1.Понятие языка программирования JavaScript.

HTML - это язык разметки, который мы используем для визуального и смыслового структурирования нашего web контента, например, определяем параграфы, заголовки, таблицы данных, или вставляем изображения и видео на страницу.

CSS - это язык стилей с помощью которого мы придаём стиль отображения нашего HTML контента, например придаём цвет фону (background) и шрифту, придаём контенту многоколоночный вид.

Ядро языка JavaScript состоит из некоторого количества обычных возможностей, которые позволяют делать следующее:

Хранить данные внутри переменных. В примере выше, мы, например, запрашивали ввод нового имени, которое нужно было ввести, затем сохраняли имя в переменной name.

Операции над фрагментами текстов (известными в программировании как "строки"). Запускать код в соответствии с определёнными событиями происходящими на web странице.

API - это готовые наборы блоков кода, которые позволяют разработчику реализовывать программы, которые в противном случае было бы трудно или невозможно реализовать.

API-интерфейсы браузера встроены в ваш веб-браузер и могут отображать данные из окружающего компьютерного окружения или делать полезные сложные вещи.

API-интерфейс DOM (Document Object Model) позволяет вам манипулировать HTML и CSS, создавать, удалять и изменять HTML, динамически применять новые стили к вашей странице и т. д.. Каждый раз, когда вы видите всплывающее окно на странице или какое-то новое содержимое, Как мы видели выше в нашем простом демо), например, это DOM в действии.

API геолокации извлекает географическую информацию. Так Google Maps может найти ваше местоположение и нанести его на карту.

Когда вы загружаете страничку в браузере, вы запускаете ваш код (HTML, CSS и JavaScript) внутри исполняемой среды (внутри вкладки браузера). Это как будто фабрика берёт сырьё (некий код) и выдаёт продукцию (веб-страничку).

**Основы синтаксиса языка JavaScript: переменные, типы данных, преобразование типов данных.**

JavaScript заимствует большую часть синтаксиса из Java, но также испытал влияние таких языков, как Awk, Perl и Python.

JavaScript чувствителен к регистру и использует кодировку символов Unicode.

В JavaScript инструкции называются statements и разделяются точкой с запятой (;). Пробел (space), табуляция (tab) и перевод строки (newline) называются пробельными символами (whitespace). Исходный текст скриптов на JavaScript сканируется слева направо и конвертируется в последовательность входных элементов, являющихся токенами (tokens), управляющими символами, символами конца строки, комментариями или пробельными символами.

Комментарии

Синтаксис комментариев является таким же, как и в C++ и во многих других языках:

// Комментарий, занимающий одну строку./\* Комментарий,

занимающий несколько строк.

\*//\* Нельзя вкладывать /\* комментарий в комментарий \*/ SyntaxError \*/

Объявления

В JavaScript существует три вида объявлений:

var

Объявляет переменную, инициализация переменной значением является необязательной.

let

Объявляет локальную переменную в области видимости блока, инициализация переменной значением является необязательной.

const

Объявляет именованную константу, доступную только для чтения.

Переменные

Вы можете использовать переменные как символические имена для значений в вашем приложении. Имена переменных называются identifiers и должны соответствовать определённым правилам.

Идентификатор в JavaScript должен начинаться с буквы, нижнего подчёркивания (\_) или знака доллара ($); последующие символы могут также быть цифрами (0-9). Поскольку JavaScript чувствителен к регистру, буквы включают символы от "A" до "Z" (верхний регистр) и символы от "a" до "z" (нижний регистр).

Вы можете объявить переменную тремя способами:

Используя ключевое слово var. Например, var x = 42. Данный синтаксис может быть использован для объявления как локальных, так и глобальных переменных.

Просто присвоить переменной значение. Например, x = 42. Переменные, объявленные данным способом, являются глобальными. Такое объявление генерирует строгое предупреждение (strict mode). Не рекомендуется использовать данный способ.

Используя ключевое слово let. Например, let y = 13. Данный синтаксис может быть использован для объявления локальной переменной в области видимости блока.

Присваивание значений

Переменная, объявленная через var или let без присвоения начального значения, имеет значение undefined.

При попытке доступа к необъявленной переменной или переменной до её объявления будет выброшено исключение ReferenceError:​

Область видимости переменных

Когда вы объявляете переменную вне функции, то такая переменная называется *глобальной* переменной, т.к. доступна любому коду в текущем документе. Когда вы объявляете переменную внутри функции, то такая переменная называется *локальной* переменной, т.к. доступна только внутри данной функции.

До ECMAScript 6 в JavaScript отсутствовала область видимости блока; переменная, объявленная внутри блока, является локальной для \_функции \_(или *глобальной* области видимости), внутри которой находится данный блок

Поднятие переменных

Другим необычным свойством переменных в JavaScript является то, что можно сослаться на переменную, которая объявляется позже, и не получить при этом исключения. Эта концепция известна как поднятие (hoisting) переменных; переменные в JavaScript поднимаются в самое начало функции или выражения.

Поднятие функций

Для функций: только определения функций поднимаются наверх, но не функции, определённые через выражения.

Глобальные переменные

Глобальные переменные на самом деле являются свойствами *глобального объекта*. На веб-страницах глобальным объектом является window, поэтому вы можете устанавливать глобальные переменные и обращаться к ним, используя синтаксис window.variable

Следовательно, вы можете обращаться к глобальным переменным, объявленным в одном объекте window или frame из другого объекта window или frame, указав имя window или frame.

Константы

Вы можете создать именованную константу, доступную только для чтения, используя ключевое слово const. Синтаксис идентификатора константы является таким же, как и у идентификатора переменной: он должен начинаться с буквы, нижнего подчёркивания или знака $ и может содержать буквы, цифры или нижнее подчёркивание.

const PREFIX = '212';

Нельзя изменить значение константы через присваивание или повторное объявление во время выполнения скрипта. Значение должно быть указано при инициализации.

Правила, касающиеся области видимости, для констант являются такими же, как и для переменных, объявленных через let. Если ключевое слово const не указано, то идентификатор будет являться переменной.

Нельзя объявить константу с таким же именем, как у функции или переменной в одной области видимости

Типы данных

Последний стандарт ECMAScript определяет семь типов данных:

Шесть типов данных, которые являются примитивами:

Boolean. true и false.

null. Специальное ключевое слово, обозначающее нулевое или «пустое» значение. Поскольку JavaScript чувствителен к регистру, null не то же самое, что Null, NULL или любой другой вариант.

undefined. Свойство глобального объекта; переменная, не имеющая присвоенного значения, обладает типом undefined.

Хотя типов данных относительно немного, но они позволяют вам выполнять полезные функции в ваших приложениях. Объекты и функции являются другими фундаментальными элементами языка. Вы можете думать об объектах как об именованных контейнерах для значений и о функциях как о процедурах, которые ваше приложение может исполнять.

Преобразование типов данных

JavaScript — это динамически типизированный язык. Это означает, что вам не нужно указывать тип данных переменной, когда вы её объявляете, типы данных преобразуются автоматически по мере необходимости во время выполнения скрипта.

Поскольку JavaScript является динамически типизированным, это присваивание не вызовет сообщения об ошибке.

В выражениях, включающих числовые и строковые значения с оператором +, JavaScript преобразует числовые значения в строковые.

Литералы

Литералы используются для представления значений в JavaScript. Они являются фиксированными значениями, а не переменными.

Литерал массива — это список из нуля или более выражений, каждое из которых представляет элемент массива, заключённый в квадратные скобки ( [] ). Когда вы создаёте массив, используя литерал массива, он инициализируется с помощью переданных значений, которые будут являться его элементами, длина массива будет равна числу переданных аргументов.

Если массив создаётся с помощью литерала в скрипте верхнего уровня, то JavaScript интерпретирует массив каждый раз, когда вычисляет выражение, содержащее литерал. Кроме того, литерал, используемый в функции, создаётся каждый раз, когда вызывается функция.

Литералы массива также являются объектами Array. Чтобы получить более подробную информацию, прочитайте Array и упорядоченные наборы данных.

Если вы поставите запятую в конце списка элементов, то она будет проигнорирована.

Логические литералы

Логический (Boolean) тип имеет два литеральных значения: true и false.

Не путайте примитивные логические значения true и false со значениями true и false объекта Boolean. Объект Boolean является объектом-обёрткой над примитивом логического типа. Чтобы получить более подробную информацию, прочитайте Boolean.

Литерал целого числа

Целые числа могут быть записаны в десятичной, шестнадцатеричной, восьмеричной и двоичной системах счисления.

Десятичный целочисленный литерал состоит из последовательности цифр без ведущего нуля.

Ведущий ноль в целочисленном литерале указывает на то, что он записан в восьмеричной системе счисления. Восьмеричные целые числа состоят только из цифр 0-7.

Ведущие символы 0x (или 0X) указывают на то, что число шестнадцатеричное. Шестнадцатеричные целые числа могут состоять из цифр 0-9 и букв a-f и A-F.

Ведущие символы 0b (или 0B) указывают на то, что число двоичное. Двоичные числа могут включать в себя только цифры 0 и 1.

Литерал числа с плавающей точкой

Числа с плавающей точкой могут состоять из следующих частей:

Десятичное целое число, которое может иметь знак (символ "+" или "-", стоящий перед числом),

Десятичная точка ("."),

Дробная часть (другое десятичное число),

Экспонента.

Экспонента состоит из символа "e" или "E", за которым следует целое число, которое может иметь знак. Число с плавающей точкой должно состоять по крайней мере из одной цифры и либо десятичной точки, либо символа "e" (или "E").

Литерал объекта

Литерал объекта — это список из нуля или более пар, состоящих из имён свойств и связанных с ними значений, заключённый в фигурные скобки ( {} ). Вам не следует использовать литерал объекта в начале выражения, т.к. это приведёт к ошибке или к поведению, которого вы не ожидаете, потому что символ "{" будет интерпретироваться как начало блока.

Кроме того, вы можете использовать числовой или строковой литералы в именах свойств или вкладывать один объект в другой.

Именем свойства объекта может быть любая строка, в том числе пустая строка. Если имя свойства не является корректным JavaScript идентификатором, то оно должно быть заключено в кавычки. Для обращения к таким именам следует использовать квадратные скобки ( [] ), а не точку ( .

RegExp литерал

Литерал regexp - шаблон между слешами. Следующий пример литерал regex:

var re = /ab+c/;

Строковый литерал

Строковый литерал — это ноль или более символов, заключённых в двойные ( " ) или одинарные ( ' ) кавычки. Строка должна быть ограничена кавычками одного типа, т.е. либо обе одинарные, либо обе двойные.

Вы можете вызвать любой из методов объекта String для строкового литерала: JavaScript автоматически преобразует строковой литерал во временный объект String, вызовет метод, а затем уничтожит временный объект String.

Вам следует использовать строковые литералы до тех пор, пока вам специально не понадобится объект String. Чтобы получить более подробную информацию об объекте String, прочитайте String.

Использование специальных символов в строках

Кроме обычных символов вы также можете включать специальные символы в строки.

**Сценарии JavaScript. Способы внедрения сценариев JavaScript в веб-документ.**

JavaScript - это стандартный язык программирования, который может быть включён в веб-странички, чтобы обеспечить работу таких возможностей как меню, звуки и другие интерактивные элементы. Самый простой способ внедрения JavaScript в HTML-документ – использование тега <script>. Теги <script> часто помещают в элемент <head>, и ранее этот способ считался чуть ли не обязательным. Однако в наши дни теги <script> используются как в элементе <head>, так и в теле веб-страниц.

Таким образом, на одной веб-странице могут располагаться сразу несколько сценариев. В какой последовательности браузер будет выполнять эти сценарии? Как правило, выполнение сценариев браузерами происходит по мере их загрузки. Браузер читает HTML-документ сверху вниз и, когда он встречает тег <script>, рассматривает текст программы как сценарий и выполняет его. Остальной контент страницы не загружается и не отображается, пока не будет выполнен весь код в элементе <script>.

JavaScript в атрибутах событий HTML-элементов

В следующем примере функция JavaScript помещается в раздел <head> HTML-документа. Вот пример HTML-элемента <button> с атрибутом события, обеспечивающим реакцию на щелчки мышью. При нажатии кнопки генерируется событие onclick.

Внешний JavaScript

Если JavaScript-кода много – его выносят в отдельный файл, который, как правило, имеет расширение .js.

Чтобы включить в HTML-документ JavaScript-кoд из внешнего файла, нужно использовать атрибут src (source) тега <script>.

В этом примере указан абсолютный путь к файлу с именем script.js, содержащему скрипт (из корня сайта). Сам файл должен содержать только JavaScript-кoд, который иначе располагался бы между тегами <script> и </script>.

Чтобы подключить несколько скриптов, используйте несколько тегов

Независимо от того, как JS-код включается в НТМL-документ, элементы <script> интерпретируются браузером в том порядке, в котором они расположены в HTML-документе. Сначала интерпретируется код первого элемента <script>, затем браузер приступает ко второму элементу <script> и т. д.

Внешние скрипты практичны, когда один и тот же код используется во многих разных веб-страницах. Браузер скачает js-файл один раз и в дальнейшем будет брать его из своего кеша, благодаря чему один и тот же скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера. Кроме этого, благодаря внешним скриптам, упрощается сопровождение кода, поскольку вносить изменения или исправлять ошибки приходится только в одном месте.

Расположение тегов <script>

Вы уже знаете, что браузер читает HTML-документ сверху вниз и, начинает отображать страницу, показывая часть документа до тега <script>. Встретив тег <script>, переключается в JavaScript-режим и выполняет сценарий. Закончив выполнение, возвращается обратно в HTML-режим и отображает оставшуюся часть документа.

Если на странице используется много скриптов JavaScript, то могут возникнуть длительные задержки при загрузке, в течение которых пользователь видит пустое окно браузера.

Такое расположение сценариев позволяет браузеру загружать страницу быстрее, так как сначала загрузится контент страницы, а потом будет загружаться код сценария.  
 Для пользователей это предпочтительнее, потому что страница полностью визуализируется в браузере до обработки JavaScript-кoдa.

Отложенные и асинхронные сценарии

Как отмечалось ранее, по умолчанию файлы JavaScript-кода прерывают синтаксический анализ (парсинг) HTML-документа до тех пор, пока скрипт не будет загружен и выполнен, тем самым увеличивая промежуток времени до первой отрисовки страницы.  
Пока браузер не загрузит и не выполнит script.js, он не покажет часть страницы под ним. Такое поведение браузера называется «синхронным» и может доставить проблемы, если мы загружаем несколько JavaScript-файлов на странице, так как это увеличивает время её отрисовки.

Атрибут async

Async используется для того, чтобы указать браузеру, что скрипт может быть выполнен «асинхронно».

При обнаружении тега <script async src="..."> браузер не останавливает обработку HTML-документа для загрузки и выполнения скрипта, выполнение может произойти после того, как скрипт будет получен параллельно с разбором документа. Когда скрипт будет загружен – он выполнится.

Для сценариев с атрибутом async не гарантируется вы­полнение скриптов в порядке их добавления

Атрибут defer

Атрибут defer откладывает выполнение скрипта до тех пор, пока вся HTML-страница не будет загружена полностью.

Как и при асинхронной загрузке скриптов — JS-файл может быть загружен, в то время как HTML-документ ещё грузится. Однако, даже если скрипт будет полностью загружен ещё до того, как браузер закончит обработку страницы, он не будет выполнен до тех пор, пока HTML-документ не обработается до конца.

Несмотря на то, что в приведенном примере теги <script defer src="..."> включены в элемент <head> HTML-документа, выполнение сценариев не начнется, пока браузер не дойдет до закрыва­ющего тега </html>.

Кроме того, в отличие от async, относительный порядок выполнения скриптов с атрибутом defer будет сохранён.

Применение атрибута defer бывает полезным, когда в коде скрипта предусматривается работа с HTML-документом, и разработчик должен быть уверен, что страница полностью получена.

Операторы. Типы операторов языка JavaScript: операторы сравнения, арифметические операторы, строковый оператор. Приоритет операторов.

Операторы

В JavaScript есть следующие типы операторов. Данный подраздел описывает каждый тип и содержит информацию об их приоритетах друг над другом.

JavaScript поддерживает бинарные и унарные операторы, а также ещё один специальный тернарный оператор - условный оператор. Бинарная операция использует два операнда, один перед оператором и другой за ним:

operand1 operator operand2

Операторы присваивания

В результате операции присваивания операнду слева от оператора присваивания (en-US) (знак "=") устанавливается значение , которое берётся из правого операнда. Основным оператором присваивания является =, он присваивает значение правого операнда операнду, находящемуся слева. Таким образом, выражение x = y означает, что x присваивается значение y.

Операторы сравнения

Оператор сравнения (en-US) сравнивает свои операнды и возвращает логическое значение, базируясь на истинности сравнения. Операнды могут быть числами, строками, логическими величинами или объектами. Строки сравниваются на основании стандартного лексикографического порядка, используя Unicode-значения.

Арифметические операторы

Арифметические операторы (en-US) используют в качестве своих операндов числа (также литералы или переменные) и в качестве результата возвращают одно числовое значение. Стандартными арифметическими операторами являются сложение (+), вычитание (-), умножение (\*), и деление (/). При работе с числами с плавающей точкой эти операторы работают аналогично их работе в большинстве других языках программирования (обратите внимание, что деление на ноль возвращает бесконечность Infinity).

Битовые (поразрядные) операторы

Битовые операторы (en-US) обрабатывают свои операнды как последовательности из 32 бит (нулей и единиц), а не как десятичные, шестнадцатеричные или восьмеричные числа. Например, десятичное число 9 имеет двоичное представление 1001. Битовые операторы выполняют операции над таким двоичным представлением, но результат возвращают как обычное числовое значение JavaScript.

Битовые логические операторы

Битовые операторы сдвига

Битовые операторы сдвига используют два операнда: первый представляет величину, подлежащую сдвигу, а второй операнд указывает число битовых позиций на которое должен быть сдвинут первый операнд. Направление операции сдвига определяется используемым оператором.

Операторы сдвига преобразуют свои операнды в 32-битные целые числа и возвращают результат того же типа, каким является левый операнд.

Логические операторы

Логические операторы (en-US) обычно используются с булевыми (логическими) значениями; при этом возвращаемое ими значение также является булевым. Однако операторы && и || фактически возвращают значение одного из операндов, поэтому, если эти операторы используются с небулевыми величинами, то возвращаемая ими величина также может быть не булевой.

Сокращённая оценка

Так как логические выражения вычисляются слева направо, они проверяются на возможность выполнения сокращённой оценки с использованием следующих правил:

false && anything *-* сокращение с результатом false.

true || *anything -* сокращение с результатом true.

Правила логики гарантируют, что данные вычисления всегда корректны. Обратите внимание, что часть "*anything"* представленных выше выражений не вычисляется, таким образом удаётся избежать любых побочных эффектов вычисления данной части.

Строковые операторы

В дополнение к операторам сравнения, которые могут использоваться со строковыми значениями, оператор (+) позволяет объединить две строки, возвращая при этом третью строку, которая представляет собой объединение двух строк-операндов:

Условный (тернарный) оператор

Условный оператор является единственным оператором JavaScript, который использует три операнда. Оператор принимает одно из двух значений в зависимости от заданного условия. Синтаксис оператора:

condition ? val1 : val2

Если condition (условие) - истина, то оператор принимает значение val1. В противном случае оператор принимает значение val2. Вы можете использовать условный оператор во всех случаях, где может быть использован стандартный оператор.

Оператор запятая

Оператор запятая (,) просто вычисляет оба операнда и возвращает значение последнего операнда. Данный оператор в основном используется внутри цикла for, что позволяет при каждом прохождении цикла одновременно обновлять значения нескольких переменных.

Унарные операторы

Унарная операция - операция только с одним операндом.

delete

Оператор delete выполняет удаление объекта, свойства объекта, или элемента массива с заданным индексом.

Вы можете использовать оператор delete для удаления переменных, объявленных неявно, но вы не можете с его помощью удалять переменные, объявленные с помощью var.

После применения оператора delete свойство элемента меняется на undefined. Оператор delete возвращает true если выполнение операции возможно; оператор возвращает false, если выполнение операции невозможно.

Удаление элементов массива

Удаление элемента массива не влияет на длину массива. Например, если вы удалите a[3], элемент a[4] останется a[4],a[3] станет undefined.

Когда элемент массива удаляется с помощью оператора delete, то из массива удаляется значение данного элемента. В следующем примере элемент trees[3] удалён с помощью оператора delete. Однако, элемент trees[3] остаётся адресуемым и возвращает значение undefined.

Если вы хотите, чтобы элемент оставался в массиве, но имел значение undefined, то используйте ключевое слово undefined вместо оператора delete.

Оператор typeof

Оператор typeof используется одним из следующих способов:

typeof operand

typeof (operand)

Оператор typeof возвращает строку обозначающую тип невычисленного операнда. Значение operand может быть строкой, переменной, дескриптором, или объектом, тип которого следует определить. Скобки вокруг операнда необязательны.

Оператор void

Оператор void используется любым из следующих способов:

void (expression)

void expression

Оператор void определяет выражение, которое должно быть вычислено без возвращения результата. expression - это выражение JavaScript, требующее вычисления. Скобки вокруг выражения необязательны, но их использование является правилом хорошего тона.

Операторы отношения

Оператор отношения сравнивает свои операнды и возвращает результат сравнения в виде булева значения.

Оператор in

Оператор in возвращает true, если указанный объект имеет указанное свойство. Синтаксис оператора:

propNameOrNumber in objectName

где propNameOrNumber - строка или числовое выражение, представляющее имя свойства или индекс массива, а objectName - имя объекта.

Некоторые примеры способов использования оператора in:

Оператор instanceof

Оператор instanceof возвращает true, если заданный объект является объектом указанного типа. Его синтаксис:

objectName instanceof objectType

где objectName - имя объекта, тип которого необходимо сравнить с objectType, а objectType - тип объекта, например, Date или Array.

Используйте оператор instanceof, когда вам необходимо подтвердить тип объекта во время выполнения программы. Например, при перехвате исключений вы можете создать различные программные переходы для обработки исключений в зависимости от типа обрабатываемого исключения.

Приоритет операторов

*Приоритет* операторов определяет порядок их выполнения при вычислении выражения. Вы можете влиять на приоритет операторов с помощью скобок.

Выражения

*Выражением* является любой корректный блок кода, который возвращает значение.

Концептуально, существуют два типа выражений: те которые присваивают переменной значение, и те, которые вычисляют значение без его присваивания.

Все выражения в JavaScript делятся на следующие категории:

Арифметические: вычисляются в число, например: 3.14159 (Используют арифметические операторы).

Строковые: вычисляются в текстовую строку, например: "Fred" или "234" (Используют строковые операторы).

Логические: вычисляются в true или false (Используют логические операторы).

Основные выражения: Базовые ключевые слова и основные выражения в JavaScript.

Левосторонние выражения: Значениям слева назначаются значения справа.

Основные выражения

Базовые ключевые слова и основные выражения в JavaScript.

Оператор this

Используйте ключевое слово this для указания на текущий объект. В общем случае this указывает на вызываемый объект, которому принадлежит данный метод. Используйте this следующим образом:

this["propertyName"]

this.propertyName

Оператор группировки

Оператор группировки "скобки" ( ) контролирует приоритет вычисления выражений. Например, вы можете переопределить порядок - "умножение и деление, а потом сложение и вычитание", так чтобы, например, чтобы сложение выполнялось до умножения:

Левосторонние выражения

Значениям слева назначаются значения справа.

new

Вы можете использовать оператор new для создания экземпляра объекта пользовательского типа или одного из встроенных объектов. Используйте оператор new следующим образом:

var objectName = new objectType([param1, param2, ..., paramN]);

super

Ключевое слово используется, чтобы вызывать функции родительского объекта. Это полезно и с классами для вызова конструктора родителя,

Оператор расширения

Оператор расширения позволяет выражению расширяться в местах с множеством аргументов (для вызовов функций) или множестве элементов (для массивов).

Выражения. Типы выражений.Правила составления выражений.

Выражения в JavaScript представляют собой комбинации *операндов* и *операторов*.

*Операции* в выражениях выполняются последовательно в соответствии со значением приоритета (чем больше значение приоритета, тем он выше). Возвращаемый результат не всегда имеет значение того же типа, что и тип обрабатываемых данных. Например, в операциях сравнения участвуют операнды различных типов, но возвращаемый результат всегда будет логического типа.

Операнды — это данные, обрабатываемые сценарием JavaScript. В качестве операндов могут быть как простые типы данных, так и сложные, а также другие выражения.

Операторы — это символы языка, выполняющие различные операции с данными. Операторы могут записываться с помощью символов пунктуации или ключевых слов.

В зависимости от количества операндов различают следующие типы операторов:  
унарный — в операции участвует один операнд;  
бинарный — в операции участвуют два операнда;  
тернарный — комбинирует три операнда.

Простейшая форма выражения — литерал — нечто, вычисляемое само в себя, например, число 100, строка "Hellow world". Переменная тоже может быть выражением, так как она вычисляется в присвоенное ей значение.

Выражения и операторы в JavaScript

Арифметические операторы предназначены для выполнения математических операций, они работают с числовыми операндами (или переменными, хранящими числовые значения), возвращая в качестве результата числовое значение.

Если один из операндов является строкой, интерпретатор JavaScript попытается преобразовать его в числовой тип, а после выполнить соответствующую операцию. Если преобразование типов окажется невозможным, будет получен результат NaN (не число).

2. Операторы присваивания

Операторы присваивания используются для присваивания значений переменным. Комбинированные операторы позволяют сохранить первоначальное и последующее значение в одной переменной.

3. Операторы инкремента и декремента

Операции инкремента и декремента являются унарными и производят увеличение и уменьшение значения операнда на единицу. В качестве операнда может быть переменная, элемент массива, свойство объекта. Чаще всего такие операции используются для увеличения счетчика в цикле.

4. Операторы сравнения

Операторы сравнения используются для сопоставления операндов, результатом выражения может быть одно из двух значений — true или false. Операндами могут быть не только числа, но и строки, логические значения и объекты. Однако сравнение может выполняться только для чисел и строк, поэтому операнды, не являющиеся числами или строками, преобразуются.

Если оба операнда не могут быть успешно преобразованы в числа или строки, операторы всегда возвращают false.

Если оба операнда являются строками/числами или могут быть преобразованы в строки/числа, они будут сравниваться как строки/числа.

Если один операнд является строкой/преобразуется в строку, а другой является числом/преобразуется в число, то оператор попытается преобразовать строку в число и выполнить сравнение чисел. Если строка не является числом, она преобразуется в значение NaN и результатом сравнения будет false.

Чаще всего операции сравнения используются при организации ветвлений в программах.

|  |  |  |
| --- | --- | --- |
| Оператор/Операция | Описание | Приоритет |
| == Равенство | Проверяет две величины на совпадение, допуская преобразование типов. Возвращает true, если операнды совпадают, и false, если они различны. | 9 |
| != Неравенство | Возвращает true, если операнды не равны | 9 |
| === Идентичность | Проверяет два операнда на «идентичность», руководствуясь строгим определением совпадения. Возвращает true, если операнды равны без преобразования типов. | 9 |
| !== Неидентичность | Выполняет проверку идентичности. Возвращает true, если операнды не равны без преобразования типов. | 9 |
| > Больше | Возвращает true, если первый операнд больше второго, в противном случае возвращает false. | 10 |
| >= Больше или равно | Возвращает true, если первый операнд не меньше второго, в противном случае возвращает false. | 10 |
| < Меньше | Возвращает true, если первый операнд меньше второго, в противном случае возвращает false. | 10 |
| <= Меньше или равно | Возвращает true, если первый операнд не больше второго, в противном случае возвращает false. | 10 |

5. Логические операторы

Логические операторы позволяют комбинировать условия, возвращающие логические величины. Чаще всего используются в условном выражении if.

|  |  |  |
| --- | --- | --- |
| Оператор/Операция | Описание | Приоритет |
| && Логическое И | Возвращает true, только если оба операнда истинны. При выполнении операции сначала проверяется значение первого операнда. Если оно имеет значение false, то значение второго оператора не учитывается и результату выражения присваивается false. | 5 |
| || Логическое ИЛИ | Возвращает true, если хотя бы один операнд истинен, т.е. проверяет истинность как минимум одного условия. | 4 |
| ! Логическое НЕ | Изменяет значение оператора на обратное - с true на false и наоборот. | 14 |

6. Побитовые операторы

Побитовые операторы работают с операндами как с 32-битной последовательностью нулей и единиц и возвращают числовое значение, означающее результат операции, записанное в десятичной системе счисления. В качестве операндов рассматриваются целые числа, дробная часть операнда отбрасывается. Побитовые операции могут использоваться, например, при шифровании данных, для работы с флагами, разграничения прав доступа.

9. Комментарии в JavaScript

Однострочный комментарий: перед текстом комментария нужно поставить символы //.

Многострочный комментарий помещается между символами /\* и \*/.

Инструкции в языке программирования JavaScript.

Приложения на JavaScript состоят из инструкций с соответствующим синтаксисом. Одна инструкция может состоять из нескольких строк. На одной строке может находиться несколько инструкций, если они разделены точкой с запятой. Они являются не ключевыми словами, а группами ключевых слов.

Блок

Блок используется для группировки нуля и более инструкций. Блок отделяется парой фигурных скобок.

break

Прерывает текущую инструкцию цикла, ветвления или инструкцию с меткой и передаёт управление на инструкцию, следующую за прерываемой.

continue

Прерывает выполнение инструкции в текущей итерации текущего цикла или цикла с меткой и продолжает выполнение цикла со следующей итерации.

Пустая инструкция

Пустая инструкция используется тогда, когда никакой инструкции быть не должно, а по синтаксису JavaScript она требуется.

if...else

Выполняет инструкцию, если указанное условие является истинным. Если условие ложно, выполняет другую инструкцию.

switch

Вычисляет выражение, сопоставляет вычисленное значение с выражением выбора и выполняет инструкции, ассоциированные с этим выбором.

throw

Выбрасывает пользовательское исключение.

try...catch

Помечает блок инструкций и определяет реакцию на возникновение исключения внутри помеченного блока инструкций.

Объявления

var

Объявляет переменную, необязательно инициализирует её значением.

Экспериментальная возможностьlet

Объявляет локальную переменную в области видимости блока, необязательно инициализирует её значением.

Экспериментальная возможностьconst

Объявляет именованную константу только для чтения.

Функции

function

Объявляет функцию с указанными параметрами.

Экспериментальная возможностьfunction\*

Функции-генераторы, упрощающие написание итераторов.

return

Определяет значение, возвращаемое из функции.

Итерации

do...while

Создаёт цикл, выполняющий указанную инструкцию до тех пор, пока проверяющее условие равно false. Условие вычисляется после выполнения инструкции, так что указанная инструкция выполнится хотя бы один раз.

for

Создаёт цикл, состоящий из трёх необязательных выражений, заключённых в круглые скобки и разделённым точками с запятой с последующей инструкцией, выполняющейся в цикле.

DeprecatedNon-standardfor each...in

Проходит указанной переменной по всем значениям свойств объекта. Выполняет инструкции для каждого уникального свойства.

for...in

В случайном порядке проходит по перечислимым свойствам объекта. Выполняет инструкции для каждого уникального свойства.

Экспериментальная возможностьfor...of

Проходит по итерируемым объектам (включающим в себя массивы, массивоподобные объекты и итераторы с генераторами (en-US)), выполняя пользовательский хук с инструкциями, исполняющимися для значения каждого уникального свойства.

while

Создаёт цикл, выполняющий указанную инструкцию до тех пор, пока проверяющее условие равно true. Условие вычисляется перед выполнением инструкции..

Иногда нам нужно выполнить различные действия в зависимости от условий.

Для этого мы можем использовать инструкцию if и условный оператор ?, который также называют оператором «вопросительный знак».

Инструкция «if»

Инструкция if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода.

В примере выше, условие – это простая проверка на равенство (year == 2015), но оно может быть и гораздо более сложным.

Инструкция if (…) вычисляет выражение в скобках и преобразует результат к логическому типу.

Мы также можем передать заранее вычисленное в переменной логическое значение в if, например так:

Блок «else»

Инструкция if может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.

Несколько условий: «else if»

Иногда нужно проверить несколько вариантов условия. Для этого используется блок else if.

Условный оператор „?“

Иногда нам нужно определить переменную в зависимости от условия.

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.

Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

Синтаксис:

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

Технически, мы можем опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >.

Несколько операторов „?“

Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия.

Смысл оператора «вопросительный знак» ? – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его именно для этого. Когда вам нужно выполнить разные ветви кода – используйте if.

Оператор – это просто внутренняя функция JavaScript. При использовании того или иного оператора мы, по сути, просто запускаем ту или иную встроенную функцию, которая выполняет какие-то определённые действия и возвращает результат.

Унарные и бинарные операторы

Каждый оператор оперирует определённым количеством операндов. Операнд – это то, что находится слева и (или) справа от оператора. Иногда операнды называют ещё аргументами.

Операторы по количеству используемых операндов делятся на унарные и бинарные.

У унарных операторов всегда один операнд (аргумент).

Нотации записей операторов

В зависимости от того, в каком стиле записан оператор относительно операнда(ов) он имеет инфиксную, префиксную или постфиксную запись.

Инфиксную форму записи имеют операторы, которые находятся между операндами. Т.е. это все бинарные операторы:

Операторы присваивания

Оператор присваивания обозначается с помощью знака =. Он используется, когда вам нужно присвоить той или иной переменной какое-то значение.

let a; a = 7;

В данном примере на второй строчке мы переменной a, которая была объявлена ранее, присваиваем значение 7.

пытается найти переменную a; если переменная не найдена выбрасывает ошибку;

присваивает переменной a результат выражения 7;

возвращает в качестве результата выполнения функции = результат выражения 7.

В JavaScript имеются ещё составные операторы присваивания: +=, -=, \*=, /=, %=, \*\*=, <<=, >>=, >>>=, &=, ^=, |=.

Данные операторы перед присваиванием, выполняют ещё одно дополнительное действие.

Арифметические операторы

В JavaScript выделяют следующие математические операторы:

унарный плюс + и минус -

;

сложение +;

вычитание -;

умножение \*;

деление /;

остаток от деления %;

возведение в степень \*\*;

увеличение значения переменной на 1 ++;

уменьшение значения переменной на 1 --.

JavaScript – это язык с динамическим приведением типов. Во втором примере оператор минус (–) является математическим и может оперировать только с числами. Для выполнения этой операции JavaScript приведёт значения первого и второго операнда к числам (5 и 15), а затем уже выполнит вычитание.

Оператор плюс (+) не только математический, он также используется для конкатенации (соединения) строк.

Выбор операции при использовании оператора плюс (+) зависит от типа значений операндов. Если хотя бы один из операндов является строкой, то данная операция рассматривается как операция конкатенации строк. В противном случае она будет считаться математической.

В этом примере второй операнд является строкой, следовательно, данная операция будет операцией конкатенацией строк. Т.к. первый операндов не является строкой, то JavaScript приведёт его к ней. В результате будет возвращена строка «5px».

Унарный оператор плюс (+) в основном используется для явного преобразования не числовых типов данных к числам.

Оператор минус (–), также как и оператор плюс (+) может применяться к одному операнду. Если операнд является числом, то он просто изменит его знак на противоположный. В противном случае, он преобразует значение операнда к числу, а затем изменит его знак на противоположный.

Операторы инкремента (++) и декремента (--) имеют префиксную и постфиксную форму.

Оператор процент (%) вычисляет остаток от деления первого операнда на второй:

100 % 3; // 1

Как вычисляется остаток от деления показано на картинке:

Операторы сравнения

Операторы сравнения предназначены для сравнения двух операндов. Если сравнение верное, то в качестве результата этой операции возвращается true.

При сравнении маленькие и большие буквы алфавита не равны, т.к. имеют разные коды в таблице Unicode. Поэтому при операциях сравнения строки желательно приводить к одному регистру.

Во всех операциях, кроме операций с использованием строго равенства и строго не равенства, JavaScript перед операцией сравнения приводит их к одному типу. Данное действие он конечно же выполняет только в том случае, если операнды имеют разный тип данных.

В большинстве случаев при выполнении операций равенства или не равенства, их желательно выполнять с использованием операторов === и !==. Применение данных операторов позволит исключить неожиданные ситуации при сравнении операндов с различным типом данных.

Строковые операторы

В JavaScript оператор + можно использовать также для конкатенации или, другими словами, объединения строк:

В JavaScript создавать строковые значения можно также с помощью шаблонной строки (template string literal). В отличие от строк, задаваемых с помощью одинарных или двойных кавычек, этот механизм имеет множество мощных возможностей.

Одна из возможностей – это использовать JavaScript выражения внутри строк.

Обратите внимание, что шаблонная строка заключается в обратные кавычки. Внутри неё можно помещать различные выражения JavaScript, заключая их в ${}.

Здесь между выражениями стоит пробел. Он необходим, чтобы у нас получилась точно такая же строка как в предыдущем примере.

Текстовые операторы

Текстовые операторы – это такие, которые записываются не с помощью значков, а посредством символов. Например:

typeof – позволяет узнать тип того или иного значения;

instanceOf – можно проверить принадлежность объекта к тому или иному классу;

delete – удаляет то или иное свойство объекта.

Приоритет операторов

Операторов в JavaScript как вы уже знаете очень много. У всех них имеется определённый приоритет. Без него не обойтись, когда выражение строится с использованием нескольких операторов. В этом случае их приоритет будет определять порядок, в соответствии с которым они выполняются.

Список операторов, упорядоченных по приоритету (от более высокого к низкому, через тире указана их специфичность):

Группировка () – не определено;

Доступ к свойствам ., [], вызов функции (), ?. – слева направо, создание экземпляра объекта со списком аргументов new – не определено;

создание экземпляра объекта без аргументов new – справа налево;

Постфиксный инкремент ++ и декремент -- – не определено;

Логическое ! и побитовое отрицание ~, унарный плюс + и минус -, префиксный инкремент ++ и декремент --, typeof, void, delete, await – справа налево;

В этом выражении операторы \* и % имеют одинаковый приоритет, поэтому порядок выполнения этого выражения будет определяться в соответствии с ассоциативностью. Т.к. оператор \* имеет левую ассоциативность (слева направо), то сначала будет выполнено 20 \* 3, а потом остаток от деления. Другими словами, это выражение будет обрабатываться так:

(20 \* 3) % 2 // 0

Циклы в JavaScript. Операторы организации циклов.

Назначение и виды циклов

Циклы – это простой способ для многократного выполнения одних и тех же действий (кода).

При этом однократное выполнения кода в цикле называется итерацией.

В JavaScript существуют различные виды циклов, но все они, по сути, делают одно и тоже. Просто с помощью одних циклов более просто решаются одни задачи, с помощью других – иные:

for;

while;

do...while;

for...in;

for...of (появился в версии ES6).

Цикл for

Данный цикл в основном используется когда известно точное количество повторений. Этот цикл ещё называют циклом со счётчиком.

Синтаксис цикла «for»:

for (инициализация; условие; финальное выражение) { /\* тело цикла \*/ }

Основные части конструкции цикла «for»:

инициализация - это выражение, которое выполняется один раз перед выполнением цикла; обычно используется для инициализации счётчика;

условие - это выражение, истинность которого проверяется перед каждой итерацией; если выражение вычисляется как истина, то выполняется итерация; в противном случае цикл «for» завершает работу;

финальное выражение - это выражение, которое выполняется в конце каждой итерации; обычно используется для изменения счетчика;

тело цикла - инструкции, выполнение которых нужно повторять.

При этом если тело цикла состоит из одной инструкции, то её можно не заключать в фигурные скобки.

Необязательные части цикла цикла «for».

В «for» все части цикла являются не обязательными.

В этом случае инициализацию переменной можно вынести за пределы цикла.

Условие в «for» тоже является не обязательным. Без условия цикл будет выполняться бесконечное количество раз. В этом случае чтобы его прервать (выйти из цикла) необходимо использовать инструкцию break.

В «for» можно вообще опустить 3 выражения (бесконечный цикл):

var i = 1; // цикл «for»for (;;) { if (i >= 8) { break; } console.log(i); i++; }

Кроме этого, в качестве тела цикла «for» можно использовать пустое выражение (;). Это используется, когда вам не нужно выполнять ни одной инструкции.

Цикл называется вложенным, если он находится в теле другого цикла.

Цикл while

Данный цикл предназначен для многократного выполнения одних и тех же инструкций до тех пор, пока истинно некоторое условие. Цикл «while» в основном используется, когда количество повторений заранее не известно.

while (условие) { /\* тело цикла \*/ }

Истинность условия проверяется перед каждым выполнением. Если перед первой итерацией условие ложно, то цикл не выполнится ни разу.

Цикл do...while

Цикл «do...while», также как и цикл «while», выполняет одни и те же инструкции до тех пор, пока указанное условие истинно. Но в отличие от «while» в «do...while» условие проверяется после выполнения инструкций. Поэтому цикл «do...while» в любом случае выполнится не меньше одного раза, даже если условие изначально ложно.

do { /\* тело цикла \*/ } while (условие)

Цикл for...in

Цикл «for...in» предназначен для перебора перечисляемых имён свойств объекта. В JavaScript свойство является перечисляемым, если его внутренний флаг [[Enumerable]] равен true.

Свойства объекта, которые не относятся к перечисляемым, в цикле не участвуют.

Кроме этого, следует отметить, что цикл for...in проходит не только по перечисляемых свойствам этого объекта, но и по наследуемым.

Использование цикла for... in для перебора массива. В массиве свойствами являются числовые индексы.

Цикл for...in проходит по свойствам в произвольном порядке. Поэтому если при переборе массива для вас важен порядок символов, то данный цикл лучше не использовать.

При использовании цикла for…in стоит обратить внимание на то, что если вы к массиву добавили свои пользовательские свойства, то он по ним тоже пройдётся:

Если вам такой сценарий не нужен, то тогда для перебора массивов лучше использовать обычный цикл for.

Инструкции break и continue

Внутри тела цикла можно использовать специальные инструкции: break и continue.

Инструкция «break» предназначена для прекращения выполнения текущего цикла. Другими словами, она осуществляет выход и передачу управления инструкции, идущей после этого цикла.

Инструкция «continue» предназначена для прекращения дальнейшего выполнения кода и перехода к следующей итерации цикла.

Метки для break и continue

Если метка используется с ключевым словом continue, то в этом случае выполнение этого действия приведёт к немедленному переходу к следующей итерации цикла, перед которым данная метка указана.

В коде с одиночным циклом использование метки не даст никакого результата. Её есть смысл использовать только когда вам нужно выйти сразу из нескольких циклов.

Кроме этого, операторы break и continue нельзя использовать в выражениях тернарных операторов.

Чем цикл for...of отличается от for...in

Первое отличие цикла for...of от for...in заключается в том, что он может применяться только для итерируемым объектов, т.е. объектов, в которых реализован итератор (Symbol.iterator). Цикл for...in итератор не использует. Он предназначен для перебора любых объектов.

Второе отличие заключается в том, что цикл for...of перебирает объект так, как это определено в итераторе. Например, в Array итератор реализован так, что цикл for...of пройдёт только по значениям в массиве и не будет включать в перебор другие (не индексные) свойства. Цикл for...in организован по-другому, он перебирает все перечисляемые свойства (имена ключей) объекта, в том числе и наследуемые.

Рассмотрим эти отличия. Для этого возьмём предыдущий пример и добавим к нему пользовательское свойство, например, hero и установим ему значение

При использовании for...of он переберёт все значения этого массива:

При использовании for...in он переберёт все перечисляемые имена ключей этого объекта:

Чтобы получить значение ключа по его имени можно воспользоваться квадратными скобками:

Самостоятельное создание итератора для объекта

Рассмотрим ещё один пример. В этом примере мы самостоятельно определим как должен итерироваться объект. Для этого создадим объект и определим ему итератор.

Создание итератора начинается с добавления к объекту специального метода. Этот метод необходимо спроектировать так, чтобы он возвращал значения последовательно (одно за другим). Название методу согласно стандарту необходимо определить с помощью символа Symbol.iterator. Итератор должен возвращать всего один метод next(). Этот метод в свою очередь тоже должен возвращать объект, состоящий из 2 свойств: value и done. Ключ done - булевый. Он определяет есть ли ещё значения в последовательности (false - да, true - нет). Ключ value должен содержать следующее значение последовательности.

Циклические алгоритмы с заданным числом повторений: вычисление различных величин, сумм, произведений.

Циклические алгоритмы

Часто при решении задач приходится многократно выполнять одни и те же действия при различных значениях входящих в них величин. Такие многократно повторяющиеся участки вычислительного процесса называются циклами.

Цикл, содержащий внутри себя один или несколько других циклов, называется вложенным. Цикл, охватывающий другие циклы, называется внешним, а остальные - внутренними. Правила организации для внешних и внутренних циклов такие же, как и для простого цикла. Параметры этих циклов изменяются не одновременно: при одном значении параметра внешнего цикла параметр внутреннего цикла принимает по очереди все свои значения. В качестве параметров для этих циклов должны использоваться переменные с разными именами.

Циклы JavaScript обеспечивают многократное выполнение повторяющихся вычислений. Они оптимизируют процесс написания кода, выполняя одну и ту же инструкцию или блок инструкций, образующих тело цикла, заданное число раз (используя переменную-счётчик) или пока заданное условие истинно. Циклы выполняют обход последовательности значений. Однократное выполнение цикла называется итерацией.

На производительность цикла влияют количество итераций и количество операций, выполняемых в теле цикла каждой итерации.

Цикл for используется для выполнения итераций по элементам массивов или объектов, напоминающих массивы, таких как arguments и HTMLCollection. Условие проверяется перед каждой итерацией цикла. В случае успешной проверки выполняется код внутри цикла, в противном случае код внутри цикла не выполняется и программа продолжает работу с первой строки, следующей непосредственно после цикла.

1.1. Как работает цикл for

Цикл for состоит из трёх разных операций:

Шаг 1. инициализацияvar i = 0; — объявление переменной-счётчика, которая будет проверяться во время выполнения цикла. Эта переменная инициализируется со значением 0. Чаще всего в качестве счётчиков цикла выступают переменные с именами i, j и k.

Шаг 2. проверка условияi < 5; — условное выражение, если оно возвращает true, тело цикла (инструкция в фигурных скобках) будет выполнено. В данном примере проверка условия идёт до тех пор, пока значение счётчика меньше 5.

Шаг 3. завершающая операцияi++ — операция приращения счётчика, увеличивает значение переменной var i на единицу. Вместо операции инкремента также может использоваться операция декремента.

По завершении цикла в переменной var i сохраняется значение 1. Следующий виток цикла выполняется для for (var i = 1; i < 5; i++) { }. Условное выражение вычисляется снова, чтобы проверить, является ли значение счётчика i всё ещё меньше 5. Если это так, операторы в теле цикла выполняются ещё раз. Завершающая операция снова увеличивает значение переменной на единицу. Шаги 2 и 3 повторяются до тех пор, пока условие i < 5; возвращает true.

1.2. Вывод значений массива

Чтобы вывести значения массива с помощью цикла for, нужно задействовать свойство массива length. Это поможет определить количество элементов в массиве и выполнить цикл такое же количество раз.

Если значение свойства length не изменяется в ходе выполнения цикла, можно сохранить его в локальной переменной, а затем использовать эту переменную в условном выражении. Таким образом можно повысить скорость выполнения цикла, так как значение свойства length будет извлекаться всего один раз за всё время работы цикла.

2. Цикл for...in

Циклы for...in используются для обхода свойств объектов, не являющихся массивами. Такой обход также называется перечислением. При обходе рекомендуется использовать метод hasOwnProperty(), чтобы отфильтровать свойства, которые были унаследованы от прототипа.

Предположим, что в сценарии до или после создания объекта user прототип объекта Object был расширен дополнительным методом clone().

Так как цепочка наследования прототипа постоянно проверяется интерпретатором, то все объекты автоматически получают доступ к новому методу.

Чтобы избежать обнаружения этого метода в процессе перечисления свойств объекта user, используется метод hasOwnProperty(), который отфильтрует свойства прототипа.

3. Цикл while

Цикл while - цикл с предварительной проверкой условного выражения. Инструкция внутри цикла (блок кода в фигурных скобках) будет выполняться в случае, если условное выражение вычисляется в true. Если первая проверка даст результат false, блок инструкций не выполнится ни разу.

После завершения итерации цикла условное выражение опять проверяется на истинность и процесс будет повторяться до тех пор, пока выражение не будет вычислено как false. В этом случае программа продолжит работу с первой строки, следующей непосредственно после цикла (если таковая имеется).

4. Цикл do...while

Цикл do...while; проверяет условие продолжения после выполнения цикла. В отличие от цикла while, в do...while; тело цикла выполняется как минимум один раз, так как условие проверяется в конце цикла, а не в начале. Данный цикл используется реже, чем while, так как на практике ситуация, когда требуется хотя бы однократное исполнение цикла, встречается редко.

В следующем примере операторы внутри цикла выполняются один раз, даже если условие не выполняется.

5. Бесконечные циклы

При создании любого цикла можно создать бесконечный цикл, который никогда не завершится. Такой цикл может потенциально продолжать работать до тех пор, пока работает компьютер пользователя. Большинство современных браузеров могут обнаружить это и предложат пользователю остановить выполнение скрипта. Чтобы избежать создания бесконечного цикла, вы должны быть уверены, что заданное условие в какой-то момент вернёт false.

6. Вложенные циклы

Цикл внутри другого цикла называется вложенным. При каждой итерации цикла вложенный цикл выполняется полностью. Вложенные циклы можно создавать с помощью цикла for и цикла while.

7. Управление циклом

Циклом можно управлять с помощью операторов break; и continue;.

7.1. Оператор break;

Оператор break; завершает выполнение текущего цикла. Он используется в исключительных случаях, когда цикл не может выполняться по какой-то причине, например, если приложение обнаруживает ошибку. Чаще всего оператор break; является частью конструкции if.

Когда оператор break; используется без метки, он позволяет выйти из цикла или из инструкции switch. В следующем примере создаётся счётчик, значения которого должны изменяться от 1 до 99, однако оператор break прерывает цикл после 14 итераций.

Для вложенных циклов оператор break; используется с меткой, с помощью которой завершается работа именованной инструкции. Метка позволяет выйти из любого блока кода. Именованной инструкцией может быть любая инструкция, внешняя по отношению к оператору break;. В качестве метки может быть имя инструкции if или имя блока инструкций, заключенных в фигурные скобки только для присвоения метки этому блоку. Между ключевым словом break; и именем метки не допускается перевод строки.

7.2. Оператор continue;

Оператор continue; останавливает текущую итерацию цикла и запускает новую итерацию. При этом, цикл while возвращается непосредственно к своему условию, а цикл for сначала вычисляет выражение инкремента, а затем возвращается к условию.

Оператор continue; также может применяться во вложенных циклах с меткой.

Outerloop

Циклические алгоритмы с неизвестным числом повторений.

Циклы while и do...while

while - это оператор в JavaScript, позволяющий воспроизводить цикл, который выполняет задачу столько раз, пока истинно заданное условие. while чаще используется, когда количество итераций заранее неизвестно.

Синтаксис while

while (условие) {

// инструкция (тело цикла)

}

Условие проверяется перед каждым заходом в цикл, если логическое значение верно, выполняется инструкция, в противном случае следующий на циклом while код.

Инструкция выполняется столько раз, сколько раз на входе в цикл условие было истинно.

Цикл do...while

Единственное отличие цикла do…while от while это то, что проверка условия происходит после выполнения тела цикла. Таким образом инструкция будет выполнена, как минимум один раз.

Синтаксис do...while

do {

// инструкция (тело цикла)

} while (условие);

Инструкция выполнится, как минимум один раз и будет повторятся до тех пор пока условие истинно. Если тело цикла состоит из одной строки фигурные скобки можно не ставить.

Условие проверяется после выполнения инструкции, если истинно тогда запускается еще одна итерация, в противном случае выполняется код следующий за циклом do…while

Программа создает модальное окно, в котором просит написать имя и далее выводит его в console.

Прерывание цикла - break

Выход из while при условии false не единственный способ прекратить цикл. С помощью директивы break мы можем завершить замкнутый цикл в любой момент.

Без if с break цикл работал бы до тех пор пока условие было бы истинным, в нашем случае while прервался когда n стал равен 4.

while в сочетании с инструкцией break это отличное решение, когда условие при котором нужно завершить бесконечный цикл находится не в начале или конце, а где-то посередине или может находится в нескольких местах.

Завершение текущей итерации - continue

Если необходимо прервать не весь цикл, а только одну итерацию при определенных условиях - используют дерективу continue. В while она переносит поток выполнения к условию.

Метки для continue и break

Если циклы вложены друг в друга директивы continue и break могут не решать нужную задачу без помощи меток. Например при определенных условиях нам нужно выйти из всех уровней цикла.

В этом примере при a = 5 и b = 8 мы переходим к завершению внешнего цикла while. Если a не равно 5 происходит выход из внутреннего цикла.

Массивы. Виды массивов.

Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто мы понимаем, что нам необходима *упорядоченная коллекция* данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

Новая возможность

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Допустим, нам нужен последний элемент массива.

Некоторые языки программирования позволяют использовать отрицательные индексы для той же цели, как-то так: fruits[-1].

Однако, в JavaScript такая запись не сработает. Её результатом будет undefined, поскольку индекс в квадратных скобках понимается буквально.

Методы pop/push, shift/unshift

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

push добавляет элемент в конец.

shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

push добавляет элемент в конец.

pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху:

Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется двусторонняя очередь.

Внутреннее устройство массива

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это по сути обычный синтаксис доступа по ключу, как obj[key], где в роли obj у нас arr, а в качестве ключа – числовой индекс.

Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length.

Но все они утратят эффективность, если мы перестанем работать с массивом как с «упорядоченной коллекцией данных» и начнём использовать его как обычный объект.

Это возможно, потому что в основе массива лежит объект. Мы можем присвоить ему любые свойства.

Но движок поймёт, что мы работаем с массивом, как с обычным объектом. Способы оптимизации, используемые для массивов, в этом случае не подходят, поэтому они будут отключены и никакой выгоды не принесут.

Варианты неправильного применения массива:

Добавление нечислового свойства, например: arr.test = 5.

Создание «дыр», например: добавление arr[0], затем arr[1000] (между ними ничего нет).

Заполнение массива в обратном порядке, например: arr[1000], arr[999] и т.д.

Массив следует считать особой структурой, позволяющей работать с *упорядоченными данными*. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, пожалуйста, используйте их именно в таких случаях. Если вам нужны произвольные ключи, вполне возможно, лучше подойдёт обычный объект {}.

Эффективность

Методы push/pop выполняются быстро, а методы shift/unshift – медленно.

Почему работать с концом массива быстрее, чем с его началом? Давайте посмотрим, что происходит во время выполнения:

fruits.shift(); // удаляем первый элемент с начала

Просто взять и удалить элемент с номером 0 недостаточно. Нужно также заново пронумеровать остальные элементы.

Действия при операции pop:

fruits.pop(); // удаляем один элемент с конца

Метод pop не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро.

Аналогично работает метод push.

Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

Технически, так как массив является объектом, можно использовать и вариант for..in:

Но на самом деле это – плохая идея. Существуют скрытые недостатки этого способа:

Цикл for..in выполняет перебор *всех свойств* объекта, а не только цифровых.

В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые *выглядят, как массив*. То есть, у них есть свойство length и индексы, но они также могут иметь дополнительные нечисловые свойства и методы, которые нам обычно не нужны. Тем не менее, цикл for..in выведет и их. Поэтому, если нам приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

Цикл for..in оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест. Но мы всё же должны представлять разницу.

В общем, не следует использовать цикл for..in для массивов.

Немного о «length»

Свойство length автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один.

Например, единственный элемент, имеющий большой индекс, даёт большую длину:

Обратите внимание, что обычно мы не используем массивы таким образом.

Ещё один интересный факт о свойстве length – его можно перезаписать.

Если мы вручную увеличим его, ничего интересного не произойдёт. Зато, если мы уменьшим его, массив станет короче. Этот процесс необратим, как мы можем понять из примера:

Таким образом, самый простой способ очистить массив – это arr.length = 0;.

new Array()

Существует ещё один вариант синтаксиса для создания массива:

let arr = *new Array*("Яблоко", "Груша", "и тд");

Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

toString

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделённых запятыми.

Массивы не имеют ни Symbol.toPrimitive, ни функционирующего valueOf, они реализуют только преобразование toString, таким образом, здесь [] становится пустой строкой, [1] становится "1", а [1,2] становится "1,2".

Когда бинарный оператор плюс "+" добавляет что-либо к строке, он тоже преобразует это в строку, таким образом:

Не сравнивайте массивы при помощи ==

В JavaScript, в отличие от некоторых других языков программирования, массивы не следует сравнивать при помощи оператора ==.

У этого оператора нет специального подхода к массивам, он работает с ними, как и с любыми другими объектами.

Давайте ещё раз напомним правила:

Два объекта равны друг другу == только в том случае, если они ссылаются на один и тот же объект.

Если один из аргументов == является объектом, а другой – примитивом, то объект преобразуется в примитив, как описано в главе Преобразование объектов в примитивы.

…За исключением null и undefined, которые равны == друг другу и ничему больше.

Оператор строгого равенства === ещё проще, так как он не преобразует типы.

Итак, если мы всё же сравниваем массивы с помощью ==, то они никогда не будут одинаковыми, если только мы не сравним две переменные, которые ссылаются на один и тот же массив

Технически эти массивы являются разными объектами. Так что они не равны. Оператор == не выполняет поэлементное сравнение.

Сравнение с примитивами также может дать, казалось бы, странные результаты:

Здесь, в обоих случаях, мы сравниваем примитив с объектом массива. Таким образом, массив [] преобразуется в примитив с целью сравнения и становится пустой строкой ''.

Затем продолжается процесс сравнения с примитивами, как описано в главе

Так как же сравнить массивы?

Это просто: не используйте оператор ==. Вместо этого сравните их поэлементам в цикле или используя методы итерации, описанные в следующей главе.

Создание массивов и методы работы с ними в JavaScript. Особенности использования свойства length.

Методы массивов

Массивы предоставляют множество методов. Чтобы было проще, в этой главе они разбиты на группы.

Добавление/удаление элементов

Мы уже знаем методы, которые добавляют и удаляют элементы из начала или конца:

arr.push(...items) – добавляет элементы в конец,

arr.pop() – извлекает элемент из конца,

arr.shift() – извлекает элемент из начала,

arr.unshift(...items) – добавляет элементы в начало.

Есть и другие.

splice

Как удалить элемент из массива?

Так как массивы – это объекты, то можно попробовать delete:

let arr = ["I", "go", "home"];delete arr[1]; // удалить "go"alert( arr[1] ); // undefined// теперь arr = ["I", , "home"];alert( arr.length ); // 3

Вроде бы, элемент и был удалён, но при проверке оказывается, что массив всё ещё имеет 3 элемента arr.length == 3.

Это нормально, потому что всё, что делает delete obj.key – это удаляет значение с данным ключом key. Это нормально для объектов, но для массивов мы обычно хотим, чтобы оставшиеся элементы сдвинулись и заняли освободившееся место. Мы ждём, что массив станет короче.

Поэтому для этого нужно использовать специальные методы.

Метод arr.splice(str) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

arr.splice(index[, deleteCount, elem1, ..., elemN])

Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

Этот метод проще всего понять, рассмотрев примеры.

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

Отрицательные индексы разрешены

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца, как тут:

slice

Метод arr.slice намного проще, чем похожий на него arr.splice.

Его синтаксис:

arr.slice([start], [end])

Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива.

Это похоже на строковый метод str.slice, но вместо подстрок возвращает подмассивы.

Можно вызвать slice и вообще без аргументов: arr.slice() создаёт копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

concat

Метод arr.concat создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

arr.concat(arg1, arg2...)

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.

В результате мы получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и так далее…

Если аргумент argN – массив, то все его элементы копируются. Иначе скопируется сам аргумент.

Обычно он копирует только элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть:

Перебор: forEach

Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

arr.forEach(function(item, index, array) { // ... делать что-то с item});

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

Поиск в массиве

Далее рассмотрим методы, которые помогут найти что-нибудь в массиве.

indexOf/lastIndexOf и includes

Методы arr.indexOf, arr.lastIndexOf и arr.includes имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.

arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.

arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если мы ищем false, он находит именно false, а не ноль.

Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf:

find и findIndex

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод arr.find.

Его синтаксис таков:

let result = arr.find(function(item, index, array) { // если true - возвращается текущий элемент и перебор прерывается // если все итерации оказались ложными, возвращается undefined});

Функция вызывается по очереди для каждого элемента массива:

item – очередной элемент.

index – его индекс.

array – сам массив.

Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод arr.findIndex – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

filter

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true.

На тот случай, если найденных элементов может быть много, предусмотрен метод arr.filter(fn).

Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:

let results = arr.filter(function(item, index, array) { // если true - элемент добавляется к результату, и перебор продолжается // возвращается пустой массив в случае, если ничего не найдено});

Преобразование массива

Перейдём к методам преобразования и упорядочения массива.

map

Метод arr.map является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

let result = arr.map(function(item, index, array) { // возвращается новое значение вместо элемента});

sort(fn)

Вызов arr.sort() сортирует массив *на месте*, меняя в нём порядок элементов.

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr.

По умолчанию элементы сортируются как строки.

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок, и действительно выходит, что "2" > "15".

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort().

Функция должна для пары значений возвращать:

function compare(a, b) { if (a > b) return 1; // если первое значение больше второго if (a == b) return 0; // если равны if (a < b) return -1; // если первое значение меньше второго}

Давайте возьмём паузу и подумаем, что же происходит. Упомянутый ранее массив arr может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. У нас есть набор *каких-то элементов*. Чтобы отсортировать его, нам нужна *функция, определяющая порядок*, которая знает, как сравнивать его элементы. По умолчанию элементы сортируются как строки.

Метод arr.sort(fn) реализует общий алгоритм сортировки. Нам не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная быстрая сортировка или Timsort). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся нам, это предоставить fn, которая делает это сравнение.

Кстати, если мы когда-нибудь захотим узнать, какие элементы сравниваются – ничто не мешает нам вывести их на экран:

В процессе работы алгоритм может сравнивать элемент со множеством других, но он старается сделать как можно меньше сравнений.

Функция сравнения может вернуть любое число

На самом деле от функции сравнения требуется любое положительное число, чтобы сказать «больше», и отрицательное число, чтобы сказать «меньше».

Для многих алфавитов лучше использовать метод str.localeCompare, для правильной сортировки букв, таких как Ö.

reverse

Метод arr.reverse меняет порядок элементов в arr на обратный.

Например:

let arr = [1, 2, 3, 4, 5]; arr.reverse();alert( arr ); // 5,4,3,2,1

Он также возвращает массив arr с изменённым порядком элементов.

split и join

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод str.split(delim) именно это и делает. Он разбивает строку на массив по заданному разделителю delim.

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

Разбивка по буквам

Вызов split(s) с пустым аргументом s разбил бы строку на массив букв:

let str = "тест";alert( str.split('') ); // т,е,с,т

Вызов arr.join(glue) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними.

reduce/reduceRight

Если нам нужно перебрать массив – мы можем использовать forEach, for или for..of.

Если нам нужно перебрать массив и вернуть данные для каждого элемента – мы используем map.

Методы arr.reduce и arr.reduceRight похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива.

Синтаксис:

let value = arr.reduce(function(accumulator, item, index, array) { // ...}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.

Аргументы:

accumulator – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),

item – очередной элемент массива,

index – его индекс,

array – сам массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент.

Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce.

Этот метод проще всего понять на примере.

Тут мы получим сумму всех элементов массива всего одной строкой:

let arr = [1, 2, 3, 4, 5];let result = arr.reduce((sum, current) => sum + current, 0);alert(result); // 15

Здесь мы использовали наиболее распространённый вариант reduce, который использует только 2 аргумента.

Давайте детальнее разберём, как он работает.

При первом запуске sum равен initial (последний аргумент reduce), то есть 0, а current – первый элемент массива, равный 1. Таким образом, результат функции равен 1.

При втором запуске sum = 1, и к нему мы добавляем второй элемент массива (2).

При третьем запуске sum = 3, к которому мы добавляем следующий элемент, и так далее…

Поток вычислений получается такой:

В виде таблицы, где каждая строка –- вызов функции на очередном элементе массива:

|  |  |  |  |
| --- | --- | --- | --- |
|  | sum | current | result |
| первый вызов | 0 | 1 | 1 |
| второй вызов | 1 | 2 | 3 |
| третий вызов | 3 | 3 | 6 |
| четвёртый вызов | 6 | 4 | 10 |
| пятый вызов | 10 | 5 | 15 |

Здесь отчётливо видно, как результат предыдущего вызова передаётся в первый аргумент следующего.

Результат – точно такой же! Это потому, что при отсутствии initial в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.

Таблица вычислений будет такая же за вычетом первой строки.

Но такое использование требует крайней осторожности. Если массив пуст, то вызов reduce без начального значения выдаст ошибку.

Поэтому рекомендуется всегда указывать начальное значение.

Метод arr.reduceRight работает аналогично, но проходит по массиву справа налево.

Array.isArray

Массивы не образуют отдельный тип языка. Они основаны на объектах.

Поэтому typeof не может отличить простой объект от массива:

alert(typeof {}); // objectalert(typeof []); // тоже object

…Но массивы используются настолько часто, что для этого придумали специальный метод: Array.isArray(value). Он возвращает true, если value массив, и false, если нет.

alert(Array.isArray({})); // falsealert(Array.isArray([])); // true

Большинство методов поддерживают «thisArg»

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр thisArg.

Этот параметр не объяснялся выше, так как очень редко используется, но для наиболее полного понимания темы мы обязаны его рассмотреть.

Вот полный синтаксис этих методов:

arr.find(func, thisArg); arr.filter(func, thisArg); arr.map(func, thisArg);// ...// thisArg - это необязательный последний аргумент

Значение параметра thisArg становится this для func.

Если бы мы в примере выше использовали просто users.filter(army.canJoin), то вызов army.canJoin был бы в режиме отдельной функции, с this=undefined. Это тут же привело бы к ошибке.

Вызов users.filter(army.canJoin, army) можно заменить на users.filter(user => army.canJoin(user)), который делает то же самое. Последняя запись используется даже чаще, так как функция-стрелка более наглядна.

Строки.

Строки

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

Кавычки

В JavaScript есть разные типы кавычек.

Строку можно создать с помощью одинарных, двойных либо обратных кавычек

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в ${…}

Ещё одно преимущество обратных кавычек — они могут занимать более одной строки, вот так:

Выглядит вполне естественно, не правда ли? Что тут такого? Но если попытаться использовать точно так же одинарные или двойные кавычки, то будет ошибка:

Одинарные и двойные кавычки в языке с незапамятных времён: тогда потребность в многострочных строках не учитывалась. Что касается обратных кавычек, они появились существенно позже, и поэтому они гибче.

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в документации. Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

Спецсимволы

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";alert(guestList); // список гостей, состоящий из нескольких строк

В частности, эти две строки эквивалентны, просто записаны по-разному:

// перевод строки добавлен с помощью символа перевода строкиlet str1 = "Hello\nWorld";// многострочная строка, созданная с использованием обратных кавычекlet str2 = `Hello

World`;alert(str1 == str2); // true

Есть и другие, реже используемые спецсимволы. Вот список:

|  |  |
| --- | --- |
| Символ | Описание |
| \n | Перевод строки |
| \r | В текстовых файлах Windows для перевода строки используется комбинация символов \r\n, а на других ОС это просто \n. Это так по историческим причинам, ПО под Windows обычно понимает и просто \n. |
| \', \" | Кавычки |
| \\ | Обратный слеш |
| \t | Знак табуляции |
| \b, \f, \v | Backspace, Form Feed и Vertical Tab — оставлены для обратной совместимости, сейчас не используются. |

Как вы можете видеть, все спецсимволы начинаются с обратного слеша, \ — так называемого «символа экранирования».

Он также используется, если необходимо вставить в строку кавычку.

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш — \' — иначе она бы обозначала окончание строки.

Разумеется, требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Так что мы можем применить и более элегантное решение, использовав для этой строки двойные или обратные кавычки:

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Вы можете явно видеть это в выводах alert в примерах выше.

Длина строки

Свойство length содержит длину строки:

alert( `My\n`.length ); // 3

Обратите внимание, \n — это один спецсимвол, поэтому тут всё правильно: длина строки 3.

length — это свойство

Бывает так, что люди с практикой в других языках случайно пытаются вызвать его, добавляя круглые скобки: они пишут str.length() вместо str.length. Это не работает.

Так как str.length — это числовое свойство, а не функция, добавлять скобки не нужно.

Доступ к символам

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод str.at(pos).

Как вы можете видеть, преимущество метода .at(pos) заключается в том, что он допускает отрицательную позицию. Если pos – отрицательное число, то отсчет ведется от конца строки.

Таким образом, .at(-1) означает последний символ, а .at(-2) – тот, что перед ним, и т.д.

Квадратные скобки всегда возвращают undefined для отрицательных индексов.

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана — она такая навсегда.

Можно создать новую строку и записать её в ту же самую переменную вместо старой.

Изменение регистра

Методы toLowerCase() и toUpperCase() меняют регистр символов:

Поиск подстроки

Существует несколько способов поиска подстроки.

str.indexOf

Первый метод — str.indexOf(substr, pos).

Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений.

Необязательный второй аргумент позволяет начать поиск с определённой позиции.

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле.

str.lastIndexOf(substr, position)

Также есть похожий метод str.lastIndexOf(substr, position), который ищет с конца строки к её началу.

Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

Поэтому надо делать проверку на -1:

Трюк с побитовым НЕ

Существует старый трюк с использованием побитового оператора НЕ — ~. Он преобразует число в 32-разрядное целое со знаком (signed 32-bit integer). Дробная часть, в случае, если она присутствует, отбрасывается. Затем все биты числа инвертируются.

На практике это означает простую вещь: для 32-разрядных целых чисел значение ~n равно -(n+1).

В частности:

alert( ~2 ); // -3, то же, что -(2+1)alert( ~1 ); // -2, то же, что -(1+1)alert( ~0 ); // -1, то же, что -(0+1)*alert( ~-1 ); // 0, то же, что -(-1+1)*

Таким образом, ~n равняется 0 только при n == -1 (для любого n, входящего в 32-разрядные целые числа со знаком).

Соответственно, прохождение проверки if ( ~str.indexOf("…") ) означает, что результат indexOf отличен от -1, совпадение есть.

Обычно использовать возможности языка каким-либо неочевидным образом не рекомендуется, но этот трюк широко используется в старом коде, поэтому его важно понимать.

Просто запомните: if (~str.indexOf(…)) означает «если найдено».

Впрочем, если быть точнее, из-за того, что большие числа обрезаются до 32 битов оператором ~, существуют другие числа, для которых результат тоже будет 0, самое маленькое из которых — ~4294967295=0. Поэтому такая проверка будет правильно работать только для строк меньшей длины.

На данный момент такой трюк можно встретить только в старом коде, потому что в новом он просто не нужен: есть метод .includes (см. ниже).

includes, startsWith, endsWith

Более современный метод str.includes(substr, pos) возвращает true, если в строке str есть подстрока substr, либо false, если нет.

Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна:

Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:

Методы str.startsWith и str.endsWith проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

Получение подстроки

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

str.slice(start [, end])

Возвращает часть строки от start до (не включая) end.

Если аргумент end отсутствует, slice возвращает символы до конца строки:

Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов *с конца строки*:

let str = "strin*gif*y";// начинаем с позиции 4 справа, а заканчиваем на позиции 1 справаalert( str.slice(-4, -1) ); // gif

str.substring(start [, end])

Возвращает часть строки *между*start и end (не включая) end.

Это — почти то же, что и slice, но можно задавать start больше end.  
 Если start больше end, то метод substring сработает так, как если бы аргументы были поменяны местами.

Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

str.substr(start [, length])

Возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

Этот метод находится в Annex B спецификации языка. Это означает, что его должны поддерживать только браузерные движки JavaScript, и использовать его не рекомендуется. Но на практике он поддерживается везде.

Давайте подытожим, как работают эти методы, чтобы не запутаться:

|  |  |  |
| --- | --- | --- |
| метод | выбирает… | отрицательные значения |
| slice(start, end) | от start до end (не включая end) | можно передавать отрицательные значения |
| substring(start, end) | между start и end | отрицательные значения равнозначны 0 |
| substr(start, length) | length символов, начиная от start | значение start может быть отрицательным |

Какой метод выбрать?

Все эти методы эффективно выполняют задачу. Формально у метода substr есть небольшой недостаток: он описан не в собственно спецификации JavaScript, а в приложении к ней — Annex B. Это приложение описывает возможности языка для использования в браузерах, существующие в основном по историческим причинам. Таким образом, в другом окружении, отличном от браузера, он может не поддерживаться. Однако на практике он работает везде.

Из двух других вариантов, slice более гибок, он поддерживает отрицательные аргументы, и его короче писать. Так что, в принципе, можно запомнить только его.

Сравнение строк

Как мы знаем из главы Операторы сравнения, строки сравниваются посимвольно в алфавитном порядке.

Тем не менее, есть некоторые нюансы.

Буквы, имеющие диакритические знаки, идут «не по порядку»:

Это может привести к своеобразным результатам при сортировке названий стран: нормально было бы ожидать, что Zealand будет после Österreich в списке.

Чтобы разобраться, что происходит, давайте ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в UTF-16. Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

str.codePointAt(pos)

Возвращает код для символа, находящегося на позиции pos:

Все строчные буквы идут после заглавных, так как их коды больше.

Некоторые буквы, такие как Ö, вообще находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.

Правильное сравнение

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты.

Поэтому браузеру нужно знать, какой язык использовать для сравнения.

К счастью, все современные браузеры (для IE10− нужна дополнительная библиотека Intl.JS) поддерживают стандарт ECMA 402, обеспечивающий правильное сравнение строк на разных языках с учётом их правил.

Для этого есть соответствующий метод.

Вызов str.localeCompare(str2) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

Отрицательное число, если str меньше str2.

Положительное число, если str больше str2.

0, если строки равны.

У этого метода есть два дополнительных аргумента, которые указаны в документации. Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

Способы организации строк, размещение в памяти.

4.5.3. Представление строк в памяти.

Представление строк в памяти зависит от того, насколько изменчивыми являются строки в каждой конкретной задаче, и средства такого представления варьируются от абсолютно статического до динамического. Универсальные языки программирования в основном обеспечивают работу со строками переменной длины, но максимальная длина строки должна быть указана при ее создании. Если программиста не устраивают возможности или эффективность тех средств работы со строками, которые предоставляет ему язык программирования, то он может либо определить свой тип данных "строка" и использовать для его представления средства динамической работы с памятью, либо сменить язык программирования на специально ориентированный на обработку текста (CNOBOL, REXX), в которых представление строк базируется на динамическом управлении памятью.

ВЕКТОРНОЕ ПРЕДСТАВЛЕНИЕ СТРОК.

Представление строк в виде векторов, принятое в большинстве универсальных языков программирования, позволяет работать со строками, размещенными в статической памяти. Кроме того, векторное представление позволяет легко обращаться к отдельным символам строки как к элементам вектора - по индексу.



ПРЕДСТАВЛЕНИЕ СТРОК ВЕКТОРОМ ПЕРЕМЕННОЙ ДЛИНЫ С ПРИЗНАКОМ КОНЦА.

Этот и все последующие за ним методы учитывают переменную длину строк. Признак конца - это особый символ, принадлежащий алфавиту (таким образом, полезный алфавит оказывается меньше на один символ), и занимает то же количество разрядов, что и все остальные символы. Издержки памяти при этом способе составляют 1 символ на строку.



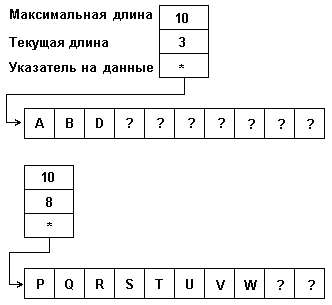
ПРЕДСТАВЛЕНИЕ СТРОК ВЕКТОРОМ ПЕРЕМЕННОЙ ДЛИНЫ СО СЧЕТЧИКОМ.

Счетчик символов - это целое число, и для него отводится достаточное количество битов, чтобы их с избытком хватало для представления длины самой длинной строки,какую только можно представить в данной машине. Обычно для счетчика отводят от 8 до 16 битов.



ВЕКТОР С УПРАВЛЯЕМОЙ ДЛИНОЙ.

Память под вектор с управляемой длиной отводится при создании строки и ее размер и размещение остаются неизменными все время существования строки. В дескрипторе такого вектора-строки может отсутствовать начальный индекс, так как он может быть зафиксирован раз навсегда установленными соглашениями, но появляется поле текущей длины строки. Размер строки, таким образом, может изменяться от 0 до значения максимального индекса вектора. "Лишняя" часть отводимой памяти может быть заполнена любыми кодами - она не принимается во внимание при оперировании со строкой. Хотя такое представление строк не обеспечивает экономии памяти, проектировщики систем программирования, как видно, считают это приемлемой платой за возможность работать с изменчивыми строками в статической памяти.



СИМВОЛЬНО - СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ СТРОК.

Списковое представление строк в памяти обеспечивает гибкость в выполнении разнообразных операций над строками (в частности, операций включения и исключения отдельных символов и целых цепочек) и использование системных средств управления памятью при выделении необходимого объема памяти для строки. Однако, при этом возникают дополнительные расходы памяти. Другим недостатком спискового представления строки является то, что логически соседние элементы строки не являются физически соседними в памяти. Это усложняет доступ к группам элементов строки по сравнению с доступом в векторном представлении строки.

ОДНОНАПРАВЛЕННЫЙ ЛИНЕЙНЫЙ СПИСОК.

Каждый символ строки представляется в виде элемента связного списка; элемент содержит код символа и указатель на следующий элемент, как показано на рис.4.7. Одностороннее сцепление представляет доступ только в одном направлении вдоль строки. На каждый символ строки необходим один указатель, который обычно занимает 2-4 байта.



Рис.4.7. Представление строки однонаправленным связным списком

ДВУНАПРАВЛЕННЫЙ ЛИНЕЙНЫЙ СПИСОК.

В каждый элемент списка добавляется также указатель на предыдущий элемент, как показано на рис.4.8. Двустороннее сцепление допускает двустороннее движение вдоль списка, что может значительно повысить эффективность выполнения некоторых строковых операция. При этом на каждый символ строки необходимо два указателя , т.е. 4-8 байт.

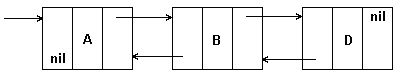


Рис.4.8. Представление строки двунаправленным связным списком

БЛОЧНО - СВЯЗНОЕ ПРЕДСТАВЛЕНИЕ СТРОК.

Это представление позволяет в болшинстве операций избежать затрат, связанных с управлением динамической памятью, но в то же время обеспечивает достаточно эффективное использование памяти при работе со строками переменной длины.

МНОГОСИМВОЛЬНЫЕ ЗВЕНЬЯ ФИКСИРОВАННОЙ ДЛИНЫ.

Многосимвольные группы (звенья) организуются в список, так что каждый элемент списка, кроме последнего, содержит группу элементов строки и указатель следующего элемента списка. Поле указателя последнего элемента списка хранит признак конца - пустой указатель. В процессе обработки строки из любой ее позиции могут быть исключены или в любом месте вставлены элементы, в результате чего звенья могут содержать меньшее число элементов, чем было первоначально. По этой причине необходим специальный символ, который означал бы отсутствие элемента в соответствующей позиции строки. Обозначим такой символ 'emp', он не должен входить в множество символов,из которых организуется строка. Пример многосимвольных звеньев фиксированной длинны по 4 символа в звене показан на рис.4.9.

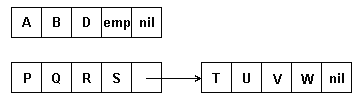


Рис. 4.9. Представление строки многосимвольными звеньями постоянной длины

Такое представление обеспечивает более эффективное использование памяти, чем символьно-связное. Операции вставки/удаления в ряде случаев могут сводиться к вставке/удалению целых блоков. Однако, при удалении одиночных символов в блоках могут накапливаться пустые символы emp, что может привести даже к худшему использованию памяти, чем в символьно-связном представлении.

Методы обработки строк.

Любые текстовые данные в JavaScript считаются строками.  Это примитивный тип, но язык позволяет работать с ним так, будто он является объектом. В том числе — использовать  встроенные в JS методы строк, которые собраны в этой шпаргалке.

Создание пользовательских функций в JavaScript: способы объявления и вызова.

Функции

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

Например, необходимо красиво вывести сообщение 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 функции

Для создания функций мы можем использовать *объявление функции*.

Пример объявления функции:

function showMessage() { alert( 'Всем привет!' );}

Вначале идёт ключевое слово function, после него *имя функции*, затем список *параметров* в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

function имя(параметры) { ...тело...}

Наша новая функция может быть вызвана по своему имени: showMessage().

Вызов showMessage() выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода.

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

Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции.

Внешние переменные

У функции есть доступ к внешним переменным, например:

let *userName* = 'Вася';function showMessage() { let message = 'Привет, ' + *userName*; alert(message);}showMessage(); // Привет, Вася

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Например:

let *userName* = 'Вася';function showMessage() { *userName* = "Петя"; // (1) изменяем значение внешней переменной let message = 'Привет, ' + *userName*; alert(message);}alert( userName ); // *Вася* перед вызовом функцииshowMessage();alert( userName ); // *Петя*, значение внешней переменной было изменено функцией

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована:

Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в вышеприведённом коде – называются *глобальными*.

*Глобальные переменные* видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры.

Когда функция вызывается в строках (\*) и (\*\*), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.

Значение, передаваемое в качестве параметра функции, также называется *аргументом*.

Другими словами:

Параметр – это переменная, указанная в круглых скобках в объявлении функции.

Аргумент – это значение, которое передаётся функции при её вызове.

Мы объявляем функции со списком параметров, затем вызываем их, передавая аргументы.

Значения по умолчанию

Если при вызове функции аргумент не был указан, то его значением становится undefined.

Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

showMessage("Аня");

Это не приведёт к ошибке. Такой вызов выведет "\*Аня\*: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра.

В приведённом выше примере, функция anotherFunction() не будет вызвана вообще, если указан параметр text.

С другой стороны, функция будет независимо вызываться каждый раз, когда text отсутствует.

Использование параметров по умолчанию в ранних версиях JavaScript

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах.

Альтернативные параметры по умолчанию

Иногда имеет смысл присваивать значения по умолчанию для параметров не в объявлении функции, а на более позднем этапе.

Во время выполнения функции мы можем проверить, передан ли параметр, сравнив его с undefined:

Современные движки JavaScript поддерживают оператор нулевого слияния??. Его использование будет лучшей практикой, в случае, если большинство ложных значений, таких как 0, следует расценивать как «нормальные».

Возврат значения

Функция может вернуть результат, который будет передан в вызвавший её код.

Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше).

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции.

Результат функции с пустым return или без него – undefined

Если функция не возвращает значения, это всё равно, как если бы она возвращала undefined:

Для длинного выражения в return может быть заманчиво разместить его на нескольких отдельных строках,

Если мы хотим, чтобы возвращаемое выражение занимало несколько строк, нужно начать его на той же строке, что и return. Или, хотя бы, поставить там открывающую скобку

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть кратким, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Благодаря префиксам, при первом взгляде на имя функции становится понятным, что делает её код, и какое значение она может возвращать.

Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием.

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Сверхкороткие имена функций

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

Например, сравним ниже две функции showPrimes(n). Каждая из них выводит простое число до n.

Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

Итого

Объявление функции имеет вид:

function имя(параметры, через, запятую) { /\* тело, код функции \*/}

Передаваемые значения копируются в параметры функции и становятся локальными переменными.

Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.

Функция может возвращать значение. Если этого не происходит, тогда результат равен undefined.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Область видимости переменных.

Одной из самых основных идей практически всех языков программирования является возможность сохранять значения в переменных, а позже извлекать или менять эти значения.

Правила добавление переменных в нашу программу так, чтобы потом можно было их найти и получить их значение, должны определяться четкими границами. Эти границы определяют, какие переменные будут нам доступны в определенных местах программы, и процесс поиска переменных по их идентификатору (имени).

Если дать более техническое определение, то оно звучит так:

Область видимости переменных или просто “Область видимости” (англ. variable scope или просто scope) — это такая область программы, в пределах которой установлена связь между некоторой переменной и её идентификатором (именем), по которому можно получить значение этой переменной. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной, либо быть свободным (вообще не связанным ни с какой из переменных).

Для понимания этой концепции можно привести следующую аналогию. Представим человека, которому была назначена деловая встреча в некотором бизнес-центре, где он раньше не был. Он заходит в нужный кабинет и так как там еще никого нет, он решает узнать время, для этого ему необходимы часы. Этот человек начинает искать часы, обнаруживает настольные часы и узнаёт по ним время.

В данной аналогии:

область видимости (scope) — это кабинет;

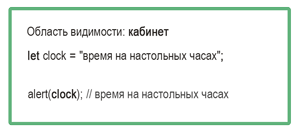
имя переменной, по которой происходит поиск нужного предмета, это “часы” (обозначим как clock);

в данной области видимости, с этим идентификатором (“часы”) ассоциируются *настольные часы*. Здесь стоит вспомнить, что переменные, это своего рода контейнеры (области памяти), для хранения необходимых значений. Именно настольные часы в рамках кабинета выступают неким контейнером для хранения времени.

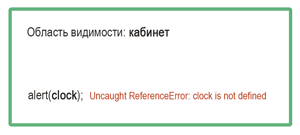
значение этой переменной обозначим как “время на настольных часах”.

Поэтому, в данном кабинете переменная, которая представляет собой “настольные часы”, в JavaScript можно объявить как

let clock = "время на настольных часах";



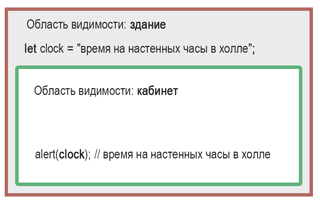
Теперь допустим, что человек в кабинете не нашел абсолютно никаких часов, другими словами в этой области видимости нет никаких переменных (ни настольных часов, ни каких-либо других), которые были бы связаны с идентификатором “часы” (clock). И в таком случае, в JavaScript запрос получения значения переменной с именем clock вызовет ошибку: Uncaught ReferenceError: clock is not defined. Которая сообщает о том, что в данной области видимости переменная, связанная с идентификатором clock, не определена.



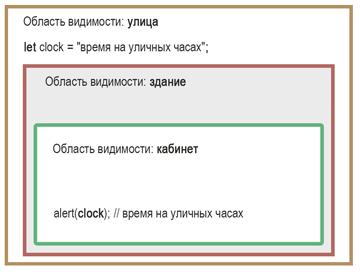
Разовьем аналогию дальше и представим, что хоть часов в кабинете и нет, этот человек знает, что в холле здания есть настенные часы. Он знает об этом, так как проходя в кабинет, отметил их наличие. И поэтому, он может получить нужное значение (время) из своего *окружения* — “выглянуть” из кабинета и посмотреть время на настенных часах.

В этой аналогии здание — это еще одна область видимости, которая также является окружением для области видимости “кабинет”. В данном здании идентификатор “часы” (clock) связан с переменной “настенные часы в холле”. И объявление этой переменной будет следующим:

let clock = "время на настенных часах в холле";



Мы можем продолжить эту аналогию. Представим, что и в здании нет часов, но они есть на улице. Поэтому, даже находясь в кабинете, он может обратиться к своему окружению — выглянуть на улицу и посмотреть время на уличных часах.

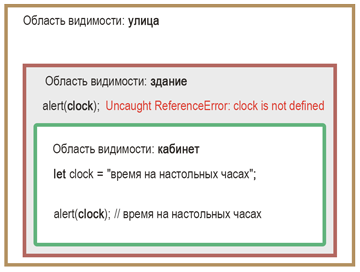


В JavaScript области видимости ограничиваются функциями (функциональная область видимости) или блоками инструкций (блочная область видимости).

Поэтому, пример с часами на JavaScript можно записать так:

function street() { //scope: street let clock = "время на уличных часах"; function building() { //scope: building function office() { //scope: office alert(clock); //покажет "время на уличных часах" } }}

Также необходимо отметить, что переменные, объявленные внутри функции, недоступны снаружи, то есть даже если часы есть в кабинете, а человек находится в здании или на улице, где нет никаких часов, то обратится к часам в кабинете он не сможет. Более того, он в принципе не имеет никакого представления о содержимом кабинета, пока туда не зайдет.

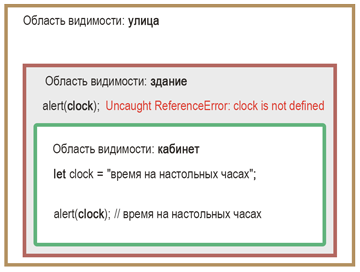


function street() { //scope: street function building() { //scope: building alert(clock); // выполнение прервётся с ошибкой: Uncaught ReferenceError: clock is not defined function office() { //scope: office let clock = "время на настольных часах"; alert(clock); // покажет "время на настольных часах" } }}

Если нам необходимо, чтобы переменная была доступна отовсюду, то её необходимо объявить вне всех функций, то есть глобально.

Глобальная область видимости - это та, у которой нет внешней области видимости (внешнего окружения). Она является отправной точкой нашей программы и самой внешней областью видимости для всех других, которые в неё вложены.

В нашей аналогии, пусть это будет вселенная, где есть часы, которые можно увидеть из любой точки.



Переменные называются глобальными, если они доступны из любой точки программы. То есть те, которые объявлены в глобальной, самой внешней области видимости. Переменные, которые объявлены внутри функции и недоступны снаружи, называются локальными переменными. Область видимости, ограниченная функцией или блоком, это локальная область видимости.

Глобальная область видимости не вложена ни в какие другие области и находится вне всех функций, являясь родительской для всех других вложенных в неё областей видимости. И для объявления глобальной переменной clock наш код можно переписать так:

//global scope: windowlet clock = "время на часах, что доступны отовсюду";function street() { //scope: street function building() { //scope: building function office() { //scope: office alert(clock); //покажет "время на часах, что доступны отовсюду" } }}

В случае, если одна и та же переменная объявлена в нескольких вложенных областях видимости, её значение будет браться из текущей области видимости, где она запрашивается. Например:

//global scope: windowlet clock = "время на часах, что доступны отовсюду";function street() { //scope: street alert(clock); //покажет "время на часах, что доступны отовсюду" function building() { //scope: building let clock = "время на настенных часах в холле"; alert(clock); //покажет "время на настенных часах в холле" function office() { //scope: office let clock = "время на настольные часах"; alert(clock); //покажет "время на настольные часах" } }}

Несмотря на то, что переменная clock объявлена и глобально, и в функциях building и office, её значение будет браться из той области видимости, где оно непосредственно запрашивалось через alert(clock).

Если в текущей области переменная не определена, как в функции street, то значение возьмется из ближайшего окружения (родительской области видимости), в котором определена необходимая переменная. В данном случае, для области видимости street, значение clock будет браться из глобальной области видимости. (О том, как именно происходит сам поиск нужных переменных, будет разбираться в следующей части).

Это называется “затенение переменных”, когда текущая переменная скрывает значение переменной объявленной в родительской области видимости. Вообще, чтобы избежать путаницы при командной разработке и неясности в том, в каких переменных что хранится, лучше не применять затенение и использовать разные имена переменных.

Пока в примере мы разбирали локальные области, которые были ограничены функциями. Но также область видимости может быть ограничена блоком инструкций.

Замыкания. Стандартные функции JavaScript. Функции временной задержки

Замыкания

Замыкание — это комбинация функции и лексического окружения, в котором эта функция была определена. Другими словами, замыкание даёт вам доступ к Scope (en-US) внешней функции из внутренней функции. В JavaScript замыкания создаются каждый раз при создании функции, во время её создания.

Лексическая область видимости

Рассмотрим следующий пример:

function init() { var name = "Mozilla"; // name - локальная переменная, созданная в init function displayName() { // displayName() - внутренняя функция, замыкание alert (name); // displayName() использует переменную, объявленную в родительской функции } displayName();}init();

init() создаёт локальную переменную name и определяет функцию displayName(). displayName() — это внутренняя функция — она определена внутри init() и доступна только внутри тела функции init(). Обратите внимание, что функция displayName() не имеет никаких собственных локальных переменных. Однако, поскольку внутренние функции имеют доступ к переменным внешних функций, displayName() может иметь доступ к переменной name, объявленной в родительской функции init().

Выполните этот код и обратите внимание, что команда alert() внутри displayName() благополучно выводит на экран содержимое переменной name объявленной в родительской функции. Это пример так называемой лексической области видимости *(lexicalscoping)*: в JavaScript область действия переменной определяется по её расположению в коде (это очевидно *лексически*), и вложенные функции имеют доступ к переменным, объявленным вовне. Этот механизм и называется Lexical scoping (область действия, ограниченная лексически).

Замыкание

Рассмотрим следующий пример:

function makeFunc() { var name = "Mozilla"; function displayName() { alert(name); } return displayName;};var myFunc = makeFunc();myFunc();

Если выполнить этот код, то результат будет такой же, как и выполнение init() из предыдущего примера: строка "Mozilla" будет показана в JavaScript alert диалоге. Что отличает этот код и представляет для нас интерес, так это то, что внутренняя функция displayName() была возвращена из внешней до того, как была выполнена.

На первый взгляд, кажется неочевидным, что этот код правильный, но он работает. В некоторых языках программирования локальные переменные-функции существуют только во время выполнения этой функции. После завершения выполнения makeFunc() можно ожидать, что переменная *name* больше не будет доступна. Однако, поскольку код продолжает нормально работать, очевидно, что это не так в случае JavaScript.

Причина в том, что функции в JavaScript формируют так называемые *замыкания*. *Замыкание* — это комбинация функции и лексического окружения, в котором эта функция была объявлена. Это окружение состоит из произвольного количества локальных переменных, которые были в области действия функции во время создания замыкания. В рассмотренном примере myFunc — это ссылка на экземпляр функции displayName, созданной в результате выполнения makeFunc. Экземпляр функции displayName в свою очередь сохраняет ссылку на своё лексическое окружение, в котором есть переменная name. По этой причине, когда происходит вызов функции myFunc, переменная name остаётся доступной для использования и сохранённый в ней текст "Mozilla" передаётся в alert.

А вот немного более интересный пример — функция makeAdder:

function makeAdder(x) { return function(y) { return x + y; };};var add5 = makeAdder(5);var add10 = makeAdder(10); console.log(add5(2)); // 7 console.log(add10(2)); // 12

Здесь мы определили функцию makeAdder(x), которая получает единственный аргумент x и возвращает новую функцию. Эта функция получает единственный аргумент y и возвращает сумму x и y.

По существу makeAdder — это фабрика функций: она создаёт функции, которые могут прибавлять определённое значение к своему аргументу. В примере выше мы используем нашу фабричную функцию для создания двух новых функций — одна прибавляет 5 к своему аргументу, вторая прибавляет 10.

add5 и add10 — это примеры замыканий. Эти функции делят одно определение тела функции, но при этом они сохраняют различные окружения. В окружении функции add5x — это 5, в то время как в окружении add10x — это 10.

Замыкания на практике

Замыкания полезны тем, что позволяют связать данные (лексическое окружение) с функцией, которая работает с этими данными. Очевидна параллель с объектно-ориентированным программированием, где объекты позволяют нам связать некоторые данные (свойства объекта) с одним или несколькими методами.

Следовательно, замыкания можно использовать везде, где вы обычно использовали объект с одним единственным методом.

Такие ситуации повсеместно встречаются в web-разработке. Большое количество front-end кода, который мы пишем на JavaScript, основано на обработке событий. Мы описываем какое-то поведение, а потом связываем его с событием, которое создаётся пользователем (например, клик мышкой или нажатие клавиши). При этом наш код обычно привязывается к событию в виде обратного/ответного вызова (callback): *callback функция - функция выполняемая в ответ на возникновение события*.

Давайте рассмотрим практический пример: допустим, мы хотим добавить на страницу несколько кнопок, которые будут менять размер текста. Как вариант, мы можем указать свойство font-size на элементе body в пикселах, а затем устанавливать размер прочих элементов страницы (таких, как заголовки) с использованием относительных единиц em:

body { font-family: Helvetica, Arial, sans-serif; font-size: 12px;}h1 { font-size: 1.5em;}h2 { font-size: 1.2em;}

Тогда наши кнопки будут менять свойство font-size элемента body, а остальные элементы страницы просто получат это новое значение и отмасштабируют размер текста благодаря использованию относительных единиц.

Используем следующий JavaScript:

function makeSizer(size) { return function() { document.body.style.fontSize = size + 'px'; };};var size12 = makeSizer(12);var size14 = makeSizer(14);var size16 = makeSizer(16);

Теперь size12, size14, и size16 - это функции, которые меняют размер текста в элементе body на значения 12, 14, и 16 пикселов, соответственно. После чего мы цепляем эти функции на кнопки примерно так:

document.getElementById('size-12').onclick = size12; document.getElementById('size-14').onclick = size14; document.getElementById('size-16').onclick = size16;

<a href="#" id="size-12">12</a><a href="#" id="size-14">14</a><a href="#" id="size-16">16</a>

Эмуляция частных (private) методов с помощью замыканий

Языки вроде Java позволяют нам объявлять частные (private) методы . Это значит, что они могут быть вызваны только методами того же класса, в котором объявлены.

JavaScript не имеет встроенной возможности сделать такое, но это можно эмулировать с помощью замыкания. Частные методы полезны не только тем, что ограничивают доступ к коду, это также мощное средство глобальной организации пространства имён, позволяющее не засорять публичный интерфейс вашего кода внутренними методами классов.

Код ниже иллюстрирует, как можно использовать замыкания для определения публичных функций, которые имеют доступ к закрытым от пользователя (private) функциям и переменным. Такая манера программирования называется модульное программирование:

var Counter = (function() { var privateCounter = 0; function changeBy(val) { privateCounter += val; } return { increment: function() { changeBy(1); }, decrement: function() { changeBy(-1); }, value: function() { return privateCounter; } };})();alert(Counter.value()); /\* Alerts 0 \*/ Counter.increment(); Counter.increment();alert(Counter.value()); /\* Alerts 2 \*/ Counter.decrement();alert(Counter.value()); /\* Alerts 1 \*/

Тут много чего поменялось. В предыдущем примере каждое замыкание имело свой собственный контекст исполнения (окружение). Здесь мы создаём единое окружение для трёх функций: Counter.increment, Counter.decrement, и Counter.value.

Единое окружение создаётся в теле анонимной функции, которая исполняется в момент описания. Это окружение содержит два приватных элемента: переменную privateCounter и функцию changeBy(val). Ни один из этих элементов не доступен напрямую, за пределами этой самой анонимной функции. Вместо этого они могут и должны использоваться тремя публичными функциями, которые возвращаются анонимным блоком кода (anonymous wrapper), выполняемым в той же анонимной функции.

Эти три публичные функции являются замыканиями, использующими общий контекст исполнения (окружение). Благодаря механизму lexical scoping в Javascript, все они имеют доступ к переменной privateCounter и функции changeBy.

Заметьте, мы описываем анонимную функцию, создающую счётчик, и тут же запускаем её, присваивая результат исполнения переменной Counter. Но мы также можем не запускать эту функцию сразу, а сохранить её в отдельной переменной, чтобы использовать для дальнейшего создания нескольких счётчиков вот так:

Заметьте, что счётчики работают независимо друг от друга. Это происходит потому, что у каждого из них в момент создания функцией makeCounter() также создавался свой отдельный контекст исполнения (окружение). То есть приватная переменная privateCounter в каждом из счётчиков это действительно отдельная, самостоятельная переменная.

Используя замыкания подобным образом, вы получаете ряд преимуществ, обычно ассоциируемых с объектно-ориентированным программированием, таких как изоляция и инкапсуляция.

Создание замыканий в цикле: Очень частая ошибка

До того, как в версии ECMAScript 6 ввели ключевое слово let, постоянно возникала следующая проблема при создании замыканий внутри цикла.

Массив helpText описывает три подсказки для трёх полей ввода. Цикл пробегает эти описания по очереди и для каждого из полей ввода определяет, что при возникновении события onfocus для этого элемента должна вызываться функция, показывающая соответствующую подсказку.

Если вы запустите этот код, то увидите, что он работает не так, как мы ожидаем интуитивно. Какое поле вы бы ни выбрали, в качестве подсказки всегда будет высвечиваться сообщение о возрасте.

Проблема в том, что функции, присвоенные как обработчики события onfocus, являются замыканиями. Они состоят из описания функции и контекста исполнения (окружения), унаследованного от функции setupHelp. Было создано три замыкания, но все они были созданы с одним и тем же контекстом исполнения. К моменту возникновения события onfocus цикл уже давно отработал, а значит, переменная item (одна и та же для всех трёх замыканий) указывает на последний элемент массива, который как раз в поле возраста.

В качестве решения в этом случае можно предложить использование функции, фабричной функции (function factory)

Вот это работает как следует. Вместо того, чтобы делить на всех одно окружение, функция makeHelpCallback создаёт каждому из замыканий своё собственное, в котором переменная item указывает на правильный элемент массива helpText.

Соображения по производительности

Не нужно без необходимости создавать функции внутри функций в тех случаях, когда замыкания не нужны. Использование этой техники увеличивает требования к производительности как в части скорости, так и в части потребления памяти.

Как пример, при написании нового класса есть смысл помещать все методы в прототип его объекта, а не описывать их в тексте конструктора. Если сделать по-другому, то при каждом создании объекта для него будет создан свой экземпляр каждого из методов, вместо того, чтобы наследовать их из прототипа.

Методы вынесены в прототип. Тем не менее, переопределять прототип — само по себе является плохой привычкой, поэтому давайте перепишем всё так, чтобы новые методы просто добавились к уже существующему прототипу.

В обоих примерах выше методы определяются один раз — в прототипе. И все объекты, использующие данный прототип, будут использовать это определение без дополнительного расхода вычислительных ресурсов. Смотрите подробное описание в статье Подробнее об объектной модели.

Стандартные функции

Здесь описаны лишь "настоящие" функции JavaScript (не являющиеся *методами* каких-либо объектов):

eval (интерпретация строки как фрагмента JavaScript-кода)

escape (замена специальных символов в строке на их коды)

unescape (операция, обратная escape)

parseFloat (преобразование строки в число)

parseInt (преобразование строки в целое число)

Если нужной вам функции здесь не оказалось, ее стоит поискать среди JavaScript-методов, реализованных для следующих объектов:

объект Math (математические функции)

объект String (обработка строк)

объект Window (открытие окон, диалоги)

eval  
*Синтаксис:* eval(выражение)   
 eval('строка')

*Действие* Вычисляется значение выражения, стоящего в скобках. Если это значение не является строкой, то оно и считается значением функции eval.

Если же значение выражения в скобках есть строка, то *строка интерпретируется и выполняется* (точно так же, как любой другой фрагмент JavaScript-кода). В этом случае значением функции eval считается значение последнего из выражений, вычисленных при выполнении JavaScript-кода.

Аргументом функции escape должна быть строка или строковое выражение. Функция escape преобразует строку следующим образом: цифры, латинские буквы и символы + - \* / . \_ @ остаются без изменения, а все остальные символы заменяются на комбинации %XX, где XX- шестнадцатеричный код соответствующего входного символа.

Восстановить первоначальный вид строки можно с помощью функции unescape.

В новых браузерах, поддерживающих работу с Unicode-символами, функция escape ведет себя по-новому: если на входе Unicode-символ, то escape возвращает последовательность %uXXXX, где XXXX- Unicode-кодировка входного символа.

На *русские буквы* функция escape действует по-разному в разных браузерах (в зависимости от того, поддерживается ли браузером Unicode и представлена ли русская буква в Unicode-кодировке). Вот, например, результаты выполнения оператора escape("П"):

%u041F (Internet Explorer 4)

%CF (Netscape 4.5)

unescape  
*Синтаксис:* unescape('строка')   
 unescape(выражение)

*Действие* Функция unescape позволяет восстановить первоначальный вид строки, которая была преобразована функцией escape. Аргументом функции unescape должна быть строка или строковое выражение.

Функция unescape преобразует строку следующим образом: если встречаются комбинации %XX, где XX- две шестнадцатеричные цифры, то каждая такая комбинация заменяется на символ с кодом XX. В остальном строка не меняется.

(В браузерах, поддерживающих работу с Unicode-символами, функция unescape, кроме комбинаций %XX, отлавливает также комбинации %uXXXX; см. выше описание функции escape.)

*Пример:*

unescape("It%27s%20me%21")

// результат: "It's me!"

parseFloat  
*Синтаксис:* parseFloat('строка')   
 parseFloat(выражение)

*Действие* Аргументом функции parseFloat должна быть строка или строковое выражение. Результат функции есть число, десятичной записью которого является данная cтрока (или ее начальные символы).

Если начальные символы строки-аргумента невозможно "прочитать" как десятичное число, результаты будут разными в разных браузерах (либо 0, либо NaN).

*Примеры* (справа в комментариях - результаты):

parseFloat('1.45kg') // 1.45

parseFloat('77.3') // 77.3

parseFloat('077.3') // 77.3

parseFloat('0x77.3') // 0

parseFloat('.3') // 0.3

parseFloat('0.1e6') // 100000

parseInt  
*Синтаксис:* parseInt( 'строка' [, основание] )   
 parseInt( выражение [, основание] )

*Действие* Первым аргументом функции parseInt должна быть строка или строковое выражение. Результат функции есть *целое* число, записью которого является данная cтрока (или ее начало).

Если присутствует второй аргумент (число основание), то строка интерпретируется как число, записанное в системе счисления с данным основанием. Допустимые значения аргумента основание: от 2 до 36.

Если второй аргумент отсутствует, то функция определяет основание системы счисления по правилам языка JavaScript. Строки с начальными символами 0x или -0x интерпретируются как шестнадцатеричные числа; cтроки с начальными символами 0 или -0- как восьмеричные числа. Все остальные строки интерпретируются как десятичные числа.

Если начальные символы строки-аргумента невозможно "прочитать" как целое число, результаты будут разными в разных браузерах (либо 0, либо NaN).

Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

setTimeout позволяет вызвать функцию один раз через определённый интервал времени.

setInterval позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout

Синтаксис:

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...);

Параметры:

func|code

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

delay

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

arg1, arg2…

Аргументы, передаваемые в функцию

Например, данный код вызывает sayHi() спустя одну секунду:

function sayHi() { alert('Привет');}*setTimeout(sayHi, 1000);*

С аргументами:

function sayHi(phrase, who) { alert( phrase + ', ' + who );}*setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон*

Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

Это также будет работать:

setTimeout("alert('Привет')", 1000);

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

setTimeout(() => alert('Привет'), 1000);

Передавайте функцию, но не запускайте её

Начинающие разработчики иногда ошибаются, добавляя скобки () после функции:

// не правильно!setTimeout(sayHi(), 1000);

Это не работает, потому что setTimeout ожидает ссылку на функцию. Здесь sayHi() запускает выполнение функции, и *результат выполнения* отправляется в setTimeout. В нашем случае результатом выполнения sayHi() является undefined (так как функция ничего не возвращает), поэтому ничего не планируется.

Отмена через clearTimeout

Вызов setTimeout возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

let timerId = setTimeout(...);clearTimeout(timerId);

В коде ниже планируем вызов функции и затем отменяем его (просто передумали). В результате ничего не происходит:

let timerId = setTimeout(() => alert("ничего не происходит"), 1000);alert(timerId); // идентификатор таймераclearTimeout(timerId);alert(timerId); // тот же идентификатор (не принимает значение null после отмены)

Как мы видим из вывода alert, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

Повторюсь, что нет единой спецификации на эти методы, поэтому такое поведение является нормальным.

Для браузеров таймеры описаны в разделе таймеров стандарта HTML5.

setInterval

Метод setInterval имеет такой же синтаксис как setTimeout:

let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...);

Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId).

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

// повторить с интервалом 2 секундыlet timerId = setInterval(() => alert('tick'), 2000);// остановить вывод через 5 секундsetTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

Во время показа alert время тоже идёт

В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа alert/confirm/prompt.

Так что если вы запустите код выше и подождёте с закрытием alert несколько секунд, то следующий alert будет показан сразу, как только вы закроете предыдущий. Интервал времени между сообщениями alert будет короче, чем 2 секунды.

Вложенный setTimeout

Есть два способа запускать что-то регулярно.

Один из них setInterval. Другим является вложенный setTimeout. Например:

/\*\* вместо:

let timerId = setInterval(() => alert('tick'), 2000);

\*/let timerId = setTimeout(function tick() { alert('tick'); *timerId = setTimeout(tick, 2000); // (\*)*}, 2000);

Метод setTimeout выше планирует следующий вызов прямо после окончания текущего (\*).

Вложенный setTimeout – более гибкий метод, чем setInterval. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.

Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд… Вот псевдокод:

let delay = 5000;let timerId = setTimeout(function request() { ...отправить запрос... if (ошибка запроса из-за перегрузки сервера) { // увеличить интервал для следующего запроса delay \*= 2; } timerId = setTimeout(request, delay);}, delay);

А если функции, которые мы планируем, ресурсоёмкие и требуют времени, то мы можем измерить время, затраченное на выполнение, и спланировать следующий вызов раньше или позже.

Вложенный setTimeout позволяет задать задержку между выполнениями более точно, чем setInterval.

Сравним два фрагмента кода. Первый использует setInterval:

let i = 1;setInterval(function() { func(i);}, 100);

Второй использует вложенный setTimeout:

let i = 1;setTimeout(function run() { func(i); setTimeout(run, 100);}, 100);

Для setInterval внутренний планировщик будет выполнять func(i) каждые 100 мс:

Обратили внимание?

Реальная задержка между вызовами func с помощью setInterval меньше, чем указано в коде!

Это нормально, потому что время, затраченное на выполнение func, использует часть заданного интервала времени.

Вполне возможно, что выполнение func будет дольше, чем мы ожидали, и займёт более 100 мс.

В данном случае движок ждёт окончания выполнения func и затем проверяет планировщик и, если время истекло, *немедленно* запускает его снова.

В крайнем случае, если функция всегда выполняется дольше, чем задержка delay, то вызовы будут выполняться без задержек вообще.

Ниже представлено изображение, показывающее процесс работы рекурсивного setTimeout:

ВложенныйsetTimeout гарантирует фиксированную задержку (здесь 100 мс).

Это потому, что новый вызов планируется в конце предыдущего.

Сборка мусора и колбэк setTimeout/setInterval

Когда функция передаётся в setInterval/setTimeout, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик мусора, даже если на неё нет других ссылок.

// функция остаётся в памяти до тех пор, пока планировщик обращается к нейsetTimeout(function() {...}, 100);

Для setInterval функция остаётся в памяти до тех пор, пока не будет вызван clearInterval.

Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

setTimeout с нулевой задержкой

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func).

Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода.

Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

setTimeout(() => alert("Мир"));alert("Привет");

Первая строка помещает вызов в «календарь» через 0 мс. Но планировщик проверит «календарь» только после того, как текущий код завершится. Поэтому "Привет" выводится первым, а "Мир" – после него.

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе Событийный цикл: микрозадачи и макрозадачи.

Минимальная задержка вложенных таймеров в браузере

В браузере есть ограничение на то, как часто внутренние счётчики могут выполняться. В стандарте HTML5 говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов setTimeout повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве times. Какова реальная задержка?

Первый таймер запускается сразу (как и указано в спецификации), а затем задержка вступает в игру, и мы видим 9, 15, 20, 24....

Аналогичное происходит при использовании setInterval вместо setTimeout: setInterval(f) запускает f несколько раз с нулевой задержкой, а затем с задержкой 4+ мс.

Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам.

Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, setImmediate для Node.js. Так что это ограничение относится только к браузерам.

Итого

Методы setInterval(func, delay, ...args) и setTimeout(func, delay, ...args) позволяют выполнять func регулярно или только один раз после задержки delay, заданной в мс.

Для отмены выполнения необходимо вызвать clearInterval/clearTimeout со значением, которое возвращают методы setInterval/setTimeout.

Вложенный вызов setTimeout является более гибкой альтернативой setInterval. Также он позволяет более точно задать интервал между выполнениями.

Планирование с нулевой задержкой setTimeout(func,0) или, что то же самое, setTimeout(func) используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.

Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами setTimeout, а также для setInterval, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

Перегружен процессор.

Вкладка браузера в фоновом режиме.

Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Классы в JavaScript.

Классы

Классы в JavaScript были введены в ECMAScript 2015 и представляют собой синтаксический сахар над существующим в JavaScript механизмом прототипного наследования. Синтаксис классов не вводит новую объектно-ориентированную модель, а предоставляет более простой и понятный способ создания объектов и организации наследования.

Определение классов

На самом деле классы — это "специальные функции", поэтому точно также, как вы определяете функции (function expressions и function declarations), вы можете определять и классы с помощью: class declarations и class expressions.

Объявление класса

Первый способ определения класса — class declaration (*объявление класса*). Для этого необходимо воспользоваться ключевым словом class и указать имя класса (в примере — «Rectangle»).

class Rectangle { constructor(height, width) { this.height = height; this.width = width; }}

Подъём (hoisting)

Разница между *объявлением функции* (*function declaration*) и *объявлением класса* (*class declaration*) в том, что *объявление функции* совершает подъём (hoisting), в то время как *объявление класса* — нет. Поэтому вначале необходимо объявить ваш класс и только затем работать с ним, а код же вроде следующего сгенерирует исключение типа ReferenceError:

var p = new Rectangle(); // ReferenceErrorclass Rectangle {}

Выражение класса

Второй способ определения класса — class expression (*выражение класса*). Можно создавать именованные и безымянные выражения. В первом случае имя выражения класса находится в локальной области видимости класса и может быть получено через свойства самого класса, а не его экземпляра.

// безымянныйvar Rectangle = class { constructor(height, width) { this.height = height; this.width = width; }}; console.log(Rectangle.name);// отобразится: "Rectangle"// именованныйvar Rectangle = class Rectangle2 { constructor(height, width) { this.height = height; this.width = width; }}; console.log(Rectangle.name);// отобразится: "Rectangle2"

Примечание:Обратите внимание: выражения класса подвержены тем же проблемам с подъёмом (hoisting), что и объявления класса.

Тело класса и задание методов

Тело класса — это часть кода, заключённая в фигурные скобки {}. Здесь вы можете объявлять члены класса, такие как методы и конструктор.

Строгий режим

Тела *объявлений классов* и *выражений классов* выполняются в строгом режиме (strict mode).

Constructor

Метод constructor — специальный метод, необходимый для создания и инициализации объектов, созданных, с помощью класса. В классе может быть только один метод с именем constructor. Исключение типа SyntaxError будет выброшено, если класс содержит более одного вхождения метода constructor.

Ключевое слово super можно использовать в методе constructor для вызова конструктора родительского класса.

Методы прототипа

Смотрите также определение методов.

class Rectangle { constructor(height, width) { this.height = height; this.width = width; } get area() { return this.calcArea(); } calcArea() { return this.height \* this.width; }}const square = new Rectangle(10, 10); console.log(square.area); // 100

Статические методы и свойства

Ключевое слово static, определяет статический метод или свойства для класса. Статические методы и свойства вызываются без инстанцирования (en-US) их класса, и не могут быть вызваны у экземпляров (*instance*) класса. Статические методы, часто используются для создания служебных функций для приложения, в то время как статические свойства полезны для кеширования в рамках класса, фиксированной конфигурации или любых других целей, не связанных с реплецированием данных между экземплярами.

Привязка this в прототипных и статических методах

Когда статический или прототипный метод вызывается без привязки к this объекта (или когда this является типом boolean, string, number, undefined, null), тогда this будет иметь значение undefined внутри вызываемой функции.

Определение полей

Предупреждение: Публичные и приватные поля - это экспериментальная особенность (stage 3), предложенная комитетом TC39 по стандартам языка Javascript. Поддержка браузерами ограничена, но это нововведение может быть использовано на моменте сборки, используя к примеру Babel.

Публичные поля

Используя Javascript синтаксис определения полей, приведённый выше пример может быть изменён следующим образом:

Как видно из примера, поля могут быть объявлены как со начальным значением, так и без него.

Более подробно об этом написано в публичные поля класса.

Приватные поля

Предыдущий пример может быть изменён следующим образом, используя приватные поля:

Приватные поля могут быть изменены или прочитаны только в рамках класса и не могут быть вызваны извне. Определяя вещи, которые не видны за пределами класса, вы гарантируете, что пользователи ваших классов не могут зависеть от внутренних компонентов, которые могут изменить версию на версию.

Примечание: Приватные поля могут быть объявлены только заранее в объявлении поля.

Приватные поля не могут быть созданы позже путём присваивания им значения, в отличии от обычных свойств.

Более подробно об этом написано в Приватные поля класса.

Наследование классов с помощью extends

Если в подклассе присутствует конструктор, он должен сначала вызвать super, прежде чем использовать this.

Обратите внимание, что классы не могут расширять обычные (non-constructible) объекты. Если вам необходимо создать наследование от обычного объекта, в качестве замены можно использовать Object.setPrototypeOf():

Species

Допустим, вам хотелось бы возвращать объекты типа Array в вашем производном от массива классе MyArray. Паттерн species позволяет вам переопределять конструкторы по умолчанию.

Например, при использовании таких методов, как map(), который возвращает конструктор по умолчанию, вам хотелось бы, чтобы они возвращали родительский объект Array вместо объекта MyArray. Символ Symbol.species позволяет это реализовать:

Абстрактные подклассы, или mix-ins, — это шаблоны для классов. У класса в ECMAScript может быть только один родительский класс, поэтому множественное наследование (к примеру, от tooling classes) невозможно. Функциональность должен предоставлять родительский класс.

Для реализации mix-ins в ECMAScript можно использовать функцию, которая в качестве аргумента принимает родительский класс, а возвращает подкласс, его расширяющий:

var calculatorMixin = Base => class extends Base { calc() { }};var randomizerMixin = Base => class extends Base { randomize() { }};

Класс, использующий такие mix-ins, можно описать следующим образом:

class Foo { }class Bar extends calculatorMixin(randomizerMixin(Foo)) { }

Тело класса и задание методов. Создание классов и экземпляров классов. Наследование и прототипы.

Классы

Классы в JavaScript были введены в ECMAScript 2015 и представляют собой синтаксический сахар над существующим в JavaScript механизмом прототипного наследования. Синтаксис классов не вводит новую объектно-ориентированную модель, а предоставляет более простой и понятный способ создания объектов и организации наследования.

Определение классов

На самом деле классы — это "специальные функции", поэтому точно также, как вы определяете функции (function expressions и function declarations), вы можете определять и классы с помощью: class declarations и class expressions.

Объявление класса

Первый способ определения класса — class declaration (*объявление класса*). Для этого необходимо воспользоваться ключевым словом class и указать имя класса (в примере — «Rectangle»).

class Rectangle { constructor(height, width) { this.height = height; this.width = width; }}

Подъём (hoisting)

Разница между *объявлением функции* (*function declaration*) и *объявлением класса* (*class declaration*) в том, что *объявление функции* совершает подъём (hoisting), в то время как *объявление класса* — нет. Поэтому вначале необходимо объявить ваш класс и только затем работать с ним, а код же вроде следующего сгенерирует исключение типа ReferenceError:

var p = new Rectangle(); // ReferenceErrorclass Rectangle {}

Выражение класса

Второй способ определения класса — class expression (*выражение класса*). Можно создавать именованные и безымянные выражения. В первом случае имя выражения класса находится в локальной области видимости класса и может быть получено через свойства самого класса, а не его экземпляра.

// безымянныйvar Rectangle = class { constructor(height, width) { this.height = height; this.width = width; }}; console.log(Rectangle.name);// отобразится: "Rectangle"// именованныйvar Rectangle = class Rectangle2 { constructor(height, width) { this.height = height; this.width = width; }}; console.log(Rectangle.name);// отобразится: "Rectangle2"

Примечание:Обратите внимание: выражения класса подвержены тем же проблемам с подъёмом (hoisting), что и объявления класса.

Тело класса и задание методов

Тело класса — это часть кода, заключённая в фигурные скобки {}. Здесь вы можете объявлять члены класса, такие как методы и конструктор.

Строгий режим

Тела *объявлений классов* и *выражений классов* выполняются в строгом режиме (strict mode).

Constructor

Метод constructor — специальный метод, необходимый для создания и инициализации объектов, созданных, с помощью класса. В классе может быть только один метод с именем constructor. Исключение типа SyntaxError будет выброшено, если класс содержит более одного вхождения метода constructor.

Ключевое слово super можно использовать в методе constructor для вызова конструктора родительского класса.

Методы прототипа

Статические методы и свойства

Ключевое слово static, определяет статический метод или свойства для класса. Статические методы и свойства вызываются без инстанцирования (en-US) их класса, и не могут быть вызваны у экземпляров (*instance*) класса. Статические методы, часто используются для создания служебных функций для приложения, в то время как статические свойства полезны для кеширования в рамках класса, фиксированной конфигурации или любых других целей, не связанных с реплецированием данных между экземплярами.

console.log(Point.displayName); // "Точка" console.log(Point.distance(p1, p2)); // 7.0710678118654755

Привязка this в прототипных и статических методах

Когда статический или прототипный метод вызывается без привязки к this объекта (или когда this является типом boolean, string, number, undefined, null), тогда this будет иметь значение undefined внутри вызываемой функции.

Свойства экземпляра

Свойства экземпляра должны быть определены в методе класса:

Статические (class-side) свойства и свойства прототипа должны быть определены за рамками тела класса:

Rectangle.staticWidth = 20;

Rectangle.prototype.prototypeWidth = 25;

Определение полей

Предупреждение: Публичные и приватные поля - это экспериментальная особенность (stage 3), предложенная комитетом TC39 по стандартам языка Javascript. Поддержка браузерами ограничена, но это нововведение может быть использовано на моменте сборки, используя к примеру Babel.

Публичные поля

Используя Javascript синтаксис определения полей, приведённый выше пример может быть изменён следующим образом:

Как видно из примера, поля могут быть объявлены как со начальным значением, так и без него.

Более подробно об этом написано в публичные поля класса.

Приватные поля

Предыдущий пример может быть изменён следующим образом, используя приватные поля:

Приватные поля могут быть изменены или прочитаны только в рамках класса и не могут быть вызваны извне. Определяя вещи, которые не видны за пределами класса, вы гарантируете, что пользователи ваших классов не могут зависеть от внутренних компонентов, которые могут изменить версию на версию.

Примечание: Приватные поля могут быть объявлены только заранее в объявлении поля.

Приватные поля не могут быть созданы позже путём присваивания им значения, в отличии от обычных свойств.

Более подробно об этом написано в Приватные поля класса.

Наследование классов с помощью extends

Ключевое слово extends используется в *объявлениях классов* и *выражениях классов* для создания класса, дочернего относительно другого класса.

Если в подклассе присутствует конструктор, он должен сначала вызвать super, прежде чем использовать this.

Обратите внимание, что классы не могут расширять обычные (non-constructible) объекты.

Допустим, вам хотелось бы возвращать объекты типа Array в вашем производном от массива классе MyArray. Паттерн species позволяет вам переопределять конструкторы по умолчанию.

Например, при использовании таких методов, как map(), который возвращает конструктор по умолчанию, вам хотелось бы, чтобы они возвращали родительский объект Array вместо объекта MyArray. Символ Symbol.species позволяет это реализовать:

class MyArray extends Array { // Изменить species на родительский конструктор Array static get [Symbol.species]() { return Array; }}var a = new MyArray(1,2,3);var mapped = a.map(x => x \* x); console.log(mapped instanceof MyArray); // false console.log(mapped instanceof Array); // true

Обращение к родительскому классу с помощью super

Ключевое слово super используется для вызова функций на родителе объекта.

class Cat { constructor(name) { this.name = name; } speak() { console.log(`${this.name} издаёт звук.`); }}class Lion extends Cat { speak() { super.speak(); console.log(`${this.name} рычит.`); }}let l = new Lion('Фаззи'); l.speak();// Фаззи издаёт звук.// Фаззи рычит.

Mix-ins

Абстрактные подклассы, или mix-ins, — это шаблоны для классов. У класса в ECMAScript может быть только один родительский класс, поэтому множественное наследование (к примеру, от tooling classes) невозможно. Функциональность должен предоставлять родительский класс.

Для реализации mix-ins в ECMAScript можно использовать функцию, которая в качестве аргумента принимает родительский класс, а возвращает подкласс, его расширяющий:

var calculatorMixin = Base => class extends Base { calc() { }};var randomizerMixin = Base => class extends Base { randomize() { }};

Класс, использующий такие mix-ins, можно описать следующим образом:

class Foo { }class Bar extends calculatorMixin(randomizerMixin(Foo)) { }

28 ноября 2022 г.

Класс: базовый синтаксис

В объектно-ориентированном программировании *класс* – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

*Википедия*

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё.

Как мы уже знаем из главы Конструктор, оператор "new", с этим может помочь new function.

Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

Синтаксис «class»

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

constructor запускается с заданным аргументом и сохраняет его в this.name.

…Затем можно вызывать на объекте методы, такие как user.sayHi().

Методы в классе не разделяются запятой

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке.

Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются.

Что такое класс?

Итак, что же такое class? Это не полностью новая языковая сущность, как может показаться на первый взгляд.

Давайте развеем всю магию и посмотрим, что такое класс на самом деле. Это поможет в понимании многих сложных аспектов.

В JavaScript класс – это разновидность функции.

Создаёт функцию с именем User, которая становится результатом объявления класса. Код функции берётся из метода constructor (она будет пустой, если такого метода нет).

Сохраняет все методы, такие как sayHi, в User.prototype.

При вызове метода объекта new User он будет взят из прототипа, как описано в главе F.prototype. Таким образом, объекты new User имеют доступ к методам класса.

На картинке показан результат объявления class User:

Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что мы можем сделать всё то же самое без конструкции class:

// перепишем класс User на чистых функциях// 1. Создаём функцию constructorfunction User(name) { this.name = name;}// каждый прототип функции имеет свойство constructor по умолчанию,// поэтому нам нет необходимости его создавать// 2. Добавляем метод в прототипUser.prototype.sayHi = function() { alert(this.name);};// Использование:let user = new User("Иван"); user.sayHi();

Результат этого кода очень похож. Поэтому, действительно, есть причины, по которым class можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа.

Однако есть важные отличия:

Во-первых, функция, созданная с помощью class, помечена специальным внутренним свойством [[IsClassConstructor]]: true. Поэтому это не совсем то же самое, что создавать её вручную.

В отличие от обычных функций, конструктор класса не может быть вызван без new:

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class …»

Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable вfalse для всех методов в "prototype".

И это хорошо, так как если мы проходимся циклом for..in по объекту, то обычно мы не хотим при этом получать методы класса.

Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

Также в дополнение к основной, описанной выше, функциональности, синтаксис class даёт ряд других интересных возможностей, с которыми мы познакомимся чуть позже.

Class Expression

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Аналогично Named Function Expression, Class Expression может иметь имя.

Если у Class Expression есть имя, то оно видно только внутри класса:

Мы даже можем динамически создавать классы «по запросу»:

Геттеры/сеттеры, другие сокращения

Как и в литеральных объектах, в классах можно объявлять вычисляемые свойства, геттеры/сеттеры и т.д.

При объявлении класса геттеры/сеттеры создаются на User.prototype, вот так:

Object.defineProperties(User.prototype, { name: { get() { return this.\_name }, set(name) { // ... } }});

Свойства классов

Старым браузерам может понадобиться полифил

Свойства классов добавлены в язык недавно.

В приведённом выше примере у класса User были только методы. Давайте добавим свойство:

class User { *name = "Аноним";* sayHi() { alert(`Привет, ${this.name}!`); }}new User().sayHi();

Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта.

**Создание объектов в JavaScript: способы создания, назначение свойств и методов объектов. Объектный тип Object.**

JavaScript спроектирован на основе простой парадигмы. В основе концепции лежат простые объекты. Объект — это набор свойств, и каждое свойство состоит из имени и значения, ассоциированного с этим именем. Значением свойства может быть функция, которую можно назвать *методом* объекта. В дополнение к встроенным в браузер объектам, вы можете определить свои собственные объекты. Эта глава описывает как пользоваться объектами, свойствами, функциями и методами, а также как создавать свои собственные объекты.

Обзор объектов

Объекты в JavaScript, как и во многих других языках программирования, похожи на объекты реальной жизни. Концепцию объектов JavaScript легче понять, проводя параллели с реально существующими в жизни объектами.

В JavaScript объект — это самостоятельная единица, имеющая свойства и определённый тип. Сравним, например, с чашкой. У чашки есть цвет, форма, вес, материал, из которого она сделана, и т.д. Точно так же, объекты JavaScript имеют свойства, которые определяют их характеристики.

Объекты и свойства

В JavaScript объект имеет свойства, ассоциированные с ним. Свойство объекта можно понимать как переменную, закреплённую за объектом. Свойства объекта в сущности являются теми же самыми переменными JavaScript, за тем исключением, что они закреплены за объектом. Свойства объекта определяют его характеристики. Получить доступ к свойству объекта можно с помощью точечной записи:

objectName.propertyName

Как и все переменные JavaScript, имя объекта (которое тоже может быть переменной) и имя свойства являются чувствительными к регистру. Вы можете определить свойство указав его значение. Например, давайте создадим объект myCar и определим его свойства make, model, и year следующим образом:

var myCar = new Object(); myCar.make = "Ford"; myCar.model = "Mustang"; myCar.year = 1969;

Неопределённые свойства объекта являются undefined (а не null).

myCar.color; // undefined

Свойства объектов JavaScript также могут быть доступны или заданы с использованием скобочной записи (более подробно см. property accessors). Объекты иногда называются *ассоциативными массивами*, поскольку каждое свойство связано со строковым значением, которое можно использовать для доступа к нему. Так, например, вы можете получить доступ к свойствам объекта myCar следующим образом:

myCar["make"] = "Ford"; myCar["model"] = "Mustang"; myCar["year"] = 1969;

Имена свойств объекта могут быть строками JavaScript, или тем, что может быть сконвертировано в строку, включая пустую строку. Как бы то ни было, доступ к любому имени свойства, которое содержит невалидный JavaScript идентификатор (например, имя свойства содержит в себе пробел и тире или начинается с цифры), может быть получен с использованием квадратных скобок. Этот способ записи также полезен, когда имена свойств должны быть динамически определены (когда имя свойства не определено до момента исполнения). Примеры далее:

var myObj = new Object(), str = "myString", rand = Math.random(), obj = new Object(); myObj.type = "Dot syntax"; myObj["date created"] = "String with space"; myObj[str] = "String value"; myObj[rand] = "Random Number"; myObj[obj] = "Object"; myObj[""] = "Even an empty string"; console.log(myObj);

Обратите внимание, что все ключи с квадратными скобками преобразуются в тип String, поскольку объекты в JavaScript могут иметь в качестве ключа только тип String. Например, в приведённом выше коде, когда ключ obj добавляется в myObj, JavaScript вызывает метод obj.toString () и использует эту результирующую строку в качестве нового ключа.

Вы также можете получить доступ к свойствам, используя значение строки, которое хранится в переменной:

var propertyName = "make"; myCar[propertyName] = "Ford"; propertyName = "model"; myCar[propertyName] = "Mustang";

Вы можете пользоваться квадратными скобками в конструкции for...in чтобы выполнить итерацию всех свойств объекта, для которых она разрешена. Чтобы показать как это работает, следующая функция показывает все свойства объекта, когда вы передаёте в неё сам объект и его имя как аргументы функции:

function showProps(obj, objName) { var result = ""; for (var i in obj) { if (obj.hasOwnProperty(i)) { result += objName + "." + i + " = " + obj[i] + "\n"; } } return result;}

Так что если вызвать эту функцию вот так showProps(myCar, "myCar"), то получим результат:

myCar.make = Ford myCar.model = Mustang myCar.year = 1969

Перечисление всех свойств объекта

Начиная с ECMAScript 5, есть три способа перечислить все свойства объекта (получить их список):

циклы for...in (en-US) Этот метод перебирает все перечисляемые свойства объекта и его цепочку прототипов

Object.keys(o) (en-US) Этот метод возвращает массив со всеми собственными (те, что в цепочке прототипов, не войдут в массив) именами перечисляемых свойств объекта o.

Object.getOwnPropertyNames(o) (en-US) Этот метод возвращает массив содержащий все имена своих свойств (перечисляемых и неперечисляемых) объекта o.

До ECMAScript 5 не было встроенного способа перечислить все свойства объекта. Однако это можно сделать с помощью следующей функции:

function listAllProperties(o){ var objectToInspect; var result = []; for(objectToInspect = o; objectToInspect !== null; objectToInspect = Object.getPrototypeOf(objectToInspect)){ result = result.concat(Object.getOwnPropertyNames(objectToInspect)); } return result;}

Это может быть полезно для обнаружения скрытых (hidden) свойств (свойства в цепочке прототипа, которые недоступны через объект, в случае, если другое свойство имеет такое же имя в предыдущем звене из цепочки прототипа). Перечислить доступные свойства можно, если удалить дубликаты из массива.

Создание новых объектов

JavaScript содержит набор встроенных объектов. Также вы можете создавать свои объекты. Начиная с JavaScript 1.2, вы можете создавать объект с помощью инициализатора объекта. Другой способ — создать функцию-конструктор и сделать экземпляр объекта с помощью этой функции и оператора new.

Использование инициализаторов объекта

Помимо создания объектов с помощью функции-конструктора вы можете создавать объекты и другим, особым способом. Фактически, вы можете записать объект синтаксически, и он будет создан интерпретатором автоматически во время выполнения. Эта синтаксическая схема приведена ниже:

var obj = { property\_1: value\_1, // property\_# may be an identifier... 2: value\_2, // or a number... // ..., "property n": value\_n }; // or a string

здесь obj — это имя нового объекта, каждое property\_i — это идентификатор (имя, число или строковый литерал), и каждый value\_i — это значения, назначенные *property\_i*. Имя obj и ссылка объекта на него необязательна; если далее вам не надо будет ссылаться на данный объект, то вам не обязательно назначать объект переменной. (Обратите внимание, что вам потребуется обернуть литерал объекта в скобки, если объект находится в месте, где ожидается инструкция, чтобы интерпретатор не перепутал его с блоком.)

Если объект создан при помощи инициализатора объектов на высшем уровне скрипта, то JavaScript интерпретирует объект каждый раз, когда анализирует выражение, содержащее объект, записанный как литерал. Плюс, если пользоваться функцией инициализатором, то он будет создаваться каждый раз, когда функция вызывается.

Следующая инструкция создаёт объект и назначает его переменной x, когда выражение cond истинно.

if (cond) var x = {hi: "there"};

Следующий пример создаёт объект myHonda с тремя свойствами. Заметьте, что свойство engine — это также объект со своими собственными свойствами.

var myHonda = { color: "red", wheels: 4, engine: { cylinders: 4, size: 2.2 }};

Вы также можете использовать инициализатор объекта для создания массивов. Смотрите array literals.

До JavaScript 1.1 не было возможности пользоваться инициализаторами объекта. Единственный способ создавать объекты — это пользоваться функциями-конструкторами или функциями других объектов, предназначенных для этой цели. Смотрите Using a constructor function.

Использование функции конструктора

Другой способ создать объект в два шага описан ниже:

Определите тип объекта, написав функцию-конструктор. Название такой функции, как правило, начинается с заглавной буквы.

Создайте экземпляр объекта с помощью ключевого слова new.

Чтобы определить тип объекта создайте функцию, которая определяет тип объекта, его имя, свойства и методы. Например предположим, что вы хотите создать тип объекта для описания машин. Вы хотите, чтобы объект этого типа назывался car, и вы хотите, чтобы у него были свойства make, model, и year. Чтобы сделать это, напишите следующую функцию:

function Car(make, model, year) { this.make = make; this.model = model; this.year = year;}

Заметьте, что используется this чтобы присвоить значения (переданные как аргументы функции) свойствам объекта.

Теперь вы можете создать объект, называемый mycar, следующим образом:

var mycar = new Car("Eagle", "Talon TSi", 1993);

Эта инструкция создаёт объект типа Car со ссылкой mycar и присваивает определённые значения его свойствам. Значением mycar.make станет строка "Eagle", mycar.year — это целое число 1993, и так далее.

Вы можете создать столько объектов car, сколько нужно, просто вызывая new. Например:

var kenscar = new Car("Nissan", "300ZX", 1992);var vpgscar = new Car("Mazda", "Miata", 1990);

Объект может иметь свойство, которое будет другим объектом. Например, далее определяется объект типа Person следующим образом:

function Person(name, age, sex) { this.name = name; this.age = age; this.sex = sex;}

и затем создать два новых экземпляра объектов Person как показано далее:

var rand = new Person("Rand McKinnon", 33, "M");var ken = new Person("Ken Jones", 39, "M");

Затем, вы можете переписать определение car и включить в него свойство owner, которому назначить объект person следующим образом:

function Car(make, model, year, owner) { this.make = make; this.model = model; this.year = year; this.owner = owner;}

Затем, чтобы создать экземпляры новых объектов, выполните следующие инструкции:

var car1 = new Car("Eagle", "Talon TSi", 1993, rand);var car2 = new Car("Nissan", "300ZX", 1992, ken);

Заметьте, что вместо того, чтобы передавать строку, литерал или целое число при создании новых объектов, в выражениях выше передаются объекты rand и ken как аргумент функции. Теперь, если вам нужно узнать имя владельца car2, это можно сделать следующим образом:

car2.owner.name

Заметьте, что в любое время вы можете добавить новое свойство ранее созданному объекту. Например, выражение

car1.color = "black";

добавляет свойство color к car1, и устанавливает его значение равным "black." Как бы там ни было, это не влияет на любые другие объекты. Чтобы добавить новое свойство всем объектам одного типа, вы должны добавить свойство в определение типа объекта car.

Использование метода Object.create

Объекты также можно создавать с помощью метода Object.create. Этот метод очень удобен, так как позволяет вам указывать объект прототип для нового вашего объекта без определения функции конструктора.

// список свойств и методов для Animalvar Animal = { type: 'Invertebrates', // Значение type по умолчанию displayType: function() { // Метод отображающий тип объекта Animal console.log(this.type); }};// Создаём объект Animalvar animal1 = Object.create(Animal); animal1.displayType(); // Выведет:Invertebrates// Создаём объект Animal и присваиваем ему type = Fishesvar fish = Object.create(Animal); fish.type = 'Fishes'; fish.displayType(); // Выведет:Fishes

Наследование

Все объекты в JavaScript наследуются как минимум от другого объекта. Объект, от которого произошло наследование называется прототипом, и унаследованные свойства могут быть найдены в объекте prototype конструктора.

Индексы свойств объекта

В JavaScript 1.0 вы можете сослаться на свойства объекта либо по его имени, либо по его порядковому индексу. В JavaScript 1.1 и позже, если вы изначально определили свойство по имени, вы всегда должны ссылаться на него по его имени, и если вы изначально определили свойство по индексу, то должны ссылаться на него по его индексу.

Это ограничение налагается когда вы создаёте объект и его свойства с помощью функции конструктора (как мы это делали ранее с типом *Car* ) и когда вы определяете индивидуальные свойства явно (например, myCar.color = "red"). Если вы изначально определили свойство объекта через индекс, например myCar[5] = "25 mpg", то впоследствии сослаться на это свойство можно только так myCar[5].

Исключение из правил — объекты, отображаемые из HTML, например массив forms. Вы всегда можете сослаться на объекты в этих массивах или используя их индекс (который основывается на порядке появления в HTML документе), или по их именам (если таковые были определены). Например, если второй html-тег <FORM> в документе имеет значение атрибута NAME равное "myForm", вы можете сослаться на эту форму вот так: document.forms[1] или document.forms["myForm"] или document.myForm.

Определение свойств для типа объекта

Вы можете добавить свойство к ранее определённому типу объекта воспользовавшись специальным свойством prototype. Через prototype создаётся свойство, единое для всех объектов данного типа, а не одного экземпляра этого типа объекта. Следующий код демонстрирует это, добавляя свойство color ко всем объектам типа car, а затем присваивая значение свойству color объекта car1.

Car.prototype.color = null; car1.color = "black";

Смотрите свойство prototype(en-US) объекта Function в Справочнике JavaScript для получения деталей.

Определение методов

*Метод* — это функция, ассоциированная с объектом или, проще говоря, метод — это свойство объекта, являющееся функцией. Методы определяются так же, как и обычные функции, за тем исключением, что они присваиваются свойству объекта. Например вот так:

objectName.methodname = function\_name;var myObj = { myMethod: function(params) { // ...do something }};

где objectName — это существующий объект, methodname — это имя, которое вы присваиваете методу, и function\_name — это имя самой функции.

Затем вы можете вызвать метод в контексте объекта следующим образом:

object.methodname(params);

Вы можете определять методы для типа объекта, включая определение метода в функцию конструктора объекта. Например, вы можете определить функцию, которая форматирует и отображает свойства до этого определённых объектов car. Например,

function displayCar() { var result = "A Beautiful " + this.year + " " + this.make + " " + this.model; pretty\_print(result);}

где pretty\_print — это функция отображения горизонтальной линии и строки. Заметьте, что использование this позволяет ссылаться на объект, которому принадлежит метод.

Вы можете сделать эту функцию методом car, добавив инструкцию

this.displayCar = displayCar;

к определению объекта. Таким образом, полное определение car примет следующий вид:

function Car(make, model, year, owner) { this.make = make; this.model = model; this.year = year; this.owner = owner; this.displayCar = displayCar;}

Теперь вы можете вызвать метод displayCar для каждого из объектов как показано ниже:

car1.displayCar(); car2.displayCar();

Использование this для ссылки на объект

В JavaScript есть специальное ключевое слово this, которое вы можете использовать внутри метода, чтобы ссылаться на текущий объект. Предположим, у вас есть функция validate, которая сверяет свойство value, переданного ей объекта с некоторыми верхним и нижним значениями:

function validate(obj, lowval, hival) { if ((obj.value < lowval) || (obj.value > hival)) alert("Invalid Value!");}

Вы можете вызвать эту функцию validate в каждом элементе формы, в обработчике события onchange. Используйте this для доступа к этому элементу, как это сделано ниже:

<input type="text" name="age" size="3" onChange="validate(this, 18, 99)">

В общем случае, this ссылается на объект, вызвавший метод.

Через this можно обратиться и к родительской форме элемента, воспользовавшись свойством form. В следующем примере форма myForm содержит элемент ввода Text и кнопку button1. Когда пользователь нажимает кнопку, значению объекта Text назначается имя формы. Обработчик событий кнопки onclick пользуется this.form чтобы сослаться на текущую форму, myForm.

<form name="myForm"><p><label>Form name:<input type="text" name="text1" value="Beluga"></label><p><input name="button1" type="button" value="Show Form Name" onclick="this.form.text1.value = this.form.name"></p></form>

Определение геттеров и сеттеров

*Геттер(от англ. get - получить)* — это метод, который получает значение определённого свойства. \_Сеттер (от англ. set\* — \*присвоить) — это метод, который присваивает значение определённому свойству объекта. Вы можете определить геттеры и сеттеры для любых из встроенных или определённых вами объектов, которые поддерживают добавление новых свойств. Синтаксис определения геттеров и сеттеров использует литеральный синтаксис объектов.

Ниже проиллюстрировано, как могут работать геттеры и сеттеры в объекте определённом пользователем:

var o = { a: 7, get b() { return this.a + 1; }, set c(x) { this.a = x / 2; }}; console.log(o.a); // 7 console.log(o.b); // 8 o.c = 50; console.log(o.a); // 25

Объект o получит следующие свойства:

o.a — число

o.b — геттер, который возвращает o.a плюс 1

o.c — сеттер, который присваивает значение o.a половине значения которое передано в o.c

Следует особо отметить, что имена функций, указанные в литеральной форме "[gs]et *propertyName*() { }" не будут в действительности являться именами геттера и сеттера. Чтобы задать в качестве геттера и сеттера функции с явно определёнными именами, используйте метод Object.defineProperty (или его устаревший аналог Object.prototype.\_\_defineGetter\_\_).

В коде ниже показано, как с помощью геттера и сеттера можно расширить прототип объекта Date и добавить ему свойство year, которое будет работать у всех экземпляров класса Date. Этот код использует существующие методы класса Date - getFullYear и setFullYear для работы геттера и сеттера.

Определение геттера и сеттера для свойства year:

var d = Date.prototype; Object.defineProperty(d, 'year', { get: function() { return this.getFullYear(); }, set: function(y) { this.setFullYear(y); }});

Использование свойства year заданного геттером и сеттером:

var now = new Date(); console.log(now.year); // 2000 now.year = 2001; // 987617605170 console.log(now);// Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001

В принципе, геттеры и сеттеры могут быть либо:

определены при использовании Инициализаторов объекта, или

добавлены существующему объекту в любой момент, при использовании методов добавления геттеров и сеттеров.

Когда определение геттера и сеттера использует инициализаторы объекта, всё что вам нужно, это дополнить геттер префиксом get а сеттер префиксом set. При этом, метод геттера не должен ожидать каких либо параметров, в то время как метод сеттера принимает один единственный параметр (новое значение для присвоения свойству). Например:

var o = { a: 7, get b() { return this.a + 1; }, set c(x) { this.a = x / 2; }};

Геттеры и сеттеры, могут быть добавлены существующему объекту в любой момент, при помощи метода Object.defineProperties. Первый параметр этого метода - объект, которому вы хотите присвоить геттер и сеттер. Второй параметр - это объект, имена свойств которого будут соответствовать именам создаваемых свойств, а значения - объекты определяющие геттер и сеттер создаваемых свойств. В следующем примере создаются в точности такие же геттер и сеттер, как и в примере выше:

var o = { a: 0 }; Object.defineProperties(o, { 'b': { get: function() { return this.a + 1; } }, 'c': { set: function(x) { this.a = x / 2; } }}); o.c = 10; // Запускает сеттер, который присваивает 10 / 2 (5) свойству 'a' console.log(o.b); // Запускает геттер, который возвращает a + 1 (тоесть 6)

То, какую из двух форм использовать для определения свойств, зависит от вашего стиля программирования и стоящей перед вами задачи. Если вы уже используете инициализатор объекта для определения прототипа, то, скорее всего, в большинстве случаев, вы воспользуетесь первой формой. Она более компактна и естественна. Однако, не редко, вторая форма является единственно возможной, в случаях, когда вы работаете с существующим объектом без доступа к его определению. Вторая форма наилучшим образом отражает динамическую природу JavaScript — но может сделать код сложным для чтения и понимания.

Удаление свойств

Вы можете удалить свойство используя оператор delete. Следующий код показывает как удалить свойство.

//Creates a new object, myobj, with two properties, a and b.var myobj = new Object; myobj.a = 5; myobj.b = 12;//Removes the a property, leaving myobj with only the b property.delete myobj.a;

Вы также можете воспользоваться delete чтобы удалить глобальную переменную, если ключевое слово var не было использовано при её объявлении:

g = 17;delete g;

Смотри [delete](Expressions\_and\_operators#delete) чтобы получить дополнительную информацию.

Сравнение объектов

В JavaScript объекты имеют ссылочный тип. Два отдельных объекта никогда не будут равными, даже если они имеют равный набор свойств. Только сравнение двух ссылок на один и тот же объект вернёт true.

// Две переменных ссылаются на два объекта с одинаковыми свойствамиvar fruit = {name: 'apple'};var fruitbear = {name: 'apple'}; fruit == fruitbear; // вернёт false fruit === fruitbear; // вернёт false

// Две переменные ссылаются на один общий объектvar fruit = {name: 'apple'};var fruitbear = fruit; // присвоим переменной fruitbear ссылку на объект fruit// теперь fruitbear и fruit ссылаются на один и тот же объект fruit == fruitbear; // вернёт true fruit === fruitbear; // вернёт true

fruit.name = 'grape'; console.log(fruitbear); // вернёт { name: "grape" } вместо { name: "apple" }

Подробнее смотрите Операторы сравнения (en-US).

Объекты

Как мы знаем из главы Типы данных, в JavaScript существует 8 типов данных. Семь из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое).

Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком *свойств*. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо.

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"let user = {}; // синтаксис "литерал объекта"

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом объекта* или *литеральной нотацией*.

Литералы и свойства

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект name: "John", // под ключом "name" хранится значение "John" age: 30 // под ключом "age" хранится значение 30};

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

Первое свойство с именем "name" и значением "John".

Второе свойство с именем "age" и значением 30.

Можно сказать, что наш объект user – это ящик с двумя папками, подписанными «name» и «age».

Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:alert( user.name ); // Johnalert( user.age ); // 30

Значение может быть любого типа. Давайте добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства мы можем использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

let user = { name: "John", age: 30, "likes birds": true // имя свойства из нескольких слов должно быть в кавычках};

Последнее свойство объекта может заканчиваться запятой:

let user = { name: "John", age: 30*,*}

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

Объект, объявленный как константа, может быть изменён

Объект, объявленный через const, *может* быть изменён.

Например:

const user = { name: "John"};*user.name = "Pete"; // (\*)*alert(user.name); // Pete

Может показаться, что строка (\*) должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление const защищает от изменений только саму переменную user, а не её содержимое.

Определение const выдаст ошибку только если мы присвоим переменной другое значение: user=....

Есть ещё один способ сделать константами свойства объекта, который мы рассмотрим в главе Флаги и дескрипторы свойств.

Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

// это вызовет синтаксическую ошибку user.likes birds = true

JavaScript видит, что мы обращаемся к свойству user.likes, а затем идёт непонятное слово birds. В итоге синтаксическая ошибка.

Точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и \_.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

let user = {};// присваивание значения свойству user["likes birds"] = true;// получение значения свойстваalert(user["likes birds"]); // true// удаление свойстваdelete user["likes birds"];

Сейчас всё в порядке. Обратите внимание, что строка в квадратных скобках заключена в кавычки (подойдёт любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

let key = "likes birds";// то же самое, что и user["likes birds"] = true; user[key] = true;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого мы используем её для доступа к свойству. Это даёт нам большую гибкость.

Вычисляемые свойства

Мы можем использовать квадратные скобки в литеральной нотации для создания *вычисляемого свойства*.

Пример:

let fruit = prompt("Какой фрукт купить?", "apple");let bag = { *[fruit]: 5, // имя свойства будет взято из переменной fruit*};alert( bag.apple ); // 5, если fruit="apple"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

И если посетитель введёт слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

По сути, пример выше работает так же, как и следующий пример:

let fruit = prompt("Какой фрукт купить?", "apple");let bag = {};// имя свойства будет взято из переменной fruit bag[fruit] = 5;

…Но первый пример выглядит лаконичнее.

Мы можем использовать и более сложные выражения в квадратных скобках:

let fruit = 'apple';let bag = { [fruit + 'Computers']: 5 // bag.appleComputers = 5};

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

Подведём итог: в большинстве случаев, когда имена свойств известны и просты, используется запись через точку. Если же нам нужно что-то более сложное, то мы используем квадратные скобки.

Свойство из переменной

В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем.

Например:

function makeUser(name, age) { return { name: name, age: age // ...другие свойства };}let user = makeUser("John", 30);alert(user.name); // John

В примере выше название свойств name и age совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств. Такой подход настолько распространён, что существуют специальные *короткие свойства* для упрощения этой записи.

Вместо name:name мы можем написать просто name:

function makeUser(name, age) { *return { name, // то же самое, что и name: name age // то же самое, что и age: age // ... };*}

Мы можем использовать как обычные свойства, так и короткие в одном и том же объекте:

let user = { name, // тоже самое, что и name:name age: 30};

Ограничения на имена свойств

Как мы уже знаем, имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д.

Но для свойств объекта такого ограничения нет:

// эти имена свойств допустимыlet obj = { for: 1, let: 2, return: 3};alert( obj.for + obj.let + obj.return ); // 6

Иными словами, нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов (специальный тип для идентификаторов, который будет рассмотрен позже).

Все другие типы данных будут автоматически преобразованы к строке.

Как мы видим, присвоение примитивного значения 5 игнорируется.

Мы более подробно исследуем особенности свойства \_\_proto\_\_ в следующих главах Прототипное наследование, а также предложим способы исправления такого поведения.

Проверка существования свойства, оператор «in»

В отличие от многих других языков, особенность JavaScript-объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет!

При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства:

Также существует специальный оператор "in" для проверки существования свойства в объекте.

Синтаксис оператора:

"key" in object

Для чего вообще нужен оператор in? Разве недостаточно сравнения с undefined?

В большинстве случаев прекрасно сработает сравнение с undefined. Но есть особый случай, когда оно не подходит, и нужно использовать "in".

Это когда свойство существует, но содержит значение undefined:

let obj = { test: undefined};alert( obj.test ); // выведет undefined, значит свойство не существует?alert( "test" in obj ); // true, свойство существует!

В примере выше свойство obj.test технически существует в объекте. Оператор in сработал правильно.

Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств мы используем значение null. Таким образом, оператор in является экзотическим гостем в коде.

Цикл "for..in"

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;).

Синтаксис:

for (key in object) { // тело цикла выполняется для каждого свойства объекта}

К примеру, давайте выведем все свойства объекта user:

let user = { name: "John", age: 30, isAdmin: true};for (let key in user) { // ключи alert( key ); // name, age, isAdmin // значения ключей alert( user[key] ); // John, 30, true}

Обратите внимание, что все конструкции «for» позволяют нам объявлять переменную внутри цикла, как, например, let key здесь.

Кроме того, мы могли бы использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

Упорядочение свойств объекта

Упорядочены ли свойства объекта? Другими словами, если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? Можем ли мы на это рассчитывать?

Короткий ответ: свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания.

Итого

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

Ключи свойств должны быть строками или символами (обычно строками).

Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

Запись через точку: obj.property.

Квадратные скобки obj["property"]. Квадратные скобки позволяют взять ключ из переменной, например, obj[varWithKey].

Дополнительные операторы:

Удаление свойства: delete obj.prop.

Проверка существования свойства: "key" in obj.

Перебор свойств объекта: цикл for for (let key in obj).

То, что мы изучали в этой главе, называется «простым объектом» («plain object») или просто Object.

В JavaScript есть много других типов объектов:

Array для хранения упорядоченных коллекций данных,

Date для хранения информации о дате и времени,

Error для хранения информации об ошибке.

… и так далее.

У них есть свои особенности, которые мы изучим позже. Иногда люди говорят что-то вроде «тип данных Array» или «тип данных Date», но формально они не являются отдельными типами, а относятся к типу данных Object. Они лишь расширяют его различными способами.

Объекты в JavaScript очень мощные. Здесь мы только немного углубились в действительно огромную тему. Мы будем плотно работать с объектами и узнаем о них больше в следующих частях учебника.

Контекст выполнения функции: ключевое слово this. Методы вызова и определения контекста.

Контекст выполнения

Контекст выполнения (execution context) — это, если говорить упрощённо, концепция, описывающая окружение, в котором производится выполнение кода на JavaScript. Код всегда выполняется внутри некоего контекста.

▍Типы контекстов выполнения

В JavaScript существует три типа контекстов выполнения:

Глобальный контекст выполнения. Это базовый, используемый по умолчанию контекст выполнения. Если некий код находится не внутри какой-нибудь функции, значит этот код принадлежит глобальному контексту. Глобальный контекст характеризуется наличием глобального объекта, которым, в случае с браузером, является объект window, и тем, что ключевое слово this указывает на этот глобальный объект. В программе может быть лишь один глобальный контекст.

Контекст выполнения функции. Каждый раз, когда вызывается функция, для неё создаётся новый контекст. Каждая функция имеет собственный контекст выполнения. В программе может одновременно присутствовать множество контекстов выполнения функций. При создании нового контекста выполнения функции он проходит через определённую последовательность шагов, о которой мы поговорим ниже.

Контекст выполнения функции eval. Код, выполняемый внутри функции eval, также имеет собственный контекст выполнения. Однако функцией eval пользуются очень редко, поэтому здесь мы об этом контексте выполнения говорить не будем.

Стек выполнения

Стек выполнения (execution stack), который ещё называют стеком вызовов (call stack), это LIFO-стек, который используется для хранения контекстов выполнения, создаваемых в ходе работы кода.

Когда JS-движок начинает обрабатывать скрипт, движок создаёт глобальный контекст выполнения и помещает его в текущий стек. При обнаружении команды вызова функции движок создаёт новый контекст выполнения для этой функции и помещает его в верхнюю часть стека.

Движок выполняет функцию, контекст выполнения которой находится в верхней части стека. Когда работа функции завершается, её контекст извлекается из стека и управление передаётся тому контексту, который находится в предыдущем элементе стека.

Изучим эту идею с помощью следующего примера:

Когда вышеприведённый код загружается в браузер, JavaScript-движок создаёт глобальный контекст выполнения и помещает его в текущий стек вызовов. При выполнении вызова функции first() движок создаёт для этой функции новый контекст и помещает его в верхнюю часть стека.

При вызове функции second() из функции first() для этой функции создаётся новый контекст выполнения и так же помещается в стек. После того, как функция second() завершает работу, её контекст извлекается из стека и управление передаётся контексту выполнения, находящемуся в стеке под ним, то есть, контексту функции first().

Когда функция first() завершает работу, её контекст извлекается из стека и управление передаётся глобальному контексту. После того, как весь код оказывается выполненным, движок извлекает глобальный контекст выполнения из текущего стека.

О создании контекстов и о выполнении кода

До сих пор мы говорили о том, как JS-движок управляет контекстами выполнения. Теперь поговорим о том, как контексты выполнения создаются, и о том, что с ними происходит после создания. В частности, речь идёт о  стадии создания контекста выполнения и о стадии выполнения кода.

▍Стадия создания контекста выполнения

Перед выполнением JavaScript-кода создаётся контекст выполнения. В процессе его создания выполняются три действия:

Определяется значение this и осуществляется привязка this (this binding).

Создаётся компонент LexicalEnvironment (лексическое окружение).

Создаётся компонент VariableEnvironment (окружение переменных).

В глобальном контексте выполнения this содержит ссылку на глобальный объект (как уже было сказано, в браузере это объект window).

В контексте выполнения функции значение this зависит от того, как именно была вызвана функция. Если она вызвана в виде метода объекта, тогда значение this привязано к этому объекту. В других случаях this привязывается к глобальному объекту или устанавливается в undefined (в строгом режиме).

В соответствии со спецификацией ES6, лексическое окружение (Lexical Environment) — это термин, который используется для определения связи между идентификаторами и отдельными переменными и функциями на основе структуры лексической вложенности ECMAScript-кода. Лексическое окружение состоит из записи окружения (Environment Record) и ссылки на внешнее лексическое окружение, которая может принимать значение null.

Проще говоря, лексическое окружение — это структура, которая хранит сведения о соответствии идентификаторов и переменных. Под «идентификатором» здесь понимается имя переменной или функции, а под «переменной» — ссылка на конкретный объект (в том числе — на функцию) или примитивное значение.

В лексическом окружении имеется два компонента:

Запись окружения. Это место, где хранятся объявления переменных и функций.

Ссылка на внешнее окружение. Наличие такой ссылки говорит о том, что у лексического окружения есть доступ к родительскому лексическому окружению (области видимости).

Существует два типа лексических окружений:

Глобальное окружение (или глобальный контекст выполнения) — это лексическое окружение, у которого нет внешнего окружения. Ссылка глобального окружения на внешнее окружение представлена значением null. В глобальном окружении (в записи окружения) доступны встроенные сущности языка (такие, как Object, Array, и так далее), которые связаны с глобальным объектом, там же находятся и глобальные переменные, определённые пользователем. Значение this в этом окружении указывает на глобальный объект.

Окружение функции, в котором, в записи окружения, хранятся переменные, объявленные пользователем. Ссылка на внешнее окружение может указывать как на глобальный объект, так и на внешнюю по отношении к рассматриваемой функции функцию.

Существует два типа записей окружения:

Декларативная запись окружения, которая хранит переменные, функции и параметры.

Объектная запись окружения, которая используется для хранения сведений о переменных и функциях в глобальном контексте.

В результате, в глобальном окружении запись окружения представлена объектной записью окружения, а в окружении функции — декларативной записью окружения.

Обратите внимание на то, что в окружении функции декларативная запись окружения, кроме того, содержит объект arguments, который хранит соответствия между индексами и значениями аргументов, переданных функции, и сведения о количестве таких аргументов.

Окружение переменных

Окружение переменных (Variable Environment) — это тоже лексическое окружение, запись окружения которого хранит привязки, созданные посредством команд объявления переменных (VariableStatement) в текущем контексте выполнения.

Так как окружение переменных также является лексическим окружением, оно обладает всеми вышеописанными свойствами лексического окружения.

В ES6 существует одно различие между компонентами LexicalEnvironment и VariableEnvironment. Оно заключается в том, что первое используется для хранения объявлений функций и переменных, объявленных с помощью ключевых слов let и const, а второе — только для хранения привязок переменных, объявленных с использованием ключевого слова var.

Как вы, вероятно, заметили, переменные и константы, объявленные с помощью ключевых слов let и const, не имеют связанных с ними значений, а переменным, объявленным с помощью ключевого слова var, назначено значение undefined.

Это так из-за того, что во время создания контекста в коде осуществляется поиск объявлений переменных и функций, при этом объявления функций целиком хранятся в окружении. Значения переменных, при использовании var, устанавливаются в undefined, а при использовании let или const остаются неинициализированными.

Именно поэтому можно получить доступ к переменным, объявленным с помощью var, до их объявления (хотя они и будут иметь значение undefined), но, при попытке доступа к переменным или константам, объявленным с помощью let и const, выполняемой до их объявления, возникает ошибка.

Только что мы только что описали, называется «поднятием переменных» (Hoisting). Объявления переменных «поднимаются» в верхнюю часть их лексической области видимости до выполнения операций присвоения им каких-либо значений.

▍Стадия выполнения кода

Это, пожалуй, самая простая часть данного материала. На этой стадии выполняется присвоение значений переменным и осуществляется выполнение кода.

Обратите внимание на то, что если в процессе выполнения кода JS-движок не сможет найти в месте объявления значение переменной, объявленной с помощью ключевого слова let, он присвоит этой переменной значение undefined.

Итоги

Только что мы обсудили внутренние механизмы выполнения JavaScript-кода. Хотя для того, чтобы быть очень хорошим JS-разработчиком, знать всё это и не обязательно, если у вас имеется некоторое понимание вышеописанных концепций, это поможет вам лучше и глубже разобраться с другими механизмами языка, с такими, как поднятие переменных, области видимости, замыкания.

Стандартные объекты в JavaScript: назначение, свойства и методы объектов.

Эта глава описывает все стандартные встроенные объекты JavaScript, вместе с их методами и свойствами.

Термин «глобальные объекты» (или стандартные встроенные объекты) здесь не следует путать с самим *глобальным объектом*. Далее в главе под этим обозначением понимаются *объекты в глобальном пространстве имён* (но только, если не используется строгий режим ECMAScript 5! В противном случае они возвращают undefined). Сам же *глобальный объект* в глобальном пространстве имён доступен через оператор this. На самом деле, глобальное пространство имён *состоит из* свойств глобального объекта (в том числе включая унаследованные свойства, если они есть).

Другие объекты в глобальном пространстве имён создаются либо с помощью пользовательских скриптов, либо предоставляются приложением хоста. Объекты хоста, доступные в контекстах браузера, задокументированы в справке по API (en-US). Для получения дополнительной информации о различии между DOM и ядром JavaScript, смотрите обзор технологии JavaScript.

Объект window поддерживается всеми браузерами. Он представляет окно браузера.

Все глобальные JavaScript объекты, функции и переменные автоматически становятся членами объекта window.

Глобальные переменные являются свойствами объекта window.

Глобальные функции являются методами объекта window.

Даже объект document (в HTML DOM) является свойством объекта window:

window.document.getElementById("header");

то же самое что:

document.getElementById("header");

Объект windows: назначение, свойства и методы. Определение местоположения пользователя.

Объект Window является центром прикладных интерфейсов клиентского JavaScript. Он представляет окно веб-браузера или фрейм, а сослаться на него можно с помощью идентификатора window. Объект Window определяет свойства, такие как location, которое ссылается на объект Location, определяющий URL текущего окна и т.п.

Кроме того, объект Window определяет методы, такие как alert(), который отображает диалог с сообщением, и setTimeout(), который регистрирует функцию для вызова через указанный промежуток времени.

Объект Window в клиентском JavaScript является глобальным объектом. Это означает, что объект Window находится на вершине цепочки областей видимости и что его свойства и методы фактически являются глобальными переменными и функциями. Объект Window имеет свойство window, которое всегда ссылается на сам объект. Это свойство можно использовать для ссылки на сам объект, но обычно в этом нет необходимости, если требуется просто сослаться на свойство глобального объекта окна.

Объект Window определяет также множество других важных свойств, методов и конструкторов, которое мы и рассмотрим ниже.

Таймеры

Функции setTimeout() и setInterval() позволяют зарегистрировать функцию, которая будет вызываться один или более раз через определенные интервалы времени. Это очень важные функции для клиентского JavaScript, и поэтому они были определены как методы объекта Window, несмотря на то что являются универсальными функциями, не выполняющими никаких действий с окном.

Метод setTimeout() объекта Window планирует запуск функции через определенное число миллисекунд. Метод setTimeout() возвращает значение, которое может быть передано методу *clearTimeout()*, чтобы отменить запланированный ранее запуск функции.

Метод setInterval() похож на setTimeout(), за исключением того, что он автоматически заново планирует повторное выполнение через указанное количество миллисекунд:

setInterval(updateClock, 60000); // Вызывать updateClock() через каждые 60 сек

Подобно setTimeout(), метод setInterval() возвращает значение, которое может быть передано методу *clearInterval()*, чтобы отменить запланированный запуск функции.

В следующем примере определяется вспомогательная функция, которая ожидает указанный интервал времени, многократно вызывает указанную функцию и затем отменяет запланированные вызовы по истечении другого заданного интервала времени.

По исторически сложившимся причинам в первом аргументе методам setTimeout() и setInterval() допускается передавать строку. В этом случае строка будет интерпретироваться (как с применением функции eval()) через указанный интервал времени. Спецификация HTML5 (и все браузеры, кроме IE) допускает передавать методам setTimeout() и setInterval() дополнительные аргументы после первых двух. Все эти дополнительные аргументы будут передаваться функции, вызов которой планируется этими методами. Однако если требуется сохранить совместимость с IE, эту возможность использовать не следует.

Если методу setTimeout() указать величину интервала 0 миллисекунд, указанная функция будет вызвана не сразу, а «как только такая возможность появится», т.е. как только завершат работу все обработчики событий.

Адрес документа и навигация по нему

Свойство location объекта Window ссылается на объект Location, представляющий текущий URL-адрес документа, отображаемого в окне и определяющий методы, инициирующие загрузку нового документа в окно.

Свойство location объекта Document также ссылается на объект Location:

window.location === document.location; // всегда верно

Кроме того, объект Document имеет свойство URL, хранящее статическую строку с адресом URL документа. При перемещении по документу с использованием идентификаторов фрагментов (таких как «#table-of-contents») внутри документа объект Location будет обновляться, отражая факт перемещения, но свойство document.URL останется неизменным.

Анализ URL

Свойство location окна является ссылкой на объект Location и представляет URL-адрес документа, отображаемого в данный момент в текущем окне. Свойство href объекта Location - это строка, содержащая полный текст URL-адреса. Метод *toString()* объекта Location возвращает значение свойства href, поэтому в контекстах, где неявно подразумевается вызов метода toString(), вместо конструкции location.href можно писать просто location.

Другие свойства этого объекта, такие как *protocol*, *host*, *hostname*, *port*, *pathname*, *search* и *hash*, определяют отдельные части URL-адреса. Они известны как свойства «декомпозиции URL» и также поддерживаются объектами Link (которые создаются элементами <a> и <area> в HTML-документах).

Свойства hash и search объекта Location представляют особый интерес. Свойство hash возвращает «идентификатор фрагмента» из адреса URL, если он имеется: символ решетки (#) со следующим за ним идентификатором. Свойство search содержит часть URL-адреса, следующую за вопросительным знаком, если таковая имеется, включая сам знак вопроса. Обычно эта часть URL-адреса является строкой запроса. В целом эта часть URL-адреса используется для передачи параметров и является средством встраивания аргументов в URL-адрес.

Хотя эти аргументы обычно предназначены для сценариев, выполняющихся на сервере, нет никаких причин, по которым они не могли бы также использоваться в страницах, содержащих JavaScript-код.

Загрузка нового документа

Метод assign() объекта Location заставляет окно загрузить и отобразить документ по указанному URL-адресу. Метод replace() выполняет похожую операцию, но перед открытием нового документа он удаляет текущий документ из списка посещавшихся страниц.

Когда сценарию просто требуется загрузить новый документ, часто предпочтительнее использовать метод replace(), а не assign(). В противном случае кнопка Back (Назад) браузера вернет оригинальный документ и тот же самый сценарий снова загрузит новый документ. Метод location.replace() можно было бы использовать для загрузки версии веб-страницы со статической разметкой HTML, если сценарий обнаружит, что браузер пользователя не обладает функциональными возможностями, необходимыми для отображения полноценной версии

Примечательно, что строка URL-адреса в этом примере, переданная методу replace(), представляет относительный адрес. Относительные URL-адреса интерпретируются относительно страницы, в которой они появляются, точно так же, как если бы они использовались в гиперссылке.

Кроме методов assign() и replace() объект Location определяет также метод reload(), который заставляет браузер перезагрузить документ. Однако более традиционный способ заставить браузер перейти к новой странице заключается в том, чтобы просто присвоить новый URL-адрес свойству

История посещений

Свойство history объекта Window ссылается на объект History данного окна. Объект History хранит историю просмотра страниц в окне в виде списка документов и сведений о них. Свойство length объекта History позволяет узнать количество элементов в списке, но по причинам, связанным с безопасностью, сценарии не имеют возможности получить хранящиеся в нем URL-адреса. (Иначе любой сценарий смог бы исследовать историю посещения веб-сайтов.)

Методы *back()* и *forward()* действуют подобно кнопкам Back (Назад) и Forward (Вперед) браузера: они заставляют браузер перемещаться на один шаг назад и вперед по истории просмотра данного окна. Третий метод, *go()*, принимает целочисленный аргумент и пропускает заданное число страниц, двигаясь вперед (если аргумент положительный) или назад (если аргумент отрицательный) в списке истории:

Если окно содержит дочерние окна (такие как элементы <iframe>), истории посещений в дочерних окнах хронологически чередуются с историей посещений в главном окне. То есть вызов history.back() (например) в главном окне может вызвать переход назад, к ранее отображавшемуся документу, в одном из дочерних окон, оставив главное окно в текущем состоянии.

Современные веб-приложения способны динамически изменять содержимое страницы без загрузки нового документа. Приложениям, действующим подобным образом, может потребоваться предоставить пользователям возможность использовать кнопки Back и Forward для перехода между этими динамически созданными состояниями приложения.

Управление историей посещений до появления стандарта HTML5 представляло собой довольно сложную задачу. Приложение, управляющее собственной историей, должно было создавать новые записи в списке истории окна, связывать эти записи с информацией о состоянии, определять момент щелчка на кнопке Back, чтобы перейти к другой записи в списке, получать информацию, связанную с этой записью и воссоздавать предыдущее состояние приложения в соответствии с этой информацией.

Один из приемов реализации такого поведения опирается на использование скрытого элемента <iframe>, в котором сохраняется информация о состоянии и создаются записи в списке истории просмотра. Чтобы создать новую запись, в этот скрытый фрейм динамически записывается новый документ, с помощью методов open() и write() объекта Document. Документ должен включать всю информацию, необходимую для воссоздания соответствующего состояния приложения. Когда пользователь щелкнет на кнопке Back, содержимое скрытого фрейма изменится. До появления стандарта HTML5 не предусматривалось никаких событий, которые извещали бы об этом изменении, поэтому, чтобы определить момент щелчка на кнопке Back, приходилось использовать функцию setInterval() и с ее помощью 2-3 раза в секунду проверять наличие изменений в скрытом фрейме.

Однако на практике разработчики, когда требуется реализовать подобное управление историей просмотра, предпочитают использовать готовые решения. Многие фреймворки JavaScript включают такие решения. Например, для библиотеки jQuery существует расширение history. Существуют также автономные библиотеки управления историей. Например, одной из наиболее популярных является библиотека RSH (Really Simple History - действительно простое управление историей).

Информация о браузере и об экране

Иногда сценариям бывает необходимо получить информацию о веб-браузере, в котором они выполняются, или об экране, на котором отображается браузер. В этом разделе описываются свойства navigator и screen объекта Window. Эти свойства ссылаются, соответственно, на объекты Navigator и Screen, содержащие информацию, которая дает возможность подстроить поведение сценария под существующее окружение.

Объект Navigator

Свойство navigator объекта Window ссылается на объект Navigator, содержащий общую информацию о номере версии и о производителе браузера. Объект Navigator назван «в честь» браузера Netscape Navigator, но он также поддерживается во всех других браузерах. (Кроме того, IE поддерживает свойство clientInformation как нейтральный синоним для navigator. К сожалению, другие браузеры свойство с таким именем не поддерживают.)

В прошлом объект Navigator обычно использовался сценариями для определения типа браузера - Internet Explorer или Netscape. Однако такой подход к определению типа браузера сопряжен с определенными проблемами, т.к. требует постоянного обновления с появлением новых браузеров или новых версий существующих браузеров. Ныне более предпочтительным считается метод на основе проверки функциональных возможностей. Вместо того чтобы делать какие-либо предположения о браузерах и их возможностях, гораздо проще прямо проверить наличие требуемой функциональной возможности (например, метода или свойства).

Однако иногда определение типа браузера может представлять определенную ценность. Один из таких случаев - возможность обойти ошибку, свойственную определенному типу браузера определенной версии. Объект Navigator имеет четыре свойства, предоставляющих информацию о версии работающего браузера, и вы можете использовать их для определения типа браузера:

navigator.appName

Название веб-браузера. В IE это строка «Microsoft Internet Explorer». В Firefox значением этого свойства является строка «Netscape». Для совместимости с существующими реализациями определения типа браузера значением этого свойства в других браузерах часто является строка «Netscape».

navigator.appVersion

Обычно значение этого свойства начинается с номера версии, за которым следует другая информация о версии браузера и его производителе. Обычно в начале строки указывается номер 4.0 или 5.0, свидетельствующий о совместимости с четвертым или пятым поколением браузеров. Формат строки в свойстве appVersion не определяется стандартом, поэтому невозможно организовать разбор этой строки способом, не зависящим от типа браузера.

navigator.userAgent

Строка, которую браузер посылает в http-заголовке USER-AGENT. Это свойство обычно содержит ту же информацию, что содержится в свойстве appVersion, а также может включать дополнительные сведения. Как и в случае со свойством appVersion, формат представления этой информации не стандартизован. Поскольку это свойство содержит больше информации, именно оно обычно используется для определения типа браузера.

navigator.platform

Строка, идентифицирующая операционную систему (и, возможно, аппаратную платформу), в которой работает браузер.

Сложность свойств объекта Navigator делает невозможной универсальную реализацию определения типа браузера. На раннем этапе развития Всемирной паутины было написано немало программного кода, зависящего от типа браузера, проверяющего свойства, такие как navigator.appName. Создавая новые браузеры, производители обнаружили, что для корректного отображения содержимого существующих веб-сайтов они должны устанавливать значение «Netscape» в свойстве appName. По тем же причинам потерял свою значимость номер в начале значения свойства appVersion, и в настоящее время реализация определения типа браузера должна опираться на строку в свойстве navigator.userAgent, имеющую более сложный формат, чем ранее.

В дополнение к свойствам с информацией о версии и производителе браузера, объект Navigator имеет еще несколько свойств и методов. В число стандартных и часто реализуемых нестандартных свойств входят:

navigator.onLine

Свойство navigator.onLine (если существует) определяет, подключен ли браузер к сети. Приложениям может потребоваться сохранять информацию о состоянии локально, если браузер не подключен к сети.

navigator.geolocation

Объект Geolocation, определяющий API для выяснения географического положения пользователя.

navigator.javaEnabled()

Нестандартный метод, который должен возвращать true, если браузер способен выполнять Java-апплеты.

navigator.cookiesEnabled()

Нестандартный метод, который должен возвращать true, если браузер способен сохранять cookies. Если браузер настроен на сохранение cookies только для определенных сайтов, этот метод может возвращать некорректное значение.

Диалоговые окна

Объект Window обладает тремя методами для отображения простейших диалоговых окон. Метод alert() выводит сообщение и ожидает, пока пользователь закроет диалоговое окно. Метод confirm() предлагает пользователю щелкнуть на кнопке OK или Cancel (Отмена) и возвращает логическое значение. Метод prompt() выводит сообщение, ждет ввода строки пользователем и возвращает эту строку.

Методы alert(), confirm() и prompt() чрезвычайно просты в использовании, но правила хорошего дизайна требуют, чтобы они применялись как можно реже. Диалоги, подобные этим, нечасто используются в Веб, и большинство пользователей сочтет диалоговые окна, выводимые этими методами, выпадающими из обычной практики. Единственный вариант, когда имеет смысл обращаться к этим методам - это отладка. JavaScript-программисты часто вставляют вызов метода alert() в программный код, пытаясь диагностировать возникшие проблемы.

Обратите внимание, что текст, отображаемый методами alert(), confirm() и prompt() в диалогах - это обычный неформатированный текст. Его можно форматировать только пробелами, переводами строк и различными знаками пунктуации.

Методы confirm() и prompt() являются блокирующими, т.е. они не возвращают управление, пока пользователь не закроет отображаемые ими диалоговые окна. Это значит, что, когда выводится одно из этих окон, программный код прекращает выполнение, и текущий загружаемый документ, если таковой существует, прекращает загружаться до тех пор, пока пользователь не отреагирует на запрос. В большинстве браузеров метод alert() также является блокирующим и ожидает от пользователя закрытия диалогового окна, но это не является обязательным требованием.

В дополнение к методам alert(), confirm() и prompt() в объекте Window имеется более сложный метод, showModalDialog(), отображающий модальный диалог, содержащий разметку HTML, и позволяющий передавать аргументы и получать возвращаемое значение.

Метод showModalDialog() выводит модальный диалог в отдельном окне браузера. Первым аргументом методу передается URL, определяющий HTML-содержимое диалога. Во втором аргументе может передаваться произвольное значение (допускается передавать массивы и объекты), которое будет доступно сценарию в диалоге, как значение свойства window.dialogArguments. Третий аргумент - нестандартный список пар имя/значение, разделенных точками с запятой, который, если поддерживается, может использоваться для настройки размеров и других атрибутов диалогового окна. Для определения размеров окна диалога можно использовать параметры «dialogwidth» и «dialogheight», а чтобы позволить пользователю изменять размеры окна, можно определить параметр «resizable=yes».

Окно, отображаемое эти методом, является модальным, и метод showModalDialog() не возвращает управление, пока окно не будет закрыто. После закрытия окна значение свойства window.returnValue становится возвращаемым значением метода. Обычно разметка HTML диалога должна включать кнопку OK, которая записывает желаемое значение в свойство returnValue и вызывает window.close().

В примере ниже приводится разметка HTML для использования с методом showModalDialog(). Комментарий в начале примера включает пример вызова showModalDialog(), а на рисунке показано диалоговое окно, созданное вызовом из примера. Обратите внимание, что большая часть текста, отображаемого в окне, передается методу showModalDialog() во втором аргументе, а не является жестко определенной частью разметки HTML:

Обработка ошибок

Свойство onerror объекта Window - это обработчик событий, который вызывается во всех случаях, когда необработанное исключение достигло вершины стека вызовов и когда браузер готов отобразить сообщение об ошибке в консоли JavaScript. Если присвоить этому свойству функцию, функция будет вызываться всякий раз, когда в окне будет возникать ошибка выполнения программного кода JavaScript: присваиваемая функция станет обработчиком ошибок для окна.

Исторически сложилось так, что обработчику события onerror объекта Window передается три строковых аргумента, а не единственный объект события, как в других обработчиках. (Другие объекты в клиентском JavaScript также имеют обработчики onerror, обрабатывающие различные ошибочные ситуации, но все они являются обычными обработчиками событий, которым передается единственный объект события.) Первый аргумент обработчика window.onerror - это сообщение, описывающее произошедшую ошибку. Второй аргумент - это строка, содержащая URL-адрес документа с JavaScript-кодом, приведшим к ошибке. Третий аргумент - это номер строки в документе, где произошла ошибка.

Помимо этих трех аргументов важную роль играет значение, возвращаемое обработчиком onerror. Если обработчик onerror возвращает true, это говорит браузеру о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, браузер не должен выводить собственное сообщение об ошибке. К сожалению, по историческим причинам в Firefox обработчик ошибок должен возвращать true, чтобы сообщить о том, что ошибка обработана.

Обработчик onerror является пережитком первых лет развития JavaScript, когда в базовом языке отсутствовала инструкция try/catch обработки исключений. В современном программном коде этот обработчик используется редко.

Элементы документа как свойства окна

Если для именования элемента в HTML-документе используется атрибут id и если объект Window еще не имеет свойства, имя которого совпадает со значением этого атрибута, объект Window получает неперечислимое свойство с именем, соответствующим значению атрибута id, значением которого становится объект HTMLElement, представляющий этот элемент документа.

Как вы уже знаете, объект Window играет роль глобального объекта, находящегося на вершине цепочки областей видимости в клиентском JavaScript. Таким образом, вышесказанное означает, что атрибуты id в HTML-документах становятся глобальными переменными, доступными сценариям. Если, например, документ включает элемент <button id="ok"/>, на него можно сослаться с помощью глобальной переменной ok.

Если переменная объявляется в сценарии, который в документе находится выше именованного элемента, наличие переменной будет препятствовать появлению нового свойства окна. А если переменная объявляется в сценарии, который находится ниже именованного элемента, первая же инструкция присваивания значения этой переменной затрет значение неявно созданного свойства.

Неявное использование идентификаторов элементов в качестве глобальных переменных - это пережиток истории развития веб-браузеров. Эта особенность необходима для сохранения обратной совместимости с существующими веб-страницами, но использовать ее сейчас не рекомендуется - в любой момент производители браузеров могут определить новое свойство в объекте Window, что нарушит работу любого программного кода, использующего неявно определяемое свойство с этим именем. Поиск элементов лучше выполнять явно, с помощью метода document.getElementById().

Открытие и закрытие окон

Открыть новое окно веб-браузера (или вкладку, что обычно зависит от настроек браузера) можно с помощью метода open() объекта Window. Метод Window.open() загружает документ по указанному URL-адресу в новое или в существующее окно и возвращает объект Window, представляющий это окно. Он принимает четыре необязательных аргумента:

Первый аргумент open() - это URL-адрес документа, отображаемого в новом окне. Если этот аргумент отсутствует (либо является пустой строкой), будет открыт специальный URL пустой страницы about:blank.

Второй аргумент open() - это строка с именем окна. Если окно с указанным именем уже существует (и сценарию разрешено просматривать содержимое этого окна), используется это существующее окно. Иначе создается новое окно и ему присваивается указанное имя. Если этот аргумент опущен, будет использовано специальное имя «\_blank», т.е. будет открыто новое неименованное окно.

Обратите внимание, что сценарии не могут просто так указывать имена окон и не могут получать контроль над окнами, используемыми другими веб-приложениями: они могут указывать имена только тех существующих окон, которыми им «разрешено управлять» (термин взят из спецификации HTML5). Проще говоря, сценарий может указать имя существующего окна, только если это окно содержит документ, происходящий из того же источника, или если это окно было открыто самим сценарием (или рекурсивно открытым окном, которое открыло это окно).

Кроме того, если одно окно является фреймом, вложенным в другое окно, сценарии в любом из них получают возможность управлять другим окном. В этом случае можно использовать зарезервированные имена «\_top» (для обозначения вмещающего окна верхнего уровня) и «\_parent» (для обозначения ближайшего вмещающего окна).

Третий необязательный аргумент open() - это список параметров, определяющих размер и видимые элементы графического пользовательского интерфейса нового окна. Если опустить этот аргумент, окно получает размер по умолчанию и полный набор графических элементов: строку меню, строку состояния, панель инструментов и т.д. В браузерах, поддерживающих вкладки, это обычно приводит к созданию новой вкладки. Этот третий аргумент является нестандартным, и спецификация HTML5 требует, чтобы браузеры игнорировали его.

Указывать четвертый аргумент open() имеет смысл, только если второй аргумент определяет имя существующего окна. Этот аргумент - логическое значение, определяющее, должен ли URL-адрес, указанный в первом аргументе, заменить текущую запись в истории просмотра окна (true) или требуется создать новую запись (false). Если этот аргумент опущен, используется значение по умолчанию false.

Значение, возвращаемое методом open(), является объектом Window, представляющим вновь созданное окно. Этот объект позволяет сослаться в JavaScript-коде на новое окно так же, как исходный объект Window ссылается на окно, в котором выполняется сценарий:

В окнах, созданных методом window.open(), свойство opener ссылается на объект Window сценария, открывшего его

Метод Window.open() часто используется рекламодателями для создания «всплывающих окон» с рекламой, когда пользователь путешествует по Всемирной паутине. Такие всплывающие окна могут раздражать пользователя, поэтому большинство веб-браузеров реализуют механизм блокирования всплывающих окон. Обычно вызов метода open() преуспевает, только если он производится в ответ на действия пользователя, такие как щелчок мышью на кнопке или на ссылке. Попытка открыть всплывающее окно, которая производится, когда браузер просто загружает (или выгружает) страницу, как в приведенном примере, обычно оканчивается неудачей.

Новое окно открывается при помощи метода open() и закрывается при помощи метода close(). Если объект Window был создан сценарием, то этот же сценарий сможет закрыть его следующей инструкцией:

w.close();

Отношения между фреймами

Мы уже видели, что метод open() объекта Window возвращает новый объект Window, свойство opener которого ссылается на первоначальное окно. Таким образом, два окна могут ссылаться друг на друга, и каждое из них может читать свойства и вызывать методы другого. То же самое возможно для фреймов. Сценарий, выполняющийся в окне или фрейме, может ссылаться на объемлющее или вложенное окно или фрейм при помощи свойств, описываемых ниже.

Как вы уже знаете, сценарий в любом окне или фрейме может сослаться на собственное окно или фрейм с помощью свойства window или self. Фрейм может сослаться на объект Window вмещающего окна или фрейма с помощью свойства parent.

Если фрейм находится внутри другого фрейма, содержащегося в окне верхнего уровня, то он может сослаться на окно верхнего уровня так: parent.parent. Однако в качестве универсального сокращения имеется свойство top: независимо от глубины вложенности фрейма его свойство top ссылается на содержащее его окно самого верхнего уровня.

Объектная модель документа (DOM). Объект document: назначение, свойства и методы. Иерархическая структура DOM: назначение, свойства и методы.

контексту, History, в конце концов, Selection в документе.

Основные объекты браузера

Перед тем как перейти к изучению объекта document, давайте вспомним, какие нам в JavaScript доступны объекты и за что они отвечают.

При открытии документа браузер автоматически создаёт набор объектов для JavaScript, с помощью которых Вы можете не только работать с этим документом (объект document), но и управлять самим браузером (объекты window, location, navigator, screen, history). Все эти объекты образуют объектную модель браузера (BOM - Browser Object Model).

Главным объектом в этой модели является объект window. Все остальные объекты доступны как свойства объекта window (window.document, window.location и т.д.). Если мы работаем с текущим окном, то "window." можно опускать, т.е. document, location и т.д. Объект location - отвечает за адресную строку, объект history - за кнопки вперёд и назад, объект, объект screen - за экран пользователя, объект window - отвечает за само окно, а также позволяет изменять его размеры, перемещать его и т.д., navigator - позволяет получить информацию о браузере.

Наибольший интерес среди всех этих объектов для нас предоставляет именно объект document, т.к. он отвечает за документ, загруженный в окно или вкладку браузера. Он даёт начало объектной модели документа (DOM - Document Object Model), которая стандартизована в спецификации и поддерживается всеми браузерами.

К рассмотрению этой модели мы приступим на следующих уроках. На этом уроке мы рассмотрим некоторые свойства и методы объекта document, т.е. такие которые особого отношения к объектной модели документа не имеют.

Свойства и методы объекта document

Объект document содержит следующие "общие" свойства и методы:

Свойство document.implementation — возвращает объект DOMImplementation, ассоциированный с текущим документом.

Свойство document.characterSet - возвращает кодировку, которая используется для рендеринга текущего документа. Данное значение может отличаться от кодировки указанной в HTML странице, т.к. пользователь может её переопределить, т.е. выбрать в соответствующем меню браузера другую кодировку, которая будет использоваться для отображения текущего документа.

Свойство document.inputEncoding - возвращает кодировку, которая использовалась во время синтаксического разбора (парсинга) документа. Если документ создан в памяти, то данное свойство возвращает значение null. Данное свойство доступно только для чтения.

Свойство document.lastModified - возвращает строку, содержащую дату и время последнего изменения текущего документа. Gecko и Internet Explorer возвращают время в часовом поясе локального компьютера, WebKit - в UTC. alert(document.lastModified);

Свойство document.readyState - возвращает строку, содержащую статус текущего документа. Данное свойство доступно только для чтения.

Свойство document.referrer - возвращает строку, содержащую адрес (URL) страницы, с которой пользователь пришёл на эту страницу. Если текущий документ не был открыт через ссылку (например, с помощью закладки или прямого ввода адреса в адресную строку), то данное свойство вернёт пустую строку.

Свойство document.cookie - позволяет получить или установить cookie, которые будут связаны с текущим документом.

Например, откроем новую вкладку в браузере и введём в адресную строку http://ya.ru/ и нажмём на клавишу Enter. После этого перейдём в консоль и поработаем со свойством document.cookie:

Выведем все cookie связанные с текущим документом, т.е. пары ключ=значение: document.cookie;

Запишем новый cookie: document.cookie = "test1 = Test1";

Запишем ещё один новый cookie: document.cookie = "test2 = Test2";

Выведем все cookie связанные с текущим документом: document.cookie.

Свойство document.URL - возвращает строку, содержащую полный URL адрес текущего HTML документа.

Свойство document.URI - позволяет получить или задать расположение (URI) документа. Если документ был создан с помощью объекта DocumentImplementation или если он был не определён, то данное свойство возвращает значение null. В отличие от свойства document.URL, свойство document.URI может быть использовано для любых типов документов, в то время как URL можно использовать только для HTML документов.

Свойство document.domain - возвращает строку, содержащую доменное имя сервера, с которого загружен текущий документ. Если домен текущего документа не может быть определён, то данное свойство вернёт значение null.

Метод document.write() - предназначен для вывода в документ строки, указанной в качестве параметра данного метода. Если данный метод вызывается в процессе загрузки документа, то он выводит строку в текущем месте. В том случае, если данный метод вызывается после загрузки документа, то он приводит к полной очистке этого документа и вывода строчки. Это происходит, потому что после загрузки документа браузер уже полностью построил DOM и в документ уже нельзя внести изменения таким способом, а только с помощью добавления узлов в объектную модель документа.

Метод document.writeln() - полностью аналогичен методу document.write() за тем исключением, что он добавляет символ перевода строки (\n) в конец строки, указанной в качестве параметра.

Метод document.open() — открывает поток для вывода в текущий документ строк с помощью методов document.write() и document.writeln(). Данный метод (document.open()) обычно применяется для формирования нового документа. Вызов данного метода в текущем документе приводит к его полной очистке. После того как документ открыт методом document.open() и сформирован с помощью методов document.write() и document.writeln(), его необходимо закрыть методом document.close().

Метод document.close() - предназначен для закрытия потока, который Вы открыли с помощью метода document.open().

Использование объектов HTML и объектной модели документа (DOM).

Что такое Объектная Модель Документа (DOM)?

Объектная Модель Документа (DOM) – это программный интерфейс (API) для HTML и XML документов. DOM предоставляет структурированное представление документа и определяет то, как эта структура может быть доступна из программ, которые могут изменять содержимое, стиль и структуру документа. Представление DOM состоит из структурированной группы узлов и объектов, которые имеют свойства и методы. По существу, DOM соединяет веб-страницу с языками описания сценариев либо языками программирования.

Веб-страница – это документ. Документ может быть представлен как в окне браузера, так и в самом HTML-коде. В любом случае, это один и тот же документ. DOM предоставляет другой способ представления, хранения и управления этого документа. DOM полностью поддерживает объектно-ориентированное представление веб-страницы, делая возможным её изменение при помощи языка описания сценариев наподобие JavaScript.

Например: стандарт DOM описывает, что метод getElementsByTagName в коде, указанном ниже, должен возвращать список всех элементов <p> в документе.

Все свойства, методы и события, доступные для управления и создания новых страниц, организованы в виде объектов. Например, объект document, который представляет сам документ, объект table, который реализует специальный интерфейс DOM HTMLTableElement, необходимый для доступа к HTML-таблицам, и так далее. Данная документация даёт справку об объектах DOM, реализованных Gecko-подобных браузерах.

DOM и JavaScript

Небольшой пример выше, как почти все примеры в этой справке – это JavaScript. То есть пример *написан* на JavaScript, но при этом *используется* DOM для доступа к документу и его элементам. DOM не является языком программирования, но без него JavaScript не имел бы никакой модели или представления о веб-странице, HTML-документе, XML-документе и их элементах. Каждый элемент в документе - весь документ в целом, заголовок, таблицы внутри документа, заголовки таблицы, текст внутри ячеек таблицы - это части объектной документной модели для этого документа, поэтому все они могут быть доступны и могут изменяться с помощью DOM и скриптового языка наподобие JavaScript.

Вначале JavaScript и DOM были тесно связаны, но впоследствии они развились в различные сущности. Содержимое страницы хранится в DOM и может быть доступно и изменяться с использованием JavaScript, поэтому мы можем записать это в виде приблизительного равенства:

API (веб либо XML страница) = DOM + JS (язык описания скриптов)

DOM спроектирован таким образом, чтобы быть независимым от любого конкретного языка программирования, обеспечивая структурное представление документа согласно единому и последовательному API. Хотя мы всецело сфокусированы на JavaScript в этой справочной документации, реализация DOM может быть построена для любого языка, как в следующем примере на Python:

Каким образом доступен DOM?

Вы не должны делать ничего особенного для работы с DOM. Различные браузеры имеют различную реализацию DOM, эти реализации показывают различную степень соответствия с действительным стандартом DOM (это тема, которую мы пытались не затрагивать в данной документации), но каждый браузер использует свой DOM, чтобы сделать веб страницы доступными для взаимодействия с языками сценариев.

В следующем примере внутри элемента <script> определён код JavaScript, данный код устанавливает функцию при загрузке документа (когда весь DOM доступен для использования). Эта функция создаёт новый элемент H1, добавляет текст в данный элемент, а затем добавляет H1 в дерево документа:

Важные типы данных

Данный раздел предназначен для краткого описания различных типов и объектов в простой и доступной манере. Существует некоторое количество различных типов данных, которые используются в API, на которые вы должны обратить внимание. Для простоты, синтаксис примеров в данном разделе обычно ссылается на узлы как на elements, на массивы узлов как на nodeLists ( либо просто elements ) и на атрибуты узла, просто как на attributes.

DOM-интерфейсы (DOM interfaces)

Это руководство об объектах и реальных вещах, которые вы можете использовать для управления DOM-иерархией. Есть много моментов, где понимание того, как это работает, может удивлять. Например, объект, представляющий HTML form элемент, берёт своё свойство name из интерфейса HTMLFormElement, а свойство className - из интерфейса HTMLElement. В обоих случаях свойство, которое вы хотите, находится в этом объекте формы.

Кроме того, отношение между объектами и интерфейсами, которые они реализуют в DOM может быть удивительным и этот раздел пытается рассказать немного о существующих интерфейсах в DOM и о том, как они могут быть доступны.

Интерфейсы и объекты (Interfaces and objects)

Многие объекты реализуют действия из нескольких интерфейсов. Объект таблицы, например, реализует специальный HTML Table Element Interface, который включает такие методы как createCaption и insertRow. Но так как это таблица - это ещё и HTML-элемент, table реализует интерфейс Element, описанный в разделе DOM element Reference. Наконец, так как HTML-элемент (в смысле DOM) - это узел (node) в дереве, которое составляет объектную модель для HTML- или XML-страницы, табличный элемент также реализует более общий интерфейс Node, из которого происходит Element.

Когда вы получаете ссылку на объект table, как в следующем примере, вы обычно используете все три интерфейса этого объекта, вероятно, даже не зная этого.

Основные интерфейсы в DOM (Core interfaces in the DOM)

Этот раздел перечисляет несколько самых распространённых интерфейсов в DOM. Идея не в том чтобы описать, что делают эти методы API, но в том чтобы дать вам несколько мыслей насчёт видов методов и свойств, которые вы будете часто видеть, используя DOM. Эти распространённые части API использованы в большинстве примеров раздела DOM Examples в конце этой справки.

Этот документ содержит примеры для каждого интерфейса, который вы можете использовать в своей разработке. В некоторых случаях примеры - полноценные веб-страницы с доступом к DOM в элементе <script>, также перечислены элементы, необходимые чтобы запустить скрипт в форме, и HTML-элементы, над которыми будут производиться операции DOM. Когда встречается такой случай, можно просто копировать и вставить пример в новый HTML-документ, сохранить и запустить его в браузере.

Есть случаи, однако, где примеры более лаконичные. Чтобы запустить примеры, которые лишь демонстрируют основы взаимодействия интерфейсов с HTML-элементами, вы можете подготовить тестовую страницу, в которую будете помещать функции внутрь скриптов. Следующая очень простая веб-страница содержит элемент <script> в заголовке, в который вы можете поместить функции, чтобы протестировать интерфейс. Страница содержит несколько элементов с атрибутами, которые можно возвращать, устанавливать или, другими словами, манипулировать и содержит пользовательский интерфейс, необходимый, чтобы вызывать нужные функции из браузера.

Вы можете использовать эту тестовую страницу или похожую для проверки интерфейсов DOM, которые вас интересуют и просмотра того, как они работают в браузерах. Вы можете обновить содержимое функции test() при необходимости, создать больше кнопок или добавить элементы при необходимости.

При возникновении события система генерирует сигнал, а также предоставляет механизм, с помощью которого можно автоматически предпринимать какие-либо действия (например, выполнить определённый код), когда происходит событие. Например, в аэропорту, когда взлётно-посадочная полоса свободна для взлёта самолёта, сигнал передаётся пилоту, и в результате они приступают к взлёту.

В Web события запускаются внутри окна браузера и, как правило, прикрепляются к конкретному элементу, который в нем находится. Это может быть один элемент, набор элементов, документ HTML, загруженный на текущей вкладке, или все окно браузера.

Подробнее о событиях можно посмотреть в Справочнике по событиям.

Каждое доступное событие имеет обработчик событий — блок кода (обычно это функция JavaScript, вводимая вами в качестве разработчика), который будет запускаться при срабатывании события. Когда такой блок кода определён на запуск в ответ на возникновение события, мы говорим, что мы регистрируем обработчик событий. Обратите внимание, что обработчики событий иногда называют \*\*слушателями событий (от англ. event listeners). Они в значительной степени взаимозаменяемы для наших целей, хотя, строго говоря, они работают вместе. Слушатель отслеживает событие, а обработчик — это код, который запускается в ответ на событие.

События, как понятие, относятся не только к JavaScript — большинство языков программирования имеют модель событий, способ работы которой часто отличается от модели в JavaScript. Фактически, даже модель событий в JavaScript для веб-страниц отличается от модели событий для просто JavaScript, поскольку используются они в разных средах.

На данном этапе обучения вам не нужно особо разбираться в различных средах программирования, однако важно понимать, что принцип работы *событий* в них отличается.

Способы использования веб-событий

Существует множество различных способов добавления кода обработчика событий на веб-страницы так, чтобы он срабатывал при возникновении соответствующего события. В этом разделе мы рассмотрим различные механизмы и обсудим, какие из них следует использовать.

Свойства обработчика событий

В этом курсе вы уже сталкивались со свойствами, связываемыми с алгоритмом работы обработчика событий. Вернёмся к приведённому выше

Создайте локальную копию файла random-color-eventhandlerproperty.html и откройте её в своём браузере. Это всего лишь копия простого примера про случайные цвета, который мы уже разобрали в этой статье. Теперь попробуйте изменить btn.onclick на следующие значения и понаблюдайте за результатами:

btn.onfocus и btn.onblur — Цвет изменится, когда кнопка будет сфокусирована или не сфокусирована (попробуйте нажать Tab, чтобы выбрать кнопку или убрать выбор). Эти свойства часто применяются для отображения информации о том, как заполнить поля формы, когда они сфокусированы, или отобразить сообщение об ошибке, если поле формы было заполнено с неправильным значением.

btn.ondblclick — Цвет будет изменяться только при двойном щелчке.

window.onkeypress, window.onkeydown, window.onkeyup — Цвет будет меняться при нажатии клавиши на клавиатуре, причём keypress ссылается на обычное нажатие (нажатие кнопки и последующее её отпускание *как одно целое*), в то время как keydown и keyup*разделяют* действия на нажатие клавиши и отпускание, и ссылаются на них соответственно. Обратите внимание, что это не работает, если вы попытаетесь зарегистрировать этот обработчик событий на самой кнопке - его нужно зарегистрировать на объекте window, который представляет все окно браузера.

btn.onmouseover и btn.onmouseout — Цвет будет меняться при наведении указателя мыши на кнопку или когда указатель будет отводиться от кнопки соответственно.

Некоторые события очень общие и доступны практически в любом месте (например, обработчик onclick может быть зарегистрирован практически для любого элемента), тогда как некоторые из них более конкретны и полезны только в определённых ситуациях (например, имеет смысл использовать onplay (en-US) только для определённых элементов, таких как <video>).

Встроенные обработчики событий - не используйте их

Самый ранний из введённых в сеть Web методов регистрации *обработчиков событий* базируется на HTML атрибутах (встроенные обработчики событий):

Значение атрибута — это буквально код JavaScript, который вы хотите запустить при возникновении события. В приведённом выше примере вызывается функция, определённая внутри элемента <script> на той же странице, но вы также можете вставить JavaScript непосредственно внутри атрибута

Более того, не рекомендуется смешивать HTML и JavaScript файлы, так как в дальнейшем такой код становится сложнее с точки зрения обработки (парсинга). Лучше держать весь JavaScript в одном месте. Также, если он находится в отдельном файле, вы можете применить его к нескольким документам HTML.

Даже при работе только в одном файле использование встроенных обработчиков не является хорошей идеей. Ладно, если у вас одна кнопка, но что, если у вас их 100? Вам нужно добавить в файл 100 атрибутов; обслуживание такого кода очень быстро превратится в кошмар. С помощью JavaScript вы можете легко добавить функцию обработчика событий ко всем кнопкам на странице независимо от того, сколько их было.

Примечание: Разделение логики вашей программы и вашего контента также делает ваш сайт более дружественным к поисковым системам.

Функции addEventListener() и removeEventListener()

Это не важно для простых небольших программ, но для более крупных и более сложных программ он может повысить эффективность очистки старых неиспользуемых обработчиков событий. Кроме того, это позволяет вам иметь одну и ту же кнопку, выполняющую различные действия в разных обстоятельствах — все, что вам нужно сделать, это добавить/удалить обработчики событий, если это необходимо.

Также вы можете зарегистрировать несколько обработчиков для одного и того же события на элементе.

Поскольку вторая строка будет перезаписывать значение onclick, установленное первой.

Обе функции будут выполняться при щелчке элемента.

Рассмотрим некоторые современные концепции, имеющие отношение к событиям. На данный момент не обязательно понимать их полностью, но представление о них поможет лучше понять некоторые модели кода, с которыми вы, вероятно, столкнётесь.

Объекты событий

Иногда внутри функции обработчика событий вы можете увидеть параметр, заданный с таким именем, как event, evt или просто e. Называется он объектом события и он автоматически передаётся обработчикам событий для предоставления дополнительных функций и информации.

Итак в коде выше мы включаем объект события e в функцию, а в функции — настройку стиля фона для e.target, который является кнопкой. Свойство объекта события target всегда является ссылкой на элемент, с которым только что произошло событие. Поэтому в этом примере мы устанавливаем случайный цвет фона на кнопке, а не на странице.

e.target применяют, когда нужно установить один и тот же обработчик событий на несколько элементов и, когда на них происходит событие, применить определённое действие к ним ко всем. Например, у вас может быть набор из 16 плиток, которые исчезают при нажатии. Полезно всегда иметь возможность просто указать, чтобы объект исчез, как e.target, вместо того, чтобы выбирать его более сложным способом.

Результат выглядит следующим образом (попробуйте щёлкнуть по нему):

Большинство обработчиков событий, с которыми вы столкнулись, имеют только стандартный набор свойств и функций (методов), доступных для объекта события (см. Event для ссылки на полный список). Однако некоторые более продвинутые обработчики добавляют специальные свойства, содержащие дополнительные данные, которые им необходимо выполнять. Например, Media Recorder API (en-US) имеет событие, доступное для данных, которое срабатывает, когда записано какое-либо аудио или видео и доступно для выполнения чего-либо (например, для сохранения или воспроизведения). Соответствующий объект события ondataavailable handler имеет свойство данных, содержащее записанные аудио- или видеоданные, чтобы вы могли получить к нему доступ и что-то сделать с ним.

Предотвращение поведения по умолчанию

Но если пользователь отправил данные не правильно, как разработчик, вы хотите остановить отправку на сервер и выдать сообщение об ошибке с информацией о том, что не так и что нужно сделать. Некоторые браузеры поддерживают функции автоматической проверки данных формы, но, поскольку многие этого не делают, вам не следует полагаться на них и выполнять свои собственные проверки валидации. Давайте посмотрим на простой пример.

В JavaScript мы реализуем очень простую проверку внутри обработчика события onsubmit (событие "отправить" запускается в форме, когда оно отправлено), который проверяет, пусты ли текстовые поля.

Очевидно, что это довольно слабая проверка формы - это не помешает пользователю отправить форму с пробелами или цифрами, введёнными в поля, но для примера подходит.

Всплытие и перехват событий

Последним предметом для рассмотрения в этой теме является то, с чем вы не часто будете сталкиваться, но это может стать настоящей головной болью, если вы не поймёте, как работает следующий механизм. *Всплытие* и *перехват событий* — два механизма, описывающих, что происходит, когда два обработчика одного и того же события активируются на одном элементе.

Но есть проблема: когда вы нажимаете на видео, оно начинает воспроизводиться, но одновременно вызывает скрытие <div>. Это связано с тем, что видео находится внутри <div>, это часть его, поэтому нажатие на видео фактически запускает оба вышеуказанных обработчика событий.

Всплытие и перехват событий — концепция выполнения

Когда событие инициируется элементом, который имеет родительские элементы (например, <video> в нашем случае), современные браузеры выполняют две разные фазы — фазу захвата и фазу всплытия.

На стадии всплытия происходит полная противоположность:

Браузер проверяет, имеет ли элемент, который был фактически нажат, обработчик события onclick, зарегистрированный на нем в фазе всплытия, и запускает его, если это так.

Затем он переходит к следующему непосредственному родительскому элементу и выполняет то же самое, затем следующее и так далее, пока не достигнет элемента <html>.

Всплытие также позволяет нам использовать делегирование событий. Если у какого-либо родительского элемента есть множество дочерних элементов, и вы хотите, чтобы определённый код выполнялся при щелчке (событии) на каждом из дочерних элементов, можно установить обработчик событий на родительском элементе и события, происходящие на дочерних элементах будут всплывать до их родителя. При этом не нужно устанавливать обработчик событий на каждом дочернем элементе.

Это все, что вам нужно знать о веб-событиях на этом этапе. Как уже упоминалось, события не являются частью основного JavaScript — они определены в веб-интерфейсах браузера (Web API).

Event()

Создаёт объект Event и возвращает его вызывающему.

Свойства

Event.bubblesТолько для чтения

Логическое значение, указывающее, всплыло ли событие вверх по DOM или нет.

Event.cancelBubble(en-US)

Event.cancelableТолько для чтения

Логическое значение, показывающее возможность отмены события.

Event.composed(en-US)Только для чтения

Логическое значение, показывающее может или нет событие всплывать через границы между shadow DOM (внутренний DOM конкретного элемента) и обычного DOM документа.

Event.currentTargetТолько для чтения

Ссылка на текущий зарегистрированный объект, на котором обрабатывается событие. Это объект, которому планируется отправка события; поведение можно изменить с использованием перенаправления (*retargeting*).

Event.deepPath(en-US)Non-standard

Массив DOM-узлов, через которые всплывало событие.

Event.defaultPreventedТолько для чтения

Показывает, была ли для события вызвана функция event.preventDefault().

Event.eventPhaseТолько для чтения

Указывает фазу процесса обработки события.

Event.explicitOriginalTarget(en-US)Non-standardТолько для чтения

Явный первоначальный целевой объект события (Mozilla-специфичный). Не может содержать анонимного контента.

Event.originalTarget(en-US)Non-standardТолько для чтения

Первоначальный целевой объект события до перенаправлений (Mozilla-специфичный). Может быть из анонимного контента.

Event.returnValue(en-US)Non-standard

Нестандартная альтернатива (оставшаяся от старых версий Microsoft Internet Explorer) для Event.preventDefault() и Event.defaultPrevented.

Event.scoped(en-US)Только для чтения

Логическое значение, показывающее всплывает ли данное событие через shadow root (внутренний DOM-элемента). Это свойство было переименовано в composed(en-US).

Event.srcElementNon-standard

Нестандартный синоним (остался от старых версий Microsoft Internet Explorer) для Event.target.

Event.targetТолько для чтения

Ссылка на целевой объект, на котором произошло событие.

Event.timeStampТолько для чтения

Время, когда событие было создано (в миллисекундах). По спецификации это время от начала Эпохи (Unix Epoch), но в действительности в разных браузерах определяется по-разному; кроме того, ведётся работа по изменению его на DOMHighResTimeStamp тип.

Event.typeТолько для чтения

Название события (без учёта регистра символов).

Event.isTrustedТолько для чтения

Показывает было или нет событие инициировано браузером (например, по клику мышью) или из скрипта (например, через функцию создания события, такую как event.initEvent (en-US))

Методы

Event.createEvent()(en-US)

Создаёт новое событие, которое затем должно быть проинициализировано вызовом его метода initEvent().

Event.initEvent()Deprecated

Инициализация значений созданного с помощью Document.createEvent() (en-US) события. Если событие уже отправлено, то эта функция ничего не делает.

Event.preventBubble()Non-standardВышла из употребления с версии Gecko 24

Предотвращает всплытие события. Устаревшая, используйте вместо неё event.stopPropagation.

Event.preventCapture()Non-standardВышла из употребления с версии Gecko 24

Устаревшая, используйте вместо неё event.stopPropagation.

Event.preventDefault()

Отменяет событие (если его возможно отменить).

Event.stopImmediatePropagation()

Для конкретного события не будет больше вызвано обработчиков. Ни тех, которые привязаны к этому же элементу (на котором работает обработчик, который вызывает этот Event.stopImmediatePropagation()), ни других, которые могли бы вызваться при распространении события позже (например, в фазе перехвата - capture).

Event.stopPropagation()

Остановка распространения события далее по DOM.

Event.getPreventDefault()Non-standard

Нестандартная. Возвращает значение Event.defaultPrevented. Используйте вместо неё Event.defaultPrevented.

Обработчик событий: понятие и способы задания.

События — это действия или случаи, возникающие в программируемой вами системе, о которых система сообщает вам для того, чтобы вы могли с ними взаимодействовать. Например, если пользователь нажимает кнопку на веб-странице, вы можете ответить на это действие, отобразив информационное окно. В этой статье мы обсудим некоторые важные концепции, связанные с событиями, и посмотрим, как они работают в браузерах.

При возникновении события система генерирует сигнал, а также предоставляет механизм, с помощью которого можно автоматически предпринимать какие-либо действия (например, выполнить определённый код), когда происходит событие. Например, в аэропорту, когда взлётно-посадочная полоса свободна для взлёта самолёта, сигнал передаётся пилоту, и в результате они приступают к взлёту.

События не только для веб-страниц

События, как понятие, относятся не только к JavaScript — большинство языков программирования имеют модель событий, способ работы которой часто отличается от модели в JavaScript. Фактически, даже модель событий в JavaScript для веб-страниц отличается от модели событий для просто JavaScript, поскольку используются они в разных средах.

Вы также можете использовать JavaScript для создания кросс-браузерных расширений — улучшения функциональности браузера с помощью технологии WebExtensions. В отличии от модели веб-событий здесь свойства обработчиков событий пишутся в так называемом регистре CamelCase (например, onMessage, а не onmessage) и должны сочетаться с функцией addListener. См. runtime.onMessage page для примера.

На данном этапе обучения вам не нужно особо разбираться в различных средах программирования, однако важно понимать, что принцип работы *событий* в них отличается.

Способы использования веб-событий

Существует множество различных способов добавления кода обработчика событий на веб-страницы так, чтобы он срабатывал при возникновении соответствующего события. В этом разделе мы рассмотрим различные механизмы и обсудим, какие из них следует использовать.

Свойства обработчика событий

В этом курсе вы уже сталкивались со свойствами, связываемыми с алгоритмом работы обработчика событий.

Некоторые события очень общие и доступны практически в любом месте (например, обработчик onclick может быть зарегистрирован практически для любого элемента), тогда как некоторые из них более конкретны и полезны только в определённых ситуациях (например, имеет смысл использовать onplay (en-US) только для определённых элементов, таких как <video>).

Встроенные обработчики событий - не используйте их

Значение атрибута — это буквально код JavaScript, который вы хотите запустить при возникновении события. В приведённом выше примере вызывается функция, определённая внутри элемента <script> на той же странице, но вы также можете вставить JavaScript непосредственно внутри атрибута

Для многих свойств обработчика событий существуют эквиваленты в виде атрибутов HTML. Однако, не рекомендуется их использовать — это считается плохой практикой. Использование атрибутов для регистрации обработчика событий кажется простым и быстрым методом, но такое описание обработчиков также скоро становится неудобным и неэффективным.

Более того, не рекомендуется смешивать HTML и JavaScript файлы, так как в дальнейшем такой код становится сложнее с точки зрения обработки (парсинга). Лучше держать весь JavaScript в одном месте. Также, если он находится в отдельном файле, вы можете применить его к нескольким документам HTML.

Функции addEventListener() и removeEventListener()

Внутри функции addEventListener() мы указываем два параметра — имя события, для которого мы хотим зарегистрировать этот обработчик, и код, содержащий функцию обработчика, которую мы хотим запустить в ответ. Обратите внимание, что будет целесообразно поместить весь код внутри функции addEventListener() в анонимную функцию, например:

Этот механизм имеет некоторые преимущества по сравнению с более старыми механизмами, рассмотренными ранее. Например, существует аналогичная функция removeEventListener(), которая удаляет ранее добавленный обработчик. Это приведёт к удалению набора обработчиков в первом блоке кода в этом разделе:

btn.removeEventListener('click', bgChange);

myElement.onclick = functionA; myElement.onclick = functionB;

Поскольку вторая строка будет перезаписывать значение onclick, установленное первой. Однако, если:

myElement.addEventListener('click', functionA); myElement.addEventListener('click', functionB);

Обе функции будут выполняться при щелчке элемента.

Кроме того, в этом механизме событий имеется ряд других мощных функций и опций. Эта тема выходит за рамки данной статьи, но если вы хотите изучить подробнее, переходите по ссылкам: Метод Event​Target​.add​Event​Listener() и Метод Event​Target​.remove​Event​Listener().

Свойства обработчика событий имеют меньшую мощность и параметры, но лучше совместимость между браузерами (поддерживается ещё в Internet Explorer 8).

Основные преимущества третьего механизма заключаются в том, что при необходимости можно удалить код обработчика событий, используя removeEventListener(), и так же можно добавить несколько элементов-обработчиков того же типа к элементам. Например, вы можете вызвать addEventListener('click', function() {...}) для элемента несколько раз, с разными функциями, указанными во втором аргументе. Это невозможно при использовании свойств обработчика событий, поскольку любые последующие попытки установить свойство будут перезаписывать более ранние, например:

element.onclick = function1; element.onclick = function2; etc.

Объекты событий

Иногда внутри функции обработчика событий вы можете увидеть параметр, заданный с таким именем, как event, evt или просто e. Называется он объектом события и он автоматически передаётся обработчикам событий для предоставления дополнительных функций и информации.

Примечание: вы можете найти исходник кода для этого примера на GitHub (также взгляните на его выполнение).

Итак в коде выше мы включаем объект события e в функцию, а в функции — настройку стиля фона для e.target, который является кнопкой. Свойство объекта события target всегда является ссылкой на элемент, с которым только что произошло событие. Поэтому в этом примере мы устанавливаем случайный цвет фона на кнопке, а не на странице.

Результат выглядит следующим образом (попробуйте щёлкнуть по нему):

Большинство обработчиков событий, с которыми вы столкнулись, имеют только стандартный набор свойств и функций (методов), доступных для объекта события (см. Event для ссылки на полный список). Однако некоторые более продвинутые обработчики добавляют специальные свойства, содержащие дополнительные данные, которые им необходимо выполнять.

Иногда бывают ситуации, когда нужно остановить событие, выполняющее то, что оно делает по умолчаниюКогда вы вводите данные и нажимаете кнопку отправки, естественное поведение заключается в том, что данные должны быть отправлены на указанную страницу на сервере для обработки, а браузер перенаправляется на страницу с сообщением об успехе (или остаться на той же странице, если другое не указано).

Но если пользователь отправил данные не правильно, как разработчик, вы хотите остановить отправку на сервер и выдать сообщение об ошибке с информацией о том, что не так и что нужно сделать. Некоторые браузеры поддерживают функции автоматической проверки данных формы, но, поскольку многие этого не делают, вам не следует полагаться на них и выполнять свои собственные проверки валидации. Давайте посмотрим на простой пример.

Очевидно, что это довольно слабая проверка формы - это не помешает пользователю отправить форму с пробелами или цифрами, введёнными в поля, но для примера подходит.

Всплытие и перехват событий — концепция выполнения

Когда событие инициируется элементом, который имеет родительские элементы (например, <video> в нашем случае), современные браузеры выполняют две разные фазы — фазу захвата и фазу всплытия.

На стадии захвата происходит следующее:

Браузер проверяет, имеет ли самый внешний элемент (<html>) обработчик события onclick, зарегистрированный на нем на этапе захвата и запускает его, если это так.

Затем он переходит к следующему элементу внутри <html> и выполняет то же самое, затем следующее и так далее, пока не достигнет элемента, на который на самом деле нажали.

На стадии всплытия происходит полная противоположность:

Браузер проверяет, имеет ли элемент, который был фактически нажат, обработчик события onclick, зарегистрированный на нем в фазе всплытия, и запускает его, если это так.

Примечание: Как упоминалось выше, по умолчанию все обработчики событий регистрируются в фазе всплытия и это имеет смысл в большинстве случаев. Если вы действительно хотите зарегистрировать событие в фазе захвата, вы можете сделать это, зарегистрировав обработчик с помощью addEventListener() и установив для третьего дополнительного свойства значение true.

Делегирование события

Всплытие также позволяет нам использовать делегирование событий. Если у какого-либо родительского элемента есть множество дочерних элементов, и вы хотите, чтобы определённый код выполнялся при щелчке (событии) на каждом из дочерних элементов, можно установить обработчик событий на родительском элементе и события, происходящие на дочерних элементах будут всплывать до их родителя. При этом не нужно устанавливать обработчик событий на каждом дочернем элементе.

Хорошим примером является серия элементов списка. Если вы хотите, чтобы каждый из них, например, отображал сообщение при нажатии, вы можете установить обработчик событий click для родительского элемента <ul> и он будет всплывать в элементах списка.

Стадии обработки событий (механизм всплытия и перехвата). Отмена

Всплытие

Принцип всплытия очень простой.

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

И так далее вверх по цепочке до самого document.

Поэтому если кликнуть на <p>, то мы увидим три оповещения: p → div → form.

Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде.

*Почти* все события всплывают.

Ключевое слово в этой фразе – «почти».

Например, событие focus не всплывает. В дальнейшем мы увидим и другие примеры. Однако, стоит понимать, что это скорее исключение, чем правило, всё-таки большинство событий всплывают.

event.target

Всегда можно узнать, на каком конкретно элементе произошло событие.

Самый глубокий элемент, который вызывает событие, называется *целевым* элементом, и он доступен через event.target.

Отличия от this (=event.currentTarget):

event.target – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.

this – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывёт до элемента <form>, на котором сработает обработчик.

При этом внутри обработчика form.onclick:

this (=event.currentTarget) всегда будет элемент <form>, так как обработчик сработал на ней.

event.target будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

Возможна и ситуация, когда event.target и this – один и тот же элемент, например, если клик был непосредственно на самом элементе <form>, а не на его подэлементе.

Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод event.stopPropagation().

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены.

То есть, event.stopPropagation() препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует метод event.stopImmediatePropagation(). Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости!

Всплытие – это удобно. Не прекращайте его без явной нужды, очевидной и архитектурно прозрачной.

Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами.

решили отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще у нас кликают люди. Некоторые системы аналитики так делают. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.

Существуют два варианта значений опции capture:

Если аргумент false (по умолчанию), то событие будет поймано при всплытии.

Если аргумент true, то событие будет перехвачено при погружении.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение.

Здесь обработчики навешиваются на *каждый* элемент в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события.

Существует свойство event.eventPhase, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, мы обычно и так знаем об этом в обработчике.

Чтобы убрать обработчик removeEventListener, нужна та же фаза

Если мы добавили обработчик вот так addEventListener(..., true), то мы должны передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

На каждой фазе разные обработчики на одном элементе срабатывают в порядке назначения

Формы: доступ к элементам и проверка данных, вводимых пользователем.

Перед отправкой данных на сервер важно убедиться, что все обязательные поля формы заполнены данными в корректном формате. Это называется валидацией на стороне клиента и помогает убедиться, что данные, введённые в каждый элемент формы, соответствуют требованиям.

Валидация на стороне клиента — это первичная проверка введённых данных, которая существенно улучшает удобство взаимодействия с интерфейсом; обнаружение некорректных данных на стороне клиента позволяет пользователю немедленно их исправить. Если же проверка происходит только на сервере, процесс заполнения может быть более трудоёмким, так как требует повторения одних и тех же действий отправки данных на сервер для получения обратного ответа с сообщением о том, что нужно исправить.

Однако, *не следует рассматривать* валидацию на стороне клиента как достаточную меру безопасности! Любые данные, отправляемые через форму, необходимо *дополнительно* проверять на безопасность и *на стороне сервера*, поскольку валидацию на стороне клиента достаточно просто обойти и она может не остановить злоумышленников. Чтобы лучше понимать потенциальные угрозы, рекомендуем ознакомиться с разделом Безопасность вебсайтов; валидация на стороне сервера выходит за рамки этого модуля, но о ней следует помнить.

Что такое валидация формы?

Зайдите на любой популярный сайт, имеющий форму регистрации. Вы заметите, что при вводе данных в неправильном формате, пользователя сразу уведомляют о наличии проблемы.

Это называется валидацией формы. По мере ввода, браузер и/или сервер проверяют данные, чтобы определить, соответствуют ли они требуемому формату. Валидация, выполняемая в браузере, называется валидацией на стороне клиента, а выполняемая на сервере — валидацией на стороне сервера. В этом разделе мы сосредоточимся на валидации, выполняемой на стороне клиента.

Если формат корректен, приложение позволяет отправить данные на сервер и (обычно) сохранить в базу данных; в противном случае выводится сообщение с описанием того, что нужно исправить, позволяя ввести данные снова.

Предупреждение: Никогда не доверяйте данным, передаваемым на сервер клиентской программой. Даже если ваша форма правильно валидируется и не допустит введение потенциально вредоносных данных на стороне клиента, злоумышленники по-прежнему могут изменить сетевой запрос.

Использование встроенной валидации форм

Одной из самых важных функций элементов форм HTML5 (en-US) является способность валидировать бóльшую часть пользовательских данных без использования JavaScript. Это выполняется с помощью атрибутов валидации у элементов формы.

Если данные, введённые в поле формы, соответствуют правилам перечисленных выше атрибутов, они считаются валидными, если нет — не валидными

Когда элемент валиден, справедливы следующие утверждения:

Элемент соответствует CSS-псевдоклассу :valid, позволяющему стилизовать только валидные элементы.

Если пользователь пытается отправить данные, браузер отправит форму при условии, что ничто другое (например, JavaScript) не помешает ему это сделать

Когда элемент не валиден, справедливы следующие утверждения:

Элемент соответствует CSS-псевдоклассу :invalid или, в зависимости от ошибки, другим псевдоклассам (например, :out-of-range), которые позволяют применять определённые стили к элементам, не являющимся валидными.

Если пользователь пытается отправить данные, браузер заблокирует форму и выведет сообщение об ошибке.

Самым простым в HTML5-валидации является атрибут required. Добавьте его к элементу, чтобы сделать заполнение обязательным. Элемент с данным атрибутом соответствует CSS-псевдоклассу :required, а если поле ввода пустое, вместо отправки формы отобразится сообщение об ошибке.

Попробуйте отправить форму без введения значения. Обратите внимание, что не валидное поле получает фокус, появляется сообщение об ошибке ("Заполните это поле") и блокируется отправка формы.

Ограничение допустимых значений

Валидация форм с помощью JavaScript

Если нужно управлять внешним видом встроенных сообщений об ошибке или работать с устаревшими браузерами, которые не поддерживают встроенную валидацию форм HTML, вам следует использовать JavaScript. В данном разделе мы рассмотрим различные способы делать это.

Работа с формами

Если в HTML-документе определена форма, то она доступна сценарию JavaScript как объект, входящий в объект document с именем, заданным атрибутом NAME тега FORM.

Свойства форм

Форма имеет два набора свойств, состав одного из которых фиксированный, а состав другого зависит от того, какие элементы определены в форме.

Большинство свойств первого набора просто отражает значение соответствующих атрибутов тега FORM. Что же касается массива elements, то в нем находятся объекты, соответствующие элементам, определенным в форме.

При использовании простого JavaScript, вы можете использовать select box для передвижения по сайту.

Когда нажимается любая кнопка, форма инициируется и координаты *x/y* щелчка мыши на кнопке загружаются в объект *запроса*. Например, если пользователь выбрал *Cancel!*, объект запроса будет содержать переменные *cancel.x* и *cancel.y*. Точно так же щелчок на *Continue* привел бы к переменным *continue.x* и *continue.y*.

Здесь важно отметить, что элементы *.x* и *.y* - не переменные объектов *continue* или *cancel*, а фактически часть имени *"continue.x"*. Следовательно, нельзя использовать типичный метод для определения существования одного из дочерних объектов запроса:

Благодаря названным массивам JavaScript, можно проверить какая картинка инициировала форму.

Для некоторых приложений, полезно знать координаты курсора на нажатой картинке. Эта информация, конечно, содержится в парах *image\_element.x* и *image\_element.y*. Следующий серверный JavaScript мог бы использоваться для этого:

Передача данных между формами на различных страницах

В качестве простого примера, предположите, что ваша "домашняя" страница запрашивает имя пользователя, затем использует это имя, чтобы обратиться к пользователю на следующих страницах.

Потому что объект работает не таким образом. Правильный и полный синтаксис для доступа к *VALUE* только что выбранным полем в списке - это:

Document.form.selectObject[document.form.selectObject.selectedIndex].value

Для доступак тексту элемента используйте свойство text: Document.form.selectObject[document.form.selectObject.selectedIndex].text

Форма отправляется, если нажата клавиша *enter*, в то время как единственный входной текстовый элемент формы имеет фокус. Вы можете вызывать подобное поведение в форме, имеющей более одного элемента, разбивая форму на ряд отдельных форм, так, чтобы каждая форма имела только один текстовый элемент.

Как сделать загрузку страницы при выборе флажка?

Используйте обработчик *OnChange* для вызова функции *submit()*:

**Понятие регулярного выражения. Поиск текста с помощью регулярных выражений и использование регулярных выражений.**

Регуля́рные выраже́ния (англ. regular expressions) — формальный язык, используемый в компьютерных программах, работающих с текстом, для поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. wildcard characters).

Регулярные выражения - это шаблоны, используемые для сопоставления последовательностей символов в строках. В JavaScript регулярные выражения также являются объектами.

Создание регулярного выражения

Регулярное выражение можно создать двумя способами:

Используя литерал регулярного выражения, например:

var re = /ab+c/;

Использование конструктора влечёт за собой компиляцию регулярного выражения во время исполнения скрипта. Используйте данный способ, если знаете, что выражение будет изменяться или не знаете шаблон заранее.

Написание шаблона регулярного выражения

Шаблон регулярного выражения состоит из обычных символов, например /abc/, или комбинаций обычных и специальных символов, например /ab\*c/ или /Chapter (\d+)\.\d\*/.

В случае когда поиск соответствия требует чего-то большего, чем прямое сопоставление, например нахождение последовательности символов 'b' или нахождение пробела, шаблон включает в себя специальные символы. Например, шаблон /ab\*c/ соответствует любой комбинации символов, в которой за 'a' следует ноль или более символов 'b' (\* означает ноль или более вхождений предыдущего символа), за которыми сразу же следует символ 'c'. В строке "cbbabbbbcdebc," этому шаблону сопоставляется подстрока 'abbbbc'.

В следующей таблице приводится полный список специальных символов регулярных выражений с их описаниями.

Использование скобок

Скобки вокруг любой части регулярного выражения означают что эта часть совпадаемой подстроки будет запомнена. Раз запомнена, подстрока может выбрана для использования как это описано в Using Parenthesized Substring Matches.

Для того, чтобы сопоставить подстроку без вызова совпавшей части для запоминания, внутри круглых скобок необходимо предварить паттерн сочетанием символов ' ?:'.

Работа с Регулярными Выражениями

Регулярные выражения используются в методах test и exec объекта RegExp и с методами match, replace, search, и split объекта String. Эти методы подробно объясняются в Справочнике JavaScript

Чтобы просто узнать есть ли в строке что либо соответствующее шаблону, воспользуйтесь методами test или search; а чтобы получить больше информации пользуйтесь методами exec или match (хотя эти методы работают медленнее). Если вы пользуетесь exec или match и если совпадения есть, эти методы вернут массив и обновлённые свойства объекта ассоциированного регулярного выражения а также предопределённого объекта RegExp регулярного выражения. Если совпадений нет, метод exec вернёт null (который сконвертируется в false).

Как показано во втором варианте этого примера, вы можете использовать регулярное выражение, созданное при помощи инициализатора объекта, без присваивания его переменной. Таким образом, если вы используете данную форму записи без присваивания переменной, то в процессе дальнейшего использования вы не можете получить доступ к свойствам данного регулярного выражения.

Совпадения /d(b+)d/g в двух случаях являются разными объектами регулярного выражения и, следовательно, имеют различные значения для свойства lastIndex. Если вам необходим доступ к свойствам объекта, созданного при помощи инициализатора, то вы должны сначала присвоить его переменной.

Использование скобочных выражений для нахождения подстрок

Использование скобок в шаблоне регулярного выражения повлечёт "запоминание" совпавшей подстроки.

Число возможных скобочных подстрок неограничено. Возвращаемый массив содержит все полученные совпадения, удовлетворяющие выражению в скобках. Следующий пример показывает как использовать скобочные выражения для нахождения подстрок.

Расширенный поиск с флагами

Регулярные выражения имеют четыре опциональных флага, которые делают возможным глобальный и регистронезависимый поиск. Флаги могут использоваться самостоятельно или вместе в любом порядке, а также могут являться частью регулярного выражения.

Флаг m используется, чтобы входная строка рассматривалась как многострочная. Если флаг m используется, то ^ и $ вызовет совпадение в начале или конце любой строки в строке ввода вместо начала или конца вводимой строки целиком.

Примеры

След. примеры показывают использование регулярных выражений.

Создание и использование регулярных выражений. Использование объекта RegExp.

Регулярные выражения представляют шаблон, который используется для поиска или модификации строки. Для работы с регулярными выражениями в JavaScript определен объект RegExp.

Методы RegExp

Чтобы определить, соответствует ли регулярное выражение строке, в объекте RegExp определен метод test(). Этот метод возвращает true, если строка соответствует регулярному выражению, и false, если не соответствует.

Аналогично работает метод exec - он также проверяет, соответствует ли строка регулярному выражению, только теперь данный метод возвращает ту часть строки, которая соответствует выражению. Если соответствий нет, то возвращается значение null.

Группы символов

Регулярное выражение не обязательно состоит из обычных строк, но также может включать специальные элементы синтаксиса регулярных выражений. Один из таких элементов представляют группы символов, заключенные в квадратные скобки.

Выражение [абв] указывает на то, что строка должна иметь одну из трех букв.

Если нам надо определить наличие в строке буквенных символов из определенного диапазона, то можно сразу задать этот диапазон:

В первом случае строка не должна иметь только символы из диапазона а-я, но поскольку строка "обороноспособность" состоит только из символов из этого диапазона, то метод test() возвращает false, то есть регулярное выражение не соответствует стоке.

Флаги выражений

Флаги позволяют настроить поведение регулярных выражений. Каждый флаг представляет отдельный символ, который ставится в конце регулярного выражения. В JavaScript применяются следующие флаги:

Флаг global позволяет найти все подстроки, которые соответствуют регулярному выражению. По умолчанию при поиске подстрок регулярное выражение выбирает первую попавшуюся подстроку из строки, которая соответствует выражению. Хотя в строке может быть множество подстрок, которые также соответствуют выражению. Для этого применяется данный флаг в виде символа g в выражениях

Флаг ignoreCase позволяет найти подстоки, которые соответствуют регулярному выражению, вне зависимости от регистра символов в строке. Для этого в регулярных выражениях применяется символ i

Флаг multiline позволяет найти подстроки, которые соответствуют регулярному выражению, в многострочном тексте. Для этого в регулярных выражениях применяется символ m

Флаг dotAll позволяет сопоставить точку в регулярном выражении с любым символом текста, в том числе с разделителем строки. Для этого в регулярных выражениях применяется символ s

Использование модификаторов.

В БЭМ-методологии JavaScript — это одна из технологий реализации блока. Поэтому к JavaScript наравне с CSS применяются основные принципы методологии БЭМ:

Единая предметная область

Разделение кода на части

Работа с уровнями переопределения и использование сборки

Для работы с JavaScript наравне с другими технологиями используются термины блоков, элементов и модификаторов.

Таким образом в БЭМ используется абстракция над DOM-деревом — БЭМ-дерево.

Привязка JavaScript-блоков к HTML

Первичным «каркасом» страницы является HTML-дерево документа. Блокам с JavaScript-реализацией могут соответствовать узлы в HTML с именем блока в атрибуте class. В этом случае говорится о том, что блоки имеют DOM-представление.

Такой способ привязки JavaScript-компонентов к DOM-дереву имеет следующие преимущества:

естественная деградация интерфейса на клиентах с отключенным JavaScript;

прогрессивный рендеринг — возможность начинать отрисовку элементов интерфейса до окончания загрузки всех данных страницы.

В БЭМ-методологии поведение каждого блока описывается независимо. Независимость блоков в JavaScript позволяет повторно использовать блоки и реализуется за счет использования принципов:

инкапсуляции

декларативности

наследования

Логика работы каждого блока, его опциональных элементов и модификаторов описывается в отдельных файлах. JavaScript-файлы хранятся в соответствии с правилами организации файловой структуры БЭМ-проекта.

Реализация поведения блока, как и его внешнего вида, независима и отделена от реализации других блоков не только логически, но и физически — вынесена в отдельный файл.

Принцип инкапсуляции

В БЭМ внутренняя реализация блока скрыта. Блок предоставляет API для взаимодействия с другими блоками. Таким образом достигается независимость блока, возможность его повторного использования.

Элементы всегда являются внутренней реализацией блока, поэтому обращение к ним возможно только через API самого блока.

Принцип декларативности

Логика работы блока описывается декларативно: как набор действий и условий, при которых эти действия необходимо выполнять. Это позволяет разделять функциональность блока на отдельные части и использовать уровни переопределения.

Принцип наследования

Декларативное описание поведения блоков позволяет использовать методы базового блока внутри производного, наследовать их. Новый блок может получать все свойства и методы базового.

Также можно создавать цепочки наследования

Декларативное описание работы блока обеспечивает возможность использовать уровни переопределения для JavaScript наравне с CSS:

реализовывать новое поведение блока на другом уровне переопределения, сохраняя предыдущее, наследовать и дополнять его (делать super call);

полностью перекрывать поведение блока (переопределять);

добавлять блоки с новым поведением в проект.

С помощью уровней переопределения можно создать универсальную JavaScript-библиотеку блоков, изменять ее на проектном уровне и включать в проект только необходимое поведение блоков.

Подробнее о подключении библиотеки в проект.

Запись в БЭМ-терминах позволяет:

Полностью перекрывать поведение блока на другом уровне переопределения.

Для работы с JavaScript в БЭМ-терминах и использования уровней переопределения в БЭМ создан специализированный фреймворк i-bem.js.

Работа с блоками

Взаимодействие блоков

БЭМ-методология предполагает работу с независимыми блоками. Однако на практике полная независимость блоков недостижима.

Блоки могут взаимодействовать друг с другом с помощью:

Подписки на события других экземпляров блоков.

Подписки на изменения модификаторов.

Непосредственного вызова методов других экземпляров блоков или статических методов класса другого блока.

Любых паттернов взаимодействия. Например, канала событий: все коммуникации происходят благодаря сообщениям, которые компоненты публикуют и слушают с помощью посредника.

Примеры реализации доступны в документации к i-bem.js.

БЭМ-методология рекомендует выстраивать взаимодействие между блоками в иерархическом порядке в соответствии с их расположением в DOM-дереве. Вложенный блок не должен ничего знать о родительском блоке, так как это нарушает принцип независимости компонентов.

Взаимодействие блока с элементами

Элемент — это внутренняя реализация блока. Для работы блока с его элементами принято реализовывать дополнительные хелперы блока. Обращение напрямую к элементу другого блока невозможно. Взаимодействие с элементом происходит только через API блока, которому принадлежит данный элемент.

Работа с модификаторами

Поведение блока описывается с помощью состояний. Модификаторы содержат информацию о состояниях блока. Перевод блока в другое состояние производится при помощи установки/снятия модификатора. Изменение модификатора создает событие, которое можно использовать для работы с блоком.

Например, чтобы отметить чекбокс, блоку checkbox нужно установить модификатор checked в значение true.

Для корректной работы JavaScript в БЭМ-проекте все манипуляции с модификаторами должны производиться при помощи методов-хелперов. Изменять значение модификаторов следует с помощью спец методов, а не менять напрямую CSS-класс на соответствующем DOM-узле.

Примеры реализации доступны в документации к i-bem.js.

Реакция на изменение модификаторов

В БЭМ реакция на установку/снятие модификатора описывается декларативно: изменение состояния автоматически вызывает код, который задекларирован для этого состояния. Если появляется модификатор (добавляется новый класс к DOM-узлу), то вся функциональность, свойственная этому модификатору, также применяется. Если модификатор исчезает, функциональность отключается.

Переход блока из одного состояния в другое часто сопровождается изменениями его внешнего вида. Так как добавление модификатора блоку изменяет его класс на DOM-узле, а стили написаны опираясь только на классы, изменение класса автоматически приводит к изменению внешнего вида блока.

Чтобы динамически изменять состояния блоков и элементов, используются специальные методы для установки и снятия модификаторов.

Чтобы реализовать принципы БЭМ в проекте, необходимо:

работать в единых терминах блоков, элементов и модификаторов во всех технологиях;

создавать независимые компоненты (блоки) на уровне JavaScript;

описывать поведение блока как набор действий и условий их выполнения;

обращаться к элементам блока только через API самого блока и не нарушать принцип инкапсуляции;

изменять поведение блоков, элементов и модификаторов с помощью уровней переопределения по аналогии с CSS;

разделять код на мелкие независимые части для удобства работы с отдельными блоками;

повторно использовать блоки.

Чтобы начать использовать JavaScript по БЭМ в рабочем проекте, можно начать применять принципы БЭМ-методологии без использования специализированного фреймворка. Пример проекта с JavaScript по БЭМ на jQuery описан в статье БЭМ — это не только про CSS.

Веб-сервер: задачи, принцип работы.

Понятие «веб-сервер» может относиться как к аппаратной начинке, так и к программному обеспечению. Или даже к обеим частям, работающим совместно.

С точки зрения ПО, веб-сервер включает в себя несколько компонентов, которые контролируют доступ веб-пользователей к размещённым на сервере файлам, как минимум — это *HTTP-сервер*. HTTP-сервер — это часть ПО, которая понимает URL-адреса (веб-адреса) и HTTP (протокол, который ваш браузер использует для просмотра веб-страниц).

На самом базовом уровне, когда браузеру нужен файл, размещённый на веб-сервере, браузер запрашивает его через HTTP-протокол. Когда запрос достигает нужного веб-сервера ("железо"), сервер HTTP (ПО) принимает запрос, находит запрашиваемый документ (если нет, то сообщает об ошибке 404) и отправляет обратно, также через HTTP.

Чтобы опубликовать веб-сайт, необходим либо статический, либо динамический веб-сервер.

Статический веб-сервер, или стек, состоит из компьютера ("железо") с сервером HTTP (ПО). Мы называем это «статикой», потому что сервер посылает размещённые файлы в браузер «как есть».

Динамический веб-сервер состоит из статического веб-сервера и дополнительного программного обеспечения, чаще всего *сервера приложения* и *базы данных*. Мы называем его «динамическим», потому что сервер приложений изменяет исходные файлы перед отправкой в ваш браузер по HTTP.

Например, для получения итоговой страницы, которую вы просматриваете в браузере, сервер приложений может заполнить HTML-шаблон данными из базы данных. Такие сайты, как MDN или Википедия, состоят из тысяч веб-страниц, но они не являются реальными HTML документами — лишь несколько HTML-шаблонов и гигантские базы данных. Эта структура упрощает и ускоряет сопровождение веб-приложений и доставку контента.

Активное изучение

*Активное изучение пока не доступно. Пожалуйста, рассмотрите возможность внести свой вклад.*

Погружаемся глубже

Чтобы загрузить веб-страницу, как мы уже говорили, ваш браузер отправляет запрос к веб-серверу, который приступает к поиску запрашиваемого файла в своём собственном пространстве памяти. Найдя файл, сервер считывает его, обрабатывает как ему это необходимо, и отсылает в браузер. Давайте рассмотрим эти шаги более подробно.

По всем этим причинам поиск хорошего хостинг-провайдера является ключевой частью создания вашего сайта. Рассмотрите многочисленные предложения компаний и выберите то, что соответствует вашим потребностям и бюджету (предложения варьируются от бесплатных до тысяч долларов в месяц). Вы можете найти подробности в этой статье.

Как только вы решили проблему с хостингом, вам понадобится только загрузить свои файлы на ваш веб-сервер.

Связь по HTTP

Во-вторых, веб-сервер обеспечивает поддержку HTTP (англ. *Hypertext Transfer Protocol - гипертекстовый транспортный протокол*). Как следует из названия, HTTP указывает, как передавать гипертекст (т.е. связанные веб-документы) между двумя компьютерами.

Протокол представляет собой набор правил для связи между двумя компьютерами. HTTP является текстовым протоколом без сохранения состояния.

Текстовый

Все команды являются простым человекочитаемым текстом.

HTTP задаёт строгие правила взаимодействия клиента и сервера. Мы рассмотрим сам протокол HTTP в технической статье немного позднее. Пока достаточно знать об этих правилах:

Исключительно *клиенты* могут производить HTTP-запросы, и только на *сервера*. Сервера способны только отвечать на HTTP-*запросы клиента*.

При запросе файла по HTTP, клиент должен сформировать файловый URL.

Веб-сервер *должен ответить* на каждый HTTP-запрос, по крайней мере сообщением об ошибке.

На веб-сервере HTTP-сервер отвечает за обработку входящих запросов и ответ на них.

При получении запроса, HTTP-сервер сначала проверяет, существует ли ресурс по данному URL.

Если это так, веб-сервер отправляет содержимое файла обратно в браузер. Если нет, сервер приложения генерирует необходимый ресурс.

Если ничто из этого не возможно, веб-сервер возвращает сообщение об ошибке в браузер, чаще всего “404 Not Found”.

Статический и Динамический контент

Грубо говоря, сервер может отдавать статическое или динамическое содержимое. «Статическое» означает «отдаётся как есть». Статические веб-сайты делаются проще всего, поэтому мы предлагаем вам сделать свой первый сайт статическим.

«Динамическое» означает, что сервер обрабатывает данные или даже генерирует их на лету из базы данных. Это обеспечивает большую гибкость, но технически сложнее в реализации и обслуживании, из-за чего процесс создания сайта очень сильно усложняется.

Установка и настройка веб-сервера и PHP.

Open Server — это портативный локальный WAMP/WNMP сервер, имеющий многофункциональную управляющую программу и большой выбор подключаемых компонентов. Представленный пакет программ не является очередной любительской сборкой собранной «на коленке», это первый полноценный профессиональный инструмент, созданный специально для веб-разработчиков с учётом их рекомендаций и пожеланий. Данный программный комплекс включает в себя тщательно подобранный набор серверного программного обеспечения, а так же невероятно удобную и продуманную управляющую утилиту, которая обладает мощными возможностями по администрированию и настройке всех доступных компонентов.

OSPanel широко используется с целью разработки, отладки и тестирования веб-проектов, а так же для предоставления веб-сервисов в локальных сетях.

Проект существует уже более 10 лет, он успешно зарекомендовал себя как первоклассный, простой и надёжный инструмент, необходимый каждому веб-мастеру.

Преимущество этого решения перед ближайшими аналогами заключается в грамотной реализации системы управления добавленными доменами, а также в присутствии графического интерфейса. Это позволит пользователю вовсе забыть об использовании «Командной строки» для управления собственным сервером.Многие выбирают OpenServer, поскольку этот софт крайне прост в инсталляции и конфигурировании. Также он имеет графический интерфейс, что немаловажно для огромного пласта юзеров.

Первым шагом будет переход на официальный сайт, где нужно нажать по кнопке «Скачать» (она располагается на верхней панели). Выберите базовую версию в выпадающем меню, как показано на рисунке 7, и начните ее загрузку.

Ожидайте завершения скачивания, а затем запустите полученный EXE-файл.

Задайте путь, куда будет установлен локальный сервер, и нажмите «Извлечь». Обязательно запомните этот путь, ведь по нему в будущем придется постоянно переходить. Дождитесь окончания извлечения.

Перейдите по пути установки программы и запустите находящийся там исполняемый файл.

Вы будете уведомлены, что запуск осуществляется впервые, а это значит, нужно установить патчи для Microsoft Visual C++. Обязательно подтвердите эту операцию.

После инсталляции рекомендуется произвести первый запуск OpenServer, чтобы продолжить дальнейшую настройку, а также просто убедиться в том, что программа функционирует нормально. Для этого потребуется выполнить всего несколько простых действий. После запуска сервера обратите внимание на панель задач.

В появившемся меню нажмите по кнопке «Запустить». После этого наведите курсор на «Мои проекты» и выберите там «localhost». Откроется новая страница в браузере по умолчанию, где вы получите уведомление о нормальной работе локального сервера, как показано на рисунке 10.

Настройка OpenServer – обязательный процесс, который требуется выполнить хотя бы для того, чтобы обеспечить комфорт в работе. Первоначальное конфигурирование подразумевает изменение всего нескольких параметров. Через то же меню программы перейдите в раздел «Настройки». Здесь в первой же вкладке «Основные» можете поменять визуальное оформление и выбрать оптимальный язык интерфейса. Переместитесь на вкладку «Меню», где активируйте параметр «Показывать сайты в главном меню». Это позволит сразу через меню переходить к добавленным доменам, экономя свое личное время. Здесь же можно выбрать браузер, который нужно использовать для открытия веб-страниц, если он будет отличаться от того, что установлен в ОС по умолчанию. Во вкладке "Модули" можно указать нужную версию PHP, Apache и т.д. После нажатия кнопки "Сохранить" программа предложит перезапустить сервер. Выполните это действие для применения новых настроек.

Чтобы начать работать с кодом, нужно:

создать домен в папке «OpenServer/domains», например, test.local;

прямо в корне этой папки создать файл с расширением .php либо файл с расширением .html, в котором будет находиться код PHP;

перезапустить Open Server;

найти и открыть свой test.local в подменю «Мои проекты».

Переменные — это основа любого языка программирования. Во время написания кода мы постоянно будем работать с переменными. Понять концепцию переменных очень просто.

Переменная — это контейнер, в котором содержатся данные, так же как напиток содержится в чашке.

Любую информацию, которую мы будем использовать в коде, сначала необходимо сохранить в переменной.

У переменной обязательно должно быть имя, поэтому переменная всегда состоит из имени и значения. Значение — это любая информация, которая хранится внутри переменной.

Например мы можем попросить посетителя страницы указать свой возраст, а затем использовать это значение для других целей — узнать год рождения или показать возраст самой странице.

Любую переменную следует вначале объявить, то есть дать ей имя и присвоить значение.

В синтаксисе PHP имя переменной записывается латинскими символами, но первым символом всегда должен быть знак доллара $, а затем идёт имя.

Не допускается начинать имя переменной с цифры, а также использовать любые значения, кроме букв алфавита и знака подчеркивания.

Основы синтаксиса PHP: структура PHP-сценария, основные элементы синтаксиса, способы внедрения PHP-сценария в веб-документ.

Разберёмся, из чего состоит любой язык программирования. Как и языки, на которых мы говорим, язык программирования также строится из кирпичиков, которые и составляют совокупность знаний, называемую человеческим языком.

У каждого языка есть свои правила и конструкции, следуя которым мы выражаем свои мысли и делаем их понятными для другого человека. В программировании всё точно также. Но вместо человеческого языка мы используем язык программирования PHP, а в роли нашего собеседника выступает PHP-интерпретатор. Поэтому, чтобы выразить свою мысль, мы должны сделать её понятной для интерпретатора.

Переменные

Переменные — это основа любого языка программирования. Во время написания кода мы постоянно будем работать с переменными. Понять концепцию переменных очень просто. Переменная — это контейнер, в котором содержатся данные, так же как напиток содержится в чашке. Любую информацию, которую мы будем использовать в коде, сначала необходимо сохранить в переменной.

У переменной обязательно должно быть имя, поэтому переменная всегда состоит из имени и значения. Значение — это любая информация, которая хранится внутри переменной.

Пустая переменная не будет слишком полезна. Поэтому давайте сразу что-нибудь туда поместим. Такое действие называется операцией присваивания.

Конкатенация

Этим страшным словом называется операция склеивания нескольких строк в одну.

Чтобы получить такую строку, необходимо использовать оператор конкатенации — . (точка). В качестве операндов он берёт две строки или переменные, содержащие строку, а затем возвращает новую строку.

В этом задании вы познакомитесь с конкатенацией на практике.

Типы данных

Выше уже упоминалось, что данные, помещённые в переменную, могут быть строкой текста или числом. В действительности таких типов данных гораздо больше, но первое время мы будем работать только с несколькими. Однако нам всё равно важно уже сейчас понимать, что такое «типы данных». Тип данных в языке программирования определяет допустимые значения, а также допустимые операции с данными этого типа. Например, с данными типа «целое число» допускается выполнять арифметические операции, а с данными типа «строка» — нет. Поделить строку на строку невозможно.

Оператором в программировании называется нечто, принимающее одно или более значений. Например, в арифметике есть такие операторы: +, -, /, \*. Что означает сложение, вычитание, деление и умножение. Оператор всегда работает только в паре с двумя значениями, например складывает два числа (операнда) или умножает их друг на друга.

Условные конструкции

Иногда, в зависимости от условия, нужно выполнять различные действия. Для этого используется оператор if. Например, мы спросили пол посетителя нашей страницы, сохранили его в переменную, а теперь хотим вывести разное приветствие, в зависимости от того, мужчина это или женщина. Для этого пригодятся операторы if и else.

Из примера видно, что условная конструкция состоит из трех частей:

условие if — выражение, которое возвращает истинный или ложный результат;

блок if — строки кода, которые будут выполнены, если условие вернуло истину;

блок else — строки кода, которые будут выполнены, если условие вернуло ложь.

Обратите внимание, что каждый блок обрамляется и ограничивается фигурными скобками.

Чтобы закрепить материал, пройдите этот тренажёр. В нём подробно рассказывается, как использовать условные конструкции.

Выражения

Выражение в языке программирование — это минимальный блок кода, имеющий значение (результат). Звучит несколько сложно, но на практике почти каждая строка сценария является выражением.

Важно усвоить одно главное свойство выражений — они всегда имеют результат. Также различают два типа выражений: логические и арифметические. Результатом выполнения логического выражение всегда будет истина или ложь. Например, результатом выражения 2 > 3 будет «ложь», потому что два, очевидно, меньше трех.

Другим видом выражений является арифметическое выражение. Результатом выражения $amount = 2 + 2 будет 4.

В каких случаях может понадобиться знать результат выражения? Выражения особенно полезны в условиях, то есть когда мы хотим выполнить или не выполнять какое-либо действие, в зависимости от результата выражения. Выражения также можно объединять между собой таким образом, что несколько отдельных выражений в итоге вычисляются как одно.

Обратите внимание на ключевое слово *and* — это оператор, который объединяет два отдельных выражения в одно новое. Это новое выражение будет иметь значение «истина», только если оба отдельных выражений будут истинны. Иными словами, наше условие будет выполняться только для посетителей мужского пола *И* возрастом старше восемнадцати. То есть несовершенолетние мальчики, равно как и совершенолетние девочки никакой картинки не увидят.

Понятие алгоритма

Многие из нас что-то слышали про алгоритмы на уроках информатики в школе. К сожалению, далеко не все школьные знания остаются с нами после её окончания. Тем не менее, понимание, что такое алгоритм, и умение эти самые алгоритмы строить — это очень важные навыки, без которых не получится решать даже относительно простые задачи.

Если говорить простыми словами, то алгоритм — это просто очень подробный план работы. Мы все в течение жизни что-нибудь планируем: отпуск, какое-нибудь мероприятие, своё самостоятельное обучение и тому подобное. От простого списка шагов алгоритм отличает существование условий и повторяющихся действий. Если вы в состоянии создать хороший, подробный алгоритм для реализации, скажем, какой-нибудь возможности на сайте, то можно считать, что половина дела уже сделана!

Разберём для примера один алгоритм средней сложности. На сайт нужно добавить форму обратной связи. Пользователь может заполнить эту форму, указать там свои контактные данные и написать сообщение. Информация из заполненной формы отправляется на электронную почту владельцу сайта.

Большинство программ всегда имеют отдельное окно настроек, где можно указать все основные параметры этого приложения. В PHP тоже есть свои настройки, только меняются они не через интерфейс, а путём редактирования специального файла — php.ini. В файле php.ini задаются все настройки работы PHP. Из того, что нас будет интересовать в первую очередь — это режим управления ошибками, подключение дополнительных возможностей, настройки сессий и кук.

Назначение протокола HTTP и его методов. Передача данных формы PHP-сценарию. Синтаксис определения и использования функций в PHP.

HTTP – это протокол передачи информации в интернете, который расшифровывается как «протокол передачи гипертекста» (HyperText Transfer Protocol). Например, браузер отправляет единичный запрос на сервер, который в свою очередь обрабатывает его, формирует ответ и делится с браузером этим ответом – ресурсами в виде данных.

Для чего нужен HTTP

Благодаря взаимодействию клиента (локального компьютера с браузером) и сервера (высокопроизводительного специального компьютера) в сети можно передавать данные. Изначально HTTP использовался только для гипертекстовых документов, но сейчас он может передавать любую информацию. Гипертекстовые документы также могут содержать гиперcсылки, при нажатии на которые формируется новый http-запрос, в ответе на который может содержаться другой гипертекстовый документ. Таким образом мы перемещаемся по страницам в интернете.

Как он работает

HTTP-запрос состоит из трех элементов:

стартовой строки, которая задает параметры запроса или ответа,

заголовка, который описывает сведения о передаче и другую служебную информацию.

тело (его не всегда можно встретить в структуре). Обычно в нем как раз лежат передаваемые данные. От заголовка тело отделяется пустой строкой.

Метод (иногда его называют HTTP-глаголом) – описывает, какое именно действие нужно совершить со страницей.

Чаще всего используют GET и POST, они нужны для чтения и отправки данных на сервер. Например вы зашли в соцсеть, увидели пост и решили оставить комментарий. Или зашли в интернет-магазин, решили что-то купить и оставили данные карты.

URL (Uniform Resource Locator) – единообразный идентификатор ресурса, идентифицирует ресурс и определяет его точное местоположение. Именно с помощью URL записаны ссылки в интернете.

HTTP-ответ строится примерно по тому же принципу, что и запрос:

*HTTP/Версия + Код состояния + Пояснение*

Версия совпадает с версией в запросе.

Код состояния показывает статус запроса. Это трехзначное число, благодаря которому можно узнать, получен ли запрос, обработан ли он, какие ошибки есть. Например, одна из самых известных ошибок – 404 – сообщает о том, что сервер не нашел ресурс по адресу. Возможно, в запросе опечатка, ошибка или он не соответствует протоколу.

Чем отличаются HTTP и HTTPS

HTTPS – это расширение протокола HTTP, которое обеспечивает защиту передаваемых данных. Для сайта это важный параметр, так как шифрование позволяет ему обезопасить информацию, которую туда вводят люди (пароли, реквизиты кредитных карт), от хакерских атак. HTTP-протокол передает данные в открытую, поэтому их легко перехватить.

HTTPS защищен SSL-сертификатом. Благодаря ему уязвимые данные шифруются сначала на клиенте (браузере, например) в результате чего они становятся похожи на случайный набор символов и только потом отправляются на сервер. Каждый раз при HTTP-запросе шифр меняется, поэтому успеть подобрать ключ и украсть данные довольно трудно.

Через формы можно отправлять как простую текстовую информацию, так и файлы.

Большую часть времени программирования на PHP вы будете так или иначе работать с формами и данными из них.

HTML описывает то, из каких элементов состоит форма, и как она выглядит. Но без принимающей стороны, то есть сервера, который принимает эти данные и обрабатывает их нужным образом, создавать формы нет никакого смысла.

PHP содержит множество средств для работы с формами. Это позволяет очень просто решать типичные задачи, которые часто возникают в веб-программировании:

Регистрация и аутентификация пользователя;

Отправка комментариев на форумах и социальных сетях;

Оформление заказов.

Практически любой современный сайт содержит как минимум несколько разных HTML-форм.

Отправка формы

Рассмотрим один типичный пример — форма обратной связи. Для связи пользователей с авторами сайта, как правило, используются формы обратной связи, где человек указывает имя, почту для обратной связи и текст своего сообщения.

Это очень простая форма, состоящая из трёх полей и одной кнопки отправки.

Почти весь приведённый код описывает внешний вид и содержание формы, но следует обратить внимание на два атрибута тега <form>, которые нужны для указания на способ обработки данных:

method — этот атрибут используется для определения метода HTTP, который будет использован для передачи данных на сервер. Вы уже знакомы с HTTP-методом GET, предписывающим серверу просто вернуть определённый документ.

Метод POST сообщает о намерении передать на сервер некоторую информацию, что, впрочем, не отменяет последующее получение контента.

action — содержит адрес PHP-скрипта, который должен обработать эту форму.

После нажатия на кнопку «отправить», браузер выполняет POST запрос со введёнными данными на адрес, указанный в атрибуте action.

Обработка формы

После отправки формы управление передаётся PHP-скрипту, который должен получить переданные данные, выполнить с ними какие-либо действия (например, сохранить в базе данных) и показать результат.

Результатом может быть какое-нибудь сообщение об успешном завершении операции, например, «ваши данные успешно отправлены».

Поэтому требуется в первую очередь научиться получать данные из формы в сценарии.

В PHP это делается легко — все данные из формы находятся в глобальном ассоциативном массиве $\_POST. Этот массив всегда будет неявно присутствовать в сценарии, если он был загружен по методу POST.

Каждое поле из формы будет находиться в массиве, где ключом будет значение атрибута name, а значением содержимое поля.

Функция isset служит для определения, существует ли переданная ей переменная. Так мы проверяем, что сценарий загружен методом POST, то есть была отправлена форма.

Как правило, после обработки формы в PHP, сценарий должен переадресовать пользователя на другую страницу. Это связано с тем, что если форма была отправлена через метод POST, то после обновления страницы данные будут отправлены ещё раз, а это, в большинстве случаев, нежелательное поведение.

Отправка файлов

Кроме текстовой информации, существует возможность отправлять на сервер файлы любых типов.

PHP автоматически сохраняет все загруженные файлы во временную папку на сервере. Но хранить там файлы нельзя, потому что эта директория периодически очищается, и ссылку на такой файл нельзя дать на сайте. Решение здесь только одно — переместить загруженный файл в другую папку. Перемещение файла всегда выполняют сразу после загрузки.

Для начала нужно убедиться, что в рабочей директории проекта существует папка для хранения загруженных файлов. Пусть она называется uploads.

Перемещение загруженного файла

Для перемещения файла нужно знать, где он находится сейчас, и адрес папки, в которую он будет переноситься.

Функция move\_uploaded\_file() выполняет два действия:

Проверяет, что файл действительно загружен через форму.

Перемещает загруженный файл по новому адресу.

Валидация формы

Валидация формы — это проверка содержимого её полей. Задача такой проверки — убедиться, что необходимые поля заполнены, а значения в них соответствуют ожидаемому формату.

Так, например, при регистрации пользователя на сайте, он должен заполнить поля с адресом электронной почты и придумать себе пароль. Оба поля обязательны к заполнению, но значение из поля email также должно быть корректным email-адресом.

Помимо текстовых значений формы, можно проверять формат и размер загружаемых файлов.

Общий подход к валидации

При выполнения валидации любой формы порядок действий будет всегда одним:

Сформировать массив с именами полей, обязательных к заполнению.

Сформировать массив с правилами для валидации формата полей.

Получить значения всех полей.

Приведём пример синтаксиса, используемого для описания функций:

Внутри функции можно использовать любой корректный PHP-код, в том числе другие функции и даже объявления классов.

Имена функций следуют тем же правилам, что и другие метки в PHP. Корректное имя функции начинается с буквы или знака подчёркивания, за которым следует любое количество букв, цифр или знаков подчёркивания. В качестве регулярного выражения оно может быть выражено так: ^[a-zA-Z\_\x80-\xff][a-zA-Z0-9\_\x80-\xff]\*$.

Функции не обязаны быть определены до их использования, *исключая* тот случай, когда функции определяются условно, как это показано в двух последующих примерах.

В случае, когда функция определяется в зависимости от какого-либо условия, например, как это показано в двух приведённых ниже примерах, обработка описания функции должна *предшествовать* её вызову.

Все функции и классы PHP имеют глобальную область видимости - они могут быть вызваны вне функции, даже если были определены внутри и наоборот.

PHP не поддерживает перегрузку функции, также отсутствует возможность переопределить или удалить объявленную ранее функцию.

Замечание: Имена функций регистронезависимы для символов ASCII от A до Z, тем не менее, предпочтительнее вызывать функции так, как они были объявлены.

Можно вызывать функции PHP рекурсивно.

**Выражения. Логические и строковые выражения, их назначение и способы применения.**

Выражения - это самые важные строительные элементы PHP. Почти всё, что вы пишете в PHP, является выражением. Самое простое и точное определение выражения - "все что угодно, имеющее значение".

Немного более сложными примерами выражений являются функции.

Разумеется, значения в PHP не обязаны быть целочисленными, и очень часто ими не являются. PHP поддерживает четыре типа скалярных значений: целочисленные (int), с плавающей точкой (float), строковые значения (string) и булевы (bool) значения (скалярными являются значения, которые вы не можете 'разбить' на меньшие части, в отличие, например, от массивов). PHP поддерживает также два комбинированных (не скалярных) типа: массивы и объекты. Любое значение такого типа может присваиваться переменной или возвращаться функцией.

Однако PHP, как и многие другие языки, понимает гораздо больше выражений. PHP - это язык, ориентированный на выражения и рассматривающий почти все как выражение.

Другой хороший пример ориентированности на выражения - префиксный и постфиксный инкремент и декремент

Очень распространённым типом выражений являются выражения сравнения. Результатом вычислений являются false (ложь) или true (истина). PHP поддерживает операции сравнения > (больше), >= (больше либо равно), == (равно), != (не равно), < (меньше) и <= (меньше либо равно). Он также поддерживает операторы строгого равенства: === (равно и одного типа) и !== (не равно или не одного типа). Чаще всего эти выражения используются в операторах условного выполнения, таких как if.

Если значением первого подвыражения является true (не ноль), то выполняется второе подвыражение, которое и будет результатом условного выражения. В противном случае будет выполнено третье подвыражение и его значение будет результатом.

Последнее, что стоит упомянуть, это истинность значения выражений. Во многих случаях, как правило, в условных операторах и циклах, вас может интересовать не конкретное значение выражения, а только его истинность (значение true или false). Константы true и false (регистронезависимые) - это два возможных булевых значения. При необходимости выражение автоматически преобразуется в булев тип. Подробнее о том, как это происходит, смотрите в разделе о приведении типов.

PHP предоставляет полную и мощную реализацию выражений, и их полное документирование выходит за рамки этого руководства. Вышеприведённые примеры должны дать вам представление о том, что они из себя представляют и как вы сами можете создавать полезные выражения. Далее, для обозначения любого верного выражения PHP в этой документации мы будем использовать сокращение *expr*.

*Переменные*

В РНР переменные начинаются со знака доллара ($). За этим знаком может следовать любое количество буквенно-цифровых символов и символов подчеркивания, но первый символ не может быть цифрой или подчеркиванием. Следует также помнить, что имена переменных в РНР чувствительны к регистру, в отличие от ключевых слов.

При объявлении переменных в РНР не требуется явно указывать тип переменной, при этом одна и та же переменная может иметь на протяжении программы разные типы.

Переменная инициализируется в момент присваивания ей значения и существует до тех пор, пока выполняется программа. Т.е., в случае web-страницы это означает, что до тех пор, пока не завершен запрос.

*Внешние переменные*

После того, как запрос клиента проанализирован веб-сервером и передан РНР машине, последняя устанавливает ряд переменных, которые содержат данные, относящиеся к запросу и доступны все время его выполнения. Сначала РНР берет переменные окружения Вашей системы и создает переменные с теми же именами и значениями в окружении сценария РНР для того чтобы сценариям, расположенным на сервере были доступны особенности системы клиента. Эти переменные помещаются в ассоциативный массив $HTTP\_ENV\_VARS

*Константы*

Константы объявляются в РНР при помощи функции define(): define(CONSTANT, value)

Первый параметр этой функции – имя константы, второй – её значение. При использовании константы на неё ссылаются по имени:

*Выражения*

Действительно, практически все, что вы пишете в программе — это выражение.

Совершенно точно можно сказать, что у любого выражения есть тип его

значения.

*Логические выражения*—это выражения, у которых могут быть только два значения: ложь и истина. В качестве истины может выступать любое ненулевое число, непустая строка и т. д., а под ложью подразумевается все остальное.

*Строковые выражения*

Строки в РНР — одни из самых основных объектов. Они могут содержать текст вместе с символами форматирования или даже бинарные данные. Определение строки в кавычках или апострофах может начинаться на одной строке, а завершаться — на другой.

*Строка в апострофах*

Если строка заключена в апострофы (например, 'строка'), то она трактуется почти в точности так же, как записана, за исключением двух специальных последовательностей символов:

последовательность \ ' трактуется РНР как апостроф и предназначена для вставки апострофа в строку, заключенную в апострофы;

последовательность \\ трактуется как один обратный слэш и позволяет вставлять в строку этот символ.

*Типы данных в РНР. Преобразование типов*

В РНР существуют набор основных типов данных, которые могут явно указываться при работе с переменными:

Integer (целый)—Используется для целых чисел

Double (двойной точности) — Используется для действительных чисел

String (строковый) — Используется для строк символов

Array (массив) — Используется для хранения нескольких элементов данных одного типа

Object (объект) — Используется для хранения экземпляров классов

Boolean (логический)—Используется дляхранения логических (true|false)

Есть функция gettype(), возвращающая тип, который РНР назначил переменной:

В первом случае РНР вернет string, во втором integer.

Существует также функция settype(), которая явно устанавливает тип:

Выполнение этого фрагмента кода приведет к такому же результату, как и предыдущего.

Кроме функции settype() преобразование типов в РНР можно осуществлять при помощи операторов преобразования типов. Преобразование типов осуществляется путем указания перед переменной ее нового типа, взятого в скобки:

$var = (int)$var;

Соответственно, выполнение следующего кода приведет к тому, что РНР вернет integer:

2. Управляющие структуры языка php

Управляющие структуры — это языковые структуры, которые позволяют управлять потоком выполнения программы или сценария. Их можно разделить на условные структуры (или структуры ветвления) и структуры повторения, или циклы.

К операторам выбора относят: условный оператор *(if...else)* и переключатель *(switch)*. Синтаксис условного оператора:

*if(condition)statement 1elsestatement 2*

Условие condition может быть любым выражением. Если оно истинно, то выполняется оператор statement 1. В противном случае выполняется оператор statement 2. Допустима сокращенная форма записи условного оператора, в которой отсутствуют *else* и оператор statement 2.

Проверка дополнительных условий возможна при помощи оператора *elseif*. Оператор *if* может включать сколько угодно блоков *elseif*, но *else* в каждом *if* может быть только один. Как правило, в конструкциях *if…elseif…else* оператор *else* определяет, что нужно делать, если никакие другие условия не являются true. Однако, вообще говоря, использование оператора *elseif* довольно сильно ухудшает читабельность кода, и лучше в этом случае пользоваться переключателем (*switch*).Переключатель *switch*является наиболее удобным средством для организации ветвления. Синтаксис переключателя таков:

Управляющая структура *switch* передает управление тому из помеченных *case* операторов, для которого значение константного выражения совпадает со значением переключающего выражения. Если значение переключающего выражения не совпадает ни с одним из константных выражений, то выполняется переход к оператору, помеченному меткой *default*. В каждом переключателе может быть не более одной метки *default*, однако она может отсутствовать вообще.

Альтернативный способ определения массивов состоит в использовании конструкции array():

Для явного указания индексов в этом случае применяется оператор =>:

Операторы языка PHP.

Оператором называется нечто, принимающее одно или более значений (или выражений, если говорить на жаргоне программирования), и вычисляющее новое значение (таким образом, вся конструкция может рассматриваться как выражение).

Операторы можно сгруппировать по количеству принимаемых ими значений. Унарные операторы принимают только одно значение, например, ! (оператор логического отрицания) или ++ (инкремент). Бинарные операторы принимают два значения; это, например, знакомые всем арифметические операторы+ (плюс) и - (минус), большинство поддерживаемых в PHP операторов входят именно в эту категорию. Ну и, наконец, есть всего один тернарный оператор, ? :, принимающий три значения, обычно его так и называют -- "тернарный оператор" (хотя, возможно, более точным названием было бы "условный оператор").

Если операторы имеют равный приоритет, то будут ли они выполняться справа налево или слева направо определяется их ассоциативностью.

Ассоциативность имеет смысл только для двоичных (и тернарных) операторов.

Использование скобок, кроме случаев когда они строго необходимы, может улучшить читаемость кода, группируя явно, а не полагаясь на приоритеты и ассоциативность.

Приоритет и ассоциативность оператора определяет только то, как группируется выражение, а не порядок его вычисления. Обычно PHP не указывает, в каком порядке вычисляются выражения и кода, который предполагает специфичный порядок вычисления следует избегать, потому, что поведение может меняться в разных версиях PHP или в зависимости от окружающего кода.

Операция деления ("/") возвращает число с плавающей точкой, кроме случая, когда оба значения являются целыми числами (или строками, которые преобразуются в целые числа), которые делятся нацело - в этом случае возвращается целое значение. Для целочисленного деления используйте intdiv().

При делении по модулю операнды преобразуются в целые числа (int) (путём удаления дробной части) до начала операции. Для деления по модулю чисел с плавающей точкой используйте fmod().

Базовый оператор присваивания обозначается как "=". На первый взгляд может показаться, что это оператор "равно". На самом деле это не так. В действительности оператор присваивания означает, что левый операнд получает значение правого выражения, (то есть устанавливается значением).

В дополнение к базовому оператору присваивания имеются "комбинированные операторы" для всех бинарных арифметических операций, операций объединения массивов и строковых операций, которые позволяют использовать некоторое значение в выражении, а затем установить его как результат данного выражения.

Исключением из обычного для PHP способа присваивания по значению являются объекты (object), которые присваиваются по ссылке. Принудительно скопировать объекты по значению можно с помощью специального ключевого слова clone.

Присваивание по ссылке ¶

xОператор new автоматически возвращает ссылку, поэтому присвоение результата операции new по ссылке является ошибкой.

Операторы побитового присваивания ¶

Другие операторы присваивания ¶

**Арифметические и строковые операторы. Операции присваивания, инкремента, декремента. Битовые операции. Логические и условные операции.**

ФяОперация деления ("/") возвращает число с плавающей точкой, кроме случая, когда оба значения являются целыми числами (или строками, которые преобразуются в целые числа), которые делятся нацело - в этом случае возвращается целое значение. Для целочисленного деления используйте intdiv().

При делении по модулю операнды преобразуются в целые числа (int) (путём удаления дробной части) до начала операции. Для деления по модулю чисел с плавающей точкой используйте fmod().

Замечание: Операторы инкремента/декремента влияют только на числа и строки. Массивы, объекты, булевы значения и ресурсы не будут изменены. Декремент null также не даст никакого эффекта, однако инкремент даст значение 1.

Инкрементирование или декрементирование булевых переменных не приводит ни к какому результату.

Побитовые операторы позволяют считывать и устанавливать конкретные биты целых чисел.

Побитовый сдвиг в PHP - это арифметическая операция. Биты, сдвинутые за границы числа, отбрасываются. Сдвиг влево дополняет число нулями справа, сдвигая в то же время знаковый бит числа влево, что означает что знак операнда не сохраняется. Сдвиг вправо сохраняет копию сдвинутого знакового бита слева, что означает что знак операнда сохраняется.

Используйте скобки для обеспечения необходимого приоритета операторов. Например, $a & $b == true сначала проверяет на равенство, а потом выполняет побитовое и; тогда как ($a & $b) == true сначала выполняет побитовое и, а потом выполняет проверку на равенство.

Если оба операнда для &, | и ^ строки, то операция будет производиться с кодами ASCII всех символов строки и в результате вернёт строку. Во всех остальных случаях, оба операнда будут преобразованы к целому и результатом будет целое число.

Если операнд для ~ строка, то операция будет производиться с кодами ASCII всех символов строки и в результате вернёт строку, в ином случае как операнд, так и результат, будут считаться целыми.

Оба операнда и результат выполнения << и >> всегда считаются за целое.

**Циклы в PHP. Операторы организации циклов.**

На втором месте по частоте использования, после конструкций условий (условных операторов), находятся циклы.

Циклы позволяют повторять определенное (и даже неопределенное - когда работа цикла зависит от условия) колличество раз различные операторы.

Вычисляется значение логического выражения.

Если значение истинно, выполняется тело цикла, в противном случае - переходим на следующий за циклом оператор.

Синтаксис цикла с предусловием:

while (логическое\_выражение)   
 инструкция;

В данном случае телом цикла является инструкция. Обычно тело цикла состоит из большого числа операторов.

Подобно конструкции условного оператора if, можно группировать операторы внутри тела цикла while, используя следующий альтернативный синтаксис:

while (логическое\_выражение):  
 инструкция;  
 ...  
 endwhile;

Цикл с постусловием do while

В отличие от цикла while, этот цикл проверяет значение выражения не до, а после каждого прохода (итерации). Таким образом, тело цикла выполняется хотя бы один раз. Синтаксис цикла с постусловием такой:

while (логическое\_выражение);

После очередной итерации проверяется, истинно ли *логическое\_выражение*, и, если это так, управление передается вновь на начало цикла, в противном случае цикл обрывается.

Цикл со счетчиком for

Цикл со счетчиком используется для выполнения тела цикла определенное число раз. С помощью цикла for можно (и нужно) создавать конструкции, которые будут выполнять действия совсем не такие тривиальные, как простая переборка значения счетчика.

Синтаксис цикла for такой:

for (инициализирующие\_команды; условие\_цикла; команды\_после\_итерации) { тело\_цикла; }

Цикл for начинает свою работу с выполнения инициализирующих\_команд. Данные команды выполняются только один раз. После этого проверяется условие\_цикла, если оно истинно (true), то выполняется тело\_цикла. После того, как будет выполнен последний оператор тела, выполняются команды\_после\_итерации. Затем снова проверяется условие\_цикла. Если оно истинно (true), выполняется тело\_цикла и команды\_после\_итерации, и.т.д.

Рассмотренный пример (да и вообще любой цикл for) можно реализовать и через while, только это будет выглядеть не так изящно и лаконично.

Для цикла for имеется и альтернативный синтаксис:

for(инициализирующие\_команды; условие\_цикла; команды\_после\_итерации):  
 операторы;  
 endfor;

Цикл перебора массивов foreach

В PHP4 появился еще один специальный тип цикла - foreach. Данный цикл предназначен специально для перебора массивов.

Синтаксис цикла foreach выглядит следующим образом:

foreach (массив as $ключ=>$значение)  
 команды;

Здесь команды циклически выполняются для каждого элемента массива, при этом очередная пара ключ=>значение оказывается в переменных $ключ и $значение. Приведем пример работы цикла foreach:

У цикла foreach имеется и другая форма записи, которую следует применять, когда нас не интересует значение ключа очередного элемента. Выглядит она так:

foreach (массив as $значение)  
 команды;

В этом случае доступно лишь значение очередного элемента массива, но не его ключ. Это может быть полезно, например, для работы с массивами-списками:

Внимание: Цикл foreach оперирует не исходным массивом, а его копией. Это означает, что любые изменения, которые вносятся в массив, не могут быть "видны" из тела цикла. Что позволяет, например, в качестве массива использовать не только переменную, но и результат работы какой-нибудь функции, возвращающей массив (в этом случае функция будет вызвана всего один раз - до начала цикла, а затем работа будет производиться с копией возвращенного значения).

Конструкция break

Очень часто для того, чтобы упростить логику какого-нибудь сложного цикла, удобно иметь возможность его прервать в ходе очередной итерации (к примеру, при выполнении какого-нибудь особенного условия). Для этого и существует конструкция break, которая осуществляет немедленный выход из цикла. Она может задаваться с одним необязательным параметром - числом, которое указывает, из какого вложенного цикла должен быть произведен выход.

break(номер\_цикла); // Для вложенных циклов (указывается номер прерываемого цикла)

Если нам нужно прервать работу определенного (вложенного) цикла, то нужно передать конструкции break параметр - *номер\_цикла*, например, *break(1)*. Нумерация циклов выглядит следующим образом:

for (...) // Третий цикл  
 {  
   for (...) // Второй цикл  
   {  
      for (...) // Первый цикл  
      {  
      }  
   }  
 }

Конструкция continue

Конструкция continue так же, как и break, работает только "в паре" с циклическими конструкциями. Она немедленно завершает текущую итерацию цикла и переходит к новой (конечно, если выполняется условие цикла для цикла с предусловием). Точно так же, как и для break, для continue можно указать уровень вложенности цикла, который будет продолжен по возврату управления.  
 В основном continue позволяет вам сэкономить количество фигурных скобок в коде и увеличить его удобочитаемость. Это чаще всего бывает нужно в циклах-фильтрах, когда требуется перебрать некоторое количество объектов и выбрать из них только те, которые удовлетворяют определенным условиям.

Грамотное использование break и continue позволяет заметно улучшить "читабельность" кода и количество блоков else.

Работа с циклическими операторами на языке PHP.

Цикл — это конструкция языка, которая позволяет выполнить блок кода больше одного раза.

Мы привыкли, что наши сценарии выполняются линейно: сверху вниз, строчка за строчкой, инструкция за инструкцией. Но что делать, если надо повторить какую-нибудь инструкцию несколько раз?

Но, во первых, такой способ заставляет писать много кода. Во-вторых, что если требуется вывести последовательность из миллиона чисел? И, наконец, бывают ситуации, когда заранее неизвестно сколько раз нужно выполнить определённую инструкцию.

Использование циклов значительно упрощает и укорачивает код. Также циклы незаменимы в ситуациях, когда заранее неизвестно сколько раз должен выполниться блок кода. Такое число может зависеть от множества условий и вычисляться в момент выполнения сценария.

Выражения часто используются в циклах следующим образом: выражение помещается на место <условие цикла=""> и определяет, будет ли исполнен блок кода — <тело цикла="">.

Если выражение из условия цикла возвращает истину, то выполнение сразу перейдёт к блоку «тело цикла», если же оно вернёт ложь — то тело цикла исполнено не будет и сценарий продолжит выполняться как обычно, со следующей строки после цикла.

Выходит, что циклы имеют такое название, потому что как бы «зацикливают» обычное, линейное исполнение на своем блоке кода и не дают сценарию выполняться дальше, пока условие цикла будет истинным.

Важно понимать последовательность, в которой исполняется код при использовании циклов.

Обычное исполнение кода, строчка за строчкой, пока не встретился цикл.

Встретился цикл: выполняем условие цикла.

Если условие вернуло ложь: выходим из цикла, выполняем строчку после него и продолжаем линейное исполнение.

Если условие вернуло истину: выполняем всё тело цикла.

Повторяем пункт 2.

Каждая последовательность из шагов 2-4, то есть очередное выполнение блока кода в теле цикла — называется итерацией.

Количество итераций должно быть конечным, то есть, как правило, бесконечное выполнение одного блока кода не входит в наши планы.

А значит необходимо заранее предусмотреть ситуацию, когда истинное условие станет ложным.

Данный цикл в своём теле содержит две инструкции. Первая выводит на экран цифру из переменной. Вторая инструкция увеличивает значение переменной на единицу. Теперь вопрос: сколько раз будет исполнен такой цикл?

Циклы выполняются, пока их условие остаётся истинным, а в нашем условии значение переменной должно быть меньше десяти. Так как начальное значение переменной — единица, то не сложно посчитать, что цикл выполнится ровно девять раз. На десятый раз значение переменной $last\_num станет равно десяти и условие $last\_num < 10 перестанет быть истинным.

При использовании циклов есть возможность использования операторов break и continue. Первый из них прерывает работу всего цикла, а второй - только текущей итерации.

Цикл с предусловием while работает по следующим принципам:

Вычисляется значение логического выражения;

Если значение истинно, выполняется тело цикла, в противном случае - переходим на следующий за циклом оператор.

Синтаксис цикла с предусловием:

while (логическое\_выражение)

инструкция;

В отличие от цикла while, этот цикл проверяет значение выражения не до, а после каждого прохода (итерации). Таким образом, тело цикла выполняется хотя бы один раз. Синтаксис цикла с постусловием такой:

do

{

тело\_цикла;

}

while (логическое\_выражение);

После очередной итерации проверяется, истинно ли логическое\_выражение, и, если это так, управление передается вновь на начало цикла, в противном случае цикл обрывается.

Альтернативного синтаксиса для do-while разработчики PHP не предусмотрели (видимо, из-за того, что, в отличие от прикладного программирования, этот цикл довольно редко используется при программировании web-приложений).

Массивы. Ассоциативные массивы. Базовые инструкции работы с массивами: list(), array(). Списки.

На самом деле массив в PHP - это упорядоченное отображение, которое устанавливает соответствие между *значением* и *ключом*. Этот тип оптимизирован в нескольких направлениях, поэтому вы можете использовать его как собственно массив, список (вектор), хеш-таблицу (являющуюся реализацией карты), словарь, коллекцию, стек, очередь и, возможно, что-то ещё. Так как значением массива может быть другой массив PHP, можно также создавать деревья и многомерные массивы.

Объяснение этих структур данных выходит за рамки данного справочного руководства, но вы найдёте как минимум один пример по каждой из них. За дополнительной информацией вы можете обратиться к соответствующей литературе по этой обширной теме.

Синтаксис ¶

Определение при помощи array() ¶

Массив (тип array) может быть создан языковой конструкцией array(). В качестве параметров она принимает любое количество разделённых запятыми пар key => value (ключ => значение).

array( key => value, key2 => value2, key3 => value3, ... )

Запятая после последнего элемента массива необязательна и может быть опущена. Обычно это делается для однострочных массивов, то есть array(1, 2) предпочтительней array(1, 2, ). Для многострочных массивов с другой стороны обычно используется завершающая запятая, так как позволяет легче добавлять новые элементы в конец массива.

Замечание:

Существует короткий синтаксис массива, который заменяет array() на [].

key может быть либо типа int, либо типа string. value может быть любого типа.

Дополнительно с ключом key будут сделаны следующие преобразования:

Строки (string), содержащие целое число (int) (исключая случаи, когда число предваряется знаком +) будут преобразованы к типу int. Например, ключ со значением "8" будет в действительности сохранён со значением 8. С другой стороны, значение "08" не будет преобразовано, так как оно не является корректным десятичным целым.

Числа с плавающей точкой (float) также будут преобразованы к типу int, то есть дробная часть будет отброшена. Например, ключ со значением 8.7 будет в действительности сохранён со значением 8.

Тип bool также преобразовываются к типу int. Например, ключ со значением true будет сохранён со значением 1 и ключ со значением false будет сохранён со значением 0.

Тип null будет преобразован к пустой строке. Например, ключ со значением null будет в действительности сохранён со значением "".

Массивы (array) и объекты (object) *не могут* использоваться в качестве ключей. При подобном использовании будет генерироваться предупреждение: Недопустимый тип смещения (Illegal offset type).

Если несколько элементов в объявлении массива используют одинаковый ключ, то только последний будет использоваться, а все другие будут перезаписаны.

Так как все ключи в вышеприведённом примере преобразуются к 1, значение будет перезаписано на каждый новый элемент и останется только последнее присвоенное значение "d".

Массивы в PHP могут содержать ключи типов int и string одновременно, так как PHP не делает различия между индексированными и ассоциативными массивами.

Параметр key является необязательным. Если он не указан, PHP будет использовать предыдущее наибольшее значение ключа типа int, увеличенное на 1.

Возможно указать ключ только для некоторых элементов и пропустить для других:

Как вы видите последнее значение "d" было присвоено ключу 7. Это произошло потому, что самое большое значение ключа целого типа перед этим было 6.

Доступ к элементам массива с помощью квадратных скобок ¶

Доступ к элементам массива может быть осуществлён с помощью синтаксиса array[key].

Создание/модификация с помощью синтаксиса квадратных скобок ¶

Существующий массив может быть изменён путём явной установкой значений в нем.

Это выполняется присвоением значений массиву (array) с указанием в скобках ключа. Кроме того, ключ можно опустить, в результате получится пустая пара скобок ([]).

$arr[key] = value; $arr[] = value; // key может быть int или string // value может быть любым значением любого типа

Если массив *$arr* ещё не существует или для него задано значение null или false, он будет создан. Таким образом, это ещё один способ определить массив array. Однако такой способ применять не рекомендуется, так как если переменная *$arr* уже содержит некоторое значение (например, значение типа string из переменной запроса), то это значение останется на месте и [] может на самом деле означать доступ к символу в строке. Лучше инициализировать переменную путём явного присваивания значения.

Для изменения определённого значения просто присвойте новое значение элементу, используя его ключ. Если вы хотите удалить пару ключ/значение, вам необходимо использовать функцию unset().

($arr); // Это удаляет массив полностью  
Деструктуризация массива ¶

Массивы могут быть деструктурированы с помощью языковых конструкций [] (начиная с PHP 7.1.0) или list(). Эти конструкции могут быть использованы для деструктуризации массива на отдельные переменные.

Деструктуризация массива может быть использована в конструкции foreach для деструктуризации многомерного массива во время итерации по нему.

Элементы массива будут игнорироваться, если переменная не указана. Деструктуризация массива всегда начинается с индекса 0.

Начиная с PHP 7.1.0, ассоциативные массивы также могут быть деструктурированы. Это позволяет легче выбирать нужный элемент в массивах с числовым индексом, так как индекс может быть указан явно.

Деструктуризация массива может быть использована для простой замены двух переменных местами.

Замечание:

Оператор ... не поддерживается в присваиваниях.

Полезные функции ¶

Для работы с массивами существует достаточное количество полезных функций. Смотрите раздел функции для работы с массивами.

Замечание:

Функция unset() позволяет удалять ключи массива. Обратите внимание, что массив *НЕ* будет переиндексирован. Если вы действительно хотите поведения в стиле "удалить и сдвинуть", можно переиндексировать массив используя array\_values().

Управляющая конструкция foreach существует специально для массивов. Она предоставляет возможность легко пройтись по массиву.

Это неверно, хотя и работает. Причина в том, что этот код содержит неопределённую константу (bar), а не строку ('bar' - обратите внимание на кавычки). Это работает, потому что PHP автоматически преобразует *"голую строку"* (не заключённую в кавычки строку, которая не соответствует ни одному из известных символов языка) в строку со значением этой "голой строки". Например, если константа с именем bar не определена, то PHP заменит bar на строку 'bar' и использует её.

Обратите внимание, что E\_ERROR - это такой же верный идентификатор, как и bar в первом примере.

поскольку E\_ERROR соответствует 1 и т.д.

Так что же в этом плохого?

Когда-нибудь в будущем команда разработчиков PHP возможно пожелает добавить ещё одну константу или ключевое слово, либо константа из другого кода может вмешаться и тогда у вас могут возникнуть проблемы. Например, вы уже не можете использовать таким образом слова empty и default, поскольку они являются зарезервированными ключевыми словами.

Замечание: Повторим, внутри строки (string), заключённой в двойные кавычки, корректно не окружать индексы массива кавычками, поэтому "$foo[bar]" является верной записью. Более подробно почему - смотрите вышеприведённые примеры, а также раздел по обработке переменных в строках.

Вышеприведённый код покажет 2 ключа с именем 'AA', хотя один из них на самом деле имеет имя '\0A\0A'.

Если вы преобразуете в массив значение null, вы получите пустой массив.

Сравнение ¶

Массивы можно сравнивать при помощи функции array\_diff() и операторов массивов.

Распаковка массива ¶

Массив с префиксом ... будет распакован во время определения массива. Только массивы и объекты, которые реализуют интерфейс Traversable, могут быть распакованы. Распаковка массива с помощью ... доступна, начиная с PHP 7.4.0.

Массив можно распаковывать несколько раз и добавлять обычные элементы до или после оператора ...:

Операции над массивами: удаление, слияние, сортировка.

*Массив* — это коллекция переменных, индексированных и увязанных друг с другом в одну суперпеременную, обеспечивающую простой доступ и допускающую возможность ссылаться на нее, которая предоставляет удобный способ обмена сразу несколькими значениями между строками кода, функциями и даже страницами. В этой статье приведен обзор внутреннего устройства массивов и описаны все встроенные функции PHP, предназначенные для манипулирования массивами. Но прежде чем заняться углубленным изучением этой темы, целесообразно рассмотреть наиболее распространенные способы использования массивов в реальном коде PHP.

В виде массивов представлены многие встроенные переменные среды PHP (в качестве примера можно указать массив $\_SESSION, который содержит все имена переменных и значения, распространяющиеся со страницы на страницу с помощью механизма сеансов PHP). Для того чтобы получить доступ к этим переменным, необходимо, как минимум, знать способы обращения к массивам.

Ассоциативные массивы и массивы с числовыми индексами

В массивах с числовыми индексами в качестве значений индексов используются числа, а в ассоциативных массивах – строки. В ассоциативных массивах каждому новому элементу нужно назначить уникальный строковый индекс. Массивы с числовыми индексами позволяют просто добавить элемент, а PHP автоматически назначит ему в качестве индекса первое свободное число начиная с 0. Массивы обоих типов позволяют добавлять новые элементы по одному. Ассоциативные массивы прекрасно подходят для сохранения информации о настройках, так как их ключи могут хранить смысловую информацию.

Будьте внимательны: большинство людей начинают счет не с 0, а с 1. По рассеянности вы легко можете обратиться к несуществующему элементу массива – это называется *ошибкой завышения на единицу* (off-by-one error). Чтобы получить значение индекса последнего элемента в массиве, нужно вычесть из длины массива единицу. Типичный симптом того, что по ошибке вы начали обходить массив с индекса 1, а не 0 – обнаружение того, что при попытке обратиться к последнему элементу массива такого элемента просто не находится.

Внутри PHP массивы с числовыми индексами хранятся точно так же, как и ассоциативные массивы. Массивы с числовыми индексами обеспечивают более простой способ обхода наборов данных, поскольку для доступа к следующему значению достаточно увеличить на единицу индекс предыдущего.

Создание массива

Для создания массивов в сценарии PHP предусмотрены три основных способа: присваивание значения одному из элементов будущего массива (и тем самым неявное создание массива), использование конструкции array() и вызов функции, особенностью которой является то, что она возвращает массив в качестве своего значения.

Непосредственное присваивание

Простейший способ создания массива состоит в выполнении с некоторой переменной таких действий, как будто эта переменная уже представляет собой массив, и присваивании ей значения:

Чтобы создать массив, нужно определить значения его элементов и индексов. В качестве элементов массива могут использоваться любые значения, включая строки, числа и даже другие массивы. Поле ключа должно быть скаляром. Скалярные значения – это такие значения элементарного типа, как числа или строки, включая значения TRUE и FALSE, но не данные, которые могут иметь несколько составных значений, например объекты. Кроме того, в поле ключа массива для каждого элемента должно быть уникальное значение, иначе вы можете записать новый элемент поверх уже имеющегося, с тем же ключом. Если вы попытаетесь назначить новому элементу ключ, который уже определен для другого элемента, новое значение просто заменит старое.

Конструкция array()

Еще один способ создания массива состоит в использовании конструкции array(), которая создает новый массив на основании спецификации элементов и ассоциированных с ними ключей. В своей простейшей версии конструкция array() вызывается без параметров, что приводит к созданию нового пустого массива. Более сложной по сравнению с простейшей является версия, в которой конструкция array() принимает разделенный запятыми список элементов, подлежащих сохранению, без какого-либо указания ключей.

В данном случае интерпретатор PHP действует на основании предположения, что добавляются последовательные элементы, которые должны иметь целочисленные индексы, отсчитываемые от нуля в сторону увеличения.

Следует еще раз подчеркнуть, что по умолчанию нумерация индексов массива начинается с нуля, а не с единицы. Такое соглашение о нумерации индексов массивов принято в большинстве языков программирования.

В простом примере конструкции array(), приведенном в предыдущем примере, присваивание индексов элементам осуществлялось автоматически, но такие индексы являются исключительно целочисленными и отсчитываются в сторону увеличения от нуля, поэтому указанный подход не предоставляет широкого выбора.

Выполнение этого оператора приведет к получению такого же результата, как и выполнение приведенных выше вариантов операций присваивания — каждая строка будет сохранена в массиве последовательно и этим строковым значениям будут по порядку присвоены индексы 0, 1, 2, 3.

В результате этого в массив будут добавлены те же четыре элемента в том же порядке, но с выполнением индексации с помощью ключей, соответствующих названиям цветов, а не чисел. Например, чтобы извлечь из массива название фрукта желтого цвета, достаточно вычислить такое выражение:

Код PHP// Этот элемент должен иметь значение 'Банан'$fruit['yellow'];

Функции, возвращающие массивы

Последний способ создания массива в сценарии состоит в вызове функции, которая возвращает массив. Это может быть функция, определяемая пользователем, или встроенная функция, создающая массив с помощью каких-то внутренних методов PHP.

Например, в виде динамически создаваемых массивов возвращают свои результаты многие функции, обеспечивающие взаимодействие с базой данных. Предусмотрены и другие функции, предназначенные исключительно для создания массивов, удобных для использования в качестве заготовок в применяемых впоследствии функциях манипулирования массивами. Одной из таких функций является range(), которая принимает два целых числа в качестве параметров и возвращает массив, заполненный всеми целыми числами (включительно), лежащими на числовой оси между этими параметрами.

Выборка значений

Настоящий раздел позволяет найти ответ на вопрос о том, как осуществляется выборка значений из массива после их сохранения в этом массиве.

Предусмотрено множество других способов выборки значений из массивов, в которых не используются ключи. Большинство из этих способов основано на том факте, что в массивах по умолчанию предусматривается регистрация элементов в том порядке, в каком происходила передача этих элементов на хранение. Конструкция list() применяется для присваивания нескольких значений подряд идущих элементов массива переменным. Допустим, что были выполнены два оператора:

Переменным в конструкции list() значения элементов массива присваиваются в том порядке, в каком эти элементы были первоначально сохранены в массиве. Обратите внимание на то, насколько необычной является синтаксическая структура используемой операции присваивания — конструкция list() находится слева от знака операции присваивания (=), тогда как в этой позиции обычно можно встретить только имена переменных.

В определенном смысле конструкция list() является противоположной, или обратной, конструкции array(), поскольку array() упаковывает свои параметры в массив, а конструкция list() распаковывает массив, присваивая значения отдельным переменным.

Многомерные массивы

Во всех примерах, рассматриваемых до сих пор, массивы были одномерными, поскольку ключи, заданные в фигурных скобках, определяли только один уровень доступа к массиву. Но язык PHP позволяет легко обеспечить, поддержку многомерных массивов с произвольным количеством ключей. Многомерные массивы — это просто массивы, в которых хранятся другие массивы. Ниже показан пример создания числовых и ассоциативных многомерных массивов, а ниже блок-схема, поясняющая их использование:

Следует отметить, что попытка обратиться с помощью неправильного индекса к многомерному массиву для выборки какого-либо значения не влечет за собой сурового наказания; если соответствующий ключ не будет найден, то все выражение рассматривается как несвязанная переменная.

Получение сведений о массивах

Выше было описано, как создавать массивы, сохранять значения в массивах, а затем снова извлекать значения, когда они потребуются.

Следует отметить, что все эти функции действуют только на глубину одного уровня заданного массива, поэтому для получения данных о значениях, хранящихся в многомерном массиве на более глубоких уровнях, необходимо указать, на каком уровне должна быть получена информация.

Удаление элементов из массивов

Операция удаления элемента из массива является простой и полностью аналогичной операции удаления значения, присвоенного переменной.

Итерация массивов

Массивы PHP не только сохраняют значения, ассоциированные с соответствующими ключами, но и по умолчанию образуют упорядоченный список пар "ключ-значение", хранящийся в том порядке, в котором происходила запись этих пар в массив. Такая структура массива была принята исходя из тех соображений, чтобы массив можно было использовать для обеспечения итерации по всему содержимому массива. (Следует отметить, что эту задачу трудно выполнить, просто сформировав цикл, который наращивает значение индекса, поскольку индексы массивов PHP не обязательно должны быть числовыми.)

В действительности в структуру массивов встроена своего рода скрытая система указателей. Каждая хранимая пара "ключ-значение" указывает на следующую, и одним из побочных эффектов этого становится то, что текущий указатель указывает на самый первый элемент и остается направленным именно на этот элемент, если не будет сдвинут в результате выполнения одной из итеративных функций.

В каждом массиве некоторая определенная хранимая пара "ключ-значение" запоминается как текущая, и определенная часть действий итеративных функций массива основана на принципе передвижения этого маркера текущей пары по внутреннему списку ключей и значений. Мы будем называть этот маркер *текущим указателем*, но следует учитывать, что язык

Рассматриваемая система указателей на основе связного списка представляет собой альтернативный способ получения информации о массивах и манипулирования ими. Система указателей существует наряду с системой, обеспечивающей запись в массив и выборку из массива на основе ключей.

Цикл foreach

Для организации циклической обработки массива можно использовать цикл foreach. Безусловно, создается впечатление, что эта конструкция унаследована от конструкции foreach языка Perl, но имеет немного странную синтаксическую структуру (которая, тем не менее, не совпадает со странной синтаксической структурой аналогичной конструкции из языка Perl). Синтаксическая структура конструкции foreach в языке PHP имеет две разновидности, и решение о том, какая из них должна использоваться в программе, зависит от того, требуется ли выборка и ключей, и значений массива или только значений:

Функция current() возвращает сохраненное значение, на которое в настоящее время направлен указатель. Во вновь созданном массиве с элементами текущий указатель всегда направлен на первый элемент. Функция next() вначале продвигает указатель, а затем возвращает текущее значение, на которое направлен указатель. Если функция next() вызывается в тех условиях, когда текущий указатель уже направлен на последнее хранимое значение и поэтому должен выйти за конец массива, функция возвращает ложное значение.

Средства преобразования массивов

В языке PHP предусмотрен целый ряд функций, позволяющих манипулировать данными после того, как эти данные для обеспечения удобства в работе будут сохранены в виде массивов.

Давайте на примерах рассмотрим эти функции более подробно.

Выборка ключей и значений

Функция array\_keys() возвращает ключи массива, заданного в качестве входного параметра, в форме нового массива, ключами которого являются хранимые значения. Ключами нового массива становятся обычные индексы в виде автоматически наращиваемых целых чисел, которые начинаются с нуля. Функция array\_values() выполняет точно такое же действие, если не считать того, что хранимыми значениями становятся значения из первоначального массива

По-видимому, результаты применения второй из рассматриваемых функций (array\_values()) представляют меньший интерес, поскольку эта функция фактически принимает исходный массив и формирует новый массив, ключи которого заменены последовательно возрастающими числами.

Зеркальное отображение, обращение и случайное перемешивание

Более редко применяемой функцией по сравнению с описанными выше является функция array\_flip(), которая преобразует ключи массива в значения, и наоборот. Следует отметить, что гарантируется уникальность ключей массива, а на значения массива такая гарантия не распространяется.

Операция обращения массива является менее сложной: функция array\_reverse() возвращает новый массив, в котором пары "ключ-значение" находятся в обратном порядке.

Если в программу должны быть внесены некоторые дополнительные элементы случайности, то для этого можно воспользоваться функцией shuffle(). Функция shuffle() принимает в качестве параметра массив и изменяет случайным образом порядок элементов в массиве.

Использование функций array\_flip(), array\_reverse(), shuffle()

В отличие от многих других функций работы с массивами, приведенных в данной статье, функция shuffle() является *деструктивной*. Это означает, что она применяется непосредственно к массиву, заданному в качестве параметра, и изменяет его, а не возвращает вновь созданный массив. (Функции, возвращающие новый объект и не изменяющие свои параметры, принято называть конструктивными, или недеструктивными.) Кроме всего прочего, это означает, что правильный способ вызова рассматриваемой функции случайного перемешивания элементов не является таковым:

Код PHP$array = shuffle($array); // Неправильная конструкция!

Слияние, дополнение, вырезка и вставка элементов массивов

Если требуется объединить два массива, например для получения более полного списка, то можно воспользоваться функцией array\_merge(). Эта функция принимает в качестве параметров два или несколько массивов и возвращает новый массив с перенумерованными ключами, в котором второй массив присоединен к концу первого.

Функция array\_pad() используется для создания определенного количества ведущих или заключительных пар "ключ-значение" в целях увеличения размера массива. Эта функция принимает в качестве первого параметра входной массив. Вслед за этим параметром должно быть указано количество элементов, на которое увеличивается массив, а затем — значение, присваиваемое дополнительным элементам. Если в качестве второго параметра задано положительное целое число, то массив дополняется в конце, а в случае указания отрицательного целого числа массив дополняется в начале. Если второй параметр меньше размера массива, дополнение не выполняется:

Немного более сложными являются функции array\_slice() и array\_splice(). Первая из них возвращает подмножество элементов входного массива, принимая в качестве второго и третьего параметров смещение и длину соответственно. Функция array\_splice() аналогична функции array\_slice(), но принимает четвертый параметр, который может представлять собой массив любой длины, предназначенный для вставки во входной массив.

Стеки и очереди

Стеки и очереди представляют собой абстрактные структуры данных, часто используемые в программировании, которые налагают требования по использованию определенных правил доступа к содержащимся в них объектам, причем эти правила практически не зависят от типа самих объектов. Массивы языка PHP хорошо подходят для решения задач эмуляции структур данных других типов, к тому же благодаря отсутствию строгой типизации элементов массива PHP появляется возможность легко эмулировать с помощью массивов такие структуры данных, как стеки и очереди. В языке PHP предусмотрены некоторые функции работы с массивами, специально предназначенные для этой цели. Если в программе используются исключительно такие функции, то программист может даже не учитывать, что в основе рассматриваемых структур данных лежат массивы.

*Стек* представляет собой контейнер, который сохраняет значения и поддерживает операции доступа к этим значениям по принципу обратной очереди (last-in-first-out — LIFO). Это означает, что стек поддерживает порядок сохраняемых в нем значений и предоставляет единственный способ выборки сохраненного в нем значения — извлечение (и удаление) последнего по времени сохраненного значения:

Стек - "последним вошел, первым вышел"

Обычно для описания стека применяется такая аналогия — стопка подносов в кафетерии перед одной из линий раздачи, из которой посетители берут подносы, снимая их сверху, а служащие постоянно пополняют запас подносов, также накладывая их сверху. Новые подносы можно помещать поверх уже имеющихся и брать имеющиеся сверху, но до помещенного ранее подноса невозможно добраться, не сняв перед этим помещенные позже подносы.

Очередь аналогична стеку, но действует по принципу последовательной очереди (first-in-first-out — FIFO). Обычно для описания такой очереди применяется аналогия с цепочкой людей, ожидающих обслуживания, которую в Англии обозначают queue, а в США — line. Общее правило обслуживания людей, стоящих в очереди, состоит в том, что следующим должен обслуживаться тот, кто простоял в очереди дольше всех:

Очередь - "первым вошел, первым вышел"

Для работы со стеками применяются функции array\_push() и array\_pop(). Функция array\_push() принимает в качестве параметра исходный массив, за которым может быть указано любое количество элементов, заталкиваемых в стек. Элементы вставляются в конце массива, в последовательности слева направо. Функция array\_pop() принимает в качестве параметра такой массив и удаляет элемент, находящийся в конце массива, возвращая этот элемент.

Сортировка массивов

Наконец, в языке PHP предусмотрено множество функций для сортировки массивов. Как было указано выше, иногда бывает сложно учесть различие между ситуацией, в которой применяются ассоциации "ключ-значение" в массиве, и ситуацией, в которой числовые ключи рассматриваются в качестве информации об упорядочении, подлежащей изменению после изменения порядка элементов. К счастью, в языке PHP предусмотрены варианты функций сортировки, в которых учитываются особенности ситуаций каждого из этих типов. Кроме того, язык PHP позволяет сортировать элементы по возрастанию или по убыванию, а также использовать функции упорядочения, предоставленные пользователем. Имена функций сортировки являются сокращенными, но содержат специальные буквенные обозначения (кроме части имени со словом sort), которые позволяют определить назначение функции. Ниже приведено краткое описание указанных буквенных обозначений:

Начальная буква "a" означает, что функция выполняет сортировку по значениям, но сохраняет ассоциации между парами "ключ-значение" в том виде, в котором они существуют.

Начальная буква "k" означает, что сортировка осуществляется по ключам, но ассоциации "ключ-значение" сохраняются.

Отсутствие начальной буквы "a" или "k" означает, что сортировка выполняется по значениям, но ассоциации "ключ-значение" не сохраняются. В частности, числовые ключи перенумеровываются, отражая новое упорядочение.

Буква "r" перед словом sort означает, что применяется обратный порядок сортировки.

Начальная буква "u" означает, что функция принимает второй параметр — имя определяемой пользователем функции, которая задает упорядочение любых двух элементов, подлежащих сортировке.

Как видите функция sort() не только отсортировала значения, но и заменила текстовые ключи на числовые индексы. Если такое поведение нежелательно, то следует использовать функцию asort().

Строки. Основные операции над строками (конкатенация, сравнение, замена). Базовые функции работы со строками: strlen(), strpos(), substr(), strcmp(), strcasecmp().

Строка (тип string) - это набор символов, где символ - это то же самое, что и байт. Это значит, что PHP поддерживает ровно 256 различных символов, а также то, что в PHP нет встроенной поддержки Unicode. Смотрите также подробности реализации строкового типа.

Одинарные кавычки ¶

Простейший способ определить строку - это заключить её в одинарные кавычки (символ ').

Замечание: В отличие от синтаксиса двойных кавычек и heredoc, переменные и управляющие последовательности для специальных символов, заключённых в одинарные кавычки, *не* обрабатываются.

Двойные кавычки ¶

Если строка заключена в двойные кавычки ("), PHP распознает следующие управляющие последовательности специальных символов:

Как и в строке, заключённой в одинарные кавычки, экранирование любого другого символа выведет также и сам символ экранирования.

Но самым важным свойством строк в двойных кавычках является обработка переменных. Смотрите более подробно обработку строк.

Heredoc ¶

Третий способ определения строк - это использование heredoc-синтаксиса: <<<. После этого оператора необходимо указать идентификатор, затем перевод строки. После этого идёт сама строка, а потом этот же идентификатор, закрывающий вставку.

Закрывающий идентификатор может иметь отступ с помощью пробела или табуляции, и в этом случае отступ будет удалён из всех строк в строке документа. До PHP 7.3.0 закрывающий идентификатор *должен* был находиться в самом начале новой строки.

Кроме того, закрывающий идентификатор должен соответствовать тем же правилам именования, что и любая другая метка в PHP: он должен содержать только буквенно-цифровые символы и подчёркивания и должен начинаться с нецифрового символа или символа подчёркивания.

Если закрывающий идентификатор смещён дальше, чем любая строка тела, будет выброшено ParseError

Если у закрывающего идентификатора есть отступ, табуляции также могут использоваться, однако табуляции и пробелы *не должны* смешиваться в отношении отступа закрывающего идентификатора и отступа тела (вплоть до закрывающего идентификатора). В любом из этих случаев будет выброшено ParseError. Эти ограничения на пробелы были включены, потому что смешивание табуляции и пробелов для отступов вредно для разбора.

За закрывающим идентификатором основной строки не обязательно ставить точку с запятой или новую строку.

Внимание

Если закрывающий идентификатор был найден в начале строки, то независимо от того, был ли он частью другого слова, его можно рассматривать как закрывающий идентификатор и выбросить ParseError.

Если это правило нарушено и закрывающий идентификатор не является "чистым", считается, что закрывающий идентификатор отсутствует и PHP продолжит его поиск дальше. Если в этом случае верный закрывающий идентификатор так и не будет найден, то это вызовет ошибку парсинга с номером строки в конце скрипта.

Heredoc, содержащий переменные, не может использоваться для инициализации свойств класса.

Heredoc-текст ведёт себя так же, как и строка в двойных кавычках, при этом их не имея. Это означает, что вам нет необходимости экранировать кавычки в heredoc, но вы по-прежнему можете использовать вышеперечисленные управляющие последовательности. Переменные обрабатываются, но с применением сложных переменных внутри heredoc нужно быть также внимательным, как и при работе со строками.

Также возможно использовать heredoc-синтаксис для передачи данных через аргументы функции:

Можно инициализировать статические переменные и свойства/константы класса с помощью синтаксиса heredoc:

Можно также окружать идентификатор Heredoc двойными кавычками:

Если строка указывается в двойных кавычках, либо при помощи heredoc, переменные внутри неё обрабатываются.

Существует два типа синтаксиса: простой и сложный. Простой синтаксис более лёгок и удобен. Он даёт возможность обработки переменной, значения массива (array) или свойства объекта (object) с минимумом усилий.

Сложный синтаксис может быть определён по фигурным скобкам, окружающим выражение.

Простой синтаксис

Если интерпретатор встречает знак доллара ($), он захватывает так много символов, сколько возможно, чтобы сформировать правильное имя переменной. Если вы хотите точно определить конец имени, заключайте имя переменной в фигурные скобки.

Аналогично могут быть обработаны элемент массива (array) или свойство объекта (object). В индексах массива закрывающая квадратная скобка (]) обозначает конец определения индекса. Для свойств объекта применяются те же правила, что и для простых переменных.

В PHP 7.1.0 добавлена поддержка *отрицательных* числовых индексов.

Доступ к символу в строке и его изменение ¶

Символы в строках можно использовать и модифицировать, определив их смещение относительно начала строки, начиная с нуля, в квадратных скобках после строки, например, *$str[42]*. Думайте о строке для этой цели, как о массиве символов. Если нужно получить или заменить более 1 символа, можно использовать функции substr() и substr\_replace().

Полезные функции и операторы ¶

Строки могут быть объединены при помощи оператора '.' (точка). Обратите внимание, оператор сложения '+' *здесь не работает*. Дополнительную информацию смотрите в разделе Строковые операторы.

Для модификации строк существует множество полезных функций.

Основные функции описаны в разделе строковых функций, а для расширенного поиска и замены - функции Perl-совместимых регулярных выражений.

Также существуют функции для работы с URL, и функции шифрования/дешифрования строк (Sodium и Hash).

Наконец, смотрите также функции символьных типов.

Преобразование в строку ¶

Значение может быть преобразовано в строку с помощью приведения (string), либо функции strval(). В выражениях, где необходима строка, преобразование происходит автоматически. Это происходит, когда вы используете функции echo или print, либо когда значение переменной сравнивается со строкой. Прочтение разделов руководства Типы и Манипуляции с типами сделает следующее более понятным. Смотрите также

Значение null всегда преобразуется в пустую строку.

Как вы могли видеть выше, прямое преобразование в строку массивов, объектов или ресурсов не даёт никакой полезной информации о самих значениях, кроме их типов. Более подходящий способ вывода значений для отладки - использовать функции print\_r() и var\_dump().

Большинство значений в PHP может быть преобразовано в строку для постоянного хранения. Этот метод называется сериализацией и может быть выполнен при помощи функции serialize().

Подробности реализации строкового типа ¶

Строковый тип (string) в PHP реализован в виде массива байт и целого числа, содержащего длину буфера. Он не содержит никакой информации о способе преобразования этих байт в символы, предоставляя эту задачу программисту. Нет никаких ограничений на содержимое строки, например, байт со значением 0 ("NUL"-байт) может располагаться где угодно (однако, стоит учитывать, что некоторые функции, как сказано в этом руководстве, не являются "бинарно-безопасными", т.е. они могут передавать строки библиотекам, которые игнорируют данные после NUL-байта).

Данная природа строкового типа объясняет почему в PHP нет отдельного типа "byte" - строки играют эту роль. Функции, возвращающие нетекстовые данные - например, произвольный поток данных, считываемый из сетевого сокета - тем не менее возвращают строки.

Принимая во внимание тот факт, что PHP не диктует определённую кодировку для строк, можно задать вопрос, как в таком случае кодируются строковые литералы. Например, строка "á" эквивалентна "\xE1" (ISO-8859-1), "\xC3\xA1" (UTF-8, форма нормализации C), "\x61\xCC\x81" (UTF-8, форма нормализации D) или какому-либо другому возможному представлению? Ответом является следующее: строка будет закодирована тем образом, которым она записана в файле скрипта. Таким образом, если скрипт записан в кодировке ISO-8859-1, то и строка будет закодирована в ISO-8859-1 и т.д. Однако, это правило не применяется при включённом режиме Zend Multibyte: в этом случае скрипт может быть записан в любой кодировке (которая указывается ясно или определяется автоматически), а затем конвертируются в определённую внутреннюю кодировку, которая и будет впоследствии использована для строковых литералов.

Разумеется, чтобы приносить пользу, строковые функции должны сделать некоторые предположения о кодировке строки. К несчастью, среди PHP-функций довольно большое разнообразие подходов к этому вопросу:

Некоторые функции предполагают, что строка закодирована в какой-либо однобайтовой кодировке, однако, для корректной работы им не требуется интерпретировать байты как определённые символы. Под эту категорию попадают, например, substr(), strpos(), strlen() и strcmp(). Другой способ мышления об этих функциях представляет собой оперирование буферами памяти, т.е. они работают непосредственно с байтами и их смещениями.

Другие функции ожидают передачу кодировку в виде параметра, возможно, предполагая некоторую кодировку по умолчанию, если параметр с кодировкой не был указан. Такой функцией является htmlentities() и большинство функций из модуля mbstring.

Другие функции используют текущие установки локали (смотрите setlocale()), но оперируют побайтово.

Наконец, есть функции, подразумевающие, что строка использует определённую кодировку, обычно UTF-8. Сюда попадают большинство функций из модулей intl и PCRE (в последнем случае, только при указании модификатора u).

В конечном счёте, написание корректных программ, работающих с Unicode, означает осторожное избегание функций, которые не работают с Unicode и, скорее всего, испортят данные, и использование вместо них корректных функций, обычно из модулей intl и mbstring. Однако, использование функций, способных работать с Unicode, является самым началом. Вне зависимости от тех функций, которые предоставляет язык, необходимо знать спецификацию самого Unicode. Например, если программа предполагает существование в языке только строчных и заглавных букв, то она делает большую ошибку.

Операции над строками.

Строки в PHP.

Свойства строки в PHP:

Строка может содержать любые символы.

Строка может содержать произвольное количество символов. Длина строки ограничивается только объемом оперативной памяти, что позволяет читать в строку целые файлы и работать с ними, как с обыкновенными строками, например сравнивать.

Конкатенация строк.

Для конкатенации (присоединения строк) используется оператор "." (точка).

Сравнение строк.

В PHP операнды сравниваются как строки, только в том случае, если они - строки. В противном случае они сравниваются, как числа. При этом любая строка, которую интерпретатору не удается перевести в число, будет восприниматься, как 0.

Лучше всего для сравнения использовать = = = (тройной оператор равенства - оператор эквивалентности). Он всегда позволяет производить корректное сравнение, т.к. он сравнивает величины и по значению, и по типу.

Базовые функции работы со строками.

Функция strlen() возвращает длину строки, то есть количество символов, которое содержит строка.

Функция strpos() ищет подстроку в строке. Функции нужно передать три параметра:

строку, в которой ищем подстроку.

подстроку.

позицию, начиная с которой происходит поиск. Данный параметр необязательный и по умолчанию равен 0.

Если строка не найдена, функция возвращает false, в противном случае - номер позиции, с которой начинается вхождение подстроки в строку:

Функция substr (string $str, int $from, int $lenght)возвращает заданный участок строки, то есть подстроку. Возвращается подстрока строки $str, начиная с позиции $from длины $length. Если параметр $from будет отрицательным, то отсчет подстроки будет производиться с конца строки, а не с начала. Параметр $length является необязательным.

Функция strasecmp (sstring $str1, string $str2)  сравнивает две строки без учета регистра символов, то есть, строки BOOK, BOOK и book для этой функции равны.

Замена в строке.

Функция str\_replace ($str1, $str2, $source) заменяет в строке $source (которая в процессе работы функции не изменяется) все вхождения подстроки $str1 на подстроку $str2 и возвращает результат в виде новой строки.

Пример замены символов новой строки \n на тэг <br>:

$str = str\_replace ("\n", "<br>", $str);

Функция WordWrap() оказывается очень полезной, когда нужно отформатировать текст письма перед его отправкой.

При изменении регистра русских букв могут возникнуть проблемы, избежать которых поможет правильная настройка локали. Локалью называется совокупность настроек системы, таких как формат даты и времени, язык, кодировка. Для установки локали используется функция SetLocale(), которой следует передать два параметра - категорию устанавливаемых параметров и локаль.

URL - кодирование и декодирование.

Функция UrlEncode() используется для кодирования данных. Кодирование необходимо для передачи данных через Интернет, содержащих специальные символы национальных алфавитов, например русского языка. При передаче целого файла его необходимо закодировать с помощью функции UrlEncode().

В следующем листинге приведен пример, демонстрирующий совместную работу функций UrlEncode() и UrlDecode():

Использование слэшей.

Для вывода кавычек и наклонной черты необходимо использовать цитирование, то есть, запись этих символов через наклонную черту

Для добавления слэшей используется функция AddSlashes (string $str), а для удаления - StripSlashes (atring $str).

Обе функции возвращают модифицированную строку и не изменяют исходную.

Замена специальных символов на их HTML - эквиваленты.

Функция HtmlSpecialChars (string $str) заменяет специальные символы (кавычки, "больше", "меньше" и другие) на их HTML - эквиваленты, для того, чтобы на Web - странице они выглядели сами собой. Например при использовании функции HtmlSpecialChars, знак "больше" в коде будет заменен строкой &gt, а кавычки - &quot.

Преобразование кодировок.

В PHP применяется функция convert\_cyr\_string(), преобразующая русский текст из одной кодировки в другую. Данной функции необходимо передать три параметра:

Исходную строку.

Исходную кодировку.

Требуемую кодировку.

Функция md5() кодирует строку, используя алгоритм MD5 (Message Digest Algorithm). Вероятность того, что две разные строки будут иметь одинаковый хэш - код, стремится к нулю, а это позволяет эффективно использовать данную функцию для шифрования паролей. Зашифрованное с его помощью сообщение невозможно расшифровать - для алгоритма MD5 не существует алгоритма дешифровки. Для проверки подлинности пароля нужно зашифровать пароль заново и сравнить зашифрованные строки: если они равны, значит пароль правильный. Например:

Сценарий выведет только первоев сообщение, поскольку строки (значит, и их хэш - коды) равны. Рекомендация по использованию этой функции:

Пароль не должен храниться в сценарии, тем более в открытом виде. пароль нужно хранить в зашифрованном виде в базе данных или в каком-нибудь файле на диске.

Введенный пользователем пароль должен зашифровываться и сравниваться уже с зашифрованной строкой, которая прочитана из файла или базы данных.

Кроме функции md5() можно использовать функцию crypt(), которая реализует алгоритм DES, но данный алгоритм менее эффективен, поэтому для серьезных проектов не применяется.

Функция crc32 () используется для вычисления 32 - битной контрольной суммы строки. Данная функция пригодна для регистрации программного продукта через Internet, является стандартной, и менее рекомендуема для серьезных проектов.

Для работы со строками в PHP существует множество полезных функций.

strlen(string $st)

Одна из наиболее полезных функций. Возвращает просто длину строки, т. е., сколько символов содержится в $st. Строка может содержать любые символы, в том числе и с нулевым кодом (что запрещено в Си).

strpos(string $where, string $what, int $fromwhere=0)

Пытается найти в строке $where подстроку (то есть последовательность символов) $what и в случае успеха возвращает позицию (индекс) этой подстроки в строке. Необязательный параметр $fromwhere можно задавать, если поиск нужно вести не с начала строки $from, а с какой-то другой позиции.

substr(string $str, int $start [,int $length])

Данная функция тоже востребуется очень часто. Ее назначение — возвращать участок строки $str, начиная с позиции $start и длиной $length. Если $length не задана, то подразумевается подстрока от $start до конца строки $str. Если $start больше, чем длина строки, или же значение $length равно нулю, то возвращается пустая подстрока. Однако эта функция может делать и еще довольно полезные вещи. К примеру, если мы передадим в $start отрицательное число, то будет считаться, что это число является индексом подстроки, но только отсчитываемым от конца $str (например, -1 означает "начиная с последнего символа строки"). Параметр $length, если он задан, тоже может быть отрицательным.

Сравнивает две строки посимвольно (точнее, побайтово) и возвращает: 0, если строки полностью совпадают; -1, если строка $str1 лексикографически меньше $str2; и 1, если, наоборот, $str1 "больше" $str2. Так как сравнение идет побайтово, то регистр символов влияет на результаты сравнений.

strcasecmp(string $str1, string $str2)

То же самое, что и strcmp(), только при работе не учитывается регистр букв. Например, с точки зрения этой функции "ab" и "AB" равны.

Функции. Общий синтаксис написания функции. Виды функций: вложенные функции, условно определяемые функции.

Приведём пример синтаксиса, используемого для описания функций:

Внутри функции можно использовать любой корректный PHP-код, в том числе другие функции и даже объявления классов.

Имена функций следуют тем же правилам, что и другие метки в PHP. Корректное имя функции начинается с буквы или знака подчёркивания, за которым следует любое количество букв, цифр или знаков подчёркивания. Подсказка

Смотрите также Руководство по именованию.

Функции не обязаны быть определены до их использования, *исключая* тот случай, когда функции определяются условно, как это показано в двух последующих примерах.

В случае, когда функция определяется в зависимости от какого-либо условия, например, как это показано в двух приведённых ниже примерах, обработка описания функции должна *предшествовать* её вызову.

Все функции и классы PHP имеют глобальную область видимости - они могут быть вызваны вне функции, даже если были определены внутри и наоборот.

Можно вызывать функции PHP рекурсивно.

Замечание: Рекурсивный вызов методов/процедур с глубиной более 100-200 уровней рекурсии может вызвать переполнение стека и привести к аварийному завершению скрипта. В частности, бесконечная рекурсия будет считаться программной ошибкой.

В этой главе представлены общие концепции функционального программирования -- одного из самых влиятельных принципов в области разработки приложений. При помощи функций можно создавать компоненты многократного использования, которые легко изменяются при необходимости и оказываются особенно удобными при разработке web-приложений, не содержащих серьезных различий на концептуальном и практическом уровне. Функциональное программирование помогает создавать более короткие и удобочитаемые программы.

Что такое функция?

*Функцией*  называется фрагмент программного кода, обладающий уникальным именем и предназначенный для решения конкретной задачи. Функция вызывается по имени в разных точках программы, что позволяет многократно выполнять фрагмент с указанным именем. Преимущество такого решения заключается в том, что блок кода пишется всего один раз, а затем легко модифицируется по мере необходимости.

Определение и вызов функций

Определить новую функцию в PHP несложно. Функции могут создаваться в любой точке программ PHP, однако по соображениям структурной организации кода удобнее разместить все функции, используемые сценарием, в самом начале сценарного

файла. Существует и другой способ, заметно повышающий эффективность программирования и способствующий многократному использованию кода, -- выделение функций в отдельный файл (называемый  *библиотекой).*  Библиотеки удобны тем, что их функции можно использовать в разных приложениях, не создавая лишних копий и не рискуя допустить ошибки в процессе копирования.

Имя функции должно подчиняться условиям, приведенным для идентификаторов в главе 2. После имени функции следуют обязательные круглые скобки, в которые заключается необязательный список входных параметров ($параметр1, $параметр2, .... $параметрn). Вследствие относительно либеральных принципов определения переменных в PHP указывать тип входных параметров не нужно. Хотя такой подход имеет свои преимущества, следует помнить, что механизм PHP не проверяет аргументы на соответствие тем типам, которые должны обрабатываться функцией. Случайные ошибки в использовании входных параметров могут привести к неожиданным последствиям (чтобы убедиться в том, что параметр относится к нужному типу, можно проверить его стандартной функцией gettype( )). После закрывающей круглой скобки следуют фигурные скобки, в которые заключается программный код, ассоциируемый с именем функции.

Если ваш web-сайт состоит из нескольких страниц, достаточно вызвать эту функцию в конце каждой страницы -- и вам не придется заново переписывать один и тот же текст. А когда наступит 2002 год, одно простое изменение текста, выводимого этой функцией, приведет к автоматическому обновлению всех страниц. Если бы не преимущества функционального программирования, вам пришлось бы вручную редактировать все страницы, на которых выводится лицензионная информация.

Рассмотрим разновидность функции display\_copyright(), которой при вызове передается параметр. Предположим, вы отвечаете за администрирование нескольких web-сайтов, каждому из которых присвоено отдельное имя. На каждом сайте имеется собственный административный сценарий с несколькими переменными, относящимися к этому сайту; к их числу принадлежит переменная $site\_name с именем сайта. В этом случае функцию display\_copyright() можно записать следующим образом:

Переменная $site\_name, значение которой присваивается за пределами display\_copy-right(), передается функции в качестве параметра. Переданное значение можно использовать и модифицировать в любом месте функции, однако любые изменения будут действовать лишь внутри этой функции. Впрочем, специальные ключевые слова позволяют сделать так, чтобы изменения параметров распространялись и за пределы display\_copyright(). Эти ключевые слова были представлены в главе 2, в общем обзоре области видимости переменных и ее отношения к функциям.

Вложенные функции

Функции можно вызывать внутри других функций -- по аналогии с тем, как одна управляющая конструкция (if, while, for и т. д.) может находиться внутри другой. Такая возможность удобна в любых программах, и в больших, и в малых, поскольку она увеличивает степень модульности приложения и упрощает сопровождение программы.

Параметр Y функции date( ) указывает, что возвращаемое значение представляет собой текущий год, отформатированный в виде четырех цифр. Если системная дата установлена правильно, PHP при каждом выполнении сценария будет выводить год. Функция PHP date( ) отличается исключительной гибкостью и поддерживает 25 разных флагов форматирования даты и времени.

Также допускается объявление функций внутри других функций. Тем не менее, вложенное объявление еще не делает функцию «защищенной», то есть не ограничивает возможность ее вызова той функцией, в которой она была объявлена. Более того, вложенная функция не наследует параметров родительской функции; параметры должны передаваться ей точно так же, как и любой другой функции. Впрочем, вложенные объявления функций все равно могут использоваться из соображений удобства сопровождения и наглядности. Пример вложенного объявления приведен в листинге 4.1.

Хотя вложенные функции не защищены от вызова из других точек сценария, они не могут вызываться  *до*  вызова своей родительской функции. При попытке вызвать вложенную функцию раньше вызова родительской функции выводится сообщение об ошибке.

Возврат значений функцией

По завершении работы функции часто бывает полезно вернуть некоторое значение, для чего результат вызова функции обычно присваивается некоторой переменной. Функции могут возвращать значения любых типов, в том числе массивы и списки.

Вывести соответствующее сообщение.

Существует и другой способ использования возвращаемых значений, при котором вызов функции включается прямо в условную/циклическую команду. В следующей программе (листинг 4.3) сумма счета пользователя сравнивается с предельным размером кредита. Алгоритм на псевдокоде выглядит так:

Если накопленная сумма счета превышает предельный размер кредита, функция возвращает ложное значение (0).

Если условие команды i f оказывается ложным, работа функции еще не завершена. В этом случае общая сумма не превышает предельного размера кредита, поэтому функция должна вернуть логическую истину.

Вызвать функцию check\_limit( ) в условии команды if. Проверить, какое значение было возвращено при вызове -- истинное или ложное. В зависимости от результата проверки выполняется то или иное действие.

Если при вызове check\_limit( ) было получено значение TRUE, мы предлагаем пользователю продолжить закупку. В противном случае пользователь информируется о превышении кредита.

Функция также может возвращать сразу несколько значений при помощи списка.

Рекурсивные функции

Ситуация, при которой функция многократно вызывает сама себя, пока не будет выполнено некоторое условие, открывает замечательные возможности. При правильном использовании  *рекурсивные*  функции уменьшают объем программы и делают ее более выразительной. Рекурсивные функции особенно часто используются при выполнении повторяющихся действий -- например, при поиске в файлах/массивах и построении графических изображений (например, фракталов). Если функция вызывается достаточно часто, рекурсия делает программу более эффективной. Тем не менее, при использовании рекурсии необходима осторожность, поскольку ошибки могут привести к зацикливанию программы.

Функции-переменные

Одной из интересных возможностей PHP являются  *функции-переменные*  (variable functions), то есть динамические вызовы функций, имена которых определяются во время выполнения программы. Хотя в большинстве web-приложений можно обойтись и без функций-переменных, они значительно сокращают объем и сложность программного кода, а также часто снимают необходимость в условных командах if.

Вызов функции-переменной представляет собой имя переменной, за которым следует пара круглых скобок. В круглых скобках могут перечисляться параметры (однако присутствие параметров не обязательно). Обобщенный синтаксис функции-переменной:

$имя\_функции( );

Листинг 4.6 демонстрирует интересную концепцию функций-переменных и наглядно показывает, что функции-переменные способствуют уменьшению объема программного кода. Если бы не эта возможность, функцию пришлось бы выбирать командой if или switch; это привело бы к заметному увеличению объема программного кода и риску появления дополнительных ошибок при кодировании.

Построение библиотек функций

Библиотеки функций -- одно из самых эффективных средств экономии времени при построении приложений. Предположим, вы написали серию функций для сортировки массива. Вероятно, эти функции будут неоднократно использоваться в разных приложениях. Вместо того чтобы постоянно переписывать эти функции в новый сценарий или копировать их через текстовый буфер, гораздо удобнее разместить все функции сортировки в отдельном файле и присвоить ему легко узнаваемое имя (например, array\_sorting.inc). Библиотека array\_sorting.inc служит накопителем для всех функций сортировки.

Предположим, вы хотите воспользоваться функциями библиотеки array\_sorting.inc в сценарии.

Функции. Область видимости

Функции в PHP выполняют важную роль. Они обеспечивают сокращение кода за счет вынесения его в отдельные места. За урок мы научимся создавать функции и передавать в них значения.

Видеоурок

Функции можно назвать небольшими подпрограммами, куда можно вынести повторяющийся код и обращаться к нему, когда это будет нужно. Функции значительно облегчают построение программ, так как нам не надо копировать однотипный код множество раз, а можно просто воспользоваться одной общей функцией.

Многие путают функции и методы и не понимают отличий между ними. На самом деле отличий нет, так как что методы, что функции являются одним и тем же. Функции что записаны вне классов называют функциями, а функции что записаны внутри классов называются методами.

Точно такая же ситуация обстоит с переменным. В классах переменные называются полями, а вне классов - переменными.

В PHP функции создаются при помощи ключевого слова function. Каждая функция может иметь какие-либо параметры или же не иметь их вовсе. Функции способны что-либо возвращать в ходе выполнения кода, если это требуется.

Теперь функция принимает параметр, который будет отображен в консоли.

Если функция должна что-либо вернуть, то прописываем тип данных который будет возвращен.

Область видимости

В PHP есть несколько полей видимости: локальная и глобальная. Если записать глобальную переменную, то такая переменная будем видна повсюду и с ней можно работать отовсюду в документе. Если записать локальную переменную, то такая переменная будет видна лишь в той области, где она записана.

Для создания глобальной переменной её необходимо прописать вне функции, класса или же объекта. Для создания локальных переменных вам достаточно поместить переменную в функцию. Переменная в функции будет видна лишь внутри блока с функцией и нигде более.

Если из функции мы хотим получить доступ к переменной записанной вне функции, то необходимо прописывать ключевое слово global

Помимо этого можно создавать статические переменные, которые не будут становиться пустыми при новом их использовании:

Особенности применения функций для решения практических задач.

Давайте начнем с определения слова «функция». Функция – это отдельный фрагмент кода, который выполняет определенную задачу (или функцию!).

Определение функции

Функция ключевого слова позволяет PHP знать, что ниже, как вы уже догадались, определение функции. Далее, addNumbers – это имя функции. Открывающая и закрывающая скобки содержат аргументы (иногда называемые параметрами), значения, которые вы передадите в функцию при ее вызове. Функции не должны принимать аргументы, в этом случае скобки будут пустыми. Здесь переменные $num1 и $num2 являются произвольными именами, на которые ссылаются, хотя и внутри функции.

Функции не будут выполняться, пока вы не напишете некоторый код для этого

Поскольку в функции использовался return, общая сумма, хранящаяся в $result будет возвращена в оператор, который выполняет функцию. В этом случае значение будет присвоено $total .

При именовании функции имя должно быть уникальным. Он может состоять из букв, цифр и подчеркивания, но не должен начинаться с цифры (те же правила для именования переменной).

Организационные функции

Теперь, когда вы увидели синтаксис функции, следующий шаг – решить, куда поместить код функции.

Поскольку функции выполняются только по запросу, вы можете разместить их вместе в начале вашего кода. Вот пример:

Переместите три функции в их собственный файл с именем functions.php .

Принятие переменного количества аргументов

До сих пор вы имели дело только с функциями, которые принимают фиксированное количество аргументов. Если вы передадите только один аргумент, когда требуется два, вы увидите предупреждающее сообщение, подобное этому:

**Файлы. Текстовые и бинарные файлы.**

Файл – именованный объект, хранящий данные на каком-либо носителе (дискета, винчестер, СD). Файл, как и массив, - это совокупность данных, потому они немного похожи.

Файлы хранятся в каталогах. Допускаются вложенные каталоги (подкаталоги).

Различают два вида файлов: текстовые и бинарные.

Текстовый файл – компьютерный файл, содержащий текстовые данные, как правило, организованные в виде строк.

Текстовый файл может содержать как форматированный (шрифт, начертание, размер и т.п), так и неформатированный текст. В MS-DOS и Microsoft Windows для файлов с неформатированным текстом обычно используется расширение txt. Тем не менее, текстовыми могут являться файлы с любым другим расширением или без оного. Например, исходные коды программ обычно хранятся в файлах с расширениями, соответствующими языку программирования, на котором написаны программы .bas, .pas, .c– это тоже текстовые файлы. Форматированный текс (текст с разметкой) хранится в файлах с расширением .rtf, .htm, .html.

Текстовым файлам противопоставляются двоичные файлы, в которых содержатся данные, не рассчитанные на интерпретацию в качестве текса (например, файлы, хранящие закодированные звук или изображение). Двоичный (бинарный) файл – в широком смысле: последовательность произвольных байтов. Двоичные файлы применяются для хранения нетекстовой информации: изображений (bmp, jpg), исполняемых файлов (exe) и др. Но можно привести в качестве примера формат doc, который используется для хранения текстовой информации. Двоичные файлы хранят информацию почти в том виде, в котором она представляется в памяти компьютера во время работы программы. Поэтому при чтении такого файла практически не выполняется никаких преобразований, что ускоряет процесс чтения. Самый большой недостаток двоичных файлов – их непереносимость.

Файлы делятся на текстовые и нетекстовые (последние иногда называют двоичными, или бинарными). Файл, содержащий программу на Си, - текстовый; файл, который вы можете создать, используя, например, встроенный редактор Norton Commander – тоже текстовый. А вот файл, содержащий, например, рисунок в формате JPEG (да и в любом другом графическом формате), - двоичный. Текстовые файлы отличаются от двоичных двумя особенностями: во-первых, они делятся на строки, каждая из которых заканчивается переводом строки, состоящим из двух символов с кодами 0х0D 0х0А; во-вторых, текстовые файлы заканчиваются «признаком конца файла» - символом с кодом 0х1А (точнее, должны заканчиваться, это условие соблюдается не всегда).

Чтобы ограничить доступ к файлу, в PHP используется функция flock(). Эта функция блокирует доступ к файлу, когда он уже занят одним пользователем, а все остальные запросы ставит в очередь. При освобождении файла он разблокируется, передается первому в очереди пользователю и снова блокируется.

Работа с каталогами.

РНР предоставляет набор функций для работы с каталогами, а именно — функции для создания, удаления и вывода оглавления каталогов. Для этого в РНР имеются функции opendir() и closedir (), аналогичные функциям fopen () и f close () для файлов. Функция opendir () имеет следующий синтаксис:

$dir\_reference =opendir($dir\_path)

где $dir\_path представляет собой путь к открываемому каталогу. Функция opendir () выведет сообщение об ошибке, если указанный каталог не существует. При успешном завершении функция opendir () возвратит дескриптор каталога. Функция closedir() принимает единственный параметр - дескриптор каталога из вызова opendir ().

**PHP и MySQL**

MySQL – это многопоточный быстрый сервер баз данных, разработанный компанией TcX. MySQL является идеальным решением для малых и средних приложений, созданных на PHP и работает на множестве платформ. MySQL поддерживает стандарт языка запросов SQL и имеет множество расширений к стандарту. MySql сервер загружается автоматически вместе с web – сервером и постоянно работает на компьютере – сервере. MySQL поддерживает трехуровневую структуру: базы данных, таблицы, записи и может работать сразу с несколькими базами данных:

Работа с датой и временем.

В распределенных системах, таких, как Интернет, время играет особую роль. Из-за незначительного расхождения системных часов игрок на рынке Forex может потерять десятки тысяч долларов в течение нескольких минут; система деловой разведки ошибется в составлении прогноза; серверы NNTP в процессе синхронизации потеряют важную информацию, нужную пользователю и т.д.

time()

Возвращает текущее абсолютное время. Это число равно количеству секунд, которое прошло с полуночи 1 января 1970 года (с начала эпохи UNIX).

getdate( )

Считывает информацию о дате и времени. Возвращает ассоциативный массив, содержащий информацию по заданному или по текущему (по умолчанию) времени. Массив содержит следующие элементы:

Форматирование даты и времени. Аргументы: строка формата и абсолютное время. Второй аргумент необязателен. Возвращает строку с заданной или текущей датой в указанном формате

Формирование локальной даты и времени.  
 Аргументы: строка формата и абсолютное время. Второй аргумент необязателен. Возвращает строку с заданной или текущей датой в указанном формате. При этом названия месяцев и дней недели извлекается из локали, выбранной с помощью функции setlocate(). Строка формата может содержать следующие коды:

date — Форматирует временную метку Unix

Описание ¶

date(string $format, ?int $timestamp = null): string

Возвращает строку, отформатированную в соответствии с указанным в параметре format шаблоном. Используется метка времени, заданная параметром timestamp (метка времени Unix), или текущее системное время, если параметр timestamp не задан. Таким образом, параметр timestamp является необязательным и по умолчанию равен значению, возвращаемому функцией time().

format

Необязательный параметр timestamp представляет собой метку времени типа int, по умолчанию равную текущему локальному времени, если timestamp не указан или null. Другими словами, значение по умолчанию равно результату функции time().

Для получения метки времени из строкового представления даты можно воспользоваться функцией strtotime(). Кроме того, некоторые базы данных имеют собственные функции для преобразования внутреннего представления даты в метку времени (например, функция MySQL » UNIX\_TIMESTAMP).

Регулярные выражения. Использования регулярных выражений в PHP.

Регулярные выражения - это специальные шаблоны для поиска подстроки в тексте. С их помощью можно решить одной строчкой такие задачи: «проверить, содержит ли строка цифры», «найти в тексте все адреса email», «заменить несколько идущих подряд знаков вопроса на один».

Любое выражение начинается с символа-ограничителя (delimiter по англ.). В качестве него обычно используют символ /, но можно использовать и другие символы, не имеющие специального назначения в регулярках, например, ~, # или @. Альтернативные разделители используют, если в выражении может встречаться символ /. Затем идет сам шаблон строки, которую мы ищем, за ним второй ограничитель и в конце может идти одна или несколько букв-флагов. Они задают дополнительные опции при поиске текста. Вот примеры флагов:

i - говорит, что поиск должен вестись без учета регистра букв (по умолчанию регистр учитывается)

u - говорит, что выражение и текст, по которому идет поиск, исплоьзуют кодировку utf-8, а не только латинские буквы. Без него поиск русских (и любых других нелатинских) символов может работать некорректно, потому стоит ставить его всегда.

Сам шаблон состоит из обычных символов и специальных конструкций. Ну например, буква «к» в регулярках обозначает саму себя, а вот символы [0-5] значат «в этом месте может быть любая цифра от 0 до 5». Вот полный список специальных символов (в мануале php их называют *метасимволы*), а все остальные символы в регулярке - обычные:

Язык регулярных выражений требует писать бекслеш один раз. Однако в строках в одиночных и двойных кавычках в PHP бекслеш тоже имеет особое значение: мануал про строки. Ну например, если написать $x = "\$"; то PHP воспримет это как специальную комбинацию и вставит в строку только символ $ (и движок регулярных выражений не узнает о бекслеше перед ним). Чтобы вставить в строку последовательность \$, мы должны удвоить бекслеш и записать код в виде $x = "\\$";.

Специальные конструкции в регулярках

\d ищет одну любую цифру, \D - один любой символ, кроме цифры

\w соответствует одной любой букве (любого алфавита), цифре или знаку подчеркивания \_. \W соответствует любому символу, кроме буквы, цифры, знака подчеркивания.

Также, есть удобное условие для указания на границу слова: \b. Эта конструкция обозначает, что с одной стороны от нее должен стоять символ, являющийся буквой/цифрой/знаком подчеркивания (\w), а с другой стороны - не являющийся. Ну, например, мы хотим найти в тексте слово «кот». Если мы напишем регулярку /кот/ui, то она найдет последовательность этих букв в любом месте - например, внутри слова «скотина». Это явно не то, что мы хотели. Если же мы добавим условие границы слова в регулярку: /\bкот\b/ui, то теперь искаться будет только отдельно стоящее слово «кот».

Мануал

Синтаксис регулярных выражений в PHP, подробное описание

Функции для работы с регулярными выражениями

Также, есть полезный сайт Regex101, где можно протестировать свою регулярку и проверить, что она найдет в тексте. Помни, что на том сайте бекслеши надо писать ровно один раз, и ставить флаг u не требуется.

**Cookie: понятие, назначение. HTTP-заголовки ответа сервера: понятие, назначение.**

Ку́ки (англ. cookie, букв. — «печенье») — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса.

Заголовки HTTP позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом. В HTTP-заголовке содержится не чувствительное к регистру название, а затем после (:) непосредственно значение. Пробелы перед значением игнорируются.

Пользовательские собственные заголовки исторически использовались с префиксом X, но это соглашение было объявлено устаревшим в июне 2012 года из-за неудобств, вызванных тем, что нестандартные поля стали стандартом в RFC 6648; другие перечислены в реестре IANA, исходное содержимое которого было определено в RFC 4229. IANA также поддерживает реестр предлагаемых новых заголовков HTTP.

HTTP-заголовки сопровождают обмен данными по протоколу HTTP. Они могут содержать описание данных и информацию, необходимую для взаимодействия между клиентом и сервером. Заголовки и их статусы перечислены в реестре IANA, который постоянно обновляется.

Заголовки могут быть сгруппированы по следующим контекстам:

Основные заголовки применяется как к запросам, так и к ответам, но не имеет отношения к данным, передаваемым в теле.

Заголовки запроса содержит больше информации о ресурсе, который нужно получить, или о клиенте, запрашивающем ресурс.

Заголовки ответа (en-US) содержат дополнительную информацию об ответе, например его местонахождение, или о сервере, предоставившем его.

Заголовки сущности содержат информацию о теле ресурса, например его длину содержимого или тип MIME (en-US).

Заголовки также могут быть сгруппированы согласно тому, как прокси (proxies) обрабатывают их:

Сквозные заголовки Эти заголовки должны быть переданы конечному получателю сообщения: серверу для запроса или клиенту для ответа. Промежуточные прокси-серверы должны повторно передавать эти заголовки без изменений, а кеши должны их хранить.

Хоп-хоп заголовки (Хоп-хоп заголовки) Эти заголовки имеют смысл только для одного соединения транспортного уровня и не должны повторно передаваться прокси или кешироваться. Обратите внимание, что с помощью общего заголовка Connection могут быть установлены только заголовки переходов.

Управление кэшированием: понятие, назначение.

Что такое кэширование?

В сфере вычислительной обработки данных кэш – это высокоскоростной уровень хранения, на котором требуемый набор данных, как правило, временного характера. Доступ к данным на этом уровне осуществляется значительно быстрее, чем к основному месту их хранения. С помощью кэширования становится возможным эффективное повторное использование ранее полученных или вычисленных данных.

Как работает кэширование?

Данные в кэше обычно хранятся на устройстве с быстрым доступом, таком как ОЗУ (оперативное запоминающее устройство), и могут использоваться совместно с программными компонентами. Основная функция кэша – ускорение процесса извлечения данных. Он избавляет от необходимости обращаться к менее скоростному базовому уровню хранения.

Небольшой объем памяти кэша компенсируется высокой скоростью доступа. В кэше обычно хранится только требуемый набор данных, причем временно, в отличие от баз данных, где данные обычно хранятся полностью и постоянно.

Обзор кэширования

ОЗУ и работающие в памяти сервисы. Поскольку ОЗУ и работающие в памяти сервисы обеспечивают высокие показатели скорости обработки запросов, или IOPS (количество операций ввода-вывода в секунду), кэширование повышает скорость извлечения данных и сокращает расходы при работе в больших масштабах. Чтобы обеспечить аналогичный масштаб работы с помощью традиционных баз данных и оборудования на базе жестких дисков, требуются дополнительные ресурсы. Использование этих ресурсов приводит к повышению расходов, но все равно не позволяет достигнуть такой низкой задержки, какую обеспечивает кэш в памяти.

С помощью кэширования можно значительно сократить задержки и повысить производительность операций ввода-вывода в секунду для многих рабочих нагрузок приложений с большой нагрузкой на чтение, например порталов для вопросов и ответов, игровых ресурсов, порталов для распространения мультимедиа и социальных сетей.

В этих приложениях можно обращаться к очень большим наборам данных в режиме реального времени через кластеры машин, которые охватывают сотни узлов. Управление этими данными в дисковом хранилище является узким местом таких приложений из-за низкой скорости работы базового

Буферизация. Хэширование.

Хотя блокировки и не влияют на результат выполнения процесса и не отражаются на его виртуальном времени, они сказываются на реальном времени его выполнения. Для интерактивных процессов они могут стать основным фактором, определяющим время реакции процесса. Невыгодны блокировки и для ОС, так как каждая блокировка – это переключение процессов, а, следовательно, накладные расходы. Одним из способов, позволяющим избежать блокировок (или, по крайней мере, уменьшить их количество) является буферизация данных. Буферизация, таким образом, сглаживает различия в скоростях работы производителя и потребителя информации и позволяет избежать излишних блокировок.

Особенно существенный эффект буферизация может дать при последовательном вводе-выводе, так как при вводе ОС может предсказать, какой блок информации понадобится следующим и произвести его упреждающее чтение в буфер. Еще один полезный эффект буферизации – ОС имеет возможность сгруппировать данные в буфере и производить обмен с внешним устройством большими блоками.

Список свободных буферов организован как простая FIFO-очередь: ядро ОС выбирает буфер из головы списка, освобождающийся буфер включает в конец списка. Таким образом, буфер, попавший в список свободных, еще некоторое время находится в этом списке, сохраняя свое содержимое, и может быть выбран из списка, если к нему будет обращение. При обращении к дисковому вводу-выводу основным параметром является адрес блока на внешней памяти. ОС сначала ищет в кеше соответствующий блоку буфер и только при неудачном поиске в кеше назначает свободный буфер и производит физическое обращение к диску. Для ускорения поиска в кеше все буфера распределяются по нескольким спискам, именуемым хеш-очередями. Номер списка определяется как результат весьма простой функции хеширования, аргументом которой является адрес на внешней памяти. Применение хеширования позволяет равномерно распределить буфера по хеш-очередям. Каждый буфер может входить только в одну хеш-очередь, но также может быть включен и в список свободных. Процесс, выдавший запрос на ввод-вывод, блокируется в случаях:

если буфера нет в хеш-очереди и список свободных блоков пуст;

если буфер есть в хеш-очереди, но занят.

При освобождении блока разблокируются все процессы, ждущие освобождения этого или любого блока.

Управление сессиями: принципы, основные функции.

Безопасность сессии ¶

Модуль сессии не гарантирует, что хранимая сессионная информация доступна только пользователю, который создал сессию. Необходимо принять дополнительные меры по защите конфиденциальности сессии, основываясь на связанных с ней данных.

Оценка важности данных, передаваемых в рамках сессии, важна для выбора мер по защите этой информации; все имеет свою цену и обычно дополнительные меры защиты приводят к ухудшению удобства для конечного пользователя. Например, если необходимо защитить пользователя от простейших методов социальной инженерии, следует включить session.use\_only\_cookies. В данном случае со стороны пользовательского ПО обязательна поддержка cookie, иначе механизм сессий не будет работать.

Неадаптивное управление сессиями ¶

В настоящее время PHP использует адаптивное управление сессиями по умолчанию. Адаптивное управление сессиями несёт дополнительные риски.

Если session.use\_strict\_mode включён и обработчик сохранения сессии это поддерживает, неинициализированный сессионный ID отвергается и создаётся новый. Это защищает от атак, которые принуждают пользователя использовать заранее известный ID. Атакующий может размещать ссылки или отправлять письма, которые содержат сессионный ID.

Определённый пользователем обработчик сохранения также может поддерживать строгий сессионный режим, путём реализации функции/метода проверки корректности идентификатора сессии. Все определённые пользователем обработчики сохранения обязаны его реализовывать.

Пересоздание идентификаторов сессий сильно уменьшает риск кражи сессии, соответственно надо на периодической основе запускать session\_regenerate\_id(). Например, пересоздавать идентификатор сессии каждые 15 минут для особо секретных данных. Даже если сессию украдут, она достаточно скоро станет истёкшей и попытка её использовать приведёт к ошибке истёкшей сессии.

Идентификатор сессии *должен* пересоздаваться при повышении привилегий пользователя, например при аутентификации. Функция session\_regenerate\_id() должна вызываться до записи авторизационной информации в $\_SESSION. (session\_regenerate\_id() сохраняет данные текущей сессии автоматически). Убедитесь, что только текущая сессия отмечена как авторизованная.

Несмотря на то, что менеджер сессий может прозрачно управлять временными метками, эта функциональность не реализована. Данные старых сессий сохраняются до момента запуска сборщика мусора. В то же время, разработчики должны убедиться, что данные истёкших сессий удалены.

Потенциальный недобросовестный доступ будет невозможно отследить и проанализировать, если данные сессий будут немедленно удаляться.

Вместо немедленного удаления старых сессий вы должны сохранять их непродолжительное время, например, установив специальный флаг и время окончательного истечения сессии в $\_SESSION, запретив кому-либо обращаться к этим данным.

Если пользователь пытается получить доступ к истёкшей сессии, вы не должны его предоставлять. В этом случае рекомендуется удалять статус "авторизован" со всех активных сессий пользователя, поскольку это очень похоже на атаку.

session.use\_only\_cookies и правильное использование session\_regenerate\_id() могут привести к персональной DoS посредством установки неудаляемой cookie. Если такое происходит, то вы можете попросить пользователя удалить cookie и предупредить его о возможных проблемах с безопасностью. Атакующий может устанавливать вредные cookie через уязвимость в веб-приложении (то есть JavaScript-инъекция), уязвимость в браузерном плагине и т.д.

Внимание

Не недооценивайте риск DoS. use\_strict\_mode=On обязателен для общей безопасности идентификаторов сессий! Все сайты должны использовать use\_strict\_mode.

DoS (отказ в обслуживании) может произойти только тогда, когда аккаунт находится под атакой. Наиболее частая предпосылка для него - JavaScript-инъекция.

Удаление данных сессии ¶

Данные истёкших сессий должны быть недоступны и удалены. Существующий механизм управления сессиями делает это не очень хорошо.

Данные истёкших сессий надо удалять так быстро, как только возможно. С другой стороны, данные активных сессий НЕ ДОЛЖНЫ удаляться сразу же. Для обеспечения этих противоречивых требований, вы ДОЛЖНЫ самостоятельно реализовать механизм контроля за истёкшими сессиями на базе временных меток.

Устанавливайте и управляйте временными метками жизни сессии через $\_SESSION. Запрещайте доступ к данным истёкших сессий. Если обнаружена попытка доступа к данным устаревшей сессии, снимайте статус авторизации со всех активных сессий пользователя и вынуждайте его переавторизоваться. Доступ к данным истёкшей сессии может означать атаку. Для обеспечения такого поведения вы должны отслеживать все активные сессии пользователя.

Замечание: Доступ к истёкшей сессии может также произойти из-за нестабильного сетевого доступа и/или конкурентного доступа к приложению/сайту. Сервер может попытаться установить новый идентификатор сессии через cookie, но пакет "Set-Cookie" может не дойти до клиента из-за плохой связи. Одно соединение может вызвать пересоздание идентификатора посредством session\_regenerate\_id(), а другое, в то же время, может не получить нового идентификатора. Следовательно, вы должны запретить доступ к истёкшим сессиям чуть-чуть позже. То есть управление сессиями на базе временных меток является обязательным.

Короче говоря, не уничтожайте данные сессии ни с помощью session\_regenerate\_id(), ни session\_destroy(), а используйте механизм доступа к сессии на базе временных меток. Пусть session\_gc() сам удаляет старые сессии из хранилища.

Сессии и блокировки ¶

По умолчанию данные сессии заблокированы, чтобы избежать состояния гонки. Блокировка обязательна для обеспечения консистентности данных сессии между запросами.

Активные сессии ¶

Разработчики должны следить за активными сессиями каждого пользователя и оповещать его, сколько есть активных сессий, с каких IP (и где географически), как долго они активны и т.д. PHP не сделает этого за вас. Вы должны это делать.

Есть несколько путей реализации. Вы можете хранить всю нужную информацию в специальной базе данных. Так что, когда сессия будет удалена сборщиком мусора, вы должны это отслеживать и соответственно обновлять свою базу данных.

Управление сессиями на базе временных меток является обязательным для определения устаревших сессий. Если обнаружена попытка доступа к устаревшей сессии, необходимо сбросить флаги авторизации для всех активных сессий пользователя.

Если пользователь не авторизован, проверьте, корректен ли одноразовый ключ автологина. Если ключ корректен, авторизуйте пользователя и установите ему новый одноразовый ключ. Ключ автологина обязательно должен быть одноразовым, то есть никогда не используйте его повторно, а всегда создавайте новый.

Ключ автологина - это очень долгоживущий ключ авторизации. Его надо защищать по максимуму. Используйте параметры path/httponly/secure/SameSite при установке cookie для его защиты. Никогда не передавайте ключ автологина, кроме случаев, когда это необходимо.

Разработчик должен реализовать функционал, который отключает автоматический вход в систему и удаляет ненужные "cookie", установленные для его реализации.

CSRF (Межсайтовая подделка запроса) ¶

Сессии и авторизация не защищают от атак типа CSRF. Разработчики должны самостоятельно реализовывать защиту.

output\_add\_rewrite\_var() может быть использована для защиты от CSRF. Читайте руководство для подробностей.

Замечание: До PHP 7.2.0 использовался один и тот же буфер вывода и INI-настройки для "trans sid". Так что использование output\_add\_rewrite\_var() с PHP более ранних версий не рекомендуется.

Многие фреймворки поддерживают защиту от CSRF. Обратитесь к документации своего фреймворка для более подробной информации.

Начиная с PHP 7.3, для сессионной cookie можно установить атрибут SameSite. Это обеспечит дополнительную защиту против CSRF.

Функции PHP для работы с БД MySQL: установка соединения с сервером MySQL, выбор базы данных, обработка ошибок, выполнения запросов к серверу баз данных, обработка результатов запроса, получения информации о результатах SQL-запросов.

PHP поддерживает работу с базой данных MySQL. Специальные встроенные функции для работы с MySQL позволяют просто и эффективно работать с этой СУБД: выполнять любые запросы, читать и записывать данные, обрабатывать ошибки.

Сценарий, который подключается к БД, выполняет запрос и показывает результат, будет состоять всего из нескольких строк. Для работы с MySQL не надо ничего дополнительно устанавливать и настраивать; всё необходимое уже доступно вместе со стандартной поставкой PHP.

Что такое mysqli?

mysqli (MySQL Improved) — это расширение PHP, которое добавляет в язык полную поддержку баз данных MySQL. Это расширение поддерживает множество возможностей современных версий MySQL.

Как выглядит работа с базой данных

Типичный процесс работы с СУБД в PHP-сценарии состоит из нескольких шагов:

Установить подключение к серверу СУБД, передав необходимые параметры: адрес, логин, пароль.

Убедиться, что подключение прошло успешно: сервер СУБД доступен, логин и пароль верные и так далее.

Сформировать правильный SQL запрос (например, на чтение данных из таблицы).

Убедиться, что запрос был выполнен успешно.

Получить результат от СУБД в виде массива из записей.

Использовать полученные записи в своём сценарии (например, показать их в виде таблицы).

Функция mysqli connect: соединение с MySQL

Перед началом работы с данными внутри MySQL, нужно открыть соединение с сервером СУБД. В PHP это делается с помощью стандартной функции mysqli\_connect(). Функция возвращает результат — ресурс соединения. Данный ресурс используется для всех следующих операций с MySQL.

Проверка соединения

*Первое, что нужно сделать после соединения с СУБД — это выполнить проверку, что оно было успешным.* Эта проверка нужна, чтобы исключить ошибку при подключении к БД. Неверные параметры подключения, неправильная настройка или высокая нагрузка заставит MySQL отвергать новые подключения. Все эти ситуации приведут к невозможности соединения, поэтому программист должен проверить успешность подключения к серверу, прежде чем выполнять следующие действия.

Соединение с MySQL устанавливается один раз в сценарии, а затем используется при всех запросах к БД.

Результатом выполнения функции mysqli\_connect() будет значение специального типа — ресурс. Если подключение к MySQL не удалось, то функция mysqli\_connect() вместо ресурса вернет логическое значение типа «ложь» — false. Хорошей практикой будет всегда проверять результат выполнения этой функции и сравнивать его с ложью.

Выполнение запросов

Установив соединение и определив кодировку мы готовы выполнить свои первые SQL-запросы. Вы уже умеете составлять корректные SQL команды и выполнять их через консольный или визуальный интерфейс MySQL-клиента. Те же самые запросы можно отправлять без изменений и из PHP-сценария. Помогут в этом несколько встроенных функций языка.

Два вида запросов

Следует разделять все SQL-запросы на две группы:

Чтение информации (SELECT).

Модификация (UPDATE, INSERT, DELETE).

При выполнении запросов из среды PHP, запросы из второй группы возвращают только результат их исполнения: успех или ошибку.

Запросы первой группы при успешном выполнении возвращают специальный ресурс результата. Его, в свою очередь, можно преобразовать в ассоциативный массив (если нужна одна запись) или в двумерный массив (если требуется список записей).

Добавление записи

Вернёмся к нашему проекту — дневнику наблюдений за погодой. Начнём практическую работу с заполнения таблиц данными. Для начала добавим хотя бы один город в таблицу cities.

Выражение INSERT INTO используется для добавления новых записей в таблицу базы данных.

Составим корректный SQL-запрос на вставку записи с именем города, а затем выполним его путём передачи этого запроса в функцию mysqli\_query(), чтобы добавить новые данные в таблицу.

Обратите внимание, что первым параметром для функции mysqli\_query() передаётся ресурс подключения, полученный от функции mysqli\_connect(), вторым параметром следует строка с SQL-запросом.

При запросах на изменение данных (не SELECT) результатом выполнения будет логическое значение — true или false, которое будет означать, что запрос выполнить не удалось. Для получения строки с описанием ошибки существует функция mysqli\_error($link).

Функция insert id: как получить идентификатор добавленной записи

Следующим шагом будет добавление погодной записи для нового города. Погодные записи хранит таблица weather\_log, но, чтобы сослаться на город, необходимо знать идентификатор записи из таблицы cities.

Здесь пригодится функция mysqli\_insert\_id(). Она принимает единственный аргумент — ресурс соединения, а возвращает идентификатор последней добавленной записи.

Теперь у нас есть всё необходимое, чтобы добавить погодную запись. Вот как будет выглядеть комплексный пример с подключением к MySQL и добавлением двух новых записей:

Чтение записей

Другая частая операция при работе с базами данных в PHP — это получение записей из таблиц (запросы типа SELECT). Составим SQL-запрос, который будет использовать SELECT выражение. Затем выполним этот запрос с помощью функции mysqli\_query(), чтобы получить данные из таблицы.

В примере выше результат выполнения функции mysqli\_query() сохранён в переменной $result. В этой переменной находятся не данные из таблицы, а специальный тип данных — так называемая ссылка на результаты запроса.

Чтобы получить действительные данные, то есть записи из таблицы, следует использовать другую функцию — mysqli\_fetch\_array() — и передать ей единственным параметром эту самую ссылку. Теперь каждый вызов функции mysqli\_fetch\_array() будет возвращать следующую запись из всего результирующего набора записей в виде ассоциативного массива.

Цикл while здесь используется для «прохода» по всем записям из полученного набора записей. Значение поля каждой записи можно узнать просто обратившись по ключу этого ассоциативного массива.

Как получить сразу все записи в виде двумерного массива

Иногда бывает удобно после запроса на чтение не вызывать в цикле mysqli\_fetch\_array для извлечения очередной записи по порядку, а получить их сразу все одним вызовом. PHP так тоже умеет.

Функция mysqli\_fetch\_all($res, MYSQLI\_ASSOC) вернёт двумерный массив со всеми записями из результата последнего запроса.

Как узнать количество записей

Часто бывает необходимо узнать, сколько всего записей вернёт выполненный SQL запрос. Это может помочь при организации постраничной навигации, или просто в качестве информации. Узнать число записей поможет функция mysqli\_num\_rows(), которой следует передать ссылку на результат запроса.

**Работа с СURL.**

В PHP включена поддержка libcurl - библиотека функций, написанной Дэниелом Стенбергом (Daniel Stenberg), которая позволяет взаимодействовать с множеством различных серверов по многим различным протоколам. В настоящее время libcurl поддерживает протоколы http, https, ftp, gopher, telnet, dict, file и ldap. libcurl также умеет работать с сертификатами HTTPS, посылать запросы к HTTP-серверам методами POST и PUT, загружать файлы по протоколам HTTP и FTP (последнее можно сделать с помощью модуля FTP), использовать прокси-серверы, cookies и аутентификацию пользователей.

]Для подробного описания констант PHP\_INI\_\*, обратитесь к разделу Где могут быть установлены параметры конфигурации.

Краткое разъяснение конфигурационных директив.

curl.cainfostring

Значение по умолчанию для директивы CURLOPT\_CAINFO. Данное значение должно быть указано в виде абсолютного пути.

До PHP 8.0.0 модуль определял три типа ресурсов: дескриптор curl, дескриптор curl\_multi и дескриптор curl\_share.

Перечисленные ниже константы определены данным модулем и могут быть доступны только в том случае, если PHP был собран с поддержкой этого модуля или же в том случае, если данный модуль был динамически загружен во время выполнения.