «`let`» и «`const`» не могут быть использованы в составе одно-операторных (*англ. single-statement*) блоков в условиях или циклах. То есть к ошибке приведет запись

```
if(true)
    let a=1;
```

В то же время, если заменить «`let`» на «`var`» либо использовать группирующий оператор (взять «`{let a=1;}`» в фигурные скобки), то ошибка исчезнет.

Каждый из деклараторов «`var`», «`let`» или «`const`» имеет свои особенности и может привести к более удобным способам организации кода для различных задач. Отдавать однозначное предпочтение одному из них нет необходимости, так же как и считать какой-либо заведомо худшим. В наиболее современном стандарте языка «ES9(2018)» предусмотрены все перечисленные операторы.

# Рекурсия

Заканчивая вводное изучение функций, следует отметить такую возможность работы с функциями, как рекурсию. Рекурсией называют ситуацию, в которой функция вызывает сама себя.

В математике существует понятие рекуррентной формулы, близкое по смыслу с рекурсией в программировании, — формулы определения новых значений величин с использованием предыдущих значений величин. Эти термины иногда путают из-за схожего звучания и подобного смысла. Страйтесь использовать их грамотно, согласно ситуации.

При помощи рекурсии удобнее всего создавать программные реализации рекуррентных математических формул. Наверное, самым популярным примером для демонстрации рекурсии является функция вычисления факториала числа. Напомним, факториалом числа называют произведение всех целых чисел от 1 до данного числа. То есть  $n! = 1 \cdot 2 \cdot \dots \cdot n$ . Например,  $2! = 1 \cdot 2 = 2$ ,  $3! = 1 \cdot 2 \cdot 3 = 6$ ,  $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$  и т.д.

Рассчитать факториал числа «**n**» можно при помощи обычного цикла-счетчика:

```
factorial = 1;
for(i=2; i<=n; i++)
    factorial *= i;
```

Для получения значения факториала вводится переменная «**factorial**», которая инициализируется значением «1». Затем циклически для всех чисел «**i**» от двойки до

данного числа «**n**» эта переменная умножается на величину «**i**» (**factorial \*= i**). По окончанию цикла в переменной «**factorial**» будет содержаться итоговый результат умножения  $n! = 1 \cdot 2 \cdot \dots \cdot n$ .

Для составления рекурсивного алгоритма нужно отметить, что получить факториал числа **n** можно умножив это число **n** на факториал предыдущего числа  $(n-1)!$ , т.к. в нем уже есть все произведения, кроме самого числа **n** ( $4! = 1 \cdot 2 \cdot 3 \cdot 4 = (1 \cdot 2 \cdot 3) \cdot 4 = 3! \cdot 4$ ):

$$n! = (n-1)! \cdot n.$$

Это и есть пример рекуррентной математической формулы, определяющей факториал числа **n!**, используя определение факториала предыдущего числа  $(n-1)!$ . В таком случае нужно отдельно указать значение факториала первого числа **1!=1**, иначе определение никогда не прекратится из-за ссылок на предыдущие числа.

Аналогично математическому определению, рекурсивная функция должна состоять из двух частей: 1) возврат определенного значения при некотором значении параметра и 2) повторный вызов себя с другим значением аргумента. Соблюдая изложенные требования, опишем функцию для расчета факториала:

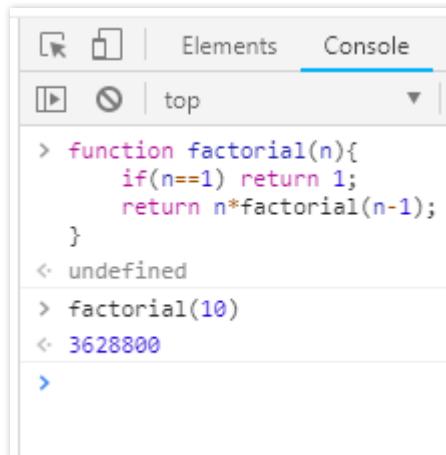
```
function factorial(n) {
    if(n==1) return 1;
    return n*factorial(n-1);
}
```

Первая инструкции функции проверяет параметр на известную величину (**n==1**) и, в случае совпадения,

возвращает конкретное значение (`return 1`). Если же проверка не проходит, возвращается результат выражения «`n*factorial(n-1);`», что приведет к повторному вызову функции, но уже с аргументом `n-1`, умножению его на `n` и передаче в точку вызова. Каскад вызовов закончится уменьшением `n` до `1`, после чего все возвраты соберутся в выражение, умножающее числа между собой.

Ведите или скопируйте в консоль это определение и нажмите «`Enter`», убедитесь в отсутствие ошибок.

Затем вызовите эту функцию, набрав «`factorial(10)`» в консоли. Убедитесь, что полученный результат соответствует приведенному на рисунке.



The screenshot shows a browser's developer tools console tab labeled "Console". The console interface includes buttons for back, forward, and search, and dropdown menus for "Elements" and "Console". The console area displays the following interaction:

```

> function factorial(n){
    if(n==1) return 1;
    return n*factorial(n-1);
}
<- undefined
> factorial(10)
<- 3628800
>

```

Рисунок 9

Сравнивая два способа вычислить факториал — при помощи цикла и при помощи рекурсии — может показаться, что рекурсивный способ более громоздкий и сложный. С первого взгляда так оно и есть, но это только с первого взгляда. Дело в том, что функции, в отли-

чие от циклов, могут выполняться асинхронно, то есть параллельно одна с другой, используя свободные ядра процессора или разные процессоры, если их несколько. При кажущейся сложности, рекурсивное решение может выполняться значительно быстрее и равномерно использовать аппаратные ресурсы. Детали этого материала выходят за рамки первого знакомства с функциями и будут рассмотрены в дальнейших уроках.

Возможности рекурсии значительно превосходят работу с рекуррентными математическими объектами. В частности, рекурсия представляет собой альтернативу циклам при работе с комплексными данными.

Например, поставим задачу вывести ряд чисел от единицы до введенного пользователем числа с дополнительным ограничением — циклы использовать не разрешается. С первого взгляда кажется, что не зная величины предельного числа, без циклов задача не может быть решена. Но тут на помощь приходит рекурсия. Поскольку нам понадобится пользовательский ввод, оформим программу в виде отдельного файла, а не в консоли. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке Sources — файл js1\_11.html).

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <script>
      var num = +prompt("Final number:");
      alert(stringWithNumbers(num));
    </script>
  </body>
</html>
```

```
function stringWithNumbers(n) {  
    if(n==1) return "1";  
    return stringWithNumbers(n-1) +  
        ", " + n;  
}  
</script>  
</body>  
</html>
```

Основу программы составляет функция «[stringWithNumbers](#)», которая формирует строку, содержащую ряд чисел. По канонам рекурсии, она содержит «выход» с возвратом конкретного значения [1](#) и повторный вызов себя с уменьшенным значением аргумента. К полученному значению дописывается текущее число, и результат возвращается в точку вызова.

Взаимодействие с пользователем заключается в запросе граничного числа и выводе итогового результата. Обратите внимание, что функция «[stringWithNumbers](#)» описана после того, как она используется в команде [«alert»](#). Это снова демонстрирует действие поднятия определений, только на этот раз для функций.

Сохраните файл и откройте его при помощи браузера. В появившемся диалоговом окне введите число и нажмите [«OK»](#). Должно появиться новое окно с рядом чисел от [1](#) до введенного Вами числа.

**Задание:** модифицируйте программу, чтобы числа выводились в обратном порядке — начиная с введенного числа и до единицы. Предусмотрите реакцию на неправильный ввод пользователя.

Вторым классическим примером использование рекурсии является генератор чисел Фибоначчи. Каждое из этих чисел является суммой двух предыдущих чисел:  $f_n = f_{n-1} + f_{n-2}$ . Первыми двумя числами идут две единицы. Начало ряда чисел Фибоначчи будет 1, 1, 2(1+1), 3(2+1), 5(3+2), 8, 13 и т.д.

**Найдите ошибку.** Следующий код, описывающий функцию для генератора чисел Фибоначчи, содержит ошибку. Исправьте ее перед запуском кода.

```
function Fibonacci(n) {
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

(В функции не предусмотрен возврат «конечных» значений, с которых начинается ряд. Первой строкой функции должна быть «`if(n<3) return 1;`», возвращающая единицу для параметра «`n`» равного 1 или 2, то есть меньшего чем 3).

**Задание:** создайте программу для вывода чисел Фибоначчи. Количество чисел запрашивается у пользователя, сам ряд выдается в отдельном диалоговом окне. Предусмотрите реакцию на неправильный ввод пользователя.

Под конец приведем одно терминологическое уточнение. Термин «рекурсия» не ограничивается повторным само-вызовом функции. Если в теле функции происходит вызов самой себя, то такую ситуацию называют «авто-рекурсией» или «само-рекурсией».

Если одна функция вызывает другую, а та, в свою очередь, вызывает первую, то говорят о «взаимной» или

«косвенной» рекурсии. В случае косвенной рекурсии цепочка взаимных вызовов может быть и более сложной, включающей три и более различные функции, вызывающие одна другую.

Чаще всего, под словом «рекурсия» (без указания конкретного типа) подразумевают именно вариант с автo-рекурсией. Этот вариант гораздо чаще применяется и в практическом программировании, и в теоретических обоснованиях параллельных вычислений. Тем не менее, следует помнить о других вариантах рекурсии и правильно использовать терминологию.

## Задание для самостоятельной работы

1. Создайте функцию `stringFrom(...)`, возвращающую строку, состоящую из значений всех переданных аргументов. Например, вызов `stringFrom('I have', 5, 'apples')` вернет строку «`I have 5 apples`»; вызов `stringFrom('X value is', true)` вернет строку «`X value is true`».
2. Создайте функцию, возвращающую значение минимального из всех переданных аргументов.
3. Создайте функцию `numbers()`, которая будет подсчитывать количество переданных числовых аргументов. Например, `numbers(1, 2, "a")` вернет значение `2`, `numbers(true, 2, false) — 1`, `numbers() — 0`.
4. Создайте функцию `mean()`, которая рассчитает среднее значение от всех числовых аргументов, игнорируя аргументы нечислового типа. Например, `mean (1, 2, "a")` вернет значение `1.5` (среднее `1` и `2`), `mean(true, 2, false) — 2`, `mean() — 0`.
5. Напишите рекурсивную функцию, которая проверяет, является ли переданный аргумент степенью двойки (например, числа  $8=2^3$ ,  $32=2^5$  — это степени двойки, а числа `7` или `12` — нет). Подсказка: если число «`x`» делится на два, то нужно проверить, является ли число «`x/2`» степенью двойки.
6. Напишите рекурсивную функцию, которая выводит число `N` «справа налево», то есть последняя цифра числа становится первой, предпоследняя — второй и т.д. (например, ввод `N=123`, вывод `321`; ввод `N= 12`, вывод `21`). Обеспечьте ввод пользова-

телем числа **N** и вывод его «справа налево» вызовом функции. Подсказка: последняя цифра числа «**x**» это остаток от деления на **10** (**x%10**), а остальные цифры можно отделить, поделив «**x**» на **10** нацело (`parseInt(x/10)`).



## Unit 5.

# Детальнее о функциях

© Денис Самойленко.

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

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

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

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