Функции

Оглавление

[Функции 0](#_Toc453755118)

[Области видимости 2](#_Toc453755119)

[Замыкания 3](#_Toc453755120)

[Домашнее задание 9](#_Toc453755121)

[Используемая литература 9](#_Toc453755122)

# Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, красиво вывести сообщение необходимо при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

**Объявление**

Пример объявления функции:

|  |
| --- |
| function showMessage() {  alert( 'Привет всем присутствующим!' );  } |

Вначале идет ключевое слово function, после него имя функции, затем список параметров в скобках (в примере выше он пустой) и тело функции – код, который выполняется при её вызове.

Объявленная функция доступна по имени, например:

|  |
| --- |
| function showMessage() {  alert( 'Привет всем присутствующим!' );  }  showMessage();  showMessage(); |

Этот код выведет сообщение два раза. Уже здесь видна главная цель создания функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

**Параметры функций**

При вызове функции ей можно передать данные, которые та использует по своему усмотрению.

Например, этот код выводит два сообщения:

|  |
| --- |
| function showMessage(from, text) { // параметры from, text  from = "\*\* " + from + " \*\*"; // здесь может быть сложный код оформления  alert(from + ': ' + text);  }  showMessage('Маша', 'Привет!');  showMessage('Маша', 'Как дела?'); |

Параметры копируются в локальные переменные функции.

Например, в коде ниже есть внешняя переменная from, значение которой при запуске функции копируется в параметр функции с тем же именем. Далее функция работает уже с параметром:

|  |
| --- |
| function showMessage(from, text) {  from = '\*\*' + from + '\*\*'; // меняем локальную переменную from  alert( from + ': ' + text );  }  var from = "Маша";  showMessage(from, "Привет");  alert( from ); // старое значение from без изменений, в функции была изменена копия |

**Аргументы по умолчанию**

Функцию можно вызвать с любым количеством аргументов.

Если параметр не передан при вызове – он считается равным undefined.

Например, функцию показа сообщения showMessage(from, text) можно вызвать с одним аргументом:

|  |
| --- |
| showMessage("Маша"); |

При этом можно проверить, и если параметр не передан – присвоить ему значение «по умолчанию»:

|  |
| --- |
| function showMessage(from, text) {  if (text === undefined) {  text = 'текст не передан';  }  alert( from + ": " + text );  }  showMessage("Маша", "Привет!"); // Маша: Привет!  showMessage("Маша"); // Маша: текст не передан |

При объявлении функции необязательные аргументы, как правило, располагают в конце списка.

Для указания значения «по умолчанию», то есть, такого, которое используется, если аргумент не указан, используется два способа:

* Можно проверить, равен ли аргумент undefined, и если да – то записать в него значение по умолчанию. Этот способ продемонстрирован в примере выше.
* Использовать оператор ||:

|  |
| --- |
| function showMessage(from, text) {  text = text || 'текст не передан';  ...  } |

Второй способ считает, что аргумент отсутствует, если передана пустая строка, 0, или вообще любое значение, которое в логическом контексте является false.

# Области видимости

Глобальными называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции function, то они – «глобальные».

В JavaScript все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object).

В браузере этот объект явно доступен под именем window. Объект window одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера, но нас здесь интересует только его роль как глобального объекта.

В других окружениях, например Node.JS, глобальный объект может быть недоступен в явном виде, но суть происходящего от этого не изменяется, поэтому далее для обозначения глобального объекта мы будем использовать "window".

Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами window.

Например:

|  |
| --- |
| var a = 5; // объявление var создаёт свойство window.a  alert( window.a ); // 5 |

Выполнение скрипта происходит в две фазы:

На первой фазе происходит инициализация, подготовка к запуску.

Во время инициализации скрипт сканируется на предмет объявления функций вида Function Declaration, а затем – на предмет объявления переменных var. Каждое такое объявление добавляется в window.

Функции, объявленные как Function Declaration, создаются сразу работающими, а переменные – равными undefined.

На второй фазе – собственно, выполнение.

Присваивание (=) значений переменных происходит, когда поток выполнения доходит до соответствующей строчки кода, до этого они undefined.

В коде ниже указано содержание глобального объекта на момент инициализации и далее последовательно по коду:

|  |
| --- |
| // На момент инициализации, до выполнения кода:  // window = { f: function, a: undefined, g: undefined }  var a = 5;  // window = { f: function, a: 5, g: undefined }  function f(arg) { /\*...\*/ }  // window = { f: function, a: 5, g: undefined } без изменений, f обработана ранее  var g = function(arg) { /\*...\*/ };  // window = { f: function, a: 5, g: function } |

# Замыкания

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске.

Мы будем называть этот объект «лексическое окружение» или просто «объект переменных».

При запуске функция создает объект LexicalEnvironment, записывает туда аргументы, функции и переменные. Процесс инициализации выполняется в том же порядке, что и для глобального объекта, который, вообще говоря, является частным случаем лексического окружения.

В отличие от window, объект LexicalEnvironment является внутренним, он скрыт от прямого доступа.

**Пример**

Посмотрим пример, чтобы лучше понимать, как это работает:

|  |
| --- |
| function sayHi(name) {  var phrase = "Привет, " + name;  alert( phrase );  }  sayHi('Вася'); |

При вызове функции:

До выполнения первой строчки её кода, на стадии инициализации, интерпретатор создает пустой объект LexicalEnvironment и заполняет его.

В данном случае туда попадает аргумент name и единственная переменная phrase:

|  |
| --- |
| function sayHi(name) {  // LexicalEnvironment = { name: 'Вася', phrase: undefined }  var phrase = "Привет, " + name;  alert( phrase );  }  sayHi('Вася'); |

Функция выполняется.

Во время выполнения происходит присвоение локальной переменной phrase, то есть, другими словами, присвоение свойству LexicalEnvironment.phrase нового значения:

|  |
| --- |
| function sayHi(name) {  // LexicalEnvironment = { name: 'Вася', phrase: undefined }  var phrase = "Привет, " + name;  // LexicalEnvironment = { name: 'Вася', phrase: 'Привет, Вася'}  alert( phrase );  }  sayHi('Вася'); |

В конце выполнения функции объект с переменными обычно выбрасывается и память очищается. В примерах выше так и происходит. Через некоторое время мы рассмотрим более сложные ситуации, при которых объект с переменными сохраняется и после завершения функции.

Из функции мы можем обратиться не только к локальной переменной, но и к внешней:

|  |
| --- |
| var userName = "Вася";  function sayHi() {  alert( userName ); // "Вася"  } |

Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем LexicalEnvironment, а затем, если её нет – ищет во внешнем объекте переменных. В данном случае им является window.

Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется [[Scope]]. Это свойство закрыто от прямого доступа, но знание о нём очень важно для понимания того, как работает JavaScript.

При создании функция получает скрытое свойство [[Scope]], которое ссылается на лексическое окружение, в котором она была создана.

В примере выше таким окружением является window, так что создаётся свойство:

|  |
| --- |
| sayHi.[[Scope]] = window |

Это свойство никогда не меняется. Оно всюду следует за функцией, привязывая её, таким образом, к месту своего рождения.

При запуске функции её объект переменных LexicalEnvironment получает ссылку на «внешнее лексическое окружение» со значением из [[Scope]].

Если переменная не найдена в функции – она будет искаться снаружи.

Именно благодаря этой механике в примере выше alert(userName) выводит внешнюю переменную. На уровне кода это выглядит как поиск во внешней области видимости, вне функции.

Если обобщить:

Каждая функция при создании получает ссылку [[Scope]] на объект с переменными, в контексте которого была создана.

При запуске функции создаётся новый объект с переменными LexicalEnvironment. Он получает ссылку на внешний объект переменных из [[Scope]].

При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом – по этой ссылке.

Выглядит настолько просто, что непонятно – зачем вообще говорить об этом [[Scope]], об объектах переменных. Сказали бы: «Функция читает переменные снаружи» – и всё. Но знание этих деталей позволит нам легко объяснить и понять более сложные ситуации, с которыми мы столкнёмся далее.

Значение переменной из внешней области берётся всегда текущее. Оно может быть уже не то, что было на момент создания функции.

Например, в коде ниже функция sayHi берёт phrase из внешней области:

|  |
| --- |
| var phrase = 'Привет';  function say(name) {  alert(phrase + ', ' + name);  }  say('Вася'); // Привет, Вася (\*)  phrase = 'Пока';  say('Вася'); // Пока, Вася (\*\*) |

На момент первого запуска (\*), переменная phrase имела значение 'Привет', а ко второму (\*\*) изменила его на 'Пока'.

Это естественно, ведь для доступа к внешней переменной функция по ссылке [[Scope]] обращается во внешний объект переменных и берёт то значение, которое там есть на момент обращения.

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны.

Таково стандартное определение, которое есть в Wikipedia и большинстве серьёзных источников по программированию. То есть, замыкание – это функция + внешние переменные.

Тем не менее, в JavaScript есть небольшая терминологическая особенность.

Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.

Иногда говорят «переменная берётся из замыкания». Это означает – из внешнего объекта переменных.

Внутри функции можно объявлять не только локальные переменные, но и другие функции.

К примеру, вложенная функция может помочь лучше организовать код:

|  |
| --- |
| function sayHiBye(firstName, lastName) {  alert( "Привет, " + getFullName() );  alert( "Пока, " + getFullName() );  function getFullName() {  return firstName + " " + lastName;  }  }  sayHiBye("Вася", "Пупкин"); // Привет, Вася Пупкин ; Пока, Вася Пупкин |

Здесь, для удобства, создана вспомогательная функция getFullName().

Вложенные функции получают [[Scope]] так же, как и глобальные. В нашем случае:

|  |
| --- |
| getFullName.[[Scope]] = объект переменных текущего запуска sayHiBye |

Благодаря этому getFullName() получает снаружи firstName и lastName.

Заметим, что если переменная не найдена во внешнем объекте переменных, то она ищется в ещё более внешнем (через [[Scope]] внешней функции), то есть, такой пример тоже будет работать:

|  |
| --- |
| var phrase = 'Привет';  function say() {  function go() {  alert( phrase ); // найдёт переменную снаружи  }  go();  }  say(); |

Рассмотрим более «продвинутый» вариант, при котором внутри одной функции создаётся другая и возвращается в качестве результата.

В разработке интерфейсов это совершенно стандартный приём, функция затем может назначаться как обработчик действий посетителя.

Здесь мы будем создавать функцию-счётчик, которая считает свои вызовы и возвращает их текущее число.

В примере ниже makeCounter создает такую функцию:

|  |
| --- |
| function makeCounter() {  var currentCount = 1;  return function() { // (\*\*)  return currentCount++;  };  }  var counter = makeCounter(); // (\*)  // каждый вызов увеличивает счётчик и возвращает результат  alert( counter() ); // 1  alert( counter() ); // 2  alert( counter() ); // 3  // создать другой счётчик, он будет независим от первого  var counter2 = makeCounter();  alert( counter2() ); // 1 |

Как видно, мы получили два независимых счётчика counter и counter2, каждый из которых незаметным снаружи образом сохраняет текущее количество вызовов.

Где? Конечно, во внешней переменной currentCount, которая у каждого счётчика своя.

Если подробнее описать происходящее:

1. В строке (\*) запускается makeCounter(). При этом создаётся LexicalEnvironment для переменных текущего вызова. В функции есть одна переменная var currentCount, которая станет свойством этого объекта. Она изначально инициализуется в undefined, затем, в процессе выполнения, получит значение 1:

|  |
| --- |
| function makeCounter() {  // LexicalEnvironment = { currentCount: undefined }  var currentCount = 1;  // LexicalEnvironment = { currentCount: 1 }  return function() { // [[Scope]] -> LexicalEnvironment (\*\*)  return currentCount++;  };  }  var counter = makeCounter(); // (\*) |

1. В процессе выполнения makeCounter() создаёт функцию в строке (\*\*). При создании эта функция получает внутреннее свойство [[Scope]] со ссылкой на текущий LexicalEnvironment.
2. Далее вызов makeCounter() завершается и функция (\*\*) возвращается и сохраняется во внешней переменной counter (\*).

На этом создание «счётчика» завершено.

Итоговым значением, записанным в переменную counter, является функция:

|  |
| --- |
| function() { // [[Scope]] -> {currentCount: 1}  return currentCount++;  }; |

Возвращённая из makeCounter() функция counter помнит (через [[Scope]]) о том, в каком окружении была создана.

Это и используется для хранения текущего значения счётчика.

Далее, когда-нибудь, функция counter будет вызвана. Мы не знаем, когда это произойдёт. Может быть, прямо сейчас, но, вообще говоря, совсем не факт.

Эта функция состоит из одной строки: return currentCount++, ни переменных ни параметров в ней нет, поэтому её собственный объект переменных, для краткости назовём его LE – будет пуст.

Однако, у неё есть свойство [[Scope]], которое указывает на внешнее окружение. Чтобы увеличить и вернуть currentCount, интерпретатор ищет в текущем объекте переменных LE, не находит, затем идёт во внешний объект, там находит, изменяет и возвращает новое значение:

|  |
| --- |
| function makeCounter() {  var currentCount = 1;  return function() {  return currentCount++;  };  }  var counter = makeCounter(); // [[Scope]] -> {currentCount: 1}  alert( counter() ); // 1, [[Scope]] -> {currentCount: 1}  alert( counter() ); // 2, [[Scope]] -> {currentCount: 2}  alert( counter() ); // 3, [[Scope]] -> {currentCount: 3} |

# Домашнее задание

* Напишите функцию min, принимающую два аргумента, и возвращающую минимальный из них.

|  |
| --- |
| console.log(min(0, 10)); // → 0  console.log(min(0, -10)); // → -10 |

* Символ номер N строки можно получить, добавив к ней .charAt(N) ( “строчка”.charAt(5) ) – схожим образом с получением длины строки при помощи .length. Возвращаемое значение будет строковым, состоящим из одного символа (к примеру, “к”). У первого символа строки позиция 0, что означает, что у последнего символа позиция будет string.length – 1. Другими словами, у строки из двух символов длина 2, а позиции её символов будут 0 и 1.Напишите функцию countBs, которая принимает строку в качестве аргумента, и возвращает количество символов “B”, содержащихся в строке.Затем напишите функцию countChar, которая работает примерно как countBs, только принимает второй параметр — символ, который мы будем искать в строке (вместо того, чтобы просто считать количество символов “B”). Для этого переделайте функцию countBs.
* \* Ноль чётный. Единица нечётная. У любого числа N чётность такая же, как у N-2.Напишите рекурсивную функцию isEven согласно этим правилам. Она должна принимать число и возвращать булевское значение. Потестируйте её на 50 и 75. Попробуйте задать ей -1. Почему она ведёт себя таким образом? Можно ли её как-то исправить?

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)