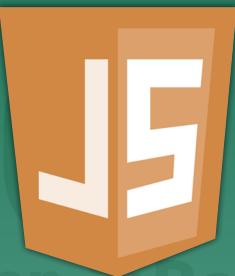


Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery



UNITS 1

Введение в JavaScript

Contents

Введение в JavaScript.....	5
Сценарии, выполняемые на стороне клиента	6
Что такое JavaScript?.....	12
История создания JavaScript	14
Различия между JavaScript и Java, JScript, ECMAScript.....	17
Версии JavaScript.....	19
Понятие Document Object Model	22
Понятие Browser Object Model	26
Внедрение в HTML документы.	
Редакторы кода JavaScript	30
Тег <noscript>.....	34
Основы синтаксиса	40
Регистrozависимость	41
Комментарии	43
Ключевые и зарезервированные слова	44

Переменные. Правила именования переменных.....	47
Типы данных	54
Операторы	59
Арифметические операторы.....	61
Операторы отношений	64
Логические операторы	71
Оператор присваивания.....	76
Битовые операторы.....	82
Приоритет операторов.....	87
Оператор typeof	90
Задание для самостоятельной работы	92
Взаимодействие с пользователем	93
Ввод/вывод данных. Диалоговые окна	94
Условия	101
Что такое условие?	101
If.....	105
if else.....	108
Тернарный оператор ?	112
Switch	114
Задание для самостоятельной работы	120
Циклы	121
Циклы	122
Что такое цикл?.....	126
While	129
Do while.....	134
For.....	136
For in, for of.....	144
Break, continue	145
Понятие метки	148
Задание для самостоятельной работы	150

Функции	151
Что такое функция?	152
Синтаксис объявления функции.....	156
Параметры функции.....	160
Возвращаемое значение функции.	
Ключевое слово return.....	167
Задание для самостоятельной работы	172
Детальнее о функциях	173
Объект arguments	174
Цель и задачи объекта.....	174
Свойство length.....	176
Особенности функций в JavaScript	181
Область видимости переменной	185
Поднятие объявлений	195
Различия деклараций var, let и const	199
Рекурсия.....	202
Задание для самостоятельной работы	209

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

Введение в JavaScript

Сценарии, выполняемые на стороне клиента

Развитие языка разметки гипертекста HTML привело к совсем иному взгляду на понятие «текст» и его эволюцию — «гипертекст». Первоначально в структуру текста были введены дополнительные сущности (*entities*), отделяемые от основного текста так называемыми тегами. Текст начал разделяться на, собственно, сам текст (*пассивный текст*) и гипертекст — текстовые элементы, которые несут в себе некие управляющие инструкции.

Для выполнения этих инструкций пришлось модифицировать средства просмотра документов, точнее, уже гипер-документов. Выражаясь современным языком, появились браузеры — программы, способные отделить обычный текст от гипертекста и обеспечить функциональность последнего.

Задачами первых тегов служили простейшее оформление текста (****, *<i>* и т.п.), а также обеспечение ссылок на различные части документа или на другие документы (<a>). Закон возрастающих потребностей никто не отменил и от гипер-составляющей стали хотеть все большего и большего.

На сегодня мы имеем сложнейшую структуру HTML документов, позволяющих бесконечное вложение объектов в другие объекты, каскадное применение (и отмена) стилей отображения с собственным языком CSS, поддержку событий наведения мыши (`::hover`) и пользовательской активности (`::visited`).

Однако большего хочется всегда. Пришло время вспомнить, что компьютерные программы — это тоже тексты. И задать вопрос: можно ли еще масштабнее расширить понятие гипертекста, введя в него элементы программирования — средств практически неограниченного влияния на отображение страницы, ее поведение и реакцию на различные условия? Очевидно, что реализация такой возможности не могла быть обойдена стороной, и разработчики вполне серьезно задались этой перспективой.

Если в гипертекст будет включен код программы, то кто-то должен эту программу выполнять. Поскольку обработкой гипертекста и отображением страницы занимается браузер, логично, что и выполнять программы придется именно ему. Для того чтобы понять появление «сценариев» давайте рассмотрим особенности, которые связаны с самим процессом выполнением программ.

Исторически сложилось так, что процесс создания и исполнения программ осуществлялся двумя принципиально разными путями. Первый путь предусматривал преобразование исходного кода (текста программы) в исполняемый код, понятный операционной системе или даже сразу процессору вычислительной машины. Исполняемый код также называют нативным (*от англ. native — родной*). Этот код получается в результате процесса компиляции программы и представляет собой исполняемый (запускаемый) файл, чаще всего с расширением *EXE* (*от англ. executable — исполняемый*).

Нативный код сильно зависит от технического состава вычислительной машины, операционной системы, ее версии и т.п. Вспомните, как старые программы прихо-

дится запускать «в режиме совместимости» с устаревшими системами. Для того чтобы использовать программу на разных устройствах ее перекомпилируют из исходного кода для каждого конкретного случая (рис. 1). Типичными представителями компилирующих технологий являются языки C/C++, Delphi, Assembler, Fortran и др.

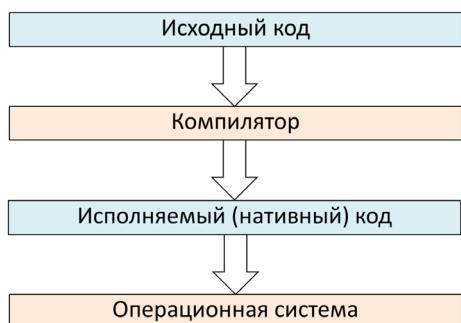


Рисунок 1



Рисунок 2

Для лучшей совместимости программ с различными устройствами параллельно развивалась технология, условно называемая транслирующей. Исходный код при

помощи транслятора преобразовывался в промежуточный байт-код. Этот байт-код исполнялся специальным согласующим звеном с операционной системой — виртуальной машиной или платформой (рис. 2).

Задача обеспечения совместимости перекладывается на плечи разработчиков платформы. После того, как платформа будет создана для данного устройства, на нем будет возможен запуск программы без повторного ее перекомпиляирования (транслирования). Это же является и недостатком данной технологии — для запуска байт-кода необходимо предварительно установить или запустить эту самую платформу. Среди транслирующих технологий можно назвать Java, C#, Python, Visual Basic и др.

Возвращаясь к вопросу внедрения программного кода в структуру HTML документов отметим, что использовать компиляцию с созданием для HTML исполнимого кода в случае Интернета является плохой идеей. Разные посетители страницы используют различные операционные системы, их компьютеры (планшеты, смартфоны) оснащены разными процессорами и т.п. Для каждой новой ситуации пришлось бы создавать различный исполнимый код из одной и той же исходной программы.

Идея трансляции в данном случае выглядит гораздо более перспективной, т.к. браузер, по сути, является промежуточной платформой и исполнителем программы. Однако программа в HTML документе предназначена не для операционной системы, а для самого же браузера. Точнее, для окна (или вкладки) в котором открыт данный HTML документ. То есть браузер, конечно, должен «общаться» с операционной системой для отработки команд,

но он не должен передавать ей управление, направляя результаты обработки команд обратно себе на активное окно или вкладку.

Такая схема выполнения программ получила название интерпретатора (см. рис. 3). Подобным образом работают и другие «диалоговые» средства, например, командная строка Windows (или DOS) — пользователь вводит команду, она выполняется при помощи операционной системы и результат ее выполнения возвращается в диалоговое окно командной строки.

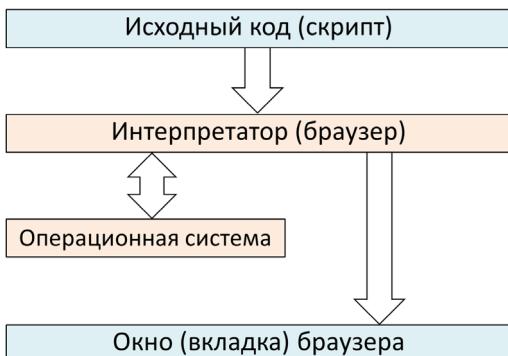


Рисунок 3

В результате, наиболее логичным решением стало оставить внедряемую в HTML документ программу в виде ее исходного кода (изначального текста), а в браузер добавить специальное средство (интерпретатор), выполняющее программу для данного конкретного устройства, на котором браузер установлен.

В отличие от программ, которые компилируются (в исполняемый код) или транслируются (в байт-код) и готовы к исполнению, программа в виде исходного кода (тек-

ста) для интерпретирующих систем получила название «скрипт» (*от англ. script — запись, надпись*). В одном из вариантов перевода используется термин «сценарий» (не путайте с литературно-драматическим произведением). В терминологии Веб-разработки скриптами (или сценариями) называют специальные текстовые вставки в HTML документе, обрабатываемые браузером как программы.

Благодаря распространению и популяризации программирования с применением интерпретирующих систем слово «сценарий» прочно вошло в терминологию программистов и веб-разработчиков. Сами же интерпретаторы получили альтернативное название — языки сценариев или сценарные языки (*иногда — скриптовые языки от англ. scripting language*).

Следует отметить, что идея интерпретирующих языков далеко не новая. Первый язык-интерпретатор был разработан еще в 1960 году и назывался BASIC (*Beginner's All-purpose Symbolic Instruction Code — многоцелевой символьный код для начинающих*). Этот язык какое-то время изучался в рамках школьной и даже университетской информатики в разных странах. Однако термины «скрипт» и «сценарий» попали в обиход гораздо позже, с развитием Веб-технологий.

Подводя итоги, дадим определение: сценарии, выполняемые на стороне клиента, или скрипты — это компьютерные программы, представленные в виде исходного текста и включенные в состав HTML документа (в т.ч. по ссылке на другой документ или файл). Эти программы выполняются браузером по мере загрузки страницы либо в моменты обращения к этим сценариям (вызыва сценарiev) в ответ на действия пользователя, отсчет времени и т.п.

Что такое JavaScript?

Разобравшись с тем, что в состав HTML документа можно включать программы, которые будут выполнены браузером в момент необходимости, логично задать следующий вопрос: «А на каком языке пишут эти программы?». Для этих целей было разработано и стандартизировано специальное средство, известное сегодня как JavaScript.

Прежде всего, JavaScript — это язык программирования. Следует отметить, что его современное использование вышло за рамки скриптового языка браузеров, но об этом позже. Пока что будем считать, что JavaScript — это тот самый язык, на котором пишут клиентские сценарии.

Язык JavaScript непосредственно поддерживается всеми современными браузерами и не требует никаких дополнительных подключенных библиотек или установленных расширений. Достаточно указать теги `<script> </script>`, между которыми можно писать команды сценария.

Как язык программирования, JavaScript имеет почти неограниченные возможности по обработке данных: поддерживает арифметические, логические, символьные операции; может управлять содержимым веб-страницы — расположением, размером, цветом, свойствами элементов, в т.ч. делать это динамически, чем обеспечивая их создание, удаление или анимацию. С его помощью можно связываться с сервером, отправлять или получать произвольные данные. Он способен реагировать на различные события, в частности на действия

пользователя, создавая эффект «общения» с посетителем страницы.

В то же время, JavaScript имеет некоторые ограничения по сравнению с обычными (не скриптовыми) языками программирования. Введены они для обеспечения общей безопасности, например, чтобы зайдя на «плохую» веб-страницу посетитель не оказался с отформатированным диском С или отправленной на чужой сервер папкой «документы».

Так, средствами JavaScript нельзя оперировать с файловой системой компьютера (или другого устройства), кроме случаев, когда пользователь сам выбирает файл, нажимая соответствующую кнопку или «перетягивая» его на страницу. Также ограничен доступ к настройкам системы и другим программам, выполняемым на устройстве.

JavaScript не позволяет управлять другими страницами или вкладками браузера, если эти вкладки не были созданы в результате работы данного сценария JavaScript. Политикой «одного источника» ограничен обмен данными с другими серверами, но разрешен с тем сервером, с которого была загружена данная страница.

Не следует воспринимать приведенные выше ограничения как недостатки языка JavaScript. В некотором смысле, это даже его преимущества, поднимающие доверие к этому языку и делающие его все более популярным. Следует отметить, что в тех случаях, когда JavaScript используется как самостоятельный язык, все эти ограничения снимаются, наделяя JavaScript полной функциональностью современных языков программирования.

История создания JavaScript

Популярность и распространенность JavaScript сопровождается широким и детальным описанием его истории и эволюции. Вы можете с легкостью прочитать в Интернете очерки об авторах этого языка, а также заметки самих авторов о процессе его создания, о прежних названиях языка и о причинах их смены. В рамках нашего урока выделим лишь некоторые основные моменты, связанные с появлением и развитием JavaScript.

В первой половине 90-х годов XX века вычислительная техника и сетевое обеспечение достигли такого уровня, что связь удаленных компьютеров стала доступна массовому потребителю. Из секретных военных технологий сети перешли к мирному использованию. Оцененное пользователями удобство общения по электронной почте и мессенджерам перевело линию развития общества в информационно-коммуникационную стезю, а перед разработчиками поставило новые задачи по созданию средств разработки сетевых ресурсов.

Дело в том, что существовавшие на то время средства программирования были не очень удобными для разработки сетевых приложений. Хотя сделать сайт можно практически на любом языке, но лучше, когда это делается при помощи специальных средств.

В середине 90-х годов появились несколько таких средств: PHP, Java, Ruby, ASP (использованы современные названия), а также были созданы расширения для

существующих языков, упрощающих веб-разработку. Однако все они были нацелены на работу с серверной частью сайта, анализирующей запросы, связывающейся с базами данных и обрабатывающей выборки из них — на задачи, не возникающие на стороне клиента. Возникла задача разработать специальный язык для клиентских сценариев, возможно, адаптировав для этого один из перечисленных выше.

Клиентская часть сайта (то, что мы видим на вкладке браузера) обычно разрабатывалась дизайнерами, понимающими язык разметки HTML, но не имеющими большого опыта программирования на других языках. Среди требований к новому языку сценариев добавилась простота — возможность быстрого освоения веб-дизайнерами.

Один из вариантов простого решения предложила компания Microsoft в виде языка VBScript (*Visual Basic Scripting Edition*). Дело в том, что язык бейсик (*Basic*) был ранее очень популярен, входил в состав пакетов операционных систем от Microsoft (*MS-DOS*) и, как уже отмечалось ранее, изучался многими школьниками и студентами. Программировать на бейсике умели практически все. Использовать привычный и знакомый язык для новых задач, действительно, выглядело как хорошая перспектива. Однако, плохая поддержка языка другими браузерами (кроме Microsoft Internet Explorer) привела к его слабому и медленному развитию и, в итоге, проигрышу альтернативному решению — JavaScript.

Недостатки VBScript дополнительно показали, что языки, разработанные для других задач, довольно сложно переделываются под новые задачи Веб-разработки. Ло-

гичным стал вывод, что проще создать новую технологию «с нуля». Так и появился язык, известный сегодня как JavaScript, — относительно простое средство с конкретным назначением (без претензий на всеобщую универсальность) и рядом ограничений, отмеченных в предыдущем разделе.

Детальнее про историю создания и эволюции JavaScript можно прочитать, например, на сайте <https://auth0.com/blog/a-brief-history-of-javascript/>.

Различия между JavaScript и Java, JScript, ECMAScript

Современное название языка JavaScript было выбрано в свое время исходя из маркетинговых соображений, чтобы заимствовать высокую (на то время) популярность языка программирования Java. Ради справедливости, следует отметить, что кроме созвучного названия, других связей между этими языками нет. Java — это язык разработки приложений, выполняемых специальной платформой (виртуальной машиной), имеет сильную типизацию и использует объектно-ориентированную парадигму. Тогда как JavaScript — язык сценариев, выполняемых браузером, имеет слабую типизацию и использует прототипное программирование. То есть языки отличаются целью, структурой и исполнителем — по сути всем, кроме некоторых синтаксических конструкций, применяющихся также и в других языках.

Стандартизация развивающихся веб-технологий (HTTP, HTML, CSS и т.п.) потребовала и стандартизации используемого в них языка клиентских сценариев. И тут снова свою роль сыграли маркетинговые особенности. Оказалось, что, хотя формально языком JavaScript не владеет какая-либо компания или организация, но название «JavaScript» является зарегистрированным товарным знаком компании Oracle Corporation и не может использоваться другими авторами. Поэтому в стандарте ECMA-262 Европейской ассоциации производителей компьютеров (*European Computer*

Manufacturers Association — Ecma International) язык был описан как ECMAScript.

То есть ECMAScript — это язык, для которого описан и опубликован стандарт синтаксиса, типов данных, блоков, функций и прочих особенностей. JavaScript при этом — это другой язык, реализующий стандарт ECMAScript.

JScript — это альтернативный язык сценариев от компании Microsoft, также реализующий стандарт ECMAScript. JScript развивался параллельно с JavaScript и являлся конкурирующей технологией. В отличие от JavaScript, JScript имел доступ к ресурсам системы и мог применяться для создания локальных приложений. В силу большей популярности JavaScript, JScript ушел из ниши скриптовых языков и превратился в JScript.NET — язык со строгой типизацией, ориентированный на платформу .NET.

Версии JavaScript

Для указания версий языка JavaScript применяется несколько различных обозначений. Одним из них является внутренняя система нумерации версий. Согласно с этой системой, для самой первой, оригинальной сборки языка в далеком 1996 году была введена версия «1.0». Последней из принятый на данный момент версий является «1.8», а точнее — «1.8.5». Работы над версией «2.0» ведутся, но окончательно версия не утверждена. Версии сменялись в связи с появлением новых типов браузеров и зачастую кроме цифры версии указывают название браузера, из-за которого версия была обновлена. Детально про историю версий можно почитать, например, на Википедии по ссылке https://en.wikipedia.org/wiki/JavaScript#Version_history.

Параллельно с внутренними версиями JavaScript развиваются стандарты языка ECMAScript. Они обозначаются сокращением «ES», после которого указывается номер версии. Версия «ES1» была выпущена в 1997 году, «ES2» — в 1998 году, «ES3» — в 1999 году, а версия «ES4» — так и не была принята из-за слишком радикальных изменений, предложенных для внесения. Эти версии морально устарели и имеют лишь историческое значение, как, впрочем, и версии 1.0-1.7 JavaScript, соответствующие этим стандартам.

Прорыв в стандартизации языка произошел спустя 10 лет с принятием в 2009 году версии «ES5». Этот стандарт просуществовал 6 лет, был реализован всеми бра-

узерами и может считаться основой для изучения языка JavaScript реализовавшего ES5, начиная с версии 1.8,. Программы, разработанные с применением «ES5» будут работать и на более современных версиях ES.

В 2015 вышел обновленный стандарт «ES6», расширивший синтаксические возможности языка. С этого же 2015 года было принято решение обновлять стандарты ECMAScript каждый год и вместо номера стандарта указывать год его введения. Так стандарт «ES6» получил альтернативное обозначение «ES2015», «ES7» — «ES2016» и так далее.

В большинстве случаев, для указания особенностей используемого языка JavaScript применяется реализуемый им стандарт ECMAScript, а не внутренняя цифровая версия, так как стандарты ежегодно обновляются, а внутренняя версия JavaScript замерла на цифре «1.8.5» в 2010 году. Поддержка стандартов ECMAScript обеспечивается на уровне отдельных браузеров без внесения изменений в цифровую версию самого JavaScript. В таблице приведены основные браузеры и их версии, начиная с которых включено поддержку стандартов.

	ES5 (2009)	ES6 (2015)	ES7 (2016)
Chrome	23 (2012)	42 (04.2017)	68 (05.2018)
Firefox	21 (2013)	54 (06.2017)	—
IE / Edge*	10 (2012)	14 (08.2016)	—
Safari	6 (2012)	10 (09.2016)	—
Opera	15 (2013)	31 (08.2017)	55 (07.2018)

* Стандарт ES6 (2015) поддерживается только браузером Edge

Как видно из таблицы, на сегодня можно с уверенностью говорить о практически полной поддержке стандарта ES6 (2015) всеми популярными браузерами. В то же время использовать нововведения стандарта ES7 (2016) нужно крайне осторожно, т.к. в некоторых браузерах возможна некорректная работа. Браузер Internet Explorer (*IE*) остановился на поддержке ES5 (2009), после чего ему на замену пришел Edge.

Понятие Document Object Model

Разобравшись с тем, что такое язык JavaScript, обратим наше внимание на то, с чем этот язык работает. Как уже упоминалось, основным назначением JavaScript является работа с веб-страницей на стороне клиента, то есть с данными, отображаемыми на вкладке браузера. В терминологии JavaScript эти данные, полученные от сервера, обобщаются термином «документ» (*document*).

Исторически, разные браузеры по-разному строили документ и его составные части. Это усложняло разработку клиентских сценариев программистами и требовало введения некоторой унификации. Организация W3C («*World Wide Web Consortium*»), основанная для поддерживания и развития стандартов веб-технологий, разработала и опубликовала стандарт, устанавливающий единые требования к составу документа.

Стандарт ввел понятие «Объектная модель документа» DOM (*Document Object Model*) и обобщил способ организации документа в виде взаимосвязанной сети объектов. Для того чтобы понять смысл этой сетевой организации рассмотрим следующий фрагмент HTML кода в виде ненумерованного списка:

```
<ul>
    <li>first element</li>
    <li> second element
        <span>child Node 0</span>
        <a>child node 1</a>
```

```
<p> child node 2 </p>
</li>
<li>third element</li>
</ul>
```

Список (``) содержит в себе три элемента (``), второй из них имеет в своем составе несколько дополнительных элементов: группу (``), ссылку (`<a>`) и абзац (`<p>`). На данный момент неважно как этот список выглядит на странице браузера, важнее, как этот список организуется в документе.

По правилам объектной модели DOM каждый HTML блок представляет собой узел (*англ. node*) общего структурного дерева. Узлы, содержащие в своей структуре другие узлы, являются для них родительскими (*parent node*). В примере таковыми являются сам список (``) и второй элемент списка (` second element`). В свою очередь, вложенные узлы называются дочерними (*child node*) по отношению к их родительскому узлу. В соответствии с порядком их создания выделяется первый (*first child*) и последний (*last child*) дочерние элементы.

Узлы, находящиеся на одном уровне, называются соседними (*sibling*). Для перехода между соседними узлами существуют отношения следующего соседа (*next sibling*) и предыдущего соседа (*previous sibling*). В приведенном примере соседями являются списочные элементы (`first element`, ` second element`, `third element`). Отношения между узлами DOM в нашем примере приведены на рисунке 4.

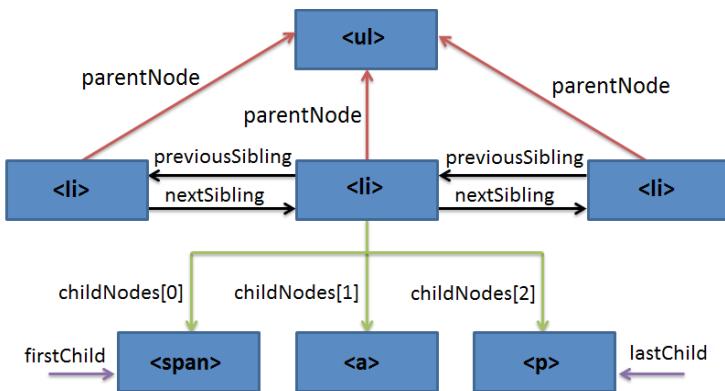


Рисунок 4

Понятие и содержание документа в модели DOM является более широким, чем его HTML содержание. Убедимся в этом практически. Запустите браузер, откройте новую вкладку и перейдите на сайт itstep.org. Когда сайт загрузится, откройте консоль разработчика — нажмите на клавиатуре кнопку «F12» и в появившемся окне выберите вкладку «Console». Введите в консоли слово **document** и нажмите «Enter». Вы увидите ответ «#document» и символ раскрытия группы в виде треугольника ▾ слева от надписи. Нажмите мышью на этот треугольник, после чего раскроется детальное описание объекта (см. рис. 5). Как видно, это знакомый Вам HTML код страницы.

Исследуем состав документ более детально. Введите еще раз **document** и сразу после него поставьте точку (пробел не добавлять). Появится выпадающая подсказка с огромным перечнем составных частей документа. Выберите мышью или стрелками клавиатуры, например, свойство **URL** и после выбора нажмите «Enter». Появится ссылка на загруженный сайт (см. рис. 5).

Понятие Document Object Model

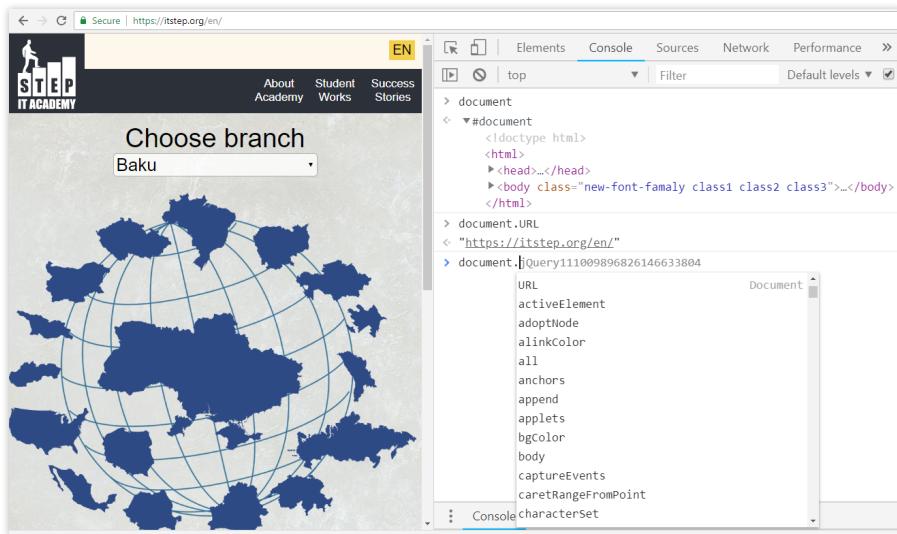


Рисунок 5

Анализируя состав объекта «`document`» можно сделать вывод, что кроме HTML он содержит разнообразную информацию о веб-странице. Детальнее о различных составных частях документа мы будем говорить в последующих уроках.

Отметим, что стандарт DOM — это довольно объемная информация, требующая для своего понимания навыков программирования и опыта разработки веб-ресурсов. На данном вводном этапе подытожим, что DOM позволяет представить веб-страницу как программный объект, собирающий в себе все необходимые данные в виде дерева объектов-узлов и дополнительных параметрах страницы. Более того, структура и имена некоторых узлов устанавливаются стандартами и являются одинаковыми во всех браузерах.

Понятие Browser Object Model

Рассмотренный в предыдущем разделе объект «документ» группирует и представляет доступ к основным данным, размещенным на вкладке браузера. Эти данные должны быть одинаковыми для различных браузеров, различных пользователей, различных устройств, чтобы все посетители сайта видели одну и ту же информацию.

Кроме данных, являющихся общими для всех посетителей, конкретная вкладка конкретного браузера имеет свои данные, уникальные для каждого конкретного пользователя и его браузера, например, свою собственную «историю» — перечень ранее открытых вкладок. Если в браузере выполнить команду «вернуться к предыдущей странице», то ее результат не будет зависеть от свойств и данных сайта, а только от браузера, а точнее, его вкладки. Более того, эти данные меняются от запуска к запуску, от закрытия и открытия вкладок.

Для того чтобы не нарушать стандартизованную модель документа (*DOM*) и в то же время иметь возможность хранить локальные (собственные) данные, в браузере создана своя объектная модель ВОМ (*Browser Object Model*). Эта модель является некоторой оболочкой над объектом `«document»`, на ряду с которым появляются объекты, отвечающие за параметры, относящиеся только к браузеру, и напрямую не связанные с загруженной страницей.

Описанные данные собирает в себе объект `«window»`, представляющий основу объектной модели браузера. На

на рисунке 6 представлены основные элементы этой модели. С большинством из этих элементов мы познакомимся в дальнейших уроках и детально на их разъяснении пока останавливаться не будем.

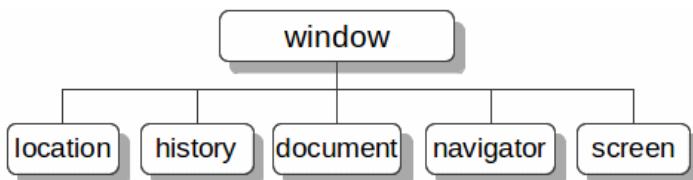


Рисунок 6

Отметим дополнительно, что ВОМ проявляется еще и в том, что «**window**» играет роль глобального объекта — общего пространства для всех других объектов, причем как для HTML, так и для JavaScript.

Со стороны JavaScript этому объекту принадлежат переменные, создаваемые пользователем (дальнейшее в разделе «Область видимости переменной»).

Также в объект «**window**» попадают все элементы HTML, которым присвоены идентификаторы (атрибут **id**). Для того чтобы в этом убедиться создайте новый HTML файл, в котором создайте параграф: (*код также доступен в папке Sources — файл js1_1.html*).

```
<p id="p1">Paragraph with id = p1</p>
```

После этого откройте файл в браузере, убедитесь, что набранная нами надпись отображается корректно. Вызовите консоль разработчика (клавиша **F12**). Введите имя идентификатора «**p1**» и нажмите «**Enter**». В ответ консоль выдаст HTML код нашего параграфа.



Рисунок 7

Ведите еще раз «`p1`» и поставьте точку. Появится подсказка о составе объекта, соответствующего параграфу «`p1`». Выберите поле `innerHTML`. Добавьте знак «`=`» и в кавычках напишите «`New text`». В итоге строка в консоли должна выглядеть как

```
p1.innerHTML="New text"
```

Нажмите «`Enter`» и убедитесь, что текст на вкладке изменился.

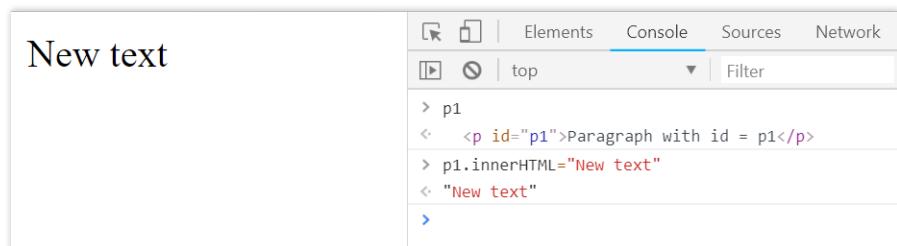


Рисунок 8

Как мы смогли убедиться, наличие у HTML элемента идентификатора (атрибута `id`) приводит к созданию в объекте «`window`» нового объекта с именем как идентификатор.

Тот факт, что переменные JavaScript и объекты HTML с заданным атрибутом «`id`» попадают в один объект ВОМ

«**window**», накладывает ограничения на выбор их имен. Если два разных объекта (или переменные, или объект и переменная) будут иметь одинаковое имя, то доступным окажется только тот, который был создан позже. Он последним переопределит общее имя и «займет» его под себя. Объект «**window**» не может иметь двух разных полей с одинаковым именем.

С точки зрения HTML верстки создание нескольких объектов с одинаковым «**id**» не является грубой ошибкой и на вид страницы не влияет. Однако в скриптовой части такая ситуация недопустима. Поэтому следует крайне внимательно относиться к выбору идентификаторов, особенно для больших и сложных сайтов.

С учетом того, что разработка сайтов часто связана с работой нескольких программистов и дизайнеров, а также необходимостью доработки и расширения сайта с течением времени старайтесь использовать неповторимые имена идентификаторов. Для того чтобы доработав новую функцию сайта, случайно не попасть в существующий идентификатор или переменную, чем нарушить работоспособность уже готовой части.

Внедрение в HTML документы. Редакторы кода JavaScript

Будучи специально разработанными для внедрения в HTML документы сценарии JavaScript включаются в них довольно просто. Как уже упоминалось, никаких дополнительных действий по настройке браузера совершать не требуется, не нужно подключать расширения, дополнения и т.п.

Внедрить код сценария в HTML документ можно двумя способами. Первый — указать теги `<script></script>` в теле документа и между ними поместить код. Второй — написать код в виде отдельного файла (например, `file.js`) и подключить его в документ, указывая файл-источник `<script src="file.js"></script>`.

Обычно отдельными файлами создаются большие по размеру скрипты, прямое указание которых в HTML документе будет мешать его чтению или редактированию. Также возможно указывать удаленный файл, используя ссылку в качестве источника. Чаще всего в отдельных файлах описываются дополнительные функции и не используются сценарии, которые выполняются сразу при загрузке файла в документ.

Непосредственно в HTML документе размещают относительно небольшие сценарии, выполняющиеся при его загрузке и влияющие на свойства страницы и ее отображение. В этих сценариях могут использоваться

функции, описанные в отдельных файлах, если файл был подключен раньше, чем вызываются его функции.

Поскольку JavaScript коды представляют собой обычный текст, для работы с ними подойдет любой текстовый редактор. Для создания и редактирования JavaScript сценариев обычно применяют тот же инструмент, что и для работы с HTML. На данный момент можно подобрать средство на любой вкус — от простейших небольших блокнот-подобных редакторов (например, Notepad, AkelPad, Notepad++) до мощнейших интегрированных средств разработки комбинирующих в себе несколько языков программирования (в частности, Visual Studio, NetBeans, Eclipse).

Сказать, что какой-то из перечисленных продуктов однозначно лучше других, невозможно. Набор основных инструментов есть практически во всех редакторах, а вот дополнительные функции, вроде набора цветных тем оформления, автоматической замены всех подобных слов, подсказок по тегам и их атрибутам — у каждого свои. Очевидно, что «на вкус и цвет» всем не угодишь, и каждый выбирает по себе. Работать с JavaScript кодами можно в любом редакторе, но насколько это понравится именно Вам — предсказать сложно.

Если у Вас есть предыдущий опыт работы или предпочтения относительно какого-то редактора или среды разработки, то можно использовать их и для редактирования JavaScript кода. Единственное, что необходимо отметить, так это неприменимость для работы с JavaScript сценариями текстовых процессоров на подобие Microsoft Word или OpenOffice Writer. Если быть точным, то

применить эти программы можно, но придется указать ряд определенных настроек. В любом случае, текстовые процессоры предназначены совсем для других задач и, даже если настроить их для работы с кодом, эта работа будет неудобной.

Давайте создадим HTML документ и внедрим в него JavaScript сценарий. Создайте новый файл с именем *js1_2.html*, откройте его выбранным Вами текстовым редактором и скопируйте или напечатайте в него следующее содержимое (*код также доступен в папке Sources — файл js1_2.html*). Для файла можно задать произвольное название, но в дальнейшем описании примера считается, что выбрано именно это имя.

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

    <body>
        <script>
            console.log("Hello from script");
        </script>
    </body>
</html>
```

Сохраните файл и откройте его при помощи браузера. Должна появиться пустая вкладка без содержимого. Откройте консоль разработчика (клавиша **F12**) и убедитесь, что в ней выведена надпись «[Hello from script](#)». Рядом с ней указано «[js1_2.html:7](#)», это означает, что данная надпись появилась в результате работы 7-й строки файла

[js1_2.html](#). Несложно убедиться, что эта строка содержит команду `console.log("Hello from script")`.

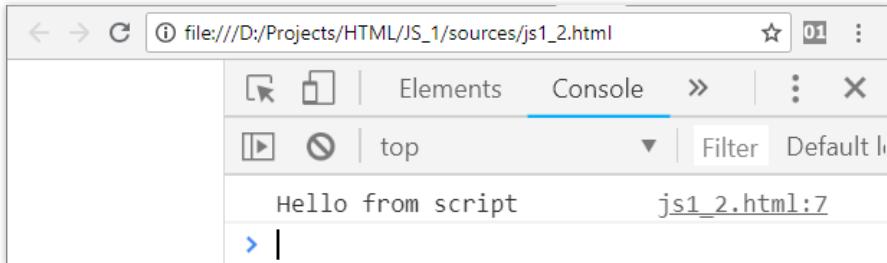


Рисунок 9

Как мы увидели, скрипт внедряется в документ при помощи тегов `<script> </script>`. При загрузке страницы в браузере скрипт автоматически выполняется без дополнительного вмешательства с нашей стороны. Также мы познакомились с командой вывода в консоль «`console.log()`», применяемой для вывода технической информации и используемой для отладки сценариев.

Тег `<noscript>`

Если мы внедряем сценарии в нашу страницу, то мы хотим быть уверенными в том, что они будут выполнены в полном объеме. Иногда от выполнения скриптов зависит не только красота оформления страницы, а ее функциональность и взаимодействие с пользователем. Что же делать, если он (пользователь) все-таки отключит выполнение скриптов в браузере, не подтвердит их включение по сообщению (см. рис. выше) или его устройство вообще не поддерживает работу JavaScript? Для возможности решения подобных задач предусмотрен специальный HTML тег `<noscript>`.

Этот тег может применяться как в заголовочной части HTML документа (`<head>`), так и в его теле (`<body>`). Всё, что заключено между открывающим и закрывающим тегами `<noscript>` и `</noscript>` будет обрабатываться только в том случае, если выполнение скриптов невозможно по каким-либо причинам. В обратном случае, содержимое этих тегов будет проигнорировано и не будет обрабатываться.

Хотя отключение скриптов или отсутствие поддержки JavaScript крайне маловероятно, тем не менее, хорошим стилем разработки веб-ресурсов должно быть предусмотрено и такой вариант настройки конечного устройства. Давайте рассмотрим простейший пример, иллюстрирующий проверку работоспособности клиентских сценариев.

Создайте новый HTML файл. Скопируйте или напечатайте в него следующее содержимое (код также доступен в папке *Sources* — файл *js1_3.html*).

```
<!doctype html>
<html>
    <head>
        <noscript>
            <style>
                #msg{
                    position:absolute;
                    left:0;
                    top:0;
                    width:100%;
                    height:100%;
                    background:#bbb;
                    text-align:center;
                    padding-top:49vh;
                }
            </style>
        </noscript>
        <style>
            *{
                font-size:30px;
            }
        </style>
    </head>
    <body>
        <div id="msg">Allow scripts to use this page
        </div>
        <script>
            window.msg.innerHTML =
            "JavaScript enabled. All right.";
        </script>
    </body>
</html>
```

Основным контентом документа является блок (<div id="msg">) с изначальным текстовым содержимым «Allow scripts to use this page». В качестве активной составля-

ющей (сценария) применена уже использованная нами ранее технология замены содержимого блока (`window.msg.innerHTML = "JavaScript enabled. All right."`). Если выполнение сценариев разрешено, то пользователь увидит надпись, сформированную этим сценарием. Иначе — первоначальный текст, предупреждающий о необходимости разрешить выполнение скриптов.

Дополнительно применим стилизацию блока в случае неработоспособности JavaScript в браузере. Для этого поместим стилевое определение между тегами `<noscript>` и `</noscript>`. Именно таким образом стиль будет активирован только при отключенном JavaScript. В противном случае данное определение будет просто проигнорировано.

Главная идея — расположить блок во весь экран, закрыв собой все содержимое страницы. Его в нашем простом примере нет, но в реальном проекте оно, конечно же, будет. Для достижения идеи используем абсолютное позиционирование блока (`position:absolute`), устанавливаем верхнюю левую точку блока в начало страницы (`left:0; top:0`), ширину и высоту блока задаем на 100% размера (`width:100%; height:100%`). При таком стиле блок займет все пространство вкладки браузера.

Задаем блоку серый фон (`background:#bbb`), выравниваем надпись по центру (`text-align:center`), а также смещаем ее по вертикали к центру окна, задавая верхний отступ на уровне 49% от высоты вкладки (`padding-top:49vh`).

Откройте созданный файл в браузере. Убедитесь, что при включенном по умолчанию JavaScript страница выглядит так, как показано на рисунке 10.

JavaScript enabled. All right.

Рисунок 10

Теперь самое аккуратное — отключаем выполнение скриптов в браузере. В браузере Chrome наберите в адресной строке «<chrome://settings/content/javascript>» и нажмите на переключатель «**Allowed (recommended)**» который должен поменяться на «**Blocked**».

В браузере «Opera» настройки очень похожи. Наберите «<opera://settings/content/javascript>» и увидите такой же переключатель.

В браузере Mozilla Firefox наберите «<about:config>», затем в строке поиска введите «**javascript.enabled**» и сделайте двойной щелчок на найденном поле, чтобы значение «**default=true**» поменялось на «**modified=false**» (обратное переключение также совершается двойным щелчком мыши).

В браузере «Edge» настройки меняются через свойства политики безопасности операционной системы.

В браузере «Safari» перейдите в меню «Настройки» и снимите отметку с поля «**Включить JavaScript**».

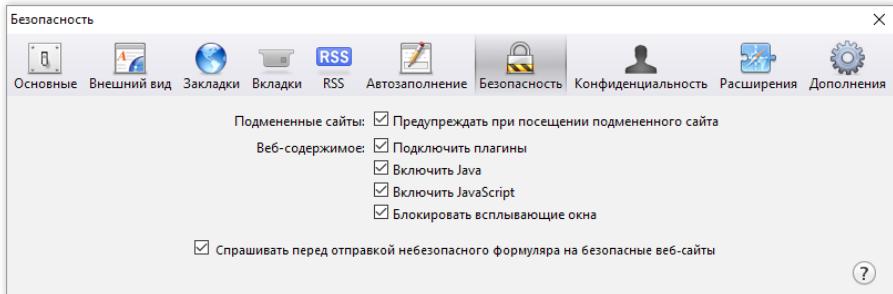


Рисунок 11

После отключения сценариев обновите страницу. Должны вступить в силу стилевые определения для основного блока, размещенные в теге `<noscript>`. А также его текстовое содержимое не будет изменено скриптом, оставшись в первоначальном виде. Внешний вид страницы должен соответствовать следующему рисунку

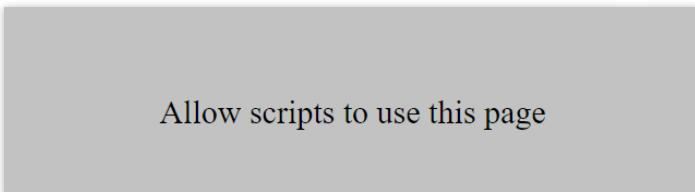


Рисунок 12

Еще раз отметим, что предусмотренная реакция нашего документа на возможное отключение JavaScript в браузере является хорошим стилем разработки и рекомендуется для внедрения в реальных проектах. Конечно, лучше всего предусмотреть нормальную работоспособность сайта и при отключенной поддержки сценариев, но, если это не удается, то, по крайней мере, нужно выдать соответствующее предупреждение для пользователя.

Убедитесь в том, что популярные ресурсы «выдерживают» отключение JavaScript. Пока еще в браузере выключена поддержка сценариев, перейдите на сайт Академии Шаг (<https://itstep.org/ua/>), Фейсбука (<https://www.facebook.com>) или Гугла (<https://www.google.com/>). Обратите внимание на то, что загрузка сайтов проходит normally и общая функциональность сохраняется, хоть и с некоторыми ограничениями. Во многом это обеспечивается применением тега `<noscript>`.

Восстановите изначальные настройки безопасности браузера, включив поддержку сценариев JavaScript для нормальной дальнейшей работы с нашими уроками и другими сайтами.

Основы синтаксиса

Разобравшись с предназначением языка JavaScript и способами внедрения его в состав HTML документа, перейдем к более подробному изучению самого языка.

Наверно, это может показаться странным, но языки программирования во многом похожи на языки обычного человеческого общения. По формальному определению, язык — это знаковая система, предназначенная для передачи смысловой информации. И язык программирования, и язык общения — это средства для подобных целей — передачи информации путем использования специальных символов, слов и их комбинаций. То, что в языке общения называют фразой, в JavaScript называют скриптом. Аналогом литературного произведения на компьютерном языке является программа.

Исходя из основного определения, любой язык состоит из двух основных аспектов — синтаксического и семантического. Семантический аспект касается передачи смысла, а синтаксический — правил использования символов и слов. Например, мы хотим вывести числа от 1 до 10 на экран — это смысл, семантика программы. То, как мы это сделаем, как напишем программу — это синтаксис или синтаксический аспект.

Забегая наперед отметим, что ошибки в программах также делятся на синтаксические и семантические. Синтаксические ошибки возникают, если неправильно использовать языковые знаки — опечататься в выражении или неправильно расставить «знаки препинания»,

отделяющие «слова» друг от друга. Скажем, если вместо «`document`» написать «`documet.`», или для доступа к полю случайно вместо точки «`document.URL`» поставить запятую «`document,URL`». В то же время семантические ошибки приводят к искажению смысла, передаваемого программой. Например, вместо вывода чисел от 1 до 10 произойдет вывод чисел от 0 до 9.

Синтаксические ошибки может обнаружить компилятор (транслятор или интерпретатор) и выдать соответствующее предупреждение на этапе анализа программы. Семантические ошибки обнаружить гораздо сложнее, т.к. программа запускается, работает и завершается успешно. Вот только результат ее работы отклоняется от желаемого. А заметить это может только тот пользователь, который знает о первоначально заложенном смысле программы. Или разработчик, если не поленится провести полное тестирование программы, а не только факт ее запуска.

Регистрозависимость

Поскольку язык — это знаковая система, на первом этапе нужно разобраться с этими знаками. По аналогии с языками общения в языках программирования также применяются литературные символы — буквы. Однако, в отличие от языков общения, большие и маленькие буквы могут иметь (или не иметь) различный смысл.

В письменном языке принято начинать предложение с большой буквы, но из-за написания слова с большой буквы его смысл не меняется. То есть выражения «Принтер печатал» и «Печатал принтер» несут одинаково-

вый смысл, хотя написаны по-разному: с большой или маленькой буквы. Однако в ряде случаев регистр (размер) буквы имеет значение. Например, «Надежда» — это имя, тогда как «надежда» — нет; «OPT» — это аббревиатура, а «орт» — это единичный вектор в алгебре. Зависимость смысла слова от регистра символов называют «регистровозависимость». Обратная ситуация описывается термином «регистронезависимость».

Языки программирования также могут обладать или не обладать регистровозависимостью. Так язык общения с базами данных SQL является регистронезависимым. В нем выражения «`FROM`», «`from`» и «`From`» являются одинаковыми. Тогда как язык JavaScript относится к регистровозисимым. То есть для него приведенные выражения отличаются друг от друга и являются тремя различными словами.

Зависимость языка от регистра символов не является отрицательным или положительным его качеством и никак не влияет на возможности или сферу применения языка. Просто при работе с ним следует помнить о правилах написания выражений и следовать этим правилам.

В случае JavaScript регистровозависимость не позволяет, например, подменить выражение «`document`» на «`Document`» или «`DOCUMENT`», а разрешает использовать только первый вариант (на самом деле, второй и третий варианты тоже могут быть использованы, но не доступа к документу, а для других задач, также отличных друг от друга). Будьте внимательны при чтении дальнейших примеров, обращайте внимание на написание больших и маленьких символов.

Комментарии

Кроме исполнимого кода в состав программы включаются еще и авторские примечания — комментарии. Их можно сравнить с «заметками на полях» — дополнительной информацией, никак не относящейся и не влияющей на содержание основного произведения, но выражающей мнение автора или читателя о нем.

```
// this function calculates mean value  
function calcMVal() {....}
```

Как Вы наверняка догадались, строка комментариев в JavaScript начинается с символов «`//`».

Комментарии используются для того, чтобы напомнить о примененных методиках. Например, указать сайт, с которого был взят пример или формула для вычислений. Также комментарии могут служить неким переводом, выражением в «нормальных словах» того, что делает данный фрагмент кода.

```
a = b % 2 // % - remainder of division  
semesterBegin = '2018-06-04' // from itstep.org site
```

Пример иллюстрирует, что комментарий может быть размещен в той же строке, что и код. По окончанию кода вставляется разделитель «`//`» и дальнейшая часть строки не анализируется интерпретатором, оставаясь лишь пометкой для уточнения деталей.

Комментарии могут использоваться для внешнего оформления программы, отделяя блоки один от другого выразительными средствами.

```
*****  
Computation  
*****/  
...  
...  
*****  
Displaying  
*****/  
...  
...
```

Большие блоки комментариев заключают между маркерами «`/*`» и «`*/`». Первый маркер начинает блок комментария, второй его завершает. Кроме оформления, довольно часто подобный прием используют и для временного отключения какого-либо кода, помещая его в блок комментариев.

Ключевые и зарезервированные слова

Основу любого языка составляют слова — символьные конструкции, имеющие самостоятельный смысл. В случае разговорного языка смысл слов можно узнать из толкового словаря, найдя в нем нужную символьную последовательность.

В языках программирования также существуют слова, смысл которых установлен их разработчиками. Такие слова называют **ключевыми**.

В отличие от разговорного языка, слов в языке программирования немного. Согласно последнему на данный момент стандарту Standard ECMA-262 9th Edition (известному нам также как ECMAScript 2018 или ES9)

ключевыми являются следующие слова: `await`, `break`, `case`, `catch`, `class`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `export`, `extends`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `let`, `new`, `return`, `static`, `super`, `switch`, `this`, `throw`, `try`, `typeof`, `var`, `void`, `while`, `with`, `yield`.

Эти слова воспринимаются интерпретатором как команды, выражения, предназначенные для выполнения. Соответственно, такие слова нельзя использовать для других нужд, например, в качестве имен переменных или функций. Хоть это и не запрещено, но крайне не рекомендуется использовать слова, отличающиеся от ключевых только регистром символов — «`FOR`», «`Var`» или «`typeOf`».

Для «запаса на развитие» языка некоторые слова являются зарезервированными. Они могут не иметь реализации, то есть не соответствовать реальным командам транслятора. Тем не менее, их использование для имен также запрещено, поскольку со временем в новых версиях Javascript реализация произойдет.

В стандарте ES9 зарезервированными являются слова: `enum`, `implements`, `package`, `protected`, `interface`, `private`, `public`.

Перечень ключевых и зарезервированных слов можно уточнить в описании стандарта по ссылке <https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf> (с. 208–209).

Некоторые ключевые и зарезервированные слова старых стандартов (ES1-3) были «освобождены» в более новых редакциях. Таковыми являются: `abstract`, `boolean`, `byte`, `char`, `double`, `final`, `float`, `goto`, `int`, `long`, `native`, `short`, `synchronized`, `transient`, `volatile`.

Хотя применение слов из последнего перечня не запрещено в современных программах, в старых браузерах из-за этого возможны ошибки. Вероятность такой ошибки крайне мала, т.к. практически все браузеры поддерживают минимум ES5, в котором приведенные слова не являются ключевыми. Отказаться от применения этих слов можно лишь во избежание критики со стороны коллег-программистов.

Переменные. Правила именования переменных

Важнейшей частью программирования, послужившей когда-то причиной перехода от первого ко второму поколению языков, являются переменные. В формальном определении переменная — это именованная область памяти.

В простом представлении переменная отвечает за возможность сохранять данные и обеспечивает доступ к ним во время работы программы. В самом простом случае переменную можно сравнить с памятью на калькуляторе: нажали кнопку «**M**» и результат где-то сохранился. При необходимости этот результат можно из памяти извлечь и использовать в других расчетах.

При программировании реальных задач довольно часто приходится иметь дело с промежуточными результатами и дополнительными величинами. Для примера рассмотрим процесс сложения чисел «в столбик». Пусть надо сложить 85 и 77. Складывая последние цифры чисел (5 и 7), мы получаем 12, значит «два пишем, один в уме».

На втором этапе мы складываем уже три цифры: 8, 7 и единицу, запомненную на предыдущем этапе. Получаем 16 и снова «6 пишем, 1 в уме». На третьем этапе запомненную единицу мы записываем в первую позицию конечного результата.

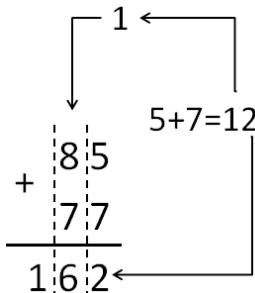


Рисунок 13

Говоря «в уме» мы подразумеваем, что нам как-то нужно запомнить (а лучше записать) единицу, которую следует учесть в дальнейших расчетах. Если мы хотим научить компьютер сложению в столбик, необходимо предусмотреть место, где мы будем запоминать (записывать) данные о переносе (запоминание единицы называется переносом). Для компьютера «в уме» означает «в оперативной памяти». Не имея специального свободного места в памяти, данные просто негде будет хранить. Это место в памяти, с точки зрения программы, и является переменной.

На каждом последующем этапе сложения, может быть, снова придется запоминать единицу, а, возможно, и нет, если сумма цифр будет меньше 10. Выделять для новых данных отдельное место в памяти необходимости нет, так как после использования запомненной единицы один раз ее можно «забывать», то есть заменять следующим результатом «запоминания». Это еще раз подчеркивает смысл слова «переменная» — ее значение может меняться (переменяться) в процессе работы программы.

Когда значение переменной меняется, оно заменяет собой предыдущее значение, хранимое в памяти. Старое

значение утрачивается навсегда, стирается из памяти. Если терять старое значение нельзя и в нем будет необходимость в дальнейшей программе, то следует создать еще одну переменную, отвечающую за другую область памяти, не пересекающуюся с остальными переменными. Количество различных переменных, которые можно создать в программе, ограничивается только объемом оперативной памяти конкретного компьютера.

Для того чтобы использовать переменную в программе необходимо дать ей имя. Имя переменной связывает программное (текстовое) выражение и реальное (физическое) место в оперативной памяти, где хранятся данные. Эти хранимые данные также называются «значением переменной».

Имя переменной представляет собой текстовую запись (литерал), уникальную для каждой переменной и позволяющую однозначно их разделять между собой. Поскольку в языке программирования некоторые литералы используются для операторов, операций, их разделения или группирования, имя переменной не может быть произвольным набором символов.

Принципы и ограничения, накладываемые на возможные имена переменных, называют правилами именования. Правила можно разделить на строгие и рекомендуемые. Часть правил мы уже приводили выше, повторим их здесь для обобщения.

Строгие правила обязательны для исполнения, нарушение их приведет к ошибкам выполнения программы.

- Для имен переменных запрещается использование ключевых или зарезервированных слов (см. раздел «Ключевые и зарезервированные слова»).

- В именах переменных не допускаются пробельные и разделительные символы (табуляция, пробел, запятая, точка с запятой и т.п.).
- Имена переменных не могут содержать символы операций (математических, логических, битовых, — всех), например «`+`», «`!`» или «`&`». Особое внимание следует обратить на то, что знак минуса «`-`», часто используемый для имен классов в HTML, для имен переменных в JavaScript не разрешен.
- В именах переменных не допускаются специальные символы операторов, например «`.`», «`{}`» или «`?:`», а также кавычки всех типов. Чем перечислять все подобные символы, проще сказать, что из них в именах переменных допускаются только «`_`» (*нижнее подчеркивание*) и «`$`» (*знак доллара*).
- Имя переменной не может начинаться с цифры (но может содержать цифры в середине или в конце).

В следующей таблице приведены выражения, которые недопустимы для имен переменных, а также их допустимые варианты с близким по виду написанием:

Выражение	Причина запрета	Допустимые варианты
the X	используется разделительный символ (пробел)	theX the_X
star*	используется символ операции умножения « <code>*</code> »	Star starS
e}{ile	используются символы группирующего оператора « <code>{}</code> »	eXile exile
Bill.Gates	используется символ оператора доступа « <code>.</code> » (точка)	BillGates Bill_Gates

Выражение	Причина запрета	Допустимые варианты
last-element	используется символ операции вычитания «-»	lastElement last_element
[data]	используются символы оператора разыменования «[]»	Data _data_
2GB	имя начинается с цифры	_2GB GB_2
isOdd?	используется символ тернарного оператора «?»	isOdd is_Odd
-=*=!*=-, :-)	все возможные нарушения	-

Рекомендованные правила призваны улучшить читаемость кода, отклонение от них не приводит к ошибкам и программа может выполняться. Очень часто такие правила устанавливают компании для поддержания собственного корпоративного стиля. Однако есть и общие рекомендации, следование которым сделает код значительно лучше для восприятия. Среди таких рекомендаций можно выделить следующие:

- Имена переменных должны отражать смысл хранимого в них значения. Например, для хранения суммарного значения можно использовать имя «`sum`», для обозначения возраста — «`age`» и т.п.
- Имена не должны совпадать (с точностью до регистра) или быть очень похожими на ключевые и зарезервированные слова, на стандартные объекты и функции, а также на имена других переменных. Например, нежелательно использовать имена на подобие «`New`», «`Document`» или «`Window`», переменные с именами «`x1`» и «`X1`» в одной программе.

- Не рекомендуется использовать символы «**l**» (L маленькая) и «**O**» (буква «o» большая), если они не есть частью слова, т.к. эти символы похожи на единицу и ноль соответственно. Плохо будут читаться выражения «**xl**» (похоже на «**x1**») или «**n[O]**» (похоже на «**n[0]**»). При доработке таких программ ошибки практически гарантированы.
- Если имя переменной состоит из нескольких слов, то следует придерживаться единых правил их выделения. Для имен стандартных функций и объектов в Javascript применяется стиль «**lowerCamelCase**», согласно которому слова пишутся подряд, первое слово со строчной (маленькой) буквы, остальные — с прописной (большой).
- Для поддержки возможности править код при помощи простых редакторов (в т.ч. с мобильных устройств) рекомендуется для имен переменных использовать только латинские символы и цифры, ограничив применение спецсимволов Юникода.
- Предпочтение следует отдавать английскому переводу слов вместо написания их транслитерацией. При выводе проекта на международный уровень это упростит работу в группах программистов.
- Для больших проектов нежелательно использовать короткие имена переменных. С большой вероятностью они могут совпасть с переменными из других блоков и повлиять на их работу (детали в разделе «Область видимости переменных»).

Рекомендованные правила не обязательны для исполнения и призваны улучшить внешний вид и читаемость

кода, его анализ, сопровождение, переработку. Следование правилам именования создает «почерк программиста», делает программы узнаваемыми в профессиональном кругу. Согласитесь, некрасивый почерк обычного письма ничего плохого о человеке не говорит, но гораздо приятнее читать текст, написанный красивым почерком. А некоторыми записями можно буквально любоваться, как красиво они выведены. При чтении программ возникают аналогичные эмоции.

В следующей таблице приведены некоторые имена переменных, допустимые с точки зрения программного синтаксиса, но содержащие отклонения от рекомендованных правил

Имя переменной	Замечания
asdf, qwerty, xxxx	имя не отражает смысла хранимых данных
For, BREAK, klass,	имя подобно ключевым словам или операторам
al («L» small), a0, xl («i» capital)	используются символы схожего написания
kilkist, geld, qian	используется транслитерация вместо перевода
sumofarow, date_of_act	отклонение от стиля lowerCamelCase

Хорошими примерами для имен переменных можно считать следующие: «firstLetter», «totalWeight», «dayOfWeek», «cardNumber», «nikName». Даже без анализа программного кода вполне понятно, какой смысл несут сохраненные в них данные.

Типы данных

Структура памяти компьютеров устроена таким образом, что в одной ячейке памяти может храниться только два возможных значения — «0» и «1», образующих один бит информации. Для удобства работы с памятью, чтобы не обращаться к каждому биту отдельно, биты группируются в байты — блоки, обрабатываемые в памяти за одну операцию чтения или записи. Обычно предполагается, что байт состоит из восьми бит, хотя бывают и отклонения от указанного количества в специализированных устройствах.

Повторим выводы предыдущего раздела, что переменная — это именованная область памяти, предназначенная для хранения некоторых данных. И объединим это утверждение с тем фактом, что в памяти могут храниться только блоки единиц и нулей. Выходит, что любая переменная сопоставляется с некоторой последовательностью из «0» и «1» в некотором участке памяти, и других данных содержать в себе просто физически не может.

Однако в задачах прикладного программирования требуются другие данные — числа, символы, даты и т.п. Для того чтобы обеспечить работу с понятными нам данными, нужны определенные правила, которые будут отвечать за преобразование битовой последовательности, сохраненной в памяти (в переменной), к другому представлению. Подобные правила нужны и для операций с разными данными, ведь сложение чисел и сложение дат — это разные операции.

При этом каждая переменная должна хранить в себе информацию о том, какое правило преобразования следует к ней применять, т.к. разные переменные могут отвечать за разные величины. Эта хранимая информация о переменной называется ее «типовом данных».

Количество различных типов данных обычно устанавливается конкретным языком программирования и различно в разных языках. В JavaScript их семь. В C++ около 20, в SQL — около 30. В большинстве языков, в т. ч. и в JavaScript, можно создавать собственные (пользовательские) типы данных, поэтому в самом языке описывается небольшое количество базовых (фундаментальных) типов из которых можно сконструировать другие типы.

Рассмотрим, какие фундаментальные типы данных есть в JavaScript. В стандарте ES9 к ним относятся: **Undefined**, **Null**, **Boolean**, **String**, **Symbol**, **Number**, **Object**.

1. Тип «**Undefined**» имеет только одно значение «**undefined**». Этот тип имеет любая переменная, которой еще не было присвоено значение.
2. Тип «**Null**» также имеет единственное значение «**null**». Это значение применяется там, где ожидается получение объекта, но по каким-либо причинам данный объект не был получен. Другими словами, переменная была создана, но значение в ней отсутствует. Если «**undefined**» соотносится с переменной, которая не участвовала в операции присваивания, то «**null**» возникает в результате неудачной операции присваивания.
3. Тип «**Boolean**» имеет два значения «**true**» и «**false**». Этот тип используется для проверки условий и от-

ношений. Детальнее работа с данным типом данных будет описана в следующем разделе.

4. Тип «**String**» служит для представления строковых (символьных) данных — надписей, выводимых для пользователя или данных, вводимых пользователем на странице.
5. Тип «**Number**» предназначен для хранения чисел. Стандартом установлено, что этот тип имеет ровно **18437736874454810627** различных значений, представляющих число двойной точности в формате 64 бит. Кроме самих чисел среди этих значений есть несколько специальных:
 - ▷ «**NaN**» (*Not-a-Number*) применяется при невозможности преобразовать результат к числу, по сути, обозначает ошибку;
 - ▷ «**Infinity**» — для представления положительной бесконечности ($+\infty$);
 - ▷ «**-Infinity**» — для представления отрицательной бесконечности ($-\infty$);
 - ▷ «**-0**» — для разделения математического формализма **+0** и **-0**.
6. Тип «**Object**» представляет собой набор (коллекцию), объединяющий переменные других типов данных, в т.ч. и других объектов. Частично мы уже рассматривали объекты в разделах об объектной модели документа и браузера.
7. Тип «**Symbol**» объединяет в себе значения, которые не относятся к типу «**String**» но могут быть использованы в качестве ключа в типе «**Object**». Перечень встроенных «символов» можно посмотреть, набрав

в консоли «[Symbol](#)» и поставив в конце точку. Эти символы используются для специализированных алгоритмов и описаны в пункте 6.1.5 стандарта ES9. Для изучения основ программирования этот тип использовать не будет.

Основными типами, предназначенными для работы с данными, являются «[Number](#)» (для работы с числами), «[String](#)» (для работы с текстами) и «[Object](#)» (для группировки данных). Типы данных «[Null](#)» и «[Undefined](#)» используются для проверки успешного выполнения операций и для хранения данных не применяются. Тип данных «[Boolean](#)» обеспечивает сам механизм проверок, хотя может применяться и в других задачах. Тип «[Symbol](#)» имеет крайне специфическое применение и практически не применяется для хранения данных.

Говоря о типах данных, к которым относятся определенные переменные, следует отметить, что языки программирования разделяются на языки со статической и динамической типизацией. В языках со статической типизацией переменная может иметь только один тип данных, который указывается при ее создании (объявлении) и в течение работы программы этот тип не меняется.

Языки с динамической типизацией определяют тип переменной во время каждого обращения к ней. В некоторых языках тип устанавливается при записи значения в переменную и сохраняется до следующей записи (так работает, в частности, язык Python). В JavaScript тип переменной может меняться и при ее чтении в зависимости от того, какое выражение обрабатывается в данный момент.

То есть JavaScript является языком с динамической типизацией, и жесткой привязки конкретной программной переменной к конкретному типу данных нет. Это является отличительной особенностью языка и должно учитываться при правильном составлении выражений, о чём пойдет речь в дальнейших разделах.

Операторы

В одном из вариантов, компьютерную программу можно представить себе как набор некоторых действий, выполнение которых приведет к решению определенного задания, ради которого программа и создается. Действия, из которых состоит программа, могут быть разнообразными, но, конечно же, выполнимыми для того устройства, на котором будет выполняться программа. С одной стороны, всё, что может процессор, так это преобразовывать одни последовательности нулей и единиц в другие по своим сложным внутренним правилам. С другой стороны, языки программирования для того и создаются, чтобы скрыть от программиста сложные правила, заменив их более-менее понятными для человека выражениями.

Путь к упрощению, «очеловечиванию» инструкций языка может привести к избыточности выражений, характерных для языков общения. Разные выражения при этом становятся близкими по смыслу и несут практически одинаковый посыл. Учитывая тот факт, что каждое из таких выражений должно быть переведено на «язык процессора», возникает вполне резонный вопрос об оптимальности: можно ли выделить некоторую основу, базовый набор выражений из которых можно строить любые другие выражения?

Ответом на этот вопрос послужило появление операторов. Операторы или программные инструкции (англ. *statements*) представляют собой синтаксические конструкции (выражения) языка, описывающие опре-

деленные базовые действия. Сами по себе операторы относительно просты, однако при их помощи, комбинируя различные операторы, можно конструировать более сложные выражения, и, в конце концов, написать всю программу. Другими словами, любая программа (кроме пустой, конечно) состоит из набора операторов.

Существует альтернативное определение оператора как наименьшей самостоятельной (автономной) части языка программирования. В нем акцент делается на фундаментальности операторов — нет инструкций в языке, которые будут проще (меньше), чем оператор, и при этом описывать действие (имеется в виду, что меньшее выражение написать можно, но оно будет либо ошибочным, либо бездейственным).

Параллельно с понятием оператора (*statement*) существует понятие операции (*operator*). По смыслу они очень близки и часто путаются, особенно в статьях и книгах, переведенных с английского языка. Отличительной особенностью операции является наличие результата. То есть после выполнения действия остается его значение, например, сложения двух чисел имеет свой результат. Действие оператора может не иметь результата, например разделяющий оператор «;» применяется для отделения операторов и операций между собой и, очевидно, значения не возвращает.

Составные части операции называются operandами. Другими словами, operandы — это то, что участвует в операции. Например, в выражении «**2+3**» оператор представлен инструкцией «**+**», operandы — числами 2 и 3, а само выражение «**2+3**» является операцией, имеющей результат.

В зависимости от характера описываемых действий, операторы и операции разделяют на группы. Например, группа арифметических операторов отвечает за основные математические действия, группа логических операторов — за сравнения и отношения, и т.д. Рассмотрим далее основные группы операторов и операции, которые к ним относятся.

Арифметические операторы

Одним из основных способов обработки данных является их математическое преобразование. Математические (или арифметические) операции представляют собой процесс получения числового результата от некоторого выражения. В это выражение могут входить числа, базовые арифметические операции, а также отдельно определенные математические функции.

Числа представляют собой обычные константы, применяемые для расчетов. Как и в большинстве языков программирования, в JavaScript поддерживается несколько способов записать число:

- «С фиксированной точкой» — запись, совпадающая с обычной математической формой. Например, **123.456**, **-0.25**, или **10**. Ведущий ноль в числе можно опускать, записывая значение **0.1** как **«.1»**. Использовать запятую вместо точки нельзя.
- «С плавающей точкой» или «экспоненциальная форма» представляет число в виде **M·10^E**. **M** называют мантиссой числа, **E** — экспонентой. При записи мантисса отделяется от экспоненты буквой **«E»**. Например, **1E2 = 1·10² = 100**; **3.4E-2 = 3.4·10⁻² =**

0.034. Таким образом удобно задавать большие числа, вроде **1E6 = 1000000**, или, наоборот, малые: **1E-6 = 0.000001**. К тому же так понятнее, сколько нулей следует за числом или перед ним. Данная форма часто применяется в инженерных и научных расчетах.

- Числа в позиционных системах разного базиса можно представить, используя префикс «**0**» (ноль) и символ базиса. Бинарное (двоичное) число записывается с префиксом «**0b**», например, «**0b1110**» соответствует десятичному числу «**14**». Число в системе с базисом **8** (восьмеричной системе) имеет префикс «**0o**» (*от англ. octal*). Система с базисом **16** (шестнадцатеричная система) использует префикс «**0x**». Такие формы делают лучше читаемыми арифметико-логические (битовые) операции, а также часто применяются для указания кодов символов.
- Специальные константы из типа данных **Number**. Кроме приведенных выше (см. раздел «типы данных») констант **+/- Infinity**, **+/-0** и **Nan** возможны выражения **Number.MAX_VALUE** — максимально возможное число, **Number.MIN_VALUE** — минимально возможное число, **Number.EPSILON** — минимально возможная разница между числами, **Number.MAX_SAFE_INTEGER** и **Number.MIN_SAFE_INTEGER** — максимальное и минимальное «безопасное» целое (некоторые функции и преобразования не могут работать с большими значениями).

Основные арифметические операции JavaScript приведены в следующей таблице:

Название операции	Запись в JavaScript	Математическая запись
Сложение	$x + y$	$x + y$
Вычитание	$x - y$	$x - y$
Умножение	$x * y$	$x \cdot y$
Деление	x / y	$x:y$
Остаток от деления	$x \% y$	$x \bmod y$
Возведение в степень	$x ** y$	x^y
Инверсия (смена знака)	$-x$	$-x$
Инкремент (увеличение на 1)	$x++$ или $++x$	$x+1$
Декремент (уменьшение на 1)	$x--$ или $-x$	$x-1$

Дополнительные математические функции имеют специфические области применения и имеют имена, по-добные привычным для нас математическим выражениям. Догадаться о назначении функций несложно, если иметь представление о соответствующем разделе математики. В противном случае маловероятно, что возникнет необходимость такие функции применять и использовать.

При составлении математических выражений следует помнить о приоритете операций. Умножение и деление имеют высший приоритет перед сложением и вычитанием. То есть выражение « $3+2*2$ » будет равно **7**, т.к. сначала буде выполнено умножение $2*2$ и лишь затем сложение. Если нужно поменять приоритет действий, то следует применить группирование операций в круглые скобки. Например, выражение « $(3+2)*2$ » будет равно **10**, т.к. сложение $3+2$ будет выполнено раньше за счет скобок и затем умножено на **2**. Детальнее о приоритете операций поговорим позже в соответствующем разделе.

В отличие от математических формул, в которых можно применять разные скобки для группировки выра-

жений, в JavaScript допускаются только круглые. Скобки можно сколько угодно вкладывать друг в друга, главное, не забывать, что каждой открытой скобке должна соответствовать закрытая. Например, допустимым является выражение « $1+(2*(3/(4%5))+6)$ », но если пропустить скобку « $1+(2*(3/(4%5)+6))$ », то возникнет ошибка.

Пример: необходимо составить программу для расчета формулы:

$$f = x + \frac{2}{x+1} \cdot y.$$

Программист использовал следующий код, найдите в нем ошибку: $f = x + (2 / x + 1)^* y$.

Ответ: приоритет операции деления выше, чем у операции сложения. Значит блок $(2 / x + 1)$ эквивалентно формуле

$$\frac{2}{x} + 1,$$

где двойка делится только на « x » и к результату добавляется единица. В условии задания требовалось рассчитать блок

$$\frac{2}{x+1},$$

то есть выражение « $x+1$ » нужно взять в скобки при делении. Правильное решение: $f = x + 2 / (x + 1)^* y$.

Операторы отношений

После проведения алгебраических расчетов может возникнуть следующая задача — сравнить полученные результаты между собой или с их ожидаемым значением.

Например, один банк предлагает 1% на депозит каждый месяц, а другой повышает ставку с 0.9% до 1.1% в течение года. Какой из банков окажется выгоднее по итогам года?

Проведя расчеты, окажется, что при первой схеме начислений мы получим $x1=12.6825\%$ годовых, тогда как при второй $x2=12.6821\%$. Как определить что выгоднее? Ответ очевиден: нужно сравнить результаты ($x1$ и $x2$) и выбрать больший из них. Для того чтобы автоматизировать этот процесс, в программировании применяются операторы отношений.

Операции отношений или, как их еще часто называют, операции сравнения применяются для проверки предполагаемых алгебраических отношений (больше-меньше) или равенства между операндами. В одной операции участвуют только два операнда.

В отличие от понятных человеку заданий — выбрать из двух чисел большее (или меньшее), операции данной группы только проверяют предполагаемые отношения или равенства. В качестве результата выполнения операции получаются значения типа Boolean: «`true`» или «`false`». Результат «`true`» свидетельствует о том, что проверка пройдена, «`false`» — об обратной ситуации. Другими словами, операция отношения « $x1 > x2$ » дает ответ на вопрос « $x1$ больше $x2$?». Значение «`true`» означает ответ «да», тогда как «`false`» — «нет».

Операторы отношений применяются для составления условий и ветвления программы — выполнения различных действий в зависимости от результата проверки, а также для организации циклов — многократного повторения одного блока. Например, при авторизации

нужно проверить пароль, введенный пользователем, на равенство с его правильным значением. В зависимости от результата (равен или не равен) пользователь увидит различную реакцию программы.

Примером на проверку отношений может быть возрастной ценз. В таком случае возраст пользователя должен быть сравнесен с предельным значением выражениями «больше» или «меньше» и, в зависимости от результата, программа будет выполняться по-разному.

Правила составления условий и ветвления программы будут раскрыты далее (см. раздел «условия»), в данном разделе рассмотрим сами операторы и их применение в операциях отношений и сравнений.

Поскольку JavaScript является языком с динамическим преобразованием типов, операции сравнения делятся на строгие и нестрогие. В нестрогих операциях проверяются только значения операндов — могут ли эти значения быть равны, если оба операнда преобразовать к одному типу. В строгих операциях кроме проверки значений дополнительно проверяются типы, то есть операнды разных типов не могут быть равны между собой ни при каких значениях.

Нестрогие операции сравнения имеют запись «`==`» (проверка на равенство) или «`!=`» (проверка на неравенство). Тот факт, что операции являются нестрогими, проявляется в предварительном преобразовании операндов к примитивным типам (насколько это возможно) и последующим сравнении. Так, например, проверку пройдут сравнения «`3==3`» (число 3 сравнивается со строкой '3'), «`1==true`» (число 1 сравнивается со значением типа Boolean), «`' ==false`» (строка с пробелом сравнивается

со значением типа Boolean). Во всех этих случаях равны между собой значения, которые получаются после преобразования операндов к числовой форме. Откройте консоль браузера, введите приведенные выражения и убедитесь в сделанных выводах (рис. 14).

```

Elements Console
top
> 3==='3'
< true
> 1==true
< true
> ' '==false
< true
>

```

Рисунок 14

Строгие операции проверяют равенство при помощи выражения «`====`», а неравенство — «`!=`». В этом случае операнды разного типа в любом случае не могут быть равными между собой, даже если содержат подобные значения. Используйте консоль, чтобы убедиться, что все предыдущие выражения приведут к отрицательному результату (`false`) (рис. 15).

```

Elements Console
top
> 3==='3'
< false
> 1==true
< false
> ' '==false
< false
>

```

Рисунок 15

Понятие строгой и нестрогой операции применяется только к сравнениям. Операции отношений не имеют такого деления. Для проверки отношений двух операндов применяются следующие операторы:

Запись	Название	Истина если...	Пример
<	меньше	левый операнд меньше правого	2<3 (true) 2<2 (false)
>	больше	левый операнд больше правого	3>2 (true) 2>2 (false)
<=	меньше-равно	левый операнд меньше или равен правому	2<=3 (true) 2<=2 (true)
>=	больше-равно	левый операнд больше или равен правому	3>=2 (true) 2>=2 (true)

примечание — операторы «`<=`» и «`>=`» должны быть записаны именно в такой последовательности знаков. Неправильная запись оператора «`=<`» вызовет ошибку. Однако выражение «`=>`» в JavaScript допустимо и не приводит к ошибке, но имеет совершенно иное значение никак не связанное со сравнениями и отношениями.

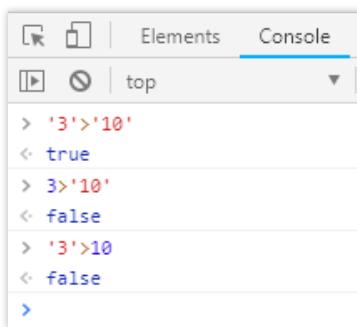
Не будучи разделенными на строгие и нестрогие, операции отношения работают иначе, чем операции сравнения. Если операнды имеют одинаковый тип, то преобразование к примитивным типам не проводится. То есть если сравниваются строки, содержащие числа, то сравнение будет производиться по правилам сравнения строк, а оно отличается от сравнения чисел.

Строки сравниваются символ за символом до тех пор, пока один из них не окажется «большим» — следующим далее по алфавиту (по таблице символов).

Например, сравнение строк «'3'>'10'» приведет к положительному результату, т.к. первый символ левой строки «3» больше первого символа правой строки «1» и дальше сравнение не продолжается. Если необходимо проводить сравнения чисел нужно быть крайне внимательным, чтобы они не оказались в строковом представлении.

Если один из операндов отношения является числом, то второй также приводится к числу. То есть сравнение «3>'10'» будет ложным (*false*). Также ложным будет сравнение «'3'>10».

В обоих случаях один из аргументов является строкой, другой — числом. Из-за этого сравнение происходит по более простому типу — числовому (рис. 16).



The screenshot shows a browser's developer tools console tab labeled "Console". It displays the following interactions:

```

> '3'>'10'
< true
> 3>'10'
< false
> '3'>10
< false
>

```

The first two lines show the comparison of strings '3' and '10'. The third line shows the comparison of a string '3' and a number '10', resulting in false because the string is converted to a number. The fourth line shows the comparison of two numbers, resulting in false.

Рисунок 16

Если необходимо проверять отношения для чисел, но нет уверенности, что переменные хранят их именно в числовом представлении, то применяется прием добавления символа «+» перед переменной. Операция «+x» будет пытаться преобразовать переменную «x» к числовому типу (рис. 17).

```
x='10'  
y='3'  
y>x  
true  
+y > +x  
false
```

Рисунок 17

На рисунке иллюстрируется действие описанного приема. В переменную `x` помещается строковое представление числа «10», в `y` — строковую «3». Если сравнивать переменные между собой, то отношение «`y > x`» будет истинным (для строк '`'3'>'10'`), тогда как выражение «`+y > +x`» будет ложным, поскольку сравниваются числовые значения переменных.

```
x='10px'  
y='3px'  
y > x  
true  
+y > +x  
false  
+x > +y  
false
```

Рисунок 18

Однако описанный прием имеет некоторые ограничения. Если в состав строки в переменной `x` будут включены не-цифры, то выражение «`+x`» приведет к результату «`Nan`». Отметим, что при работе с CSS свойствами довольно часто кроме числа передаются единицы измерения, например, «`px`» (рис. 18).

В таком случае сравнение переменных как строк возможно, но приводит к некорректному результату (с точки зрения логики сравнения чисел). Однако сравнения как «`+y > +x`», так и «`+x > +y`» будут ложными (что вообще-то странно для разных выражений с точки зрения математики). Ситуация возникает из-за того, что преобразования со знаком «`+`» приводят к ошибкам, а появление ошибки в отношениях автоматически считает их ложными.

Забегая наперед скажем, что описанная ситуация решается применением функции «`parseInt`», которая выделяет число игнорируя единицы измерения. То есть сравнение «`parseInt(y) > parseInt(x)`» (или наоборот) приведет к вполне ожидаемым результатам.

Логические операторы

В программировании часто возникают ситуации, когда необходимо проверить сразу несколько условий. В качестве примера рассмотрим задачу по составлению программы, которая должна включать сигнал будильника по рабочим дням в 7:00 утра.

Проведем анализ условий, по которым программа должна включить сигнал. Во-первых, необходимо проверить текущее время на равенство конкретному значению (7:00). Во-вторых, нужно проверить день недели на то, что

он рабочий. При этом предполагаем, что день считается рабочим, если он будет понедельником или вторником, или средой, или четвергом, или пятницей.

Сразу обратим внимание на то, что мы применяем для условий разные соединительные союзы: время должно быть равно 7:00 **И** день является рабочим. При этом день считается рабочим, если он понедельник **ИЛИ** вторник, **ИЛИ** среда (и т.д.).

Для того чтобы немного сократить перебор условий, мы можем пойти от обратного — день является рабочим если он **НЕ** является выходным. Тогда вместо перечисления рабочих дней можно сказать, что день **НЕ** суббота **И**, в то же время, день **НЕ** воскресенье.

В математической логике доказывается, что для составления сложных условий рассмотренных трех логических операций вполне достаточно. Они так и называются: логическое «**И**», логическое «**ИЛИ**» и логическая инверсия или операция «**НЕ**».

Задание для самостоятельной работы. Определите, какие логические операции будут необходимы для составления условий в следующих задачах:

- требуется, чтобы при авторизации совпали (с хранимым значением) логин и пароль, введенные пользователем, но как альтернатива вместо логина может быть указана электронная почта;
- для отбора целевой группы социологического опроса нужно выбрать респондентов, возрастом от 30 до 40 лет, имеющих высшее образование и не работающих в банковской сфере.

Рассмотрим логические операции и их применение в JavaScript более детально.

Логическая инверсия (операция «НЕ» или «NOT») меняет результат операнда на противоположный («`true`» на «`false`», «`false`» на «`true`»). В языке JavaScript инверсия записывается при помощи знака восклицания перед переменной или выражением: «`!x`» или «`!(x > y)`». Инверсия действует на один объект, то есть имеет один operand.

В приведенной выше задаче, инверсия может быть использована для проверки условия «не работающих в банковской сфере». В программе такая запись может иметь вид «`!(profession == 'banker')`».

Логическое «И» или в английском варианте «AND» требует два операнда (два условия). Операция записывается при помощи двойного символа амперсанд «`&&`». Результат операции равен «`true`» только если все operandы равны «`true`». Эту операцию также называют логическим умножением, поскольку результат «`1`» («`true`») достигается, только если два множителя равны «`1`» (значение «`false`» считается равным нулю).

В предложенных выше задачах при помощи логического «И», например, будут объединяться условия совпадения логина и пароля. То есть логин равен хранимому значению И пароль равен хранимому значению. В программе подобное выражение могло бы иметь вид «`login=="user" && password=="MySecret"`».

Логическое «ИЛИ» (в английском варианте «OR») также рассчитано на два операнда. Результат операции равен «`false`» только если все operandы равны «`false`». За это операцию «ИЛИ» сравнивают с логическим сложени-

ем (получить «`0`» при сложении можно, только если все слагаемые равны «`0`»). Программная запись операции имеет вид двойного «прямого слеша» или так называемого символа «трубы» — «`||`».

Операция «**ИЛИ**» используется для реализации альтернатив. Например, для возможности указать логин **ИЛИ** электронную почту. В программном формализме возможна подобная запись `«login=="user" || email=="user@itstep.org"»`.

Логические операции, подобно математическим, могут состоять из нескольких базовых выражений. Например, проверка «логин или почта + пароль» может быть представлена в виде

```
(login=="user" || email=="user@itstep.org") &&  
password=="MySecret"
```

Для логических операций также существует приоритет выполнения. Операция «**И**» (аналог умножения) имеет приоритет выше, чем операция «**ИЛИ**» (аналог сложения). Поэтому в приведенном примере использованы скобки для правильной композиции условия.

Рассмотрим другие примеры.

Задание: выбрать респондентов, возрастом от 30 до 40 лет.

Решение: уточним, что возраст будет учитываться включительно (возможно равенство), тогда выражение будет иметь вид

```
age>=30 && age<=40
```

То есть требуется, чтобы возраст был больше 30 **И**, в то же время, возраст был меньше **40** (включительно, то есть с равенством)

Задание: выбрать респондентов, возрастом от 30 до 40 лет, имеющих высшее образование и не работающих в банковской сфере

Решение: к предыдущему выражению нужно добавить условие о высшем образовании при помощи логического **«И»**, а также инвертированное условие о работе в банковской сфере, тоже при помощи **«И»**

```
(age>=30 && age<=40) && (education=="higher") && !  
(profession == "banker")
```

В данном выражении некоторые скобки можно убрать, так как используются операции одного приоритета, однако расстановка скобок улучшает читаемость выражения, выделяя его составные части.

Найдите ошибку: мы выводим пользователю сообщение о подтверждение действия: «введите **‘OK’** для форматирования диска» и принимаем результат ввода в переменную **«x»**. С учетом того, что пользователь мог ввести **«ок»**, **«Ок»** или **«OK»** пробуем предусмотреть все варианты ввода (помним — регистр символов имеет значение и все эти три строки разные). С точки зрения программы, нас интересует условие для отказа, — если пользователь ввел что угодно, кроме **«ок»**, **«Ок»** или **«OK»**, программа должна выполнить определенное действие по отмене результата запроса (форматирование не должно начаться), то есть условие должно быть истинным, если

введенная строка (`x`) не совпадает ни с одним из допустимых вариантов. Правильным ли будет выражение (если нет, то почему?):

```
(x != 'OK') || (x != 'Ok') || (x != 'ok')
```

Анализ. Предположим, пользователь ввел «`No`». Тогда каждое из сравнений даст результат «`true`», т.к. введенная строка действительно не совпадает ни с одним из значений. Объединенные логическим «**ИЛИ**» условия также приведут к результату «`true`», что и требуется от программы.

Предположим, что пользователь ввел «`Ok`». Тогда первое сравнение даст «`true`», второе — «`false`» (тут совпадение) и третье — «`true`». Логическое «**ИЛИ**» даст в итоге результат «`true`», т.к. хотя бы один из аргументов равен «`true`». А это поведение программы неправильное, ввод строки «`Ok`» должен привести к итоговому результату «`false`».

Ошибка заключается в применении логического «**ИЛИ**» для соединения условий. Если хотя бы одно из условий ложно, также ложным должен быть весь результат. Таким свойством обладает оператор «**И**». Правильная запись условия

```
(x != 'OK') && (x != 'Ok') && (x != 'ok')
```

Оператор присваивания

Все рассмотренные в предыдущих разделах операции приводят к получению некоторого результата. Этот результат может быть выведен в консоль или использован в других операциях.

Бывают случаи, когда полученный результат необходимо сохранить для использования в дальнейших блоках программы. Например, пользователь ввел свое имя и при всех дальнейших обращениях к нему нужно это имя указывать. Значит, в программе это имя следует как-то сохранить и использовать по мере необходимости в диалогах с пользователем.

Как вы уже знаете из предыдущих разделов, для хранения данных, которые могут потребоваться во время выполнения программы, используются переменные (англ. — *variables*). Процесс сохранения результата в переменной называется присваиванием (англ. — *assignment*).

Оператор присваивания записывается при помощи знака «`=`»:

`x = 7`

Слева от оператора указывается переменная, в которую следует записать данные. Если быть точнее, то указывается имя переменной. Справа от оператора приводятся сами данные.

С обеих сторон от знака «`=`» можно добавлять пробелы. Для красоты оформления кода это даже рекомендуется делать. Если идут несколько присваиваний подряд, рекомендуется выравнивать их по знаку «`=`» используя, если нужно, и по нескольку пробелов: `x = 7; x2 = 8; y = 9; y2 = 10`.

В правой части инструкции присваивания может находиться не только конкретное значение, но и выражение, приводящее к значению в результате вычислений. Например,

`x = 1 + 2 * 3`

В таком случае сначала происходит вычисление выражения и только после получения его окончательного результата выполняется присваивание. В выражении присваивания также допустимо использовать переменные. Например:

`x = 1 + 2 * z`

где «`z`» — некоторая переменная. Более того, в выражении можно использовать ту же переменную, которая находится в левой части инструкции присваивания, если она, конечно, содержит ранее присвоенное значение. Рассмотрим следующий пример кода:

`x = 1
x = x + 1.`

В первой строке переменной «`x`» присваивается конкретное значение (1). Оно помещается в память, отведенную для переменной «`x`».

Во второй строке снова указывается операция присваивания, но вместо значения приведено выражение. Значит, сначала будет произведен расчет этого выражения. На момент расчета в переменной «`x`» хранится значение «`1`», присвоенное в предыдущей строке. Оно будет подставлено в выражение «`x + 1`», которое приведет к результату «`2`».

После расчета конечного значения выражения будет выполнено присваивание, то есть полученный результат будет перемещен в переменную «`x`». В результате выполнения второй строки кода значение переменной «`x`» изменится с «`1`» на «`2`».

Следует отметить, что в результате присваивания новое значение попадает в ту же память, в которой находи-

дилось старое значение, так как переменная «`x`» привязывается к одному участку памяти и не меняет его в течение работы всей программы. Вследствие чего новое значение заменяет собой старое. Из-за этого присваивание часто называют «разрушающим присваиванием» (англ. — *destructive assignment*), подчеркивая необратимость действия и полную потерю предыдущего значения, хранимого в переменной.

Рассмотрим чуть подробнее детали и особенности этого процесса.

Как мы уже знаем, переменная представляет собой некий способ доступа к данным, хранящимся в памяти компьютера, а также содержит в себе информацию о способе преобразования этих данных к той или иной форме (тип данных). Когда выполняется присваивание, происходят следующие действия (см. рис. 19).

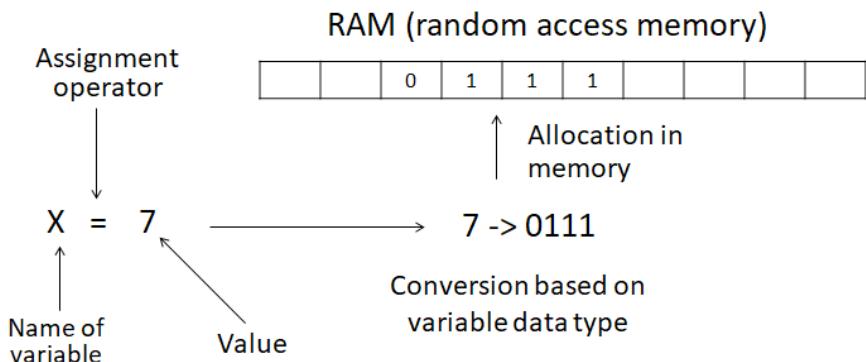


Рисунок 19

Данные, которые нужно сохранить (значение), преобразовываются в двоичный вид с учетом того типа данных, который имеет операция или значение. Напомним, что

в некоторых случаях указать тип данных можно и для самого значения, например запись «`x=7`» задает значение в виде числа (**Number**), тогда как запись «`x="7"`» указывает на строковое (**String**) значение результата.

После преобразования двоичное представление результата перемещается в ту ячейку памяти, за которую отвечает переменная. В приведенном на рисунке примере число 7 превращается в двоичную запись «`0111`» и в таком виде записывается в память. Если говорить совсем точно, то числовой тип данных в JavaScript использует для хранения 64 бита памяти. То есть число 7 на самом деле будет содержать 61 ноль в начале битовой записи и три единицы в конце.

На уровне программы мы не вмешиваемся в описанные процессы и не должны помнить о деталях преобразований, чтобы ими пользоваться. Все эти формальности скрываются под механизмами оператора присваивания, упрощая нам основную работу по созданию программы. Однако следует не забывать, что в JavaScript тип хранимых в переменной данных может поменяться в зависимости от полученного результата конкретной операции. Если после присваивания «`x=7`» в переменной «`x`» окажется число 7, то после повторной операции «`x="7"`» оно будет заменено на строку «`"7"`».

Особенности присваивания

Исторически, присваивание было оператором (инструкцией языка), т.к. не имело возвращаемого результата, а лишь выполняло описанное выше действие по сохранению результата в памяти. Однако, с развитием

языков и технологий программирования ситуация изменилась. В качестве результата присваивания, уже как операции, предлагалось два варианта. Первый предполагал возвращать то же значение, которое было присвоено. Это бы позволило использовать цепную форму записи присваивания «`x=y=z=1`», т.к. результатом операции «`z=1`» было бы то же число «`1`», которое можно поместить в следующую переменную и так далее по цепочке.

Второй вариант возвращаемого значения предполагался в виде «выталкиваемого» предыдущего значения переменной, которое перезаписывается. Это позволило бы сохранить его, если есть такая необходимость. То есть запись «`y=x=1`» помещала бы в «`x`» единицу, а предыдущее значение «`x`» переместила бы в переменную «`y`».

В конечном итоге предпочтение было отдано первому варианту и запись «`x=y=z=1`» является допустимой в JavaScript (попробуйте ввести ее в консоль), как и во многих других современных языках программирования. Второй же вариант получил развитие как обменная идiomа (*англ. copy-and-swap idiom*) и нашел широкое применение в функциях, выполняющих схожее с присваивание действие — если функция перезаписывает какой-то результат, то старое значение возвращается как результат работы функции.

Наличие возвращаемого значения привело к тому, что присваивание приобрело статус «операции», оставаясь при этом оператором — инструкцией языка программирования. Это добавило путаницы к и без того непростым различиям определений «операции» и «оператора» — присваивание является и тем, и другим.

Также следует отметить, что в ряде случаев формально различают два процесса — присваивание и инициализация. Инициализация — это первичное наделение переменной значением, обычно, при ее создании. Последующие процессы передачи данных на хранение считаются присваиваниями. При внешней схожести процессов в некоторых языках программирования для них даже существуют разные операторы.

Битовые операторы

Напомним, что основным способом хранения информации в компьютере являются последовательности бит — единиц и нулей. Также напомним, что основные операции центрального процессора компьютера могут лишь преобразовывать одни битовые последовательности в другие. Для того чтобы использовать такие операции в собственных программах предусмотрены битовые операторы.

С одной стороны, эти операции сложны для интуитивного понимания и не имеют простых аналогов в привычной для нас математике. С другой стороны, эти операции имеют максимальную скорость выполнения, так как не требуют перевода на «язык процессора» и поддерживаются процессором непосредственно.

Необходимость в битовых операциях возникает крайне редко. Чаще всего для них существуют более понятные, хотя и менее эффективные, аналоги. Для языков низкого уровня, которые общаются с процессором почти напрямую, такие операции применяются часто. Для высокоуровневых языков, к которым относится Java-

Script, область применения битовых операций ограничена следующими моментами:

- использование быстродействующих аналогов математических операций;
- желание сэкономить оперативную память;
- криптографические преобразования.

Рассмотрим особенности битовых операций. Эти операции рассчитаны на работу с данными, значение которых может быть либо «**0**», либо «**1**». Результат операции также принадлежит этому множеству. Поскольку различных комбинаций из единиц и нулей не так уж и много, битовые операции пронумерованы и названы собственными именами. Полный их перечень можно узнать в разделе математики «дискретная математика», мы же остановимся на наиболее применимых в программировании случаях.

Пожалуй, одной из самых простых и понятных битовых операций является инверсия. Она применяется к одному операнду и меняет все его биты на противоположные: ноль на единицу, единицу на ноль. В программе битовая инверсия записывается символом тильда «**~x**». Операция по названию и по смыслу подобна логической инверсии («**!**»), рассмотренной в предыдущем разделе, но, действует не только на логическую переменную, а на полное ее битовое представление.

Например, выражение «**!10**» будет вычислено как логическое, т.к. использован логический оператор инверсии «**!**». В результате выражение будет равно «**false**», т.к. любое ненулевое число (в данном случае **10**) соответствует значению «**true**» для типа Boolean.

В случае же использования битового оператора инверсии, выражение «`~10`» приведет к результату «`-11`», получающемуся, если в битовом (двоичном) представлении числа «`10`» все нули поменять на единицы, а единицы — на нули (можете проверить это и предыдущее утверждение, вводя их в консоль браузера). Почему в результате инверсии числа «`10`» получается именно «`-11`» объяснить не так уж и просто, так как придется использовать терминологию теории кодирования. Главным выводом можно сделать то, что смешивать или путать логические и битовые операции не стоит.

Другие битовые операции принимают два операнда, то есть работают с двумя значениями. Одно из них, пусть «`x`», может быть «`1`» либо «`0`» в второе «`y`» также. Результат выполнения операции «`x op y`» проще всего представить в виде «таблицы истинности» этой операции.

<code>y</code>	<code>x</code>	0	1
0	0 op 0	0 op 1	
1	1 op 0	1 op 1	

Таблица позволяет определить результат операции для всех возможных значений аргументов «`x`» и «`y`». Поскольку эти значения могут быть только «`0`» или «`1`», основная часть таблицы имеет размер 2×2 . В ячейке на пересечении столбца и строки с нужными значениями вписывается значение операции (которое тоже ограничено величинами «`0`» или «`1`»).

Как несложно убедиться, существует всего 16 различных способов расставить нули и единицы в таблице

размером 2×2 . Это значит, что и функций существует только 16. Уже упоминалось, что все они имеют собственные названия и детально изучаются в математике, но для программирования ценность представляют только некоторые из них.

x OR y ($x y$)		
y	x	
0	0	1
1	1	1

x AND y ($x & y$)		
y	x	
0	0	0
1	0	1

x XOR y ($x ^ y$)		
y	x	
0	0	1
1	1	0

Дизъюнкция или битовое «ИЛИ» (англ. — OR) равна нулю, только если все операнды равны нулю. В соответствии с синтаксисом JavaScript записывается в виде прямого слеша «`x | y`».

Конъюнкция или битовое «И» (англ. — AND), наоборот, равна единице, только если все операнды равны единице. Записывается при помощи символа амперсанд: «`x & y`».

Операция «исключающее или» (англ. — *exclusive or*) более известная как операция «XOR» равна нулю, если значения operandов одинаковы, и единице — если различны. Имеет запись «`x ^ y`».

Приведенные битовые операции подобны по смыслу логическим операциям, но между ними существует принципиальная разница. Логические операции применяются к операндам типа Boolean и имеют такой же результат. Битовые операции применяются к произвольным данным и действуют на каждый бит операндов отдельно. В результате операции также получается новая битовая последовательность.

$$\begin{array}{r} \& 1\ 1\ 0\ 0 \\ & 1\ 0\ 1\ 0 \\ \hline 1\ 0\ 0\ 0 \end{array} \quad \begin{array}{r} | \ 1\ 1\ 0\ 0 \\ | \ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 1\ 0 \end{array} \quad \begin{array}{r} ^ \ 1\ 1\ 0\ 0 \\ ^ \ 1\ 0\ 1\ 0 \\ \hline 0\ 1\ 1\ 0 \end{array}$$

Рисунок 20

К группе битовых операций также относятся операции битового сдвига. Эти операции применяются уже не к отдельным битам операндов, а к полной битовой записи числа. В результате битового сдвига все биты числа сдвигаются на несколько позиций. Запись «`x >> n`» соответствует сдвигу всех бит на `n` позиций вправо. При этом сохраняется знак числа (самый левый бит не сдвигается). Выражение «`x >>> n`» приводит и к сдвигу самого левого бита. Сдвиг влево на `n` позиций записывается как «`x << n`».

Битовые сдвиги применяются, например, в задачах последовательной передачи данных, когда возможна передача только одного бита (например, в радиоканале — WiFi, или в оптическом кабеле). Для передачи блока данных последовательно сдвигают его биты и передают их одним за другим.

Приведем примеры, когда применение битовых операций может улучшить программный код, увеличивая

скорость выполнения отдельных операций в десятки, а то и сотни раз:

- Умножение на **2** эквивалентно битовому сдвигу влево на **1** позицию: выражение «**x=x*2**» приводит к тому же результату, что и «**x=x<<1**», умножение на **4, 8, 16** и другие степени двойки также эквивалентны битовым сдвигам на **2, 3** или **4** позиции соответственно.
- Полнотью аналогично, целочисленное деление на **2** можно заменить на битовый сдвиг вправо.
- Проверка на четность числа путем получения остатка от деления (**x%2==0**) может быть заменена на проверку последнего бита числа «**x&1==0**».

Использовать битовые операции следует только тогда, когда есть полное понимание их целесообразности. В остальном же эти операции относятся к программированию низкого уровня и в программах на JavaScript вряд ли понадобятся.

Приоритет операторов

Если в одном выражении встречается несколько различных операций, то их порядок выполнения регулируется «приоритетом» — некоторой важностью данной операции. Немного о приоритете мы говорили при рассмотрении математических выражений. В частности было указано, что операции умножения и деления выполняются раньше операций сложения и вычитания. Это вполне согласуется с правилами обычной алгебры.

В случае если в выражении появляются операции разных групп (математические, логические, битовые и т.д.),

то порядок их выполнения (приоритет) становится более сложным и последовательным. Этот порядок можно представить в виде следующей таблицы. Чем выше в таблице операция, тем раньше она будет выполнена в одном и том же выражении. Операции одинакового приоритета выполняются слева направо одна за другой.

Операция / оператор	Обозначение
Группировка выражений	()
Унарные операторы	 ++ -- - !
Возведение в степень	**
Умножение деление остаток от деления	* / %
Сложение вычитание	+ -
Сдвиг битов	<< >> >>>
Отношения	< ≤ > ≥
Равенства, неравенства	== != ==== !==
Битовое «И»	&
Битовое исключающее «ИЛИ»	^
Битовое «ИЛИ»	
Логическое «И»	&&
Логическое «ИЛИ»	

В качестве примера в следующей таблицы приведено несколько выражений, которые требуется рассчитать в программе, их правильные и неправильные записи с объяснениями. Вместо чисел в реальных программах могут применяться переменные, на приоритет операций это не повлияет и ошибки в неправильной записи сохранятся.

Выражение	Неправильно	Объяснение	Правильно
$2 \cdot (3+5)$	$2*3+5$	Приоритет умножения (*) выше, чем у сложения (+). Будет рассчитано $2 \cdot 3 + 5$	$2*(3+5)$
$\frac{1}{2+3}$	$1/2+3$	Приоритет деления (/) выше, чем у сложения. Будет рассчитано $\frac{1}{2} + 3$	$1/(2+3)$
$\frac{1}{2 \cdot 3}$	$1/2*3$	Операции одного приоритета выполняются слева направо. Будет рассчитано $\frac{1}{2} \cdot 3$	$1/2/3$ или $1/(2*3)$
$(2 \cdot 3)5$	$2*3**5$	Приоритет степени (**) выше, чем у умножения. Будет рассчитано $2 \cdot (3)5$	$(2*3)**5$
$2^{\frac{1}{2}}$	$2^{**1/2}$	Приоритет степени (**) выше, чем у деления. Будет рассчитано $2^{\frac{1}{2}}$	$2^{**(1/2)}$

Для иллюстрации роли приоритета в логических операциях используем рассмотренное ранее выражение (из раздела «логические операторы»), применяемое для задачи «выбрать респондентов, возрастом от 30 до 40 лет, имеющих высшее образование и не работающих в банковской сфере»

```
(age>=30 && age<=40) && (education=="higher") && !
(profession == "banker")
```

и уберем из него скобки, получив

```
age>=30 && age<=40 && education==  
"higher" && !profession == "banker"
```

Казалось бы, применение одинаковых операций (`&&`) не должно влиять на расстановку скобок, однако обратим внимание на логическую инверсию `«!»`, приоритет которой выше, чем у логического `«И»` (`&&`). В таком случае, инверсия будет вычисляться первой, и вместо выражения.

```
! (profession == "banker")
```

будет вычислено

```
(!profession) == "banker"
```

Что, очевидно, не соответствует условию из поставленной задачи.

Пользуясь приведенной таблицей приоритетов операций можно оценить результат того или иного выражения. При наличии сомнений относительно правильности формулировки выражения всегда можно воспользоваться оператором группировки (скобками), чтобы наверняка расставить приоритеты операций в выражении.

Оператор `typeof`

Поскольку типы данных в JavaScript могут меняться в процессе выполнения программы, иногда бывает необходимо определить текущий тип данных, который сохранен в некоторой переменной. Как отмечалось выше, результат операций может существенно меняться (вплоть до противоположного) если ее операнды будут того или иного типа.

Чтобы узнать тип данных переменной или результата выражения применяется оператор «`typeof`». В качестве значения оператор возвращает название типа. Например, выражение «`typeof 2`» даст результат «`number`», «`typeof "2"`» — «`string`», «`typeof true`» — «`boolean`». И так далее для всех типов данных. Исключением является выражение «`typeof null`», результат которого представляет собой «`object`». Это сделано по историческим причинам с целью «обратной совместимости» программ в различных версиях языка. Проверить, что переменная «`x`» содержит именно значение «`null`» можно строгим сравнением «`x==null`».

Полученное в результате работы «`typeof`» название типа является строковой величиной. Можно убедиться в этом набрав в консоли «`typeof typeof 1`», то есть запросить тип данных, возвращаемый выражением «`typeof 1`». Для сравнения полученного и ожидаемого типа необходимо название последнего брать в кавычки. То есть для проверки инициализации переменной «`y`» вместо «`typeof y == undefined`» необходимо использовать выражение «`typeof y == 'undefined'`». Последнее условие является одним из самых популярных способов проверки переменных на предмет наличия в них данных.

Хотя «`typeof`» является оператором (языковой инструкцией), его часто используют в форме функции «`typeof(x)`». Это не является ошибкой, т.к. оператор «`typeof`» воспринимает скобки просто как группирующий оператор. Однако, скобки, во-первых, добавляют лишние символы в код и, во-вторых, вызывают еще один оператор (группировки). Поэтому операторная форма записи (без скобок) является более предпочтительной перед функциональной.

Задание для самостоятельной работы

- Предложите имя переменной для хранения данных о максимальной скорости передачи данных (**maximum data transfer speed**).
- Предложите имя переменной для хранения текущего дня недели (**day of week**).
- Составьте инструкцию, вычисляющую значение выражения $2 + \frac{6}{1+2}$.
- Составьте выражение, которое истинно при значениях переменной «**x**» из диапазона **0–9** и ложно для других значений (например, **x=0** — истина, **x=3** — истина, **x=9** — истина, **x=-1** — ложь, **x=10** — ложь).
- Составьте выражение, которое истинно при четных значениях переменной «**x**» из диапазона **0–10** и ложно для других значений (например, **x=0** — истина, **x=3** — ложь, — истина, **x=-1** — ложь, **x=10** — истина, **x=12** — ложь).
- Составьте инструкции, вычисляющие и определяющие типы данных, следующих выражений: **1+true** (сумма числового и логического значений), **'1'+2** (сумма символьного и числового значений), **'1'+false** (сумма символьного и логического значений).

Взаимодействие
с пользователем

Ввод/вывод данных. Диалоговые окна

Одной из основных функций практически любой компьютерной программы является управление разнообразными потоками данных. Одни данные поступают в программу «извне», другие данные «выходят» из программы. Процессы передачи данных в программу и получения данных от нее называют «вводом/выводом данных» (англ. — *Input/Output, IO*).

Под вводом данных обычно подразумеваются процессы чтения данных из файла, получение их со сканнера, а также набор этих данных пользователем на клавиатуре, рисование их световым пером или диктовка на микрофон. Вывод данных — процесс обратного потока информации, обработанной программой: запись файлов, печать на различных устройствах, отображение на мониторах или проекторах, воспроизведение звука динамиками или колонками.

Частной задачей ввода/вывода является обеспечение взаимодействия программы с пользователем. В простейшем случае от пользователя ожидается ввод с клавиатуры некоторых данных, обработка их и отображение полученных результатов. В JavaScript существует несколько способов как получения данных от пользователя, так и вывода результатов их анализа. На данном этапе рассмотрим возможности, которые предоставляют для этих задач диалоговые окна.

Диалоговые окна — это дополнительные небольшие окна, которые появляются на веб-странице и сообща-

ют пользователю некоторую информацию, просят подтверждения действий или ввода некоторых данных.

Простейшее диалоговое окно предназначено для выдачи текстового сообщения пользователю. Такое окно создается командой «`alert()`» в скобках которой указывается значение для отображения. Это может быть простой текст, взятый в кавычки «`alert("Message")`», числовое значение «`alert(10)`» либо имя переменной «`alert(x)`». В последнем случае в окне будет выведено содержимое переменной.

Быстрее всего проверить функциональность диалогового окна можно из консоли браузера. Откройте консоль на любой вкладке браузера и введите команду «`alert("Message")`». Нажмите «Enter» и на странице появится дополнительное окно с сообщением:

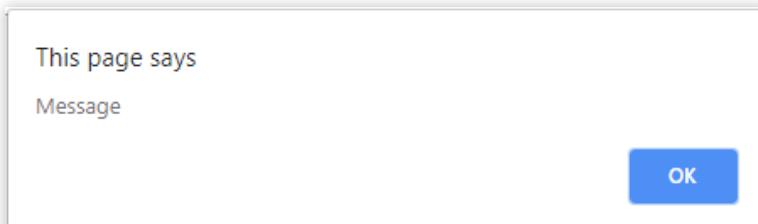


Рисунок 21

Обратите внимание, что вернуться в консоль не получается, пока окно-сообщение не будет закрыто. Такое поведение диалоговых окон называется «модальным» — блокирующим родительское окно на время работы диалога.

После нажатия кнопки «OK» сообщение закрывается и в консоли появляется результат выполнения введенной команды — «`undefined`». Это означает, что на самом деле никакого результата не возвращается. Команда «`alert`»

только оповещает пользователя и не получает от него никакого отклика.

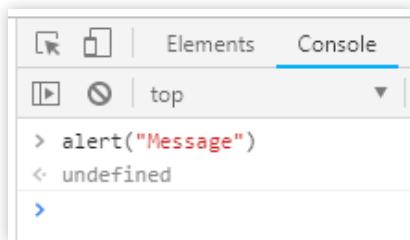


Рисунок 22

Если же отклик от пользователя требуется для коррекции дальнейшего выполнения программы, применяется команда «`confirm()`». Эта команда также выводит сообщение, но на диалоговом окне присутствуют две кнопки «`OK`» и «`Cancel`» (текст кнопок может отличаться в зависимости от языковых настроек браузера). Введите в консоли команду «`confirm("Continue?")`». После нажатия кнопки «`Enter`» должно появиться диалоговое окно как на рисунке

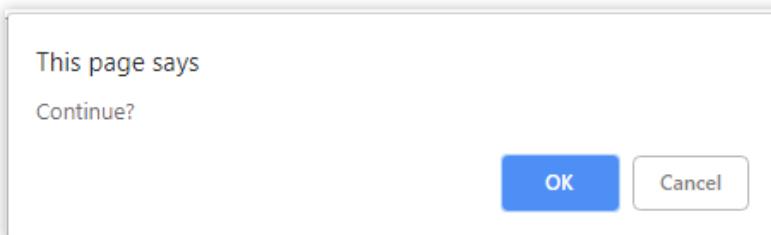


Рисунок 23

Убедитесь, что данное диалоговое окно также является модальным, то есть другие команды не выполняются, пока окно не закрыто. Нажмите «`OK`», окно закроется

и в консоли появится результат «`true`». Повторите ввод команды «`confirm("Continue?")`» (для быстрого повтора ранее введенных команд нажмите стрелку «вверх» на клавиатуре). На этот раз нажмите «`Cancel`». В таком случае результат выполнения будет «`false`».

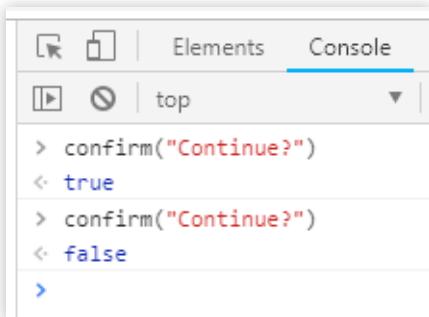


Рисунок 24

Анализируя полученные результаты несложно сделать вывод, что окно, созданное командой «`confirm`» возвращает значение типа «`Boolean`», при этом кнопке «`OK`» соответствует значение «`true`», а кнопке «`Cancel`» — «`false`». Обычно подобные окна используются для подтверждения пользователем некоторых действий или отказа от них.

Для того, чтобы получить от пользователя большее количество данных, предусмотрено диалоговое окно «`prompt`». Внешне оно похоже на окно «`confirm`», но содержит дополнительное поле для ввода данных. Команда «`prompt`» принимает два аргумента — сообщение пользователю и текст, который будет отображен в строке ввода изначально. Исследуем возможности этого диалогового окна — введем в консоли браузера «`prompt("Code for this`

`operation", "")», нажмем «Enter» и увидим новое окно с сообщением, переданным как аргумент команды и пустой строкой ввода (благодаря второму аргументу команды — пустым кавычкам).`

Как и все предыдущие диалоговые окна, окно «prompt» также является модальным, в чем несложно убедиться, попытавшись ввести в консоль новую команду. Введем в строке ввода некоторые данные.

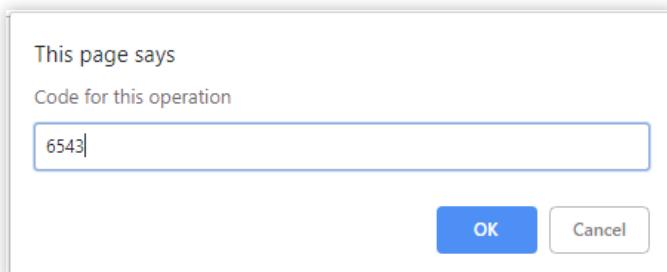


Рисунок 25

После ввода данных можно нажать кнопку «OK» или клавишу «Enter» на клавиатуре. Окно закроется и в консоли появится результат, представляющий введенную нами строку. Обратите внимание, что хотя в примере на рис. выше мы вводили число, результатом работы команды «prompt» все равно является строка, что видно по наличию кавычек в ответе консоли. Вспомните, как в этом можно убедиться при помощи оператора «typeof», и проведите эту проверку.

Повторите команду «`prompt("Code for this operation", "")`» еще раз (напоминаем, повтор команды — стрелка вверх). На этот раз нажмите «Cancel». В качестве возвращенного результата получим значение «`null`».

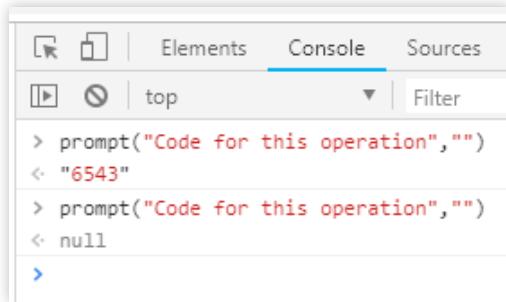


Рисунок 26

Это значение может использоваться для программного контроля того, что пользователь отменил ввод данных. Однако, браузер «[Safari](#)» в случае отмены ввода возвращает не «`null`», а пустую строку. Об этом следует помнить при составлении универсальных (кросс-браузерных) сценариев.

С учетом того, что пустая строка не несет в себе никаких данных, нет принципиальной разницы, какая кнопка была нажата — «[OK](#)» или «[Cancel](#)», если данных все равно нет. С точки зрения логики взаимодействия с пользователем, пустая строка и значение «`null`» эквивалентны и означают одно и то же: пользователь не ввел данные для программы. Поэтому обрабатывать пустую строку и «`null`» вполне логично в одном блоке команд.

Задание: напишите логическое выражение для проверки того, что переменная «`x`» содержит значение «`null`» либо пустую строку.

Еще одной особенностью поведения относительно команды «`prompt`» обладает браузер «[Internet Explorer](#)». Если в нем не указать второй аргумент команды записав, например, «`prompt("Code for this operation")`», то в резуль-

тате в строке ввода изначально будет присутствовать надпись «`undefined`». В других браузерах строка ввода остается пустой при отсутствии второго аргумента, в том числе и в наследнике «IE» — браузере «Edge». Хотя «IE» теряет популярность все больше и больше, лучше не забывать указывать второй аргумент для команды «`prompt`».

Также следует отметить, что использование диалоговых окон в современных сайтах практически отсутствует в силу их скромного дизайна и невозможности его украсить, в т.ч. даже просто увеличить шрифт. Для поддержания общего оформления страницы можно создавать собственные средства диалога с пользователем, что обычно и делается на серьезных ресурсах. Стандартные диалоговые окна находят применение на этапе отладки сценариев, когда полный дизайн сайта еще не готов.

Условия

Разобравшись с тем, как можно организовать взаимодействие программы с пользователем при помощи диалоговых окон, перейдем к рассмотрению способов реакции программы на данные, полученные от пользователя.

При помощи команды `prompt("Code for this operation", "")` программа может получить введенный пользователем код подтверждения некоторого действия и выполнить это действие, но только если полученный код правильный. Если же введенный код окажется неправильным, то действие будет считаться неподтвержденным и выполниться не должно. Можно вывести предупреждение об ошибочном коде, а можно просто отказаться от дальнейшей работы программы.

Таким образом, при составлении программ нам нужен инструмент, позволяющий выполнять какие-то инструкции только при соблюдении определенных критериев, а при отклонении от них либо выполнять другие инструкции, либо вообще ничего не выполнять. Такой инструмент реализуют условия и условные операторы.

Что такое условие?

Под условием (*англ. — condition*) в программировании подразумевается особый вид выражений, приводящих к одному из двух возможных результатов. В языках программирования, поддерживающих тип данных «`Boolean`» (в частности в `JavaScript`), эти результаты соответствуют константам `«false»` и `«true»`. Альтернативно, можно говорить «условие ложно» либо «условие истинно». В старых

языках, не имеющих поддержки типа «Boolean», результатами условий считались значения «0» или «1».

Много хороших примеров условных выражений предоставляет современный маркетинг: «при покупке на сумму свыше X вы получите скидку 10%», «при заказе 3-х товаров доставка бесплатно», «при заправке полностью бака — кофе в подарок». Каждое выражение имеет два исхода — либо вы получите указанное предложение, либо вам в этом предложении будет отказано. А какой из исходов будет реализован, определяет предшествующее условие, которое может быть либо выполненным (истинным), либо нет (ложным).

Также и в программировании, основная цель условия состоит в разделении программы на два пути — по одному из них программа пойдет, если условие истинно, по другому — в противном случае. Имеется в виду, что в программе будет два блока команд, и какой из них выполнится при работе программы, будет известно после проверки дополнительных условий. Как вариант, второй блок может быть пустым (то есть отсутствовать), в таком случае, при ложном условии ничего выполняться не будет.

В графическом представлении алгоритмов условие изображается в виде ромба, внутри которого записывается условное выражение. Из боковых углов ромба выходят две стрелки с подписями «+» и «-». Они определяют пути дальнейшего выполнения программы в зависимости от истинности (+) или ложности (-) данного условия. Алгоритмические схемы или аналогичные им UML диаграммы улучшают наглядность при разработке или анализе программ.

Условия

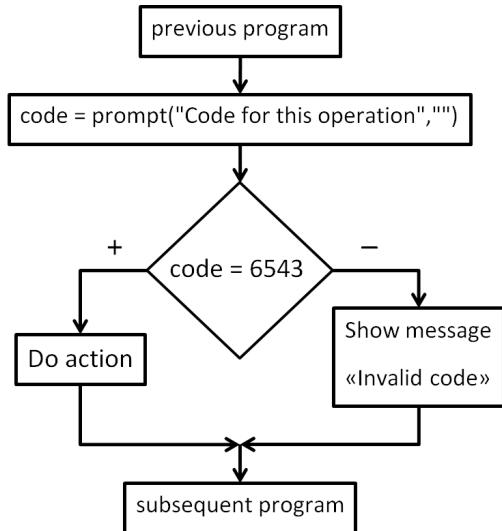


Рисунок 27

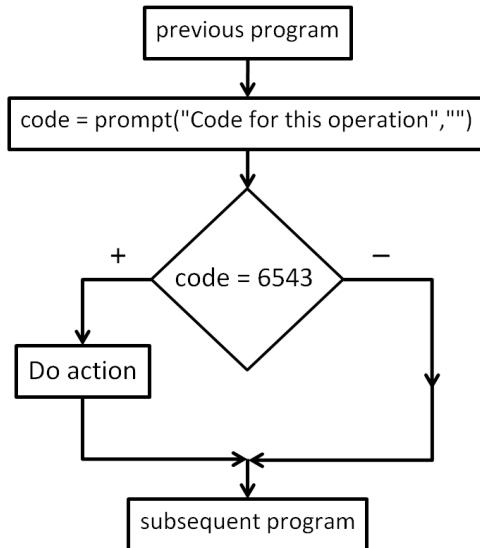


Рисунок 28

Обратим внимание на то, что оба блока команд должны быть описаны на этапе создания программы, и в момент ее запуска возможны оба пути выполнения. Ведь мы заранее не знаем, введет пользователь правильный код или нет.

В практическом программировании условия обычно применяются для:

- Подтверждения действий, авторизации (проверки логина и пароля), разблокировки программ.
- Проверки ограничений: на возраст пользователя, на время суток, на день недели.
- Проверки правильности пользовательского ввода. Очень часто, особенно в Веб-разработке, от пользователя требуется адрес электронной почты или номер телефона. При вводе этих данных пользователь может опечататься, забыть переключить раскладку клавиатуры или нарочно исказить вводимые данные. Каждое из вводимых значений должно быть проверено отдельным условием.
- Проверки успешной работы других частей программы, если от них зависят дальнейшие процессы, проверки на отсутствие ошибок расчетов или преобразований.

Обычно, условие состоит из сравнений и отношений, объединённых логическими операциями. В условиях могут присутствовать и другие операции или выражения, но итоговым результатом условия все равно должны быть «ложь» или «истина». Больше примеров условий будет приведено в последующих разделах урока.

Еще несколько слов о различной терминологии.

- Поскольку условное выражение имеет значение (истина или ложь), его можно назвать «условной операцией» (помним, операциями называют выражения, которые приводят к некоторому результату).
- В отдельных языках программирования (например, в языке PROLOG), а также в математической логике применяется термин «предикат». В различных источниках можно увидеть варианты «предикатное выражение» или «предикативное выражение».
- В память об одном из основателей математической логики — о Джордже Буле (*англ. George Boole*) условия также называют «булевыми выражениями». Иногда обобщенно — «логическими выражениями».

Все эти термины являются синонимами и могут применяться различными авторами учебников или статей для обозначения одних и тех же программных конструкций — условий.

If

Основным средством, позволяющим обработать условие, то есть выполнить некоторую часть программы в зависимости от результата условного выражения, является оператор «**if**». Его называют условным оператором или оператором ветвления, поскольку в результате проверки некоторого условия этот оператор обеспечит дальнейшее выполнение программы по тому или иному пути.

Синтаксис оператора выглядит следующим образом. После слова «**if**» в круглых скобках записывается условие

и, после скобок, — выражение (инструкция), которое нужно выполнить по результатам проверки условия.

```
if(condition)
    statement;
```

Выражение будет обработано и выполнено лишь в том случае, если условие приведет к результату «**true**», то есть если условие «выполняется» (истинно).

Если после проверки условия необходимо выполнить несколько инструкций, то их объединяют группирующим оператором «**{}**» (фигурные скобки)

```
if(condition) {
    statement1;
    statement2;
    statement3;
}
```

Для того чтобы проверить, как работает условный оператор, введите в консоль браузера выражение

```
if(1>0) alert("Yes")
```

В нем в качестве условия используется отношение «**1>0**», а инструкцией является рассмотренное нами ранее диалоговое окно-сообщение с текстом «**Yes**». Поскольку условие изначально истинно, команда «**alert("Yes")**» должна быть выполнена. Нажмите «**Enter**» и убедитесь в появлении сообщения. После закрытия этого сообщения в консоли появится надпись «**undefined**». Вспомните, что это означает?

Поменяйте условное выражение на другое, используя рассмотренные в предыдущих разделах примеры, и про-

верьте при каких случаях сообщение «Yes» появляется, а в каких нет.

Приведем несколько практических примеров, базирующихся на условном операторе:

Задание: в переменной «`x`» хранится разница вчерашнего и сегодняшнего курса валюты. Нас интересует только величина отклонения, то есть если значение отрицательное, то нужно убрать знак. Другими словами, нужно определить модуль числа «`x`».

Решение: при помощи условного оператора это сделать проще и быстрее, чем через библиотеку математических функций. Если число «`x`» меньше нуля, применим оператора смены знака «`-x`» и результат сохраним в той же переменной (`x = -x`):

```
if (x<0)
    x = -x;
```

Задание: необходимо проверить, была ли переменная «`x`» определена ранее в программе. Если нет, то нужно присвоить ей значение «`0`».

Решение: мы уже упоминали способ проверки на то, что переменная не была определена (см. раздел «Оператор `typeof`»), поместим его в условный оператор:

```
if(typeof x =='undefined')
    x = 0;
```

Задание: в переменной «`x`» хранится сумма покупки. Если она превышает `1000`, уменьшить ее на скидку в `10%`.

Решение: уменьшение на 10% эквивалентно вычитанию из числа одной десятой его части. То есть «**x**» уменьшенный на 10% это «**x – x/10**» или «**x – 0.9x**». После вычитания, результат нужно сохранить в той же переменной. С учетом этого запишем:

```
if(x>1000)
    x = x - 0.9*x
```

If else

В случае если по итогам проверки условия нужно выполнить два разных блока команд, применяется расширенная (или полная) форма условного оператора, содержащая две секции инструкций. Записывается полная форма условного оператора следующим образом:

```
if(condition)
    statement_if_true;
else
    statement_if_false;
```

Если результат проверки условия «**condition**» будет истинным, выполнится первое выражение «**statement_if_true**». В противном случае — второе «**statement_if_false**». Выделяя термин «полная» форма условного оператора отметим, что рассмотренная в предыдущем разделе форма называется сокращенной.

Полностью аналогично сокращенной форме, для выполнения нескольких выражений в полной форме условного оператора необходимо применить к ним групп-

пирующий оператор «{}». Для лучшей читаемости программы при использовании группировки рекомендуется добавлять одинаковые отступы слева перед сгруппированными выражениями.

```
if(condition) {
    statement_if_true1;
    statement_if_true2;
    statement_if_true3;
}

else {
    statement_if_false1;
    statement_if_false2;
    statement_if_false3;
}
```

Приведем несколько практических примеров использования полной формы условного оператора:

Задание: в переменной «`x`» хранится некоторое число. Необходимо проверить его четность и результат сохранить в переменной «`parity`»

Решение: четное число делится на `2` нацело. То есть имеет нулевой остаток от деления на `2`. В формализме JavaScript остаток от деления вычисляется оператором «`%`». Условие на четное значение будет выглядеть как «`x % 2 == 0`». В противном случае, число является нечетным

```
if(x % 2 == 0)
    parity = "even";
else
    parity = "odd";
```

Условные операторы могут быть вложенными один в другой. То есть в любом из условных блоков могут быть использованы новые условные операторы со своими блоками. В них также могут быть условные операторы — ограничений на степень вложенности нет. Это позволяет проверять условия одно за другим, обрабатывая на каждое из них отдельно.

В качестве примера рассмотрим следующую задачу. Банковское округление (англ. *banker's rounding*) обозначает округление к ближайшему чётному числу. То есть 2.7 округляется до 2, а 3.2 до 4 (хотя при обычном округлении оба значения привели бы к результату 3). В переменной «`x`» хранится дробное число, нужно написать программу, реализующую банковское округление этого числа и сохраняющую результат в переменной «`bx`».

Решение. Во-первых, попробуем использовать обычное округление `Math.round(x)` и проверим, может мы сразу получим четное число, тогда его нужно просто сохранить в переменной «`bx`»:

```
if( Math.round(x) % 2 == 0 )
    bx = Math.round(x)
```

Если же результат обычного округления нечетен, то у нас есть два возможных варианта: а) число было округлено вверх, например 2,7 до тройки или б) число было округлено вниз, как 3,2 к той же тройке. В первом случае от результата округления надо вычесть единицу, получив меньшее из четных чисел (2,7 округляется к двойке). Во втором варианте единицу к результату нужно добавить

(3,2 округляется к четверке). Это можно реализовать вторым условным оператором, вложенным в блок «`else`» первого оператора. Итоговый код, решающий задачу, примет следующий вид:

```
if( Math.round(x) % 2 == 0 )
    bx = Math.round(x)
else {
    if( x < Math.round(x) )
        bx = Math.round(x) - 1
    else
        bx = Math.round(x) + 1
}
```

Благодаря условному оператору мы формируем множество путей работы программы, то есть задаем ее поведение, реакцию на изменяющееся окружение или действия пользователя. При сохранении постоянства кода программы, реальный ход его выполнения может значительно отличаться для разных ситуаций из-за того, что ветвления на каждом из условий приведут к различному набору действий. Начинаясь с одной и той же первой строки кода, дальнейший путь и место окончания заранее непредсказуемы и определяются в процессе работы программы.

Отдельно обратим внимание на то, что в JavaScript, за счет динамического преобразования типов, в качестве аргумента для оператора «`if`» может использоваться не только условное выражение, но и любая другая операция (например, арифметическая). В таком случае ее результат будет автоматически преобразовываться к типу «`Boolean`» и анализироваться итог такого преобразования. К «ложному» результату преобразуются значения «`0`»

(числовой ноль), «`null`», «`undefined`» или пустая строка. Другие результаты считаются истинными.

Например, в рассмотренной выше проверке на четность используется условие `(x % 2 == 0)`. Так как остаток от деления на 2 может быть только 1 или 0, можно воспользоваться динамическим преобразованием типов, записав условный оператор как `«if(x % 2)»` — без сравнения с нулем. Только следует отметить, что при значении «`0`» будет выполнен блок `«else»`, тогда как полное сравнение `(x % 2 == 0)` в этом же случае выполняет основной блок. То есть по сравнению с предыдущим примером в сокращенном варианте условные блоки нужно поменять местами.

```
if(x % 2)
    parity = "odd";
else
    parity = "even";
```

Тернарный оператор ?

Довольно часто в программировании возникают ситуации, когда по результату проверки какого-то условия в некоторую переменную нужно поместить то или иное значение. Снова вернемся к примеру проверки числа `«x»` на четность, в результате которой переменная `«parity»` принимала одно из значений: `«even»` или `«odd»`.

По сути, в переменную `«parity»` присваивается одно из двух значений в зависимости от условия проверки на четность переменной `«x»`. Читаемость программы улучшила бы конструкция, которая начиналась с записи `«parity=»`, после чего следовало бы выражение выбора значения.

Для реализации такой возможности был разработан тернарный оператор «?:». С его помощью приведенный блок команд можно записать как:

```
parity = x % 2 == 0 ? "even" : "odd"
```

Для лучшей разделимости составных частей тернарного оператора, условие берут в круглые скобки, подобно тому, как это делается в условном операторе. Согласитесь, следующее выражение выглядит более понятным, чем предыдущее:

```
parity = (x % 2 == 0) ? "even" : "odd"
```

Выражение начинается с «**parity**=». Это сразу подсказывает нам, что результатом операции будет присвоение переменной «**parity**» какого-то значения. Далее указывается условие «**x % 2 == 0**», что говорит о присваивании в зависимости от результата условия. Затем следуют два значения, разделенными символами «?» и «:», составляющими суть тернарного оператора. После символа «?» следует значение, которое будет возвращено при истинном результате условия; после символа «:» — при ложном результате.

Вместо конкретных значений («**even**» и «**odd**») возможно использовать выражения, приводящие к нужным результатам (операции). Правда, при этом читаемость тернарной инструкции может быть хуже, чем у аналогичного условного оператора, особенно при высокой сложности выражений.

Возможности тернарного оператора в JavaScript пре-восходят простое разделение значений при присваива-

нии. В состав оператора можно включать несколько операций, разделяя их запятыми. Однако это значительно ухудшает читаемость кода и крайне нежелательно к использованию на практике. Здесь мы эти возможности раскрывать не будем, следуя рекомендациям применять тернарный оператор, вместо условного, только для улучшения итоговой читаемости кода.

Switch

Ограниченный выбор одного из двух путей, реализуемый условным оператором, делает его неудобным для обработки множественных условий. С одной стороны, существует возможность вкладывать условные операторы один в другой, перебирая все варианты ветвления один за другим. С другой стороны, хотелось бы иметь возможность упростить подобный перебор.

Приведем простой пример. В программе мы получаем аббревиатуру протокола (HTTP, HTTPS, FTP) и хотим сформировать ее расшифровку, а также предусмотреть случай несовпадения полученного протокола ни с одним из этих значений. С применением вложенного ветвления это можно сделать следующим образом:

```
if(protocol == "HTTP") description =
    "Hypertext transfer protocol";
else if(protocol == "HTTPS") description =
    "Secure hypertext transfer protocol";
else if(protocol == "FTP") description =
    "File transfer protocol";
else description = "Unsupported protocol";
```

Очевидно, что с увеличением множества возможных значений протоколов, вложенность условий также возрастает, ухудшая читаемость кода и удобство его оформления в целом.

Для того чтобы упростить подобный множественный анализ применяется оператор «**switch**». При помощи него поставленная выше задача может быть решена как:

```
switch(protocol) {  
    case "HTTP":  
        description = "Hypertext transfer protocol";  
        break;  
    case "HTTPS":  
        description = "Secure hypertext  
                        transfer protocol";  
        break;  
    case "FTP":  
        description = "File transfer protocol";  
        break;  
    default :  
        description = "Unsupported protocol";  
}
```

Этот код читается гораздо лучше, да и выглядит понятнее, чем приведенный ранее вариант с вложенными условными операторами.

В круглых скобках после оператора «**switch**» указывается переменная или выражение, которое должно быть проверено на соответствие некоторому значению. Далее следуют фигурные скобки, в которых каждое из условий записывается после ключевого слова «**case**». Оператор сравнения «**==**» при этом опускается, приводится только само значение для сравнения.

Отдельным блоком выступает секция «`default`», которая будет выполнена, если ни одно из перечисленных в «`case`» выражений не будет обработано. Его можно сравнить с неким итоговым «`else`» по аналогии с условным оператором.

Исторически сложилось так, что оператор «`switch`» выполнял все инструкции, следующие далее за подходящим «`case`». Если значение в определенном «`case`» будет равно переменной из оператора «`switch`», то все нижеследующие команды данного блока, а затем все остальные команды всех последующих блоков «`case`», в том числе и команды блока «`default`», будут выполнены один за другим в порядке описания. В JavaScript сохранен это формализм и следует об этом помнить при его использовании.

С одной стороны, это позволяет не группировать команды при помощи фигурных скобок, сокращая разметку. С другой стороны, требует применение инструкции прерывания «`break`», если дальнейшее выполнение кода не должно быть реализовано. Повторимся — это дань традициям, и практически во всех языках программирования оператор «`switch`» работает подобным образом.

Приведем пример, иллюстрирующий описанное поведение оператора «`switch`». Пусть задача состоит в том, чтобы сформировать строку, начиная с введенного пользователем значения переменной «`x`» и заканчивая значением «`5`». Добавим ограничение, что значение «`x`» не должно быть больше 5 и меньше 1.

Данную задачу можно решить следующим способом. Для ее решения создайте новый HTML документ, наберите или скопируйте в него следующее содержимое (код также доступен в папке *Sources* — файл `js1_5.html`)

```
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <script>
            x = +prompt("Input x=");
            var str = "";
            switch(x) {
                case 1: str += "1";
                case 2: str += "2";
                case 3: str += "3";
                case 4: str += "4";
                default: str += "5";
            }
            alert(str);
        </script>
    </body>
</html>
```

Поясним, как работает данная программа. Вначале у пользователя запрашивается первое значение для задачи при помощи диалогового окна «`prompt`». Как нам известно, значение из окна «`prompt`» будет передано в строковом типе (см. предыдущий раздел), поэтому мы применим его преобразование к числовому типу при помощи унарного оператора «`+`». Полученный результат сохраняем в переменной «`x`» (`x = +prompt("Input x")`).

Далее создается переменная «`str`» и ей присваивается пустая строка. В операторе «`switch`» проводится анализ переменной «`x`». Если значение переменной равно `1`, то срабатывает самый первый «`case`» и в строку «`str`» дописывается символ «`1`» при помощи операции добавления «`+=`».

Согласно принципу работы оператора «`switch`», далее будут выполнены все блоки команд после первого успешного блока «`case`», то есть к строке «`str`» будут последовательно дописаны символы от «`2`» до «`5`» за счет выполнения всех дальнейших команд из других блоков.

Если значение «`x`» равно `2`, то совпадение будет найдено во втором «`case`» и, по аналогии, в строку будут последовательно собраны все остальные символы за счет команд последующих блоков. Команда из первого блока «`case`» при этом выполнена не будет. Выполняются только те команды, которые описаны ниже подходящего блока «`case`».

В конце программы сформированная строка выводится при помощи окна-сообщения «`alert`».

Сохраните файл, откройте его при помощи браузера. В появившемся диалоговом окне впишите число от `1` до `5` и нажмите «`OK`». Убедитесь в появлении нового сообщения с числовой последовательностью.

Обратите внимание, что введенные пользователем данные мы преобразовывали к числу. Это необходимо потому, что в операторе «`switch`» используется только строгое сравнение «`==`» (см. раздел 13). Для того чтобы в этом убедиться, удалите оператор «`+`» перед «`prompt`», сохраните файл и обновите страницу в браузере. При любом вводе в первом диалоговом окне результатом работы будет «`5`» за счет выполнения блока «`default`», т.к. сравнение числовых данных, указанных в «`case`», и строкового результата диалога «`prompt`» всегда будет приводить к ложному результату.

Задача для самостоятельного решения: пользователь вводит число «`x`». Считаем, что пользовательский

ввод ограничен значениями от 1 до 10. Программа выводит числа от «**x**» до 5, если $x \leq 5$, иначе — числа от «**x**» до 10.

Найдите ошибку: программа должна запросить у пользователя число от 1 до 6 и вывести сообщение о четности числа (**odd** — нечетное, **even** — четное), либо "Out of range", если число выходит за указанный диапазон

```
<script>
    x = +prompt("Input number from 1 to 6");
    switch(x) {
        case 2:
        case 4:
        case 6: alert("even");
            break;
        case 1:
        case 3:
        case 5: alert("odd");
        default: alert("Out of range");
    }
</script>
```

(Перед вариантом «**default**» пропущен оператор **«break»**. Если будет введено нечетное число, то выведутся оба окна — **alert("odd")** и **alert("Out of range")**, т.к. в операторе **«switch»** выполняются все инструкции, указанные ниже подходящего условия).

Задание для самостоятельной работы

1. Напишите скрипт, который запрашивает у пользователя подтверждение некоторого действия (используя диалог `confirm`) и после его ответа выводит сообщение «Подтверждено» или «Отменено».
2. Напишите скрипт, который запрашивает у пользователя пароль подтверждения некоторого действия. Допускается три возможных пароля («`Step`», «`Web`» и «`JavaScript`»). После ввода пароля скрипт должен вывести сообщение «Подтверждено» или «Отменено».
3. Напишите скрипт, который запрашивает у пользователя число «`x`», проверяет его на принадлежность диапазону `0..100` и выводит соответствующее сообщение (например, `10` — принадлежит, `-10` — не принадлежит, `0` — принадлежит, `200` — не принадлежит).
4. Напишите скрипт, который запрашивает у пользователя два числа «`x`» и «`y`», сравнивает их величины и выводит одно из сообщений: «`x > y`», «`x < y`» или «`x=y`» в зависимости от введенных данных.
5. Напишите скрипт, который запрашивает у пользователя число «`x`», «ранжирует» его по диапазонам `0..100`, `101..200`, `201..300` и выводит сообщение о принадлежности или несоответствии ни одному из них (например, `30` принадлежит диапазону `0..100`; `250` — диапазону `201..300`; `-10` или `500` — ни одному).
6. Напишите скрипт, который запрашивает у пользователя цифру и выводит ее название: `0` — «ноль», `1` — «единица», `2` — «двойка» и т.д. Если переменная не является цифрой, выводится сообщение «не цифра».

Циклы

Циклы

Отвлечемся от программирования и представим, что нам надо приготовить обед, для чего необходимо начистить некоторое количество картошки. Формально, для того чтобы это сделать нужно повторить однообразное действие «почистить картошку» для нескольких разных (или одинаковых) картофелин.

Разберем процесс чистки более детально. Во-первых, необходимо взять картофелину, осмотреть, удалить «глазки» при их наличии. Во-вторых, очистить ее от кожуры. В-третьих, промыть и сложить в емкость для чищенной картошки. Затем эти действия должны быть повторены в той же последовательности для других картофелин. Обобщая, получаем циклическое повторение набора действий для исходного набора картошки.

Следующая особенность циклического процесса «почистить картошку» касается вопроса, когда же следует остановиться, прекратить процесс, ведь действия, которые повторяются, ничем не ограничены. Ответ на этот вопрос вытекает из деталей того задания, которое перед нами было поставлено изначально. А это задание могло быть поставлено по-разному: 1) «начистить 5 картофелин», 2) «начистить всё, что лежит в пакете» или 3) «начистить сколько получится за 10 минут».

В первом случае нам заранее известно количество картофелин, то есть мы знаем сколько раз нужно повторить действие, нужно просто подсчитывать количество почищенных картофелин (количество повторенных дей-

ствий). Во втором случае количество заранее неизвестно, но получив пакет, мы можем пересчитать его содержимое и действовать аналогично первому случаю, либо руководствоваться другим условием — повторять действия, пока не опустеет пакет. Третий вариант практически ничего не сообщает нам о количестве повторов, оно будет зависеть от множества факторов, как то размеры картофелин, наша скорость и опытность по их чистке и т.п. Однако условие остановки нам все же известно, только зависит оно от другого параметра — прошедшего времени.

Задачи подобного характера довольно часто возникают и при построении сценариев, когда требуется повторное выполнение одних и тех же команд для разных (а иногда и одинаковых) данных. Представим, что перед программистом поставили задачу разработать кредитный калькулятор, и рассмотрим, к каким повторным действиям нам придется прибегать. В простейшем случае предполагается, что кредит берется на год и погашается равномерными платежами каждый месяц. Например, 1000 у.е. было взято под 20% годовых, что предусматривает возврат суммы в 1200 у.е. Значит, в каждый из 12 месяцев года платеж по кредиту будет составлять 100 у.е. Программа должна вывести график погашения кредита:

Месяц 1 — платеж 100 у.е.

Месяц 2 — платеж 100 у.е.

...

Месяц 12 — платеж 100 у.е.

Данный пример можно соотнести с чисткой известного количества картошки, т.к. нам заранее известно сколько раз нужно вывести надпись на экран. С одной

стороны, это позволяет написать 12 одинаковых инструкций по выводу данных, только в каждой из них указать свой номер месяца. С другой стороны, такой подход кажется неоптимальным, особенно, если его обобщать и на большее количество повторений.

Вторая функция калькулятора будет допускать, что срок кредита пользователь вводит самостоятельно. Программа все так же должна вывести график погашения кредита, но уже не на 12 месяцев, а на указанный срок.

В такой формулировке задача напоминает «начистить пакет картошки». До того, как мы начали выполнять задачу количество повторных действий неизвестно. Обратите внимание: задача сформулирована, а количество не определено. Однако когда мы приступим к выполнению задания (и получим пакет), количество будет установлено. Также и в калькуляторе — расчет начинается после пользовательского ввода, а значит, к началу счета срок будет уже определен.

Тем не менее, программу мы пишем до того, как пользователь введет цифры, и она должна быть готова обработать любые введенные данные. Идея написать несколько инструкций вывода подряд приобретает еще больше недостатков, так как мы не можем знать, сколько инструкций использовать в программе. Можно, конечно, написать инструкции «с запасом» и каждую из них поместить в условный оператор

```
...
if(term >= 12) console.log("Month 12 – sum 100");
if(term >= 13) console.log("Month 13 – sum 100");
if(term >= 14) console.log("Month 14 – sum 100");
...
```

Однако, даже при наличии хорошего «запаса» все равно такая программа будет иметь предел, после которого расчет будет невозможен. С одной стороны, никто не будет давать кредит на 100 лет, то есть рассчитать «запас» все же можно. С другой стороны, такое решение выглядит совсем непривлекательно.

Дополним наш кредитный калькулятор еще одной функцией — пользователь вводит сумму, которую он готов выплачивать ежемесячно, а программа должна рассчитать срок кредитования исходя из этой суммы.

В таком случае срок кредитования нам неизвестен даже после ввода данных пользователем. Программа должна моделировать процесс: каждый месяц задолженность возрастает на кредитную ставку и уменьшается на ежемесячный платеж. Тот месяц, в котором задолженность будет погашена, станет последним и позволит рассчитать срок кредитования. По степени определенности данных имеем аналогию с чисткой картошки в течение 10 минут — количество будет известно только по прошествии заданного времени и не может быть определено до его окончания.

При таком алгоритме действий составить программу в виде последовательных инструкций будет совсем не-приемлемо (хотя и принципиально возможно). Каждая следующая команда должна сопровождаться рядом проверок, по результатам которых следует либо продолжать моделирование, либо остановиться и вывести итоги. И таких команд нужно предусмотреть на все допустимые сроки кредитования.

При наличии принципиальных отличий рассмотренных вариантов задачи между собой, во всех них можно

выделить общую черту — необходимость повторять одинаковые действия (один и тот же блок команд). Разница будет заключаться лишь в том, когда повторение следует прекратить: после 12 месяцев, после введенного срока или после обнуления задолженности.

Что такое цикл?

Программные механизмы, позволяющие несколько раз выполнить один и тот же блок кода, называют циклами. При работе цикла программа снова и снова «возвращается» на начало блока команд, выполняя его заново, инструкция за инструкцией, до того, как цикл будет прекращен (остановлен).

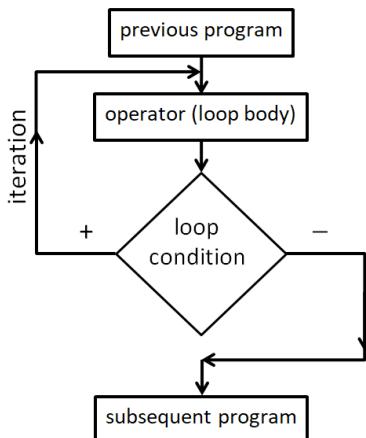


Рисунок 29

Повторяемый блок кода носит название «тело цикла» (*loop body*), а процесс его выполнения — итерация (*iteration*). Фраза «на второй итерации» означает «при втором повторе тела цикла». Обычно, повтор (*итерация*) цикла

происходит в зависимости от некоторого выражения, называемого цикловым условием (*loop condition*). Если это условие истинно, цикл будет повторяться, если ложно — прекратится. Схематически, работу цикла иллюстрирует рисунок 29.

Если циклового условия нет, повторения цикла не прекращаются, и получается «бесконечный цикл». Такой цикл выполняется в течение всей работы сценария и прерывается только после его завершения. Бесконечные циклы могут применяться для постоянного мониторинга каких-либо параметров (например, сетевой активности), поддержания анимации или пользовательского ввода в играх. С негативной стороны, бесконечный цикл может получиться в результате ошибок программиста. В таком случае возможно «зависание» сценария, потеря его управляемости. Скорее всего, при этом придется закрыть вкладку браузера, исправить ошибки и заново открыть документ.

Если в цикле необходим подсчет номера итерации, то для этого вводится отдельная переменная, меняющая свое значение при каждом повторе цикла. Эта переменная-счетчик носит название «цикловая переменная» (англ. *cycle index* или *counter*). Обычно, она принимает значение равное номеру повтора цикла 0-1-2-3... или 1-2-3-4..., хотя бывают задачи, когда удобнее использовать другие множества значений, например, только четные числа. В таком случае, величина, на которую изменяется цикловая переменная, называют «шагом» (*step*).

По способу организации повторного выполнения кода, циклы делят на циклы-счетчики и условные ци-

клы. Последние дополнительно подразделяются на циклы с предусловием и циклы с постусловием.

Циклы-счетчики (или циклы со счетчиком) выполняются заданное количество раз. Количество повторов может задаваться как числом, так и значением переменной, но это значение должно быть известно до начала цикла. Задача по расчету кредитных платежей на конкретный срок является типичным примером использования циклов данного вида. Даже если изначально срок кредита неизвестен, в момент расчета платежей это число уже будет введено пользователем, и количество повторов цикла будет известным.

Условные циклы (или циклы с условием) выполняются до тех пор, пока некоторое условие является истинным. В некоторых языках программирования (JavaScript к ним не относится) есть разновидности циклов, выполняющиеся пока условие ложно. При помощи таких циклов, например, может быть решена задача по расчету кредита исходя из размера ежемесячного платежа. Не зная перед количеством повторов цикла, нам всё же известно условие, по которому цикл должен быть прерван.

Если цикловое условие проверяется до перехода к телу цикла, то такие циклы называют «циклы с предусловием», если после выполнения тела — «циклы с постусловием». Цикл с предусловием может не выполниться ни разу, если условие изначально ложно. Цикл с постусловием будет выполнен, по крайней мере, один раз до первой проверки условия. В остальном, эти циклы подобны и часто могут быть заменены один на другой. В некоторых языках программирования (например, в Python) есть только один из этих циклов.

Задание. Программа «угадай число» загадывает некоторое число, а пользователь пытается его угадать, вводя свои предположения с клавиатуры. В случае если предложенный вариант не совпал с загаданным, пользователю предоставляется следующая попытка, и так до тех пор, пока число не будет угадано. Определите, какой тип цикла будет предпочтительным для программы «угадай число». (Пользователь должен ввести свой вариант числа хотя бы один раз — нельзя угадать число, не введя предположения. Поэтому предпочтительным будет цикл с постусловием.)

While

Цикл с предусловием создается при помощи оператора «**while**». Общий синтаксис оператора довольно прост:

```
while(condition)
    statement;
```

За ключевым словом «**while**» в круглых скобках указывается условное выражение, после скобок — инструкция для выполнения (тело цикла). Цикл обеспечивает повторное выполнение тела до тех пор, пока условие остается истинным. Как обычно, в случае необходимости включения в тело цикла нескольких языковых инструкций, применяется группирующий оператор «{}».

```
while(condition) {
    statement1;
    statement2;
    statement3;
}
```

Для того чтобы цикл имел окончание, в теле цикла должны быть предусмотрены инструкции, оказывающие некоторое влияние на условие повторения цикла. В противном случае, цикл либо не выполнится ни разу (при ложном условии), либо будет повторяться бесконечно (при истинном). Следует обратить внимание на то, что это довольно популярная ошибка даже у опытных программистов — забыть включить в тело цикла инструкции по управлению цикловым условием.

Далее приведены два примера, выводящие в консоль числа от 1 до 9, один правильный, другой — содержащий ошибку. Причем, ошибка эта не синтаксическая, фрагмент может быть выполнен. В теле цикла отсутствует инструкция «`i++`», влияющая на цикловое условие «`i < 10`», из-за чего цикл становится бесконечным — переменная «`i`» всегда равна 1, то есть всегда меньше 10.

Правильно	Неправильно
<pre>i = 1; while(i < 10){ console.log(i); i++; }</pre>	<pre>i = 1; while(i < 10){ console.log(i); }</pre>

Цикл «`while`» удобно применять в случаях, когда

- заранее неизвестно количество повторов тела цикла,
- условие окончания цикла прямо не зависит от цикловой переменной,
- цикл зависит от входящих данных, например, от действия пользователя.

В качестве примера рассмотрим реализацию кредитного калькулятора, требующую условного цикла. Пусть

величина ежемесячного платежа хранится в переменной «`monthlyPayment`», сумма кредита — в переменной «`creditAmount`», кредитная ставка — в переменной «`creditRate`». Дополнительно в переменной «`month`» будем хранить количество моделированных месяцев. Тогда программный фрагмент будет выглядеть следующим образом:

```
month = 0;
while(creditAmount > 0){
    creditAmount *= creditRate;
    creditAmount -= monthlyPayment;
    month++;
}
```

Изначально устанавливаем счетчик месяцев в ноль. Запускаем цикл с условием (`creditAmount > 0`), то есть пока сумма кредита не погашена. В теле цикла увеличиваем остаток кредита на величину ставки (`creditAmount *= creditRate`) и тут же уменьшаем ее на ежемесячный платеж (`creditAmount -= monthlyPayment`). Затем увеличиваем на 1 счетчик месяцев.

По окончанию цикла в переменной «`month`» будет храниться срок кредита (количество месяцев) до его окончательного погашения.

Рассмотрим еще одну задачу: мы хотим накопить 1 миллион у.е., получая проценты по банковскому вкладу (депозиту). Каждый год на наш вклад начисляется 10% и эти начисления добавляются к основному вкладу. То есть при вкладе 1000 у.е. в конце первого года будет начислено 100 у.е. и они добавятся к вкладу, увеличив его до 1100 у.е. В конце второго года 10% уже будет на-

числено на текущую сумму и составит 110 у.е., увеличив в итоге вклад до 1210 у.е. Нас интересует, за сколько лет наш вклад достигнет 1 миллиона у.е.

Задача подобна предыдущей, только сумма вклада не уменьшается, а увеличивается, и условие выхода заключается не в обнулении суммы, а достижения заданного предела (1 млн.). Соответственно, для данной задачи также предпочтительно использовать условный цикл.

Оформим решение задачи в виде отдельного самостоятельного документа. Создайте новый файл, наберите или скопируйте в него следующее содержание (*код также доступен в папке Sources — файл js1_6.html*)

```
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <script>
            var fund = +prompt("Initial deposit:");
            var years = 0;
            while(fund < 1e6) {
                years++;
                fund += fund * 0.1; // 0.1 = 10%
            }
            alert("You'll become millionaire after" +
                  years + " years");
        </script>
    </body>
</html>
```

Для ввода данных воспользуемся диалоговым окном **«prompt»**. Поскольку начальная сумма вклада должна быть величиной числового типа, применим оператор **«+»**

перед результатом ввода и сохраним его в переменной «`fund`». Введем переменную-счетчик «`years`» инициализировав ее значением `0`. Далее циклично увеличиваем «`years`» на `1` (`years++`), а вклад — на `10%` (`fund += fund * 0.1`). Условием повторения цикла будет выражение, сравнивающее накопленную сумму с миллионом. Обратите внимание, для удобства подсчета нулей, а также для сокращения записи «`1000000`» использована экспоненциальная форма представления числа миллион — «`1e6`».

По окончанию цикла выводим окно-сообщение «`alert`» с данными о количестве лет, необходимых для выполнения условия задачи.

Сохраните файл, откройте его при помощи браузера. В появившемся диалоговом окне введите сумму первого вклада, нажмите «`OK`». Результат работы скрипта должен выглядеть как сообщение о протяженности пути к миллиону.

Найдите ошибку: программа должна запросить у пользователя три числа и вывести их в консоль браузера

```
<script>
    var limit = 3;
    var numbers = 0, number;
    while(numbers < limit) {
        number = +prompt("Type number:");
        console.log(number);
    }
</script>
```

(В цикле пропущены инструкции, меняющие цикловое условие, и цикл становится бесконечным. В теле нужно дописать «`numbers++`».)

Do while

Вторая разновидность условных циклов, — цикл с постусловием, — создается при помощи оператора «**do while**» по следующей схеме:

```
do
    statement
while(condition)
```

Тело цикла (*выражение statement*) выполняется повторно до тех пор, пока условие (*condition*) остается истинным. Несмотря на то, что операторные рамки «**do**» и «**while**» позволяют однозначно определить начало и конец тела цикла, при использовании нескольких инструкций в цикле их все равно нужно объединять группирующим оператором «**{}**».

```
do {
    statement1;
    statement2;
    statement3;
} while(condition)
```

В отличие от цикла «**while**», цикл с постусловием при ложном условии выполнится один раз, поскольку условие повтора проверяется после выполнения тела. По аналогии с предыдущим циклом следует помнить о необходимости влияния на условие повторения во избежание «**зацикливания**» — бесконечной работы цикла из-за неизменности циклового условия. Циклы с постусловием удобно применять как раз в тех случаях, когда тело цикла должно быть выполнено, по крайней мере, один раз.

Например, мы хотим получить два случайных числа, но не равных между собой. То есть при совпадении двух чисел нам нужно повторить запрос случайного числа (сопадение часто бывает, когда количество чисел небольшое, например, при играх с кубиком). Если после повтора снова обнаруживается равенство, нужно обеспечить еще один повтор, и так далее.

Первое число получаем обычным образом «`x1=Math.random()`». Второе число надо рассчитать как минимум один раз, а при совпадении с первым — рассчитывать повторно. Для этого как нельзя лучше подходит цикл с постусловием: «`do x2= Math.random(); while(x2==x1)`». Если второе число отличается от первого, то никаких повторов не будет. Если же совпадет — сработает условие цикла и расчет будет повторен.

```
x1=Math.random();
do
    x2= Math.random();
while(x2==x1)
```

Другой пример: у пользователя нужно получить подтверждение некоторого действия. Мы не хотим использовать диалог «`confirm`», т.к. в нем можно случайно нажать пробел, «`Enter`» или «`Esc`». Нам нужно уверенное подтверждение, то есть пользователь должен ввести либо «`yes`», либо «`no`». В ином случае мы будем выводить запрос повторно, ожидая одного из двух ответов. Так как запрос нужно выводить как минимум один раз, цикл с постусловием будет предпочтительным:

```
<script>
    var txt;
    do
        txt = prompt("Confirm: yes or no")
    while(txt!="yes" && txt!="no")
</script>
```

В остальном же особенности, замечания и предостережения по применению условных циклов одинаковы, независимо от месторасположения циклового условия.

For

Цикл-счетчик (или цикл со счетчиком) организуется при помощи оператора «**for**». Синтаксис оператора, кроме тела цикла, содержит три декларативных блока:

```
for (initialization; condition; expression)
    statement
```

Первый блок «**initialization**» используется для начальной инициализации цикловой переменной. Этот блок выполняется один раз перед началом цикла. Второй блок «**condition**» содержит условие, при котором цикл продолжается. Обычно это условие содержит ограничение на цикловую переменную. Третий блок «**expression**» задает выражение для изменения цикловой переменной. Второй и третий блоки выполняются на каждой итерации цикла.

Телом цикла является одна языковая инструкция «**statement**», которой может быть группирующий оператор «**{}**», содержащий несколько собственных инструкций.

Типичным примером записи оператора «`for`» является следующий:

```
for(i=0; i<5; i++)
    console.log(i);
```

В первом блоке инициализируется цикловая переменная «`i=0`», во втором указывается условие на эту переменную «`i<5`», в третьем — алгоритм ее изменения «`i++`» (увеличение на `1`). В данном примере телом цикла является команда вывода значения цикловой переменной в консоль браузера. Традиционно цикловую переменную называют «`i`» (*от англ. iteration — итерация*), но это не является обязательным и для нее может быть выбрано произвольное имя.

Наберите или скопируйте приведенный выше фрагмент с описанием цикла в консоль и нажмите «`Enter`». Как результат работы цикла в консоли должны появиться цифры от нуля до четырех.



Рисунок 30

Сам оператор «`for`» возвращает значение последней операции в теле цикла, поэтому в консоли появляется ответ «`undefined`», возвращенный командой «`console.log(4)`».

Работа цикла начинается с выполнения первого блока оператора `for «i=0»` и проверки условия во втором блоке `«i<5»`. Если условие истинно, выполняется тело цикла. После этого активируется третий блок `«i++»`, увеличивающий цикловую переменную, и снова проверяется условие `«i<5»`.

Согласно с описанным механизмом работы, цикл `«for»` является разновидностью циклов с предусловием (условие проверяется до выполнения тела). Чтобы в этом убедиться заменим «верхний предел» изменения переменной на ноль, т.е. введем изначально ложное условие.

Ведите в консоль тот же цикл, но с условием `«i<0»` (повтор команды в консоли обеспечивается нажатием «стрелки вверх» на клавиатуре). Нажмите `«Enter»` и обратите внимание на отсутствие результатов работы цикла (только ответ `«undefined»` от самого оператора `«for»`). Если бы условие проверялось после выполнения тела, мы бы наблюдали, по крайней мере, один результат его срабатывания, и в консоль был бы выведен ноль.

Содержание всех трех блоков оператора `«for»` не ограничивается описанными выше правилами и может быть практически произвольным. Прежде всего, эти блоки вообще не являются обязательными и могут быть оставлены пустыми (разделители блоков `«;»` при этом должны оставаться). Например, если цикловая переменная инициализирована до цикла, первый блок можно не указывать (все дальнейшие примеры можно скопировать в консоль и проверить на работоспособность)

```
i=0;  
for( ; i<5; i++)  
    console.log(i);
```

Аналогично, если цикловая переменная меняет значение в теле цикла, то последний блок также можно не указывать

```
i=0;  
for( ; i<5 ; ) {  
    console.log(i);  
    i++;  
}
```

Обратите внимание, что в результате выполнения последнего примера в консоли появится две цифры «4». Вторая цифра соответствует значению, которое возвращает сам оператор «**for**» — результат последней команды в теле цикла (**i++**). При этом надпись «**undefined**» из консоли исчезает.

Если в теле цикла содержатся механизмы собственной остановки, то и второй блок, отвечающий за цикловое условие, становится ненужным (подробнее об операторе «**break**» будет рассказано в следующем разделе)

```
i=0;  
for( ; ; ) {  
    console.log(i);  
    i++;  
    if(i>4)  
        break;  
}
```

С учетом того, что проверка (**i>4**) перенесена в конец тела, данная реализация цикла приобретает форму цикла с постусловием.

Все блоки оператора «**for**» могут содержать по нескольку операций, разделенных запятой. Например, следующий цикл содержит две переменные «**i**» и «**j**», одна увеличивается, начинаясь с **0**; другая уменьшается, стартуя с **10**. Цикл продолжается, пока переменные не равны между собой (**i!=j**). Другими словами, цикл прекращается, когда значения переменных оказываются равными друг другу.

```
for(i=0, j=10; i!=j; i++, j--)
    console.log(i, j)
```

Тело цикла содержит вывод в консоль обеих переменных. Результат выполнения такого цикла будет подобен следующему (см. рис. 31).



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console window displays the following code and its execution results:

```
> for(i=0, j=10; i!=j; i++, j--) console.log(i, j)
  0 10
  1 9
  2 8
  3 7
  4 6
< undefined
```

The code defines a for loop with initial values **i=0** and **j=10**. The condition for the loop is **i!=j**. Inside the loop, the code **console.log(i, j)** is executed, which prints the current values of **i** and **j** to the console. The loop continues until **i** equals **j**, at which point the loop exits and the value **< undefined** is printed.

Рисунок 31

Описанные манипуляции с блоками дают обширные возможности управления циклом «**for**». В частности, им можно заменить все рассмотренные ранее условные циклы. Однако при этом следует не забывать о читаемости кода и правилах «хорошего тона». Все-таки оператор «**for**» изначально предназначался для циклов-счетчиков и именно для этих целей его желательно использовать. В свою

очередь, условные циклы следует создавать специально для того введенными операторами «**while**» или «**do-while**».

В качестве примера рассмотрим реализацию оставшихся функций кредитного калькулятора. В простейшем случае мы рассчитываем ежемесячные платежи кредита, оформленного на год. Будем считать, что годовая ставка кредита составляет **20%** и она сразу начисляется на всю сумму, не меняясь в зависимости от ежемесячных платежей.

```
fund = +prompt("Credit sum");
creditBody = fund + 0.2*fund;
monthlyPayment = creditBody / 12;
for(i=1; i<=12; i++)
    console.log("month " + i +
                " payment "+monthlyPayment);
```

В первой строке кода мы запрашиваем у пользователя желаемую сумму кредита и сохраняем ее в переменной «**fund**». Используем преобразование введенных данных к числовому типу при помощи оператора «**+**», указанного перед вызовом диалогового окна «**prompt**».

Затем рассчитываем тело кредита «**creditBody**», добавляя к введенной сумме **20%**, что соответствует коэффициенту **0.2**, умноженному на введенную сумму. Затем вычисляем ежемесячный платеж «**monthlyPayment**» путем деления тела кредита на 12 месяцев.

Далее организовываем цикл. Поскольку срок кредитования нам задан условием задачи, используем цикл-счетчик «**for**», классически, цикловой переменной даем имя «**i**». Начальное значение устанавливаем равным 1 (первый месяц). Предельное значение — 12. С учетом того, что

само число 12 также должно быть обработано, используем нестрогое сравнение «`i<=12`». В теле цикла выводим в консоль номер месяца и сумму ежемесячного платежа.

Наберите или скопируйте в консоль браузера приведенный программный фрагмент, нажмите «`Enter`». Откроется диалоговое окно «`prompt`» в котором введите сумму, например, 1000. После ввода в консоли появится график платежей, подобный приведенному на рисунке 32.

```

> fund = +prompt("Credit sum");
creditBody = fund + 0.2*fund;
monthlyPayment = creditBody / 12;
for(i=1; i<=12; i++)
    console.log("month "+i+" payment "+monthlyPayment);
month 1 payment 100
month 2 payment 100
month 3 payment 100
month 4 payment 100
month 5 payment 100
month 6 payment 100
month 7 payment 100
month 8 payment 100
month 9 payment 100
month 10 payment 100
month 11 payment 100
month 12 payment 100
<- undefined
>

```

Рисунок 32

Как мы уже знаем, оператор цикла «`for`» возвращает результат последней операции, поэтому в конце вывода появляется «`undefined`».

Второй вариант калькулятора будет запрашивать еще и срок кредитования. Будем считать, что срок должен вводиться в месяцах. Тогда в программе появится дополнительная переменная «`months`», хранящая в себе

указанный срок. Аналогично сумме кредита она будет запрошена у пользователя диалоговым окном «[prompt](#)».

```
fund = +prompt("Credit sum");
months = +prompt("Credit term (months)");
creditBody = fund + 0.2 * fund * months / 12;
monthlyPayment = creditBody / months;
for(i=1; i<= months; i++)
    console.log("month " + i + " payment " + monthlyPayment);
```

В расчетной части программы константа «[12](#)» заменяется на введенный пользователем срок, а также пересчитывается процентная ставка: добавляется множитель «[month / 12](#)», определяющий количество лет кредита. Операции вывода остаются такими же.

Наберите или скопируйте в консоль браузера приведенный программный фрагмент, нажмите «[Enter](#)». Откроется диалоговое окно «[prompt](#)» в котором введите сумму, например, [1000](#). Появится второе диалоговое окно с запросом срока, введите [6](#). После ввода в консоли появится график платежей с учетом введенных данных (рис. 33).

The screenshot shows a browser's developer tools console tab labeled 'Console'. The code entered is:

```
> fund = +prompt("Credit sum");
> months = +prompt("Credit term (months)");
> creditBody = fund + 0.2 * fund * months / 12;
> monthlyPayment = creditBody / months;
> for(i=1; i<= months; i++)
>     console.log("month " + i + " payment " + monthlyPayment);
>
```

The output in the console is:

```
month 1 payment 183.333333333334
month 2 payment 183.333333333334
month 3 payment 183.333333333334
month 4 payment 183.333333333334
month 5 payment 183.333333333334
month 6 payment 183.333333333334
< undefined
>
```

Рисунок 33

Задание: используйте функцию округления для того, чтобы полученные числа отображались с двумя цифрами после запятой — в «денежном формате».

For in, for of

Наиболее популярное применение циклы-счетчики находят при работе с массивами или коллекциями для перебора их содержимого. Детальнее о таких программных конструкциях будет рассказано в следующем уроке, но для полноты рассмотрения оператора «**for**» приведем здесь его разновидности, адаптированные для указанных задач.

Циклы «**for in**» и «**for of**» позволяют перебирать свойства комплексных объектов (объединяющих в себе несколько других объектов), используя сокращенную запись оператора «**for**».

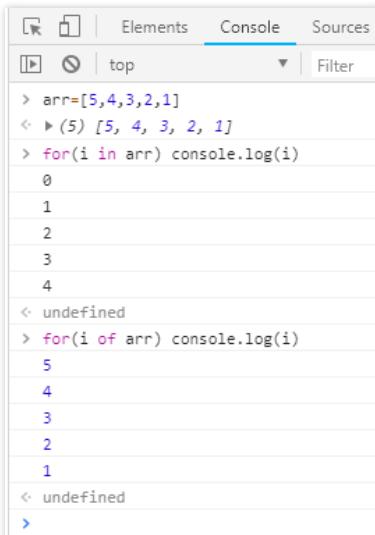
```
for (variable in object) statement
    for (variable of object) statement
```

Здесь «**variable**» — цикловая переменная (точнее, ее имя), «**object**» — массив, коллекция или объект, содержащие некий набор внутренних элементов.

Проще всего пояснить работу этих циклов и различия между ними на следующем примере. Создадим массив из нескольких значений. Введем в консоль браузера команду «**arr=[5, 4, 3, 2, 1]**» и нажмем «**Enter**». Теперь организуем циклы, перебирающие эти массивы. Вводим последовательно в консоль: «**for(i in arr) console.log(i)**», нажимаем «**Enter**», затем «**for(i of arr) console.log(i)**» и снова нажимаем «**Enter**».

Результатом работы первого цикла являются числа **0–4**, отвечающие за индексы (номера) элементов массива,

тогда как второй цикл выводит само содержимое массива: числа **5–1**. И те, и другие данные могут потребоваться для различных задач, из-за чего и предусмотрены две формы оператора цикла.



```

Elements    Console    Sources
top
> arr=[5,4,3,2,1]
<  ▶ (5) [5, 4, 3, 2, 1]
> for(i in arr) console.log(i)
0
1
2
3
4
< undefined
> for(i of arr) console.log(i)
5
4
3
2
1
< undefined
>

```

Рисунок 34

Сокращенные формы записи являются лучше читаемыми и экономят несколько символов программного кода при оптимизации программ. К особенностям их работы вернемся, когда более детально изучим массивы, коллекции и объекты.

Break, continue

Для того чтобы иметь возможность дополнительного управления процессом выполнения цикла предусмотрены операторы «**break**» и «**continue**». Эти операторы могут применяться во всех рассмотренных выше циклах.

Оператор «`break`» полностью останавливает выполнение цикла, независимо от состояния циклового условия. Такое действие может понадобиться, если в процессе выполнения цикла появится некоторое недопустимое значение, например, в ожидаемых числовых данных появится не-цифра, или пользователь захочет остановить программу «вручную». Также прерывание может потребоваться при возникновении условий, не позволяющих продолжать цикл, например, пропадет подключение к сети или отключится устройство, с которого читаются данные.

Оператор «`continue`» останавливает выполнение данной итерации цикла. Если цикловое условие позволяет, то будет запущена следующая итерация. Такое поведение может быть полезно при игнорировании каких-либо данных, например, пробелов и дефисов при анализе номера телефона.

В некоторых случаях операторы «`break`» и «`continue`» применяются не для реакции на непредвиденные ситуации, а для нормальной организации логики работы циклов. Рассмотрим несколько примеров:

Задача: нужно генерировать не больше десяти случайных чисел из диапазона **1–7**. В случае если выпадает четверка, генерация прекращается.

Во-первых, адаптируем генератор случайных чисел JavaScript для работы в указанном диапазоне. Существующий генератор «`Math.random()`» выдает дробные числа в диапазоне от 0 до 1. Если мы умножим это число на **6** и округлим при помощи команды `Math.round()`, то получим диапазон **0–6**. Добавив единицу, получим диапазон, заданный условиями задачи. Итоговое выражение для получения случайного числа будет выглядеть следующим образом

```
rnd = Math.round(Math.random() * 6) + 1
```

Далее организовываем цикл. Поскольку нам известно граничное количество чисел, используем цикл-счетчик «**for**». В теле цикла проверяем число на равенство четырем и, в случае равенства, останавливаем цикл оператором «**break**»

```
for(i=0; i<10; i++){
    rnd = Math.round(Math.random() * 6) + 1;
    if(rnd==4) break;
    console.log(rnd);
}
```

Этот код можно набрать в консоли браузера и запустить его нажатием «**Enter**». Для повторного выполнения нажмите стрелку вверх на клавиатуре и снова «**Enter**». Убедитесь, что каждый раз случайные числа выводятся по-разному, останавливаясь при первом выпадении четверки.

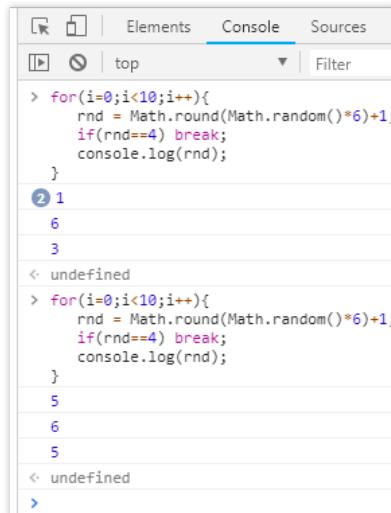


Рисунок 35

Поменяем условие задачи — нужно сгенерировать ровно 10 случайных чисел, но без четверок.

В данном случае появление четверки должно прерывать текущую итерацию цикла и запускать следующую, пока не наберется 10 чисел. Т.к. числа случайные и появление четверок непредсказуемо, количество итераций нам заведомо неизвестно. Значит, стоит отдать предпочтение условному циклу. Код тела цикла, в принципе, остается таким же, только в отличие от предыдущего условия задачи, остановка цикла заменяется на остановку итерации, то есть оператор «**break**» заменяется на «**continue**». Найдите ошибку в приведенном коде и исправьте ее перед запуском

```
i=0;
while(i<10) {
    rnd = Math.round(Math.random()*6)+1;
    if(rnd==4) continue;
    console.log(rnd);
}
```

(Здесь снова пропущены команды, влияющие на цикловое условие. В теле цикла необходимо добавить инструкцию «**i++**»).

Понятие метки

Операторы управления выполнения циклом «**break**» и «**continue**» действуют на тот цикл, в теле которого они применяются. Если анализируемые данные являются многомерными или имеют глубокую степень вложенности, то их перебор, скорее всего, будет организован

при помощи нескольких циклов, вложенных друг в друга. В такой ситуации может появиться необходимость прервать итерацию или выполнение «верхнего» цикла, а не только данного. Для реализации такой возможности в JavaScript предусмотрены метки.

Метка представляет собой идентификатор (имя), сформированный согласно общим правилам именования переменных, после которого указывается двоеточие «`:`». Метка служит для указания определенного места в коде, как правило — цикла. В следующем примере использованы две метки: «`loopI`» и «`loopJ`», относящиеся к циклам с соответствующей цикловой переменной (`i` или `j`).

```
loopI: for(i=0;i<5;i++)
    loopJ: for(j=0;j<5;j++) {
        console.log(i,j);
        if(j==3) break loopI;
    }
```

После оператора «`break`» указывается метка цикла, который нужно прервать. Если метку не указывать или указать «`loopJ`», то при условии (`j==3`) будет прекращаться вложенный цикл, при этом внешний цикл будет продолжать работать. Если же указать метку «`loopI`», то прерываться будет первый цикл, полностью останавливая работу приведенного блока.

Аналогичным образом можно использовать метки и с оператором «`continue`», прерывая итерацию заданного цикла.

Задание для самостоятельной работы

(в следующих задачах вывод данных может быть реализован как с помощью диалоговых окон, так и средствами консоли разработчика)

1. Напишите скрипт, который запрашивает у пользователя число **N** и выводит все четные числа от **2** до **N** или **N-1**, если **N** нечетное. Например: ввод **10**, вывод **2 4 6 8 10**; ввод **7**, вывод **2 4 6**.
2. Напишите скрипт, который запрашивает у пользователя число **N** и выводит все нечетные числа от **N** (или **N-1**, если **N** четное) до **1** в порядке убывания. Например, ввод **7**, вывод **7 5 3 1**; ввод **10**, вывод **9 7 5 3 1**.
3. Напишите скрипт, который запрашивает у пользователя число **N** и выводит все числа, на которые делится **N**, включая число **1** (примеры: ввод **N=10**, вывод **1, 2, 5; 11**, вывод **1**).
4. Напишите скрипт, который принимает от пользователя величину годовой депозитной ставки (в процентах) и выводит количество лет, по прошествии которых вклад увеличится вдвое.
5. Напишите скрипт, который выводит ровно **10** случайных чисел из диапазона **1–20**, кроме тех, которые делятся на **4**.
6. Из-за утечки из бака охлаждения ежедневно вытекает **10%** налитой воды. При объеме воды менее **10** литров возникает аварийная ситуация. Составьте программу, которая запрашивает у пользователя первоначальный объем воды и рассчитывает, на сколько дней работы этого хватит.

Функции

Что такое функция?

Рассмотренные в предыдущем разделе циклы позволяют многократно исполнять один и тот же блок программы, но только все его повторы будут происходить последовательно — один за другим. После прекращения работы цикла возврат в его тело невозможен (если он, конечно, не вложен в другой цикл). Однако в реальных программах часто появляется необходимость повторять один и тот же блок, но в разных ее участках, а не только последовательно.

Приведем аналогию, напрямую не связанную с программированием. Представим, что мы сотрудники канцелярии и наша задача состоит в подготовке ответов на обращения клиентов некоторой организации, то есть мы должны составлять и отправлять письма с ответами на поступающие вопросы. Так как организация из года в год ведет одну и ту же деятельность, вопросы клиентов будут довольно часто повторяться или, по крайней мере, будут очень похожими на какие-то из предыдущих вопросов. Набравшись определенного опыта мы, наверняка, составим набор заготовок писем, в которые надо будет подставить только конкретные имена, адреса, даты и всё — письмо готово к отправке. Наличие таких шаблонов-заготовок значительно упростит нам работу, исключит потенциальные ошибки, ускорит рабочий процесс.

При составлении компьютерных программ возникают подобные ситуации. Определенный набор действий приходится повторять в разных частях одной программы

или в нескольких разных программах. При этом в ряде случаев в действия нужно «подставить» новые данные, а иногда и полностью повторить весь код. Конечно, есть возможность обычного текстового клонирования (копирования-вставки), но это увеличивает размер кода, ухудшает его читаемость и может приводить к проблемам повторного использования одних и тех же переменных, т.к. при копировании будут повторены все описанные имена. Хотелось бы иметь возможность создания неких шаблонов-заготовок кода, которые можно было бы использовать, ссылаясь на них, а не копируя содержимое.

Для того чтобы реализовать такую возможность был разработан механизм функций. В программировании функциями называют отдельные самостоятельные фрагменты кода, которые могут быть вызваны в разных местах основной программы (или в других функциях) при помощи ссылки на их имя. Единожды составив и настроив код, выполняющий некоторое действие (подготовив шаблон), мы можем оформить его в виде функции и свободно использовать в произвольных местах, в которых нужно воспользоваться этим кодом.

На самом деле мы уже неоднократно сталкивались с использованием функций. Например, диалоговые окна (`alert`, `prompt` и `confirm`) являются типичными их представителями. Рассмотрим, что происходит, когда в программе появляется инструкция `«alert(x)»`.

Интерпретатор языка JavaScript обнаруживает имя `«alert»` и устанавливает, что за ним закреплена определенная функция (подпрограмма) и в эту функцию нужно передать значение переменной `«x»`. То есть для работы

функции нужно подставить определенные данные вместо «*x*». Дальше интерпретатор останавливает выполнение основной программы и переходит к подпрограмме, «запомнив» значение «*x*». Это называется вызовом функции (*function call*).

Инструкции функции «`alert`» рисуют окно и выводят в него запомненное при вызове значение «*x*» (подставляют в шаблон конкретные данные). После чего ожидают закрытия окна пользователем. Подпрограмма заканчивается, и интерпретатор возвращается к тому месту, откуда произошел ее вызов. Этот процесс называется возврат из функции (*function return*).

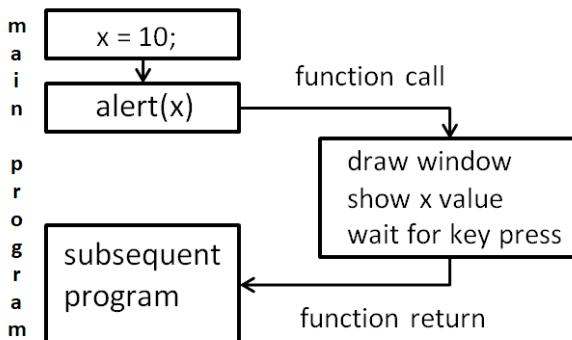


Рисунок 36

При возврате интерпретатор также может «запомнить» некоторые данные, которые следует передать из функции в основную программу. Например, когда нам был нужен пользовательский ввод, мы использовали инструкцию

```
x = prompt("Input x=")
```

Оператор присваивания сначала должен вычислить выражение в правой части от знака «`=`». При этом происходит вызов функции: интерпретатор запоминает то, что надо для работы функции (строка `Input x=»`), и передает управление в подпрограмму с именем `prompt`. После окончания работы функции результат, полученный от пользователя в окне, будет запомнен интерпретатором и перемещен в переменную `x` при возврате в основную программу.

В виде функций реализуются дополнительные возможности программирования, прямо не предоставляемые набором базовых операторов языка. Многие из таких функций изначально включены в состав JavaScript, например, диалоговые окна или математические функции. Для того чтобы разделять функции, поставляемые с языком, и функции, созданные программистом в своей программе, последние также называют «пользовательскими» функциями. Принципиальной разницы между стандартными и пользовательскими функциями нет, название лишь уточняет, о каком виде функций идет речь.

Забегая наперед отметим, что вызов функции не всегда останавливает выполнение основной программы. Есть особый вид функций, которые могут выполняться параллельно с работой программы и других функций. Они носят название «асинхронных». При этом функции, которые останавливают работу программы, называют «синхронными». Пока что мы будем иметь в виду только синхронные функции, об асинхронных будет рассказано в дальнейших разделах.

Синтаксис объявления функции

Пользовательские функции создаются при помощи ключевого слова «**function**». Синтаксис объявления функции имеет следующий вид:

```
function nameOfFunction(argument1, argument2) {  
    body  
}
```

После ключевого слова «**function**» следует имя функции (**nameOfFunction**), созданное согласно правилам (и рекомендациям) именования. После имени в круглых скобках указываются параметры, которые будут приниматься в функции. Если параметры не нужны, то круглые скобки оставляют пустыми (но не убирают). Количество параметров принципиально не ограничено.

```
function functionWithoutArguments () {  
    body  
}
```

Далее в фигурных скобках описывается тело функции, то есть сам код функции как самостоятельной программы или шаблона для подстановки в другие места программы. В отличие от условных и цикловых операторов, в случае с объявлением функции фигурные скобки обязательны, даже если функция состоит из одного оператора.

Как уже было сказано, тело функции можно представить себе, как отдельную программу. Эта программа является достаточно независимой: в ней могут описываться собственные переменные, организовывать свои циклы, проверяться необходимые условия и т.п. — в теле функции доступны все средства программирования.

Рассмотрим в качестве примера функцию,ирующую 5 однотипных блоков-заголовков (`<h2></h2>`) с надписями `Header 1 ... Header 5`. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке Sources — файл `js1_7.html`).

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

    <body>
        <script>
            function show5Blocks() {
                for(i=1;i<=5;i++)
                    document.write("<h2> Header "+i+
                                  "</h2>");
            }
            show5Blocks();
        </script>
    </body>

</html>
```

Изначально, страница, создаваемая данным файлом, пустая. В ней нет никаких HTML элементов. В блоке сце-

нариев (`<script>`) описывается функция `show5Blocks()`. В теле функции используется цикл-счетчик «`for`», который 5 раз записывает на страницу (командой `document.write`) строку, сформированную разметкой «"`<h2> Header +i+"</h2>"`». В данном случае оператор «`+`» используется для сложения строк. В результате будут получаться требуемые надписи `Header 1 ... Header 5` для каждого значения `«i»`, перебираемого циклом.

Обратим внимание, что само по себе описание функции не выполняет действий, указанных в ее теле. Для того чтобы эти действия запустились, функцию необходимо вызвать. Это обеспечивается указанием ее имени, после которого ставятся круглые скобки `«show5Blocks()»`. Именно круглые скобки сообщают интерпретатору про необходимость вызова функции, то есть про переход к командам ее тела.

Сохраните файл и откройте его в браузере. В результате на странице должны появиться 5 заголовков:

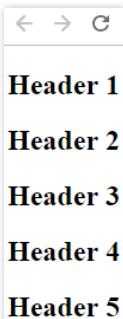


Рисунок 37

Приведенный пример, кроме определения функции, дополнитель но демонстрирует их возможности по соз-

данию новых элементов Веб-страницы. После работы функции на изначально пустой странице появляются HTML заголовки. Их содержание можно просмотреть при помощи средств разработчика и убедиться, что они являются полноценными элементами страницы, имеют все необходимые HTML атрибуты и свойства.

Параметры функции

В силу определенной независимости функции в ней может быть ограничен доступ к переменным из основной программы или других функций, откуда происходит ее вызов. Сравнивая функцию с некоторым шаблоном-заготовкой письма, можно привести аналогию, что при со-ставлении шаблона мы не можем знать конкретных дан-ных для записи, а можем лишь указать, что здесь нужно указать конкретный адрес, а здесь — адресата. Конкрет-ные данные будут известны, когда мы будем использовать шаблон, то есть в аналогии — вызывать функцию.

Если в теле функции нужно использовать какие-либо данные для подстановки извне, то необходимо принять меры по обеспечению доступа к ним. В компьютерной терминологии это называется механизмом «передачи данных в функцию». Передаваемые из программы данные называются «аргументами», а со стороны функции эти данные принимаются как «параметры». В используемой нами аналогии, параметры — это пока что пустые места, куда будут вписаны адрес или адресат, а аргументы — это сами значения адреса и адресата.

Технически, параметры функций играют роль пере-менных, значения которых устанавливаются в момент вызова функции другой программой. В теле функции параметры можно использовать в тех же языковых ин-струкциях, что и обычные переменные. При вызове функ-ции вместо параметров будут подставлены конкретные значения аргументов.

Рассмотрим пример, иллюстрирующий передачу данных: необходимо создать функцию, которая будет увеличивать значение аргумента на 1 и выводить полученный результат на страницу.

Этап 1: подготовка окружения. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке Sources — файл js1_8.html)

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <p id="Log"></p>
    <script>
    </script>
  </body>
</html>
```

Документ содержит основные разметочные определения HTML страницы, в состав которой входит абзац с идентификатором «Log» (`<p id="Log"></p>`), предназначенный для будущего отображения данных.

Этап 2: определение функции. В раздел `<script>` помещаем следующий код

```
function incAndLog(x) {
  x = x+1;
  alert("inc x = " + x);
  Log.innerHTML += "<br>inc x = " + x;
}
```

Для имени функции выбран литерал «`incAndLog`», символизирующий смысл работы функции: инкремент (увеличение на 1) и вывод результата в лог (абзац с идентификатором «`Log`»). Придерживаемся правил написания составных имен «`lowerCamelCase`».

В качестве параметра функции после ее имени в круглых скобках указываем «`x`». На данном этапе параметр называется формальным, то есть он еще не имеет значения, но может использоваться в теле функции как обычная переменная. В приведенной выше аналогии с письмами формальный параметр — это пустое место шаблона для записи конкретных данных (адреса).

Согласно поставленной задаче, увеличиваем параметр на 1 (`x = x + 1`) и выводим полученное значение, формируя сообщение «`"inc x = " + x`» (надпись «`inc x =`» и плюс само значение переменной «`x`»).

Вывод сообщения обеспечиваем двумя способами. Во-первых, вызываем диалоговое окно «`alert`» с указанным текстом сообщения. Во-вторых, добавляем полученное сообщение в состав абзаца (`Log.innerHTML`), дополнив сообщение разрывом строки «`
`». Напомним, что обращение к элементу с заданным идентификатором возможно по его имени (`Log`).

Этап 3: вызов функции и передача аргументов. После определения функции в раздел `<script>` добавим следующий код:

```
x = 2;  
incAndLog(x);
```

Сначала вводится переменная «`x`» и ей присваивается значение 2. Затем вызывается функция `incAndLog(x)`, в которую передается аргумент «`x`».

При обращении к созданному файлу браузер начнет загружать страницу. Начиная от начала разметки, первым будет создан абзац (`<p id="Log"></p>`). Затем начнется выполнение блока сценария (скрипта). В нем сначала будет определена функция «`incAndLog`». Само определение функции не приводит к выполнению ее тела, только «заготовливает» ее для будущего использования.

После определения, вводится переменная «`x=2`» и вызывается функция «`incAndLog(x)`». Во время вызова формальный параметр «`x`» в теле функции связывается с аргументом «`x`», определенным как переменная перед вызовом функции. То есть параметр приобретает значение и в дальнейшем теле функции в качестве начального значения параметра «`x`» будет использовано значение «`2`», переданное при вызове функции.

В теле функции значение «`x`» увеличивается на 1 (`x=x+1`), после чего формируется сообщение с учетом нового значения параметра (`2+1=3`). Сообщение выводится при помощи диалогового окна и, затем, это же сообщение копируется на страницу.

Описанную последовательность вызовов функций при работе созданного сценария можно представить в виде схемы (см. рис. 38).

Сохраните созданный файл и откройте его при помощи браузера. После загрузки страницы должно появиться диалоговое окно с сообщением «`inc x = 3`». Обратите внимание, что сама страница остается пустой. То есть

сообщение не дублируется на странице, пока выполняется функция диалогового окна.

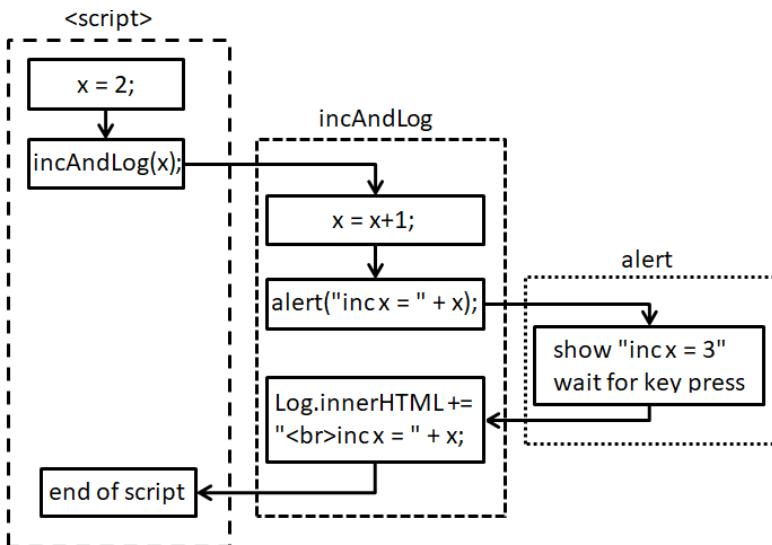


Рисунок 38

Закройте диалоговое окно и убедитесь, что на странице появляется такое же сообщение «**inc x = 3**».

Следует обратить внимание, что при передаче аргумента в функцию его значение попадает в переменную-параметр «**x**» путем копирования. В результате чего изменения, внесенные в значение параметра (**x = x + 1**) не отображаются на значениях аргумента в точке вызова функции, т.к. функция работает с копией аргумента. Проверим это на следующем примере.

Внесите изменения в предыдущий или создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке Sources — файл *js1_9.html*).

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

Дополнительно к предыдущему примеру на страницу выводятся сообщения о значении переменной «`x`» до и после вызова функции. Вполне очевидно, что до вызова функции значение «`x`» равно `2`. А вот что произойдет после выполнения тела функции, ведь в ней используется переменная-параметр с точно таким же именем и она увеличивается на `1`?

Сохраните файл и откройте его при помощи браузера. Убедитесь, что окно «`alert`» содержит такое же сообщение, как и в предыдущем примере (`inc x = 3`). После закрытия окна на странице появляются три строки сообщения.

Первая строка выводится до вызова функции и содержит, как и ожидалось, значение `2`. Затем идет стро-

ка, сформированная самой функцией, сообщающей, что внутри функции «`x`» имеет значение `3`. Третей идет строка, отображающая значение «`x`» после вызова функции. Как видно (см. рис. 39), это значение все так же равно `2`. Убеждаемся в том, что функция работала со своей копией аргумента и не повлияла на внешнюю переменную даже при полном совпадении их имен.

```
← → C
x = 2
inc x = 3
x = 2
```

A screenshot of a terminal window. At the top, there are three buttons: a left arrow, a right arrow, and a 'C' button. Below them is a horizontal line. The terminal displays three lines of text:
x = 2
inc x = 3
x = 2

Рисунок 39

Возвращаемое значение функции. Ключевое слово return

Для обеспечения обратного потока данных, — из функции в основную программу, — применяется механизм возврата значений. Подобно тому, как значения из программы попадают в функцию через параметры, значения из функции могут попасть в программу (в точку вызова функции).

В объявлении функции для возврата значения используется ключевое слово «`return`», после которого указываются передаваемые из функции данные. Выполнение команды `«return»` прекращает работу функции, даже если в ее теле есть дальнейшие инструкции. Управление передается в точку вызова функции, а переданные данные могут быть использованы как результат, заменяющий собой имя функции. Этот результат может быть помещен (присвоен) в некоторую переменную, либо включен в состав выражения.

Рассмотрим пример: необходимо создать функцию, вычисляющую куб переданного аргумента (напомним, куб — это результат умножения числа самого на себя 3 раза: $x^3=x\cdot x\cdot x$)

Этап 1: определение функции

```
function cube(x) {  
    return x*x*x;  
}
```

Расчет значения куба числа достаточно несложный, поэтому нет необходимости создавать вспомогательные переменные или использовать дополнительные языковые конструкции. Тело функции состоит из единственного выражения «`return x*x*x;`». Его выполнение заключается в вычислении умножения `«x*x*x»` и запуска механизма возврата полученного результата.

Для исследования таких небольших функций удобно использовать консоль браузера вместо создания отдельного документа. Откройте консоль разработчика на любой странице браузера. Введите в консоль (можно скопировать и вставить) приведенное выше определение функции, убедитесь в отсутствии ошибок.

Этап 2: вызов и использование функции

Попробуем вызвать функцию. Наберем в консоли

```
cube(2)
```

и нажмем «`Enter`». В качестве ответа в консоли появляется результат «`8`». Этот результат, например, можно сохранить в переменной «`y`»:

```
y = cube(2)
```

или использовать в расчете

```
2 * cube(2)
```

или передать как аргумент в другую функцию

```
alert( cube(2) )
```

Возвращаемое значение функции. Ключевое слово return

The screenshot shows a browser's developer tools with the 'Console' tab selected. The session log is as follows:

```
> function cube(x) { return x*x*x; }
< undefined
> cube(2)
< 8
> y = cube(2)
< 8
> 2 * cube(2)
< 16
> alert( cube(2) )
< undefined
>
```

Рисунок 40

Ключевое слово «`return`», если сказать очень упрощенно, подставляет рассчитанный справа от него результат в то место программы, в котором был произведен вызов функции. Дальнейший код будет использовать полученный результат так, как будто вызова функции не было, а в данном месте кода было бы записано точно такое же значение.

В завершение первого знакомства с функциями приведем реализацию банковского округления. Сравните следующий код с примером из раздела «условия»:

```
function bankerRound(x) {
    if(Math.round(x)%2 == 0)
        return Math.round(x);
    else{
        if(x<Math.round(x) )
            return Math.round(x)-1;
        else
```

```

        return Math.round(x)+1;
    }
}

```

В отличие от исходного примера, вместо расчета итогового значения «**bx**», мы применили несколько команд «**return**». Такой прием может применяться для ускорения работы функции — когда нужное значение уже получено, нет необходимости его сохранять в отдельную переменную и анализировать дальнейшие условия. Можно завершить работу функции и передать в точку вызова итоговый результат.

Ведите или скопируйте определение функции в консоль браузера. Попробуйте ее вызывать с различными значениями аргумента

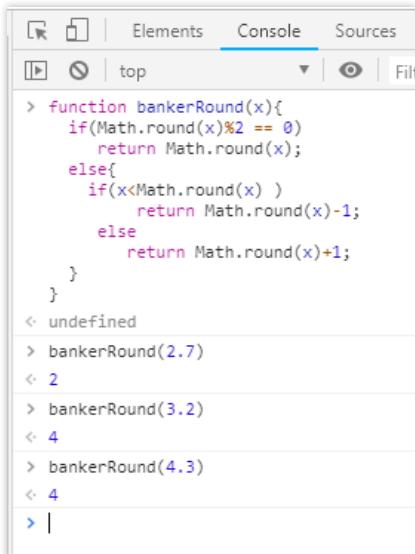


Рисунок 41

Найдите ошибку. Следующая функция должна вернуть строку «**odd**» если переданный в нее аргумент является нечетным числом или «**even**» — если четным. Правильным ли является следующее определение для такой функции?

```
function getParity(x) {  
    if(x % 2 == 0)  
        parity = "even";  
    else  
        parity = "odd";  
}
```

(В функции пропущена инструкция возврата значения. В конце тела должно быть указано «**return parity**». Либо в каждой ветке условного оператора вместо «**parity=**» можно написать «**return**».)

Задание для самостоятельной работы

1. Создайте функцию `sayError()`, которая будет выводить (при помощи диалогового окна `alert`) сообщение с текстом «`Some error occurred!`».
2. Создайте функцию `showError(x)`, которая будет выводить (при помощи диалогового окна `alert`) сообщение с текстом «`Error X occurred!`», где `X` — текст из аргумента функции (например, вызов `showError('Out of memory')` должен вывести сообщение «`Error Out of memory occurred!`»).
3. Создайте функцию `createHeaders(N)`, которая создаст на странице `N` заголовков второго уровня (`<h2>`) с надписями `Header1, Header2 ... HeaderN`.
4. Создайте функцию `checkPassword(x)`, которая вернет значение `true`, если в качестве аргумента в нее будет передан допустимый пароль (одно из значений «`Step`», «`Web`» или «`JavaScript`»). Иначе функция должна вернуть `false`.
5. Создайте функцию определения знака числа `sign(x)`, которая вернет значение `-1`, если аргумент «`x`» — отрицательное число, `1` — если положительное, `0` — если аргумент «`x`» равен нулю.
6. Предложите имя (согласно правилам именования) и создайте функцию, которая будет возвращать названия дней недели по их номеру: `0-Sunday, 1-Monday, 2-Tuesday, 3-Wednesday, 4-Thursday, 5-Friday, 6-Saturday`.

Детальное
о функциях

Объект arguments

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

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

Цель и задачи объекта

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

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

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

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

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

Во-вторых, проанализируем состав данных, отображенных в консоли. В появившихся результатах нажимаем на треугольный символ слева от объекта, раскрывая дополнительные детали (см. рис. 42).

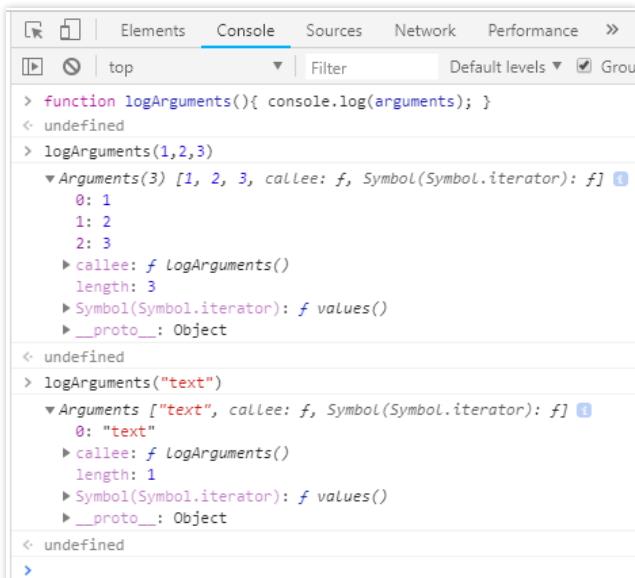


Рисунок 42

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

Свойство `length`

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

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

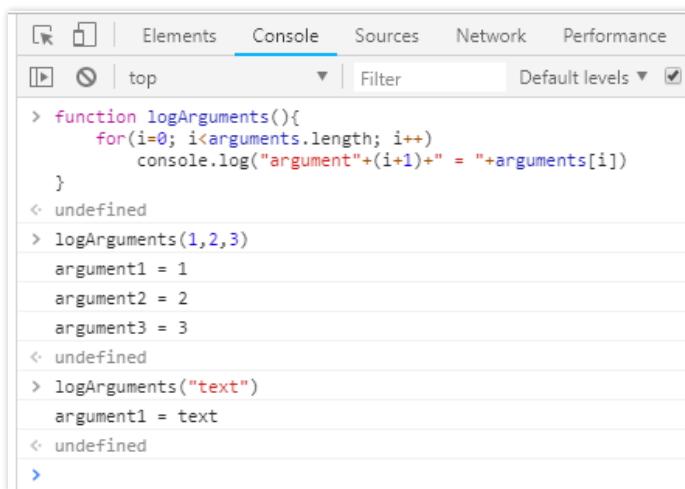
Модифицируем определение функции «`logArguments`» для вывода переданных в функцию аргументов, только без отображения остальных полей объекта «`arguments`».

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

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

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

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



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The console output displays the execution of the `logArguments` function. It shows the definition of the function, its execution with arguments (1, 2, 3) and ("text"), and the resulting output in the console.

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

Рисунок 43

Наблюдаем новые результаты вызова функции, что свидетельствует о выполнении нового тела, то есть о замене старого определения функции на новое (рис. 43).

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

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

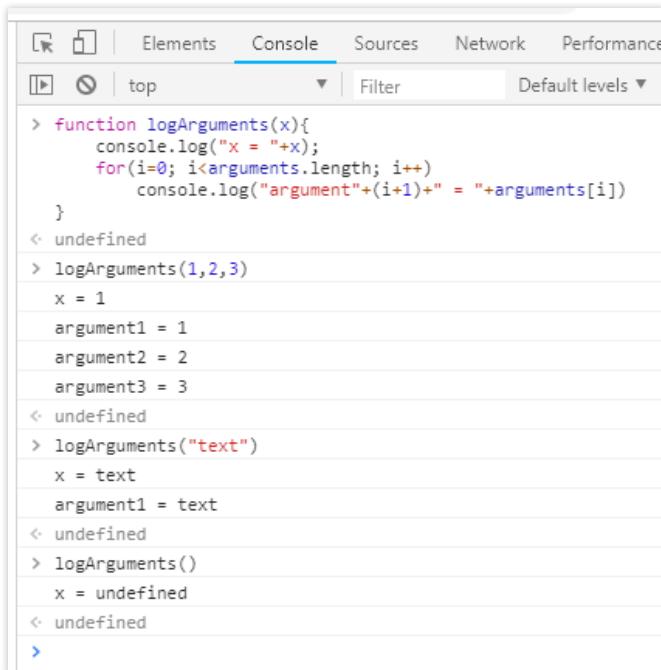
```
function logArguments(x) {  
    console.log("x = "+x);  
    for(i=0; i<arguments.length; i++)  
        console.log("argument"+(i+1)+" = "+arguments[i])  
}
```

Повторим запросы на вызов функции и убедимся в возможности одновременного доступа к первому аргументу как по имени «`x`», так и через свойство «`arguments[0]`» (см. рис. 44).

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

Объект arguments

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



The screenshot shows the Google Chrome DevTools interface with the 'Console' tab selected. The console output displays the following code and its execution results:

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

Рисунок 44

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

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

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

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

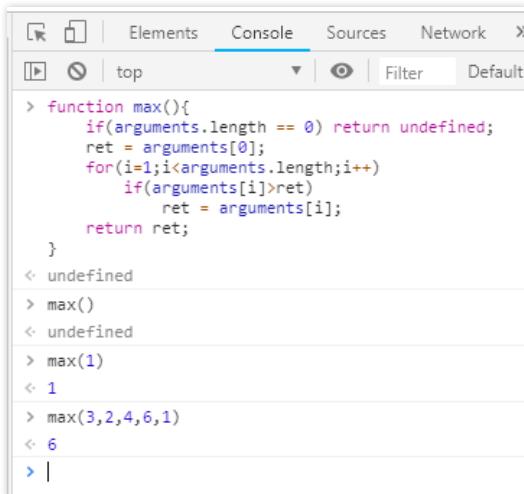


Рисунок 45

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

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

Ведите или скопируйте определение функции в консоль браузера. Попробуйте ее вызвать с различными значениями аргумента (рис. 45).

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

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

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

Особенности функций в JavaScript

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

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

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

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

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

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

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

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

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

Следующий код функции иллюстрирует описанные особенности.

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

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

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

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

Область видимости переменной

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

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

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

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

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

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

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

Для того чтобы устраниТЬ описанную ошибку, определим переменную «`x`», присвоив ей значение 1. Введем в консоли «`x=1`» и нажмем «`Enter`». После чего снова вызовем функцию «`logX()`». Как результат увидим значение «`1`» (см. рис. 46).

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

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

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

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

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

A red error message 'Uncaught ReferenceError: x is not defined' is highlighted, indicating that the variable 'x' is not defined in the current scope of the 'logX()' function. The console also shows the assignment of 'x' to values 1 and 2, and its subsequent printing.

Рисунок 46

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

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

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

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

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

Для иллюстрации эффекта побочного действия используем ранее созданный файл с функцией `incAndLog` (*исходный код доступен в папке Sources — файл js1_9.html*) и внесем в него изменения — исключим параметр «`x`» из объявления функции (вместо объявления «`function incAndLog(x)`» запишем «`function incAndLog()`»). Остальной код остается без изменений:

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

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

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

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

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

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

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

Рисунок 47

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

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

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

Область видимости переменной

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

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

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

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

Ведите или скопируйте в консоль этот цикл и нажмите «Enter». В результате мы увидим только один запуск функции вместо ожидаемых трех (см. рис. 48).

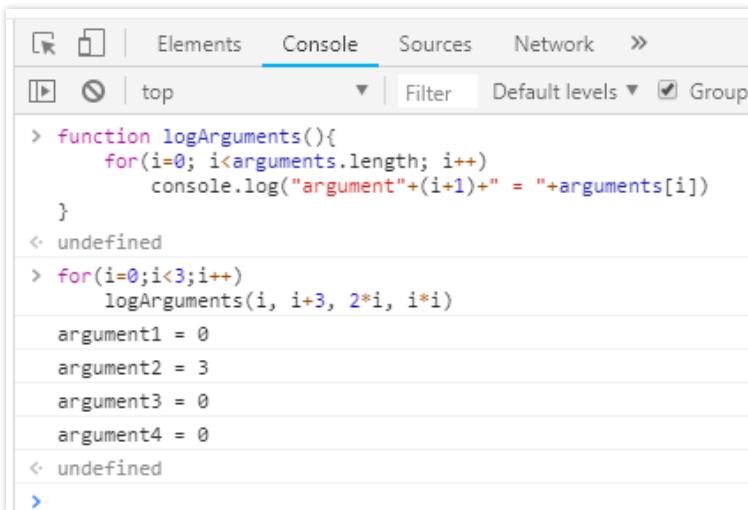


Рисунок 48

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

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

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

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

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

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

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

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

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

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

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

Приведем еще один пример, иллюстрирующий положительное использование эффекта побочного действия функций. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке *Sources* — файл *js1_10.html*).

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

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

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

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

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

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

Поднятие объявлений

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

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

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

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

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

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

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

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

```

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

```

Рисунок 49

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Рекурсия

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

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

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

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

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

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

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

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

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

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

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

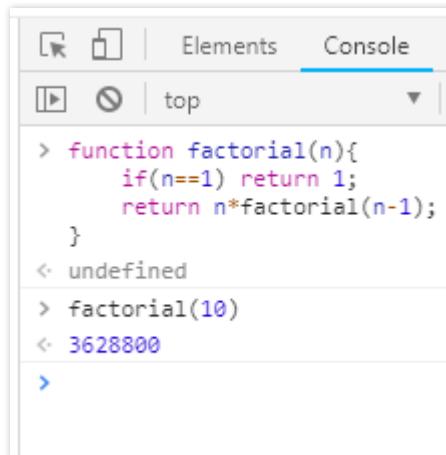
```
function factorial(n) {
    if(n==1) return 1;
    return n*factorial(n-1);
}
```

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

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

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

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



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

Рисунок 50

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

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

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

Например, поставим задачу вывести ряд чисел от единицы до введенного пользователем числа с дополнительным ограничением — циклы использовать не разрешается. С первого взгляда кажется, что не зная величины предельного числа, без циклов задача не может быть решена. Но тут на помощь приходит рекурсия. Поскольку нам понадобится пользовательский ввод, оформим программу в виде отдельного файла, а не в консоли. Создайте новый файл, наберите или скопируйте в него следующее содержание (код также доступен в папке Sources — файл js1_11.html).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Задание для самостоятельной работы

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

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



Unit 1. Введение в JavaScript

© Денис Самойленко.

© Компьютерная Академия «Шаг», www.itstep.org.

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

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

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