ФГБОУ ВО «Воронежский государственный
технический университет»

М.Ю. Сергеев Т.И. Сергеева

# ОСНОВЫ ВЕБ-ПРОГРАММИРОВАНИЯ

Утверждено Редакционно-издательским советом
университета в качестве учебного пособия

Воронеж 2016

УДК 681.3

Сергеев М.Ю. Основы веб-программирования: учеб. пособие / М.Ю. Сергеев, Т.И. Сергеева. Воронеж: ФГБОУ ВО «Воронежский государственный технический университет», 2016. 253 с.

В учебном пособии рассматриваются основные вопросы разработки веб-приложений с помощью языков программирования JavaScript и PHP.

Издание соответствует требованиям Федерального государственного образовательного стандарта высшего образования по направлению подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» (направленность «Вычислительные машины, комплексы, системы и сети»), дисциплине «Проектирование и разработка Web-приложений».

Учебное пособие предназначено для студентов четвертого курса.

Табл. 21. Ил. 15. Библиогр.: 5 назв.

Научный редактор д-р техн. наук, проф. С.Л. Подвальный

Рецензенты: кафедра вычислительной математики
и прикладных информационных технологий
Воронежского государственного университета
(зав. кафедрой д-р техн. наук,
проф. Т.М. Леденева);
д-р техн. наук, проф. В.Ф. Барабанов

# ВВЕДЕНИЕ

Пособие рассматривает основные вопросы разработки веб-приложений с помощью языков программирования JavaScript и PHP.

Пособие содержит описание основных функций данных языков, методов создания сценариев для повышения интерактивности веб-страницы, функций и модулей специальных пользовательских библиотек языке JavaScript. Особое внимание в учебном пособии уделяется вопросам взаимодействия веб-приложений с базами данных, созданных средствами СУБД MySQL.

Первая глава – это введение в язык JavaScript. В данной главе рассмотрены правила написания кода на языке JavaScript, базовые элементы языка, работа с массивами и циклами, регулярными выражениями и данными различного типа.

Вторая глава посвящена созданию динамических веб-страниц, модифицированию веб-страниц с использованием jQuery, работе с событиями JavaScript, использованию AJAX.

Третья глава является введением в язык PHP. Рассмотрены базовые конструкции языка, работа с массивами, датами, форматирование данных, работа с файлами и взаимодействие с веб-формами, введение в MySQL.

Учебное пособие полностью соответствует требованиям стандарта по дисциплине «Проектирование и разработка Web-приложений» и предназначено для студентов очной полной формы обучения.

# 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА JAVASCRIPT

**JavaScript** – это язык программирования, созданный для веб-разработки. Он дает пользователям возможность изучить механику и элементы программирования веб-страниц. JavaScript и Java – это разные языки. Хотя они имеют общего разработчика (Sun Microsystems), одинаковый синтаксис и ключевые слова.

JavaScript относится к языкам объектно-ориентированного программирования. Это означает, что существуют объекты (например, окно) с определенными свойствами (например, длиной) и методами. Методы или действиями (например, открытие) выполняются в ответ на события – действия, совершенные по отношению к объекту (например, щелчок мышью).

JavaScript является интерпретируемым языком. Это значит, что обработка кода осуществляется по ходу выполнения программы, без необходимости его компиляции. Для создания файлов, содержащих код, понадобится лишь текстовый редактор, и веб-браузер, который осуществит интерпретацию кода. Интерпретированный код, подобный JavaScript, обычно не может выполняться так же эффективно и быстро, как компилированный код. Однако JavaScript может эффективно применяться в задачах, которые ставятся при создании динамических страниц.

JavaScript обрабатывает данные на компьютере пользователя. Это значит, что при обработке данных код выполняется на компьютере пользователя, освобождая сервер для реализации более сложных задач. В этом случае пользователь быстрее получает результаты выполнения кода. Многие веб-приложения используют обработку данных и со стороны сервера, и со стороны пользователя. Разделение обработки позволяет получить преимущества обоих способов.

## 1.1. Добавление кода JavaScript на страницы

Существует два способа добавления кода JavaScript на веб-страницы: добавление прямо в код используемой страницы или использование отдельного файла.

В первом варианте код пишется непосредственно на веб-странице, внутри пары тегов <script></script>. Тег <script> может быть помещен либо в секцию <head>, либо в секцию <body> страницы. Обычно скрипт, который выводит данные прямо на страницу, помещен в секцию <body> веб-страницы. Скрипты, помещенные в секцию <head>, обычно используются другими скриптами, расположенными в секции <body>. В примере ниже демонстрируется использование обоих способов.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
<script language="javascript" type="text/javascript">
document.title = "Пример кода JavaScript"
</script>
</head>
 <body>
<script language="javascript" type="text/javascript">
document.write ("Заголовок данной веб-страницы - "+
document.title)
</script>
</body>
</html>
```

По мере того как скрипты будут становиться сложнее, возникнет необходиомость повторно использовать одни и те же коды внутри страницы. Возможно уменьшить размер веб-страниц и повторно использовать один и тот же код, вынеся большую его часть в отдельный текстовый файл с расширени-

ем .js. При необходимости запустить код из JavaScript-файла (например, myjavascript.js) достаточно будет добавить соответствующий атрибут к тегу <script>, например:

<**script** type="text/ javascript" src="myjavascript.js"> </**script**>.

## 1.2. Правила написания кода на JavaScript

**Комментирование скрипта**

JavaScript позволяет добавлять комментарии к скрипту двумя способами: в одной строке или в нескольких строчках.

Если комментарий ставится на одной строке, то он начинается с сочетания //. Например,

```
// Это комментарий
```

Комментарий, который занимает несколько строк, ограничивается с помощью /* */. Например,

```
/* Это комментарий, который можно
написать на нескольких строках */
```

**Общие правила написания кода**

Наиболее важными правилами являются следующие:

– некоторые элементы JavaScript чувствительны к регистру, в том числе команды и имена переменных JavaScript;

– каждая инструкция (команда) в JavaScript обычно заканчивается точкой с запятой (инструкция – это строчка кода, законченная и синтаксически правильная для выполнения определенной задачи);

– текст, то есть любое сочетание букв и цифр, которое также называют *строкой,* должен быть заключен в одинарные (') или двойные ("") кавычки – разница в их использовании

разъяснена в нижеследующем примечании. Кавычки обязательно должны быть одинаковыми и парными;

– большинство функций требуют, чтобы их параметры были заключены в круглые скобки;

– несколько параметров должны разделяться запятыми (,).

**Специальные символы и управляющие последовательности**

Перед большинством символов, которым JavaScript предписывает специальное применение, ставится обратная косая черта. Вот список наиболее часто применяемых управляющих последовательностей или специальных символов:

\" создает двойные кавычки;
\' создает одинарные кавычки;
\\ создает обратную косую черту;
\b создает символ возврата (backspace);
\r создает символ возврата каретки;
\n создает символ перевода строки;
\t создает символ табуляции.

**Зарезервированные слова**

Данные слова не должны использоваться для назначения имен элементов JavaScript, так как они являются частью языка и обладают специальным значением для интерпретатора JavaScript. Они приведены в табл. 1.

Таблица 1

Зарезервированные слова JavaScript

| | | | | |
|---|---|---|---|---|
| abstract | boolean | break | byte | case |
| Catch | char | class | const | continue |
| debugger | default | delete | do | double |
| Else | enum | export | extends | false |
| Final | finally | float | for | function |
| Goto | if | implements | import | in |

Продолжение табл. 1

| | | | | |
|---|---|---|---|---|
| Native | New | null | package | private |
| protected | public | return | short | static |
| Super | switch | synchronized | this | throw |
| throws | transient | true | try | typeof |
| Var | void | volatile | while | with |

## 1.3. Базовые элементы JavaScript

**Переменные и постоянные данные**

Информация (или данные), использованная в JavaScript, попадает в одну из двух категорий, которая определяется присвоенным именем: *данные* или *тип данных*.

Переменные данные могут содержать разные значения в разное время, пока выполняется скрипт. Они инициализируются ключевым словом var, за которым следует имя, присвоенное переменной.

Постоянные данные содержат одинаковые значения в любой момент выполнения скрипта.

Типы данных приведены в табл. 2.

Таблица 2

Типы данных JavaScript

| Тип данных/ подтип | Имя | Описание | Примеры |
|---|---|---|---|
| Булево | bool | TRUE или FALSE, нечувствительны к регистру, но чаще пишутся заглавными буквами (верхним регистром) | TRUE, FALSE |
| Числа/числа с плавающей запятой | float | Дробные числа; могут быть отрицательными и использовать экспоненциальное представление чисел | 7.34, -21.89, 2.31e3 |

Продолжение табл. 2

| | | | |
|---|---|---|---|
| Числа/целые числа | int | Целое число без дробей, может быть отрицательным | 43, 928, -4 |
| Числа/нуль | null | Отсутствие любого значения | NULL |
| Строка | string | Набор символов (одной из 256 букв, чисел и специальных символов), закрытый в одиночные или двойные кавычки | "Миша", "Seattle", "Дом 101, улица 9-го мая" |
| Неопределенный | | Переменная, которой было присвоено имя, но не присвоено значение | |

**Имена переменных**

Имена переменных формируются по следующим правилам:

– должны начинаться с буквы, символа $ или знака подчеркивания;

– имена переменных могут содержать только буквы, числа, символы $ и знак подчеркивания;

– имена переменных чувствительны к регистру;

– имена переменных не должны совпадать с зарезервированными словами.

JavaScript, как правило, распознает используемый тип данных по контексту в скрипте, избавляя от задачи его указания. Тем не менее, следует учитывать следующие моменты:

– булево FALSE равнозначно целому 0, нулю с плавающей запятой — 0.0, пустой строке или строке со значением "0", массиву нулевых элементов или значению NOLL. Любые другие значения задаются выражением TRUE;

– целые числа по умолчанию используют десятичную систему. Чтобы перевести число в шестнадцатеричную систему следует поставить перед ним 0x;

– очень большие целые числа (больше 2 147 483 647) считаются числами с плавающей запятой;

– при выполнении деления двух целых чисел, получается число с плавающей запятой, за исключением случаев, когда числа могут быть поделены без остатка;

– числа с плавающей запятой не могут быть абсолютно точными до последней цифры из-за бесконечной прогрессии таких дробей, как одна третья;

– строка, содержащая число (целое или с плавающей запятой) сразу после левой скобки, может использоваться как число. Например, строки **"18.2"** и **"4 машины"** могут быть использованы как числа, тогда как **"его 4 машины"** – не могут.

Когда переменная создана, то в ней возможно хранить данные любого типа по своему усмотрению. Чтобы сразу задать значение переменной следует использовать знак равенства:

```
var s = 0;
var a = “Фамилия”
var zakaz = true
```

**Операторы**

Для работы с данными в JavaScript существует множество операторов. Наиболее важные из них приведены в табл. 3.

Таблица 3

Операторы JavaScript

| Тип оператора | Имя операции | Пример | Объяснение |
|---|---|---|---|
| Арифметические | | | Выполняет арифметические вычисления над двумя операндами – переменными или числами |
| + | Прибавить | a + b | Суммирует два операнда |
| - | Вычесть | a - b | Дает разницу между двумя операндами |

Продолжение табл. 3

| | | | |
|---|---|---|---|
| * | Умножить | a * b | Умножает два операнда |
| / | Разделить | a / b | Делит два операнда |
| -- | Декремент | --a<br>a-- | Вычитает 1 из a, после чего возвращает a.<br>Возвращает a, после чего вычитает 1 из a |
| | Инкремент | a++ | Прибавляет 1 к a, после чего возвращает a.<br>Возвращает a, после чего прибавляет 1 к a |
| % | Модуль | a % b | Возвращает остаток от операции деления |
| - | Унарный минус | -a | Меняет значение с отрицательного на положительное или с положительного на отрицательное |
| Присваивание | | | Заменяет одно значение на другое |
| = | Присвоить | a = 7 | a присвоено значение 7 |
| += | Прибавить и присвоить | a += 2 | a увеличено на 2, и ему присвоено новое значение |
| -= | Вычесть и присвоить | a += 2 | a уменьшено на 2, и ему присвоено новое значение |
| *= | Умножить и присвоить | a *= 2 | a умножено на 2, и ему присвоено новое значение |

Продолжение табл. 3

| | | | |
|---|---|---|---|
| /= | Разделить и присвоить | a /= 2 | a поделено на 2, и ему присвоено новое значение |
| %= | Вычислить модуль и присвоить | a %= 2 | a поделено на 2, и ему присвоено значение, равное остатку после деления |
| Побитовые операторы | | | Логические следствия устанавливают конкретные биты в 0 или 1 для целочисленных значений |
| & | И | a & b | Устанавливает 1 для бита, в котором 1 установлено как для a, так и для b |
| \| | ИЛИ | a \| b | Устанавливает 1 для бита, в котором 1 установлено для a, b или обоих операндов |
| ^ | Исключить | a ^ b | Устанавливает 1 для бита, в котором 1 установлено либо для a, либо для b, но не для обоих операндов |
| ~ | НЕ | -a | Устанавливает противоположный бит для каждого операнда |
| << | Сдвинуть влево | a << b | Сдвигает двоичный код a на b битов влево ( |
| >> | Сдвинуть вправо | a >> b | Сдвигает двоичный код a на b битов вправо |

Продолжение табл. 3

| Операторы сравнения | | | Сравнивает два значения и возвращает TRUE либо FALSE |
|---|---|---|---|
| == | Равно | a = b | TRUE, если a равно b |
| === | Строго равно | a === b | TRUE, если a строго равно b |
| ! = | Не равно | a != b | TRUE, если a не равно b |
| !== | Не строго равно | a ! == b | TRUE, если a не строго равен b |
| < | Меньше чем | a < b | TRUE, если a меньше чем b |
| > | Больше чем | a > b | TRUE, если a больше чем b |
| <= | Меньше чем или равно | a <= b | TRUE, если a меньше чем или равно b |
| >= | Больше чем или равно | a >= b | TRUE, если a больше чем или равно b |
| Логические операторы | | | Логические следствия |
| && | И | a && b | TRUE, если как a, так и b TRUE |
| \|\| | ИЛИ | a \|\| b | TRUE, если либо a, либо b TRUE |
| ! | НЕ | !a | TRUE, если a не TRUE |
| Другие | | | |
| | Условный оператор | Condition ? value : value | Инструкция if-then-else (если-тогда-иначе), где условие установлено слева от ?, и значения помещены по сторонам от : |

Окончание табл. 3

| | | | |
|---|---|---|---|
| Delete | Оператор удаления | delete wd 2 | Удаляет объект, свойство или элемент массива |

Если несколько операторов совмещаются в одном выражении, порядок приоритета, начиная с высшего (выполненного первым), таков: ++, --, !, delete, *, /, %, +, -, ., «, », <, <=, >, >=, ==, !=, ===, !==, &, ^, |, &&, ||, ?:, =, +=,-=, *=, /=, &=. Чтобы обойти порядок приоритета, можно использовать круглые скобки. Другие примечания об операторах JavaScript:

– когда арифметический оператор (+) используется для совмещения текста (например, window.alert ("Михаил" + "Кузнецов"), чтобы вывести на экран диалоговое окно с текстом «Михаил Кузнецов») или текста и чисел (например, window.alert (12 + "дом, " + "ул. Ленина"), чтобы вывести диалоговое окно с текстом «12 дом, улица Ленина»), такая операция называется *конкатенация;*

– модуль числа (%) не дает процент; он возвращает остаток, часть, оставшуюся после деления операндов;

– знак равенства (=) не значит «равно», он значит «присвоить» или «заменить»;

– чтобы сравнить две переменные, можно использовать «равенство» (==) или «строгое равенство». Разница в том, что «==» проверяет, равны ли два значения, независимо от типа значений, а «===» проверяет не только равенство значений, но и равенство их типа;

– если a = "2" и b = 2.0, они не идентичны, потому что один операнд является строкой, а второй – числом с плавающей точкой, но они равны по своему значению.

**Инструкции и выражения**

JavaScript содержит либо комментарии, либо инструкции (команды). *Инструкция* – это строка кода, которая закончена и синтаксически правильна для выполнения задачи, обычно это все, что находится между знаками точки с запятой и от-

крывающим и закрывающим тегами JavaScript. Инструкцией чаще всего является строка кода, завершенная точкой с запятой, но можно использовать и несколько инструкций на одной строке или одну инструкцию на нескольких строках. Инструкции содержат одно или больше выражений.

*Выражения* – это все, чему назначены значения. *Значениями* является все, что можно присвоить переменной, поэтому значением может быть любой из типов данных: целые числа, числа с плавающей точкой, строки или булевы объекты. Хотя тип данных NULL является отсутствием значения, здесь он все же считается значением.

Выражения могут формировать блоки, используемые для создания других выражений. Например, a = 2 – это три выражения: 2, a и a = 2.

**Функции**

*Функция* – это часть скрипта, которая производит какое-либо действие и может быть многократно использована внутри большего скрипта. Некоторые функции требуют присвоения им значений или *аргументов,* которые функция использует, чтобы вернуть значение. Другие функции просто выводят значение при запросе.

**Глобальные функции**

У JavaScript нет больших библиотек внутренних глобальных функций для выполнения ежедневных заданий, которые не ориентированы на конкретный *объект.* Однако JavaScript содержит полный комплект *методов,* вызывающих функции, которые относятся к объектам (методы и объекты описаны далее). Наиболее часто используемые глобальные функции JavaScript перечислены в табл. 4.

Таблица 4

Некоторые глобальные функции JavaScript

| Функция | Описание |
|---|---|
| escape( ) | Возвращает шестнадцатеричный код строки |
| eval( ) | Вычисляет строку кода JavaScript без обращения к конкретному объекту |

Продолжение табл. 4

| | |
|---|---|
| isFinite( ) | Вычисляет аргумент, чтобы определить, является ли он конечным числом |
| isNaN( ) | Вычисляет аргумент, чтобы определить, не является ли он числом |
| parseFloat() | Анализирует строку аргумента и возвращает число с плавающей точкой |
| parseInt( ) | Анализирует строку аргумента и возвращает целое число |
| unescape( ) | Возвращает строку в стандарте ASCII для шестнадцатеричного кода |

**Пользовательские функции**

Пользовательские функции задаются ключевым словом function, используя следующую форму:

function ***имя(аргумент1, аргумент2,...)***
{
[любые инструкции JavaScript];
return ***значение;***
}
имя (*аргумент1, аргумент2,...*)

Функции без аргументов задаются назначением им имени и пустыми круглыми скобками, например:

function clearform()

Имя функции формируется по таким же правилам как и имя переменной.

Пример пользовательской функции, переводящей мили в километры приведен в коде ниже:

<**html**>
<**head**>

```
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1251">
<script language="javascript" type="text/javascript">
function tokm(miles)
{
        var km = miles * 1.6;
        return km;
}
</script>
</head>
 <body>
<script language="javascript" type="text/javascript">
document.write ("<b>Указания:</b><br>")
document.write ("Вам нужно проехать 2.4 мили или "+
tokm(2.4) + " км и повернуть направо.")
</script>
</body>
</html>
```

**Объекты, свойства и методы**

*Объекты* — это визуальные элементы веб-страницы: окна, кнопки, флажки и т. д., а также более абстрактные элементы (математические вычисления и массивы).

Для базового программирования хватает объектов, заложенных в JavaScript, но в языке также предусмотрена возможность создавать собственные объекты. Более того, у объектов могут быть подобъекты, и структура элементов страницы может со временем начать походить на иерархию генеалогического древа.

Заложенные объекты определяются ключевым именем, а пользовательским объектам назначаются уникальные имена, использующие те же правила, что и для назначения переменных. На веб-странице может быть несколько экземпляров объекта. Например, возьмем объект «окно». Каждое окно на веб-странице идентифицируется уникальным именем, а также ха-

рактеристиками, или *свойствами; методами,* определяющими действия, которые они могут совершить; и *событиями,* вызванными пользователем, которые влияют на него. Отношение объекта и его составляющих показано через использование *точечного синтаксиса,* посредством которого имя объекта отделяется от его составляющих точкой. Например window.alert() или document.write().

**Обработчики событий**

JavaScript позволяет легко использовать такие события, или *триггеры,* как щелчки кнопками мыши и открытие страниц, выполненных пользователем на веб-странице, с помощью заданных *обработчиков событий.* Обработчики событий распознают событие, происходящее на странице, и выполняют задачи, значительно повышая уровень интерактивности между вашей страницей и пользователем. Например, когда пользователь наводит указатель мыши на кнопку, появляется окно, предупреждающее о последствиях щелчка по ней (щелчок вызовет другое действие, назначенное другим обработчиком событий).

В табл. 5 перечислены обработчики событий JavaScript и триггеры, необходимые для запуска заданных действий.

Таблица 5

Обработчики событий JavaScript

| Обработчики событий | Триггеры событий |
|---|---|
| **onAbort** | Загрузка изображения прервана |
| **onBlur** | Убран фокус с элемента |
| **onChange** | Содержание формы изменено |
| **onClick** | На элемент нажали один раз |
| **onDblClick** | На элемент нажали дважды |
| **onDragDrop** | Объект перетащили в окно |
| **onError** | Возникла ошибка при загрузке веб-страницы или изображения |
| **onFocus** | Пользователь взял элемент в фокус |

Продолжение табл. 5

| | |
|---|---|
| **onKeyDown** | Пользователь нажал определенную клавишу |
| **onKeyPress** | Пользователь нажал и удерживает определенную клавишу |
| **onKeyUp** | Пользователь отпустил определенную клавишу |
| **onLoad** | Веб-страница закончила загружаться в браузер |
| **onMouseDown** | Пользователь щелкнул кнопкой мыши |
| **onMouseMove** | Пользователь двигает указатель мыши |
| **onMouseOut** | Пользователь убрал указатель мыши со ссылки |
| **onMouseOver** | Пользователь навел указатель мыши на ссылку |
| **onMouseUp** | Пользователь отжал кнопку мыши |
| **onMove** | Окно или фрейм перемещены |
| **onOpen** | Веб-страница открыта в браузере |
| **onReset** | Снят флажок с формы |
| **onResize** | Изменены размеры окна или фрейма |
| **onSelect** | Выбрано поле ввода в форме |
| **onSubmit** | Пользователь подтверждает введенную в форму информацию |
| **onUnload** | Пользователь открывает другую веб-страницу |

### 1.4. Создание массива и доступ к его элементам

Для создания массива и сохранения в нем каких-либо значений сначала следует объявить его имя (как и в случае с переменной), а затем приложить к имени список значений, разделенных запятыми: каждое значение представляет одну из единиц списка.

Чтобы обозначить массив, следует поместить список единиц в квадратные скобки – [ ]. Например, чтобы создать массив, содержащий названия семи дней недели, следует написать такой код:

```
var days = ['Mon', "rues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'];
```

Квадратные скобки очень важны. Они сообщают интерпретатору Script о том, что он имеет дело с массивом. Вы также можете создать пустой массив, не содержащий элементов:

```
var playList = [ ] ;
```

Элементы будут добавлены в него только по ходу работы программы.

Также можно встретить и другой способ создания массива – с использованием ключевого слова Array:

```
var days = new Array('Mon', 'Tues', 'Wed');
```

Данный способ допустим, но метод, предлагаемый здесь, использует более простой код.

Допускается сохранять в массиве любые смешанные значения. Другими ми, в одном и том же массиве могут быть числа, последовательности символов и булевы значения:

```
var prefs = [1, 223, 'www.oreilly.com', false];
```

Уникальный номер, называемый индексом, указывает позицию каждого элемента в массиве. Чтобы получить доступ к отдельно взятому элементу, следует использовать его индекс. Массивы индексируются с нуля, это означает, что первый элемент имеет индекс 0, а второй – 1.

Поскольку индексный номер последнего элемента массива всегда меньше общего числа элементов в массиве, вы должны знать, сколько единиц в массиве, чтобы получить доступ к последней из них. К счастью, эта задача несложна, поскольку среди свойств массива есть его длина – общее число единиц. Чтобы получить доступ к свойству «длина», следует набрать после имени массива точку и слово **length**, например, **days.length** возвращает количество единиц в массиве, называемом days (если создан иной массив, например, **playList**, то получите длину в таком виде: playList.length). Таким образом, можете получить доступ к значению, сохраненному в массиве последним:

```
days[days.length-1]
```

Также можете использовать переменную, содержащую номер в качестве индекса:

```
var i = 0;
alert(days[i]);
```

Последняя строка кода эквивалентна записи *alert(days[0])*.

## 1.5. Работа с элементами массива

**Добавление элемента в конец массива**

Чтобы добавить элемент в конец массива, возможно использовать значение индекса на один больше, чем последнее значение из списка массива. Допустим, создан массив под названием **properties**:

```
var properties =  ['red1',   '14px',   'Arial'];
```

В данный момент он содержит три единицы. Таким образом, последний элемент в этом массиве — properties[2]. Добавим еще один элемент в массив:

```
properties[3] = 'bold';
```

В данной строке кода четвертой позиции массива присваивается элемент 'bold', в результате получается массив из четырех элементов: [ 'red' , ' 14px' , ' Arial' , ' bold' ]. Следует заметить, что при добавлении нового элемента, используется значение индекса, равное общему числу элементов, находящихся в данный момент в массиве, поэтому, используя в качестве индекса свойство массива **length**, можно быть уверенным, что всегда добавляете новый элемент в конец массива. Например, возможно переписать код следующим образом:

```
properties[properties.length] = 'bold';
```

Также можно использовать команду **push ()**, которая добавит в массив то, что будет указано в круглых скобках. Рассмотрим этот способ добавления элемента в конец массива properties:

```
properties.push('bold') ;
```

В данном случае 'bold' добавляется в качестве нового элемента в конец массива. Можно использовать любой тип значения, например, строку, число, булевое значение и даже переменную.

Одно из преимуществ команды **push ()** состоит в том, что она позволяет добавлять в массив более одного элемента. Например, если требуется добавить три значения в конец массива properties:

```
properties.push('bold', 'italic', 'underlined');
```

**Добавление элемента в начало массива**

Если требуется добавить единицу в начало массива, можно использовать команду **unshift()**. Добавим значение 'bold' в начало массива **properties**:

```
var properties =  ['red',   '14px',   'Arial'];
properties.unshift('bold');
```

После выполнения этого кода массив properties будет содержать четыре элемента: ['bold',  'red',  '14px',  ' Arial']. Как и в случае с **push()**, возможно использовать unshift() для вставки нескольких единиц в начало массива:

```
properties.unshift('bold',   'italic',   'underlined');
```

Команды **push()** и **unshift()** возвращают значение. Точнее, когда **push()** и **unshift()** завершают выполнение своих задач, они возвращают количество элементов в дополненном массиве.

```
var p=  [0,1,2,3];
var totalItems = p.push(4,5);
```

После выполнения этого кода значение, сохраненное в totalItems, будет равно 6, поскольку в массиве p содержится шесть элементов.

**Удаление элементов из массива**

Если требуется удалить элемент из начала или из конца массива, то можно использовать методы **pop()** или **shift()**. Первый удаляет элемент из конца массива, второй – из начала.

Аналогично методам **push()** и **unshift()**, **pop()** и **shift()** возвращают значения после выполнения своих задач — удаления элементов из массива. На самом деле они возвращают уже удаленное значение. Например, приведенный ниже код удаляет значение и сохраняет его в переменной **removedItem**:

```
var  p =  [0,1,2,3];
var removedItem = p.pop();
```

Значение removedItem после выполнения этого кода – 3, а массив p теперь содержит [0,1,2].

**Добавление и удаление элементов с помощью команды splice()**

Техники, использовавшиеся в предыдущих разделах для добавления и удаления элементов, работают только в начале и в конце массива. Что делать, если нужно удалить элемент, занимающий третью позицию в массиве, либо добавить в третью позицию другой элемент?

JavaScript предлагает команду splice(), позволяющую добавлять элементы в массив и удалять их из него.

**Удаление элементов с помощью команды splice()**

Сообщите команде splice(), какой элемент и откуда должен быть удален (индексный номер первого элемента, подлежащего удалению), а также, сколько элементов следует удалить. Например, вы удаляете элементы из массива фруктов:

```
var fruit=['яблоко', 'груша', 'киви', 'гранат'];
```

Данный код создает массив из четырех элементов. Для удаления из массива груши и киви надо указать команде splice() начать со второго элемента (его индексный номер – 1) и удалить два элемента:

```
fruit.splice(1, 2)
```

Результат изображен на рис. 1.

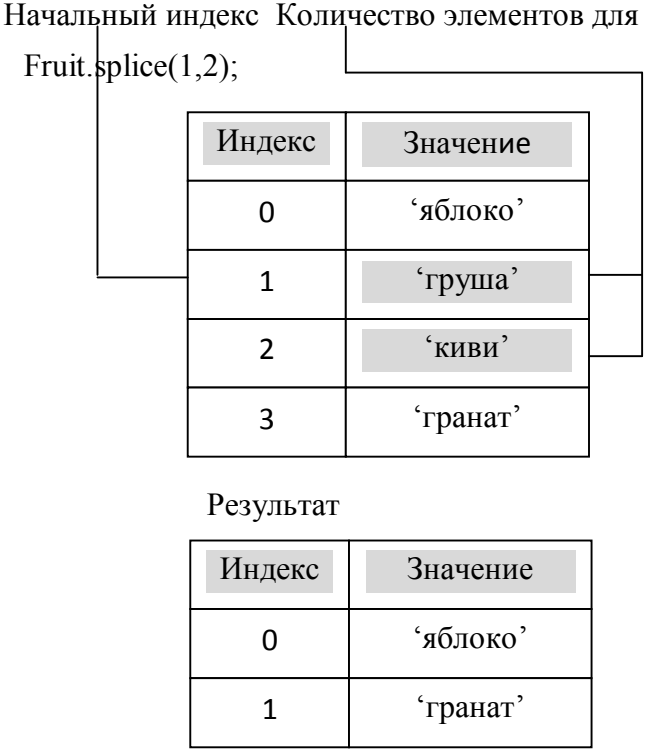


Рис. 1. Результат удаления элементов из массива
с помощью команды splice()

**Добавление элементов с помощью команды splice()**

Команда splice() позволяет также добавлять элементы в середину массива. Следует задать значение индекса, чтобы было известно, куда должны быть добавлены новые элементы; 0, чтобы обозначить, что ни один элемент не должен быть удален; список элементов, которые требуется вставить: одно и более значений, разделенных запятыми. Например, рассмотрим уже использовавшийся массив fruit.

```
var fruit=['яблоко', 'груша', 'киви', 'гранат'];
```

Если требуется добавить два элемента между грушей и киви в этот список, то можно использовать splice() следующим образом:

fruit.splice(2, 0, 'виноград', 'апельсин');

Этот код добавляет две строки символов – 'виноград' и 'апельсин', начиная с индекса 2 (рис. 2).

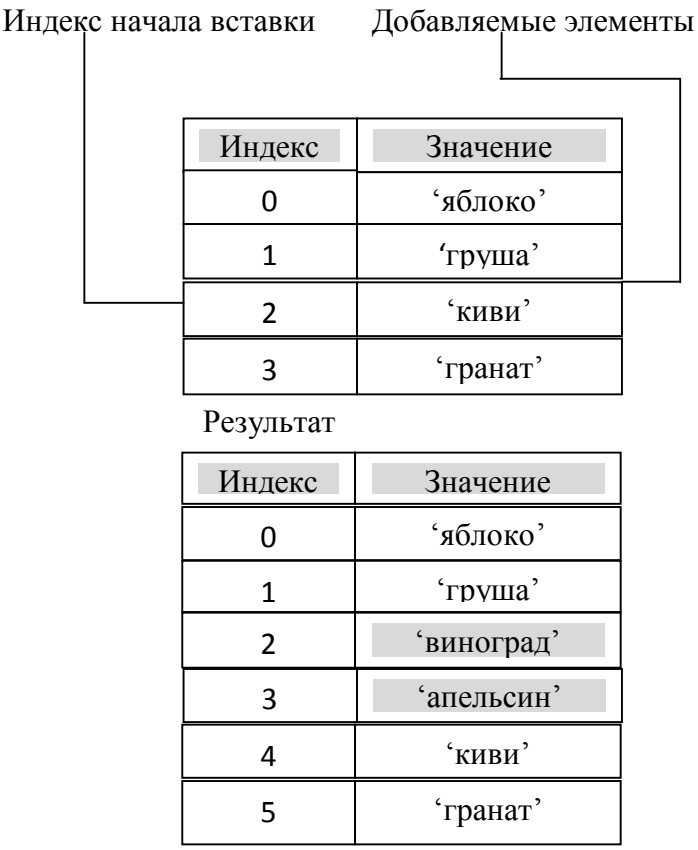


Рис. 2. Результат добавления элементов

**Замена элементов с помощью команды splice()**

Возможно добавить элементы в массив, и удалить из него элементы в ходе одной и той же операции. Этот маневр применяется в том случае, когда требуется заменить один или более элементов массива новыми.

Алгоритм тот же, что и при добавлении элемента, но во втором фрагменте информации указывается не 0, а количество элементов, которое следует удалить. Итак, снова воспользуемся массивом с фруктами:

```
var fruit=['яблоко', 'груша', 'киви', 'гранат'];
```

Допустим, требуется заменить киви и гранат виноградом и апельсином. Тогда следует записать следующую конструкцию:

```
fruit.splice(2, 2, 'виноград', 'апельсин');
```

В данном случае первая цифра 2 обозначает позицию в индексе, с которой необходимо начать, вторая цифра 2 указывает, сколько элементов удалить, а дальше указывается какие элементы надо добавить для замены (рис. 3).

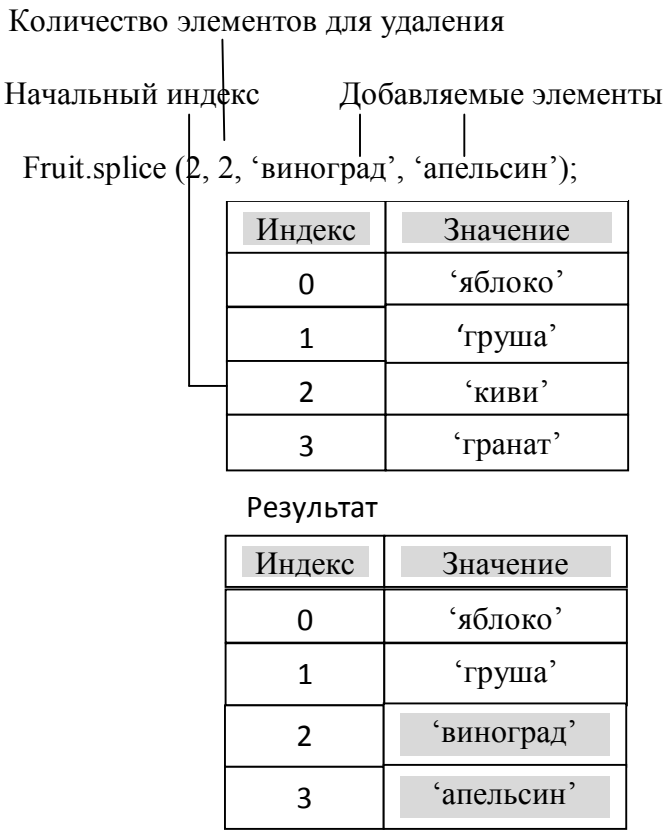


Рис. 3. Результат замены элементов массива
с помощью команды splice()

## 1.6. Условные выражения

**Конструкция if/else**

Инструкция **if/else** – главная конструкция для принятия решений в JavaScript. Она позволяет указать, что если (if) какое-то выражение является истинным (TRUE), будет выполнена определенная группа инструкций, в противном случае (else) выполнится другая группа. Это выглядит следующим образом:

```
if (условие) {
        группа операторов 1 (выполняется если условие – TRUE)
}
else {
        группа операторов 2 (выполняется если условие – FALSE)
}
```

Часть, начинающаяся с **else** является опциональной.

Многие условные выражения определяют соотношение между двумя элементами: их равенство, один больше другого или один меньше другого. Следует помнить, что при проверке равенства необходимо использовать двойной знак равенства (==), а не одинарный, который означает присваивание.

**Конструкция switch**

Инструкция **switch** похожа на инструкции типа if/else. Она используется при необходимости сравнить одну переменную с несколькими значениями и сделать, что-то зависящее от самого значения. Вместе со **switch** задействуются следующие дополнительные ключевые слова: **case, break** и **default**. Это выглядит следующим образом:

```
switch (avariable) {
case “1” :
группа операторов 1;
break;
case “2” :
группа операторов 2;
break;
case “3” :
группа операторов 3;
break;
default:
```

группа операторов (выполняется, если все case возвращают FALSE);
}

Каждое выражение **case** в инструкции **switch** сравнивает переменную **switch** со значением **case**, которое может быть строкой. Если данные значения равны, выполняются инструкции, следующие за выражением case, после чего выражение **break** отсылает скрипт к первой инструкции после закрывающей фигурной скобки **switch**. Если ни одно из выражений case не было истинным, выполняются инструкции, следующие за выражением **default**, а скрипт завершает инструкцию **switch**.

## 1.7. Работа с циклами

На языке программистов многократное выполнение одной и той же задачи называется циклом. Поскольку циклы очень часто встречаются в программировании, JavaScript предлагает несколько их типов. Все они делают одно и то же, но немного разными способами.

**Циклы While**

Цикл **while** повторяет отрезок кода, пока остается истинным определенное условие. Базовая структура цикла **while** такова:

```
while (условие)  {
//   повторяющийся JavaScript
}
```

Первая строка вводит утверждение **while**. Условие помещается в скобки, следующие за ключевым словом **while**. Интерпретатор JavaScript выполняет весь код, находящийся в скобках, если условие истинно.

Однако в отличие от условного выражения, когда интерпретатор JavaScript достигает закрывающей скобки утвер-

ждения **while**, он вместо перехода к следующей строке программы возвращается к началу цикла **while** и тестирует условие повторно. Если условие вновь оказывается верным, интерпретатор опять выполняет код JavaScript. Процесс продолжается до тех пор, пока условие не станет ложным; после этого программа переходит к выполнению выражения, следующего за циклом.

Допустим, требуется напечатать на странице числа от 1 до 5. Рассмотрим один из способов сделать это:

```
document.write('Number 1 <br>') ;
document.write('Number 2 <br>') ;
document.write('Number 3 <br>') ;
document.write('Number 4 <br>') ;
document.write('Number 5 <br>') ;
```

Следует заметить, что строки кода почти идентичны: от строки к строке изменяется только число. В данной ситуации для достижения той же цели более эффективно использовать цикл:

```
var num = 1;
while  (num <= 5)   {
document.write('Number ' + num +  '<br>');
num++;
}
```

Первая строка кода var num = 1; не является частью цикла **while**, она устанавливает переменную для сохранения числа, печатаемого на странице.

Вторая строка – это начало цикла. Она задает условие проверки. Пока число, сохраненное в переменной, меньше или равно 5, код в скобках выполняется. Когда условие теста выставляется в первый раз, значение этой переменно равно 1, испытание пройдено (так как 1 меньше 5), выполняется команда

document.write(), записывая на страницу 'Number l<br>' (<br>, – это HTML-тег, означающий переход на другую строку, он предназначен для того, чтобы гарантировать: каждая строка на веб-странице печатается отдельно).

Последняя строка цикла num++ очень важна. Она не просто увеличивает значение переменной num на 1 так, что на страницу выводится следующее число (например, 2), но также позволяет условию проверки стать ложным.

Поскольку код JavaScript в утверждении **while** повторяется до тех пор, пока условие верно, следует изменить один из элементов условия так, чтобы условие стало ложным, цикл прекратился и можно было перейти к следующей части скрипта. Если условие теста никогда не становится ложным, имеет место так называемый бесконечный цикл – программа, которая никогда не закончится.

Этот простой пример также демонстрирует гибкость циклов. Например, вы хотели записать номера не от 1 до 5, а от 1 до 100. Вместо добавления множества дополнительных строк команды document.write() можно просто выполнить условие теста следующим образом:

```
var num = 1;
while (num <= 100)   {
document .write ( 'Number ' + num + '<br>');
num = num ++;
}
```

Теперь цикл будет выполнен 100 раз, на веб-странице появится 100 строк.

**Циклы for**

JavaScript предлагает еще один тип цикла – **for**. Он немного компактнее, но чуть более сложен. Циклы **for** обычно используются для повторения серии шагов определенное количество раз. Поэтому они часто включают особую переменную

цикла, условие и способ изменения переменной цикла. Во многих случаях цикл **for** помогает достичь тех же целей, что и цикл **while**, но с использованием меньшего количества строк кода. Вот, например, код предыдущего примера с использованием цикла **for**:

```
for  (var num=l; num<=100;  num++)   {
document.write('Number '  + num +  '<br>');
}
```

В первой части (var num = 1;) инициализируется переменная цикла. Это происходит только один раз в самом начале работы цикла. Вторая часть условия предназначена для тестирования того, нужно ли в очередной раз выполнять код. Третья часть – действие, происходящее в конце каждого оборота цикла (обычно это изменение значения переменной цикла) до тех пор, пока условие теста не станет ложным и цикл не закончится.

Поскольку циклы **for** являются простым способом повторять серии шагов определенное количество раз, они действительно хороши для работы с элементами массива. Например, вывод элементов массива на экран может быть записан с использованием цикла for следующим образом:

```
var days =  [ 'Monday' ,  'Tuesday' ,  'Wednesday' , 'Thursday',
'Friday',  'Saturday',  'Sunday'];
for (var i = 0;  i<days. length;  i++)   {
document.write(days[i]  + ',   ');
}
```

В приведенных примерах значение переменной изменялось до определенного числа, а затем цикл останавливался, но возможен и обратный процесс. Например, если требуется вывести на печать элементы массива в обратном порядке (иначе говоря, последний элемент массива печатается первым). Можно сделать это так:

```
var example =  ['первый', 'второй', 'третий', 'последний'];
for (var j = example.length ;  j > 0;  j--)   {
document.write(example[j-1]  +  '<br>');
}
```

В данном примере начальное значение переменной цикла – общее количество единиц в массиве (4). После каждого прохождения цикла вы проверяете, является ли значение j большим, чем 0. Если это так, то код в фигурных скобках выполняется. Затем из j вычитается 1 (j--), и проверка проводится снова. Единственная сложность – способ доступа к элементам массива, применяемый в программе (example[j-1]). Поскольку массивы начинаются с индекса 0, общее количество элементов в массиве на один меньше, чем значение индекса последнего элемента. Здесь начальным значением j является общее число элементов массива, поэтому, чтобы достичь последнего элемента, нужно вычитать по 1 после каждого выполнения цикла.

**Циклы Do/While**

Есть еще один, менее распространенный тип циклов, известный как do/while. Они работают практически так же, как циклы while. Базовая структура выглядит так:

```
do {
// повторяющийся javascrip
} while (условие);
```

В циклах данного типа условный тест происходит в конце, после того как цикл выполнен. В результате выполнение кода JavaScript, находящегося в фигурных скобках, происходит хотя бы один раз. Даже если условие уже не выполняется, тест более не проводится после выполнения кода.

## 1.8. Работа со строками

Строки – один из основных типов данных, с которым приходится работать (пользовательский ввод с полей формы, путь к изображению, URL, HTML, который можно изменить на странице). Следовательно, в JavaScript имеется много методов работы со строками и управления ими.

**Определение длины строки**

Для получения длины строки можно использовать свойство length, которое возвращает длину строки в символах. Для определения числа символов в строке добавьте точку после имени переменной, причем за точкой следует свойство length: name.length.

Например, необходимо убедиться, что в пароле определенное количество символов. Можно использовать для этого условное выражение:

```
var password =  ‘new_pass’;
if  (password.length <= 6)   {
alert('Данный пароль слишком короткий.');
} else if  (password.length > 15)   {
alert('Данный пароль слишком длинный');
}
```

**Изменение регистра строки**

JavaScript предлагает два метода перевода целых строк в верхний или нижний регистр.

Метод **toUpperCase()** переводит все содержимое строки в верхний регистр. Например, следующий код переводит слово ‘Hello’ в верхний регистр и выводит его в виде сообщения ‘HELLO’:

```
var greetings=’Hello’;
alert(greetings.toUpperCase());
```

Чтобы перевести всю строку в нижний регистр, следует использовать метод **toLowerCase()**:

```
var greetings=’Hello’;
alert(greetings.toUpperCase());  //  hello
```

Ни один из этих методов на самом деле не изменяет самой строки, сохраненной в переменной, они просто возвращают строку либо только в нижнем, либо только в верхнем регистре. В рассмотренном примере greetings по-прежнему содержит «Hello» даже после появления предупреждения.

**Поиск в строке: метод indexOf()**

JavaScript предлагает несколько способов поиска слов, чисел или других серий знаков в строках.

Один из методов поиска строки – **indexOf()**. Порядок такой: после имени строковой переменной string печатается точка, затем **indexOf()** и в скобки вставляется искомая строка:

```
string.indexOf('строка, которую надо найти')
```

Метод **indexOf()** возвращает число: если искомая строка не найдена, то метод возвращает -1.

Когда метод **indexOf()** обнаруживает искомую строку, он возвращает число, равное начальной позиции строки. Рассмотрим пример, который послужит пояснением:

```
var quote = 'быть или не быть';
var searchPosition = quote.indexOf('Быть');  // возвращает 0
```

В данном случае **indexOf()** ищет фрагмент **быть** в строке **быть или не быть**. Длинная строка начинается с **быть**, поэтому **indexOf()** находит искомый фрагмент в самом начале. Первая позиция считается 0, вторая буква («ы») – 1, а третья (в данном случае «т») – 2.

Метод **indexOf()** начинает поиск с начала строки. Можно искать и с конца строки, используя метод lastIndexOf().

Например, в цитате из Шекспира слово **быть** встречается два раза, поэтому можно определить первое **быть** с помощью метода **indexOf()**, а второе – с использованием **lastIndexOf()**:

```
var quote = "быть или не быть";
var firstPosition = quote.indexOf('быть');  // возвращает 0
var lastPosition = quote.lastIndexOf('быть');  // возвращает 12
```

Результат применения этих двух методов показан на рис. 4. В обоих случаях, если **быть** не существует в данной строке, результат будет -1, и если поиском слова занимается только indexOf() или lastIndexOf(), то будет возвращено одно и то же значение – начальная позиция искомой строки в рамках более длинной строки.

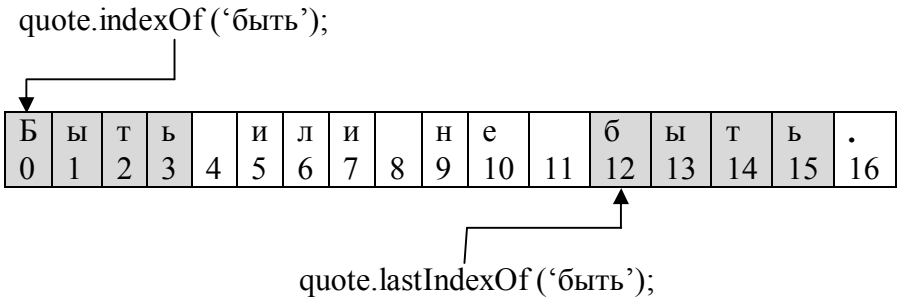


Рис. 4. Иллюстрация работы методов indexOf() и lastIndexOf()

**Извлечение части строки с помощью метода slice()**

Чтобы извлечь часть строки, следует использовать метод slice(), который возвращает результат извлечения. Например, имеется строка http://www.sawmac.com и требуется исключить часть http://. Один из способов сделать это — извлечь часть строки, следующей за http://, вот так:

```
var url =  'http://www.sawmac.com';
var domain = url.slice(7) ;  // www.sawmac.com
```

Метод slice() требует в качестве аргумента число, которое присваивается индексу первого символа извлекаемой строки (рис. 5). В данном примере (url.slice(7)) 7 обозначает восьмую букву в строке (не следует забывать, что первой букве соответствует индекс 0). Данный метод возвращает все символы, начиная со стартовой позиции, исходя из переданного в качестве аргумента индекса и до конца строки.

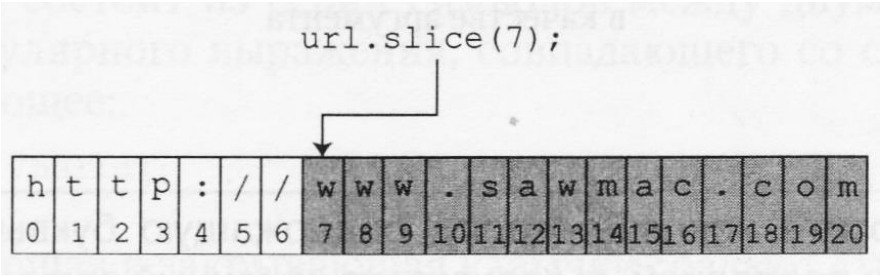


Рис. 5. Иллюстрация работы метода slice()

Также возможно извлечь определенное количество символов из строки, присвоив методу slice() второй аргумент. Вот базовая структура метода slice() в данном случае:

string.slice(start, end);

Стартовое значение – это число, указывающее первый знак извлекаемой строки; конечное значение может запутать, так как это не позиция последней буквы извлекаемой строки, а позиция последней буквы плюс 1. Например, если требуется извлечь первые 4 буквы строки **быть или не быть**, следует указать 0 как первый аргумент, а 4 – как второй. Как видно на рис. 3, 0 – это первая буква в строке, а 4 – пятая, но последняя указанная буква не извлекается из строки. Другими словами символ, указанный в качестве второго аргумента, никогда не извлекается из строки.

**Замечание.** Если требуется извлечь из строки определенное количество знаков можно просто добавить это число к стартовому значению. Например, чтобы вернуть первые 10 букв строки, первый аргумент будет 0 (первая буква) а второй – 0 + 10 или просто 10: slice(0,10).

Также возможно указывать отрицательные числа, например, quote.slice(-6,-1). Отрицательное число считается с конца строки, как показано на рис. 6.

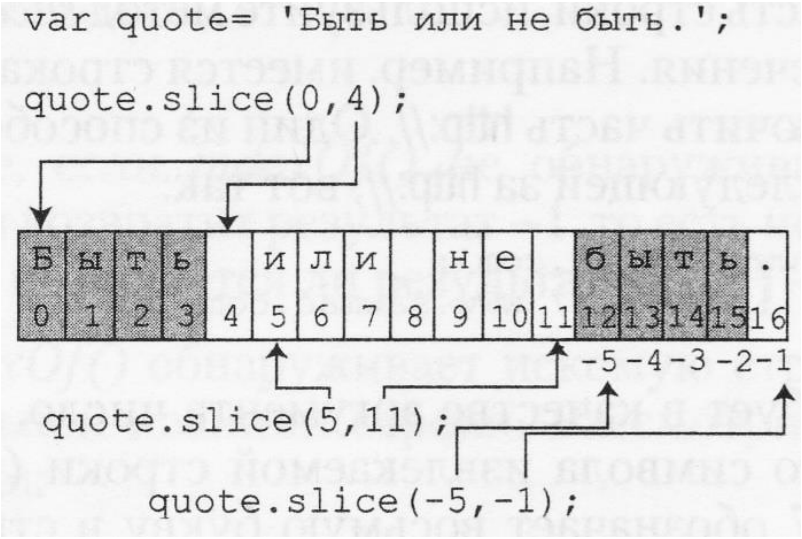


Рис. 6. Иллюстрация работы метода slice() с двумя аргументами

**Замечание.** Если требуется извлечь строку, содержащую буквы, начиная с 6-й с конца и до последней, можно просто опустить конечный аргумент:

quote.slice(-6);

## 1.9. Работа с числами

Числа – важная часть программирования. Они помогают выполнять такие задачи, как подсчет итоговой суммы продаж, определение расстояния между двумя точками, имитация бросания игральных костей путем генерирования случайных чисел от 1 до 6. В JavaScript существует множество различных способов работы с числами.

**Замена строки числом**

При создании переменной, возможно сохранить в ней число:

```
var a = 3.25;
```

Однако случается, что число представляется в виде строк, например, при использовании метода prompt() для получения пользовательского ввода. Если кто-нибудь напечатает 3.25, в итоге получится строка, содержащая число. Иначе говоря, в результате получается 3.25 (строка), а не 3.25 (число).

Для преобразования строки в число JavaScript предлагает несколько способов.

**Number()** преобразует любую переданную ему строку в число:

```
var a = '3';
a = Number(a);  // a стало теперь числом 3
```

Проблема сложения двух строк, содержащих числа может быть решена так:

```
var a = '3';
var b = '4' ;
var total = Number (a)  + Number (b);  // 7
```

Более быстрый способ предоставляет оператор +, выполняющий то же действие, что и метод **Number()**. Просто следует добавить + перед переменной, содержащей строку, интерпретатор JavaScript превратит строку в число:

```
var a = '3';
```

```
var b = '4' ;
var total = +a + +b // 7
```

Недостатком каждого из этих способов является то, что, если строка содержит что-либо, кроме чисел (точку, знаки + или – и т. д), в итоге получается значение JavaScript NaN, означающее «не-число».

Метод **parseInt()** также превращает строку в число. Однако, в отличие Number(), **parseInt()** попытается превратить в число даже буквенную последовательность, если она начинается с чисел. Данная команда может быть удобна, если имеется строка вроде «20 лет» в качестве ответа на вопрос о чьем-то возрасте:

```
var age =  '20 лет';
age = parseInt(age,  10);  // 20
```

Метод parseInt() ищет в начале последовательности число, символ + или - и продолжает искать числа, если встречает не-число. Итак, в примере, приведенном выше, он возвращает число 20 и игнорирует остаток последовательности: «лет».

Второй аргумент метода parseInt() указывает, в какой системе счисления представлено число для поиска. Если указать 2, то будет искаться число в двоичной системе счисления, 8 – в восьмеричной и т.д.

**parseFloat()** похож на **parseInt()**, но он используется, если строка может содержать разделительную точку десятичной дроби. Например, если имеется строка вида «4.5 акров», то можно использовать **parseFloat()** для возвращения всего значения, включая десятичные знаки:

```
var space =  '4.5 акров';
space = parseFloat(space);  // 4.5
```

Если бы использовался метод **parseInt()** в примере, рассмотренном выше, то в итоге получилось бы просто число 4, поскольку **parseInt()** возвращает только целые числа.

Чтобы подтвердить, что строка является числом, следует использовать метод **isNaN()**. Он берет строку в качестве аргумента и тестирует, является ли она числом. Если строка содержит что-нибудь, кроме знаков «плюс» или «минус» (для положительных и отрицательных чисел), за которыми следуют числа и возможные десятичные значения, она считается нечислом. Таким образом, «-23.25» – это число, а «24 км» – нет. Данный метод возвращает значение «истина» (если строка не является числом) или «ложь» (если она является числом).

**Округление чисел**

Возможно округлять число, используя метод **round()** с объектом **Math**:

```
Math.round(number)
```

Число (или переменная, содержащая число) передается методу **round()**, и он возвращает целое число. Если первоначальное число содержит после десятичного знака цифры до 5, оно округляется в меньшую сторону, например, 4,4 округляется до 4, а 4,5 – в большую сторону, до 5.

```
var decimalNum = 10.25;
var roundedNum = Math.round(decimalNum) ;  // 10
```

**Замечание**

JavaScript предлагает еще два метода округления чисел: **Math.ceil()** и **Math.floor()**, которые схожи с методом **Math.round()**. Однако, **Math.ceil()** всегда округляет число в большую сторону (например, Math.ceil(4.0001) возвращает 5), тогда как **Math.floor()** всегда округляет число в меньшую сторону: Math.floor(4.99999) возвращает 4. Рекомендуется использовать мнемоническое правило: ceiling (потолок) — это вверху, а floor (пол) — внизу.

**Форматирование вещественных чисел**

В данном случае JavaScript предлагает метод **toFixed()**, который позволяет преобразовывать число в строку, совпадающую с желаемым числом с десятичными знаками. Чтобы

пользоваться им, следует добавить после числа точку (или после имени переменной, содержащей число), а затем toFixed(2):

```
var cost = 10;
var printCost = '$'  + cost.toFixed(2);   //  $10.00
```

Число, присваиваемое методу toFixed(), определяет, сколько десятичных знаков нужно указать. В случае с валютой следует использовать 2, чтобы получить в итоге числа: 10.00 или 9.90; используя 3, в итоге получается 3 десятичных знака, например, 10.000 или 9.900.

Если число имеет больше десятичных знаков, чем задано в методе, оно округляется до количества указанных десятичных знаков. Например:

```
var cost = 10.289;
var printCost =  '$'  + cost.toFixed(2) ;   // $10.29
```

В данном примере 10,289 округляется до 10,29.

**Генерирование случайного числа**

JavaScript предлагает метод Math.random() для генерирования случайных чисел. Он возвращает случайно сгенерированное число в интервале от 0 до 1 (например, .9716907176080688 или .10345038010895868). Можно выполнять или тестировать простые математические операции, генерируя целые числа от 0 и выше. Например, для генерирования чисел от 0 до 9 следует использовать следующий код:

```
Math.floor(Math.random()*10);
```

Данный код разделяется на две части. Math.random()*10 генерирует случайное число между 0 и 10. Таким образом, может быть сгенерировано число 4.190788392268892; и поскольку случайное число находится между 0 и 10, оно никогда не равно 10. Чтобы получить целое число, случайный результат присваивается методу Math.floor(), округляющему все десятичные знаки в меньшую сторону до ближайшего целого числа. Таким

образом, 3.4448588848 становится 3, а .1111939498984 становится 0.

Если требуется получить случайное число от 1 до другого числа, следует просто умножить Random() на большее число этой области и добавить к целому 1. Например, если необходимо сымитировать броски игральной кости, чтобы получить числа от 1 до 6, то можно написать следующий код:

```
var roll = Math.floor(Math.random()*6 +1);     // 1,2,3,4,5 или 6
```

**Функция для выбора случайного числа**

Если приходится часто пользоваться случайными числами, то можно применять простую функцию, помогающую при выборе случайного числа между любыми двумя числами, например, между 1 и 6 или 100 и 1000. Следующая функция вызывается с использованием двух аргументов; первый – минимальное возможное значение (например, 1), второй – максимальное возможное значение (например, 6):

```
function rndNum(from,  to)   {
return Math.floor((Math.random()*(to - from +1))  + from);
}
```

Для использования этой функции следует добавить ее на веб-страницу и вызвать ее:

```
var diceRoll = rndNum(1,6);  // получает число между 1 и 6
```

## 1.10. Работа с датами и временем

Если требуется отслеживать актуальную дату или время, то можно воспользоваться специальным объектом JavaScript – Date (Дата). Он позволит определять год, месяц, день недели, час и даже больше. Для его использования следует создать переменную и сохранить в ней новый объект Date:

```
var now = new Date();
```

Команда new Date() создает объект Date, в котором содержатся текущие дата и время. Однажды создав его, можно получать доступ к различным образцам даты и времени, используя относящиеся к Date методы, перечисленные в табл. 6. Например, для получения текущего года можно использовать метод getFullYear():

```
var now = new Date();
var year = now.getFullYear();
```

Таблица 6

Методы для доступа к данным объекта Date

| **Метод** | **Что он возвращает** |
|---|---|
| getFullYear() | Год. Например, 2008 |
| getMonth() | Месяц как целое число между 0 и 11:0 – январь, 11 – декабрь |
| getDate() | День месяца как число от 1 до 31 |
| getDay() | День недели как число от 0 до 6: 0 – воскресенье, 6 – суббота |
| getHours() | Количество часов по 24-часовому циферблату (число от 0 до 23). Например, 11 вечера = 23 |
| getMinutes() | Число минут от 0 до 59 |
| getSeconds() | Число секунд от 0 до 59 |
| getTime() | Общее количество секунд, начиная с полуночи 1.01.1970 |

**Получение месяца**

Чтобы получить месяц от объекта Date, следует использовать метод getMonth(), возвращающий номер месяца:

```
var now = new Date();
var month = now.getMonth();
```

Однако вместо возвращения номера месяца (1 – это январь), данный метод возвращает число на 1 меньше. Например, январь – это 0, февраль – 1 и т. д. Если нужно возвратить номер, совпадающий с обычным пониманием месяцев, просто следует добавить 1:

```
var now = new Date();
var month = now.getMonth()+1;      // соответствует реальной нумерации месяца
```

В JavaScript не существует встроенной команды, сообщающей имя месяца. К счастью, способ нумерации в JavaScript удобен, когда нужно определить точное название месяца. Можно достичь этого, создав массив с названиями месяцев, а затем получать доступ к имени, используя его индекс:

```
var months = ['Январь', 'Февраль', 'Март', 'Апрель', 'Май', 'Июнь', 'Июль', 'Август', 'Сентябрь', 'Октябрь', 'Ноябрь', 'Декабрь'];
var now = new Date ();
var months = months[now.getMonth()];
```

В первой строке создается массив с двенадцатью названиями месяцев в том порядке, в котором они идут (январь – декабрь). Следует помнить, что для доступа к элементу массива используется индекс и что массивы нумеруются с 0. Итак, для доступа к первому элементу массива с названиями месяцев следует использовать months[0]. Таким образом, с помощью getMonth() можно получить число, используемое как индекс для массива месяцев, посредством чего легко узнать название этого месяца.

**Получение дня недели**

Метод getDay() возвращает день недели. Как и в методе getMonth(), интерпретатор JavaScript возвращает число на 1 меньше, чем номер дня: 0 – это воскресенье, первый день неде-

ли по принятой на Западе традиции, 6 – суббота. Поскольку для обычных людей название дня недели обычно важнее, чем его номер, можно использовать массив для сохранения названий дней и метод getDay() для доступа к отдельному дню в массиве:

```
var days = [ 'Воскресенье', 'Понедельник', 'Вторник', 'Среда',
' Четверг' , ' Пятница' , ' Суббота' ];
var now = new Date();
var dayOfWeek = days[now.getDay()];
```

**Получение времени**

Объект Date также содержит актуальное время, поэтому можно отобразить это время на веб-странице или использовать его, чтобы узнать, в какое время суток посетитель просматривал страницу.

Методы getHours(), getMinutes() и getSeconds() можно использовать для возвращения часов, минут и секунд. То есть, чтобы отобразить время на веб-странице, можно добавить следующий код в HTML туда, где требуется видеть время:

```
var now = new Date();
var hours = now.getHours();
var minutes = now.getMinutes() ;
var seconds = now.getSeconds();
document .write (hours + ":" + minutes + ":" + seconds);
```

Данный код производит вывод 6:35:56 для обозначения 6 часов, 35 минут и 56 секунд.

**Создание несегодняшней даты**

Используя метод Date(), также можно указывать дату и время в будущем и прошлом. Основной формат таков:

new Date(год, месяц, день, час, минута, секунда, миллисекунда);

Например, чтобы создать Date для полудня 1 января 2010 г., можно написать следующее:

var ny2010 = new Date(2010, 0, l, 12, 0, 0, 0);

Данный код переводится как «Создать новый объект Date для времени 1.01.2010, 12:00:00:00». Необходимыми параметрами являются только год и месяц. Если не требуется указывать точное время, можно опустить миллисекунды, секунды, минуты и т. д. Например, создание объекта «1 января 2010 года» выглядит так:

var ny2010 = new Date(2010, 0, 1);

**Создание даты, являющейся определенным днем недели**

Интерпретатор JavaScript считает дату количеством миллисекунд, истекших с 1 января 1970 г. Другими словами, создание даты – это присвоение значения, равного количеству миллисекунд для этой даты:

new Date(количество миллисекунд);

Итак, другой способ создания даты для 1.01.2010 таков:

var ny2010 = new Date(1262332800000);

Конечно, большинство людей – не живые калькуляторы, и дату человек воспринимает не так. Однако миллисекунды очень удобны при создании даты, отстоящей от другой даты на конкретный период времени. Например, при настройке cookie с использованием JavaScript следует указать время, через которое cookie должно быть удалено из браузера посетителя. Чтобы

гарантировать, что cookie исчезнет через неделю, необходимо указать одну неделю, начиная с сегодняшнего дня.

Для создания даты, отстоящей от сегодняшнего дня на одну неделю можно сделать следующее:

```
var now = new Date();                    // текущий момент
var nowMS =  now.getTime(); // миллисекунды, соответствующие
// текущему моменту
var week = 1000*60*60*24*7;   // вычисление количества
// миллисекунд в неделю
 var oneWeekFromNow = new Date(nowMS + week);
```

В первой строке текущие дата и время сохраняются в переменной now. Далее метод getTime() извлекает число миллисекунд, истекших с 1.01.1970 до сегодняшнего дня. Третья строка считает общее количество миллисекунд в неделе (1000 миллисекунд * 60 секунд * 60 минут * 24 часа * 7 дней). В итоге код создает новую переменную, добавляя число миллисекунд в неделе к сегодняшнему дню.

## 1.11. Регулярные выражения

**Нахождение шаблонов в строках**

Иногда необходимо найти строку, но не точное значение, а особый шаблон. Например, если требует убедиться в том, что пользователь, заполнявший форму заказа, указал номер телефона в правильном формате, то следует не искать определенный номер телефона, например, 273-98-25, а определенный общий шаблон: три числа, дефис, два числа, дефис, два числа. Необходимо проверить, какое значение ввел пользователь, и если оно подходит под данный шаблон (например, 345-56-89, 223-90-78, 655-78-67 и т. д.), то все хорошо. Но если введенная информация не подходит под этот образец (например, посетитель написал dhrt565dfdf), то можно отправить ему со-

общение: «Введен несуществующий номер телефона» или «Введен номер телефона в неверном формате».

JavaScript позволяет использовать регулярные выражения, чтобы находить в строках шаблоны. Регулярное выражение – это серия символов, образец шаблона, который требуется найти.

Для создания шаблона используются символы, как *, +, ? и \. Интерпретатор JavaScript сопоставляет их с реальными символами из строки: числами, буквами и т. д.

**Замечание.** В многих случаях часто используются сокращение regex в качестве сокращения для regular expression (регулярное выражение).

Для создания регулярного выражения в JavaScript необходимо создать объект, который состоит из серии символов между двумя слэшами. Например, для регулярного выражения, совпадающего со словом hello, следует напечатать программный код:

```
var myMatch = /hello/;
```

Как открывающая и закрывающая кавычки создают строку, открывающий и закрывающий слэш и создают регулярное выражение.

Существует несколько методов, используемых при работе со строками, которые имеют преимущество над регулярными. Базовый метод – search(). Он работает, как метод indexOf(), но вместо того, чтобы пытаться найти одну строку в другой, более крупной, он ищет в строке шаблон. Например, вы хотите отыскать «быть» в строке «быть или не быть». Ранее это делалось с помощью метода indexOf(). Можно сделать то же самое с помощью регулярного выражения:

```
var myRegEx = /быть/;   // не надо заключать в кавычки
var quote =  'быть или не быть.';
var foundPosition = quote.search(myRegEx);    // возвращает 0
```

Если search() находит совпадение, он возвращает позицию первой совпавшей буквы, а если не находит, то возвращает -1. Так, в примере, приведенном выше, переменная foundPosition равна 0, поскольку «быть» начинается с самого начала строки (с первой буквы).

Метод indexOf() работает практически так же. Преимущество регулярных выражений – в том, что он могут находить в строке шаблон, то есть выполнять гораздо более сложные операции по сравнению с методом indexOf(), который всегда ищет точное совпадение двух строк.

**Создание регулярных выражений**

При том, что регулярное выражение может быть создано из одного или нескольких слов, более часто используют комбинацию букв и специальных символов, чтобы определить шаблон, соответствие которому требуется найти. Регулярные выражения могут содержать различные символы для обозначения разных типов знаков. Например, точка (.) означает отдельный знак, любой; \w соответствует любой букве или числу (но не пробелам или символам $ или %). В табл. 7 приведены символы, наиболее часто используемые при поиске по шаблонам.

Таблица 7

Символы, часто используемые в регулярных выражениях

| Символ | Значение |
|---|---|
| . | Один любой символ. Это может быть буква, число, пробел или другой знак. |
| \w | Любой знак, который может входить в слово. Это может быть буква (в обоих регистрах, цифра от 0 до 9, знак подчеркивания (_). |
| \W | Любой знак, который НЕ может входить в слово. То есть любые знаки, не соответствующие \w. |
| \d | Любая цифра от 0 до 9. |
| \D | Любой символ, кроме цифры. Протовоположно \d. |

Продолжение табл. 7

| | |
|---|---|
| \s | Пробел, знак табуляции, знак возврата каретки и знак новой строки. |
| \S | Все знаки, кроме знака табуляции, знака возврата каретки **и** знака новой строки. Противоположно \s. |
| ^ | Начало строки. Полезно, если обязательным условием является то, что впереди искомой подстроки нет символов (то есть она начинает строку). |
| $ | Конец строки. Полезно, если обязательным условием является то, что искомая подстрока должна заканчивать строку. К примеру, **/com$/** соответствует подстрока **«com»,** но только когда она занимает последние три символа строки. То есть **/com$/** соответствует подстроке **«com»** в строке «infocom», но не в строке «communication». |
| \b | Пробел, начало строки, конец строки, любой другой символ, не являющийся буквой или цифрой (к примеру, +,= или'). Можно использовать \b для обозначения начала и **конца** слова, даже если слово находится в начале или конце строки. |
| [] | Любой символ из помещенных в скобки. К примеру,[aeoun] совпадет с любой из букв в квадратных скобках. Для обозначения последовательностей символов **можно использовать** тире. Так, запись [a-z] ссылается на любую букву латинского алфавита в нижнем регистре. Запись [0-9] ссылается на цифры, она аналогична \d. |
| [^] | Все символы, исключая те, которые помещены в скобки. К примеру, [^aeuioAEUIO] ссылается на все символы, кроме гласных букв. [^0-9] ссылается на все символы, **не** являющиеся цифрами (аналогично \D). |
| \| | Сошлется на символ, который справа от \|, **или** же на символ слева. К примеру, a\|b сошлется **или** на a, **или** на b, **но** не на оба сразу. |

Окончание табл. 7

| | |
|---|---|
| \ | Используется, чтобы «блокировать» служебные символы регулярных выражений (\, *, ., / и т. д.). Это может быть необходимо, если подобный символ должен быть найден в строке. К примеру, «.» в синтаксисе регулярных выражений означает «любой символ» поэтому, если вы хотите включить символ точки в поиск, его нужно «блокировать»: «\.». |

Изучение регулярных выражений – одна из тем, которую лучше показывать на примерах, поэтому следует разобрать несколько примеров регулярных выражений, чтобы вникнуть в тему. Предположим, требуется найти совпадение чисел в строке, допустим, чтобы проверить, является ли строка американским Zip-кодом.

**Пример 1**. Совпадение с одним числом.

Первый шаг заключается в том, чтобы обрисовать, как найти совпадение числа. Если обратиться к табл. 7, то можно увидеть для этого специальный символ регулярных выражений (\d), совпадающий с любым отдельно взятым числом.

**Пример 2**. Совпадение с пятью числами в строке.

Поскольку \d совпадает с отдельно взятым числом, совпадение с пятью числами достигается с помощью следующего регулярного выражения:

\d\d\d\d\d.

**Пример 3**. Совпадение только с пятью числами.

Такое регулярное выражение выбирает цель в первой части строки, с которой должно совпасть. То есть иногда получается совпадение с частью полного слова или набора символов. Данное регулярное выражение совпадает с первыми пятью числами в строке, которые оно встречает. Например, оно совпадет с 12345 в числе 12345678998. Очевидно, что 12345678998 — это не Zip-код, поэтому требуется регулярное выражение, нацеленное только на пять чисел.

Символ \b (называемый словоразделом) совпадает с любым небуквенным или нечисловым символом, поэтому можно переписать данное регулярное выражение так: \b\d\d\d\d\d\b. Можно также использовать символ ^ чтобы получить совпадение с началом строки, и символ $ – для совпадения с концом строки. Данный прием удобен, если требуется найти строку, полностью совпадающую с вашим регулярным выражением. Например, если кто-нибудь напечатает «kjasdflkjsdf 88888 Iksadflkjsdkfjb» в поле для Zip-кода в форме заявки, можно попросить посетителя разъяснить (и исправить) такой Zip-код, прежде чем оформить заказ. В итоге ищется что-то подобное: 97213 (без иных символов в последовательности). В данном случае регулярное выражение будет таким: ^\d\d\ d\d\d$.

**Пример 4**. Запуск регулярного выражения в JavaScript.

Предположим, что пользовательский ввод зафиксирован в переменной zip, и следует проверить, является ли информация, введенная в форму, правильным Zip-кодом из пяти чисел:

```
var zipTest = /*\d\d\d\d\d$/;  //создаем регулярное выражение
if (zip.search(zipTest) == -1)
{
alert('Вы ввели некорректный почтовый код');
}
else
{
// действия в случае, если код верен
}
```

Регулярное выражение в этом коде действует, но не слишком ли много работы — печатать \d пять раз? А что, если потребуется найти совпадение для 100 чисел в строке? К счастью, в JavaScript есть знак, который обозначает многократное появление одного и того же символа (см. табл. 2). Необходимо помещать этот знак сразу после того символа, совпадение с которым требуется найти.

Например, чтобы найти совпадение с пятью числами, можно написать \d{5}. \d означает совпадение с одним числом, затем {5} сообщает интерпретатору JavaScript, что необходимо найти совпадение с 5 числами. То есть \d{100} найдет совпадение со 100 числовыми знаками в строке.

**Другие примеры**

Допустим, требуется найти в строке имена всех GIF-файлов. Кроме того, необходимо извлечь имя файла и каким-то образом использовать его в вашем. Другими словами, нужно найти все строки, которые совпадают с основным шаблоном имени файла GIF, например, logo.gif, banner.gif или ad.gif.

**Пример 1**. Определить общий шаблон для этих имен.

Чтобы построить регулярное выражение, следует сначала установить искомый шаблон символов. В данном случае ищутся все файлы, чьи имена оканчиваются на .gif. Другими словами, перед .gif может быть любое количество чисел, букв или других символов.

**Пример 2**. Найти .gif.

Поскольку ищется текстовую строку .gif то можно подумать, регулярным выражением должно быть просто .gif. Однако, согласно табл. 1, точка имеет собственное значение – совпадение с любым символом. Таким образом, .gif совпадет не только с .gif, но и с tgif. Чтобы создать регулярное выражение с точкой, следует добавить перед ней слэш (\.), что означает «найти символ точки». Итак, регулярное выражение для нахождения .gif – это \.gif.

**Пример 3**. Найдите любое число символов перед .gif.

Чтобы найти любое количество символов, вы можете использовать .*, что означает «найти один символ (.) от нуля раз и больше (*)». Это регулярное выражение совпадает с любыми буквами любой строки. Однако, создав регулярное выражение вроде .*\.gif, можно получить больше совпадений, чем просто имена файлов. Например, если вы имели в виду строку «Данный файл – это logo.gif», регулярное выражение .*\.gif совпадет с целой строкой, тогда как на самом деле вы хотели

отыскать просто logo.gif. Чтобы осуществить это, используйте символ \S, совпадающий с любым не-пробелом: \S*\gif найдет logo.gif в строке.

**Пример 4**. Сделать поиск нечувствительным к регистру.

В данном регулярном выражении есть еще одна загвоздка: оно найдет файлы, оканчивающиеся только на .gif, но ведь .GIF также является подходящим расширением файла, поэтому данное регулярное выражение не укажет имени файла logo.GIF. Чтобы заставить регулярное выражение игнорировать разницу между буквами верхнего и нижнего регистров, следует использовать при его создании аргумент i.

```
/\S*\.gif/i
```

Необходимо заметить, что i выходит за пределы шаблона и находится справа от /, означающего конец регулярного выражения.

**Пример 5**. Запуск скрипта.

```
var testString = 'Данный файл — это logo.gif'; // исходная строка
var gifRegex = /\S*\.gif/i;      // регулярное выражение
var results = testString.match(gifRegex);
var file = results[0];           // logo.gif
```

Этот код извлекает из последовательности символов имя файла (о том, как работает метод match(), будет рассказано далее).

**Группирование частей шаблона**

Можно использовать круглые скобки, чтобы создавать в шаблоне подгруппы. Эти подгруппы очень удобны (если один из символов табл. 8 используется), чтобы найти совпадения со многими различными строками по одному и тому же шаблону.

Таблица 8

Символы, используемые для обозначения совпадения с множественными появлениями одного и того же символа или шаблона

| Символ | Совпадает |
|---|---|
| ? | Ноль или одно появление предыдущего символа. Он необязателен, но если он имеется, то должен быть только в одном экземпляре. Например, регулярное выражение colou?r совпадет и с color, и с colour, но не с colouur |
| + | Одно или больше появлений предыдущего символа. Он должен появиться как минимум один раз |
| * | Ноль или более появлений предшествующего символа. Он необязателен и может являться любое количество раз. Например, .* совпадает с любым количеством знаков |
| {n} | Точное число появлений предыдущего символа. Например, \d{3} совпадает только с 3 числами в строке |
| {n,} | n и более появлений предшествующего символа. Например, a{2,} совпадет с буквой a два и более раз: с сочетанием aa в слове aardvark или aaaa в слове aaaahhhh |
| {n,m} | Предшествующий символ появляется минимум n раз, но не более m раз. То есть \d{3,4} совпадет с 3 или 4 числами в строке (но не с двумя и не с пятью числами в строке) |

Например, требуется узнать, содержит ли строка подстроки Apr или April. Они обе начинаются с Apr, поэтому точно известно, что совпадение должно начинаться с этой последовательности, но нельзя просто указать на совпадение с Apr, так как таким образом вы найдете Apr в Apricot или Aprimecorp. Итак, вы должны найти совпадение Apr, за которым следует пробел или другое окончание слова (это регуляр-

ное выражение \b из табл. 1), то есть April, за которым следует окончание слова. Это значит, что il необязательно. Вот как вы можете использовать скобки:

```
var sentence = 'April is the month of spring.';
var aprMatch = /Apr(il)?\b/;
if{sentence.search(aprMatch)   != -1)   {
// найдено Apr или April
} else {
//не найдено
}
```

Регулярное выражение /Apr(il)?\b/ обязательно требует использования Apr, но второстепенный шаблон (il) необязателен (знак ? означает «ни разу или один раз»). Наконец, \b совпадает с концом слова, то есть не нужно находить Apricot или Aprilshowers.

**Замечание.** Достаточно обширную библиотеку регулярных выражений можно найти на сайте www.regexlib.com.

**Методы для поиска с использованием шаблонов**

**Пример 1**. Совпадение с шаблоном

Метод search(), описанный на ранее, – один из способов узнать, содержит ли строка шаблон регулярного выражения. Метод match() работает по-другому. Возможно использовать его со строкой, не только чтобы проверить, существует ли в данной строке шаблон – также можно извлечь этот шаблон, чтобы позже использовать его в других скриптах. Например, есть поле для ввода текста, в котором посетитель может оставить комментарий относительно сайта. Возможно, потребуется проверить, содержится ли в комментарии URL, и, если это так, получить URL для дальнейшей обработки.

Следующий код находит и извлекает URL с использованием match():

```
// Создание переменной, содержащей строку с URL
var text='Мой web-сайт это www.mysite.com';
// Создание регулярного выражения '
var urlRegex = /((\bhttps?:\/\/) | (\bwww\.))\S*/
// Нахождение совпадения с регулярным выражением в строке
var url = text.match(urlRegex);
alert (url [0] ) ;  // www.mysite.com
```

Сначала код создает переменную, в которой сохраняется строка, включающая URL www.mysite.com. Далее задается регулярное выражение для поиска URL. Наконец, к строке применяется метод match(). Функция match() – это метод для строк, поэтому все начинается с имени переменной, содержащей строку, потом к ней добавляется точка, а затем match(). Таким образом методу match() передается регулярное выражение для поиска совпадения.

В примере, данном выше, переменная url содержит результат совпадения. Если шаблон регулярного выражения не найден в строке, в результате выдается специальное значение JavaScript, называемое null. Если совпадение происходит, то скрипт возвращает массив, первым значением которого является совпавший текст. Например, здесь переменная url содержит массив, и в первом элементе массива записан совпавший с шаблоном текст. В данном случае url[0] содержит www.mysite.com.

**Замечание.** В JavaScript значение null считается равносильным значению «ложь», поэтому можно проверить, совпадает ли метод match() с чем-нибудь:

```
var url = text.match(urlRegex);
if (! url) {
// не совпадает
} else {
//совпадает
```

}

**Пример 2.** Совпадение с каждым образцом шаблона

Метод match() работает двумя различными способами в зависимости от того, как было настроено регулярное выражение. В примере, приведенном выше, метод возвращает массив с первым образцом совпавшего текста. Так, если имеется длинная строка, содержащая много URL, возвращается только первый найденный URL.

Однако также можно задействовать свойство регулярного выражения global для поиска в строке более чем одного совпадения.

Поиск становится глобальным, когда добавляется g в конец регулярного выражения.

```
var urlRegex = /((\bhttps?:\/\/)|(\bwww\.))\S*/g
```

Следует обратить внимание, что g находится за пределами / (используемого для закрывания шаблона). Это регулярное выражение осуществляет глобальный поиск. Когда оно используется с методом match(), то ищет все совпадения в строке и возвращает массив совпавших фрагментов текста – это отличный способ найти все ссылки в записи блога или, например, все образцы данного слова в большом текстовом блоке.

Возможно переписать код из предыдущего примера, используя глобальный поиск, следующим образом:

```
// Создание переменной, содержащей строку с гиперссылкой
var text='Не так уж и много замечательных сайтов, подобных www.mysite.com и www.mynewsite.com';
// Создание регулярного выражения с пометкой глобального поиска
var urlRegex = /((\bhttps?:\/\/)|(\bwww\.))\S*/g
// Нахождение совпадения с регулярным выражением в строке
```

```
var url = text.match(urlRegex);
alert(url[0]); // www.mysite.com
alert(url[1]);  // www. mynewsite.com
```

Можно определить количество совпадений, получив доступ к свойству length результирующего массива: url.length. В данном примере будет возвращено число 2, поскольку в строке примера было найдено две гиперссылки. Затем возможно получить доступ к каждой совпавшей строке, используя индекс соответствующего ей элемента. Так, в данном примере url[0] – это первое совпадение, а url[l] – второе.

**Пример 3.** Замена текста

Можно использовать регулярные выражения для замены текста в строке. Например, имеется строка, содержащая дату в следующем формате: 10.11.2008. Однако нужна дата, отформатированная так: 10/11/2008. Это можно сделать с помощью метода replace():

```
string.replace(найти, заменить на);
```

Метод replace() имеет два аргумента: первый – регулярное выражение, которое вы желаете найти в строке; второй – строка, заменяющая все найденные в регулярном выражении совпадения. Итак, чтобы изменить формат с 10.11.2008 на 10/11/2008, можно поступить следующим образом:

```
var date = '10.11.2008';  // строка
var replaceRegex = /\./g // регулярное выражение
var date = date.replace(replaceRegex,   '/');  // заменяет . на /
alert(date);  // 10/11/2008
```

В первой строке создается переменная, в которой сохраняется строка 10.11.2008. В реальной программе эта строка может быть введена в поле формы. Во второй строчке кода создается регулярное выражение: символы / означают начало и

конец шаблона регулярного выражения; \. означает точку, а g – глобальную замену, каждый экземпляр точки должен быть заменен. Если опустить символ g, то будет заменена только первая точка и в результате получится: 10/11.2008. В третьей строчке кода происходит замена – каждая точка меняется на /, и результат сохраняется в переменную date. Наконец, отформатированная по-новому дата отображается в окне предупреждения.

**Примеры регулярных выражений**

**Пример 1.** Адрес электронной почты

Проверка правильности почтового адреса – обычная рутинная работа в ходе приема пользовательского ввода из формы. Многие люди не хотят сообщать адрес электронной почты, обосновывая это как «не ваше дело», либо просто пишут его неправильно (например, missing&sawmac.commm). Следующее регулярное выражение поможет проверить, содержит ли строка адрес электронной почты в правильном формате:

[-\w.]+@([A-z0-9][-A-z0-9]+\.)+[A-z]{2,4}

Это регулярное выражение состоит из следующих частей.

[- \w.]+ совпадает с дефисом, любым словом или одной или несколькими точками. То есть шаблон совпадет с bob, bob.smith или bobsmith.

@ – обычный знак в адресе электронной почты: missing@sawmac.com.

[A-z0- 9] совпадает с буквой или числом.

[-A-z0-9]+ совпадает с одним или несколькими образцами букв, чисел или дефисов.

\. совпадает со знаком точки, например, в адресе sawmac.com.

+ совпадает с одним образцом (или более) шаблонов и включает три данных выше шаблона. Этот символ предназначен для имен поддоменов: bob@mail.sawmac.com.

[A-z]{2,4} — любая буква, встречающаяся 2, 3 или 4 раза. Позволит найти сочетание сот в .сот или uk в .uk.

**Замечание.** Данное выражение не проверяет реальности адреса, то есть работает ли он. Оно просто проверяет, соответствует ли данная строка формату адреса электронной почты.

**Пример 2.** Дата

Дата может быть записана различными способами, например, 09/28/2008, 9-28-2007, 09 28 2007 или даже 09.28.2007 (это форматы США; в других регионах день может записываться перед месяцем: 28.09.2007). Поскольку посетители вольны вводить дату в любом из этих форматов возможно понадобиться проверять, правильно ли оформлена дата.

Вот регулярное выражение, проверяющее правильность ввода даты:

([01]?\d)[-\/ .]([0123] ?\d) [-\/ .](\d{4})

Оно состоит из следующих частей.

() окружает два следующих шаблона регулярных выражений с целью их группирования. Вместе они образуют номер месяца.

[01]? совпадает с 0 или 1, а ? делает такое совпадение необязательным для первого числа в месяце. В любом случае оно не может быть больше 1 – нет 22-го месяца. Дополнительно для месяцев от января по сентябрь вы должны ввести просто 5, а не 05.

\d совпадает с любым числом.

[-\/ .] совпадает с дефисом, слэшем, обратным слэшем, точкой или пробелом. Между днем и месяцем допустимы разделители в виде этих знаков.

( ) – следующий второстепенный шаблон, в котором содержится день месяца.

[0123]? совпадает с 0, 1, 2, или 3 ноль и более раз. Поскольку в месяце нет 40-го дня, вы ограничиваете первую циф-

ру в номере месяца одной из этих четырех цифр. Этот шаблон необязателен (что определяется символом ?), поскольку кто-нибудь может ввести 9 вместо 09 для девятого дня месяца.

\d совпадает с любой цифрой.

[-\/ .] равен предыдущему.

( ) содержит год.

\d{4} совпадает с любыми четырьмя цифрами, например, 1908 или 2880.

**Пример 3.** Веб-адрес

Совпадение с веб-адресом полезно, когда пользователя просят ввести адрес сайта и хотят убедиться, что он указал именно адрес. Или етребуется просмотреть какой-либо текст и учесть все имеющиеся URL. Основное регулярное выражение для URL:

((\bhttps?:\/\/)|(\bwww\.))\S*

Это выражение несколько необычно, поскольку в нем используется много скобок для группирования различных частей. Рис. 7 послужит руководством по этому регулярному выражению. Первая пара скобок (обозначена 1) охватывает две другие пары скобок (2 и 3). Символ | между двумя этими парами означает «или». Иначе говоря, регулярное выражение должно совпадать с 2 или 3.

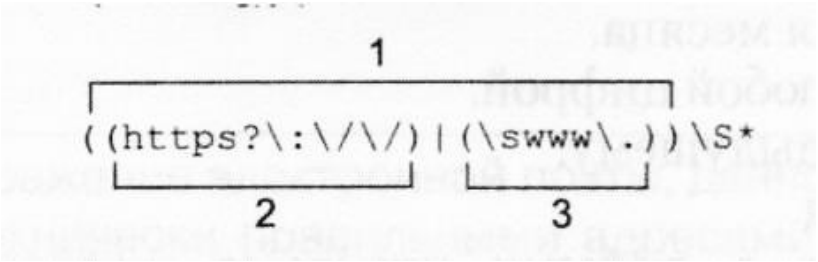


Рис. 7. Структура регулярного выражения web-адреса

Пояснения к рис. 7:

( – начало внешней группы (1 на рис. 7);

( – начало внутренней группы (2 на рис. 7);

\b – совпадает с началом слова;

http – совпадает с началом полного веб-адреса: http;

s? – необязательное s; поскольку веб-страницу можно послать через безопасное соединение, правильный почтовый адрес также может начитаться с https;

:\/\/ – совпадает с ://; поскольку прямой слэш имеет особое значение требуется предварить его обратным слэшем, чтобы найти совпадение с символом слэша;

) – окончание внутренней группы (2 на рис. 7); вместе эта группа будет совпадать с любым из http:// или https://;

| – совпадает с первой или со второй группой (2 или 3 на рис. 7);

( – начало второй внутренней группы (3 на рис. 7);

\b совпадает с началом слова;

www\. совпадает с www;

) – конец второй внутренней группы (3 на рис. 7); эта группа отвечает URL, который не содержит http://, а начинается с www;

) – конец внешней группы (1 на рис. 7); с этой точки зрения, регулярное выражение совпадет с текстом, который начинается с http://, https:// или www;

\S* совпадает с нулем или более символов, не являющихся пробелом.

Это выражение не обладает защитой от неумелого обращения (оно совпадет с бессмысленным адресом, например, http://#$*%&*@*), но оно достаточно простое и успешно совпадет с реальными URL.

**Замечание.** Чтобы посмотреть, содержится ли в строке что-нибудь, кроме URL, следует использовать символы ^ и $ в начале и в конце регулярного выражения и удалить символы \b:

^((https?.\/\/)|(www\.))\S*$

# 2. СОЗДАНИЕ ДИНАМИЧЕСКИХ ВЕБ-СТРАНИЦ
## 2.1. Динамическое модифицирование веб-страниц

**Объектная модель документа**

Когда браузер загружает HTML-файл, он выводит содержимое этого документа на экран. Но это не все, что браузер делает с тегами, атрибутами и содержимым файла, он также создает и запоминает «модель» HTML, использованного на этой странице. Такое представление содержимого называется объектной моделью документа (Document Object Model, DOM).

Другими словами, браузер запоминает теги HTML, их атрибуты и порядок, в котором они появляются на веб-странице. Объектная модель документа также предоставляет инструменты, необходимые для навигации на веб-странице, изменения ее содержания и добавления на страницу нового HTML. Сама по себе объектная модель документов не является частью JavaScript, это стандарт консорциума W3C, к которому производители большинства браузеров привели свои программы. Объектная модель документов позволяет JavaScript обмениваться информацией с веб-страницей и изменять на ней HTML.

Чтобы понять принцип работы объектной модели документов, рассмотрим пример очень простой веб-страницы:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  "http://www.w3.org/TR/ iml4/strict.dtd">
<html>
<head>
<title>Новая страница</title>
</head>
<body class="home">
<h1 id="header">Заголовок</h1>
<p>Some <strong>Important</strong> text</p>
</body>
</html>
```

На этом и на всех остальных сайтах одни теги заключены в другие, например, тег <html> включает все остальные, тег <body> – многие теги с содержимым, загружаемым в окно браузера. Вы можете представить от ношения между тегами в виде модели, напоминающей генеалогическое древо (рис. 8). Тег <html> – это корень дерева (прапрапрадедушка всех тегов страницы), остальные теги – это ветви, например, <head> и <body>, каждый из которых содержит собственный набор тегов.

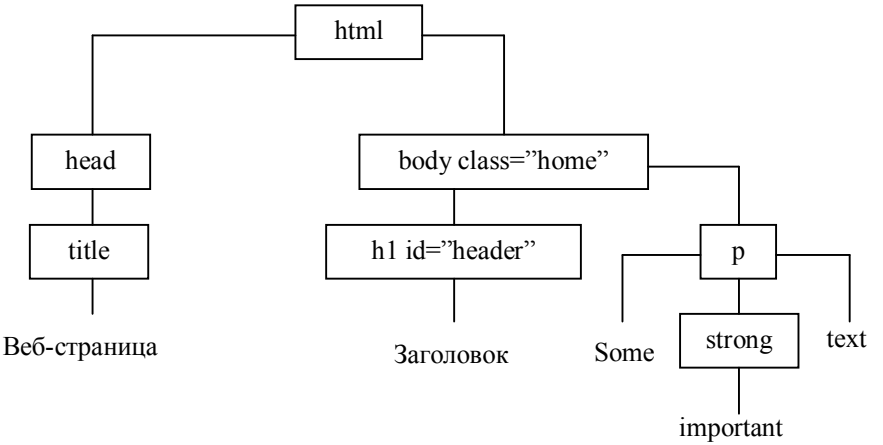


Рис. 8. Структура HTML-страницы
в виде генеалогического древа

Браузер также отслеживает текст, появляющийся в теге (например, «Заголовок» в теге <h1> на рис. 1), а также атрибут, присваиваемый каждому тегу (на рис. атрибуты присваиваются тегам <body> и <h1>). На деле объектная модель документов считает теги (также называются «элементами»), атрибуты и текст отдельными единицами – узлами.

**Выбор элемента страницы**

Браузер считает веб-страницу просто организованным набором тегов, атрибутов тегов и текста, или, на языке объектной модели, связкой узлов. Итак, JavaScript для манипулирова-

ния содержимым должен обмениваться информацией с узлами страницы. Есть два основных метода доступа к узлам:

*getElementByID()* и *getElementsByTagName()*.

**getElementByID()**

Получение элемента по ID означает нахождение отдельно взятого узла, имеющего уникальный идентификатор ID. Например, на рис. 1 тег <h1> имеет атрибут ID со значением «header». Получить доступ к данному узлу можно при помощи такого кода:

```
document.getElementById('header')
```

Часть document из document.getElementByld (‘header’) – это ключевое слово, которое указывает на целый документ. Оно обязательно, то есть нельзя просто написать просто getElementById(). Команда getElementById() – это не метод объекта document, а часть «header» – это просто строка с именем искомого ID, переданная методу как аргумент.

**Замечание**

Метод getElementById() требует в качестве аргумента одну строку – имя ID-атрибута тега. Например:

```
document. getElementById('header')
```

Однако это не означает, что методу можно передать только литерал строки. Также можно передать ему переменную, содержащую строку с именем искомого ID:

```
var lookFor = 'header';
var foundNode = document.getElementById(lookFor);
```

Можно присвоить результаты работы этого метода переменной, чтобы сохранить ссылку на отдельный тег и позже иметь возможность управлять им в программе. Например,

необходимо использовать JavaScript, чтобы изменить заголовок в HTML из примера в начале темы. Можно сделать так:

```
var headLine = document.getElementById('header');
headLine.innerHTML =  'JavaScript был тут!';
```

Команда getElementById() возвращает на отдельно взятый узел, ссылку, которая в данном примере сохраняется в переменной headline. Это очень удобно и позволяет просто ссылаться на имя переменной всякий раз, когда требуется манипулировать данным тегом, что значительно удобнее пространного document.getElementById('Имя ID').Например, во второй строке кода для доступа к свойству тега innerHTML используется переменная: headline.innerHTML (что такое innerHTML, будет объяснено далее).

**getElementsByTagName()**

Принцип работы данного метода схож с getElementById(), но вместо ID, указывается имя искомого тега. Например, чтобы найти все гиперссылки на странице, следует написать следующее:

```
var pageLinks = document.getElementsByTagName('a');
```

Метод getElementsByTagName() возвращает список узлов, а не отдельно взятый узел. Данный список является массивом, так что можно получить доступ к отдельному узлу в нем, используя индекс. Также можно перебрать все элементы с помощью сочетания свойства length и цикла for.

Например, первый элемент в массиве pageLinks из кода, приведенного выше, – это pageLinks[0] (первый тег <a> на странице), а pageLinks.length – это общее количество тегов <a> на странице]

**Замечание**

Используя эти два метода, достаточно легко допустить опечатку. Как правило, и новички, и профессионалы пишут ID

заглавными буквами. Но заглавной должна быть только первая буква. Подобным образом Elements – это множественное число getElementsByTagName():

```
document. getElementById('banner');
document.getElementsByTagName('a');
```

getElementById() и getElementsByTagName() можно также использовать: вместе. Например, имеется веб-страница, содержащая тег , который имеет ID прикрепленного к нему баннера. Если вы хотите узнать, сколько ссылок находится в теге , можно использовать getElementById(), чтобы вернуть , а затем getElementsByTagName(), чтобы искать ссылки в . Вот как это работает:

```
var banner = document.getElementById('banner');
var bannerLinks = banner.getElementsByTagName('a');
var totalBannerLinks = bannerLinks.length;
```

Метод для нахождения элемента с помощью ID document.getElementById() осуществляет поиск по всему документу.

В отличие от него, метод document.getElementsByTagName() позволяет находить теги определенного типа, производя поиск как по всему документу, так и среди тегов отдельно взятого узла. Например, в коде, приведенном выше, переменная banner содержит ссылку на тег , поэтому код banner.getElementsByTagName( 'a' ) ищет только теги <a> исключительно внутри этого .

**Выбор соседних узлов**

Тег, находящийся внутри другого, называется дочерним. Тег <h1> в образце HTML в начальном примере (рис. 8) является дочерним по отношению к тегу <body>, и поскольку он также находится внутри тега <html>, то является дочерним и по

отношению к нему тоже. Теги, содержащие другие теги, называются родительскими. Так, на рис. 8 тег <p> является родительским по отношению к тегу .

Объектная модель документов может дать доступ к «родительскому», «дочернему» или «братскому» узлу. Эти отношения изображены на рис. 8: если узел расположен непосредственно в другом узле, как текст Some в теге <p>, то это «ребенок»; узел, содержащий другой узел (подобно тому, как тег  содержит текст important), является «родителем». Узлы, у которых один и тот же «родитель», как, например, два текстовых узла Some и strong, называются «братьями».

DOM предлагает несколько способов доступа к ближним узлам.

**Способ 1.** .childNodes – свойство узла, содержащее список всех узлов, являющихся дочерними по отношению к данному. Данный список аналогичен массиву, возвращаемому методом getElementsByTagName. Предположим, что в HTML-файл добавляется следующий код JavaScript:

```
var headline = document.getElementById('header');
var headlineKids = headline.childNodes;
```

Переменная headlineKids будет содержать список тегов, являющихся дочерними по отношению к тегу, имеющему ID «header» (в данном примере тег <h1>). Сейчас дочерним является только текстовый узел, содержащий текст «Заголовок». Если необходимо знать, какой текст находится в этом узле, добавьте еще одну строку кода:

```
var headlineText = headlineKids[0].nodeValue;
```

Первый дочерний элемент в списке – headlineKids[0], поскольку только он является дочерним по отношению к заго-

ловку (см. рис. 8). Чтобы получить текст, находящийся в текстовом узле, необходимо использовать свойство nodeValue.

**Способ 2**. .parentNode – свойство, ссылающееся на родительский узел выбранного узла. Например, если вы хотите знать, какой тег включает в себя тег <h1> (рис. 8), напишите следующий код:

```
var headline = document.getElementById('header');
var headlineParent = headline.parentNode;
```

Переменная headlineParent в данном случае ссылается на тег <body>.

**Способ 3**. .nextSibling и .previousSibling – свойства, указывающие на узел, идущий сразу за данным узлом (nextSibling), либо на узел, предшествующий данному (previousSibling). Например, на рис. 8 теги <h1> и <p> являются «братьями»: тег <p> идет прямо после закрывающего тега </h1>.

```
var headline = document.getElementById('header');
var headlineSibling = headline.nextSibling;
```

Переменная headlineSibling – это ссылка на тег <p>, следующий за тегом <h1>. Если осуществляется попытка получить доступ к несуществующему братскому узлу, JavaScript возвращает значение null. Например, можно проверить, есть ли у узла братский узел previousSibling, следующим образом:

```
var headline = document.getElementById('header');
var headlineSibling = headline.prevSibling;
if (! headlineSibling)  {
alert ('Данному узлу другие узлы не предшествуют!');
}  else  {
```

```
// некие действия с предшествующим узлом
}
```

**Добавление содержимого на страницу**

Программы JavaScript часто должны добавлять содержимое на страницу, удалять или изменять его.

Добавление содержимого с использованием объектной модели документа – очень нудная работа. Она заключается в создании каждого узла содержимого и в помещении результата на страницу. Другими словами, если требуется добавить тег , а также еще пару тегов и текст, то следует отдельно создать каждый узел и правильно разместить его по отношении к остальным. Однако, производители браузеров разработали значительно более простой метод – innerHTML.

Свойство innerHTML не является стандартной частью объектной модели документа. Впервые оно было использовано в Internet Explorer, сейчас все современные браузеры, понимающие JavaScript, поддерживают его. Проще говоря, свойство innerHTML представляет собой весь HTML, находящийся в узле. Например, в HTML-коде исходного примера находится тег <p>. Итак, innerHTML для этого тега <p> таков: Some important text. Для доступа к этому HTML используется следующий код JavaScript:

```
//получить список всех тегов <p> на странице
var pTags = document.getElementsByTagName('p');
//получить первый тег <p> на странице
var theP = pTags[0];
alert(theP.innerHTML);
```

В данном случае переменная theP ссылается на узел для первого параграфа страницы. Последняя строка кода открывает окно предупреждения и отображает весь код, находящийся внутри этого тега. Например, добавление JavaScript в HTML в исходном примере заставит появляться в окне предупреждения текст Some important text.

**Замечание**

innerHTML – это предполагаемая часть нового стандарта HTML 5, разработанного в рамках W3C (www.w3.org/TR/html5).

Используя innerHTML, можно не только узнать, что находится внутри узла. Переопределив данное свойство, можно изменить содержимое узла:

```
var headLine = document. getElementById ('header');
headLine.innerHTML =  'Этот текст был добавлен с помощью JavaScript';
```

В данном примере содержимое 'Заголовок' внутри тега с ID «headH меняется на 'Этот текст был добавлен с помощью JavaScript'. innerHTML не ограничен текстом: можете переопределять свойство innerHTML для дополнения HTML, включая теги и их атрибуты.

**Дополнение**. При выборке узлов надо иметь в виду, что в нее включаются не только теги, но и другие элементы кода HTML (атрибуты, комментарии и т.д.). Поэтому следует проверять, к какому типу относится элемент выборки с помощью свойства nodeType. Например:

```
var headline = document.getElementById('header');
var headlineKids = headline.childNodes;
for (var i=0; i<headlineKids.length; i++)
if (headlineKids[i].nodeType==1)
        alert(headlineKids[i]);
```

Если элемент является тегом, то его свойство nodeType равно 1, если атрибутом – то 2 и т.д. (до 12).

## 2.2. Модифицирование веб-страниц с использованием jQuery

**Общие сведения о JQuery**

Во многих программах на JavaScript приходится иметь дело с одними и теми же задачами, которые необходимо решать снова и снова: выбор элемента, добавление нового содержимого, модифицирование атрибутов тега определение значения поля формы и реакция программы на различные действия в ней. Детали этих основных видов деятельности могут быть достаточно сложными, особенно если требуется, чтобы программа работала во всех основных браузерах. Библиотеки JavaScript позволяют быстро решить многие трудоемкие задачи программирования.

**Библиотека JavaScript** – это коллекция кодов, предлагающая простые решения многих повседневных задач программистов. В ней содержатся готовые функции JavaScript, которые можно добавлять на веб-страницу. Существует множество библиотек JavaScript, и элементы многих из них используются на ведущих сайтах таких как Yahoo, NBC, Amazon, Digg, CNN, Apple, Microsoft, Twitter и многих других.

Одной из самых популярных библиотек для JavaScript является библиотека JQuery. Она обладает следующими преимуществами:

– сравнительно небольшой размер файла (минимальная версия библиотеки занимает 55 Кбайт, а заархивированная – 30 Кбайт);

– понятность –для ее освоения требуется всего лишь некотрое знакомство с CSS.

– испытанность и надежность – jQuery используется на тысячах сайтов, включая очень популярные, с большим трафиком: Digg, Dell, the Onion, Warner Bros. Records, NBC и Newsweek;

– бесплатность;

– большое сообщество разработчиков;

– большое количество плагинов.

Команда jQuery регулярно обновляет свою библиотеку. Новейшая версия доступна на сайте jquery.com в разделе Download.

После загрузки файла jQuery, следует поместить его где-нибудь на разрабатываемом сайте, например, в корневом каталоге. Некоторые веб-дизайнеры создают отдельный каталог специально для файлов JavaScript (js или libs) и сохраняют в него файл jQuery, а также другие файлы с расширением JS.

Далее необходимо прикрепить загруженный файл к веб-странице. Он является обычным внешним файлом с расширением JS, поэтому он присоединяется к странице так же, как и любой внешний файл JavaScript. Допустим, файл jquery.js сохранен в каталог js в корневую папку сайта. Чтобы прикрепить файл к домашней странице, нужно добавить следующий тег в заголовок страницы:

```
<script type= "text/javascript" src="js/jquery.js"></script>
```

Прикрепив файл jQuery, можно создавать собственные скрипты, в которых используются преимущества продвинутых функций jQuery. Например, возможно прикрепить другой внешний файл JavaScript с собственным кодом или добавить второй тег <script> на веб-страницу и начать программировать:

```
<script type="text/javascript" src="js/jquery.js"></script>
<script type="text/javascript">
// здесь размещается ваш скрипт
</script>
```

**Получение доступа к элементам веб-страницы с использованием jQuery**

jQuery предлагает очень мощную технику выбора элементов и работы с их коллекциями – селекторы CSS. Селектор CSS – это просто инструкция, которая сообщает браузеру, с

помощью какого тега выполняется данный вид оформления. Например, h1 – это селектор, определяющий стиль для каждого тега <h1>; .copyright, в свою очередь, – это селектор класса, оформляющий любой элемент класса «copyright»:

```
<p class="copyright">Copyright, 2012</p>
```

С помощью jQuery можно выбрать один и более элементов, пользуясь специальной командой, вызывающей объект jQuery. Основной синтаксис таков:

```
$('селектор')
```

Допустим, требуется выбрать тег banner со специфическим ID при помощи jQuery. Нужно написать следующее:

```
$('#banner')
```

#banner – это селектор CSS, используемый для оформления тега с ID banner, # показывает, что идентифицируется ID. Допустим, вы захотели изменить HTML внутри элемента. Напишите следующее:

```
$((banner ' ).html ('<hl>JavaScript был тут!</hl>');
```

Подробнее работа с элементами страницы с помощью jQuery будет рассмотрена далее. Но сначала следует подробнее разъяснить, как использовать jQuery для получения доступа к элементам.

**ID-селекторы**

Можно получить доступ к любому элементу страницы, у которого есть ID, используя jQuery и селектор CSS ID. Например, имеется следующий HTML:

Специальное сообщение

Для выбора элемента с использованием метода объектной модели документа можно записать следующий код:

var messagePara = document.getElementById('message');

Метод jQuery выглядит так:

var messagePara = $('#message');

В отличие от метода объектной модели документа, не просто используется имя ID ('message'), а применяется синтаксис CSS для определения селектора ID (' #message').

**Селекторы элементов**

jQuery также имеет замену для метода getElementsByTagName(). Просто следует передать имя тега jQuery. Например, используя старый метод объектной модели документа для выбора каждого тега <a> на странице, можно написать бы следующий код:

var linksList = document.getElementsByTagName ('a');

С jQuery следует написать так:

var linksList = $('a');

**Селекторы классов**

jQuery предлагает простой метод выбора всех элементов с определенным именем класса. Просто можно использовать селектор класса CSS следующим образом:

$('.submenu')

Следует обратить внимание, что нужно ставить точку перед именем селектора класса. Единожды выбрав теги нужного класса, можно манипулировать ими, используя jQuery. Например, чтобы скрыть все теги с именем класса .submenu, можно написать следующее:

```
$('.submenu').hide();
```

**Продвинутые селекторы**

jQuery также предлагает более сложные селекторы CSS для точного указания на нужные теги.

Селекторы вложенности позволяют указывать на тег внутри другого тега. Например, создан неупорядоченный список ссылок и добавлен ID navBar тегу <ul> данного списка: <ul ID="navBar">. Если требуется выбрать только ссылки помеченного списка, то нужно использовать селектор вложенности следующим образом:

```
$('#navBar a')
```

Селектор, названный в списке последним, – это цель (в данном случае a), в то время как каждый селектор слева представляет тег, включающий цель.

Дочерний селектор указывает на тег, также являющийся дочерним (прямым «потомком» другого тега). Для создания дочернего селектора сначала следует назвать родительский элемент, за которым следует >, а затем – дочерний элемент. Например, для выбора тегов <p>, которые являются дочерними по отношению к тегу <body>, нужно написать следующее:

```
$('body > p')
```

Дополнительные селекторы позволяют выбрать тег, появляющийся сразу после другого тега. Например, есть невидимая панель, которая появляется только после щелчка кнопкой

мыши на вкладке. В HTML вкладка может быть представлена тегом заголовка (например, <h2>), а скрытая панель – тегом , который следует за заголовком. Чтобы сделать тег  (то есть панель) видимым, требуется получить к нему доступ. Можно легко сделать это с помощью jQuery и дополнительного «братского» селектора:

```
$('h2 + div')
```

Чтобы создать дополнительный «братский» селектор, следует просто добавить символ + между двумя селекторами (они могут быть любого типа: ID, классы или элементы). Селектор справа следует выбрать, только если он идет сразу после селектора, указанного слева.

**Селекторы атрибутов**

Селекторы атрибутов позволяют выбирать элементы в зависимости от того, есть ли у селектора определенный атрибут, и даже проверять, имеет ли атрибут указанное значение. С помощью селектора атрибута можно находить теги <img>, имеющие набор атрибутов alt, или даже искать совпадение с тегом <img>, имеющим определенное текстовое значение alt. Или можно найти все ссылки, ведущие за пределы сайта, и добавить к ним код, чтобы они открывались в новом окне.

Селектор атрибута добавляется после имени элемента, чей атрибут проверяется. Например, чтобы найти теги <img> и получить набор атрибутов alt, необходимо записать следующее:

```
$('img[alt]')
```

Рассмотрим набор различных селекторов атрибутов.

[атрибут] выбирает элементы, имеющие специфические атрибуты. Например, $ (a[href]) находит все теги <a>, имеющие набор атрибутов href.

[атрибут=значение] выбирает элементы, имеющие определенный атрибут со специфическим значением. Например, чтобы найти все окна с текстом в форме, можно использовать следующий код:

```
$('input[type=text]')
```

Поскольку большинство элементов форм имеют один и тот же тег (<input>), единственный способ сообщить тип элемента формы – проверить тип его атрибута (выбор элементов форм настолько обычен что jQuery включает специальные селекторы именно для этой цели).

[атрибут^ = значение] находит совпадения элемента и атрибута, который начинается с определенного значения. Например, если требуется найти ссылки, ведущие за пределы сайта, то можно использовать такой код:

```
$('a[href^=http://]')
```

Или же можно использовать этот селектор для идентификации ссылок mailto:

```
$('a[href^=mailto:]')
```

[атрибут$=значение] совпадает с элементами, чьи атрибуты заканчиваются определенным значением, что подходит для поиска файловых расширений. Например, с помощью данного селектора можно найти ссылки:

```
$('a[href$=.pdf]')
```

[атрибут*=значение] совпадает с атрибутом, в любом месте которого содержится специфическое значение. Например, можно найти любые ссылки, указывающие на отдельно

взятый домен. Данный код помогает найти ссылку, ведущую на mysite.com:

```
$('a[href*=mysite.com]')
```

Данный селектор гибок в использовании и позволяет находить ссылки, указывающие не только на http://www.mysite.com, но и на http://mysite.com и http://www.mysite.com/library.html.

**Фильтры jQuery**

jQuery позволяет фильтровать выборки, основываясь на определенных характеристиках. Чтобы использовать фильтр, следует добавить двоеточие после основного селектора, а за ним набрать имя фильтра. Например, чтобы найти все нечетные строки в таблице, можно написать селектор jQuery следующим образом:

```
$('tr:even')
```

Данный код выбирает все нечетные теги <tr>. Чтобы сузить выборку, можно найти все нечетные строки в таблице с именем класса striped:

```
$('.striped tr:even')
```

:even и остальные фильтры работают следующим образом.

**Вариант работы 1**. :even и :odd выбирают каждый второй элемент в группе. Эти фильтры работают несколько неинтуитивно; просто следует помнить, что выборка jQuery – это список всех элементов, совпадающих с указанным селектором представленный в виде массива. Каждый элемент выборки jQuery имеет индекс, а, индексы элементов массива всегда

начинаются с 0. Итак, поскольку фильтры :even фильтруют по четным значениям индекса (например, 0, 2 или 4), этот фильтр возвращает в данном случае первый, третий и пятый (и т. д.) элементы выборки, другими словами, он в действительности выбирает каждый второй нечетный элемент. Фильтр :odd работает так же, за исключением того, что он выбирает каждый нечетный индексный номер (1, 3, 5 и т. д.).

**Вариант работы 2**. :not() находит элементы, не совпадающие с данным типом селектора. Например, выбрать все теги <a>, кроме тех, которые относятся к классу navButton, можно следующим образом:

```
$('a:not(.navButton)');
```

Можно использовать :not() с любыми фильтрами и с большинством селекторов jQuery. Например, чтобы найти все ссылки, начинающиеся не с http://, можно написать следующий код:

```
$ ('a:not([href^=http://])')
```

**Вариант работы 3**. :has() находит элементы, содержащие другой селектор. Например, требуется найти все теги <li>, но только если внутри них есть тег <a>:

```
$('li:has(a)')
```

Данный фильтр несколько отличается от дочерних селекторов, так как выбираются не <a>; выбираются <li>, но только те из них, которые содержат тег <li> со ссылкой в нем.

**Вариант работы 4**. :contains() находит элементы, содержащие специфический текст, Например, чтобы найти все ссылки, гласящие «Щелкни на меня!», можно создать следующий объект jQuery:

$('a:contains(Щелкни на меня!)')

**Вариант работы 5**. :hidden находит скрытые элементы, к которым относятся либо элементы, обладающие свойством CSS display, установленным в значение none; либо элементы, использующие функцию jQuery hide(); либо скрытые поля форм. Однако данный селектор не применим к элементам, чей CSS имеет свойство visibility, установленное на invisible. Допустим, на веб-странице скрыли несколько тегов . Можете найти их и сделать видимыми с использованием jQuery следующим образом:

$('div:hidden').show();

Данная строка кода не воздействует на теги , которые в настоящий момент видимы на странице. О функции jQuery show() будет рассказано далее.

**Вариант работы 6**. :visible противоположен .hidden. Он находит на странице видимые элементы.

**Понимание выборок jQuery**

При выборе одного или более элементов с использованием объекта jQuery, например, $('#navBar a'), в итоге получается не традиционный список узлов объектной модели документа, как при использовании getElementById() или getElementsByTagName(). Вместо этого получается особая выборка элементов, применимая только в jQuery. Эти элементы «не понимают» традиционных методов объектной модели; например, нельзя использовать свойство innerHTML с объектом jQuery вот так:

$('#banner').innerHTML = 'Новый текст'; //не будет работать

Между работой DOM и выборками jQuery есть два концептуальных различия. jQuery была создана, чтобы облегчить и ускорить программирование на JavaScript. Одна из целей библиотеки – предоставить как можно больше функциональности в наименьшем объеме кода. Чтобы достичь этого, jQuery использует два необычных принципа.

**Автоматические циклы**

Как правило, когда используется объектную модель документа и выбирается связка узлов, необходимо создать цикл и просмотреть каждый из выбранных узлов, а затем что-либо с ним сделать.

Поскольку просмотр списка элементов с помощью цикла является обычной задачей, jQuery имеет такую встроенную возможность. Другими словами, применяя функцию jQuery к выборке элементов, не требуется создавать цикл, поскольку функция делает это автоматически.

Например, чтобы выбрать все изображения в теге  с ID slideshow, а затем скрыть их, в jQuery нужно записать следующий код:

```
$('#slideshow img').hide();
```

Список элементов, создаваемый с помощью $ ('#slideshow img'), может включать 50 изображений. Функция hide() автоматически проходит циклом через список, скрывая каждое из изображений отдельно.

**Цепные функции**

В некоторых случаях требуется произвести над выборкой элементов несколько операций. Например, необходимо установить высоту и ширину тега  (с ID popUp), используя JavaScript. Как правило, для этого нужно написать мини-

мум две строки кода. Но jQuery позволяет сделать это с помощью одной строки кода:

```
$('#popUp').width(300).height(300);
```

jQuery использует уникальный принцип – сцепливание, который позволяет добавлять одну функцию за другой. Каждая функция соединяется со следующей с помощью точки и работает с той же коллекцией элементов jQuery, что и предыдущая. Итак, код, приведенный выше, изменяет ширину, а также высоту элемента с ID popUp.

Цепные функции jQuery позволяют быстро выполнять большое количество действий. Например, если требуется не только установить высоту и ширину тега , но и добавить текст, а также заставить его становиться видимым (предполагается, что изначально текст невидим), то можно сделать это следующим образом:

```
$('#popUp').width(300).height(300).text('Привет!').fadeIn(1
000);
```

В данном коде к тегу с ID popUp применяются четыре функции jQuery: width(), height(), text() и fadeIn().

**Замечание**

Длинная строка цепных функций jQuery может быть сложночитаемой, поэтому некоторые программисты разбивают ее на несколько строк:

```
$( '#popUp' ).width(300)
.height(300)
.text('Сообщение')
.fadeln(1000);
```

Как только будет добавлена точка с запятой в последнюю строку, интерпретатор JavaScript будет считать эти строки одним выражением.

Способность сцепления функций достаточно необычна и является особой возможностью jQuery, другими словами, нельзя добавлять в цепь функции не из jQuery (например, те, которые созданы самостоятельно, а также встроенные функции JavaScript).

**Добавление содержимого на страницу**

jQuery предлагает много функций для манипулирования содержимым страницы: от простой замены HTML до точного расположения нового HTML относительно выбранного элемента и полного удаления тегов и содержимого со страницы.

Рассмотрим некоторые из этих функций. Допустим, что есть страница со следующим HTML:

```
<div id="container">
<div id="errors">
<h2>Ошибки:</h2>
</div>
</div>
```

***.html()*** работает как свойство DOM *innerHTML*. Может читать текущий HTML внутри элемента, а также замещать текущее содержимое другим HTML. Функция *html()* используется вместе с выборкой jQuery.

Для возвращения HTML прямо в ходе поиска по запросу просто следует добавить *.html()* после выборки jQuery. Например, можно запустить следующую команду, используя вышеприведенный фрагмент HTML:

```
alert($ ('#errors').html()) ;
```

Данный код создает окно предупреждения с текстом "<h2>Ошибки: </h2>".

Если в качестве аргумента функции *.html()* передается строка, то следует заменить текущее содержимое выборки:

$('#errors').html(' В данной форме имеется четыре ошибки ');

Данная строка кода заменяет весь HTML в элементе с ID *errors*. В итоге получится:

```
<div id="container">
<div id="errors">
<p> В данной форме имеется четыре ошибки.</p>
</div>
</div>
```

Следует заметить, что заменяется тег <h2>, который уже имелся.

**Замечание**

jQuery также имеет функцию *text()*, работающую как *html()*, за исключением того, что теги HTML, переданные *text()*, кодируются так, что  переводится в <p> – можно использовать это, если необходимо отобразить угловые скобки и названия тегов на странице. Например, можно применить это, чтобы дать кому-либо просмотреть образец кода HTML.

**append*()*** добавляет HTML как последний дочерний элемент выбранного элемента. Допустим, выбран тег , но вместо замены содержания  просто следует добавить немного HTML перед закрывающим тегом . Предположим, запускается следующий код на странице с HTML, приведенном в начале:

$('#errors').append(' В данной форме имеется четыре ошибки ');

После выполнения этой функции получается следующий код HTML:

```
<div id="container">
<div id="errors">
<h2 >Ошибки:</h2 >
<p> В данной форме имеется четыре ошибки </p>
</div>
</div>
```

**prepend*()*** схожа с *append(),* но добавляет HTML прямо после открывающего тега элементов выборки. Например, следующий код применяется к тому же HTML, приведенному выше:

```
$('#errors').prepend('<p>В данной форме имеется четыре ошибки </p>');
```

После выполнения функции *prepend()* получается следующий код HTML:

```
<div id="container">
<div id="errors">
<p> В данной форме имеется четыре ошибки </p>
<h2 >Ошибки:</h2 >
</div>
</div>
```

Теперь вновь добавленное содержимое появляется сразу после открывающего тега .

Если необходимо добавить HTML непосредственно рядом с элементами выборки – или до открывающего тега выбранного элемента, или прямо за закрывающим тегом элемента – следует использовать функции *before()* или *after()*. Например, принято проверять текстовое поле в форме, чтобы убедиться,

что посетитель не оставил его пустым. Предположим, что HTML этого поля до подтверждения выглядит так:

<input type="text" name="userName" id="userName">

Теперь допустим, что, когда посетитель отправляет форму, это поле пусто. Можно написать программу, проверяющую поле и добавляющую за ним сообщение об ошибке. Чтобы добавить сообщение после этого поля, можно использовать функцию *.after()* так:

$('userName').after('Имя пользователя обязательно!');

Данная строка кода выводит на веб-страницу сообщение об ошибке, HTML же будет выглядеть так:

< input type="text" name="userName" id="userName">
Имя пользователя обязательно!

Функция *.before()* просто помещает новое содержимое перед выбранным элементом.

**Замена и перемещение выборок**

Иногда необходимо полностью заменить выбранный элемент или переместить его. Например, создано всплывающее диалоговое окно с помощью JavaScript. Когда посетитель щелкает на кнопку **Закрыть**, требуется, конечно же, убрать диалоговое окно. Чтобы сделать это, можно воспользоваться функцией jQuery *remove()*. Допустим, всплывающее диалоговое окно имеет ID *popup;* чтобы его удалить, можно использовать следующий код:

$('#popup').remove();

Функция *.remove()* не ограничена только одним элементом. Предположим, требуется удалить все теги , имеющие класс *error.* Можно сделать это следующим образом:

```
$('span.error').remove();
```

Также можно полностью заменить выборку новым содержимым. Предположим, что есть тег <img> с ID *product101,* который вы просто хотите заменить текстом. Вот как это делается с помощью jQuery:

```
$('#product101').replace('<p>Добавлено в корзину</p>');
```

Данный код удаляет тег <img> со страницы и заменяет его тегом <p>.

**Замечание**

jQuery также включает функцию *clone()*, позволяющую копировать выбранный элемент.

**Установка и чтение атрибутов тегов**

**Добавление класса**

***addClass()*** добавляет элементу отдельный класс. Например, чтобы добавить класс *externalLink* всем ссылкам, ведущим за пределы сайта, можно использовать следующий код:

```
$('a[href^=http://]').addClass('externalLink');
```

Представим, что изначально был такой HTML:

```
<a href = "http://www.oreilly.com/">
```

Приведенный код изменит его на следующий:

<a href="http://www.oreilly.com/" class="externalLink">

Чтобы как-либо полезно использовать эту функцию, следует создать стиль класса CSS заранее и добавить его к таблице стиля страницы. Затем, когда JavaScript присоединит к элементу имя класса, браузер сможет применить к данному элементу свойства стиля заранее определенного правила CSS.

**Замечание**

При использовании функций addClass() и removeClass(), им передается просто имя класса, без точку, которая обычно используется при создании селектора класса.

Функция *addClass()* не исключает прежних классов, она просто добавляет новый класс.

**Удаление класса**

***removeClass()*** противоположна *addClass().* Она удаляет указанный класс из выбранных элементов. Например, если требуется удалить класс *highlight* из тега  с ID *alertBox,* нужно сделать следующее:

```
$('#alertBox') .removeClass('highlight');
```

**Переключение класса**

Переключение – распространенный способ показа элемента в действии или в бездействии.

Допустим, на вашей веб-странице есть переключатель (флажок или радиокнопка), который, будучи включенным, изменяет класс тега <body>. Предположим, переключатель, изменяющий стиль страницы, имеет ID *changeStyle.* Требуется, чтобы он отключал класс с именем *altStyle,* когда посетитель на нем щелкает. Следующий щелчок должен включить данный класс. Можно добавить следующий код:

```
$('#changeStyle').click(function()  {
```

```
$('body').toggleClass('altStyle');
});
```

Здесь происходит событие, которое позволяет скриптам реагировать на действия. Щелчок на переключателе – событие на странице. В строке кода, выделенной жирным шрифтом, демонстрируется функция *toggleClass(),* которая либо добавляет, либо удаляет класс *altStyle* при каждом щелчке.

**Чтение и изменение свойств CSS**

Функция jQuery *css()* позволяет прямо изменять свойства CSS определенного элемента, то есть не просто применять к элементу стиль класса, а непосредственно добавлять границу, фоновый цвет, устанавливать ширину или задавать расположение. Можно использовать функцию *css()* тремя способами: находить текущее значение свойства CSS определенного элемента, устанавливать свойство CSS определенного элемента или настраивать множественные свойства CSS одновременно.

Для определения текущего значения свойства CSS следует передать имя свойства функции *css()*. Предположим, требуется узнать фоновый цвет в теге  с ID *main:*

```
var bgColor = $('#main').css('background-color');
```

После выполнения данного кода переменная *bgColor* будет содержать строку со значением фонового цвета (jQuery возвращает или 'transparent', если цвет не установлен, или значение одного из основных цветов, например, 'rgb (255, 255, 0)').

**Замечание**

jQuery может не всегда возвращать те значения CSS, которые ожидаются. Например, jQuery не понимает кратких обозначений свойств CSS, например, font, margin, padding или border. Вместо них следует использовать полные свойства CSS,

например, font-face, margin-top, padding-bottom или border-bottom-width для доступа к стилям, которые имеют в CSS сокращенные обозначения. Кроме того, jQuery переводит все значения единиц в пиксели, поэтому, даже если используется CSS для установки размера тега <body> в масштабе 150%, jQuery возвращает значение в пикселях при проверке свойства font-size.

Функция *css()* также позволяет задавать свойство CSS для элемента. Для этого функции следует передать два аргумента: имя свойства CSS и значение. Например, чтобы изменить размер гарнитуры для тега <body> до 200 %, можно сделать следующее:

```
$('body').css('font-size', '200%');
```

Второй присваиваемый аргумент может быть строкой, например, '200%', или числовым значением, которое jQuery переведет в пиксели. Например, чтобы изменить заполнение в тегах класса *.pullquote* на 100px можно написать следующий код:

```
$('.pullquote').css('padding', 100) ;
```

В данном примере j Query устанавливает свойство *padding* на 100 пикселей.

**Замечание**

Устанавливая свойство CSS с помощью функции css(), можно использовать метод сокращений CSS.Так можно добавить черную границу толщиной в один пиксель вокруг всех параграфов класса highlight:

```
$('p.highlight').css('border', '1px solid black');
```

**Одновременное изменение нескольких свойств CSS**

Если нужно динамически подсветить тег  (возможно в качестве реакции на действия посетителя), то можно изменить фон в теге  и добавить вокруг него границу:

```
$('#highlightedDiv').css('background-color','#FF0000');
$('#highlightedDiv').css('border',   '2px solid #FE0037');
```

Другой способ – присвоение функции *.css()* так называемой объектной константы. *Литерал объекта* (или *объектная константа)* – это список пар свойство/значение. После каждого имени свойства добавляется двоеточие (:), за которым следует значение; пары свойство/значение разделяются запятыми; все это берется в скобки – {}. Таким образом, литерал объекта для значений свойств CSS, приведенных выше, выглядит так:

```
{ 'background-color' : '#FF0000', 'border' : '2px solid #FE0037' }
```

Поскольку литерал объекта может быть неудобным для чтения, если она написана в одну строку, многие программисты разбивают ее на несколько строк. Следующий код функционально не отличается от предыдущего однострочного:

```
{
'background-color' : '#FF0000',
'border' : '2px solid #FE0037'
}
```

Чтобы использовать функцию *css()* с объектной константой, следует просто передать ей ее:

```
$('#highlightedDiv').css({
'background-color' : '#FF0000',
'border' : '2px solid   #FE0037'
});
```

**Чтение, установка и удаление атрибутов HTML**

Поскольку изменение классов и свойств CSS с использованием JavaScript – обычные задачи, в jQuery для этого есть встроенные функции. Однако, функции *addClass()* и *css()* – на самом деле просто сокращения для атрибутов HTML *class* и *style.* jQuery содержит функции общего назначения для работы с атрибутами HTML – функции *attr()* и *removeAttr().*

Функция *attr()* позволяет читать указанный атрибут HTML тега. Например, чтобы определить актуальный графический файл в теге <img>, следует передать строку 'src' (свойство *src* тега <img>) функции:

```
var imageFile =  $('#banner img' ).attr('src' );
```

Функция *attr()* возвращает значение атрибута, установленное в HTML. Этот код возвращает свойство *src* для первого тега <img> внутри другого тега с ID *banner,* так что переменная *imageFile* будет содержать путь к картинке: например, 'images/banner.png' или 'http://www.thesite.com/images/banner.png'.

Передавая второй аргумент функции *attr(),*можно установить атрибут тега. Например, чтобы перейти к иному изображению, требуется изменить свойство *src* тега <img> следующим образом:

```
$('#banner img').attr('src','images/newImage.png');
```

Если необходимо полностью удалить атрибут тега, следует использовать функцию *removeAttr().* Например, данный код удаляет свойство *bgColor* из тега <body>:

```
$(‘body’).removeAttr('bgColor') ;
```

**Работа с элементами выборок**

Как уже говорилось ранее, одним из уникальных свойств jQuery является то, что большинство функций автоматически прорабатывают в цикле каждый элемент выборки. Например, чтобы на странице исчезли все <img>, нужна следующая строка кода JavaScript:

```
$('img').fadeOut();
```

Функция *.fadeOut()* заставляет элемент исчезать медленно. Прикрепив ее к выборке j Query, содержащей много элементов, функция прорабатывает выборку в цикле и заставляет медленно исчезать каждый найденный элемент. Во многих случаях требуется просмотреть в цикле серию элементов и с каждым из них выполнить определенное действие. jQuery предлагает функцию *.each()* именно для этой цели.

Предположим, необходимо составить список всех внешних ссылок, имеющихся на странице, поместить его в библиографической рамке внизу страницы и назвать, например, «**Другие сайты, упомянутые в этой статье**». В любом случае такую рамку вы можете создать так:

– возвратить все ссылки, ведущие за пределы вашего сайта;

– получите атрибут HREF каждой ссылки (URL);

– добавить эту ссылку в другой список ссылок в библиографической рамке.

jQuery не располагает функцией, осуществляющей непосредственно описанные шаги, но можно воспользоваться функцией *each(),* чтобы сделать это самостоятельно:

```
$('selector').each();
```

**Анонимные функции**

Чтобы использовать функцию *each(),* ей следует передать особый аргумент – анонимную функцию. Она содержит действия, которые следует выполнить с каждым элементом.

Такая функция называется анонимной, потому что, в отличие от изученных уже функций, которые создавались ранее, ей не дается имя. Базовая структура анонимной функции:

```
function()  {
//здесь размещается код
}
```

Поскольку эта функция не имеет имени, ее нельзя вызвать напрямую. Вместо этого используется анонимная функция, чтобы передать ее другой функции:

```
$('selector').each(function()  {
// здесь помещается код
});
```

Пора разместить код внутри анонимной функции. Функция *each()* действует как цикл, то есть инструкции, содержащиеся в анонимной функции, будут поочередно применяться к каждому возвращенному элементу. Допустим, на странице есть 50 изображений и добавляется следующий код JavaScript к одному из скриптов страницы:

```
$('img').each(function()  {
alert('Найдено изображение!');
});
```

Появится 50 диалоговых окон с предупреждением «Найдено изображение!» Это очень раздражает, поэтому повторять такие вещи не рекомендуется.

**this и $(this)**

Чтобы получить доступ к элементу, прорабатываемому в настоящий момент в цикле, следует использовать специальное ключевое слово *this.* Оно относится ко всем элементам, вызы-

вающим анонимную функцию. Так, при первом проходе цикла *this* относится к первому элементу выборки jQuery, при втором проходе цикла ко второму элементу и т. д.

При работе с jQuery *this* относится к элементу традиционной объектной модели документа, поэтому можно получить доступ к традиционным свойствам объектной модели, таким как *innerHTML* или *childNodes*. Чтобы превратить *this* в ее эквивалент, принятый в jQuery, следует написать: $(this).

Чтобы лучше разобраться, как использовать *$(this),* рассмотрим еще раз задачу, описанную ранее (создание списка внешних ссылок в библиографической рамке внизу страницы).

Предположим, что в HTML-коде страницы уже есть тег , готовый для вставки в него внешней ссылки:

```
<div id="Библиография">
<h3>Web-страницы, упомянутые в данной статье</h3>
<ul id="bibList">
</ul>
</div>
```

Первый шаг – получение списка всех ссылок, ведущих за пределы сайта. Можно сделать это с помощью селектора атрибутов:

```
$('a[href^=http://]')
```

Теперь, чтобы выполнить цикл для каждой ссылки, добавляем функцию *each()*:

```
$('a[href^=http://]').each ()
```

Затем добавляем анонимную функцию:

```
$('a[href^=http://]').each(function () {
```

```
});
```

Первый шаг в анонимной функции – возвращение URL для ссылки. Поскольку каждая ссылка имеет собственный URL, нужен доступ к актуальному элементу на каждом этапе цикла. Ключевое слово *$(this)* позволяет сделать это следующим образом:

```
$('a[href^=http://]').each(function()  {
var extLink = $(this).attr('href');
});
```

Код, выделенный жирным шрифтом, делает несколько вещей. В нем создается новая переменная *(extLink),* в которой сохраняется значение атрибута *href* актуального элемента. При каждом проходе цикла *$(this)* относится к иной ссылке на странице, то есть в каждой итерации цикла значение переменной *extLink* меняется.

После этого нужно просто присвоить новый элемент списка тегу <ul>:

```
$('a[href ^=http://]').each(function( )  {
var extLink = $(this).attr('href');
$(#bibList').append('<li>'  + extLink +  '</li>');
});
```

**Замечание**

Скрипт, используемый в этом примере, хорошо подходит, чтобы проиллюстрировать, как используется ключевое слово $(this), но он не является самым лучшим способом записи на страницу всех внешних ссылок. Во-первых, если на странице нет ссылок, тег , жестко закодированный в HTML страницы, по-прежнему будет появляться, но будет пустым. Кроме того, если у посетителя страницы отключен JavaScript, он не увидит ссылок, а только пустую рамку. Более рациональ-

но использовать JavaScript и для создания охватывающего тега .

## 2.3. Работа с событиями в JavaScript

**События JavaScript**

Браузеры запрограммированы на распознавание основных действий, таких как загрузка страницы, передвижение указателя мыши, нажатие клавиши на клавиатуре или изменение размера окна браузера. Каждое из этих действий на странице является *событием*. Чтобы сделать веб-страницу интерактивной, пишутся программы, отвечающие на события.

Подготовка веб-страницы к ответу на события проходит в два этапа.

Этап 1. Идентификация элемента страницы, который должен отвечать на событие.

Хотя на щелчок мыши в любой точке страницы может отвечать весь документ более обычной практикой является присвоение событий отдельным элементам страницы – ссылкам, полям форм или даже отдельным параграфам текста.

Этап 2. Определение, какая функция выполняется, когда оно происходит.

Существует несколько методов присвоения функции для ответа на событие. Но основная идея заключается в создании функции, выполняемой при каждом запуске события.

События JavaScript можно поделить на ряд категорий.

**События мыши**

**click**. Это событие запускается, когда пользователь нажимает и отпускает кнопку мыши. Обычно оно присваивается ссылке. Событие **click** действует и в том случае, если ссылка нажимается посредством клавиатуры.

**dblclick** (double-click). Запускается, когда пользователь дважды нажимает и отпускает кнопку мыши. Следует отметить, что двойной щелчок кнопкой мыши равносилен двум одиночным щелчкам, поэтому не следует присваивать события

одиночного и двойного щелчка одному и тому же тегу. В противном случае функция щелчка будет выполнена дважды до того, как запустится функция двойного щелчка.

**mousedown**. Это первая часть щелчка – момент, когда пользователь нажимает кнопку мыши, но не отпускает ее. Удобно при перетаскивании элементов по странице.

**mouseup**. Событие mouseup – это вторая часть щелчка – момент, когда пользователь отпускает кнопку.

**mouseover**. Запускается, когда пользователь проводит указателем мыши по элементу страницы. Используя mouseover, можно присвоить навигационной кнопке обработчик событий, заставляя появляться подменю, когда посетитель проводит указателем мыши по кнопке.

**mouseout**. Запускается, когда пользователь убирает указатель мыши с элемента. Можно использовать это событие, чтобы обозначить, когда посетитель убрал указатель мыши со страницы, либо чтобы спрятать появляющееся меню, когда указатель покидает его.

**mousemove**. Это событие запускается, когда пользователь передвигает указатель мыши, то есть оно происходит постоянно. Можно использовать **mousemove** для отслеживания актуального положения указателя на экране.

**События документа/окна**

**load**. Запускается, когда браузер заканчивает загрузку всех файлов веб-страницы: самого файла HTML, связанных изображений, flash-анимации, внешних файлов CSS и JavaScript. Обычно веб-дизайнеры используют это событие для запуска любой программы JavaScript, работающей с веб-страницей.

**resize**. Запускается, когда пользователь изменяет размер окна браузера, щелкая кнопку максимальной ширины окна, либо задает ширину окна браузера, передвигая его границу. Internet Explorer, Opera и Safari запускают при изменении размера окна событие resize много раз, Firefox же запускает его

только один раз, когда пользователь отпускает стрелку, изменяющую размер окна.

**scroll**. Запускается, когда пользователь использует полосу прокрутки или клавиатуру (клавиши Вверх/Вниз/Home/End и т. д.) либо прокручивает веб-страницу с помощью колесика мыши. Если на странице нет полос прокрутки, событие scroll все равно запускается.

**unload**. Запускается, когда пользователь щелкает на ссылке для перехода на другую страницу, закрывает вкладку в окне браузера или само окно. Safari и Internet Explorer запускают событие unload, когда пользователь щелкает на ссылке, чтобы уйти со страницы, или просто закрывает окно или вкладку на странице. Opera и Firefox позволяют нажимать кнопку, закрывающую окно, не запуская события unload.

**События форм**

**submit**. Запускается, когда посетитель подтверждает форму с помощью щелчка на кнопке Submit (Подтвердить) или просто нажатием клавиши Enter, когда курсор находится в текстовом окне. Наиболее часто это событие используется при проверке форм, чтобы убедиться, что все требуемые поля правильно заполнены, до того как данные отсылаются на веб-сервер.

**reset**. Кнопка Reset (Отменить) используется, чтобы отменить все изменения, сделанные в форме. Возвращает состояние страницы на тот момент, когда она была загружена. Можно запустить скрипт, когда посетитель пытается обнулить форму, используя событие reset.

**change**. Многие поля форм запускают это событие при изменении их статуса, например, когда кто-нибудь нажимает переключатель или выбирает пункт из раскрывающегося меню. Можно использовать **change**, чтобы сразу же проверить выбор, сделанный в меню, или посмотреть, какой переключатель был выбран.

**focus**. Запускается при переходе в текстовое окно или щелчке в нем кнопкой мыши. Иначе говоря, теперь внимание браузера сфокусировано на этом элементе страницы. Так, выбирая переключатель или щелкая на флажке, пользователь вводит эти элементы в фокус.

**blur**. Противоположно событию **focus**. Запускается, когда поле выводится из фокуса (при нажатии клавиши табуляции или при щелчке за пределами поля). Событие **blur** также полезно при проверке форм. Например, когда кто-либо вводит свой электронный адрес в текстовое поле и клавишей табуляции переходит в следующее поле, можно сразу же проверить, что ввел посетитель, чтобы убедиться, что это – правильный почтовый адрес.

**Замечание**

События **focus** и **blur** применимы также к ссылкам на странице. Когда пользователь переходит с помощью кнопки Tab на ссылку, запускается событие focus, а когда он «сходит» со ссылки с помощью кнопки Tab (или щелчка), наступает событие blur.

**События клавиатуры**

**keypress**. Это событие совпадает с моментом, в который пользователь нажимает клавишу. Причем, чтобы оно сработало, ему не нужно отпускать клавишу. Важно, что **keypress** вызывается снова и снова до тех пор, пока клавиша удерживается нажатой.

**keydown**. Подобно событию **keypress**, оно запускается при нажатии клавиши. На самом деле, оно запускается сразу после события **keypress**. В Firefox и Opera **keydown** запускается только один раз; в Internet Explorer и Safari – работает так же, как и **keypress**, – оно продолжается все то время, пока клавиша нажата.

**keyup**. Запускается, когда пользователь отпускает клавишу.

**Использование событий с функциями**

**Встроенные события**

Простейший способ активации функции в момент запуска события называется *регистрацией встроенных событий.* В данном случае *обработчик событий* присваивается HTML страницы напрямую. Например, чтобы заставить появиться окно предупреждения, когда пользователь наводит указатель мыши на какую-либо ссылку, можно написать следующее:

<a href="page.html" **onmouseover="alert('Это ссылка!');"**>Ссылка</a>

В данном случае событие – это наведение указателя мыши, обработчик событий вызывается с помощью *onmouseover.* Имена обработчика событий создаются простым добавлением слова *on* к началу наименования события, то есть обработчик события наведения указателя мыши называется *onmouseover*, а обработчик события щелчка – *onclick* и т. д.

Можно использовать технику встроенных событий даже для вызова предварительно созданной функции:

<body onload="startSlideShow()">

Строка кода, приведенная выше, вызывает функцию *startSlideShow* (сама функция должна быть определена где-либо на странице) после загрузки страницы и всех необходимых файлов.

**Традиционная модель**

Любой браузер, понимающий JavaScript, может пользоваться преимуществами присвоения обработчика событий тегу. У этой техники нет официального названия, но многие называют ее *традиционной моделью.* Она позволяет присваивать обработчик событий элементу страницы без сложностей, связанных с «блужданием» по HTML тела страницы. Данный ме-

тод включает идентификацию элемента страницы, которому требуется присвоить событие, а затем присвоение ему обработчика событий.

Допустим, нужно запрограммировать окно предупреждения, появляющееся при загрузке страницы. Для этого можно добавить следующий код в любой из тегов <script> – в заголовке страницы или во внешнем файле JavaScript:

```
function message() {
alert(«Добро пожаловать!»);
}
window.onload=message;
```

В первых трех строках создана простая функция *message.* Если программа ее вызывает, открывается окно с текстом «Добро пожаловать!». Событие происходит в строке 4, где объекту *window* присвоен обработчик события *onload* – в момент загрузки страницы вызывается функция.

Следует обратить внимание, что функция, присвоенная обработчику событий *onload,* очень похожа на значение, присваиваемое переменной. Знак равенства в window.onload=message; по существу сохраняет ссылку на функцию в обработчике событий. Вот почему после имени функции нет скобок: *message,* а не *message()*. Если после имени есть скобки, то функция выполняется сразу же. Так, код window.onload=message() вызывает функцию до загрузки страницы.

**Современный способ**

Два описанных выше метода ответа на события, происходящие на странице, используются с тех пор, как появился Netscape Navigator 3 (это было весьма давно) и все браузеры его понимали. У подобных техник есть один существенный недостаток –можно присвоить тегу или событию только одну функцию. Например, на обработчик событий *onload* может от-

вечать только одна функция, поэтому в коде, приведенном ниже, по существу второй обработчик событий стирает первый:

```
window.onload=message;
window.onload=setUpPage;
```

Чтобы справиться с подобными и другими проблемами, связанными со старыми методами обработки событий, консорциум W3C ввел инновацию, называемую ***приемниками событий***. Понимаемый под этим концепт очень похож на обработчики событий: выберите элемент (например, ссылку) и присвойте ей функцию, которая выполняется, когда происходит определенное событие. У любого элемента страницы может быть несколько приемников событий, поэтому можно присваивать несколько функций одному и тому же событию или тегу.

Firefox, Safari и Opera используют модель приемников событий W3C. Для привязки к событию функции используется метод addEventListener():

```
element.addEventListener(имя события, обработчик)
```

Имя события указывается с префиксом «on». Обработчик – это функция, которая будет запускаться, когда произойдет событие.

Но есть еще и Microsoft Internet Explorer, который избрал особый путь работы с событиями:

```
Element.attachEvent(имя события, обработчик)
```

Поэтому, надо указывать и тот и другой вариант привязки события к элементу, что не очень удобно.

Можно также воспользоваться библиотекой JavaScript, например, jQuery, которая предлагает способы единообразного обращения с событиями во всех современных браузерах.

**Способ jQuery**

В jQuery процесс создания обработчика событий будет состоять из следующих этапов.

Этап 1. Выбор нужных элементов.

Присваивая события, следует выбрать элементы, с которыми посетитель будет взаимодействовать. Например, на что будет щелкать посетитель – на ссылку, ячейку таблицы или изображение?

Этап 2. Присваивание события.

В jQuery большинство событий объектной модели документа (DOM) имеют эквивалентную функцию. Итак, чтобы присвоить событие элементу, следует добавить точку, имя события и пару скобок. Так, например, если необходимо добавить событие наведения указателя мыши к каждой ссылке на странице, можете это сделать следующим образом:

```
$('a').mouseover();
```

Чтобы добавить событие щелчка элементу с ID *menu,* следует написать так:

```
$('#menu').click();
```

Этап 3. Присваивание функции событию.

Требуется определить, что будет происходить при запуске события. Чтобы это сделать, требуется присвоить событию функцию. В ней должны содержаться команды, которые будут выполняться при запуске события.

Можно присвоить заранее определенное имя функции:

```
$('#start').click(startSlideShow);
```

Как было сказано ранее, когда событию присваивается функция, то следует опускать *()*, обычно добавляемые после

имени функции для ее вызова. Другими словами, следующий код работать не будет:

```
$('#start').click(startSlideShow())
```

Также можно присвоить событию *анонимную функцию*. Общий вид анонимной функции:

```
function()  {
// ваш код здесь
}
```

Рассмотрим простой пример. Допустим, имеется веб-страница со ссылкой, которой присвоен ID *menu.* Когда посетитель наводит указатель мыши на эту ссылку, должен появился ранее скрытый список дополнительных ссылок, имеющий ID *submenu*.

Процесс можно подразделить на четыре шага.

Шаг 1. Выбор меню:

```
$('#menu')
```

Шаг 2. Прикрепление события:

```
$('#menu').mouseover();
```

Шаг 3. Добавление анонимной функции:

```
$('#menu').mouseover(function()  {
})
```

Шаг 4. Добавление необходимых действия (в данном случае показывается подменю):

```
$('#menu').mouseover(function()  {
```

```
$('#submenu').show();
}) ;
```

**События jQuery**

**Ожидание загрузки HTML**

Когда страница загружается, браузер пытается немедленно запустить все скрипты, с которыми сталкивается. Таким образом, скрипты в заголовке страницы могут быть исполнены до окончания загрузки страницы. К сожалению, из-за этого свойства часто возникают проблемы. Поскольку многие программы на JavaScript связаны с управлением содержанием веб-страницы – отображением всплывающего сообщения при щелчке на ссылке, скрыванием отдельных элементов веб-страницы, окрашиванием в разный цвет четных и нечетных строк таблицы и т. д. – в итоге будут возникать ошибки, если ваша программа попытается управлять элементами, которые еще не загрузились и не отображаются в браузере.

Удобный способ решения данной проблемы связан с использованием события загрузки, чтобы дождаться, пока страница загрузится полностью и отобразится до выполнения кода JavaScript.

К сожалению, задержка с выполнением кода JavaScript до тех пор, пока не загрузится страница, может привести к странным результатам. Событие загрузки запускается только после того, как загрузятся все файлы веб-страницы, то есть все изображения, ролики, внешние таблицы стилей и т.д. В результате, если на странице много графики, посетитель может любоваться пустой страницей несколько секунд, пока не загрузятся все рисунки и не начнется выполнение кода JavaScript. Если код JavaScript вносит на страницу много изменений, например, оформляет строки таблицы, скрывает видимые в данный момент меню или даже контролирует вид страницы, посетитель вдруг увидит, как страница меняется прямо у него на глазах.

jQuery предлагает специальную функцию *ready(),* которая ожидает окончания загрузки HTML, а потом запускает скрипты страницы. Таким образом, JavaScript может непосредственно управлять веб-страницей без ожидания медленной загрузки изображений или роликов.

Основная структура функции *ready()* такова:

```
$(document).ready(function()   {
//здесь находится код JavaScript
});
```

В контексте полной веб-страницы функция будет выглядеть следующим образом:

```
TYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/ html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char-
set-UTF-8">
<title>Заголовок страницы</title>
<script type="text/javascript" src="js/jquery.js"></script>
<script type= "text/javascript">
$(document).ready(function( )  {
// весь ваш JavaScript размещается здесь.
});    // конец функции ready()
</script>
</head>
<body>
Контент веб-страницы...
</body>
</html>
```

**Замечание**

Поскольку функция ready() используется практически каждый раз при добавлении jQuery на страницу, ее можно за-

писывать кратко. Для этого следует удалить $(document).ready и напечатать:

```
$(function() {

});
```

**Дополнительные события jQuery**

jQuery также предлагает события для решения двух достаточно обычных задач, касающихся интерактивности: наведение указателя мыши на какой-либо объект и снятие указателя с этого элемента; переключение между двумя действиями посредством щелчка кнопкой мыши.

***Событие hover()***

События наведения и снятия указателя мыши часто используются вместе. Поскольку использование этих событий в паре обычно, jQuery предлагает краткий способ ссылки на оба. Функция *hover()* работает, как любое другое событие, кроме того, что вместо принятия одной функции в качестве аргумента, она принимает две функции. Первая функция выполняется при прохождении указателя мыши по элементу, а вторая – когда указатель покидает элемент. Структура выглядит так:

```
$('#селектор').hover(функция1, функция2);
```

Можно использовать функцию hover() с двумя анонимными функциями. Допустим, что кто-то наводит указатель мыши на ссылку с ID *menu* и требуется, чтобы появился на данный момент невидимый div с ID *submenu.* Когда указатель сдвигают со ссылки, подменю снова оказывается скрытым. Для осуществления этого можно использовать функцию *hover():*

```
$('menu').hover(
function() {
$('#submenu').show();
```

```
},
function()  {
$('#submenu').hide();
});
```

Можно вместо анонимных функций использовать обычные:

```
function showSubmenu() {
$('#submenu').show();
}
function hideSubmenu()  {
$('#submenu').hide();
}
$('#menu').hover(showSubmenu, hideSubmenu);
```

**Событие toggle()**

Событие *toggle()* в jQuery работает точно так же, как *hover(),* за исключением того, что вместо ответа на события наведения и убирания указателя оно отвечает на щелчки кнопкой мыши. Первый щелчок запускает первую функцию, второй – вторую функцию. Можно использовать это событие, если требуется чередовать два состояния с помощью щелчков. Например, элемент может появляться на странице с первым щелчком и исчезать со вторым.

Допустим, необходимо заставить появляться подменю  (из примеров с *hover(),* приведенных выше), когда пользователь щелкает на ссылке в первый раз, и исчезать после следующего щелчка на ссылке. В коде из предыдущего примера следует заменить «hover» на «toggle» следующим образом:

```
$('#menu').toggle(
function()  {
$('#submenu').show();
},
```

```
function()  {
$('#submenu').hide();
}
);
```

Или можно использовать функции с именем так:

```
function showSubmenu()  {
$('#submenu').show();
}
function hideSubmenu()  {
$('#submenu').hide();
}
$('#menu').toggle(showSubmenu, hideSubmenu);
```

**Объект события**

Когда браузер запускает событие, то записывает информацию о нем и сохраняет его в так называемом объекте события (Event Object). Объект события содержит сведения, собранные в момент, когда произошло событие, такие как координаты мыши по горизонтали и вертикали, элемент, с которым произошло событие, была ли нажата при запуске события клавиша Shift.

В jQuery объект события доступен через функцию, созданную для обработки данного события. Объект передается функции как аргумент, и, чтобы получить к нему доступ, всего лишь следует добавить к функции имя параметра. Например, требуется найти позиции указателей X и Y, когда происходит щелчок в каком-либо месте страницы.

```
$(document).click(function(evt)  {
var xPos = evt.pageX;
var yPos = evt.pageY;
alert('X:'  + xPos + ' Y:' + yPos);
});
```

Здесь важна переменная *evt*. Когда функция вызывается (то есть кто-нибудь щелкает кнопкой мыши в окне браузера), объект события сохраняется в переменной *evt*. Можно получить в теле функции доступ к различным свойствам объекта события, используя точечный синтаксис, например, *evt.pageX* возвращает расположение указателя по горизонтали (или количество пикселей от левого края окна).

Объект события имеет различные свойства и их список в каждом браузере может быть разным. В табл. 9 перечислены некоторые общие свойства.

Таблица 9

Общие свойства объекта события

| **Свойство события** | **Описание** |
|---|---|
| pageX | Расстояние (в пикселях) от указателя мыши до левого края окна браузера |
| pageY | Расстояние (в пикселях) от указателя мыши до верхнего края окна браузера |
| screenX | Расстояние (в пикселях) от указателя мыши до левого края монитора |
| screenY | Расстояние (в пикселях) от стрелки мыши до верхнего края монитора |
| shiftKey | Является истинным, если была нажата клавиша Shift, когда происходило событие |
| which | Используется с событием нажатия клавиши для определения числового кода нажатой клавиши |
| target | Объект, который был целью события, например, для события click() – это элемент, на котором щелкнули кнопкой мыши |
| data | Объект jQuery, использованный с функцией bind() для передачи данных функции, управляющей событием |

**Замечание**

При доступе к объекту события, соответствующему событию keypress(), можно получить числовой код нажатой клавиши. Если требуется проверить, что нажималась определенная клавиша (a, K, 9 и т.д.), следует преобразовать значение свойства which с помощью метода JavaScript, который превращает номер клавиши в точную букву или символ клавиатуры:

String.fromCharCode(evt.which)

**Отмена обычного поведения событий**

Некоторые элементы HTML имеют заранее запрограммированные ответы на события. Например, при щелчке на ссылке обычно загружается новая веб-страница; щелчок на кнопке подтверждения формы отсылает данные на веб-сервер для обработки. Иногда вы хотите, чтобы браузер вел себя не так, как обычно. Допустим, если форма подтверждена (событие *submit()*),можно пожелать остановить ее отправку, пока посетитель не внесет в форму недостающих важных данных.

Функция *preventDefault()*позволяет отменить нормальное поведение браузера. Она является частью объекта события, поэтому доступ к ней осуществляется в рамках функции, управляющей событием. Допустим, на странице есть ссылка с ID *menu.* Она указывает на другую страницу меню (чтобы посетитель, у которого отключен JavaScript, мог попасть на эту страницу меню). Однако добавлен код JavaScript, и если посетитель щелкает на ссылке, меню появляется на той же странице. Но, как правило, браузер все же проследует по ссылке на страницу меню, требуется избежать этого «по умолчанию» следующим образом:

```
$('menu').click(function(evt){
// JavaScript вставляется сюда
evt .preventDefault();  // перехода по ссылке нет
});
```

Другой вариант –возврат значения «ложь» как результата выполнения функции события. Например, следующий код функционально аналогичен коду, приведенному выше:

```
$('menu').click(function(evt){
// JavaScript вставляется сюда
return false;  // перехода по ссылке нет
});
```

**Удаление событий**

Иногда требуется удалить событие, которое ранее было присвоено тегу. jQuery-функция *unbind()* позволяет это сделать. Чтобы использовать ее необходимо создать объект jQuery с элементом, связанное с которым событие нужно удалить. Затем следует добавить функцию *unbind(),* передав ей в виде строки имя события. Например, если требуется отменить ответ всех тегов с классом *tabButton* на событие щелчка кнопкой мыши, можете написать так:

```
$('.tabButton').unbind('click');
```

Ниже приведен пример краткого скрипта, иллюстрирующего, как работает функция *unbind()*.

```
$('a').mouseover(function()   {
alert('Вы поместили указатель мыши на меня!');
});

$('#disable').click(function()   {
$('a').unbind('mouseover'); I }) ;
});
```

Первые три строки добавляют функцию к событию наведения указателя мыши на все ссылки (теги <a>) на странице. Если навести указатель на ссылку, появится окно с сообще-

нием «Вы поместили указатель мыши на меня!». Однако, поскольку постоянное появление окна с сообщением будет раздражать, следующие строки кода позволяют посетителю отключить предупреждение. Когда пользователь щелкает на элементе с ID *disable* (например, на кнопке формы), событие наведения указателя мыши удаляется со всех ссылок и сообщение больше не появляется.

**Расширенное управление событиями**

Метод *bind()* – это более гибкий способ работы с событиями по сравнению со специфичными для jQuery функциями, такими как *click()* или *mouseover()*. Он позволяет не только указывать событие и функцию для ответа на него, но и передавать дополнительные данные для использования функцией управления событием. Это дает то, что различные элементы и события (например, щелчок на ссылке или наведение указателя мыши на другое изображение) передают различную информацию одной и той же функции управления событиями, иными словами, одна и та же функция может действовать по-разному в зависимости от запускаемого события.

Формат функции *bind()* следующий:

```
$('#селектор').bind('событие', данные, имя функции);
```

Первый аргумент – это строка, содержащая имя события (например, щелчок, наведение указателя мыши и т.д.). Второй аргумент – это данные, которые нужно передать функции (в форме литерала объекта либо переменной, содержащей литерал объекта). Литерал объекта (или объектная константа) – это список имен свойств и их значений:

```
{
firstName :   'Боб',
lastName  :   'Смит'
}
```

Можно сохранить литерал объекта в переменной следующим образом:

```
var linkVar =  {message:'Привет от ссылки'};
```

Третий аргумент, передаваемый функции *bind(),* – другая функция, та самая, которая выполняет действие при запуске события. Это может быть или анонимная функция, или функция с именем, иначе говоря, данная часть не отличается от использования обычных событий jQuery.

**Замечание**

Передача данных при использовании функции bind() не обязательна. Если требуется использовать bind(), просто чтобы присвоить событие и функцию, можно опустить аргумент с данными:

```
$('#селектор').bind('событие', имя функции);
```

Функционально данный код идентичен следующему:

```
$('#селектор').click(имя функции);
```

Предположим, необходимо создать всплывающее окно для ответа на событие, но сообщение в окне должно быть разным в зависимости от того, какой элемент инициировал событие. Один из способов сделать это – создать переменные, хранящие различные объектные константы, а затем переслать параметры функции *bind()* разным элементам:

```
var linkVar =  {message:'Привет от ссылки'};
var pVar = {message:'Привет от параграфа'};
function showMessage(evt)   {
alert(evt.data.message);
}
$('a').bind('click',linkVar,showMessage);
```

```
$('p').bind('mouseover',pVar,showMessage);
```

В данном коде создаются две переменные: *linkVar* в первой и *pVar* во второй строке. Каждая переменная содержит объектную константу с одним и тем же именем переменной *(message),* но с различным текстом. Функция *showMessage()* принимает объект события и сохраняет его в переменной *evt.* Эта функция запускает команду *alert(),* отображая свойство *message* (которое само является свойством объекта *data)*.

## 2.4. Улучшение веб-форм

### Структура форм и выбор их элементов

HTML предлагает разнообразные теги для построения веб-форм. Наиболее важен тег <form>, определяющий начало (открывающий тег <form>) и конец (закрывающий тег </form>) формы. Он также указывает на метод, который форма использует для передачи данных (отправка или получение), и определяет, в какое место в Сети данные формы должны быть отправлены.

Для правильной работы формы следует создать средства управления ею – кнопки, текстовые поля и меню – с помощью тегов <input>, <textarea> или <select> соответственно. Большинство элементов форм используют тег <input>. Например, текстовые поля, поля для ввода пароля, переключатели, флажки и кнопки подтверждения используют один и тот же тег <input>. Следует указать на нужную форму с помощью атрибута *type*. Например, так создается текстовое поле с помощью тега <input>:

```
<input name="user" type="text">
```

### Выбор элементов форм

Прежде чем начать работу с элементами страницы, следует сначала их выбрать. Например, для определения значения,

хранящегося в поле формы, необходимо выбрать это поле. Подобным образом, если требуется скрыть или показать элементы *form*, можно воспользоваться JavaScript для идентификации данных элементов.

Простейшим путем выбора элемента *form* является присвоение ему идентификатора (ID):

```
<input name="user" type="text" id="user">
```

Затем можно применить метод *getElementById()* для выбора данного элемента:

```
var userField=document.getElementById('user');
```

Или можно использовать функцию выбора jQuery:

```
var userField = $('#user');
```

Выбрав поле, следует с ним что-нибудь сделать. Например, необходимо определить значение, введенное посетителем в поле. Если поле формы имеет ID ***user,*** можно использовать jQuery для доступа к значениям поля (про указанную в примере функцию val будет рассказано далее):

```
var fieldValue = $('#user').val();
```

Как выбрать все элементы ***form*** определенного типа? Например, нужно добавить события нажатия для всех переключателей на странице. Традиционный метод выбора элементов DOM здесь не очень поможет. Например, метод getElementsByTagName() позволяет выбрать теги <input> формы:

```
var fields = document.getElementsByTagName('input');
```

Однако, поскольку тег <input> используется не только переключателями, но и текстовыми полями, полями ввода пароля, флажками, кнопками подтверждения, кнопками сброса и скрытыми полями, единственное, что остается, – выбрать все поля ***input*** и затем пройти по списку в поиске соответствия атрибута ***type*** элементу, который нужен (например, ***radio).***

Можно также воспользоваться библиотекой jQuery. Используя один из селекторов jQuery, можно легко идентифицировать все поля определенного типа и работать с ними. Например, после заполнения формы посетителем нужно проверить, все ли ее поля содержат определенные значения. Можно выбрать все текстовые поля

$(':text')

Затем следует пройти по результатам с помощью *each()*, чтобы убедиться, что каждое поле содержит значение.

В табл. 10 приведены селекторы jQuery для работы с полями формы.

Таблица 10

Селекторы jQuery для работы с выбранными типами полей форм

| **Селектор** | **Пример** | **Описание** |
|---|---|---|
| :input | $(‘:input’) | Выделяет все элементы типа input, textarea, select и button, то есть все элементы форм |
| :text | $(‘:text’) | Выбирает все текстовые поля |
| :password | $(‘:password’) | Выбирает все поля для ввода пароля |
| :radio | $(‘:radio’) | Выбирает все переключатели |
| :checkbox | $(‘:checkbox’) | Выбирает все флажки |
| :submit | $(‘:submit’) | Выбирает все кнопки подтверждения (отправки) |

Продолжение табл. 10

| | | |
|---|---|---|
| :image | $(‘:image’) | Выбирает все кнопки-рисунки |
| :reset | $(‘:reset’) | Выбирает все кнопки сброса |
| :button | $(‘:button’) | Выбирает все поля типа buttons (кнопки) |
| :file | $(‘:file’) | Выбирает все поля файла (применяется для закачки файла) |
| :hidden | $(‘:hidden’) | Выбирает все скрытые поля |

Можно комбинировать селекторы друг с другом. Например, есть две формы на странице и требуется выбрать текстовые поля только из в одной из них.

Исходя из того, что нужная форма имеет ID *signup,* можно выбрать поля только данной формы следующим образом:

```
$('#signup :text')
```

Кроме того, jQuery предлагает весьма полезные фильтры, которые отбирают поля формы, соответствующие определенному состоянию.

***:checked*** выбирает все поля, помеченные или включенные, то есть флажки (checkboxes) и переключатели. Например, если нужно найти все поля такого рода, можно использовать следующий код:

```
$(':checked')
```

Этот фильтр можно применять для поиска переключателя внутри группы. Например, есть группа переключателей и требуется найти значение переключателя, которое выбрал по-

сетитель сайта. Данная группа использует один и тот же HTML-атрибут *name;* допустим, рассматриваемая группа переключателей использует для этого атрибута имя *email.* Применение селектора атрибута jQuery в сочетании с фильтром *:checked* позволяет найти значение помеченного переключателя:

```
var checkedValue = $('input[name=email]:checked').val();
```

***:selected*** выбирает все помеченные элементы *option* внутри списка или меню, позволяя узнать, какой выбор сделал посетитель (тег <select>). Например, есть тег <select> с ID ***state,*** в котором перечислены все 50 штатов США. Для выяснения того, какой же штат выбрал посетитель сайта, можно написать следующее:

```
var selectedState=$('#state :selected').val();
```

В отличие от примера с фильтром *:checked,* между ID и фильтром имеется пробел (' #state : selected'). Это объясняется тем, что этот фильтр выбирает теги <option>, а не тег <select>. Говоря проще, данный выбор jQuery означает «найди все выбранные варианты, находящиеся внутри тега <select> и имеющие ID state». Пробел делает свою работу подобно нисходящему селектору CSS: сначала он находит элемент с нужным ID, а затем внутри его ищет все выбранные элементы.

**Получение и ввод значений элементов форм**

Иногда возникает необходимость проверить значение элемента *form*. Например, нужно убедиться, что электронный адрес посетителя введен в соответствующее поле. Или требуется знать значение поля для подсчета общей стоимости заказа. С другой стороны, может понадобиться задать значение элемента ***form.***

jQuery предлагает простую функцию для выполнения этих задач. Функция ***val()*** может как задавать, так и считывать значения поля формы. Если она вызывается ее без аргументов, она будет считывать значения; при передаче функции значения она введет его в поле формы. Например, есть поле для ввода электронного адреса клиента с ID *email.* Его содержимое можно получить, введя следующий код:

```
var fieldValue = $('#email').val();
```

**Замечание**

Функция val() находит даже значение выбранного варианта в меню (тег <select>).

Задать значение полю можно простой передачей этого значения функции val(). Например, есть форма для заказа товаров и требуется автоматически рассчитать общую стоимость заказа, исходя из количества единиц товара, заявленного клиентом. Можно получить количество заказанных единиц, умножить его на цену товара и затем ввести значение в итоговое поле.

Код для выполнения этой задачи выглядит так:

```
var unitCost=9.95;
var amount=$('#quantity').val();  // получить значение
var total=amount*unitCost;
total = total.toFixed (2);
$('#total').val(total);  // задать значение
```

Строка 1 кода создает переменную, которая хранит значение цены товара. Строка 2 создает еще одну переменную и извлекает значение, введенное посетителем в поле с ID *quantity,* соответствующее количеству заказанного товара. Строка 3 определяет общую стоимость, полученную путем умножения цены на количество, а строка 4 форматирует ре-

зультат, включая в него два десятичных знака. Наконец, строка 5 вводит итоговое значение в поле с ID.

**Проверка наличия меток для кнопок и флажков**

Хотя функция *val()* полезна для получения значений элементов *form*, в некоторых случаях подобное значение имеет смысл лишь в том случае, если посетитель выбрал какое-то конкретное поле. Например, переключатели и флажки требуют от посетителя сделать выбор определенного значения (или нескольких значений в случае с флажками).

В HTML атрибут *checked* определяет, помечен ли данный элемент, Например, для установки флажка при загрузке веб-страницы следует добавить атрибут *checked* следующим образом (синтаксис XHTML):

<input type="checkbox" name="news" id="news" **checked="checked"/>**

Поскольку *checked* является атрибутом HTML, легко можно использовать jQuery для проверки статуса флажка:

```
if ($('#news').attr('checked'))  {
// флажок отмечен
} else {
// флажок не отмечен
}
```

Код $ ('#news ').attr('checked') возвращает значение *true*, если флажок установлен. Если он не установлен, возвращается значение *undefined,* интерпретируемое JavaScript как *false.* Таким образом, данная условная инструкция позволяет выполнить один набор задач, если флажок установлен, либо другой набор задач, если флажок не установлен.

Атрибут *checked* применим и к переключателям. Можно использовать функцию *attr()* таким же образом, чтобы проверить, установлен ли атрибут *checked* для переключателя.

**События формы**

**Подтверждение (Submit)**

Когда посетитель подтверждает заполнение формы нажатием кнопки Submit (Подтвердить) или клавиши Enter, происходит событие *submit.*

Для выполнения функции при наступлении события подтверждения формы следует сначала выбрать форму, а затем применить функцию jQuery *submit()*, чтобы добавить скрипт. Предположим, нужно убедиться, что поле с именем посетителя заполнено при передаче формы. Сделать это можно добавлением события подтверждения в форму и проверкой значения перед ее отправкой. Если поле пустое, нужно дать знать об этом посетителю и остановить процесс передачи формы; в противном случае форма будет передана с пустым полем.

Предположим, что форма имеет ID *signup,* а поле ввода имени – ID *username.* Верифицировать эту форму с помощью jQuery можно так:

```
$(document).ready(function()  {
$('#signup').submit(function()  {
if ($('#username').val()  == '')  {
alert('Please supply a name in the Name field.');
return false;
}
});  // конец submit()
});  // конец ready()
```

Строка 1 задает требуемую функцию *$(document).ready(),* так что код выполняется только после загрузки HTML страницы. Строка 2 присоединяет функцию к событию *submit* формы. Строки 3-5 выполняют верификацию.

Если поле пустое, появляется окно предупреждения, дающее посетителю понять, что он ошибся.

Строка 5 очень важна: она останавливает процесс передачи формы. Если пропустить этот шаг, тогда форма будет передана без имени пользователя.

Событие *submit* применяется только к формам, так что сначала следует выбрать форму, а затем присоединить к ней событие *submit.* Выбор формы осуществляется с помощью ID, содержащегося в теге <form> HTML, или, если страница имеет единственную форму, с использованием селектора элемента:

```
$('form').submit(function ()  {
// код, который подлежит выполнению после отсылки формы
}) ;
```

**Активное состояние (Focus)**

Когда курсор находится в текстовом поле (либо после щелчка в поле кнопкой мыши, либо при переходе в него с помощью клавиши Tab), оно переходит в состояние, называемое *focus.* Фокус представляет собой событие, запускаемое браузером и указывающее, что курсор находится или ином поле. Некоторые дизайнеры применяют его для удаления любого текста, который уже имеется в поле. Например, есть следующий HTML внутри формы:

```
<input name="username" type="text" id="username" value="Пожалуйста, введите ваше имя пользователя">
```

Этот код создает текстовое поле в форме с текстом «Пожалуйста, введите ваше имя пользователя» внутри его. Этот метод позволяет давать указания пользователю по заполнению формы. Посетитель не должен сам стирать ненужный текст, это делается автоматически при активации

```
$('#username').focus(function()  {
var field = $(this);
```

```
if (field.val()==field.attr('defaultValue'))  {
field.val (‘’) ;
}
});
```

Строка 1 выбирает поле (имеющее ID *username)* и присваивает функцию событию фокуса. Строка 2 создает переменную ***field,*** которая хранит ссылку на выборку jQuery; функция *$(this)* ссылается на выбранный текущий элемент внутри выборки jQuery, в нашем случае – поле формы.

Строка 4 обусловливает стирание содержимого поля. Она задает новое значение поля (пустую строку с двумя кавычками), удаляя таким образом любое имевшееся в поле значение. Не нужно стирать это поле всякий раз, когда оно оказывается в фокусе. Например, пользователь вошел в поле и вместо прежней надписи «Пожалуйста, введите ваше имя пользователя» ввел свой текст. Если затем он вернется в данное поле, введенное имя не должно исчезнуть. Вот когда вступает в игру условная инструкция, содержащаяся в строке 3.

Текстовые поля имеют атрибут *defaultValue,* представляющий текст, содержащийся в поле при первой загрузке страницы. Даже если этот текст стерт, браузер будет его помнить. Условная инструкция проверяет, соответствует ли текущий текст поля *(field.val())* тому, который был первоначально *(field.attr('defaultValue')).* Если имеется совпадение, тогда интерпретатор JavaScript стирает текст в поле.

**Неактивное состояние (Blur)**

Когда пользователь покидает поле или щелкает кнопкой мыши за его пределами, браузер запускает событие *blur,* которое обычно используют для выполнения скрипта верификации формы, после того как кто-то покидает заполненное поле.

Предположим, есть поле для сбора информации о количестве товаров, нужных клиенту. HTML может иметь такой вид:

```
<input name="quantity" type="text" id="quantity">
```

Необходимо убедиться, что поле содержит только числа (то есть 1, 2 или не **один, два** или **девять)**. Можно это сделать сразу после того, как посетитель покинул данное поле:

```
$('#quantity').blur(function ()  {
var fieldValue=$(this).val() ;
if  (isNaN(fieldvalue))  {
alert('Пожалуйста, введите число');
}
}) ;
```

Строка 1 присваивает функцию событию *blur*. Строка 2 извлекает значение поля и сохраняет его в переменной *fieldValue*. Строка 3 проверяет числовую природу значения методом *isNaN()*. Если значение не является числом, выполняется строка 4 и выводится сообщение об ошибке.

**Щелчок кнопкой мыши (Click)**

Событие нажатия происходит при щелчке кнопкой мыши на элементе формы.

Как и в случае с другими событиями, можно использовать функцию jQuery *click()* для ее присвоения событию щелчка в поле формы:

```
$(':radio').click(function()  {
//функция применится к любому переключателю при щелчке на нем
}) ;
```

**Замечание**

Событие щелчка кнопкой мыши применимо к текстовым полям, но не также, как в случае события фокуса. Фокус случается при щелчке в текстовом поле или при переходе в

текстовое поле, событие же щелчка происходит только при щелчке кнопкой мыши в текстовом поле.

**Смена (Change)**

Событие *change* применяется к меню формы (раскрывающемуся списку). При выборе элемента в списке, вызывается событие смены.

Для применения события смены к меню следует использовать функцию jQuery *change()*. Предположим, есть меню с ID *country* с названиями стран; всякий раз, когда делается новый выбор, следует быть уверенным, что это не текст «Выберите страну» (часто эта опция добавляется первой в список, чтобы служить подсказкой для пользователя). Можно использовать в данном случае код:

```
$('#country').change(function()  {
if  ($(this).val()=='Выберите страну')   {
alert('Пожалуйста, выберите страну в данном меню.');
}
}
```

**Усовершенствование полей формы**

**Перенос в первое поле формы**

Обычно, для того чтобы начать заполнение формы, пользователь должен попасть в первое текстовое поле и начать вводить текст. Если страница предусматривает регистрацию, зачем заставлять посетителей искать нужное место и только потом начинать регистрироваться? Почему бы просто не поместить курсор сразу в нужное поле, подготовив его таким образом к вводу данных? С JavaScript это проще простого.

Можно просто выбрать текстовое поле, после чего выполнить функцию jQuery *focus()*. Предположим, требуется, чтобы курсор был в поле ввода имени пользователя. Допустим, что это поле имеет ID *username*. Чтобы с помощью JavaScript

поместить фокус, то есть курсор, в это поле, можно написать следующее:

```
$(document).ready(function()  {
$('#username').focus () ;
}) ;
```

В данном примере текстовое поле имеет ID *username.* Однако возможно создать универсальный скрипт, который всегда будет придавать фокус первому текстовому полю формы без необходимости присвоения ему ID:

```
$(document).ready(function()  {
$(':text')[0].focus();
});
```

jQuery предлагает удобный способ выбора всех текстовых полей – *$(':text').* В данном случае сначала выбираются все текстовые поля (возвращая массив элементов), а затем задействуется только первый элемент массива – часть *[0].* Наконец, этот элемент переводится в активное состояние с помощью метода *focus();.*

Если ваша страница содержит более одной формы (например, формы «Поиск на сайте» и «Подпишитесь на рассылку новостей»), то нужно уточнить селектор для идентификации формы, чье текстовое поле должно получить фокус. Чтобы задать фокус нужному полю, просто следует добавить ID (например, *signup)* в форму и применить этот код:

```
$(document).ready(function ()  {
$('#signup :text')[0].focus();
});
```

Теперь селектор *$('#signup :text')* выбирает только поля внутри формы для подписки.

**Выключение и включение полей**

Поля формы обычно предназначены для заполнения. В конце концов, какой толк от текстового поля, которое нельзя заполнить? Однако бывает так, что нет необходимости, чтобы посетитель заполнял некоторые текстовые поля, устанавливал флажки или выбирал вариант в меню. Скажем, есть поле, которое следует заполнять, только если для предыдущего поля установлен флажок.

Чтобы «отключить» поле формы, можно воспользоваться атрибутом *disabled*. Деактивирование означает, что возле пункта не может быть установлен флажок или переключатель, в текстовые поля нельзя ввести текст, в меню – выбрать пункт, нельзя также нажать кнопку подтверждения. Некоторые браузеры практикуют изменение цвета для таких полей, например, делают их светло-серыми.

Чтобы отключить поле, можно просто задать для атрибута *disabled* значение *true*. Например, для отключения всех полей ввода формы можно использовать следующий код:

```
$(':input').attr('disabled', true);
```

Обычно вы выключаете поле в ответ на событие – например, щелчок по кнопке-флажку. Например, чтобы пользователь не вводил ФИО жены, в случае, если установил флажок «Холост» в форме, можно написать следующий код:

```
$(' #single').click(function()   {
$('#FIO_wife').attr('disabled', true);
```

Разумеется, отключая поле, в дальнейшем следует предусмотреть возможность вновь включить его. Для этого можно просто задать для атрибута *disabled* значение *false.* Например, для активации всех полей формы следует использовать код:

```
$(':input').attr('disabled',  false);
```

Вернемся к предыдущему примеру. Если пользователь выбирает параметр «Женат», то поле для ввода ФИО жены должно иметь активный статус. Полагая, что переключатель для параметра «Женат» имеет ID *married,* можно написать такой код:

```
$('#married').click(function()   {
$('#FIO_wife').attr('disabled',  false);
}) ;
```

**Скрытие и показ параметров формы**

Помимо отключения полей, имеется еще один путь не утруждать посетителей заполнением ненужных полей – их просто можно скрыть. В предыдущем примере сделать это можно следующим образом:

```
$('#single').click(function()   {
$('#FIO_wife').hide() ;
}) ;
$('#married').click(function()  {
$('#FIO_wife').show();
});
```

К преимуществам скрытия полей (в противоположность их отключению) относится упрощение макета (компоновки) формы. В конце концов, отключенное поле является видимым, что может привлекать (а точнее, отвлекать) внимание.

## 2.5. Основы Ajax

**Общие сведения об Ajax**

JavaScript – мощное, но не всесильное средство. Если необходимо представить информацию из базы данных, ото-

слать e-mail с результатами заполнения формы или просто загрузить дополнительный HTML, требуется связаться с веб-сервером. Для этого обычно нужно загрузить новую веб-страницу.

Разумеется, ожидание загрузки новых страниц занимает какое-то время. При этом от сайтов люди ждут быстроты и интерактивности, как и от компьютерных программ. Одним из вариантов решения данной проблемы – технология под названием Ajax.

Ajax позволяет веб-странице запрашивать веб-сервер и получать ответ, после чего обновляться без необходимости загружать новую страницу.

Термин Ajax появился в 2005 г. с целью охарактеризовать сущность новых сайтов, разработанных Google: Google Maps, Gmail и Google Suggest. Слово сложилось из нескольких: Asynchronous JavaScript и XML (асинхронные JavaScript и XML). Ajax является не официальной технологией типа HTML, JavaScript или CSS. Под этим термином следует понимать сочетание ряда технологий – JavaScript, браузера и веб-сервера – для получения и представления обновленной информации без загрузки новой страницы.

Технология Ajax помогает реализовать следующие вещи:

- показ нового HTML-содержимого без перезагрузки страницы;
- отсылка формы и немедленный показ результата;
- регистрация без необходимости покидать страницу;
- назначение рейтинга (например, на сайте с каталогом музыкальных альбомов или кинофильмов);
- обзор информации базы данных.

Среди приведенных выше задач нет ничего революционного, за исключением упоминания об отсутствии необходимости загружать новую страницу. Тех же результатов можно достичь с помощью обычного HTML и некоторого серверного программирования (например, для сбора данных форм или до-

ступа к информации баз данных). Однако Ajax повышает реактивность веб-страниц и компетентность пользователя при работе с сайтом. По сути, Ajax позволяет создавать сайты, больше напоминающие прикладные программы, нежели веб-страницы.

**Компоненты Ajax**

**Браузер** – очевидно, что он необходим для просмотра веб-страниц и применения JavaScript. В большинство современных браузеров встроен специальный объект, делающий использование Ajax возможным – XMLHttpRequest. Он позволяет JavaScript общаться с веб-сервером и получать в ответ информацию.

**JavaScript** – выполняет главные функции Ajax: посылает запрос на веб-сервер, ожидает ответ, обрабатывает его и (как правило) обновляет страницу, добавляя новые данные или изменяя облик страницы определенным образом.

В зависимости от того, что пользователь хочет от своей программы, JavaScript может посылать информацию заполненной формы, запрашивать дополнительные записи из базы данных или просто отправлять фрагмент данных (типа рейтинга, только что присвоенного книге посетителем).

После отправки данных на сервер программа JavaScript будет готова принять ответ, например, дополнительные записи из базы данных или сообщение типа «Ваш голос учтен». Получив информацию, JavaScript обновляет страницу. Обновление веб-страницы включает манипулирование ее объектной моделью для управления, изменения и удаления тегов и HTML-контента.

**Веб-сервер –** получает запросы от браузера и отправляет ему информацию. Сервер может просто вернуть определенный HTML или открытый текст, а может - документ XML или данные JSON. Например, если сервер получает информацию от формы, он способен добавить ее в базу данных и отослать обратно сообщение, подтверждающее, что данные добавлены.

**Взаимодействие с веб-сервером**

Стержневой компонент программы Ajax – объект XMLHttpRequest, иногда называемый просто XHR. Он встроен в большинство современных браузеров и обеспечивает для JavaScript возможность посылать информацию на веб-сервер и получать данные в ответ. Данный процесс состоит из пяти этапов.

**Этап 1. Создание экземпляра объекта XMLHttpRequest**

В общем виде создание объекта XMLHttpRequest в JavaScript можно записать так:

```
var newXHR = new XMLHttpRequest();
```

Этот код работает в таких браузерах как Firefox, Opera, Chrome и т.д., но не работает в Internet Explorer 6. Эту проблему можно решить, используя разные библиотеки JavaScript, например, JQuery.

**Этап 2. Применение метода XHR open() для определения типа данных, которые будут посылаться, и места, куда они уйдут**

Посылать данные можно двумя способами: GET или POST, они аналогичны тем настройкам, которые использовались в HTML-формах. Метод GET посылает информацию на веб-сервер как часть URL, например, shop.php?productID=34. В рассматриваемом примере данные – это информация после символа «?»: productID=34, указывающая на пару название-значение, где productID является названием, а 34 – значением. Можно представить название как имя поля формы, а значение как данные, которые посетитель вводит в это поле.

Метод POST посылает данные отдельно от URL. Если метод GET обычно используется для получения данных с сервера, то метод POST служит для обновления информации на

сервере (например, чтобы добавить, обновить или удалить запись в базе данных).

Метод open() можно использовать и для выбора страницы на сервере, куда направляются данные. Например, следующий код сообщает объекту XHR, какой метод использовать (GET) и какую страницу на сервере запрашивать:

```
newXHR.open('GET', 'shop.php?productID=34');
```

**Этап 3. Создание функции для обработки результатов**

Когда веб-сервер возвращает результат (новая информация из базы данных, подтверждение о передаче формы или просто тексте сообщение), обычно нужно с ним что-то сделать – написать сообщение «Форма успешно передана» или обновить таблицу на странице. В любом случае придется написать функцию JavaScript для обработки результата.

Также нужно сообщить объекту XHR о функции обратного вызова посредством следующего кода:

```
newXHR.onreadystatechange = myCallbackFunction;
```

**Этап 4. Отправка запроса**

Для отправки запроса на сервер следует применить метод объекта XHR send(). В случае использования метода GET, код выглядит следующим образом:

```
ne wXHR. send(null);
```

null указывает, что не посылается никакой дополнительной информации (данные посылаются с помощью URL). Используя метод POST, следует предоставить данные в качестве параметра метода send():

ne wXHR. send(' q=j avascript');

**Этап 5. Принятие ответа**

После того как сервер обработал запрос, он посылает ответ браузеру. Реально ответ обрабатывает функция обратного вызова, созданная на шаге 3, но и объект XHR получает при этом часть информации, включая статус запроса, текст ответа и, возможно, XML ответа.

Статус ответа – это число, указывающее на характер ответа сервера, например, известный шифр 404, означающий, что запрошенный файл не найден. Если все прошло по плану, можно получить статус 200 или, быть может, 304. В случае ошибки при обработке страницы пользователь получит статус 500 и сообщение «Внутренняя ошибка сервера», а если запрашиваемый файл защищен паролем, то можно получить ошибку 403: Доступ запрещен.

Текстовой ответ, который получает страница, хранится в свойстве объекта XHR responseText. Если сервер отвечает файлом XML, он хранится в свойстве объекта responseXML.

Какие бы данные сервер ни возвращал, все они обрабатываются функцией обратного вызова для обновления вебстраницы. По завершению работы функции обратного вызова заканчивается полный цикл Ajax.

**Работа в Ajax с помощью средств jQuery**

**Использование функции load()**

Простейшей из функций Ajax, предлагаемых jQuery, является load(). Она загружает HTML-файл в указанный элемент на странице (например, страницу с новостями в блок для новостей и т.д.).

Чтобы применить функцию load(), сначала следует воспользоваться селектором j Query для идентификации элемента на странице, где должен быть размещен запрашиваемый HTML; затем необходимо вызвать функцию load() и передать URL страницы, которую нужно получить. Например, есть тег

 с ID headlines и требуется загрузить HTML из файла todays_news.html в данный тег. Сделать это можно следующим образом:

```
$('#headlines').load ('todays_news.html');
```

В процессе выполнения кода браузер запрашивает todays_news.html у веб-сервера. После его загрузки браузер заменяет текущее содержимое тега  с ID headlines данными из нового файла.

HTML-файл может представлять собой полную веб-страницу (включая теги <html>, <head> и <body>) или только ее фрагмент, например, запрашиваемый файл может иметь всего лишь тег <p> и абзац текста. В этом случае этот фрагмент HTML и вставляется на текущую (полную) страницу.

Кроме того, функция load() позволяет определять, какую часть загруженного HTML-файла добавлять на страницу. Например, запрашиваемая страница является обычной страницей с сайта: она включает такие привычные элементы, как баннер, панель навигации и нижний колонтитул. Допустим, пользователя интересует лишь часть содержимого этой страницы, хранящаяся в отдельно взятом теге * * с ID news. Для этого можно использовать следующий код:

```
$('#headlines').loadCtodays_news.html #news');
```

В данном случае браузер загружает страницу *todays_news. html,* но не вставляет все ее содержимое в тег headlines, а извлекает только тег * * (и все, что там есть) с ID news.

**Функции get() и post()**

Функции j *Query* get() и post() являются простыми инструментами для отправки и получения данных с веб-сервера. Как отмечалось ранее, обработка объекта XMLHttpRequest ме-

тодами GET или POST несколько различается. Однако *j Query* учитывает данные различия, поэтому они не играют никакой роли, так что функции get() *и* post() работают практически идентично.

Базовая структура этих функций такова:

*$.get(url, data, callback);*
или:
*$.post(url, data, callback);*

Обе функции поддерживают себя самостоятельно и не связаны с какими-либо элементами на странице, поэтому после знака $ имя селектора не указывается.

Функции get() и post() принимают три аргумента. Аргумент url содержит путь к серверному сценарию обработки данных (например, 'processForm*.php')*. Аргумент data является либо строкой, либо литералом объекта JavaScript, содержащими данные, которые необходимо послать на сервер. Аргумент callback является функцией обратного вызова, которая обрабатывает информацию, возвращаемую сервером.

При выполнении функции get() или post() браузер посылает данные на выбранный URL. Когда сервер возвращает ответные данные, браузер передает их функции обратного вызова, которая обрабатывает эту информацию и обновляет страницу.

**Форматирование данных, посылаемых на сервер**

В любом случае посылаемые данные необходимо отформатировать способом, понятным для функций get() и post(). Их второй аргумент содержит данные, посылаемые на сервер. Их можно форматировать как строку запроса или как литерал объекта JavaScript.

**Строка запроса**

Примером строки запроса может служить:

www.myshop.com/products.php?prodID=18&sessID:=1234

Она содержит две пары имя/значение – prodID=18 и sessID=1234. Она выполняет в принципе ту же задачу, что и создание двух переменных, prodID и sessID, с хранением в них двух значений. Строка запроса является обычным методом передачи информации в URL.

Код для отправки данных на сервер с помощью Ajax может выглядеть так:

```
$.get('products.php', 'prodID=18'); или
$.post('products.php', 'prodID= 18');
```

Если требуется послать на сервер более одной пары имя/значение, следует вставить знак & между каждой из них:

```
$.post('products.php', 'prodID=l 8&sessID=1234');
```

Если значения параметра включают в себя пробел и знак & их нужно заменять кодами этих символов. Пробел имеет код %20, знак & – %26.

```
$.post('products.php', 'prodName=Fish%20%26%20Chips');
```

Можно также воспользоваться методом encodeURIComponent(), который возвращает правильно зашифрованную строку, например:

```
var queryString = 'prodName^' + encodeURIComponent('Fish & Chips'); $.post('products.php', queryString);
```

**Литерал объекта**

Для коротких и простых фрагментов данных (не содержащих знаки пунктуации) метод строки запроса работает хорошо. Но более надежным способом, поддерживаемым функ-

циями j Query get() и post(), является использование литерала объекта для хранения данных, литерал объекта (объектная константа) – это метод JavaScript для хранения пар имя/значение. Базовая структура литерала объекта такова:

```
{
имя1: ‘значение 1’,
имя2: ‘значение2’
}
```

Можно передавать литерал объекта непосредственно функциям get() или post():

```
$.post('products.php',  { prodID: 18 });
```

Можно либо передать литерал объекта непосредственно функции get() или post(), либо сначала сохранить его в переменной, а затем передать ее функции get() или post():

```
var data= { prodID: 18 }; $.post('products.php', data);
```

Разумеется, можно включить любое количество пар имя/значение в объект, передаваемый функции get() или post():

```
var data = {prodID: 18, sessID: '1234'};
$.post('products.php', data);
```

**Функция jQuery serialize()**

Создание строки запроса или литерала объекта для всех пар имя/значение формы может оказаться весьма трудным. Требуется извлечь имя и значение для каждого элемента формы, а затем объединить их, чтобы составить единую длинную строку запроса или большой литерал объекта JavaScript. К счастью, jQuery имеет функцию, облегчающую процедуру конвертации данных формы в информацию, которую могут использовать функции get() и post().

Функция serialize() подходит для любой формы (или даже выбранных полей). Для ее применения сначала следует создать выборку jQuery, включающую форму, а затем присоединить функцию serialize(). Например, так можно получить всю информацию введенную пользователем в форме с ID login:

```
var formData = $('#login').serialize();
```

Теперь можно послать эти данные на сервер:

var formData = $('#login').serialize(); $.get('login.php', formData, loginResults);

Данный код посылает форму в файл login.php, используя метод GET. Последний аргумент функции get() – loginResults – представляет собой функцию обратного вызова: берет данные, возвращаемые с сервера, и обрабатывает их.

**Обработка данных с сервера**

Функция обратного вызова определенным образом обрабатывает данные и, как правило, обновляет веб-страницу, заменяя посланные сведения результатами, полученными с сервера, или, например, просто выводит сообщение на странице. Процесс обновления содержимого страницы облегчается функциями j Query html() и text().

Чтобы разобраться с полным циклом запрос/ответ, рассмотрим пример с присвоением рейтингов фильмам. Посетитель может выбрать рейтинг, щелкнув на одной из пяти ссылок, каждая из которых указывает на соответствующий рейтинг. После щелчка на ссылке номер рейтинга и ID фильма посылаются на сервер, программа которого вносит эту информацию в базу данных, а затем возвращает среднее значение рейтинга для данного фильма. Это среднее значение показывается на веб-странице.

Чтобы на данной странице обойтись без JavaScript, каждая ссылка указывает на динамическую серверную страницу, которая может обрабатывать рейтинг, присвоенный посетителем. Например, ссылка пятизвездочного (высшего) рейтинга может иметь вид: rate.php?rate=5&moviel23. Имя серверного файла, обрабатывающего рейтинги, – rate.php, тогда как строка ?rate=5&movie=123 включает два фрагмента информации для сервера: рейтинг (rate=5) и номер, который идентифицирует фильм (movie=123). Можно использовать JavaScript для перехвата нажатий на ссылки и преобразования их в Ajax-вызовы на сервер:

```
1  $('#message a').click(function() {
2  var href=$(this).attr('href');
3  var querystring=href.slice(href.indexOf('?')+l);
4  $.get('rate.php', querystring, processResponse);
5  return false;   // блокировать переход по ссылке
6  } ) ;
```

Строка 1 выбирает каждую ссылку (тег <a>) внутри другого тега с ID message (в данном примере каждая ссылка, используемая для присвоения рейтинга фильму, находится внутри тега  с ID message*)* событию нажатия для каждой их этих ссылок присваивается функция

Строка 2 извлекает атрибут **href** ссылки, так, например, переменная href может содержать URL типа rate.php?rate=5&movie=123.

Строка 3 извлекает часть кода после «?» в URL, используя метод *slice()* для получения части строки и метод indexOf() для определения положения «?» (эта информация используется slice() для определения начала разрезания).

Строка 4 представляет Ajax-запрос. Используя метод GET,запрос, содержащий строку запроса для ссылки, направляется серверному файлу rate.php. Результаты передаются функции обратного вызова processResponse.

Строка 5 прекращает стандартное поведение ссылки и не дает браузеру загрузить связанную с ней страницу.

Теперь следует создать функцию обратного вызова. Она принимает данные и строку со статусом ответа (success, если сервер возвратил данные). В данном случае это функция processResponse. Код для получения ответа с сервера может глядеть так:

```
1   function processResponse(data, status) {
2   var newHTML;
```

```
3        if (status=="success") {
// Ваш голос учтен
4        newHTML = '<h2>Your vote is counted</h2>';
// Средний рейтинг для этого фильма составляет
5        newHTML += '<p>The average rating for this movie
is ';
6        newHTML += data + '.</p>';
7        } else {
// Случилась ошибка
8        newHTML=’<h2>There has been an error.</h2>’;
//Пожалуйста, повторите попытку позже
9        newHTML+=’<p>Please try again later.</p>';
10       }
11       $(‘#message’).html(newHTML);
12       }
```

Функция принимает два аргумента: data и status. Строка 2 создает новую переменную, содержащую текст HTML, который будет показан на странице (например, «Ваш голос учтен»). В строке 3 условная инструкция проверяет, был ли ответ сервера свидетельствующим об успехе, и если да, то переменная newHTML пополняется новым HTML, включая теги <h2> и <p>. Ответ сервера не проявляется вплоть до строки 6 – здесь опция ответа (хранящаяся в переменной data) добавляется в newHTML. В данном случае сервер возвращает строку со средним рейтингом фильма, например, '3 stars'.

Инструкция else создает сообщение об ошибке в случае, если сервер не смог успешно ответить на запрос.

Наконец строка 11 изменяет HTML на веб-странице с помощью функции jQuery html() путем замены содержимого тега  с ID *message* новым HTML.

В данном примере функция обратного вызова была определена за пределами функции get(); однако можно воспользоваться анонимной функцией, если нужно держать вместе весь код Ajax:

```
$.get('file.php', data, function(data, status) {
//код функции обратного вызова помещается сюда
});
```

## 2.6. Работа с объектами в JavaScript

### Объекты в JavaScript

Объект является фундаментальным типом данных в языке JavaScript. Объект – это составное значение: он объединяет в себе набор значений (простых значений или других объектов) и позволяет сохранять или извлекать эти значения по именам. Объект является неупорядоченной коллекций свойств, каждое из которых имеет имя и значение. Имена свойств являются строками, поэтому можно сказать, что объекты отображают строки в значения.

Объекты в языке JavaScript являются динамическими – обычно они позволяют добавлять и удалять свойства – но они могут также использоваться для имитации статических объектов и «структур», которые имеются в языках программирования со статической системой типов.

Любое значение в языке JavaScript, не являющееся строкой, числом, true, false, null или undefined, является объектом. И даже строки, числа и логические значения, не являющиеся объектами, могут вести себя как неизменяемые объекты.

Свойство имеет имя и значение. Именем свойства может быть любая строка, включая и пустую строку, но объект не может иметь два свойства с одинаковыми именами.

### Создание объектов

Объекты можно создавать с помощью литералов объектов, ключевого слова new и функции **Object.create()**.

**Литералы.** Самый простой способ создать объект заключается во включении в программу *литерала объекта* – за-

ключенного в фигурные скобки списка свойств (пар имя/значение), разделенных запятыми. Примеры:

```
var empty = {};                          //Объект без свойств
var point = { x:0 , y:0};                //Два свойства
var book = {              //Имена свойств в виде литералов
                "main title": "JavaScript",
                "sub-title": "The Definitive Guide",
                author : {//Объект как значение свойства
        firstname : "David",
        surname : "Flanagan"
}
```

Литерал объекта – это выражение, которое создает и инициализирует новый объект всякий раз, когда производится вычисление этого выражения. Значение каждого свойства вычисляется заново, когда вычисляется значение литерала. Это означает, что с помощью единственного литерала объекта можно создать множество новых объектов, если этот литерал поместить в тело цикла или функции, которая будет вызываться многократно, и что значения свойств этих объектов будут отличаться друг от друга.

**Оператор new.** Оператор **new** создает и инициализирует новый объект. За этим оператором должно следовать имя функции. Функция, используемая таким способом, называется *конструктором* и служит для инициализации вновь созданного объекта. Базовый JavaScript включает множество встроенных конструкторов для создания объектов базового языка. Примеры:

```
var o = new Object(); //Создание нового пустого объекта
var a = new Array();  //Создание пустого массива
var d = new Date();     //Создание объекта текущего времени
```

Помимо этих встроенных конструкторов имеется возможность определять свои собственные функции-конструкторы для инициализации вновь создаваемых объектов.

**Прототипы.** Почти каждый объект в языке JavaScript имеет второй объект, ассоциированный с ним. Этот второй объект называется прототипом, и первый объект наследует от прототипа его свойства.

Все объекты, созданные с помощью литералов объектов, имеют один и тот же объект-прототип, на который в программе JavaScript можно сослаться так: **Object.prototype**. Объекты, созданные с помощью ключевого слова **new** и вызова конструктора, в качестве прототипа получают значение свойства **prototype** функции-конструктора. Поэтому объект, созданный выражением **new Object()**, наследует свойства объекта **Object.prototype**, как если он был создан с помощью литерала в фигурных скобках {}. Аналогично прототипом объекта, созданного выражением **new Array()**, является **Array.prototype**, а прототипом объекта, созданного выражением **new Date()**, является **Date.prototype**.

**Object.prototype** – один из немногих объектов, которые не имеют прототипа: у него нет унаследованных свойств. Другие объекты-прототипы являются самыми обычными объектами, имеющими собственные прототипы. Все встроенные конструкторы (и большинство пользовательских конструкторов) наследуют прототип **Object.prototype**. Например, **Date.prototype** наследует свойства от **Object.prototype**, поэтому объект **Date**, созданный выражением **new Date()**, наследует свойства от обоих прототипов, **Date.prototype** и **Object.prototype**. Такая связанная последовательность объектов-прототипов называется *цепочкой прототипов*.

**Метод Object.create().** Метод **Object.create()** создает новый объект и использует свой первый аргумент в качестве прототипа этого объекта. Дополнительно **Object.create()** может

принимать второй необязательный аргумент, описывающий свойства нового объекта.

**Object.create()** является статической функцией, а не методом, вызываемым относительно некоторого конкретного объекта. Чтобы вызвать эту функцию, достаточно передать ей желаемый объект-прототип:

```
var o1 = Object.create({ x:1, y:2});          //o1 наследует
//свойства x и y
```

Чтобы создать объект, не имеющий прототипа, можно передать значение null, но в этом в случае вновь созданный объект не унаследует ни каких-либо свойств, ни базовых методов, таких как **toString()** и т.д.:

```
var o2 = Object.create(null);          //o2 не наследует
//ни свойств, ни методов
```

Если в программе требуется создать обычный пустой объект, то можно передать в качестве аргумента Object.prototype:

```
var o3 = Object.create(Object.prototype);
```

**Получение и изменение свойств**

Получить значение свойства можно с помощью операторов точки (.) и квадратных скобок ([]). Примеры:

```
var author = book.author;
var name = author.surname;
var title = book[“main title”];
```

Чтобы создать новое свойство или изменить значение существующего свойства, также используются операторы точки и квадратные скобки, как в операциях чтения значений свойств, но само выражение помещается слева от оператора присваивания:

```
book.edition = 6;
book[“main title”] = “ECMAScript”;
```

**Наследование**

Объекты в языке JavaScript обладают множеством «собственных свойств» и могут также наследовать множество свойств от объекта-прототипа. Для создания объектов с определенными прототипами можно использовать функцию inherit(). Пример:

```
var o = {};            //o наследует методы объекта Object.protype
o.x = 1;               //и обладает собственным свойством x
var p = inherit(o);    //p наследует свойства объектов
                       //o и Object.protype
p.y = 2;               //и обладает собственным свойством y
var q = inherit(p);    //q наследует свойства объектов
                       //p, o и Object.prototype
q.z = 3;               //и обладает собственным свойством z
```

Предположим, что программа присваивает некоторое значение свойству **x** объекта **o**. Если объект **o** уже имеет собственное свойство (не унаследованное) с именем **x**, то операция присваивания просто изменит значение существующего свойства. В противном случае в объекте **o** будет создано новое свойство с именем **x**. Если прежде объект **o** наследовал свойство **x**, унаследованное свойство теперь окажется скрыто вновь созданным собственным свойством с тем же именем.

**Удаление свойств**

Оператор **delete** удаляет свойство из объекта. Его единственный операнд должен быть выражением обращения к свойству. Примеры:

```
delete book.author;          //Теперь объект book
//не имеет свойства author
```

```
delete book[“main title”];     //Теперь он не имеет свойства
//“main title”
```

Оператор **delete** удаляет только собственные свойства и не удаляет унаследованные (Чтобы удалить унаследованное свойство, необходимо удалить его в объекте-прототипе, в котором оно определено. Такая операция затронет все объекты, наследующие этот прототип).

Выражение **delete** возвращает значение **true** в случае успешного удаления свойства или когда операция удаления не привела к изменению объекта (например, при попытке удалить несуществующие свойство). Выражение **delete** также возвращает true, когда этому оператору передается выражение, не являющееся выражением обращения к свойству.

**Классы в JavaScript**

Классы в языке JavaScript основаны на использовании механизма наследования прототипов. Если два объекта наследуют свойства от одного и того же объекта-прототипа, говорят, что они принадлежат одному классу.

Если два объекта наследуют один и тот же прототип, обычно (но не обязательно) это означает, что они были созданы и инициализированы с помощью одного конструктора.

Конструктор – это функция, предназначенная для инициализации вновь созданных объектов. Применение ключевого слова **new** при вызове конструктора автоматически создает новый объект, поэтому конструктору остается только инициализировать свойства этого нового объекта. Это означает, что все объекты, созданные с помощью конструктора, наследуют один и тот же объект-прототип и соответственно, являются членами одного и того же класса.

Пример реализации класса:

```
<script type="text/javascript">
function Train (num, p_naz, v_otpr) {
```

```
            this.num = num;
            this.p_naz = p_naz;
            this.v_otpr = v_otpr;
        }
        Train.prototype = {
            PrintTrain: function() {
                var
tr='<tr><td>'+this.num+'</td><td>'+this.p_naz+
         '</td><td>'+this.v_otpr+'</td></tr>';
                return tr;
            },
            PrintRec: function() {
                var tr = "<br>Номер поезда : "+this.num+"
        Пункт назначения : "+this.p_naz+" Время отправления
:"+this.v_otpr;
                return tr;
            }
        };
        var t = new Train(12, 'Казань', '12:30');

        </script>
```

В данном примере создается класс, содержащий следующие сведения о поезде:

– номер поезда;

– пункт назначения;

– время отправления.

В класс добавлены два метода. Первый распечатывает все поля объекта класса в виде строки таблицы, второй – в виде текстовой строки.

В нижней части скрипта вызывается конструктор **Train()** с использованием ключевого слова **new**. Новый объект создается автоматически перед вызовом конструктора и доступен как значение **this**. Конструктору **Train()** остается лишь инициализировать его.

Все свойства объекта **Train()** являются прототипами. Выражение вызова конструктора **Train()** автоматически использует свойство **Train.prototype** как прототип нового объекта **Train**.

**Конструктор и идентификация класса.** Важным применением конструкторов является их использование в операторе **instanceof** при проверке принадлежности объекта классу. Если имеется объект **t**, и необходимо проверить, является ли он объектом класса **Train**, такую проверку можно выполнить так:

```
t instanceof  Train          //вернет true, если t наследует Train.prototype
```

В действительности оператор **instanceof** не проверяет, был ли объект **t** инициализирован конструктором **Train**. Он проверяет, наследует ли этот объект свойство **Train.prototype**.

Роль конструктора в JavaScript может играть любая функция, поскольку выражению вызова конструктора необходимо лишь свойство **prototype**. Следовательно, любая функция автоматически получает свойство **prototype**. Значением этого свойства является объект, который имеет единственное неперечислимое свойство **constructor**. Значением свойства **constructor** является объект функции:

```
var F = function(){};  //Это объект функции
var p = F.prototype;   //Это объект-прототип, связанный с ней
var c = p.constructor; //Это функция, связанная с протипом
c === F;               //Вернет true
```

Наличие предопределенного объекта-прототипа со свойством **constructor** означает, что объекты обычно наследуют свойство constructor, которое ссылается на их конструкторы.

Поскольку конструкторы играют роль идентификаторов класса, свойство constructor определяет класс объекта:

```
var o =  new F();                    //Создать объект класса F
o.constructor === F;  //Вернет true:
//свойство constructor определяет класс.
```

В предыдущем примере класс Train, замещает предопределенный проект **Train.prototype** своим собственным. А новый объект-прототип не имеет свойства **constructor**. По этой причине экземпляры класса **Train** не имеют свойства constructor. Решить эту проблему можно, явно добавив конструктор в прототип:

```
Train.prototype = {
constructor: Train,      //Явно установить обратную ссылку на
 // конструктор
        PrintTrain: function() {…},
        PrintRec: function() {…}

};
```

# 3. ВВЕДЕНИЕ В PHP И MYSQL

## 3.1. Введение в PHP

### Общие сведения о PHP

PHP предоставляет возможности программирования на стороне сервера, давая веб-разработчику доступ к серверу, на котором размещен сайт, и ко всему, что там хранится и запускается. PHP также дает доступ к файловой системе сервера, что делает возможным чтение и запись информации независимо от баз данных. PHP можно также использовать для создания веб-страниц, отправляемых клиенту, при этом страницы могут быть настроены на клиентской стороне для конкретного поль-

зователя, например для отображения определенного языка или соответствующего уровня безопасности.

По умолчанию PHP-документу присваивается расширение .php. Когда веб-сервер встречает в запрашиваемом файле это расширение, он автоматически передает файл PHP-процессору. После обработки пользователю возвращается файл, пригодный для отображения в браузере. В простейшем случае это будет только код HTML.

Для запуска команд PHP нужно изучить новый тег. Его открывающая часть имеет следующий вид:

<?php

Внутри тега помещаются целые фрагменты кода PHP, и они заканчиваются только когда встречается закрывающая часть тега, имеющая следующий вид:

?>

Классическая программа Hello World на PHP может иметь следующий вид:

```
<?php
        echo “Hello World”
?>
```

Данный тег очень гибок в использовании. Некоторые программисты открывают данный тег в начале документа и закрывают его в самом конце и выводят любой код HTML путем непосредственного использования команды PHP.

Другие программисты предпочитают помещать в эти теги как можно меньшие фрагменты кода PHP, и именно в тех местах, где нужно воспользоваться динамическими сценариями, а весь остальной документ составлять из стандартного кода HTML.

PHP предоставляет ряд возможностей, которые одновременно являются и мощными, и удобными. К таким средствам можно отнести возможность отобразить информацию в любом месте скрипта, несколько способов добавления комментариев, а также набор полезных соглашений по программированию.

**Отображение информации**

При отладке скрипта часто полезно отобразить комментарий или переменную во время действия скрипта. PHP предоставляет два способа осуществить это действие – при помощи функций *echo()* и *print(),* которые очень похожи и во многих случаях действуют одинаково. Например:

```
<html>
        <body>
<h1>Демонстрация функций Print и Echo</hl>
<?php
print "Это результат действия Print <br />";
echo "Это результат действия Echo";
?>
</body>
</html>
```

Аргументы функций совсем не обязательно заключать в круглые скобки, хотя при желании вы можете использовать скобки. Кавычки обязательны, за исключением случая, когда аргумент является числом. Допускается использовать одинарные и двойные кавычки.

Различие между ними заключается в том, что *print()* – это функция, возвращающая значение, а *echo* значения не возвращает. Поэтому *echo()* в общем случае работает быстрее *print()*, но ее нельзя использовать как часть более сложного выражения.

**Комментирование скрипта**

PHP позволяет добавлять комментарии к скрипту тремя способами,:

1) на одной строке, начиная с символа #

*#Это комментарий*

2) на одной строке, начиная с //

*//Это тоже комментарий*

3) на нескольких строках, заключенных между /* */

```
/*Это комментарий, который может
быть на нескольких строках. */
```

**Соглашения по программированию**

PHP не очень строг в отношении соглашений по программированию. Однако есть определенные правила написания кода PHP:

– в конце каждой инструкции PHP должна быть поставлена точка с запятой (;). Единственное исключение – последняя инструкция перед закрывающим тегом ?>, которая может заканчиваться точкой с запятой, но это не является обязательным;

– текст, любая комбинация букв и цифр, также называемая *строкой*, в качестве аргумента должны быть заключены в кавычки, одинарные (' ') или двойные (" "). Обе кавычки в паре должны быть одинаковыми;

– любые допустимые числа, включая числа с десятичной точкой, можно не заключать в кавычки;

– несколько аргументов разделяются при помощи запятых (,);

– при использовании большинства функций аргументы требуется заключать в круглые скобки.

Как и во многих других языках в PHP существуют специальные символы, которые задаются с помощью добавления обратной косой черты перед ними. Наиболее распространенными из них являются:

\" – символ двойной кавычки;
\’ – символ одинарной кавычки;
\\ – обратная косая черта;
\$ – символ доллара;
\r – возврат каретки;
\n – перевод строки;

\t – табуляция.

**Основной синтаксис PHP**

**Типы данных.** PHP не так чувствителен к типам данных, как другие языки. Обычно не нужно указывать тип данных, например целый или строковый. PHP определит тип самостоятельно по контексту, в котором используются данные.

Тем не менее, желательно знать, с каким типами данных возможно работать в PHP.

Допустимые типы данных приведены в табл. 11.

Таблица 11

Типы данных PHP

| Тип данных | Наименование | Описание | Примеры |
|---|---|---|---|
| Массивы | array | Набор из двух или более значений, которые могут принадлежать к любому из типов данных (кроме объекта или ресурса), представленный в виде списка элементов, разделенных запятыми | 98101, ‘РФ’, ‘Воронеж’, ‘ул. 9 Января’ |
| Булевский тип | bool | «TRUE» или «FAL.SE»; не чувствительны к регистру, но обычно в верхнем регистре | TRUE, FALSE |
| Числа с плавающей точкой | float | Дробные десятичные числа; могут быть отрицательными, использовать экспоненциальное представление | 7.34,-21.89,2.31e3 |
| Целые числа | int | Целые числа без дробной части; могут быть отрицательными | 43, 928, -4 |
| Пустой тип | null | Отсутствие любого значения | NULL |

Продолжение табл. 11

| | | | |
|---|---|---|---|
| Строки | string | Последовательность символов, заключенная в одинарные или двойные кавычки | "Михаил", 'Воронеж', "ул. 9 января, 110" |
| Объекты | object | Объект с набором свойств и методов | |
| Ресурсы | resource | Ссылка на внешний элемент; обычно используется с MySQL | |

Можно выделить следующие особенности типов данных:

– булево значение *FALSE* эквивалентно целому числу 0, числу с плавающей запятой 0.0, пустой строке или строке "0", массиву с нулем элементов или *NULL.* Все остальные значения соответствуют *TRUE;*

– целые числа по умолчанию являются десятичными (с основанием 10). Чтобы сделать число восьмеричным (с основанием 8), нужно добавить перед числом 0 (ноль), например 02 или 06. Чтобы сделать число шестнадцатеричным (с основанием 16), нужно добавить перед числом 0x, например 0x4 или 0x8*;*

– большие целые числа (больше чем *2 147 483 647)* считаются числами с плавающей точкой;

– если выполняется деление двух целых чисел, то в результате получается число с плавающей точкой, за исключением случаев, когда деление выполняется без остатка;

– числа с плавающей точкой не являются точными до последней цифры из-за бесконечной прогрессии в дробях типа 1/3. Поэтому нельзя выполнять сравнение двух чисел с плавающей точкой;

– строка, содержащая число (целое или с плавающей точкой), расположенное сразу после левой кавычки, может быть использована как число. Например, строки *"18.2"* и *"4*

*машины"* можно использовать как числа, в то время как строку *"его 4 машины"* как число использовать нельзя.

**Переменные и константы**

При написании скрипта PHP необходимо присваивать имена отдельным элементам, чтобы можно было использовать эти элементы повторно. Существует два общепринятых типа элементов, которым вы можете присваивать имена:

– переменные, которые могут содержать различные значения во время выполнения скрипта. Они начинаются со знака доллара ($), после которого ставится присвоенное вами имя;

– константы, которые содержат одно и то же значение на протяжении всего времени выполнения скрипта. Рекомендуется всем константам присваивать имена в верхнем регистре, например *NULL* является константой.

При создании переменных следует придерживаться пяти правил:

– имена переменных после знака $ должны начинаться с буквы или с символа подчеркивания;

– имена переменных могут содержать только символы: a-z, A-Z, 0-9 и _ (подчеркивание);

– имена переменных не должны содержать пробелов – если имя переменной нужно составить из более чем одного слова, то в качестве разделителя следует использовать символ подчеркивания;

– имена переменных чувствительны к регистру символов;

– имена переменных не должны совпадать с зарезервированными словами.

В табл. 12 перечислены ключевые слова, которые нельзя использовать для наименования переменных, констант или других обозначений в PHP.

Таблица 12

Ключевые слова PHP

| | | | | | |
|---|---|---|---|---|---|
| abstract | and | array | as | bool | break |
| case | catch | class | clone | const | continue |
| declare | default | do | echo | else | elseif |
| enddeclare | endfor | endforeach | endif | endswitch | endwhile |
| extends | false | final | float | for | foreach |
| function | global | goto | if | implements | int |
| interface | instanceof | namespace | new | null | object |
| or | print | private | protected | public | static |

Имена констант формируются примерно по тем же правилам, что и имена переменных. Для создания константы лучше всего использовать функцию define():

```
define("CONSTANT", "Hello World");
echo CONSTANT;
```

Различия между константами и переменными:

– у констант нет приставки в виде знака доллара ($);

– константы могут быть определены и доступны в любом месте без учета области видимости;

– константы не могут быть переопределены или аннулированы после первоначального объявления; и

– константы могут иметь только скалярные значения.

Начиная с версии PHP 5.3.0 можно использовать для объявления константы ключевое слово *const*:

```
const USER_NAME = "Fred Smith";
echo USER_NAME;
```

**Операторы**

Для присвоения значений после создания переменной или константы в PHP определено множество операторов различного типа, как показано в табл. 13.

Таблица 13

Операторы PHP

| Оператор | Наименование | Пример | Объяснение |
|---|---|---|---|
| Арифметические | | | Выполняют арифметические действия |
| + | Сложение | *$a* + $b | Сумма |
| - | Вычитание | *$a* - $b | Разность |
| * | Умножение | *$a* * $b | Произведение |
| / | Деление | *$a* / $b | Частное |
| % | Деление по модулю | *$a* % $b | Остаток от деления |
| Присваивания | | | Заменяет значение другим |
| = | Присвоение | *$a* = 7; | Устанавливает *$a* равным 7 |
| += | Инкремент | *$a* = 7;<br>$a += 2;<br>*возвращает* 9 | Увеличивает *$a* на 2 |
| .= | Присвоение с конкатенацией | *$a = "Павел ";*<br>*$a .= "Петров";*<br>*возвращает "Павел Петров"* | Добавляет строку к существующей строке |
| []= | Присвоение с добавлением | *$array []= $something* | Добавляет *$something* в конец массива |

Продолжение табл. 13

| | | | |
|---|---|---|---|
| Побитовые | | | Включают или выключают биты целого числа |
| & | И | *$a &* $b | Устанавливает биты, которые установлены и в *$a,* и в $b |
| \| | ИЛИ | *$a* \| $b | Устанавливает биты, которые установлены или в *$a,* или в $b |
| ^ | Исключающее ИЛИ | *$a* ^ $b | Устанавливает биты, которые установлены или в *$a,* или в $b, но не в обоих |
| ~ | Отрицание | *~$a* | Устанавливает биты, которые не установлены в *$a,* и сбрасывает биты, которые установлены |
| << | Сдвиг влево | *$a* << $Ь | Сдвигает биты *$a* на $b шагов влево (один шаг является умножением на 2) |
| >> | | *$a* >> *$*b | Сдвигает биты *$a* на $b шагов вправо (один шаг является делением на 2) |
| Сравнения | | | Сравнивают два значения |
| == | Равно | *$a* = $b | Возвращает *TRHE,* если *$a* равно $b |
| === | Идентично | *$a* === $b | Возвращает *TRUE,* если *$a* идентично $b |
| != или <> | Не равно | *$a* != $b или $a <> $b | Возвращает *TRUE,* если *$a* не равно $b |
| !== | Не идентично | *$a*! = $b | Возвращает *TRUE,* если *$a* не идентично $b |

Продолжение табл. 13

| | | | |
|---|---|---|---|
| < | Меньше чем | *$a < $b* | Возвращает *TRUE,* если *$a* меньше чем *$b* |
| > | Больше чем | *$a > $b* | Возвращает *TRUE,* если *$a* больше чем *$b* |
| < = | Меньше или равно | *$a <= $b* | Возвращает *TRUE,* если *$a* меньше или равно *$b* |
| >= | Больше или равно | *$a >= $b* | Возвращает *TRUE,* если *$a* больше или равно *$b* |
| Инкрементные | | | Изменяют значение на единицу |
| ++ | Инкремент | *++$a*<br>*$a++* | Добавляет единицу к *$a* и возвращает *$a.* Возвращает *$a* , затем добавляет единицу к *$a* |
| -- | Декремент | *--$a*<br>*$a--* | Вычитает единицу из *$a* и возвращает *$a.* Возвращает *$a* , затем вычитает единицу<br>из *$a* |
| Логические | | | Возвращают логическое значение выражения |
| *and*<br>**&&** | И | *$a and $b* | Возвращает *TRUE,* если и *$a,* и *$b* являются *TRUE* |
| *or*<br>\|\| | ИЛИ | *$a or $b* | Возвращает *TRUE,* если или *$a,* или *$b* является *TRUE* |
| *xor* | Исключающее Или | *$a xor $b* | Возвращает *TRUE,* если или *$a,* или *$b* является *TRUE,* но не оба |
| | Отрицание | *!$a* | Возвращает *TRUE,* если *$a* не является *TRUE* |

Окончание табл. 13

| Другие | | | |
|---|---|---|---|
| @ | Отключение ошибки | *@my function()* | Отключает отображение ошибок, созданных данным выражением |
| . | Конкатенация строк | *$y="Новый"; $z=" текст"; echo $y.$z; будет выведено «Новый текст»* | Соединяет между собой строки $a и $b |

Следует отметить ряд особенностей связанных с операторами.

Если несколько операторов комбинируется в одном выражении, они будут выполнены в соответствии с представленным ниже приоритетом, начиная с наивысшего: ++, --,

!, ~, @, *, /, %, +, -, ., <<, >>, <, <=, >, >=, <>, ==, !=, ===, !==, &, ^, |, &&, ||, =, +=, -=, .=, and, xor, or. Порядок выполнения можно изменить при помощи круглых скобок.

Результат деления по модулю (%) не определяет, сколько процентов первое число составляет от второго; результатом данной операции является остаток от целочисленного деления первого числа на второе;

Знак равенства (=) обозначает не «равно», а «присвоить» или «заменить». Для сравнений типа «является ли a равным b» следует использовать удвоенный знак равенства (==);

Если *$a = 2* и *$b = 2.0,* то переменные *$a* и *$b* не являются идентичными, поскольку первая – это целое число, а вторая – число с плавающей точкой.

**Инструкции и выражения**

Скрипт PHP может содержать комментарии и инструкции. *Инструкцией* является все, что находится между точками с запятой или открывающим и закрывающим тегами PHP. Ча-

сто инструкция представляет собой одну строку кода, оканчивающуюся точкой с запятой, но можно расположить несколько инструкций в одной строке; кроме того, одна инструкция может занимать несколько строк. В большинстве инструкций содержится одно или более выражений, однако некоторые инструкции, например break или else**,** являются единичным ключевым словом.

*Выражением* является все, что обладает значением или оценивается как значение. Значения представляют собой то, что может быть присвоено переменной, поэтому они могут относиться к любому типу данных: к целым числам, числам с плавающей запятой, строкам, булеву типу, массивам или объектам. Пустой тип представляет собой отсутствие значения, но в данном случае он тоже является значением.

Выражения составляют блоки, которые могут быть использованы для построения других выражений. Например, *$a = 2* содержит три выражения: *2, $a* и *$a = 2*.

**Функции**

*Функция* – это часть скрипта, которая что-то выполняет и может быть повторно вызвана из большего скрипта. Внутренние функции уже существуют, а пользовательские представляют собой функции, которые создает программист при написании программного кода. Одни функции требуют наличия *аргументов***,** то есть значений, передаваемых функциям для вычисления возвращаемого значения; другие при вызове только возвращают значение.

Пользовательская функция определяется при помощи ключевых слов *function* и *return* в следующем виде:

function *имя*($аргумент1, $аргумент2,...)
*{*
*[любые инструкции PHP];*
return *$возвращаемоеЗначение;*
*}*

Для имени функции используются те же правила наименования, что и для имен переменных, констант и других обозначений. Имена функций чувствительны к регистру, могут начинаться с букв a-z, A-Z или символа подчеркивания (_), быть любой длины и содержать буквы, цифры, символы подчеркивания, а также символы западноевропейских алфавитов. Знак $ при вызове функции не пишется.

## 3.2. Базовые конструкции языка PHP. Работа с массивами

### Базовые конструкции языка PHP

**Инструкции if/else.** Инструкции *if /else* являются основной конструкцией для принятия решений в PHP. При помощи этой инструкции возможно задать следующее условие: если некоторое выражение равно *TRUE,* будет выполнена некоторая группа инструкций, в противном случае выполнится другая группа. Конструкция принимает форму, представленную ниже:

```
if (условное выражение) {
инструкции, выполняемые, если условие равно TRUE;
}
else {
инструкции, выполняемые, если условие равно FALSE;
}
```

Группа инструкций *else* является необязательной и необходима только в том случае, если требуется выполнить действие, отличающееся от продолжения скрипта, когда условное выражение равно *FALSE.* Вы также можете создавать вложенные инструкции *if /else* при помощи *elseif,* например:

```
if (условное выражение) {
инструкции, выполняемые, если условие равно TRUE;
}
elseif (второе условное выражение) {
инструкции, выполняемые, если второе условие равно TRUE;
}
```

**else {**

инструкции, выполняемые, если второе условие равно FALSE;

}

Во всех случаях результатом условного выражения должно быть булево значение *TRUE* или *FALSE (*1 или 0). Если при этом существует переменная, содержащая значение, отличное от *NULL, FALSE* или 0, такая переменная равна *TRUE.*

Многие условные выражения являются сравнениями, которые проверяют, равны ли два элемента или один из них больше другого. Следует помнить, что при проверке равенства в PHP необходимо использовать удвоенный знак равенства (==), а не единичный знак, который означает присвоение.

**Тернарный оператор.** Сокращенным методом для принятия решения *if /else* в PHP является использование тернарного оператора (? :), где ? заменяет собой *if* и ставится после условного выражения, а : заменяет *else.* Представленная ниже инструкция производит те же результаты, что и расположенная следом инструкция *if.*

```
echo ($a) ? "Истина" : "Ложь", "<br />";
if ($a) {
echo "Истина", "<br />";
}
else {
echo "Ложь", "<br />";
}
```

Соединение нескольких инструкций *if,* как при использовании *elseif,* при применении тернарного оператора не рекомендуется.

**Инструкции while и do-white.** Инструкции *while* и *do-while* являются конструкциями для организации циклов, позволяющими повторно выполнять часть кода до того момента, когда условное выражение перестанет быть равным *TRUE.*

Инструкция *while,* являющаяся основой этого набора, принимает следующие формы:

**while** (условное выражение) (
инструкции, выполняемые, пока условие равно TRUE;
}

**while** (условное выражение) :
инструкции, выполняемые, пока условие равно TRUE;
**endwhile;**

Инструкция *do-while* похожа на инструкцию *while,* только условное выражение расположено не в начале, а в конце данной инструкции. Инструкция *do-while* принимает следующую форму:

**do {**
инструкции, выполняемые, пока условие равно TRUE;
}
**while** (условное выражение);

**Инструкции for и foreach.** Инструкции for и foreach являются дополнительными конструкциями для организации цикла. Инструкция for, схожая со своими аналогами в других языках, размещает инициализацию счетчика, ограничивающее условие и приращение счетчика в виде последовательности выражений сразу за ключевым словом *for.* Данная инструкция может принимать одну из следующих форм:

**for** (инициализация; условное выражение; выражение приращения)
{
инструкции, выполняемые, пока условие равно TRUE;
}

**for** (инициализация; условное выражение; выражение приращения):
инструкции, выполняемые, пока условие равно TRUE;
**endfor;**

В своей основной форме выражение *for* может быть записано, например, таким образом: *for ($i = 1; $i <= 5; $i++)*, где *$i*++увеличивает *$i* после выполнения.

**Инструкция switch.** Инструкция *switch* напоминает последовательность инструкций *if/else.* Эта инструкция используется при необходимости сравнить значение переменной с некоторым набором различных значений и в зависимости от результата сравнения выполнить какое-либо действие. Совместно с инструкцией *switch* используется три ключевых слова: *case, break* и *default,* принимая следующую форму:

```
switch ($переменная) {
case 1:
инструкции, выполняемые, пока TRUE;
break;
case 2:
инструкции, выполняемые, пока TRUE;
break;
case 3:
инструкции, выполняемые, пока TRUE; break;
default:
инструкции, выполняемые, пока все FALSE;
}
```

Каждое выражение *case* в инструкции *switch* сравнивает значение переменной *switch* со значением *case,* которое может быть строкой. Если значения равны, выполняются инструкции после соответствующего выражения *case,* после чего выражение *break* направляет выполнение скрипта на первую инструкцию после закрывающей фигурной скобки инструкции *switch.* Если ни одно из выражений case не было удачным, выполняются инструкции, следующие за выражением default, после чего выполнение скрипта выходит из инструкции *switch*.

**Массивы.** Одной из особенностей PHP является то, что часть массивов может использовать в качестве ссылок числовые индексы, другая позволяет работать с буквенно-цифровыми индикаторами.

Массив с числовыми индексами может быть задан следующим образом:

```
<?php
$paper[] = “Copier”;
$paper[] = “Inkjet”;
$paper[] = “Laser”;
$paper[] = “Photo”;
?>
```

В данном случае при каждом присваивании массиву $paper значения для хранения последнего используется первое же свободное место, а значение существующего в PHP внутреннего указателя увеличивается на единицу, чтобы указывать на свободное место, готовое для следующей вставки значения.

Можно, как и в более традиционных языках программирования, указывать конкретное место элементов в массиве.

```
<?php
$paper[0] = “Copier”;
$paper[1] = “Inkjet”;
$paper[2] = “Laser”;
$paper[3] = “Photo”;
?>
```

С помощью цикла можно вывести добавленные элементы на экран.

```
<?php
$paper[] = “Copier”;
$paper[] = “Inkjet”;
$paper[] = “Laser”;
$paper[] = “Photo”;
for ($j = 0; $j < 4; $j++)
        echo “$j: $paper[$j]<br>”;
?>
```

**Ассоциативные массивы.** Использование данных массивов позволяет ссылаться на элементы массива по именам, а не по номерам.

Например, рассмотренный выше массив можно создать следующим способом:

```
<?php
```

```
$paper[‘copier’] = “Copier & Multipurpose”;
$paper[‘inkjet’] = “Inkjet Printer”;
$paper[‘laser’] = “Laser Printer”;
$paper[‘photo’] = “Photographic Paper”;
echo $paper[‘laser’];
?>
```

Теперь у каждого элемента вместо числа (не содержащего никакой полезной информации, кроме позиции элемента в массиве) имеется уникальное имя, по которому на него можно сослаться где-нибудь в другом месте (например, в инструкции echo).

**Присваивание с использованием ключевого слова array.** Помимо вышеуказанных, существует более краткий и быстрый способ присваивания значений с использованием ключевого слова **array**.

Пример ниже иллюстрирует создание обоих видов массивов с помощью ключевого слова array.

```
<?php
$p1 = array(“Copier”, “Inkjet”, “Laser”, “Photo”);
echo “Элемент массива p1:” . $p1[2] . ”<br>”;
$p2 = array(    ‘copier’ => “Copier & Multipurpose”,
‘inkjet’ => “Inkjet Printer”,
‘laser’ => “Laser Printer”,
‘photo’ => “Photographic Paper”);
echo “Элемент массива p2: “ . $p2[‘inkjet’] . “<br>”;
?>
```

В первой части этого кодового фрагмента массиву $p1 присваивается укороченное описание товара. Здесь используются четыре элемента, поэтому они занимают позиции от 0 до 3. Инструкция echo выводит следующий текст:

Элемент массива p1: Laser

Во второй части кода массиву $p2 присваиваются ассоциативные идентификаторы и сопутствующие им длинные

описания товаров. Для этого используется формат index => value. Использование оператора => похоже на использование оператора присваивания =, за исключением того, что значение присваивается индексу, а не переменной. После этого индекс приобретает неразрывную связь с этим значением до тех пор, пока ему не будет присвоено другое значение. Поэтому команда echo выводит следующий текст:

Элемент массива p2: Inkjet Printer

**Цикл foreach…as.** Данный цикл специально ориентирован на работу с массивами. Используя цикл foreach можно перебрать все элементы массива и произвести с ними какие-нибудь действия.

Процесс начинается с первого элемента и заканчивается последним, поэтому не необходимости указывать пределы цикла. Например:

```
<?php
$paper = array(“Copier”, “Inkjet”, “Laser”, “Photo”);
$j = 0;
foreach ($paper as $item)
{
        echo “$j: $item<br>”;
        $j++;
}
?>
```

Когда PHP встречает инструкцию foreach, он извлекает первый элемент массива и помещает его значение в переменную, указанную после ключевого слова as, и при каждом возвращении управления инструкции foreach в эту переменную помещается значение следующего элемента массива. В данном случае переменной $item присваиваются по очереди все четыре значения, хранящиеся в массиве $paper. Как только будут использованы все значения, выполнение цикла завершается.

С ассоциативным массивом foreach работает следующим образом:

```
<?php
$paper = array(          ‘copier’ => “Copier & Multipurpose”,
‘inkjet’ => “Inkjet Printer”,
‘laser’ => “Laser Printer”,
‘photo’ => “Photographic Paper”);
foreach ($paper as $item => $description)
        echo “$item: $description<br>”;
?>
```

Ассоциативным массивам не требуются числовые индексы, поэтому каждый элемент массива $paper вводится в пару «ключ – значение», представленную переменными $item и $description, из которых эта пара выводится на экран в следующем виде:

copier: copier & Multipurpose
inkjet: Inkjet Printer
laser: Laser Printer
photo: Photographic Paper

В качестве альтернативы синтаксису foreach…as можно воспользоваться функцией list в сочетании с функцией each.

```
<?php
$paper = array(          ‘copier’ => “Copier & Multipurpose”,
‘inkjet’ => “Inkjet Printer”,
‘laser’ => “Laser Printer”,
‘photo’ => “Photographic Paper”);
while (list($item, $description) = each($paper))
        echo “$item: $description”;
?>
```

В этом примере организуется цикл while, который будет продолжать работу до тех пор, пока функция each не вернет значение FALSE. Функция each ведет себя как foreach: она возвращает из массива $paper массив, содержащий пару «ключ – значение», а затем перемещает встроенный указатель на следующую пару в исходном массиве. Когда возвращать станет нечего, функция each возвращает значение FALSE.

Функция list в качестве аргументов принимает массив (в данном случае пару «ключ – значение», возвращенную функцией each), а затем присваивает значение массива переменным, перечисленным внутри круглых скобок.

**Многомерные массивы.** Как и во многих других языках программирования в PHP можно создать многомерный массив. Многомерный массив в PHP может быть как обычным, так и ассоциативным.

Обычный многомерный массив задается следующим образом:

```
<?php
$mas = array(
        array (7, 15, 34, 12),
        array (7, 4, 10, 23),
array (78, 56, 34, 12));
echo “<pre>”;
foreach ($mas as $row)
{
        foreach ($row as $num)
                echo “$num ”;
        echo “<br>”;
}
echo “</pre>”;
?>
```

Можно вывести отдельный элемент массива на экран указав номер строки и столбца.

```
echo $mas[1][2];
```

В этом примере для последовательного перебора массива и демонстрации его содержимого используется пара вложенных циклов foreach… as. Внешний цикл обрабатывает каждую строку и помещает ее в переменную $row, которая сама по себе является массивом. Внутренний цикл обрабатывает каждый элемент в строке, выводя его на экран. За каждым элемен-

том следует пробел. Тег <pre> обеспечивает правильную форму выводимого текста.

Многомерный ассоциативный массив создается следующим образом:

```
<?php
$products = array(
   ‘paper’ => array(‘copier’ => “Copier & Multipurpose”,
‘inkjet’ => “Inkjet Printer”,
‘laser’ => “Laser Printer”,
‘photo’ => “Photographic Paper”),
‘pens’ => array(
                    ‘ball’ => “Ball Point”,
                    ‘hilite’ => “Highlighters”,
                    ‘marker’ => “Markers”));
echo “<pre>”;
foreach ($products as $section => $items)
        foreach ($items as $key => $value)
                echo “$section: \t$key\t($value)<br>”;
echo “</pre>”;
?>
```

В данном примере используется двумерный массив $products включающий в себя две строки-подмассива – paper и pens.

Для вывода на экран элементов используется цикл foreach… as. Внешний цикл извлекает из верхнего уровня массива основные разделы, а внутренний цикл извлекает для категорий в каждом разделе пары «ключ - значение».

В инструкции echo используется управляющий символ PHP \t, который выводит знак табуляции.

**Функции для работы с массивами**

**print_r()**

Данная функция выводит на экран все элементы массива в удобочитаемом виде (индекс => значение). Например:

```
<?php
```

```
$mas = array(1,2,3,4,5);
print_r($mas);
?>
```

На экран будет выведено следующее:

```
Array
(
        [0] => 1
        [1] => 2
[2] => 3
[3] => 4
[4] => 5
)
```

**is_array()**

Данная функция определяет является ли переменная массивом. Например:

```
echo (is_array($mas)) ? “Это массив” : “Это не массив”;
```

Возвращаемое переменной значение является логическим (TRUE если она является массивом, FALSE – если нет).

**count()**

Данная функция возвращает количество элементов в массиве:

```
echo count($mas);
```

Для многомерных массивов в функции рекомендуется указывать дополнительный параметр:

```
echo count($matr, 1);
```

Данный параметр имеет два значения: 1 – если требуется вернуть общее количество элементов и 0 – если требуется вернуть количество элементов верхнего уровня (строк).

**sort() и rsort()**

Сортировка массивов по возрастанию или убыванию является настолько часто применяемой процедурой, что в PHP создан ряд специальных функций для сортировки массивов.

Функция sort() сортирует элементы в массиве в порядке возрастания:

```
sort($mas, SORT_NUMERIC);
sort($mas, SORT_STRING);
```

Кроме имени массива в функции устанавливаются флаги сортировки. Наиболее распространенными вариантами являются SORT_NUMERIC и SORT_STRING – числовая и строковая сортировки, соответственно.

Массив также можно отсортировать в обратном порядке, используя функцию rsort():

```
rsort($mas, SORT_NUMERIC);
rsort($mas, SORT_STRING);
```

**shuffle()**

Данная функция перемешивает элементы массива в случайном порядке:

```
shuffle($mas);
```

**explode()**

Данная функция позволяет взять строку, содержащую несколько слов, разделенных пробелом (либо каким-то другим символом или группой символов), и поместить каждое из слов в отдельный элемент массива.

Например:

```
<?php
$temp = explode(‘ ’, “Это предложение из пяти слов”);
print_r($temp);
?>
```

На экран будет выведено следующее:

```
Array
(
        [0] => Это
        [1] => предложение
[2] => из
[3] => пяти
```

[4] => слов
)

Первый параметр – разделитель – не обязательно должен быть пробелом или даже одиночным символом.

**reset() и end()**

Данные функции позволяют управлять внутренним указателем на элементы массив, определяющим какой элемент массива в данный момент является текущим. Функция reset() перемещает указатель на первый элемент массива, функция end() – на последний элемент массива:

```
reset($mas);
end($mas);
```

### 3.3. Функции для обработки и форматирования данных

**Вывод данных**. Для вывода простого текста в браузер достаточно функций print и echo, которые были рассмотрены ранее. Но существует намного более мощная функция printf(), управляющая форматом выводимых данных путем вставки в строку специальных форматирующих символов.

Функция printf() ожидает, что для каждого форматирующего символа будет предоставлен аргумент, который будет отображаться с использованием заданного формата. Например, в следующем фрагменте используется спецификатор преобразования %d, для того, чтобы значение 3 отображалось в виде десятичного числа:

```
printf(“В Вашей корзине находится %d покупки”, 3);
```

Если заменить %d на %b, значение 3 будет отображено в виде двоичного числа (11). В табл. 14 показаны поддерживаемые функцией спецификаторы преобразования.

Таблица 14

Спецификаторы преобразования, используемые в функции printf()

| Спецификатор | Преобразование, осуществляемое с аргументом | Пример (для аргумента 123) |
|---|---|---|
| % | Отображение символа % (аргументы не требуются) | % |
| b | Отображение аргумента в виде двоичного целого числа | 1111011 |
| c | Отображение ASCII-символа с кодом, содержащимся в аргументе | { |
| D | Отображение аргумента в виде целого десятичного числа со знаком | 123 |
| e | Отображение аргумента с использованием экспоненциальной формы записи | 1.23000e+2 |
| f | Отображение аргумента в виде числа с плавающей точкой | 123.000000 |
| o | Отображение аргумента в виде восьмеричного целого числа | 173 |
| s | Отображение аргумента в виде строки | 123 |
| u | Отображение аргумента в виде беззнакового десятичного числа | 123 |
| x | Отображение аргумента в виде шестнадцатеричного числа с символами в нижнем регистре | 7b |

Продолжение табл. 14

| | | |
|---|---|---|
| X | Отображение аргумента в виде шестнадцатеричного числа в верхнем регистре | 7B |

В функции printf можно использовать любое количество спецификаторов, если им передается соответствующее количество аргументов и если каждый спецификатор предваряется символом %.

Например, можно использовать функцию printf для установки цвета в коде HTML с помощью значений интенсивности красного, зеленого и синего цвета (RGB). Чтобы не переводить эти числа в шестнадцатеричный формат добавим их в функцию printf:

printf("<font color='#%X%X%X'>Привет</font>", 65, 127, 245);

В данном случае, цвет устанавливается с помощью атрибута color тега font. В качестве значения указывается шестнадцатеричная константа, начинающаяся с символа #, за которым следуют три форматирующие спецификации %X, по одной для каждого из чисел. В результате эта команда выдаст следующий текст:

<font color='#417FF5'>Привет</font>

Обычно представляется удобным в качестве аргументов printf использовать переменные или выражения. Например, если значения для цветового решения хранятся в трех переменных – $r, $g, $b, то более темный оттенок можно получить с помощью выражения

printf("<font color='#%X%X%X'>Привет</font>", $r-20, $g-20, $b-20);

**Настройка представления данных.** Можно указать не только тип преобразования, но и точность отображаемого результата. Например, суммы в валюте отображаются, как прави-

ло, с точностью до двух цифр. Но в общем случае, может получиться более точный результат (например, при использовании операции деления).

Чтобы отобразить число с точностью до двух цифр, можно между символом % и спецификатором преобразования вставить строку «.2»:

printf(“Результат: $%.2f”, 123.42 / 12);

Результат: $10.29

Можно также указать, где и чем – нулями или пробелами дополнить выводимый текст. В примере ниже указаны пять возможных комбинаций.

```
<?php
echo “<pre>”;
//Дополнение пробелами до 15 знакомест
        printf(“Результат равен %15f\n”, 123.42 / 12);
//Дополнение нулями до 15 знакомест
        printf(“Результат равен %015f\n”, 123.42 / 12);
//Дополнение пробелами до 15 знакомест и вывод с точностью до двух
//десятичных знаков
        printf(“Результат равен %15.2f\n”, 123.42 / 12);
//Дополнение нулями до 15 знакомест и вывод с точностью до двух
//десятичных знаков
        printf(“Результат равен %015.2f\n”, 123.42 / 12);
//Дополнение символами # до 15 знакомест и вывод с точностью до
//двух десятичных знаков
        printf(“Результат равен %’#15.2f\n”, 123.42 / 12);
?>
```

Это пример выводит следующий текст:

```
Результат равен          10.285000
Результат равен 00000010.285000
```

Результат равен 10.29
Результат равен 000000000010.29
Результат равен ##########10.29

Следует обратить внимание на следующие моменты:

– самым правым символом спецификатора преобразования в данном случае является f, означающий преобразование в число с плавающей точкой;

– если сразу же перед спецификатором преобразования стоит сочетание точки и числа, значит, этим числом указана точность выводимой информации;

– независимо от присутствия спецификатора точности, если в общем спецификаторе присутствует число, то оно представляет собой количество знакомест, выделяемых под выводимую информацию (в предыдущем примере это число 15);

– если выводимая информация уже равна количеству выделяемых знакомест или превышает его, этот аргумент игнорируется;

– перед самым левым символом % разрешается поставить символ 0, который игнорируется, если не указано количество выделяемых знакомест. Если это количество указано, то вместо пробелов дополнение производится нулями;

– если нужно, чтобы пустующие знакоместа заполнялись не нулями или пробелами, а каким-нибудь другим символом, то можно выбрать любой символ, поставив перед ним одинарную кавычку: ‘#.

– в левой части спецификатора ставится символ %, с позиции которого и начинается преобразование.

**Дополнение строк.** Дополнить до требуемой длины можно не только числа, но и строки, выбирая для этого различные дополняющие символы и даже левую или правую границы выравнивания. Возможные варианты показаны в примере ниже:

<?php

```
echo “pre”;
$h = “Voronezh”;
printf(“[%s]\n”, $h);          //Стандартный вывод строки
//Выравнивание пробелами по правому //краю
printf(“[%-15s]\n”, $h);          //Выравнивание пробелами по левому
//краю
printf(“[%015s]\n”, $h);          //Дополнение нулями
printf(“[%’#15s]\n”, $h);          //Дополнение символами #
printf(“[%15.5s]\n”, $h);          //Выравнивание по правому краю
//с усечением до 5 символов
printf(“[%-15.7s]\n”, $h);     //Выравнивание по левому краю
                                   //с усечением до 7 символов
?>
```

Результат, выведенный на экран:

```
[Voronezh]
[        Voronezh]
[Voronezh        ]
[0000000Voronezh]
[#######Voronezh]
[           Voron]
[Voronez         ]
```

Если при указании количества знакомест длина строки уже равна этому количеству или превышает его, это указание будет проигнорировано, если только заданное количество символов, до которого нужно усечь строку, не будет меньше указанного количества знакомест.

**Использование функции sprint.** В некоторых случаях требуется не выводить на экран результат преобразования, а использовать его в самом коде программы. Для этого предназначена функция **sprintf**. Она позволяет не отправлять выходную информацию браузеру, а присваивать ее какой-нибудь переменной.

Например, по аналогии с одним из выше рассмотренных примеров, функцию sprintf можно использовать для преобразования, возвращающего шестнадцатеричное значение для цветового сочетания RGB 65, 127, 245, которое присваивается переменной $hexstring:

```
$hexstring = sprintf("%X%X%X", 65, 127, 245);
```

Или же она может пригодиться для сохранения выходной информации, которую нужно будет вывести на экран чуть позже:

```
$out = sprintf("Результат: $%.2f", 123.45 / 12);
echo $out;
```

**Функция даты и времени.** Для отслеживания даты и времени в PHP используются стандартные отметки времени Unix, представляющие собой простое количество секунд, прошедших с начала отсчета – 1 января 1970 года. Для определения текущей отметки времени можно воспользоваться функцией time:

```
echo time();
```

Поскольку значение хранится в секундах, для получения отметки времени ровно через неделю можно воспользоваться следующим выражением, в котором к возвращаемому значению прибавляется 7 дней * 24 часа * 60 минут * 60 секунд:

```
echo time() + 7 * 24 * 60 * 60;
```

Если нужно получить отметку времени для заданной даты, можно воспользоваться функцией mktime. Она выводит отметку времени 946684800 для первой секунды первой минуты первого часа первого дня 2000 года:

```
echo mktime(0, 0, 0, 1, 1, 2000);
```

Этой функции передаются следующие параметры (слева направо):

– количество часов (0 – 23);

– количество минут (0 – 59);

– количество секунд (0 – 59);

– номер месяца (1 – 12);

– номер дня (1 – 31);

– год (1970 – 2038, или 1901 – 2038 при использовании PHP 5.1.0 и выше).

**Отображение даты**. Для отображения даты используется функция date, поддерживающая множество настроек форматирования, позволяющих выводить дату любым желаемым способом. Эта функция использует следующий синтаксис:

date ($format, $timestamp);

Параметр $format должен быть строкой, в которой содержатся спецификаторы форматирования, подробно описанные в табл. 15, а параметр $timestamp должен быть отметкой времени в стандарте Unix.

Следующая команда выводит текущие время и дату в формате «Thursday April 15th, 2010 – 1:38pm»:

echo date("l F jS, Y – g:ia", time());

Таблица 15

Основные спецификаторы формата в функции date

| **Формат** | **Описание** | **Возвращаемое значение** |
|---|---|---|
| **Спецификаторы дня** | | |
| d | День месяца, две цифры с лидирующими нулями | от 01 до 31 |
| D | День недели, составленный из трех букв | от Mon до Sun |
| j | День месяца без лидирующих нулей | от 1 до 31 |
| l | День недели полностью | от Sunday до Saturday |

Продолжение табл. 15

| | | |
|---|---|---|
| N | День недели, число, от понедельника до воскресенья | от 1 до 7 |
| S | Суффикс для дня месяца | st, nd, rd или th |
| w | День недели, число, от воскресенья до субботы | от 0 до 6 |
| Z | День года | от 0 до 365 |
| **Спецификатор недели** | | |
| W | Номер недели в году | от 1 до 52 |
| **Спецификаторы месяца** | | |
| F | Название месяца | от January до December |
| m | Номер месяца с лидирующими нулями | от 01 до 12 |
| M | Название месяца, составленное из трех букв | от Jan до Dec |
| n | Номер месяца без лидирующих нулей | от 1 до 12 |
| t | Количество дней в заданном месяце | 28, 29, 30 или 31 |
| **Спецификаторы года** | | |
| L | Високосный год | 1 – Да, 0 – Нет |
| Y | Год, четыре цифры | от 0000 до 9999 |
| y | Год, две цифры | от 00 до 99 |
| **Спецификаторы времени** | | |
| a | До или после полудня, в нижнем регистре | am или pm |
| A | До или после полудня, в верхнем регистре | AM или PM |
| g | Час суток, 12-часовой формат без лидирующих нулей | от 1 до 12 |

Окончание табл. 15

| | | |
|---|---|---|
| G | Час суток, 24-часовой формат без лидирующих нулей | от 1 до 24 |
| h | Час суток, 12-часовой формат с лидирующими нулями | от 01 до 12 |
| H | Час суток, 24-часовой формат с лидирующими нулями | от 01 до 24 |
| i | Минуты с лидирующими нулями | от 00 до 59 |
| s | Секунды с лидирующими нулями | от 00 до 59 |

**Константы, связанные с датами.** Существует ряд полезных констант, которые можно использовать с командами, связанными с датами, для того чтобы они вернули дату в определенном формате. Например:

date(DATE_RSS)

возвращает текущие дату и время в формате, который используется в RSS-потоке.

Наиболее часто используются следующие константы:

– DATE_ATOM – формат для потоков Atom. PHP-формат имеет вид «Y-m-d\TH:i:sP», а выводимая информация – «2012-08-16T12:00:00+0000»;

– DATE_COOKIE – формат для cookie, устанавливаемый веб-сервером или JavaScript. PHP-формат имеет вид «l, d-M-y H:i:s T», а выводимая информация – «Thu, 16 Aug 2012 12:00:00 UTC»;

– DATE_RSS – формат для потоков RSS. PHP-формат имеет вид «D, d M Y H:i:s T», а выводимая информация – «Thu, 16 Aug 2012 12:00:00 UTC»;

– DATE_W3C – формат для консорциума Всемирной паутины – «World Wide Web Consortium». PHP-формат имеет вид «Y-m-d\TH:i:sP», а выводимая информация – «2012-08-16T12:00:00+0000».

**Функция checkdate().** С помощью данной функции можно проверить корректность даты состоящей из дня, месяца и года. Она возвращает два значения: TRUE, если передана допустимая дата; FALSE – если нет.

Например:

```
<?php
$month = 9;
        $day = 31;
        $year = 2012;
        if (checkdate($month, $day, $year))
echo “Допустимая дата”;
        else
                echo “Не допустимая дата”;
?>
```

В данном случае на экран будет выведена фраза «Недопустимая дата», т.к. проверяется существование даты 31 сентября 2012 года (в сентябре же, как известно, 30 дней).

### 3.4. Пользовательские функции и классы в PHP

**Пользовательские функции в PHP**

Пользовательская функция в PHP имеет следующий вид.

```
<?php
function имя($arg_1, $arg_2, …, $arg_n)
{
//тело функции
…
return значение;
}
```

?>

После имени функции идет список ее аргументов (или пустые скобки, если аргументов у функции нет). Массивы также могут передаваться в виде аргументов.

Внутри функции можно использовать любой корректный PHP-код, в том числе другие функции и даже объявления классов.

Имена функций следуют тем же правилам, что и другие метки в PHP. Корректное имя функции начинается с буквы или знака подчеркивания, за которым следует любое количество букв, цифр или знаков подчеркивания.

Все функции PHP имеют глобальную область видимости – они могут быть вызваны вне функции, даже если были определены внутри и наоборот.

PHP не поддерживает перегрузку функции, также отсутствует возможность переопределить или удалить объявленную ранее функцию.

По умолчанию аргументы в функцию передаются по значению (это означает, что если изменить значение аргумента внутри функции, то вне ее значение все равно останется прежним). Если необходимо разрешить функции модифицировать свои аргументы, следует передавать их по ссылке.

Чтобы аргумент всегда передавался по ссылке, можно указать амперсанд (&) перед именем аргумента в описании функции:

```
<?php
        $a=5;
        $b=6;
        function add(&$x, $y)
        {
                $x++;
                $y++;
        }
        echo "До функции<br>";
        echo "a=$a; b=$b<br>";
```

```
        add($a,$b);
        echo "После функции<br>";
        echo "a=$a; b=$b";
?>
```

Значения возвращаются при помощи необязательного оператора возврата. Возвращаемые значения могут быть любого типа, в том числе это могут быть массивы и объекты.

Возврат приводит к завершению выполнения функции и передаче управления обратно к той строке кода, в которой данная функция была вызвана.

Если конструкция return не указана, то функция вернет значение NULL.

Функция не может возвращать несколько значений, но аналогичного результата можно добиться, возвращая массив.

```
<?php

    function small_numbers()
    {
        return array (0, 1, 2);
    }
    list ($zero, $one, $two) = small_numbers();
?>
```

**Использование ключевого слова global**

Область видимости переменной – это контекст, в котором эта переменная определена. Для управления областью видимости глобальных переменных используется ключевое слово **global**.

```
<?php
    $a = 1;
    $b = 2;

    function Sum()
    {
        global $a, $b;
```

```
        $b = $a + $b;
    }

    Sum();
    echo $b;
?>
```

Вышеприведенный скрипт выведет значение **3**. После определения **$a** и **$b** внутри функции как **global** все ссылки на любую из этих переменных будут указывать на их глобальную версию. Не существует никаких ограничений на количество глобальных переменных, которые могут обрабатываться функцией.

Второй способ доступа к переменным глобальной области видимости - использование специального, определяемого PHP массива **$GLOBALS**. Предыдущий пример может быть переписан так:

```
<?php
    $a = 1;
    $b = 2;

    function Sum()
    {
$GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
    }

    Sum();
    echo $b;
?>
```

*$GLOBALS* – это ассоциативный массив, ключом которого является имя, а значением – содержимое глобальной переменной. *$GLOBALS* существует в любой области видимости.

**Использование статических переменных**

Другой важной особенностью области видимости переменной является *статическая* переменная. Статическая пере-

менная существует только в локальной области видимости функции, но не теряет своего значения, когда выполнение программы выходит из этой области видимости. Рассмотрим следующий пример:

```
<?php
        function test()
        {
                $a = 0;
                $a++;
                echo "a=$a<br>";
        }
        echo "Первый вызов функции test:<br>";
        test();
        echo "Второй вызов функции test:<br>";
        test();
?>
```

Локальная переменная **$a** при выходе из функции исчезает и ее значение стирается. При каждом вызове функции она устанавливается в 0, а потом увеличивается на единицу. Поэтому на экран будет выведено:

```
Первый вызов функции test:
a=1
Второй вызов функции test:
a=1
```

Чтобы написать полезную считающую функцию, которая не будет терять текущего значения счетчика, переменная **$a** объявляется как **static**:

```
<?php
        function test()
        {
                static $a = 0;
                $a++;
                echo "a=$a<br>";
        }
        echo "Первый вызов функции test:<br>";
```

```
    test();
    echo "Второй вызов функции test:<br>";
    test();
?>
```

Теперь **$a** будет проинициализирована только при первом вызове функции, а каждый вызов функции **test()** будет выводить значение **$a** и инкрементировать его:

```
Первый вызов функции test:
a=1
Второй вызов функции test:
a=2
```

**Объекты и классы в PHP**

Каждое определение класса начинается с ключевого слова **class**, затем следует имя класса, и далее пара фигурных скобок, которые заключают в себе определение свойств и методов этого класса.

Именем класса может быть любое слово, которое не входит в список зарезервированных слов PHP, начинающееся с буквы или символа подчеркивания и за которым следует любое количество букв, цифр или символов подчеркивания.

Класс может содержать собственные константы, переменные (называемые свойствами) и функции (называемые методами).

Пример определения класса:

```
<?php
    class SimpleClass
    {
        // объявление свойства
        public $var = 'default value';

        // объявление метода
        public function displayVar()
        {
            echo $this->var;
```

```
                }
}
?>
```

Псевдо-переменная **$this** доступна в том случае, если метод был вызван в контексте объекта. **$this** является ссылкой на вызываемый объект.

Для создания экземпляра класса используется директива **new**. Новый объект всегда будет создан, за исключением случаев, когда он содержит конструктор, в котором определен вызов исключения в случае ошибки. Рекомендуется определять классы до создания их экземпляров (в некоторых случаях это обязательно).

```
$instance = new SimpleClass();
```

Для доступа к свойству объекта используется синтаксис вида:

```
$объект->свойство
```

Подобным образом можно вызвать и метод:

```
$объект->метод()
```

Пример обращения к свойствам и методам объекта:

```
class User
        {
                public $name, $password;
                function greetings()
                {
                        echo "Hello, $this->name!";
                }
}
        $object = new User();
        $object->name = "Mike";
        $object->password = "mypass";
        print_r($object);
        echo "<br>";
        $object->greetings();
?>
```

Для вывода на экран сведений об объекте используется функция **print_r()**, которая выводит удобочитаемую информацию о переменной. В данном случае, на экран будет выведено следующее:

User Object ( [name] => Mike [password] => mypass )
Hello, Mike!

**Клонирование объектов**

Если объект уже создан, то в качестве параметра он передается по ссылке. Иными словами, присваивание объектов не приводит к их полному копированию.

Добавим в предыдущий пример следующий код:

```
$user1 = new User();
$user1->name = "Frank";
$user2 = $user1;
$user2->name = "Simon";
echo "user1 name: $user1->name <br>";
echo "user2 name: $user2->name <br>";
```

Он выведет на экран следующее:

user1 name: Simon
user2 name: Simon

Поскольку переменные **$user1** и **$user2** ссылались на один и тот же объект, то изменение свойства **$user2->name** привело и к изменению свойства **$user1->name**.

Во избежание подобной ситуации можно использовать инструкцию **clone**, которая создает новый экземпляр класса и копирует значения свойств из исходного класса в новый экземпляр.

Программный код в новой редакции:

```
$user1 = new User();
$user1->name = "Frank";
$user2 = clone $user1;
$user2->name = "Simon";
```

```
echo "user1 name: $user1->name <br>";
echo "user2 name: $user2->name <br>";
```

На экране появится следующее:
user1 name: Frank
user2 name: Simon

**Конструкторы и деструкторы**

PHP 5 позволяет объявлять методы-конструкторы. Классы, в которых объявлен метод-констуктор, будут вызывать этот метод при каждом создании нового объекта, так что это может оказаться полезным, например, для инициализации какого-либо состояния объекта перед его использованием.

Имя функции конструктора стандартно - __construct (т.е. к слову construct спереди добавляются два символа подчеркивания).

Например, для класса из предыдущих примеров можно создать вот такой конструктор:

```
class User
        {
function __construct($param1 = "John", $param2 = "pass")
                {
                        $this->name = $param1;
                        $this->password = $param2;
                }
                public $name, $password;
                function greetings()
                {
                        echo "Hello, $this->name!<br>";
                }
}
        $object = new User();
$user1 = new User("Tony", "i978m");
```

Следует обратить внимание, что в данном примере, кроме списка параметров, передаются их значения по умолчанию. Поэтому при создании объекта можно задавать параметры ($user1), а можно не задавать ($object).

Если необходимо, можно добавить метод-деструктор для объекта. Имя функции-деструктора - __destruct:

```
class User
        {
                …
                function __destruct()
                {
                        //Код деструктора
                }
}
```

**Область видимости методов и свойств объекта**

Область видимости свойства или метода может быть определена путем использования следующих ключевых слов в объявлении: **public, protected** или **private**. Доступ к свойствам и методам класса, объявленным как **public** (общедоступный), разрешен отовсюду. Модификатор **protected** (защищенный) разрешает доступ наследуемым и родительским классам. Модификатор **private** (закрытый) ограничивает область видимости так, что только класс, где объявлен сам элемент, имеет к нему доступ.

Методы класса должны быть определены через модификаторы **public**, **private**, или **protected**. Методы, где определение модификатора отсутствует, определяются как **public**.

Объекты одного типа имеют доступ к элементам с модификаторами **private** и **protected** друг друга, даже если не являются одним и тем же экземпляром. Это объясняется тем, что реализация видимости элементов известна внутри этих объектов.

**Константы классов**

Константы также могут быть объявлены и в пределах одного класса. Отличие переменных и констант состоит в том, что при объявлении последних или при обращении к ним не используется символ **$**.

Значение должно быть неизменяемым выражением, не (к примеру) переменной, свойством, результатом математической операции или вызовом функции.

Пример класса с объявленной константой:

```
<?php
class MyClass
{
const CONSTANT = 'значение константы';
function showConstant()
{
echo  self::CONSTANT . "\n";
}
}
echo MyClass::CONSTANT . "\n";
?>
```

В данном случае обращение к константе происходит через ключевое слово self (ссылка на класс в целом) и через оператор разрешения области видимости (::).

**Статические свойства и методы**

Объявление свойств и методов класса статическими позволяет обращаться к ним без создания экземпляра класса. Атрибут класса, объявленный статическим, не может быть доступен посредством экземпляра класса (но статический метод может быть вызван).

Так как статические методы вызываются без создания экземпляра класса, то псевдо-переменная **$this** не доступна внутри метода, объявленного статическим.

Доступ к статическим свойствам класса не может быть получен через оператор ->.

Пример класса со статическим свойством:

```
<?php
class Test
{
static $static_property = "Это статическое свойство";
                function get_sp()
                {
                        return self::$static_property;
                }
}
$temp = new Test();
echo "Test A: ".Test::$static_property."<br>";
echo "Test B: ".$temp->get_sp()."<br>";
echo "Test C:".$temp->static_property."<br>";
?>
```

На экран будет выведено следующее:

Test A: Это статическое свойство

Test B: Это статическое свойство

Test C:

Как видно из примера, свойство $static_property можно ссылаться напрямую из самого класса, используя в Test A оператор двойного двоеточия. Test B также может получить его значение путем вызова метода get_sp объекта $temp, созданного из класса Test. Но Test C терпит неудачу, потому что статическое свойство $static_property недоступно объекту $temp.

**Наследование**

Как только класс будет создан, из него можно будет получить подкласс, наследующий все публичные и защищенные методы из родительского класса. До тех пор пока не будут эти

методы переопределены, они будут сохранять свою исходную функциональность.

Наследование осуществляется путем использования инструкции **extends**.

```
<?php
        class User
        {
                public $name, $password;
                function __construct($par1, $par2)
                {
                        $this->name = $par1;
                        $this->password = $par2;
                }
                function greetings()
                {
                        echo "Hello, $this->name!<br>";
                }
        }
        class Subscriber extends User
        {
                public $phone, $email;
                function __construct($par1, $par2, $par3,
$par4)
                {
                        parent::__construct($par1, $par2);
                        $this->phone = $par3;
                        $this->email = $par4;
                }
                function display()
                {
                        echo "Name:".$this->name."<br>";
                        echo "Pass:".$this->password."<br>";
                        echo "Phone:".$this->phone."<br>";
                        echo "Email: ".$this->email."<br>";
                }
```

```
        }
        $object = new Subscriber("Mike", "THX1138",
"6785430129",
    "somemail@mail.com");
        $object->display();
    $object->greetings();
    ?>
```

У исходного класса **User** имеются два свойства - **$name** и **$password**, а также метод **greetings()** для вывода на экран приветствия пользователю. Подкласс **Subscriber** расширяет этот класс за счет добавления еще двух свойств - **$phone** и **$email** и включения метода, отображающего свойства текущего объекта. Вышеприведенный код выведет на экран следующую информацию:

```
Name: Mike
Pass: THX1138
Phone: 6785430129
Email: somemail@mail.com
Hello, Mike!
```

### 3.5. Работа с файлами. Взаимодействие с формами

**Основные операции с файлами**

Любой язык программирования высокого уровня должен включать в себя операции по работе с отдельными файлами и их содержимым. PHP не является исключением.

**Проверка существования файла**

Чтобы проверить факт существования файла, можно воспользоваться функцией **file_exists**, которая возвращает либо TRUE, либо FALSE и используется следующим образом:

```
if (file_exists("testfile.txt")) echo "Файл существует";
else echo "Файл \"testfile.txt\" не найден";
```

**Создание файла**

Для демонстрации процесса создания простого текстового файла рассмотрим следующий пример:

```
<?php
        $fh=fopen("testfile.txt", "w") or exit("Создать файл не удалось");
        $text = <<<_END
        Строка 1
        Строка 2
        Строка 3


_END;
        fwrite($fh, $text) or die("Сбой записи файла");
        fclose($fh);
        echo "Файл 'testfile.txt' записан успешно"
?>
```

Если данный код будет запущен через браузер, то при его успешном выполнении будет получено следующее сообщение: «Файл 'testfile.txt' записан успешно». В таком случае новый файл будет создан в той же папке, что и файл с программным кодом. В сам файл будет записано содержимое переменной **$text**:

```
Строка 1
Строка 2
Строка 3
```

В данном примере показана типовая последовательность работы с файлом:

а) работа начинается с открытия файла с помощью вызова функции **fopen()**;

б) после открытия файла можно вызывать другие функции (в данном случае **fwrite()**, но можно также использовать функции для чтения или других действий);

в) работа завершается закрытием файла (функция **fclose()**).

Каждому открытому файлу требуется поставить в соответствие файловую переменную, через которую PHP-программа управляет им. В данном примере это переменная **$fh**, которой присваивается значение, возвращаемое функцией **fopen()**.

В случае сбоя функция **fopen()** возвращает значение FALSE. В таком случае вызывается функция **exit()**, которая завершает программу и выдает пользователю сообщение об ошибке.

Функция fopen() использует два параметра. Первый – это имя файла, второй – режим его открытия. В данном случае используется режим “w”, предписывающий функции открыть файл для записи. Список всех возможных режимов приведен в табл. 16.

Таблица 16

Возможные режимы работы функции **fopen()**

| Режим | Описание |
|---|---|
| ‘r’ | Открывает файл только для чтения; помещает указатель в начало файла. |
| ‘r+’ | Открывает файл для чтения и записи; помещает указатель в начало файла. |
| ‘w’ | Открывает файл только для записи; помещает указатель в начало файла и обрезает файл до нулевой длины. Если файл не существует – пробует его создать. |
| ‘w+’ | Открывает файл для чтения и записи; помещает указатель в начало файла и обрезает файл до нулевой длины. Если файл не существует – пытается его создать. |
| ‘a’ | Открывает файл только для записи; помещает указатель в конец файла. Если файл не существует – пытается его создать. |
| ‘a+’ | Открывает файл для чтения и записи; помещает указатель в конец файла. Если файл не существует - пытается его создать. |

Продолжение табл. 16

| | |
|---|---|
| ‘x’ | Создаёт и открывает только для записи; помещает указатель в начало файла. Если файл уже существует, вызов **fopen()** закончится неудачей, вернёт FALSE и выдаст ошибку уровня E_WARNING (нефатальная ошибка). Если файл не существует, попытается его создать. |
| ‘x+’ | Создаёт и открывает для чтения и записи; иначе имеет то же поведение что и 'x'. |
| ‘c’ | Открывает файл только для записи. Если файл не существует, то он создается. Если же файл существует, то он не обрезается (в отличие от 'w'), и вызов к этой функции не вызывает ошибку (также как и в случае с 'x'). Указатель на файл будет установлен на начало файла. |
| ‘c+’ | Открывает файл для чтения и записи; иначе имеет то же поведение, что и 'c'. |

Также можно указывать 'b' и 't', если есть необходимость напрямую указать в каком режиме нужно работать с файлом – в бинарном или текстовом. Их можно использовать в комбинации с любыми другими режимами, например 'wb'.

**Чтение из файлов**

Для чтения строки из текстового файла можно воспользоваться функцией **fgets()**, как в примере ниже:

```
<?php
$fh=fopen("testfile.txt", "r") or exit("Не удалось открыть файл");
        $line = fgets($fh);
        fclose($fh);
        echo $line;
?>
```

Если используется файл из предыдущего примера, то из него будет получена первая строка:

Строка1

Можно попробовать вывести на экран все содержимое текстового файла построчно:

```
<?php
        $fh=fopen("testfile.txt", "r") or exit("Не удалось открыть файл");
        while (!feof($fh))
        {
                $line = fgets($fh);
                echo $line.'<br>';
        }
        fclose($fh);
?>
```

Здесь используется функция **feof()**, которая возвращает значение TRUE, если достигнут конец файла.

Можно также извлечь из файла всю информацию или нужное количество байт с помощью функции **fread()**:

```
<?php
        $fh = fopen("testfile.txt", "r") or exit("Не удалось открыть файл");
        $line = fread($fh, filesize("testfile.txt"));
        echo $line;
        fclose($fh);
?>
```

В качестве параметров функции **fread()** указываются файловая переменная и количество байтов, которое нужно считать из файла (в данном случае файл считывается целиком – для этого используется функция **filesize()**).

Если не использовать файловую переменную, то можно просто считать весь текст из файла с помощью функции **file_get_contents()**:

```
echo "<pre>";
$line = file_get_contents("testfile.txt");
echo $line;
echo "</pre>";
```

**Копирование файлов**

Можно полностью копировать содержимое одного файла в другой с помощью функции **copy()**:

```
<?php
        if (!copy("testfile.txt","testfile2.txt"))
                echo "Копирование невозможно";
        else
                echo "Файл успешно скопирован в 'testfile2.txt'";
?>
```

В случае удачного копирования файла в каталоге, где находится файл с PHP-кодом, появится текстовый файл textfile2.txt.

**Перемещение (переименование) файла**

Для перемещения файла его следует переименовать с помощью функции **rename()**:

```
<?php
        if (!rename("testfile2.txt","testfile2.new"))
                echo "Переименование невозможно";
        else
                echo "Файл успешно переименован в 'testfile2.new'";
?>
```

Функцию для переименования можно применять и к каталогам. Для того, чтобы избежать предупреждений при отсутствии исходных файлов, сначала для проверки их существования можно вызвать функцию **file_exists()**.

**Удаление файла**

Для удаления файла из файловой системы можно воспользоваться функцией **unlink()**:

```
<?php
        if (!unlink("testfile2.new"))
                echo "Удаление невозможно";
```

```
        else
                echo "Файл 'testfile2.new' успешно удален";
?>
```

**Обновление файлов**

Довольно часто возникает необходимость добавления содержимого уже в существующий файл. Для этого можно воспользоваться одним из режимов добавления данных (таблица 1) или же выбрать режим, поддерживающий запись, и просто открыть файл для чтения и записи и переместить указатель файла в нужное место, с которого необходимо вести запись или чтение.

Указатель файла – это позиция внутри файла, с которой будет осуществлен очередной доступ к файлу при чтении или записи.

Рассмотрим в качестве примера следующий код:

```
<?php
$fh = fopen("testfile.txt", "r+") or exit ("Сбой открытия файла");
        $text = "Строка 4";
        fseek($fh, 0, SEEK_END);
        fwrite($fh, "$text") or exit("Сбой записи в файл");
        fclose($fh);
        echo "Файл 'testfile.txt' успешно обновлен";
?>
```

Файл testfile.txt открывается для чтения и записи, для чего указывается режим работы 'r+', в котором указатель устанавливается в самое начало файла. В переменную **$text**, записывается содержимое добавляемой строки. После этого вызывается функция fseek(), чтобы переместить указатель файла в самый конец, куда затем добавляется строка из переменной **$text**.

Получившийся в итоге файл имеет следующий вид:

```
Строка 1
Строка 2
Строка 3
Строка 4
```

В данном примере функции fseek(), кроме описателя файла $fh, были переданы еще два параметра – 0 и SEEK_END. Параметр SEEK_END предписывает функции переместить указатель файла в его конец, а параметр 0 показывает, на сколько позиций нужно вернуться из этой позиции (В примере используется значение 0, потому что указатель должен оставаться в самом конце файла).

С функцией fseek() можно использовать еще два режима установки указателя SEEK_SET и SEEK_CUR. Режим SEEK_SET предписывает функции установку указателя файла на конкретную позицию, заданную предыдущим параметром (одна позиция – один байт). Поэтому в следующем примере указатель файла перемещается на позицию 18:

```
fseek($fh, 18, SEEK_SET);
```

Режим SEEK_CUR приводит к установке указателя файла на позицию, которая смещена от текущей позиции на заданное значение. Если в данный момент указатель файла находится на позиции 18, то следующий вызов функции переместит его на позицию 23:

```
fseek($fh, 5, SEEK_CUR);
```

**Блокирование файлов при коллективном доступе**

Веб-программы довольно часто вызываются многими пользователями в одно и то же время. Когда одновременно предпринимается попытка записи в файл более чем одним пользователем, файл может быть поврежден.

Для того чтобы обслужить сразу несколько одновременно обращающихся к файлу пользователей, нужно воспользоваться функцией блокировки файла **flock()**. Эта функция ставит в очередь все другие запросы на доступ к файлу до тех пор, пока программа не снимет блокировку. Когда программа обращается к файлу, которые не может быть доступен одновременно нескольким пользователям, с намерением произвести в него запись, к коду нужно добавлять задание на блокировку файла, как в примере ниже:

```
<?php
$fh = fopen("testfile.txt", "r+") or exit("Сбой открытия файла");
        $text = "Строка 4";
        fseek($fh, 0, SEEK_END);
        if (flock($fh, LOCK_EX))
        {
        fwrite($fh, "$text") or exit("Сбой записи в файл");
                flock($fh, LOCK_UN);
        }
        fclose($fh);
        echo "Файл 'testfile.txt' успешно обновлен";
?>
```

Данный пример является обновленной версии примера обновления файла.

При блокировке файла для посетителей веб-сайта нужно добиться наименьшего времени отклика: блокировку нужно ставить перед внесением изменений в файл и снимать ее сразу же после их внесения. Поэтому в примере функция **flock()** вызывается непосредственно до и после вызова функции **fwrite()**.

При первом вызове flock() с помощью параметра LOCK_EX устанавливается эксклюзивная блокировка того файла, ссылка на который содержится в переменной $fh:

```
flock($fh, LOCK_EX);
```

С этого момента и далее никакой другой процесс не может осуществлять не только запись, но и даже чтение файла до тех пор, пока блокировка не будет снята с помощью передачи функции параметра LOCK_UN:

```
flock($fh, LOCK_UN);
```

Как только блокировка будет снята, другие процессы снова получат возможность доступа к файлу.

Следует обратить внимание на то, что вызов с требованием эксклюзивной блокировки вложен в структуру инструкции if. Дело в том, что flock поддерживается не на всех систе-

мах, и поэтому есть смысл проверять успешность блокировки перед записью в файл.

**Загрузка файлов на веб-сервер**

Загрузка файлов на веб-сервер на деле представляет не особо сложный процесс. Для загрузки файла из формы нужно лишь выбрать специальный тип кодировки который называется multipart/form-data, а все остальное будет сделано веб-браузером пользователя.

Рассмотрим следующий пример:

```
<?php
        echo <<<_END
<html>
        <head>
<title>PHP-форма для загрузки файлов на сервер</title>
        </head>
        <body>
                <form method='post' action='index.php'
enctype='multipart/form-data'>
        Выберите файл: <input type='file' name='filename'>
        <input type='submit' value='Загрузить'>
                </form>
        </body>
_END;
                if ($_FILES)
                {
                        $name= $_FILES['filename']['name'];
move_uploaded_file($_FILES['filename']['tmp_name'],$name);
                        echo "Загружаемое изображение
'$name'<br>
<img src='$name'";
                }
                echo "</body></html>";
?>
```

Проанализируем программу по блокам. В первой строке многострочной инструкции **echo** задается начало HTML-документа, отображается заголовок, а затем начинается тело документа.

Далее идет форма, для передачи содержимого которой выбран метод POST, задается предназначение всех отправляемых данных программе index.php (т.е. самому исходному файлу) и указывается веб-браузеру на то, что отправляемые данные должны быть закодированы с использованием типа содержимого multipart/form-data.

Для подготовки формы в следующих строках задается отображение приглашения «Выберите файл», а затем дважды запрашивается пользовательский ввод. Сначала от пользователя требуется указать файл через поле ввода input со значением атрибута type = “file”.

Затем от пользователя требуется ввести команду на отправку данных формы, для чего служить кнопка с надписью «Загрузить». После этого форма закрывается.

PHP-код, предназначенный для приема загружаемых данных, сравнительно прост, поскольку все загружаемые на сервер файлы помещаются в ассоциативный системный массив $_FILES. Поэтому для установки факта отправки пользователем файла достаточно проверить, есть ли у массива $_FILES хоть какое-нибудь содержимое. Эта проверка осуществляется с помощью инструкции if ($_FILES).

При первом посещении страницы пользователем, которое происходит еще до загрузки файла, массив $_FILES пуст, поэтому программа пропускает этот блок кода. Когда пользователь загружает файл, программа запускается еще раз и обнаруживает присутствие элемента в массиве $_FILES.

Когда программа обнаружит, что файл был загружен, его имя, каким оно было прочитано из компьютера, занимавшегося загрузкой, извлекается и помещается в переменную **$name**. Теперь нужно только переместить файл из временного места, где PHP хранит загруженные файлы, в постоянное место

хранения. Это делается с помощью функции move_uploaded_file(), которой передается исходное имя файла, сохраняемого в текущем каталоге.

В конце программы загруженное на сервер изображение отображается путем помещения его имени в тег <img>.

**Использование массива $_FILES**

При загрузке файла на сервер в массиве $_FILES сохраняются пять элементов, показанных в табл. 17.

Таблица 17

Содержимое массива $_FILES

| **Элемент массива** | **Содержимое** |
|---|---|
| $_FILES[‘file’][‘name’] | Имя загруженного файла |
| $_FILES[‘file’][‘type’] | MIME-тип содержимого файла |
| $_FILES[‘file’][‘size’] | Размер файла в байтах |
| $_FILES[‘file’][‘tmp_name’] | Имя временного файла, сохраненного на сервере |
| $_FILES[‘file’][‘error’] | Код ошибки, получаемый после загрузки файла |

Типы содержимого обычно называют MIME-типами (Multipurpose Internet Mail Extension – многоцелевые почтовые расширения в Интернете), но поскольку позже они были распространены на все виды передаваемой через Интернет информации, то их часто называют типами информации, используемой в Интернете (Internet media types). Можно привести ряд известных MIME-типов:

application/pdf
application/zip
audio/mpeg
image/gif
image/jpeg
multipart/form-data
text/css

text/html
text/javascript
video/mp4

**Проверка допустимости**

При загрузке данных через форму важно проверять был ли файл получен, и если он получен, то был ли отправлен правильный тип данных.

Модифицируем предыдущий пример:

```
<?php
        echo <<<_END
<html>
        <head>
<title>PHP-форма для загрузки файлов на сервер</title>
        </head>
        <body>
                <form method='post' action='index.php'
 enctype='multipart/form-data'>
        Выберите файл с расширением JPG, GIF, PNG
или TIF: <input type='file' name='filename'>
                <input type='submit' value='Загрузить'>
                </form>
        </body>
_END;
                if ($_FILES)
                {
                        $name= $_FILES['filename']['name'];
                        switch($_FILES['filename']['type'])
                        {
                case "image/jpeg":      $ext = 'jpg'; break;
                case "image/gif":       $ext = 'gif'; break;
                case "image/png":       $ext = 'png'; break;
                case "image/tiff":      $ext = 'tif'; break;
                                default:                $ext = '';
                        }
```

```
                    if ($ext)
                    {
                    $n = "image.$ext";
move_uploaded_file($_FILES['filename']['tmp_name'],     $n);
echo "Загружаемое изображение '$name' под именем '$n'<br>";
                    echo "<img src='$n'>";
                    }
                }
            else echo "Загрузки изображения не произошло";
                echo "</body></html>";
    ?>
```

В данном примере добавлен блок switch/case, в котором проверяется соответствие типа файла четырем типам изображений. При обнаружении соответствия переменной **$ext** присваивается трехсимвольное расширение имени файла, относящееся к этому типу. Если соответствие не обнаруживается, то переменной **$ext** будет присвоена пустая строка.

В следующем блоке проверяется, содержит ли переменная $ext строку, и в случае положительного ответа в переменной $n создается новое имя файла, составленное из основы image и расширения, сохраненного в переменной $ext. Это означает, что программа полностью контролирует имя создаваемого файла и этим именем может быть только одно из следующих: image.jpg, image.gif, image.png или image.gif.

Остальной код PHP по сути повторяет код предыдущего примера. Он перемещает загруженное временное изображение на его новое место, а затем выводит его на экран, а вместе с ним отображает старое и новое имена изображения.

Если загружен неподдерживаемый тип изображения, программа выводит сообщение об ошибке.

**Работа с формами**

Обработка форм является многоступенчатым процессом. Сначала создается форма, в которую пользователь может вво-

дить необходимые данные. Затем эти данные отправляются веб-серверу, где происходит их разбор, зачастую совмещаемый с проверкой на отсутствие ошибок. Если код PHP найдет одно или несколько полей, требующих повторного ввода, форма может быть заново отображена вместе с сообщением об ошибке.

Для создания формы потребуются как минимум следующие элементы:

– открывающий и закрывающий теги – соответственно <form> и </form>;

– тип передачи данных, задаваемый одним из методов – GET или POST;

– одно или несколько полей для ввода данных;

– URL-адрес назначения, по которому будут отправлены данные формы.

Извлечение отправленных данных из формы

Рассмотрим пример программы, которая выводит на экран имя, введенное пользователем:

```
<?php
    if (isset($_POST['name'])) $name = $_POST['name'];
    else $name = "(Не введено)";
    echo <<<_END
    <html>
        <head>
            <title>Form Test</title>
        </head>
        <body>
        Вас зовут: $name<br>
        <form method="post" action="index.php">
            Как Вас зовут?
            <input type="text" name="name">
            <input type="submit">
        </form>
        </body>
    </html>
```

_END;
?>

Особое внимание следует обратить на начало программы, где в операторе **if** проверяется, содержит ли ассоциативный массив $_POST отправленное поле **name**.

Ассоциативный массив $_POST содержит все данные переданные из формы программе с помощью метода POST. В качестве индексов массива выступают имена переменных, в качестве значений элементов с данным индексом – значения переменных. Для передачи данных методом GET используется аналогичный массив $_GET.

Функция **isset()** проверяет, установлена ли переменная каким-либо значением отличным от NULL. Если пользователь не указал свое имя в текстовом поле и просто нажал кнопку «Отправить запрос», то программа присвоит переменной **$name** значение «(Не введено)». А если значение было отправлено, то оно сохраняется в этой переменной. После тега <body> была введена еще одна строка, предназначенная для отображения значения, сохраненного в переменной **$name**.

**Использование одного и того же имени переменной для передачи нескольких значений**

В некоторых случаях, пользователю требуется передать группу значений объединенных по какому-либо принципу: список опций, наличие определенных степеней образования. Для этого часто используются независимые переключатели или поля-флажки (checkbox).

Например, требуется указать сведения об образовании:

Среднее <input type="checkbox" name="ed"
value="Среднее">
Среднее специальное <input type="checkbox" name="ed"
value="Среднее специальное">

Высшее <input type="checkbox" name="ed"
value="Высшее">

В данном случае в качестве имени указывается одиночная переменная, а не массив. В таком варианте программе будет отправлен только последний отмеченный элемент формы. Поэтому после имени переменной следует добавить квадратные скобки, чтобы она стала массивом в котором отмечены все выбранные значения.

Среднее <input type="checkbox" name="ed[]"
value="Среднее">
Среднее специальное <input type="checkbox" name="ed[]"
value="Среднее специальное">
Высшее <input type="checkbox" name="ed[]"
value="Высшее">

Теперь, если при отправке формы установлены какие-либо значения из этих флажков, будет отправлен массив по имени ed, содержащий любые значения. В любом случае можно извлечь в переменную либо отдельное значение, либо массив значений:

$ed = $_POST[‘ed’];

В таблице 18 приведены все возможные наборы значений для массива $ed[].

Таблица 18

Возможные варианты значений для массива $ed[]

| При отправке одного значения | При отправке двух значений | При отправке трех значений |
|---|---|---|
| $ed[0]= "Среднее" | $ed[0]= "Среднее"<br>$ed[1]= "Среднее специальное" | $ed[0]= "Среднее"<br>$ed[1]= "Среднее специальное"<br>$ed[2]= "Высшее" |

Продолжение табл. 18

| | | |
|---|---|---|
| $ed[0]= "Среднее специальное" | $ed[0]= "Среднее"<br>$ed[1]= "Высшее" | |
| $ed[0]= "Высшее" | $ed[0]= "Среднее специальное"<br>$ed[1]= "Высшее" | |

Если переменная $ed является массивом, то для отображения ее содержимого можно использовать очень простой PHP-код:

```
foreach($ed as $item)
        echo "$item<br>";
```

**Расширенный пример программы для работы с формой**

Для более подробной иллюстрации объединения PHP-программы с HTML-формой рассмотрим следующий программный код, осуществляющий конвертацию введенного значения температуры по шкале Цельсия в шкалу Фаренгейта и наоборот.

```
<?php

        $f = $c = "";
if (isset($_POST['f'])) $f = sanitizeString($_POST['f']);
if (isset($_POST['c'])) $c = sanitizeString($_POST['c']);
        if ($f !='')
        {
                $c = intval((5 / 9) * ($f - 32));
                $out = "$f F равно $c C";
        }
        elseif($c !='')
        {
                $f = intval((9 / 5) * $c + 32);
                $out = "$c C равно $f F";
        }
```

```
else $out = "";

echo <<<_END
<html>
    <head>
<title>Программа перевода температуры</title>
    </head>
    <body>
        <pre>
```

Введите температуру по Фаренгейту или по Цельсию и щелкните по кнопке Перевести

```
    <b>$out</b>
    <form method="post" action="index.php">
    По Фаренгейту
<input type="text" name="f">
    По Цельсию
<input type="text" name="c">
    <input type="submit" value="Перевести">
            </form>
        </pre>
    </body>
</html>
_END;
    function sanitizeString($var)
    {
        $var = stripslashes($var);
        $var = htmlentities($var);
        $var = strip_tags($var);
        return $var;
    }
?>
```

Когда файл index.php будет открыт в браузере, результат будет сходным с рис. 9.

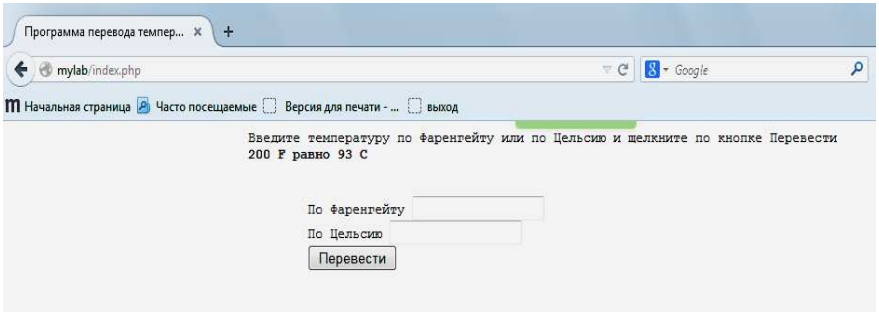


Рис. 9. Результат работы программы

Если проанализировать данную программу, то в первой строке анализируются переменные **$c** и **$f** на тот случай, если их значения не были отправлены программе. В следующих двух строках извлекаются значения либо из поля f, либо из поля c. Эти поля предназначены для ввода значений температуры по Фаренгейту или по Цельсию. Если пользователь введет оба значения, то значение по Цельсию будет проигнорировано, а переведено будет значение по Фаренгейту. В качестве меры безопасности в программе также используется новая пользовательская функция **sanitizeString()**, которая будет рассмотрена позже.

В следующей части кода (блок if…elseif…) проверяется, какую именно температуру ввел пользователь – по шкале Фаренгейта или по шкале Цельсия – и осуществляется соответствующее преобразование. Если пользователь не ввел данные, то переменной **$out** присваивается пустая строка.

Для предотвращения результатов перевода в вещественное число в обоих переводах вызывается PHP-функция intval(), представляющая результат в виде целого числа.

Теперь после выполнения всех арифметических вычислений, программа выдает HTML-код, который начинается с базовых элементов <head> и <title> и содержит вводный текст, предшествующий отображению значения переменной $out. Ес-

ли перевода температуры не осуществлялось, переменная $out будет иметь значение NULL, и выводиться на экран не будет.

Затем следует форма, настроенная на отправку данных файлу index.php (то есть самой программе) с использованием метода POST. Внутри формы содержатся два поля для ввода температуры как по Фаренгейту, так и по Цельсию. Затем отображается кнопка отправки данных имеющая надпись «Перевести», и форма закрывается.

После вывода HTML-кода, закрывающего документ, программа завершается функцией **sanitizeString()**, которая обезвреживает вводимые данные с точки зрения безопасности.

Функция **stripslashes()** удаляет экранирование символов с помощью слэш-символов (\).

Функция **htmlentities()** используется для удаления из строки нежелательного HTML-кода. Например, если пользователь введет hi, то данный код заменится строкой <b>hi</b>, которая отображается как простой текст и не будет интерпретироваться как теги HTML.

Функция **strip_tags()** удаляет все HTML-теги из строки.

Все эти функции сосредоточены в специально созданной функции **sanitizeString()**, которой в качестве параметра передаются вводимые пользователем значения. Каждое значение последовательно передается всем этим функциям для очистки, после чего возвращается как результат работы **sanitizeString()**.

## 3.6. Введение в MySQL

MySQL является одной из самых популярных СУБД в мире. Разработанная в середине 1990-х годов, она превратилась в полноценную технологию, входящую в состав многих современных наиболее посещаемых Интернет-ресурсов.

Как и PHP, MySQL является продуктом свободного пользования. Также MySQL является очень мощной и исключительно быстрой системой, способной работать даже на са-

мом скромном оборудовании, не отнимая слишком много системных ресурсов.

В названии MySQL составляющая SQL означает Structured Query Language – язык структурированных запросов. Если характеризовать его в общих чертах, то это язык, основанный на словах английского языка и используемый также в других системах управления базами данных, например Oracle и Microsoft SQL Server. Он разработан для предоставления возможности создания простых запросов к базе данных посредством команд следующего вида:

SELECT title FROM publications WHERE author = 'Stephen King';

В базе данных MySQL содержится одна или несколько *таблиц*, каждая из которых состоит из *записей* или *строк*. Внутри строк находятся разные *столбцы* или *поля*, в которых содержатся данные.

**Доступ к MySQL из командной строки**

Работать с MySQL можно тремя основными способами:

– используя командную строку;

– используя веб-интерфейс (например, утилиту из состава пакета Denwer phpMyAdmin);

– используя язык программирования (например, PHP).

Рассмотрим первый вариант.

Если MySQL установлена в составе пакета Denwer, то можно запустить исполняемый файл **mysql.exe** в каталоге WebServers\usr\local\mysql-5.5\bin (где WebServers – каталог, куда был установлен Denwer) со следующими ключами:

Z:\WebServers\usr\local\mysql-5.5\bin\mysql –u root

Эта команда предписывает MySQL зарегистрировать пользователя под именем **root** без пароля. После ее выполнения на экране появится окно командной строки.

Для того, чтобы убедиться, что все работает должным образом можно ввести следующую команду:

SHOW databases;

По данной команде будет выдан список баз данных доступный текущему пользователю. Если еще ни одной базы не было создано, на экране появится лишь список баз созданных по умолчанию – **information_schema** и т.д.

**Синтаксис команд MySQL**

Как видно из предыдущего примера, в конце команды MySQL ставится точка с запятой. Она используется для окончания текущей команды и, соответственно, отделения команд друг от друга. Если забыть поставить данный символ, то MySQL выдаст приглашение и будет ожидать его ввода.

Таким образом, можно вводить длинные команды, разбивая их на несколько строк. Можно также вводить короткие команды в одной и той же строке, разделяя их точкой с запятой. После нажатия клавиши Enter интерпретатор получит все эти команды в едином пакете и выполнит их в порядке следования.

На экране могут появляться шесть разных приглашений MySQL, позволяющих определить, на каком именно этапе многострочного ввода находится пользователь (табл. 19).

Таблица 19

Приглашения к вводу команды MySQL

| **Приглашение MySQL** | **Значение** |
|---|---|
| mysql> | MySQL готова к работе и ждет ввода команды |
| -> | Ожидание следующей строки команды |
| ‘> | Ожидание следующей строки строкового значения, которое начиналось с одинарной кавычки |
| “> | Ожидание следующей строки строкового значения, которое начиналось с двойной кавычки |

Продолжение табл. 19

| | |
|---|---|
| `> | Ожидание следующей строки строкового значения, которое начиналось с символа засечки (`) |
| /*> | Ожидание следующей строки комментария, который начинался с символов /* |

**Отмена команды**

Если пользователь, набрав часть команды, решил, что ее вообще не следует выполнять, то можно ввести комбинацию символов **\c** и нажать Enter. Тогда содержимое строки будет проигнорировано и на экран выведется приглашение на ввод новой команды:

```
mysql>dfsdfdsfd\c
mysql>
```

**Команды MySQL**

В табл. 20 приведен список наиболее востребованных команд MySQL.

Таблица 20

Основные команды MySQL

| Команда | Параметры | Назначение |
|---|---|---|
| ALTER | База данных, таблица | Внесение изменений в базу данных или таблицу |
| BACKUP | Таблица | Создание резервной копии таблицы |
| \c | | Отмена ввода |
| CREATE | База данных, таблица | Создание базы данных или таблицы |
| DELETE | (Выражение с участием таблицы и строки) | Удаление строки из таблицы |
| DESCRIBE | Таблица | Описание столбцов таблицы |

Продолжение табл. 20

| | | |
|---|---|---|
| DROP | База данных, таблица | Удаление базы данных или таблицы |
| EXIT (Ctrl+C) | | Выход |
| GRANT | (*Пользователь* подробности) | Изменение привилегий пользователя |
| HELP (\h \?) | Элемент | Отображение подсказки по элементу |
| INSERT | (Выражение с данными) | Вставка данных |
| LOCK | Таблица (таблицы) | Блокировка таблицы (таблиц) |
| QUIT (\q) | | То же самое, что и EXIT |
| RENAME | Таблица | Переименование таблицы |
| SHOW | (Множество элементов для вывода в виде списка) | Список сведений об элементах |
| SOURCE | Имя файла | Выполнение команд из указанного файла |
| STATUS (\s) | | Отображение текущего состояния |
| TRUNCATE | Таблица | Опустошение таблицы |
| UNLOCK | Таблица (таблицы) | Снятие блокировки таблицы |
| UPDATE | (Выражение с данными) | Обновление существующей записи |
| USE | База данных | Использование указанной базы данных |

Многие из этих команд будут рассмотрены далее, но сначала следует запомнить два важных положения, касающихся команд MySQL.

– команды и ключевые слова MySQL нечувствительны к регистру. Все три команды – CREATE, create и CrEaTe – абсолютно равнозначны. Но для написания более читабельного кода для команд рекомендуется использовать буквы верхнего регистра.

– имена таблиц чувствительны к регистру в Linux и Mac OS X, но нечувствительны в Windows. Поэтому из соображений переносимости нужно всегда выбирать буквы одного из регистров и пользоваться только ими. Для имен таблиц рекомендуется использовать буквы нижнего регистра.

**Создание базы данных**

Для создания новой базы данных следует воспользоваться командой CREATE. Например, создадим базу данных с именем publications:

CREATE DATABASE publications;

При успешном выполнении команды будет выведено сообщение вида «Query OK, 1 row affected (0.02 sec)».

После создания базы данных с ней можно работать. Для этого следует ввести следующую команду:

USE publications;

Теперь должно быть выведено сообщение об изменении текущей базы данных – Database changed и с ней можно проводить базовые операции – создание таблиц и т.д.

**Организация доступа пользователей**

Для создания пользователя и наделения его правами доступа к базе можно воспользоваться командами CREATE и GRANT, например:

```
CREATE USER 'mike'@'localhost' IDENTIFIED BY ‘mypass’;
GRANT ALL ON publications.* TO ‘mike’@’localhost’;
```

Кроме имени пользователя указывается еще имя хоста, например 'mike'@'localhost'.

Данная комбинация команд создает пользователя mike с паролем 'mypass' и предоставляет ему полный доступ к базе данных publications.

Можно объединить две команды в одну:

GRANT ALL ON publications.* TO 'mike'@'localhost' IDENTIFIED BY 'mypass';

Результат данной операции можно проверить, если выйти из MySQL, и запустить его из командной строки заново указав новые ключи:

Z:\WebServers\usr\local\mysql-5.5\bin\mysql –u mike –p

На экране появится приглашение для ввода пароля (можно сразу указать пароль после ключа –p в командной строке, но это некорректно с точки зрения безопасности). После указания пароля доступ к MySQL будет открыт.

**Создание таблицы**

Для создания таблицы следует воспользоваться командой CREATE (при условии, что пользователь указал нужную базу данных с помощью команды USE).

Например, создадим таблицу books для базы данных publications:

```
CREATE TABLE books (
author VARCHAR(128),
title VARCHAR(128),
type VARCHAR(16),
year CHAR(4)) ENGINE MyISAM;
```

Данная команда создает новую таблицу состоящую из 4 полей: ФИО автора (author), название книги (title), тип или жанр книги (type), год издания (year). Также указаны типы и размер создаваемых данных.

В конце команды стоит директива ENGINE MyISAM, которая указывает на тип механизма управления базой данных (например, MyISAM или InnoDB).

Для проверки корректности создания таблицы можно проверить ее существование, выведя на экран ее структуру с помощью команды DESCRIBE:

DESCRIBE books;

На экране появится описание полей таблицы базы данных (рис. 10). Для каждого поля таблицы указана следующая информация:

Field – имя каждого из полей или столбцов таблицы;

Type – тип данных, сохраняемых в поле;

Null – показывает, может ли поле содержать значение NULL;

Key – тип применяемого к полю ключа (если таковой имеется);

Default – исходное значение, присваиваемое полю, если при создании новой строки не указано никакого значения;

Extra – дополнительная информация, например, о настройке поля на автоматическое приращение его значения.

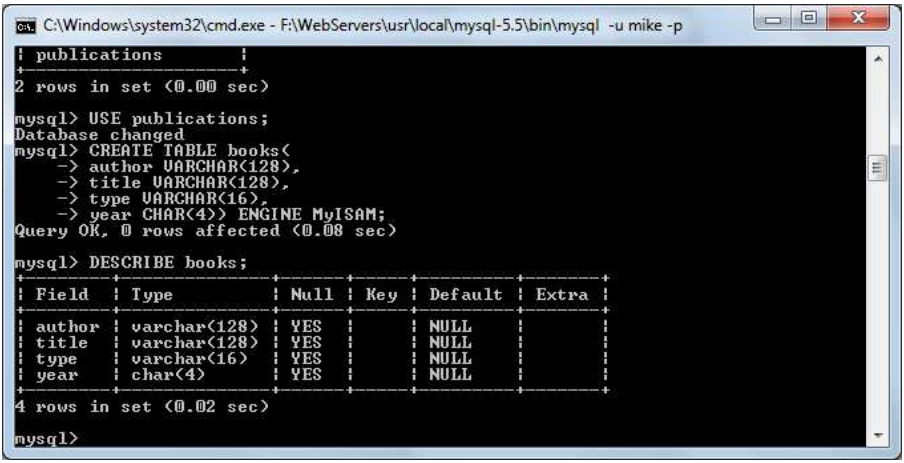


Рис. 10. Пример результата работы команды DESCRIBE

**Типы данных MySQL**

В табл. 21 перечислены основные типы данных MySQL.

Таблица 21

Основные типы данных

| Тип данных | Количество байт (формат) | Особенности |
|---|---|---|
| CHAR(n) | В точности равное n (<=255) | Требуется указать максимальную или точную длину строки |
| VARCHAR(n) | Вплоть до n (<=65535) | Требуется указать максимальную или точную длину строки |
| BINARY(n) или BYTE(n) | В точности равное n (<=255) | Оба типа данных используются для хранения строк, заполненных байтами, не имеющих никакой связи с таблицей символов |
| VARBINARY(n) | Вплоть до n (<=65535) | |
| TINYTEXT(n) | Вплоть до n (<=255) | Все типы считаются строкой с набором символов |
| TEXT(n) | Вплоть до n (<=65535) | |
| MEDIUMTEXT(n) | Вплоть до n (<=16 777 215) | |
| LONGTEXT(n) | Вплоть до n (<=4 294 967 295) | |
| TINYBLOB(n) | Вплоть до n(<=255) | Все типы считаются не набором символов, а двоичными данными. Термин BLOB означает Binary Large Object – большой двоичный объект. |
| BLOB(n) | Вплоть до n(<=65535) | |
| MEDIUMBLOB(n) | Вплоть до n (<=16 777 215) | |
| LONGBLOB(n) | Вплоть до n(<= 4 294 967 295) | |

Продолжение табл. 21

| | | |
|---|---|---|
| TINYINT | 1 (-128..127 или 0..255) | Целые числовые типы данных. Для указания знакового или беззнакового типа используется спецификатор UNSIGNED, например:<br><br>CREATE TABLE tablename (fieldname INT UNSIGNED); |
| SMALLINT | 2 (-32768..32767 или 0..65535) | |
| MEDIUMINT | 3 (-8388608..8388607) | |
| INT или INTEGER | 4 (-2 147 483 648..2 147 483 647 или 0..4 294 967 295) | |
| BIGINT | 8 (-9 223 372 036 854 775 808.. 9 223 372 036 854 775 807 или 0..184 467 440 737 09 551 615 | Целый числовой тип данных<br>Можно добавить число, показывающее отображаемую ширину данных в поле:<br>CREATE TABLE tablename (fieldname INT(4));<br>Если указать спецификатор ZEROFILL, то недостающие позиции будут заполнены нулями:<br>CREATE TABLE tablename (fieldname INT(4) ZEROFILL); |

Окончание табл. 21

| | | |
|---|---|---|
| FLOAT | 4 (-3,402823466E+38 .. 3402823466E+38) | Вещественные типы данных. Беззнаковыми не бывают |
| DOUBLE или REAL | 8 (-1,797 693 134 862 3157E+308.. 1,797 693 134 862 3157E+308) | |
| DATETIME | ‘YYYY-MM-DD HH:MM:SS’ | Типы данных даты и времени. TIMESTAMP и DATETIME отображаются одинаково, но у TIMESTAMP уже диапазон лет (1970 – 2037). TIMESTAMP удобен тем, что получает по умолчанию значение текущей даты и времени. |
| DATE | ‘YYYY-MM-DD’ | |
| TIMESTAMP | ‘YYYY-MM-DD HH:MM:SS’ | |
| TIME | ‘HH:MM:SS’ | |
| YEAR | YYYY | |

**Тип данных AUTO_INCREMENT**

Для обеспечения уникальности записи в таблице базы данных требуется ключевое поле (или комбинация полей). В случае, если значениях все полей могут повторяться, можно создать отдельное поле – счетчик записей. Для его реализации можно воспользоваться типом данных AUTO_INCREMENT.

В соответствии с названием поля, которому назначен этот тип данных, его содержимому будет устанавливаться значение, на единицу большее, чем значение такого же поля в предыдущей записи таблицы.

Добавим поле id в таблицу books, которая рассматривалась в предыдущем примере:

ALTER TABLE books ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT KEY;

Команда ALTER работает уже с существующей таблицей и может добавлять, изменять или удалять столбцы. В данном примере добавляется столбец по имени id, имеющий следующие характеристики:

INT UNSIGNED – делает столбец способным принять большое число, достаточное для того, чтобы в таблице могло храниться более 4 миллиардов записей;

NOT NULL – обеспечивает наличие значения в каждой записи столбца (появление пустого значения в данном поле недопустимо);

AUTO_INCREMENT – задает тип данных – счетчик;

KEY – поле-счетчик с автоприращением значения будет использоваться в качестве ключа.

Теперь каждая запись будет иметь уникальное число в поле id, для первой записи это будет число 1, а счет других записей будет вестись по нарастающей.

Для того, чтобы удалить поле id (для рассмотрения следующих операций с таблицами базы MySQL оно временно не нужно) можно воспользоваться командой DROP:

ALTER TABLE books DROP id;

**Операции с таблицами базы данных**

Для добавления данных к таблице используется команда INSERT. Рассмотрим ее на конкретном примере – добавление нескольких записей в таблицу books:

INSERT INTO books(author, title, type, year)

VALUES ('Mark Twain', 'The Adventures of Tom Sawyer','Novel',2005);

INSERT INTO books(author, title, type, year)

VALUES ('Emily Bronte', 'Wuthering Heights', 'Novel',2010);

INSERT INTO books(author, title, type, year)

VALUES ('Stephen King', 'The Shining','Horror novel', 2011);

Для проверки корректности ввода содержимого можно воспользоваться запросом, отображающим содержимое всей таблицы:

SELECT * FROM books;

Результат работы команды представлен на рис. 11.

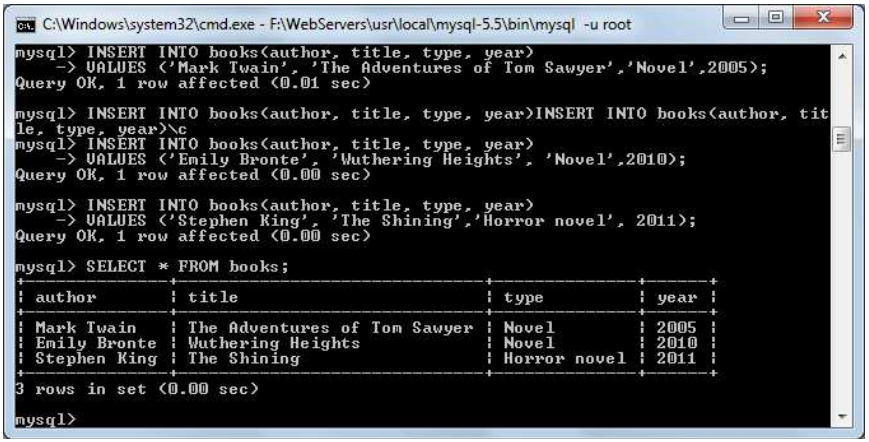


Рис. 11. Просмотр содержимого таблицы books

Теперь рассмотрим составные части команды INSERT. Ее первая часть, INSERT INTO books сообщает MySQL, куда нужно вставлять следующие за ней данные. Затем в круглых скобках перечисляются четыре имени полей: author, title, type и year, - которые отделяются друг от друга запятыми. Таким образом MySQL сообщается, что именно в эти четыре поля будут вставляться данные.

Во второй строке команды INSERT содержится ключевое слово VALUES, за которым следуют четыре строковых значения, взятых в кавычки и отделенных друг от друга запятыми. Они обеспечивают MySQL теми четырьмя значениями, которые будут вставлены в четыре ранее указанных столбца.

Каждый элемент данных будет вставлен по порядку в соответствующие столбцы.

**Переименование таблиц**

Переименование таблиц, как и любые другие изменения ее структуры или метаданных можно осуществить посредством команды ALTER. Поэтому, для того, чтобы изменить имя таблицы books на classics, используется следующая команда:

```
ALTER TABLE books RENAME classics;
```

Если требуется вернуть таблице прежнее имя, то можно ввести следующую команду:

```
ALTER TABLE classics RENAME books;
```

**Изменение типа данных поля**

Для изменения типа данных поля также используется команда ALTER, но в этом случае вместе с ней используется ключевое слово MODIFY. Поэтому для изменения данных поля с CHAR(4) на SMALLINT (для которого потребуется только 2 байта памяти, что способствует экономии дискового пространства), нужно ввести следующую команду:

```
ALTER TABLE books MODIFY year SMALLINT;
```

После этого, если для MySQL есть смысл конвертировать тип данных, система автоматически изменит данные, сохраняя их значение.

**Добавление и удаление поля**

Для добавления нового поля в таблицу уже после ее создания, можно воспользоваться командой ALTER с использованием ключевого слова ADD. Например, добавим к таблице books новое поле pages, который будет использоваться для хранения количества страниц, имеющихся в книге:

```
ALTER TABLE books ADD pages SMALLINT UNSIGNED;
```

Тип данных поля pages позволяет разместить в нем целые значения от 0 до 65535.

Для удаления поля можно воспользоваться командой ALTER с ключевым словом DROP:

```
ALTER TABLE books DROP pages;
```

**Переименование поля**

Для переименования поля можно воспользоваться командой ALTER с ключевым словом CHANGE. Например переименуем поле type в таблице books поле genre:

```
ALTER TABLE books CHANGE type genre VARCHAR(16);
```

Следует обратить внимание на добавление VARCHAR(16) в конце этой команды. Это связано с тем, что ключевое слово CHANGE требует указания типа данных даже в том случае, когда в этом нет необходимости.

**Удаление таблицы**

Удалить таблицу достаточно просто – для этого следует воспользоваться командой DROP. Для иллюстрации работы данной команды можно добавить в базу данных таблицу disposable, а затем сразу же ее удалить:

```
CREATE TABLE disposable(trash INT);
DESCRIBE disposable;
DROP TABLE disposable;
SHOW tables;
```

Следует заметить, что командой DROP надо пользоваться с большой осторожностью – ее действие носит необратимый характер и по недоразумению данной командой можно удалить целую базу.

**Индексы. Создание индекса**

Для организации быстрого поиска можно добавить в таблицу индекс – либо при создании таблицы, либо впоследствии. Индексы следует добавить к тем поля, по которым пользователи предположительно будут чаще осуществлять поиск.

Для таблицы books есть смысл добавить индекс ко всем полям с помощью следующего кода:

```
ALTER TABLE books ADD INDEX(author(20));
ALTER TABLE books ADD INDEX(title(20));
ALTER TABLE books ADD INDEX(genre(4));
ALTER TABLE books ADD INDEX(year);
```

Первые две команды создают индексы для полей авторов и названий – author и title, ограничивая каждый индекс только первыми двадцатью символами. Например, когда MySQL индексирует название «The Adventures of Tom Sawyer» на самом деле в индексе будут сохранены только первые 20 символов «The Adventures of To». Это делается для сокращения размера индекса и для оптимизации скорости доступа к базе данных (обычно первых 20 символов хватает для обеспечения уникальности названия книги).

Для поля genre выбран индекс длиной 4 символа. Для поля year ограничение на индекс не задано, т.к. оно является целым числом, а не строкой.

Результат ввода этих команд проиллюстрирован на рис. 12. Для каждого поля указано значение ключа MUL. Это означает, что в данном поле может повторяться одно и то же значение, что в принципе логично (имена авторов, книг, жанр и годы издания могут повторяться).

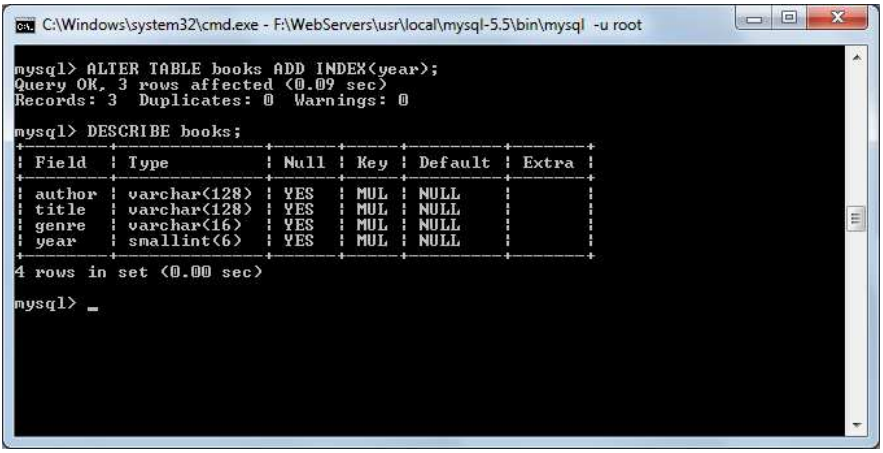


Рис. 12. Таблица books с добавленными индексами

Также индекс можно добавить командой CREATE INDEX. Например, две приведенные ниже команды являются эквивалентными:

ALTER TABLE books ADD INDEX(author(20));

CREATE INDEX author ON books (author(20));

Разница между этими командами состоит в том, что CREATE INDEX нельзя использовать для создания первичного ключа – PRIMARY KEY.

**Первичные ключи**

В данный момент таблица books имеет существенный недостаток – нет единого уникального ключа для записей таблицы.

Один из вариантов создания такого поля – использование поля-счетчика. Но можно добавить поле isbn, которое будет хранить международный стандартный книжный номер – ISBN.

Если выполнить команду, представленную ниже, то будет получено сообщение об ошибке, связанной с дубликатом записи для ключа 1: «Duplicate entry».

ALTER TABLE books ADD isbn CHAR(13) PRIMARY KEY;

Причина в том, что таблица уже заполнена данными, а эта команда пытается добавить поле со значением NULL к каждой строке, что запрещено, поскольку все поля, использующие первичный ключ должны иметь уникальное значение. Если бы таблица была пустой, то эта команда была выполнена без ошибок.

Можно решить данную проблему – создать поле, заполнить его уникальными данными, обновив каждую запись. Например:

```
ALTER TABLE books ADD isbn CHAR(13);
UPDATE books SET isbn='9781598184891' WHERE year='2005';
UPDATE books SET isbn='9780582506206' WHERE year='2010';
UPDATE books SET isbn='9780517123201' WHERE year='2011';
ALTER TABLE books ADD PRIMARY KEY(isbn);
DESCRIBE books;
```

После ввода данных команд на экране появится результат, подобный приведенному на рис. 13.

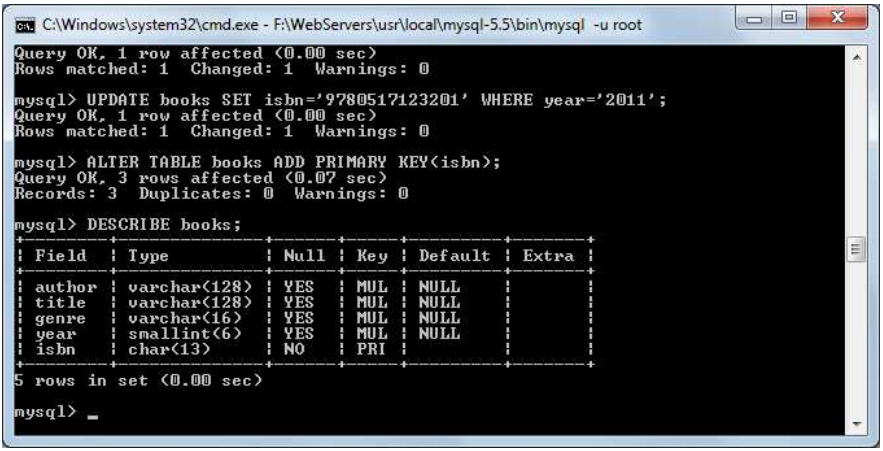


Рис. 13. Таблица books с добавленным первичным ключом

В данном случае обновление записи производилось при помощи команды UPDATE. Для указания какую именно запись обновлять используется конструкция WHERE, задающее условие отбора записи (в данном случае уникальность записи определяется значением поля year – оно у всех внесенных в таблицу записей разное).

После занесения всех значений в поле isbn оно делается первичным ключом. Как видно из рис. 4 теперь свойство Key у поля имеет значение PRI, т.е. теперь поле isbn должно иметь уникальное значение.

Все операции, рассмотренные выше можно было сделать при создании таблицы, например:

```
CREATE TABLE books (
author VARCHAR(128),
title VARCHAR(128),
genre VARCHAR(16),
year SMALLINT,
isbn CHAR(13),
INDEX(author(20)),
INDEX(title(20)),
INDEX(genre(4)),
INDEX(year),
PRIMARY KEY (isbn)) ENGINE MyISAM;
```

### 3.7. Доступ к базе данных MySQL с использованием PHP

Смысл использования PHP в качестве интерфейса к MySQL заключается в форматировании результатов SQL-запросов и придании им внешнего вида, предназначенного для вывода на веб-страницу. Обладая возможностью входа в установленную систему MySQL с помощью своего имени пользователя и пароля, то же самое можно сделать и из PHP. Но вместо использования командной строки MySQL для ввода команд и просмотра выходной инорфмации нужно будет создать стро-

ки запроса, а затем передать их MySQL. Ответ MySQL поступит в виде структуры данных, которую PHP сможет распознать. Затем с помощью команд PHP можно будет извлекать данные и приводить их к формату веб-страницы.

Процесс использования MySQL с помощью PHP заключается в следующем:

1. Подключение к MySQL.
2. Выбор базы данных, которая будет использоваться.
3. Создание строки запроса.
4. Выполнение запроса.
5. Извлечение результатов и вывод их на веб-страницу.
6. Повторение шагов с 3-го по 6-й до тех пор, пока не будут извлечены все необходимые данные.
7. Отключение от MySQL.

**Создание файла регистрации**

Большинство веб-сайтов, разработанных на PHP, содержат множество программных файлов, которым понадобится доступ к MySQL, и им будут нужны сведения, касающиеся входа в систему и пароля. Поэтому имеет смысл создать отдельный файл для их хранения, а затем его включать туда, где он необходим. Его можно назвать login.php и разместить в каталоге, где находится файл главной страницы сайта – index.php.

В файл login.php можно добавить следующие строки:

```
<?php
$db_hostname = 'localhost';
$db_database = 'publications';
$db_username = 'mike';
$db_password = 'mypass';
?>
```

В данном случае через соответствующие переменные указываются: имя сервера, имя базы данных, имя пользователя и пароль – вся информация, необходимая для подключения.

**Подключение к MySQL и выбор базы данных**

После сохранения файла login.php его можно подключить к любому PHP-файлу, которому нужен доступ к базе дан-

ных. Для этого можно воспользоваться инструкцией PHP **require** или **require_once**. Второй вариант более предпочтителен, так как файл считывается только в том случае, если он не был включен в какой-то другой файл, что исключит практически бесполезные повторные обращения к диску.

Для подключения файла можно использовать следующий программный код:

```
<?php
require_once 'login.php';
$db_server = mysql_connect($db_hostname, $db_username, $db_password);
if (!$db_server) exit ("Невозможно подключиться к MySQL: ".mysql_error());
?>
```

В этом примере запускается функция **mysql_connect()**, которой нужны три параметра: имя хоста (hostname) MySQl-сервера, имя пользователя (username) и пароль (password). В случае успешного подключения эта функция возвращает идентификатор сервера, а в случае неудачи – значение FALSE.

Если к MySQL подключиться не удалось, то выводится соответствующее сообщения, а также текст объяснения ошибки (функция **mysql_error()**).

Выбор базы данных осуществляется добавлением в код следующей строки:

```
mysql_select_db($db_database)
or exit("невозможно выбрать базу данных ".mysql_error());
```

Для выбора базы данных используется функция **mysql_select_db()**. Ей нужно передать имя требуемой базы данных. Функция **mysql_select_db()** возвращает значение TRUE или FALSE. Поэтому в примере используется оператор or, который в случае неудачи подключения к базе возвращает сообщение об ошибке.

**Создание и выполнение запроса и извлечение результата**

Отправка запроса к MySQL из PHP сводится к вызову функции **mysql_query()**. Для ее использования можно добавить в пример следующий код:

```
$query = "SELECT * FROM books";
$result = "mysql_query($query)";
if (!$result) exit ("Сбой при доступе к базе данных: ".mysql_error());
```

Сначала переменной $query присваивается значение, содержащее код предстоящего запроса. В данном случае запрашивается просмотр всех строк таблицы books.

Для выполнения запроса используется функция **mysql_error()**, которая возвращает результат в переменную $result. В случае успешного выполнения запроса переменная $result будет содержать ресурс, позволяющий извлечь результаты этого запроса. При сбое переменная result вернет значение FALSE и будет выведено соответствующее сообщение.

Результат, возвращенный функций **mysql_query()**, можно использовать для извлечения требуемых данных. Наиболее простой способ заключается в последовательном извлечении нужных значений с помощью функции **mysql_result()**.

Для извлечения результата можно добавить в программу следующий код:

```
$rows = mysql_num_rows($result);
        for ($j = 0; $j < $rows; $j++)
        {
echo 'Author: '.mysql_result($result, $j, 'author').'<br>';
echo 'Title: '.mysql_result($result, $j, 'title').'<br>';
echo 'Genre: '.mysql_result($result, $j, 'genre').'<br>';
echo 'Year: '.mysql_result($result, $j, 'year').'<br>';
echo 'ISBN: '.mysql_result($result, $j, 'isbn').'<br><br>';
        }
```

Для начала переменной $rows присваивается значение, возвращенное функцией **mysql_num_rows()**. Эта функция возвращает количество строк, возвращенных запросом.

Далее идет цикл for, который извлекает каждую ячейку данных из каждой строки с помощью функции **mysql_result()**. В качестве параметров этой функции используются: ресурс $result, возвращенный функцией **mysql_query()**, номер строки $j и имя графы, из которой следует извлечь данные.

Затем результаты, получаемые при каждом вызове функции **mysql_result()**, включаются в инструкции echo для отображения по одному полю на каждой строке с дополнительным символом перевода строки между строками. Результат запуска этой программы показан на рис. 14.

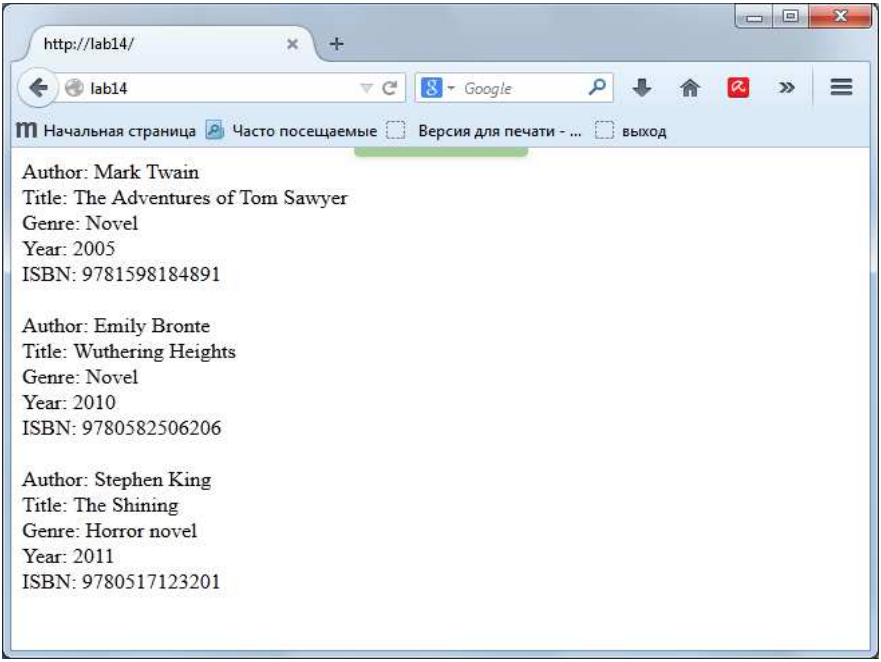


Рис. 14. Информация из таблицы publications, выведенная на экран с помощью PHP

**Извлечение строки**

Можно сделать более эффективное извлечение данных из результата SQL-запроса. Для этого можно заменить приведенный выше код (начиная с заголовка цикла for) следующим альтернативным вариантом:

```
for ($j = 0; $j < $rows; $j++)
        {
                $row = mysql_fetch_row($result);
                echo 'Author: '.$row[0].'<br>';
                echo 'Title: '.$row[1].'<br>';
                echo 'Genre: '.$row[2].'<br>';
                echo 'Year: '.$row[3].'<br>';
                echo 'ISBN: '.$row[4].'<br><br>';
        }
```

В данном случае используется функция **mysql_fetch_row()**, которая извлекает одну строку данных из $result. Данная строка присваивается переменной $row.

После извлечения данных останется только обратиться по очереди к каждому элементу массива $row, начиная с нулевого.

**Отключение**

По окончании работы с базой данных от нее следует отключиться. Это делается с помощью функции mysql_close():

```
mysql_close($db_server);
```

Здесь функции нужно передать тот идентификатор, который был возвращен функцией **mysql_connect()** и сохранен в переменной $db_server.

Все подключения к базам данных автоматически закрываются при выходе из PHP. Но в более длинных программах, где могут осуществляться подключения к базам данных и отключения от них, настоятельно рекомендуется по завершении доступа закрывать каждое подключение к базе данных.

**Практический пример**

Рассмотрим пример, которые позволяет не только выводить данные из базы MySQL, но вводить и удалять записи из таблицы. Результат его работы представлен на рис. 15.

```
<?php
        require_once 'login.php';
        $db_server = mysql_connect($db_hostname, $db_username, $db_password);
        if (!$db_server) exit ("Невозможно подключиться к MySQL: ".mysql_error());
        mysql_select_db($db_database)
    or exit("невозможно выбрать базу данных ".mysql_error());
        if (isset($_POST['author']) &&
                isset($_POST['title']) &&
                isset($_POST['genre']) &&
                isset($_POST['year']) &&
                isset($_POST['isbn']))
        {
                $author = get_post('author');
                $title = get_post('title');
                $genre = get_post('genre');
                $year = get_post('year');
                $isbn = get_post('isbn');
                $query = "INSERT INTO books VALUES".
                "('$author', '$title', '$genre', '$year', '$isbn')";
                if (!mysql_query($query, $db_server))
        echo "Сбой при вставке данных: $query <br>".
                        mysql_error()."<br><br>";
        }
        else
    if (isset($_POST['delete']) && isset($_POST['isbn_del']))
                {
                        $isbn = get_post('isbn_del');
    $query = "DELETE FROM books WHERE isbn='$isbn'";
```

```
        if (!mysql_query($query, $db_server))
                echo "Сбой при удалении данных: $query <br>".
                                        mysql_error()."<br><br>";
                        }
                echo <<<_END
                <form action="index.php" method="post">
        <pre>
                Author<input type="text" name="author">
                Title           <input type="text" name="title">
                Genre           <input type="text" name="genre">
                Year            <input type="text" name="year">
                ISBN            <input type="text" name="isbn">
                <input type="submit" value="Добавить запись">
        </pre>
                </form>
        _END;
                $query = "SELECT * FROM books";
                $result = mysql_query($query);
                if (!$result) exit ("Сбой при доступе к базе данных:
".mysql_error());
                $rows = mysql_num_rows($result);
                for ($j = 0; $j < $rows; $j++)
                {
                        $row = mysql_fetch_row($result);
                        echo <<<_END
        <pre>
                Author:         $row[0]
                Title:          $row[1]
                Genre: $row[2]
                Year:           $row[3]
                ISBN:  $row[4]
        </pre>
                <form action="index.php" method="post">
                <input type = "hidden" name="delete" value="yes">
        <input type = "hidden" name="isbn_del" value="$row[4]">
```

```
            <input type = "submit" value="Удалить запись">
                </form>
    _END;
            }
            mysql_close($db_server);
            function get_post($var)
            {
                return
mysql_real_escape_string($_POST[$var]);
            }
    ?>
```

В самом начале программы происходит подключение к самой базе данных. После этого происходит проверка, ввел ли пользователь данные для добавления новой записи. Если да, то происходит добавление новой записи в базу с помощью запроса:

```
$query = "INSERT INTO books VALUES".
    "('$author', '$title', '$genre', '$year', '$isbn')";
```

Если пользователь не ввел данные, то проверяется не нажимал ли он кнопку «Удалить запись». Если да, то из скрытого поля ввода «isbn_del» считывается значение ключевого поля удаляемой записи. После этого выполняется запрос:

```
$query = "DELETE FROM books WHERE isbn='$isbn'";
```

Если пользователь не совершал никаких действий перед вызовом программы, то происходит вывод содержимого таблицы books на экран. Каждая выведенная на экран запись снабжается кнопкой «Удалить запись», позволяющей стереть эту запись из таблицы базы данных.

В конце происходит отключение программы от базы данных.

Также в программу добавлена пользовательская функция **get_post()**. Внутри нее происходит вызов функции

**mysql_real_escape_string()**, которая экранирует специальные символы в строках для использования в выражениях SQL. После такой обработки результат можно безопасно использовать в SQL-запросе в функции **mysql_query()**.

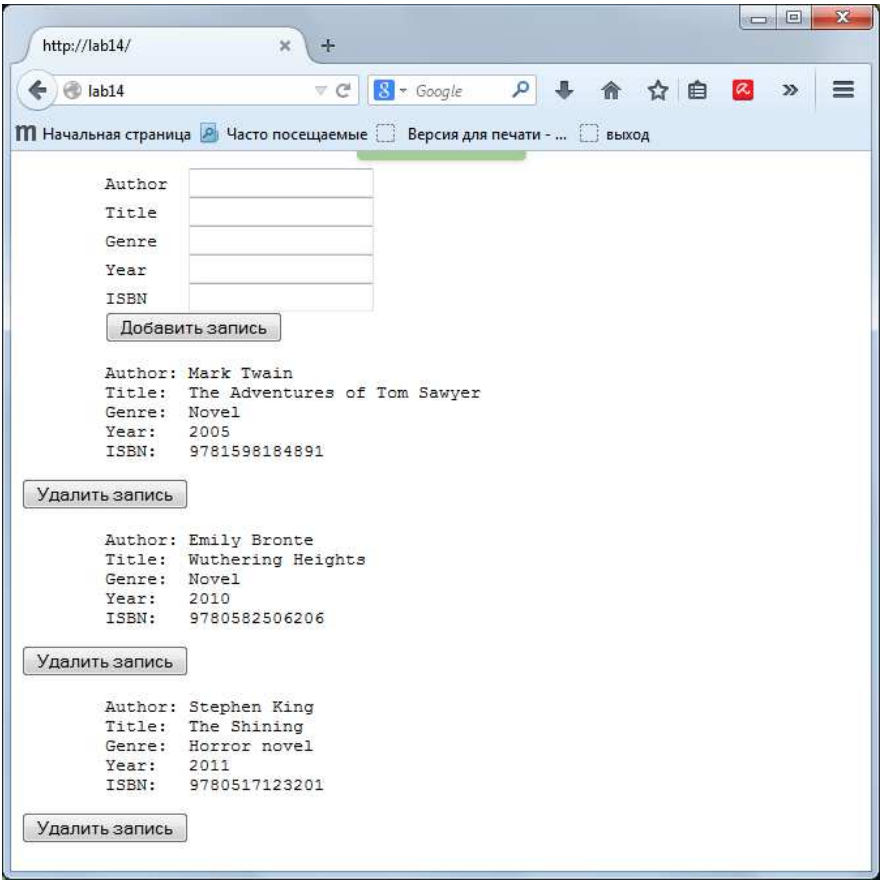


Рис. 15. Результат работы программы на PHP, работающей с базой данных MySQL

# ЗАКЛЮЧЕНИЕ

Учебное пособие рассматривает основные вопросы разработки веб-приложений с помощью языков программирования JavaScript и PHP.

Освоение программирования на JavaScript и PHP, технологий создания динамических элементов веб-страниц и веб-сайтов позволит разрабатывать качественные программы для решения различных задач представления и обработки информации в Интернете.

Пособие содержит описание основных функций данных языков, методов создания сценариев для повышения интерактивности веб-страницы, функций и модулей специальных пользовательских библиотек языке JavaScript. Особое внимание в учебном пособии уделяется вопросам взаимодействия веб-приложений с базами данных, созданных средствами СУБД MySQL.

Учебное пособие подробно излагает правила написания кода на языке JavaScript, базовые элементы языка, работу с массивами и циклами, регулярными выражениями и данными различного типа.

Особое внимание уделено созданию динамических веб-страниц, модифицированию веб-страниц с использованием jQuery, работе с событиями JavaScript, использованию AJAX.

Материалы пособия содержат описание языка PHP, рассматривает базовые конструкции языка, работу с массивами, датами, форматирование данных, работу с файлами и взаимодействие с веб-формами, описание создания базы данных и таблиц в MySQL, излагает технологию доступа к базе данных из PHP.

Таким образом, учебное пособие содержит описание основных технологий, применяемых при разработке динамических веб-ориентированных систем.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Сергеев, М. Ю. Web-дизайн: создание Web-сайтов с помощью HTML и CSS [Текст]: учеб. пособие / М.Ю. Сергеев. – Воронеж: ГОУВПО «ВГТУ», 2012. – 219 с.

2. Справочник по HTML и CSS [Электронный ресурс]: Режим доступа: World Wide Web. URL: http://htmlbook.ru/.

3. Никсон, Р. Создаем динамические веб-сайты с помощью PHP, MySQL и JavaScript [Текст] / Р. Никсон. – СПб.: Питер, 2011. – 496 с.

4. Маклафлин, Б. PHP и MySQL. Исчерпывающее руководство [Текст] / Б. Маклафлин. – СПб.: Питер, 2016. – 544 с.

5. Вилтон П. JavaScript. Руководство программиста [Текст] / П. Вилтон, Дж. МакПик. – СПб.: Питер, 2009. – 720 с.

## ОГЛАВЛЕНИЕ

Учебное издание

Сергеев Михаил Юрьевич
Сергеева Татьяна Ивановна

# ОСНОВЫ ВЕБ-ПРОГРАММИРОВАНИЯ

В авторской редакции

Подписано в печать 21.11.2016.
Формат 60×84/16. Бумага для множительных аппаратов.
Усл. печ. л. 15,9. Уч.-изд. л. 12,6. Тираж 250 экз.
Заказ. №

ФГБОУ ВО «Воронежский государственный
технический университет»
394026 Воронеж, Московский просп., 14