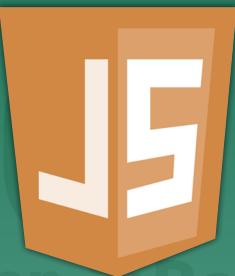


Разработка клиентских сценариев с использованием JavaScript и библиотеки jQuery



Unit 1

Введение в JavaScript

Contents

Сценарии, выполняемые на стороне клиента	4
Что такое JavaScript?.....	10
История создания JavaScript.....	12
Различия между JavaScript и Java, JScript, ECMAScript....	15
Версии JavaScript.....	17
Понятие Document Object Model	20
Понятие Browser Object Model	24
Внедрение в HTML документы.	
Редакторы кода JavaScript	28
Тег <noscript>.....	32
Основы синтаксиса	38
Регистrozависимость	39
Комментарии.....	41
Ключевые и зарезервированные слова	42
Переменные. Правила именования переменных.....	45

Типы данных	52
Операторы	57
Арифметические операторы.....	59
Операторы отношений	62
Логические операторы	69
Оператор присваивания.....	74
Битовые операторы.....	80
Приоритет операторов.....	85
Оператор typeof	88
Задание для самостоятельной работы	90

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

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.

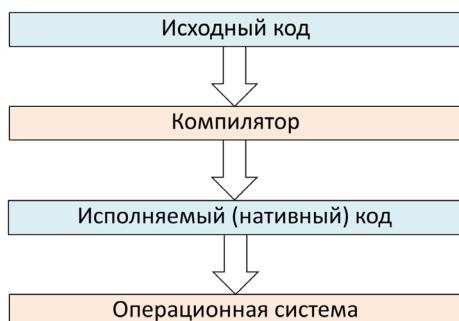


Рисунок 1



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.

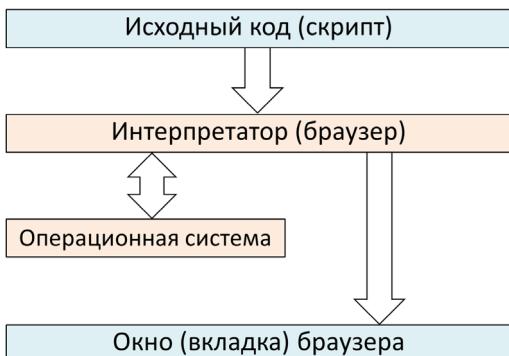


Рисунок 3

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 сценарии) 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 дизайнёрами, 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](#).

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](#).

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 браузерами 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», расширяющий 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 стан-

дарта 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.

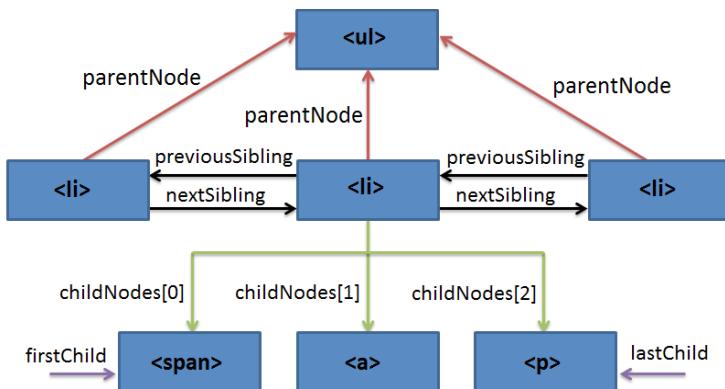


Рисунок 4

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](#). 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).

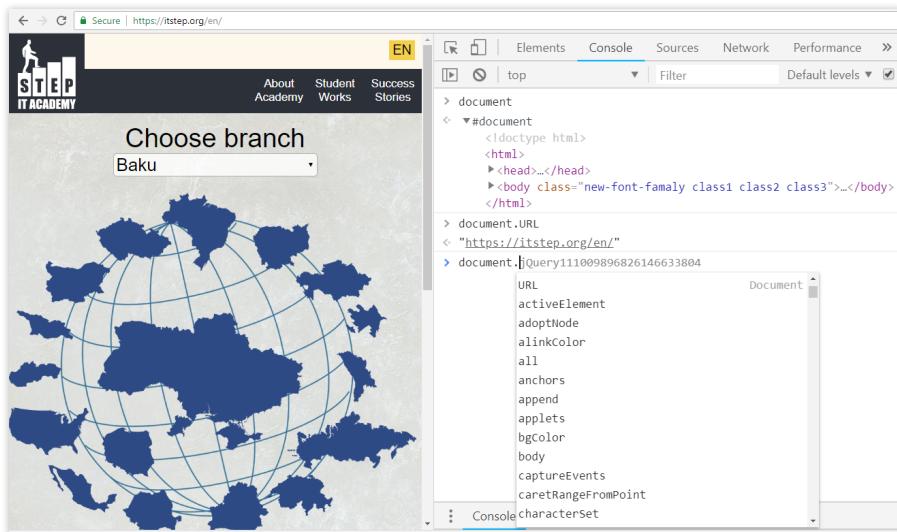


Рисунок 5

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.

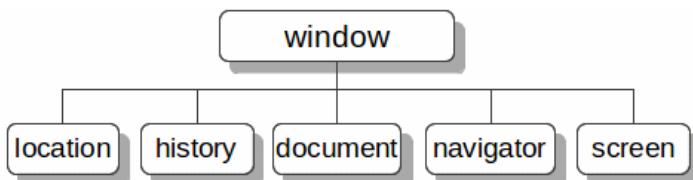


Рисунок 6

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.

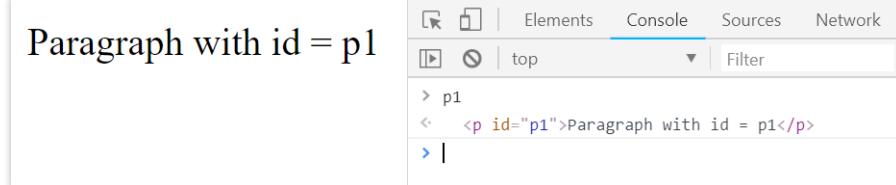


Рисунок 7

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.

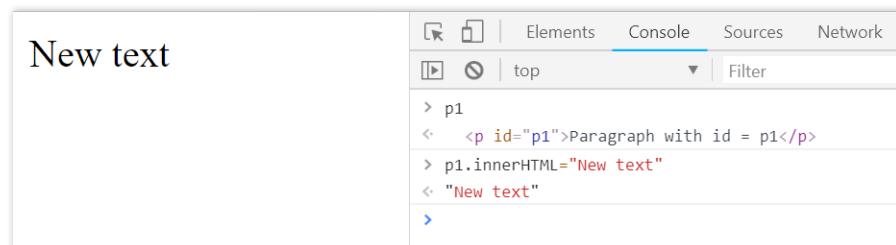


Рисунок 8

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")`.

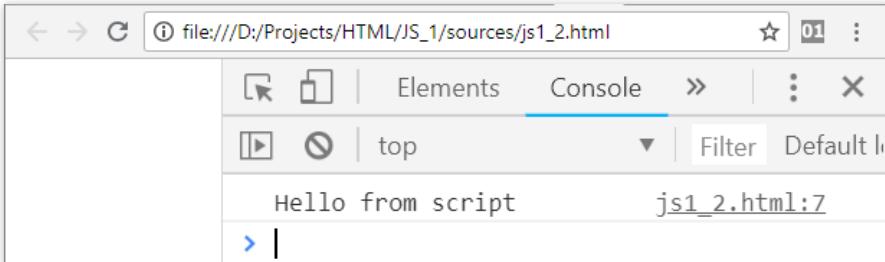


Рисунок 9

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».

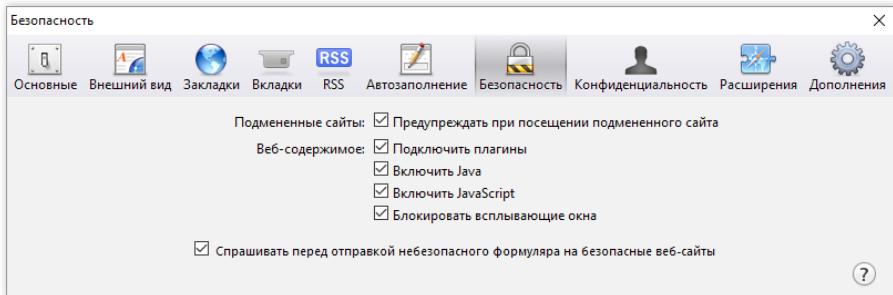


Рисунок 11

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

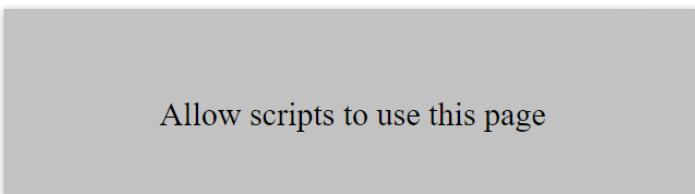


Рисунок 12

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, Фейсбука](#) или [Гугла](#). Обратите внимание на то, что загрузка сайтов проходит нормально 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 «`documemt.`», 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 одинако-

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 «регистровозависимостью». Обратная ситуация описывается термином «регистронезависимостью».

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](#) (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, throws, 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.

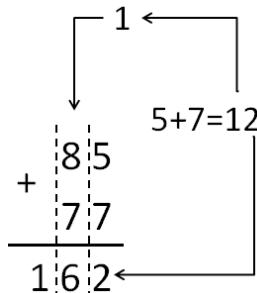


Рисунок 13

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 «о» 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).

Составные части операции называются операндами. Другими словами, операнды — это то, что участвует в операции. Например, в выражении «`2+3`» оператор представлен инструкцией «`+`», операнды — 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·10E`. `M` называют мантиссой числа, `E` — экспонентой. При записи мантисса отделяется от экспоненты буквой «`E`». Например, $1\text{E}2 = 1 \cdot 10^2 = 100$; $3.4\text{E}-2 = 3.4 \cdot 10^{-2} = 0.034$. 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	<code>x + y</code>	$x + y$
Вычитание	<code>x - y</code>	$x - y$
Умножение	<code>x * y</code>	$x \cdot y$
Деление	<code>x / y</code>	$x : y$
Остаток от деления	<code>x % y</code>	$x \bmod y$
Возведение в степень	<code>x ** y</code>	x^y
Инверсия (смена знака)	<code>-x</code>	$-x$
Инкремент (увеличение на 1)	<code>x++ или ++x</code>	$x+1$
Декремент (уменьшение на 1)	<code>x-- или -x</code>	$x-1$

Дополнительные математические функции имеют специфические области применения и имеют имена, подобные 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 будет 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 выражений, 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 сравнен с предельным значением выражениями

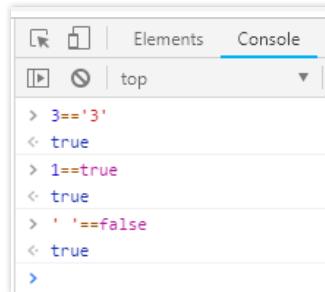
«больше» или «меньше» и, в зависимости от результата, программа будет выполняться по-разному.

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).



The screenshot shows a browser's developer tools console tab labeled "Console". It contains the following code and output:

```
> 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).



The screenshot shows a browser's developer tools console tab labeled "Console". It contains the same code as in Figure 14, but the results are different due to the use of the non-strict equality operator `==`:

```
> 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 operandов 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 вни-

мательным, 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 the 'Console' tab of a browser's developer tools. It displays the following interactions:

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

Рисунок 16

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

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

Рисунок 17

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).

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'
< "10px"
> y='3px'
< "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 до-

вольно 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 операнд.

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 операнды 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 операнды равны «`false`». За это операцию «ИЛИ» сравнивают с логическим сложением (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 «**Ок**». 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 «**Ок**» 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`» поменяется с «`1`» на «`2`».

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

мость 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).

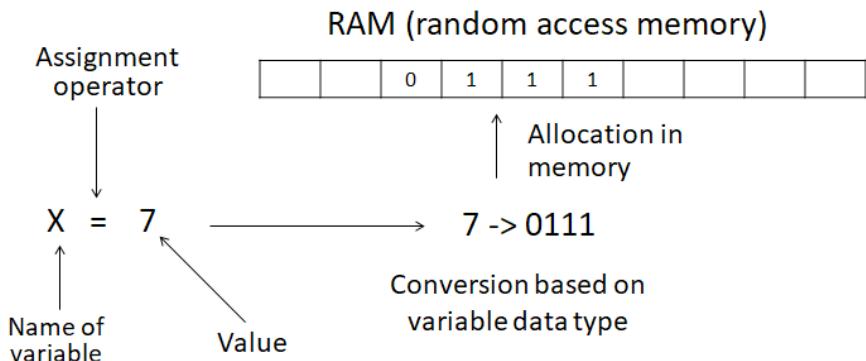


Рисунок 19

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 идома (*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 высокочувствительных языков, к которым относится JavaScript, 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 консоли 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 or y**» проще всего представить в виде «таблицы истинности» этой операции.

y	x	0	1
0	0 or 0	0 or 1	
1	1 or 0	1 or 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	1
0	0	0	1
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 operandам типа Boolean и имеют такой же результат.

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

Рисунок 20

Битовые операции применяются к произвольным данным и действуют на каждый бит операндов отдельно. В результате операции также получается новая битовая последовательность (рис. 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.

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

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 operandы 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, **x=8** — 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).

Операторы



Unit 1.

Введение в JavaScript

© Денис Самойленко.

© Компьютерная Академия «Шаг», www.itstep.org.

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

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

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