МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Юго-Западный государственный университет»
(ЮЗГУ)

М. В. Бородин, Е. А. Титенко

ИНТЕРНЕТ-ТЕХНОЛОГИИ

Учебное пособие

Курск 2013

УДК 004.738.5, 004.432.2
ББК 38/9
Б83

Рецензенты:
Доктор технических наук, профессор, проректор по информатизации
ФГОУ ВПО «Госуниверситет — УНПК» *А. В. Коськин*
Доктор технических наук, профессор, начальник НОЦ НИЦ
ФГУП «18 ЦНИИ» МО РФ *В. Н. Николаев*

Б83 **Бородин М. В., Титенко Е. А.**

Интернет-технологии : учеб. пособие / М. В. Бородин, Е. А. Титенко; Юго-Зап. гос. ун-т. Курск, 2013. 140 с.: прилож. 1. Библиогр.: 129–130.

ISBN

Учебное пособие соответствует Федеральному государственному образовательному стандарту направления 230400 «Информационные системы и технологии». Даны основные сведения о языках программирования PHP и JavaScript. Кратко рассмотрено создание Web-страниц с использованием HTML и CSS. Приводятся справочные сведения о протоколе HTTP. Предназначено для студентов, обучающихся по направлению 230400.

УДК 004.738.5, 004.432.2
ББК 38/9

© Юго-западный государственный университет, 2013
© Бородин М. В., Титенко Е. А. 2013

Предисловие

В основе данного учебного пособия лежит курс лекций по дисциплине «Интернет-технологии», читаемых в Юго-Западном государственном университете на кафедре информационных систем и технологий. Пособие может использоваться как в качестве дополнительной литературы студентами, слушающими соответствующий курс, так и для самостоятельного изучения.

Целью учебного пособия является знакомство читателя с основными технологиями, лежащими в основе современных Интернет-приложений. Упомянутые технологии рассматриваются прежде всего с точки зрения разработчика, а не дизайнера.

Пособие состоит из четырех частей и приложения, посвященного описанию протокола HTTP. В главах 1 и 2 описываются соответственно язык гипертекстовой разметки HTML и каскадные таблицы стилей. Эти главы в известной степени являются вводными — их задача заключается в том, чтобы, во-первых, заполнить потенциальные пробелы в знаниях, которые могли остаться у читателя после изучения HTML и CSS на ранних курсах, и, во-вторых, расширить эти знания до уровня, необходимого для создания динамических страниц, содержание которых генерируется сервером на основе данных, содержащихся в базе данных.

Основное содержание пособия составляют главы 3 и 4, описывающие языки программирования PHP и JavaScript. Задача этих глав — познакомить читателя как с синтаксисом и встроенными функциям конкретных языков, так и дать общее представление о применяемых в настоящее время динамических языках, типичными представителями которых являются языки PHP и JavaScript. Также в задачи глав 3 и 4 входит демонстрация ряда шаблонов проектирования, как специфичных для данных языков, так и независимых от языка, характерных для разработки Интернет-приложений либо имеющих универсальный характер. Описание шаблонов проектирования дается в виде примеров и не претендует на полноту и систематичность. За более полным изложением шаблонов проектирования читатель отсылается к соответствующей литературе.

Для успешного освоения материла, содержащегося в главах 3 и 4, необходимо знание основ программирования (желательной яв-

ляется определенная степень знакомства с языками C/C++ или Java). Также для изучения части материала главы 3 требуется знание основ реляционных баз данных и языка SQL.

Пособие не содержит справочного материла. Это сделано не только с тем, чтобы ограничить объем пособия и не отпугнуть потенциального читателя, но и поскольку автор находит, что интерактивная электронная форма представления справочной информации несравненно более удобна нежели печатная. За справочной информацией читатель отсылается, в частности, к следующим ресурсам: http://www.php.net/manual/ru/ (PHP) и https://developer.mozilla.org/ru/docs/JavaScript (JavaScript).

Авторы выражают благодарность сотрудникам и студентам кафедры информационных систем и технологий Юго-Западного государственного университета за ценные предложения и замечания.

Введение

Видимо, невозможно переоценить то значение, которое возникновение и повсеместное распространение сети Интернет имело для развития информационных систем и технологий. Однако Интернет — это не только возможность обмена данными между компьютерами, расположенными в любых точках Земного шара, но и платформа для построения приложений. Такие приложения получили название Web-приложений.

По сравнению с традиционным настольным (desktop) приложениями Web-приложения обладают целым рядом неоспоримых преимуществ:

- приложение не требует установки на клиенте. Все, что требуется от клиента — это браузер. В случае выпуска новой версии или обновления существующей необходимо обновление только на сервере, что значительно снижает издержки, связанные с поддержкой приложения;
- предоставляемый браузерами набор стандартных интерфейсов программирования обеспечивает приложению широкие возможности по созданию графического пользовательского интерфейса, абстрагируя его от особенностей аппаратного обеспечения и операционной системы клиента, благодаря чему отпадает необходимость разработки отдельных версий приложения, например, для Windows и Unix;
- хранение приложениями данных на сервере дает возможность пользователям работать с их данными, подключаясь к сети Интернет с любого компьютера или мобильного устройства;
- выполняясь в контексте браузера, приложения в достаточной степени изолированы от хранящихся на клиенте персональных данных пользователя, представляя тем самым меньшую угрозу их несанкционированного раскрытия и изменения.

Однако вместе с перечисленным, необходимо заметить, что на пути создания Web-приложений разработчика подстерегают и определенные специфические трудности:

- принципиальное отличие архитектуры Web-приложения от таковой для традиционных приложений (это выражается в частно-

сти в разделении приложения на клиентскую и серверную части и вызванных этим коммуникационных задержках);

- необходимость вести разработку клиентской и серверной частей на разных языках программирования, как правило, отличающихся от языков программирования, применяемых при разработке традиционных приложений;
- невозможность выйти за рамки того набора интерфейсов, которые предоставляются браузером;
- необходимость защиты конфиденциальной информации от ее несанкционированного раскрытия и изменения при передачи по сети;
- необходимость изоляции данных пользователей при их хранении на сервере;
- необходимость такой организации доступа к ресурсам сервера, которая делала бы возможной параллельную обработку запросов и масштабирование.

Впрочем, указанные трудности, как правило, успешно преодолеваются, в результате чего Web-приложения занимают все новые ниши, успешно конкурируя с традиционными настольными приложениями практически во всех областях.

Глава 1. HTML

Веб-страница — это текстовый документ, который обычно является представлением некоторого ресурса всемирной паутины (WWW — World-Wide Web). Веб-страница может содержать списки, таблицы, графические изображения, а также ссылки на другие ресурсы. Благодаря наличию ссылок содержимое веб-страницы часто называют гипертекстом.

Подобно разработке программы создание веб-страницы состоит в написании исходного кода с той лишь разницей, что в первом случае исходный код записывается на некотором языке программирования, а во втором — на специальном языке разметки гипертекста (HTML — Hypertext Markup Language). При написании исходного кода веб-страницы, так же, как и исходного кода программы, можно использовать как обычные текстовые редакторы типа Блокнота, так и специализированные редакторы и среды, поддерживающие подсветку синтаксиса, подсказки и автопродолжение.

1.1. Структура страницы

Приведем пример исходного кода простейшей веб-страницы.

```
<!DOCTYPE HTML PUBLIC
    "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Заголовок страницы</title>
  </head>
  <body>
    <p>Текст страницы</p>
  </body>
</html>
```

Если эту страницу открыть в браузере, то в основой части окна будет выведен текст страницы, а в заголовке окна — заголовок страницы.

Исходный код страницы начинается с объявления типа, которое имеет вид `<!DOCTYPE … >`. Объявление типа задает использующуюся версию языка HTML. Остальная часть исходного кода состоит из тегов и обычного текста. Теги могут быть открывающими

и закрывающими. В данном примере открывающие теги — это <html>, <head>, <title>, <body> и , а закрывающие — </html>, </head>, </title>, </body> и .

Теги задают структуру страницы в виде вкладываемых друг в друга элементов, причем открывающий тег задает начало одноименного элемента, а закрывающий — конец. Видно, что в данном примере страница состоит из элементов html, head, title, body и p. Элемент html заключает в себя всю страницу и поэтому называется элементом документа (имя элемента документа указывается в объявлении типа документа сразу после ключевого слова DOCTYPE). Непосредственно в элементе HTML располагаются элементы head и body, включающие в себя в свою очередь элементы title и p соответственно. Наконец, элементы title и P содержат обычный текст.

Назначение элемента определяется его именем, поэтому имена элементов являются предопределенными и встроены в язык HTML подобно тому, как ключевые слова встроены в язык программирования. Также именем элемента определяется, какие другие элементы могут располагаться в данном элементе, а также может ли элемент содержать обычный текст. Возвращаясь к вышеприведенному примеру, отметим, что:

- элемент html представляет всю страницу и должен содержать элементы head и body;
- элемент head служит для группировки различной служебной информации о странице, не составляющей самого ее содержания. Элемент head должен включать элемент title;
- элемент title задает название страницы, отображаемое в заголовке окна браузера. Содержанием элемента title должен быть обычный текст, не содержащий элементов;
- элемент body задает само содержание страницы. Он должен содержать один или несколько блоковых элементов;
- элемент p является блоковым и задает абзац текста. Содержанием элемента p может быть как обычный текст (как в вышеприведенном примере), так и строчные элементы.

Для некоторых элементов закрывающий, а иногда и открывающий тег является необязательным и может быть пропущен в

том случае, если это не вызывает неоднозначности. В частности, предыдущий пример может быть записан так:

```
<!DOCTYPE HTML PUBLIC
    "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<title>Заголовок страницы</title>
<p>Текст страницы
```

В данном случае опущены оба тега элементов `html`, `head` и `body`, а также закрывающий тег элемента `p`. Несмотря на это, с точки зрения структуры, данная страница ничем не отличается от той, которая приведена ранее, т. е., например, в данной странице присутствует элемент `html`, содержащий элементы `head` и `body`.

Для повышения удобочитаемости исходного кода страницы в нем, так же, как и в исходном коде программы, можно использовать дополнительные пробелы и переносы на новую строку, причем переносить на новую строку можно в любом месте, где допустим пробел.

1.2. Комментарии

В исходном коде страницы можно помещать комментарии. Например:

```
<!-- этот текст является комментарием -->
```

Комментарии можно ставить в любом месте, в котором допустим пробел.

1.3. Сущности

Кроме тегов и обычного текста в исходном коде могут встречаться т. н. сущности. В отличие от тегов сущность обычно не определяет элемента, а служит для подстановки в текст символа, который либо имеет специальное значение (угловые скобки, амперсанд и двойные кавычки), либо отсутствует в используемом наборе символов (например, греческие буквы). Приведем примеры:

Исходный код	Что получается
`1&lt;2`	1<2
`&laquo;Знание &mdash; сила&raquo;`	«Знание — сила»

Исходный код	Что получается
`&alpha; &#946; &#x3B3;`	α β γ
`&#x65E5`	日

Сущности бывают именованными и числовыми. В первом случае символ задается именем, а во втором — кодом в соответствии с кодировкой Unicode. В данном примере используются именованные сущности `lt`, `laquo`, `nbsp`, `mdash`, `raquo` и `alpha`, а также числовые сущности, соответствующие символам с кодами 946, $3B3_{16}$, $65E5_{16}$, 6708_{16} и $706B_{16}$. При использовании числовой сущности код символа может задаваться как в десятичной, так и в шестнадцатеричной системе счисления (в последнем случае после знака решетки ставится буква `x`).

1.4. Строчные элементы

К строчным элементам относятся:

- `big`, `small` — увеличивают и уменьшают размер шрифта;
- `sup`, `sub` — преобразуют текст в верхний или нижний индекс;
- `em`, `strong` — выделяет текст, причем `STRONG` означает более сильное выделение, чем `EM`;
- `dfn` — текст определения чего-либо;
- `q` — текст цитаты;
- `cite` — источник цитаты.

Все перечисленные элементы могут содержать как обычный текст, так и другие строчные элементы. Приведем пример:

```
Итак, <dfn>тип данных &mdash;
это <strong>множесто</strong>
значений&hellip;</dfn>.
```

Также к строчным элементам относится элемент `br`, использующийся для переноса на новую строку. Этот элемент имеет пустое содержание.

1.5. Блочные элементы

Помимо абзацев, задаваемых элементом `p`, содержимое страницы может включать заголовки (не путать с названием страницы, задаваемым элементом `title`), задаваемые элементами `h1`, `h2`, ..., `h6`, причем элемент `h1` задает заголовок первого уровня, `h2` — второго и так далее. Заголовки являются блочным элементами, а их содержание подчиняется тем же правилам, что и содержание элемента `p`.

Еще один вид блочных элементов — это `blockquote`. Этот элемент задает блочную цитату. В отличие от элемента `q` элемент `blockquote` должен содержать один или несколько блочных элементов. Обычно браузеры выделяют цитаты отступом.

Кроме абзацев, блочных цитат и заголовков еще одним видом блочных элементов являются списки. Поддерживается три вида списков: нумерованный, маркированный и список определений. Нумерованный и маркированный список задаются соответственно элементами `ol` и `ul`. Эти элементы должны содержать один или несколько элементов `li`, задающих каждый отдельный элемент списка. В свою очередь содержимым элементов `li` может быть как обычный текст, так и один или несколько блочных или строчных элементов. Поскольку списки сами являются блочными элементами, и, стало быть, могут включаться в элемент `li`, возникает возможность создавать многоуровневые списки. Например:

```
<ol>
  <li>
    Visual Basic for Applications
    <ol>
      <li>Типы данных</li>
      <li>Встроенные функции</li>
      <li>Массивы</li>
    </ol>
  </li>
  <li>HTML</li>
</ol>
```

Список определений задается элементом `dl`. Этот элемент должен содержать одну или несколько пар элементов `dt` и `dd`. Элемент `dt` задает определяемый термин, а элемент `dd` — определение.

Содержимым элемента `dt` может быть обычный текст или строчные элементы, а элемента `dd` — также еще и блочные элементы.

1.6. Атрибуты

В языке HTML элементы могут иметь не только имя, но и набор атрибутов. Атрибут состоит из имени и значения. Атрибуты перечисляются в открывающем теге. Приведем пример:

```
<p lang="ru">Текст по-русски</p>
```

Здесь элемент `p` снабжен одним атрибутом `lang`, значением которого является `ru`. Каждый атрибут может встречаться в теге не более одного раза; порядок перечисления значения не имеет. Значение атрибута можно записывать как в двойных, так и в одинарных кавычках. Если кавычки встречаются в самом значении атрибута, то их следует удвоить.

Набор допустимых атрибутов, а также множество допустимых значений каждого из атрибутов зависит от имени элемента. Так, использованный в данном примере атрибут `lang` задает язык; его значением должен быть код языка, например: `ru`, `en`, `fr`, `de`, `jp` и так далее. Этот атрибут может применяться ко всем элементам, в которых (прямо или косвенно) может содержаться текст, отображаемый на экране. Обычно атрибут `lang` применяется к элементу `html`.

1.7. Гиперссылки

Гиперссылка — это фрагмент страницы, щелкнув на котором в окне браузера пользователь может перейти к другой странице или же к другой части данной страницы. Обычно браузер выделяет ссылки подсветкой и подчеркиванием. Гиперссылка является строчным элементом и задается элементом `a`. Например:

```
<a href="more-info.html">Дополнительная информация</a>
```

Адрес перехода задается атрибутом `href`; этот адрес может указывать как на веб-страницу, так и на ресурс любого другого типа. Адрес должен иметь форму универсального идентификатора ресурса (URI — Universe Resource Identifier) и может быть как абсолютным, так и относительным. В приведенном примере адрес является относительным. Это означает, что, если, например, страница, содержащая эту ссылку, находится по адресу

http://www.example.ru/data/index.html, то, активизировав ее, пользователь перейдет на страницу, находящуюся по адресу http://www.example.ru/data/more-info.html.

Кроме перехода к другому ресурсу гиперссылку можно использовать для перехода к другой части данной страницы. Для этого соответствующую часть снабдить идентификатором, задаваемым атрибутом `id`, например:

```
<p id="more-info">Здесь размещается дополнительная
информация</p>
```

Подобно атрибуту `lang`, атрибут `id` является глобальным и может применяться к любым элементам. В пределах одной страницы не должно быть нескольких элементов с одинаковым идентификатором. При записи идентификатора можно использовать только буквы (не обязательно латинские), цифры и знаки подчеркивания и тире, причем первый символ идентификатора должен быть буквой.

Для перехода к помеченной части следует задать значение атрибута `href` в виде `#more-info`. Метку можно использовать и при переходе к другой странице; в этом случае `href` может иметь вид `info.html#more-info`.

1.8. Изображения

Для вставки изображений используется строчный элемент `img`. Адрес, по которому находится графическое изображение, задается атрибутом `src`. Как и в случае атрибута `href` адрес может быть как абсолютным, так и относительным. Обычно используются изображения в формате GIF, JPEG или PNG. Атрибут `src` является обязательным. Другой обязательный атрибут — `alt` — задает текст, который будет выводиться браузером вместо графического изображения в том случае, если загрузка последнего еще не завершена либо не удалась. Кроме обязательных атрибутов `src` и `alt` к элементу `img` можно применить атрибуты `width` и `height`, задающие размер изображения в пикселях или в процентах относительно размера включающего блока. Основное назначение атрибутов `width` и `height` — не изменение размеров (этого не следует делать из-за возможности снижения качества), а возможность сообщить браузе-

ру размеры изображения до фактической загрузки самого изображения. Элемент `img` не имеет содержания. Приведем пример:

```
<img src="photo.jpg" alt="Фотография"
     width="100" height="150">
```

1.9. Таблицы

Кроме вышеперечисленных блочных элементов существует еще один — элемент `table`. Этот элемент задает таблицу. Приведем пример простейшей таблицы:

```
<table rules="all">
  <tbody>
    <tr><th>Страна</th><th>Столица</th></tr>
    <tr><td>Россия</td><td>Москва</td></tr>
    <tr><td>Великобритания</td><td>Лондон</td></tr>
    <tr><td>Германия</td><td>Берлин</td></tr>
  </tbody>
</table>
```

Элемент `table` должен содержать в себе, по крайне мере, один элемент `tbody`. Элемент `tbody` представляет группу строк таблицы и должен включать один или несколько элементов `tr`, представляющих отдельные строки, которые, в свою очередь, должны включать один или несколько элементов `td` или `th`. Элементы `td` и `th` представляют отдельные ячейки таблицы и могут содержать как обычный текст, так и строчные и блочные элементы. Элемент `th` отличается от `td` тем, что задает ячейку, использующуюся в качестве заголовка строки или столбца. Атрибут `rules` задает разделительные линии; возможные значения этого атрибута: `none` (без разделительных линий), `groups` (разделительные линии между группами строк и столбцов), `rows` (разделительные линии между отдельными строками), `cols` (разделительные линии между отдельными строками), `all` (все разделительные линии).

К элементам `td` и `th` можно применять атрибуты **`rowspan`** и **`colspan`**, которые позволяют растянуть соответствующую ячейку на указанное количество строк и столбцов.

В элементе `table` элементам `tbody` может предшествовать по одному элементу `thead` и `tfoot`. Эти элементы задают соответственно заголовок и подвал таблицы. Их содержание аналогично элементу `tbody`.

Для задания параметров колонок могут использоваться элементы `col` и `colgroup`, включаемые в элемент `table` перед элементами `tbody`, `thead` и `tfoot`. К элементу `col` можно применить атрибут `width`, задающий ширину колонки. Ширина может задаваться как в пикселях, так и в процентах от ширины всей таблицы. Кроме этого ширина может быть относительной, заданной долей ширины таблицы за вычетом общей ширины тех столбцов, ширина которых задана в пикселях или процентах. Также к элементу `col` можно применить атрибут `span`, задающий количество столбцов, на которые воздействует данный элемент. Элемент `colgroup` позволяет сгруппировать несколько элементов `col`. К элементу `colgroup` применимы те же атрибуты, что и к элементу `col`.

Ко всем вышеперечисленным элементам, используемым для создания таблиц, также применимы следующие атрибуты выравнивания:

- `align` — горизонтальное выравнивание. Возможные значения: `left` (по левому краю), `center` (по центру), `right` (по правому краю), `justify` (по обоих краям) и `char` (по указанному символу);
- `char` — символ для выравнивания (используется, если значением атрибута `align` является `char`);
- `valign` — вертикальное выравнивание. Возможные значения: `top` (по верхнему краю), `middle` (по центру), `bottom` (по нижнему краю) и `baseline` (по базовой линии).

Приведем пример:

```
<table rules="all">
  <col>
  <col span="2" align="right"/>
  <thead>
    <tr>
      <th rowspan="2">Страна</th>
      <th colspan="2">Показатели</th>
    </tr>
    <tr>
      <th>ВВП</th>
      <th>ИРЧП</th>
    </tr>
  </thead>
  <tbody>
    <tr>
```

```
      <td>Россия</td>
      <td>2 087 815</td>
      <td>0,719</td>
    </tr>
    <tr>
      <td>Великобритания</td>
      <td>2 137 421</td>
      <td>0,849</td>
    </tr>
    <tr>
      <td>Германия</td>
      <td>2 809 693</td>
      <td>0,885</td>
    </tr>
  </tbody>
</table>
```

Наконец, таблица может иметь название. Оно задается элементом `caption`, содержимым которого может быть обычный текст и строчные элементы. Элемент `caption` размещается в элементе `table` перед всеми остальными элементами.

Еще пример таблицы:

```
<table summary="Объединение ячеек">
  <tbody>
    <tr>
      <td rowspan="3">1</td>
      <td colspan="2">2</td>
      <td rowspan="3">3</td>
    </tr>
    <tr>
      <td>4</td>
      <td>5</td>
    </tr>
    <tr>
      <td colspan="2">6</td>
    </tr>
  </tbody>
</table>
```

1.10. Формы

Web-страницы могут использоваться не только для вывода информации. Ввод информации можно организовать с использованием форм. По своей структуре данные, отправляемые формой, представляют собой набор свойств, каждое из которых состоит из

имени и значения, причем одни и те же имена свойств могут повторяться в комбинации с разными значениями.

В коде форма представляет собой элемент `form`, размещаемый в пределах тела страницы и содержащий элементы управления (поля ввода, кнопки, списки и т. п.) вперемешку с обычными элементами разметки. Одна страница может содержать несколько форм.

У элемента form могут быть следующие атрибуты:

• `action` — адрес, на который будут отправлены данные, введенные в элементы управления формой. Предполагается, что по этому адресу располагается серверный сценарий, который сможет как-либо обработать данные формы (например, эти данные могут быть занесены в базу данных). По умолчанию данные отправляются на тот же адрес, откуда была получена страница, содержащая форму. У адреса, заданного атрибутом `action`, может быть схема отличная от `http` и `https`. Например, если используется схема `mailto`, то данные формы отправляются на заданный адрес электронной почты;

• `method` — способ отправки данных. Возможными значениями этого атрибута являются ключевые слова `get` и `post`. При использовании в атрибуте action схем `http` и `https` атрибут `method` задает используемый метод протокола HTTP, а при использовании схемы `mailto` способ размещения данных в почтовом сообщении. Если для отправки данных используется метод `GET`, то данные формы подставляются в строку запроса и становятся видны на экране браузера в строке адреса; если же используется метод `POST`, то отправляемые данные формы образуют тело запроса и на экране не показываются. Также необходимо принимать во внимание следующее: браузер считает, что выполнение метода `POST` может изменить состояние сервера и поэтому каждый раз при попытке его выполнения требует от пользователя явного подтверждения;

• `enctype` — способ кодирования данных формы. Возможные значения этого атрибута: `application/x-www-form-urlencoded` (это значение по умолчанию) и `multipart/form-data`. Кодирование `multipart/form-data` используют тогда, когда форма предназначена для отправки файлов (т. е. на форме есть элемент управления, позволяющий выбрать файл, либо присоединение файлов реализовано программно с использованием сценария на языке JavaScript).

Отметим, что при использовании кодирования `multipart/form-data` значением атрибута `method` должно быть ключевое слово `post`.

Простейшие элементы управления (поля ввода, кнопки и переключатели) представляются элементом `input`. В данных формы каждому простейшему элементу может соответствовать только одно свойство. Атрибуты элемента `input`:

• `name` — имя элемента — задает имя свойства, под которым значение данного компонента будет представлено в данных формы. Если этот атрибут опустить, то данные, введенные в элемент управления, отправлены не будут, хотя сам он будет присутствовать на экране;

• `value` — значение элемента — задает значение соответствующего свойства в данных формы. Если элемент управления дает возможность пользователю редактировать содержащееся в нем значение, то данный атрибут (если он задан) интерпретируется как значение по умолчанию;

• `type` — тип элемента управления. Для задания используются следующие константы:

- `text` — поле для ввода текста;

- `password` — как предыдущее, но вместо вводимого текста отображаются звездочки или точки. Используется для ввода паролей;

- `checkbox` — независимый переключатель (обычно в виде квадратика, в котором можно поставить галочку);

- radio — зависимый переключатель (обычно в виде кружка, в котором можно поставить точку). От независимого отличается тем, что в каждый момент времени из всех имеющихся на одной форме переключателей с одинаковым значением атрибута `name` включенным может быть только один (переключатели с разным значением атрибута `name` могут быть включены одновременно);

- `submit` — кнопка, при щелчке на которой происходит отправка формы. По умолчанию после отправки формы браузер переходит по адресу, заданному в атрибуте `action`. Надпись на кнопке задается атрибутом `value`.

- `reset` — кнопка, при щелчке на которой происходит сброс значений, введенных в элементы управления формы;

- `button` — кнопка без предопределенного действия. Для задания действий используется сценарий на языке JavaScript;

- `file` — поле для выбора файла, содержимое которого будет отправлено в составе данных формы. Отправка файла возможна только в том случае, когда для формы задана кодирование `multipart/form-data`;

- `hidden` — скрытое поле. Поле данного типа не отображается на экране. Используется для того, чтобы связать с формой дополнительные данные;

• `size` — размер поля для ввода, измеренный в символах. Этот параметр не ограничивает количества символов, которое можно ввести в это поле;

• `maxlength` — максимальное количество символов, которое можно ввести в поле ввода;

• `checked` — является ли переключатель по умолчанию включенным. Этот атрибут является логическим и записывается без значения.

У элемента `input` не должно быть содержания.

Обычно элементы управления, будучи расположенными на странице, сопровождаются метками. Для задания меток следует использовать элемент `label`. Содержимым этого элемента может быть произвольный текст, причем, если среди этого текста встречается элемент управления, то по умолчанию метка будет относится именно к нему. Также помечаемый элемент управления можно задать явно, воспользовавшись атрибутом `for`, значением которого должен быть идентификатор элемента управления.

Приведем пример формы:

```
<form action="register.php" method="post">
  <label class="row">
    <span class="col1">Email</span>
    <span class="col2">
      <input name="email" type="text">
    </span>
  </label>
  <label class="row">
    <span class="col1">Password</span>
    <span class="col2">
      <input name="password" type="password">
    </span>
```

```
  </label>
  <label class="row">
    <span class="col1">Confirm password</span>
    <span class="col2">
      <input name="confirm-password" type="password">
    </span>
  </label>
  <div class="row">
    <span class="col1">Send me Newsletters</span>
      <span class="col2">
        <label>
          <input name="subscribe" type="radio"
                 value="never">Never
        </label>
        <label>
          <input name="subscribe" type="radio"
                 value="weekly">Weekly
        </label>
        <label>
          <input name="subscribe" type="radio"
                 value="monthly" checked>Monthly
        </label>
      </span>
    </div>
  <input type="submit" value="Register">
</form>
```

Кроме элемента `input` для создания элементов управления можно использовать и некоторые другие элементы.

Для создания многострочного поля ввода используется элемент `textarea`. Этот элемент поддерживает следующие атрибуты:

- `name` и `maxlength` — аналогично одноименным атрибутам элемента `input`;
- `cols` — ширина поля ввода, измеренная в символах (аналогично атрибуту `size` элемента `input`);
- `rows` — высота поля ввода, измеренная в строках;
- `wrap` — будет ли при отправке данных сохранено автоматическое разбиение на строки. Возможные значения этого атрибута представлены ключевыми словам hard (сохраняется) и soft (не сохраняется). По умолчанию автоматическое разбиение на строки не сохраняется. Отметим, что ручное разбиение на строки сохраняется в любом случае.

В отличие от элемента `input` элемент `textarea` может иметь содержание. Оно используется в качестве значения по умолчанию.

Для создания списка с возможностью выбора одной или нескольких опций используется элемент select, поддерживающий следующие атрибуты:

- `name` — аналогично одноименному атрибуты элементов `input` и `textarea`;
- `size` — количество одновременно отображаемых опций. Если этот атрибут равен единице или отсутствует, то создается раскрывающийся список. В противном случае создается обычный список. Если значение атрибута `size` меньше фактического количества опций, то список отображается с вертикальной полосой прокрутки;
- `multiple` — допускается ли выбор нескольких опций. Этот атрибут является логическим; если он установлен, то вне зависимости от атрибута `size` создается обычный (а не раскрывающийся) список. Обычно для выбора нескольких элементов используется клавиша Ctrl.

Содержимым элемента `select` должны быть опции и группы опций, задаваемые соответственно элементами option и `optgroup`. Элемент option поддерживает следующие атрибуты:

- `value` — значение свойства данных формы в случае выбора данной опции;
- `label` — метка, представляющая данную опцию на экране. Если этот атрибут отсутствует, то используется содержание элемента;
- `selected` — является ли данная опция выбранной по умолчанию. Этот атрибут является логическим.

Элемент `optgroup` поддерживает атрибут `label`, аналогичный одноименному атрибуту элемента `option`. Содержимым элемента `optgroup` должны быть только элементы option, т. е. вложенные группы не поддерживаются.

Для создания кнопки (помимо элемента `input` с атрибутом `type`, равным `button`) можно использовать элемент `button`, отличие которого заключается в том, что надпись, отображаемая на кнопке, задается содержимым элемента и может не совпадать со значением,

задаваемым атрибутом `value`. Кнопка поддерживает атрибут `type`, которые может принимать значения `submit`, `reset` и `button`.

Приведем пример:

```
<form action="add-comment.php" method="post">
  <label class="row">
    <span class="col1">Text</span>
    <span class="col2">
      <textarea name="text"></textarea>
    </span>
  </label>
  <label class="row">
    <span class="col1">Rating</span>
    <span class="col2">
      <select name="rating">
        <option value="excellent" selected>отлично</option>
        <option value="good">хорошо</option>
        <option value="fair">посредственно</option>
        <option value="poor">плохо</option>
      </select>
    </span>
  </label>
  <button type="submit">Save</button>
</form>
```

1.11. Фреймы

Фрейм — это блок в составе одной страницы, в котором отображается содержание другой страницы. Фрейм задается элементом `iframe`, у которого могут быть следующие атрибуты:

- `src` — адрес страницы, отображаемой во фрейме;
- `name` — имя фрейма;
- `width` и `height` — размеры фрейма с пикселях.

Пример:

```
<iframe width="560" height="315"
  src="http://www.youtube.com/embed/Q39NNcc81P4"></iframe>
```

Страница, отображаемая во фрейме, сама может содержать фреймы. Совокупность окна браузера и всех фреймов называют множеством контекстов; иными слова контекст — это то, где может отображаться страница.

По умолчанию при переходе по ссылке или при отправке формы загружаемая страница помещается в тот контекст, в котором

находилась соответствующая ссылка или форма (такой контекст называется текущим). Чтобы поместить загружаемую страницу в другой контекст для элементов `a` и `form` задается атрибут `target`, значением которого должно быть имя фрейма либо одно из следующих ключевых слов:

- `_self` — текущий контекст (это значение по умолчанию).
- `_parent` — контекст, являющийся родительским по отношению к текущему;
- `_top` — контекст верхнего уровня (т. е. окно браузера);
- `_blank` — новый контекст. В этом случае браузер открывает новое окно.

Контрольные вопросы

1. Что такое элемент? В чем разница между элементом и тэгом?
2. Что такое атрибут элемента?
3. Какова структура веб-страницы?
4. Что может находиться в заголовке (head) веб-страницы?
5. Что может находиться в теле (body) веб-страницы?
6. Что такое комментарий и для чего он используется?
7. Что такое сущность? Какие виды сущностей поддерживаются?
8. В чем разница между блочным и строчными элементами?
9. Какие виды списков поддерживаются в HTML?
10. Что такое гиперссылка?
11. Как вставить графическое изображение? Какие форматы графических изображений поддерживаются современными браузерами?
12. Как вставить таблицу? Какие структурные элементы таблицы поддерживаются в HTML?
13. Для чего используются формы?
14. Какие элементы управления поддерживаются в HTML?
15. Для чего используются фреймы?

Глава 2. Стили

Для настройки внешнего вида Web-страниц принято использовать таблицы стилей. Как правило, таблица стилей размещается в отдельном файле, имеющем расширение css. Так же, как и файлы Web-страниц, файлы таблиц стилей являются текстовыми. Применение таблиц стилей обеспечивает следующие преимущества:

- во-первых, отделение представления (таблицы стилей) от содержания (web-страницы) позволяет сделать исходный код Web-страницы более читабельным, а также облегчает разделение обязанностей между теми, кто отвечает за информационное наполнение, и дизайнерами;
- во-вторых, упрощается поддержание нескольких страниц в единообразном виде, поскольку в случае необходимости изменить представление достаточно внести изменение в одном месте — в таблице стилей;
- в-третьих, несмотря на увеличение общего количества файлов суммарный размер данных уменьшается, поскольку исключается дублирование в задании атрибутов представления у одинаковых элементов.

2.1. Структура таблицы стилей

Таблица стилей представляет собой набор правил. Каждое правило, в свою очередь, состоит из набора селекторов и набора определений свойств, заключенных в фигурные скобки. Селекторы указывают, на какие элементы Web-страницы действует соответствующее правило; если у правила несколько селекторов, то они записываются через запятую. Каждое определение свойства состоит из имени свойства и значения, разделяемых двоеточием. Друг от друга определения свойств отделяются точкой с запятой; также допускается точка с запятой после последнего определения свойства в составе правила (т. е. непосредственно перед закрывающейся фигурной скобкой).

В тексте таблицы стилей допускаются комментарии; каждый комментарий должен начинаться наклонной чертой и звездочкой (/*), а заканчиваться, наоборот, звездочкой и наклонной чертой (*/).

Комментарий может занимать несколько строк. Вложенные комментарии не допускаются. Пример:

```
body { /* селектор: все элементы страницы,
          отображаемые в окне браузера */
    background-color: #ddd; /* свойство: светло-серый
                               цвет фона */
}
h1 { /* селектор: заголовки первого уровня */
    color: #050; /* свойство: темно-зеленый цвет текста */
    text-align: center; /* свойство: выравнивание
                           по центру */
}
p { /* селектор: абзацы обычного текста */
    text-indent: 15mm; /* свойство: абзацный отступ,
                          равный 15 миллиметрам */
}
```

Стили могут применяться только к элементам. Если необходимо применять стиль к произвольному фрагменту, то такой фрагмент можно заключить в элемент `div` или `span`. Элемент `div` является блоковым (подобно `p` или `h1`), а элемент `span` — строковым (подобно `em` или `q`). В отличие от «смысловых» элементов (`p`, `h1`, `em`, `q` и т. п.), элементы `div` и `span` не имеют форматирования по умолчанию. Например:

```
<p>Опубликовано<span id="publication-date">
  18.09.2011</span>.</p>
```

2.2. Подключение таблицы стилей

Таблица стилей, находящаяся в отдельном файле, называется внешней. Для ее подключения используется элемент `link`, который должен располагаться в заголовке Web-страницы (т. е. в элементе `head`). Расположение таблицы стилей задается атрибутом `href`, значением которого должен быть адрес таблицы стилей относительно текущего документа. Поскольку элемент `link` может использоваться для установки различных связей между ресурсами, при подключении таблицы стилей необходимо указать атрибут `rel` со значением, представленным ключевым словом `stylesheet`. Пример:

```
<link rel="stylesheet" href="styles.css">
```

Помимо внешних таблиц стилей возможны внутренние, располагающиеся непосредственно в коде самой Web-страницы. Для определения внутренней таблицы стилей используется элемент `style`, который, как и элемент `link`, должен располагаться в заголовке Web-страницы. Содержимым элемента `style` должны быть правила таблицы стилей. Также у этого элемента должен быть атрибут `type`, задающий тип таблицы стилей; для CSS значением этого атрибута должна быть строка `text/css`. Пример:

```
<style type="text/css">
    <!--
    table {
        border: 2pt solid black;
        border-collapse: collapse;
        margin: 0 auto;
        vertical-align: center;
    }
    td {
        border: 1pt solid black;
        padding: 3pt;
    }
    -->
</style>
```

Наконец, значения одного или нескольких свойств можно определить непосредственно в самом элементе. Для этого соответствующий элемент снабжается атрибутом `style`. Пример:

```
Цена: <span style="color: red; text-decoration: line-
through">100</span> 60 р.
```

Определения, заданные атрибутом `style`, имеют приоритет перед теми, которые заданы во внешних и внутренних таблицах стилей; вместе с этим отметим, что при создании Web-страниц следует экономно пользоваться атрибутом `style`, предпочитая ему внешние или внутренние таблицы стилей.

2.3. Виды селекторов

В простейшем случае селектор представляет собой имя элемента. Чтобы правило таблицы стилей действовало не на все элементы, а лишь на некоторые из них, используют классы. Класс — это произвольный идентификатор (метка) позволяющий отличить

одни элементы от других. Класс элемента задается с помощью атрибута `class`, например:

```
<p class="summary">This is a summary</p>
```

В таблице стилей класс указывается в селекторе после точки. Пример:

```
p.summary { /* абзацы со стилем summary */
    border: 3pt solid;
}
```

В атрибуте `class` можно перечислить несколько классов, разделяя их пробелом. Также несколько классов можно перечислить в селекторе правила (точка ставится перед каждым из них). Правило воздействует только на такие элементы, которые имеют все указанные в правиле классы и, возможно, какие-то еще не указанные. Порядок перечисления классов в атрибуте `class` и в селекторе правила не важен.

Вместо имени элемента в селекторе можно указать звездочку (*), означающую, что действие правила не будет ограничиваться типом элемента. Если кроме звездочки селектор включает в себя еще что-нибудь (например, указание класса), то саму звездочку можно опустить. Пример:

```
.summary { /* любые элементы со стилем summary */
    border: 3pt solid;
}
```

Различать элементы можно не только по классу, но и по идентификатору. В селекторе перед идентификатором ставится решетка. Пример:

```
#copyright { /* некоторый элемент
                с идентификатором copyright */
    font-style: italic;
}
```

Поскольку идентификаторы элементов уникальны, может создаться впечатление, что селекторы, содержащие вместе с идентификатором еще что-то, не имеют смысла. Однако следует учитывать следующее: во-первых, одна и та же внешняя таблица стилей может использоваться несколькими страницами, в которых могут быть элементы с одинаковыми идентификаторами; во-вторых,

классы, а также другие атрибуты элемента могут изменяться сценарием, что создает необходимость определения разных значений свойств для разных наборов значения атрибутов одного и того же элемента.

Кроме классов и идентификаторов допускается различение элементов по наличию и значению произвольных атрибутов. Если необходимо отобрать элементы, для которых указан некоторый атрибут, то в селекторе этот атрибут записывается в квадратных скобках. Можно также потребовать, чтобы атрибут не только присутствовал, но и имел определенное значение; для этого в квадратных скобках после имени атрибута ставится знак равенство (=), за которым записывают соответствующее значение, заключая последнее в одинарные или двойные кавычки в случае необходимости. Наконец, если значение атрибута представляет собой список слов, можно потребовать, чтобы этот список содержал некоторое слово или несколько слов; для этого перед знаком равенства ставится тильда (~). Примеры:

```
input[type=password] { /* поля для ввода пароля */
    background-color: yellow;
}
```

Если некоторый элемент может находиться в нескольких состояниях, то для их различения надо использовать псевдоклассы, записываемые в селекторе после двоеточия. Поддерживаются следующие псевдоклассы: `visited` — посещенная ссылка, `link` — непосещенная ссылка, `active` — ссылка в тот момент, когда пользователь щелкает ее, `focus` — элемент, на который в данный момент установлен фокус ввода с клавиатуры, и `hover` — элемент, над которым в данный момент находится указатель мыши. Пример:

```
a:link { /* ссылки, которые еще не были посещены */
    color: green;
}
a:visited { /* посещенные ссылки */
    color: gray;
}
```

Селектор можно использовать для выбора элементов не только на основе их собственного состояния и собственных атрибутов, но и на основе соседних элементов. Для этого используются со-

ставные селекторы. Составной селектор представляет собой два или более простых (рассмотренных выше), соединенных пробелом, знаком больше (>) либо знаком сложения (+). Если два селектора разделены пробелом, то такое правило будет применено к элементу, только если он сам соответствует второму селектору и при этом у него есть предок (родительский элемент или предок родительского), который соответствует первому селектору. Если же вместо ограничения на произвольный предок требуется наложить ограничение непосредственно на родительский элемент, то селекторы разделяют знаком больше. Наконец, знак плюс позволяет наложить ограничение на элемент, непосредственно предшествующий данному. Примеры:

```
ul.menu li { /* элементы неупорядоченных списков,
                имеющих стиль menu */
    display: inline-block;
}
h1 + p { /* первые абзацы после заголовков
            первого уровня */
    text-indent: 0;
}
```

Вполне может получиться так, что на один и тот же элемент воздействует несколько правил. При этом может оказаться так, что одно или несколько свойств по-разному определяются более чем в одном воздействующем на данный элемент правиле. Такая неоднозначность разрешается тем, что предпочтение отдается правилу с наиболее специфичным селектором. Из двух селекторов более специфичный выбирается следующим образом:

- более специфичен тот, в котором указано больше идентификаторов;
- если количество идентификаторов одинаково, то более специфичен тот, в котором указано большее количество классов;
- если количества идентификаторов и классов одинаковы, то более специфичен тот, в котором указано большее количество элементов (количество элементов может оказаться больше единицы, если селектор сложный).

Если устранить неоднозначность на основе специфичности селекторов не удается, то предпочтение отдается последнему (в порядке встречаемости в коде страницы) правилу.

2.4. Задание размеров

При задании внешнего вида страницы часто возникает задача определения линейных размеров (высоты элемента, толщины границы и т. д.). В CSS одним из способов преставления значения свойства, задающего размер, является вещественное число в десятичной системе счисления с абсолютной или относительной единицей измерения длины. К абсолютным относятся: `cm` — сантиметры, `mm` — миллиметры, `in` — дюймы (в CSS принято, что 1 дюйм равен 2,54 сантиметрам), `pt` — пункты (в 1 дюйме 72 сантиметра), `pc` — цицеро (1 цицеро равно 12 пунктам) и `px` — пиксели (в CSS принято, что 1 пиксель равен ¾ пункта). По умолчанию в браузерах принимается, что один пиксель соответствует одной точке экрана, однако это соотношение может изменяться, если пользователь задает отличный от 100% масштаб отображения страницы. Относительные единицы позволяют задать длину на основе текущего шрифта; к относительным единицам относятся: `em` — текущий размер шрифта и `ex` — высота строчной буквы x. Между числовым значением и единицей измерения пробел не допускается. Указание единицы измерения обязательно; исключением является только 0 — после него единицу измерения можно опустить.

Наряду с заданием размера путем выражения его в какой-либо единице измерения часто допускается использование другого способа: выражение данного размера в процентах от какого-либо другого размера; в этом случае вместо единицы измерения используется знак процента.

2.5. Параметры шрифта и абзаца

Для задания наименования шрифта используется свойство `font-family`. Поскольку при создании страницы нельзя быть уверенным в наличии необходимого шрифта на том компьютере, на котором она будет просматриваться, это свойство допускает указание не одного, а произвольного количества шрифтов через запятую в порядке уменьшения приоритета (т. е. браузер выбирает первый из имеющихся на компьютере шрифтов). Кроме конкретных шрифтов в свойстве `font-family` можно указать обобщенные шрифты, задаваемые следующими ключевыми словами: `serif` — с засечками (например, Times New Roman), `sans-serif` — без засечек (на-

пример, Arial), `cursive` — имитирующий рукописный, `fantasy` — декоративный, `monospace` — моноширинный (например, Courier New). Обобщенные шрифты обычно указывают после конкретных. Пример:

```
font-family: Verdana, Helvetica, sans-serif;
/* шрифт Verdana, если он доступен, в противном случае
   шрифт Helvetice, если он доступен, в противном случае
   любой доступный шрифт без засечек */
```

Для задания размера шрифта используется свойство `font-size`, значение которого может быть выражено с использованием некоторой единицы измерения, в процентах от текущего размера шрифта либо с использованием следующих ключевых слов: medium — средний, `large`, `x-large` и `xx-large` — крупнее среднего, `small`, `x-small` и `xx-small` — мельче среднего, `larger` — крупнее текущего и `smaller` — мельче текущего.

Наряду с размером шрифта можно явно настроить высоту строки; для этого используется свойство `line-height`, значение которого может быть представлено ключевым словом `normal` либо выражено с использованием единиц измерения длины, в процентах от размера шрифта либо в виде числа без единицы измерения длины, интерпретируемого как множитель размера шрифта. Пример:

```
font-size: 12pt;  /* шрифт размером 12 пт. */
line-height: 1.5; /* через полуторный интервал */
```

Для задания начертания шрифта используются свойства:

- `font-weight` — задает жирность шрифта. Возможные значения: `normal` (нормальный), `bold` (полужирный), `bolder` (более жирный, чем текущий) и `lighter` (менее жирный, чем текущий). Кроме ключевых слов можно использовать целые числа от 100 (наименее жирный) до 900 (наиболее жирный) с шагом 100. Следует отметить, что большинство шрифтов не поддерживает все девять уровней жирности. В случае отсутствия заданного уровня браузер использует ближайший из поддерживаемых;
- `font-style` — позволяет выбрать курсивное начертание. Возможные значения: `normal` (нормальное начертание), `italic` (курсивное начертание) и `oblique` (наклонное начертание). Наклонное начертание отличается от курсивного тем, что первое

обычно получается механически путем трансформации нормального начертания;

• `font-variant` — позволяет выбрать начертание капителью (т. е. уменьшенными заглавными). Возможные значения: `normal` (нормальное начертание) и `small-caps` (начертание капителью).

Для одновременного задания нескольких параметров шрифта можно использовать псевдосвойство `font`. В значении этого свойства последовательно через пробел перечисляются начертание, размер шрифта, высота строки и, наконец, название шрифта, причем указание размера и названия шрифта является обязательным; перед высотой строки (если она указывается, ставится наклонная черта). Пример:

```
font: italic 12pt / 1.5 "Times New Roman", serif;
/* курсивный шрифт Times New Roman (или любой с засечками)
   размером 12 пт. через полутоный интервал */
```

Кроме перечисленных к параметрам шрифта можно отнести наличие подчеркивания, перечеркивания и тому подобных эффектов, называемых декорированием. Декорирование задается свойством `text-decoration`, значением которого может быть ключевое слово `none`, означающее отсутствие какого-либо декорирования, либо список следующих значений, разделенных пробелом: `underline` (подчеркивание), `overline` (надчеркивание), `line-through` (перечеркивание) и `blink` (мерцание). Отметим, что применение мерцания не рекомендуется.

Помимо декорирования для выделения текста можно использовать разрядку; для этого используется свойство `letter-spacing`, задающее межсимвольный интервал. Значение этого свойства может быть представлено ключевым словом `normal`, соответствующим межсимвольному интервалу по умолчанию, либо выражено в некоторых единицах измерения длины. Кроме межсимвольного интервала можно настроить интервал между словами; для этого используется свойство `word-spacing`, значения которого задаются аналогично.

Для настройки горизонтального выравнивания текста используется свойство `text-align`, возможными значениями которого являются: `left` (по левому краю), `right` (по правому краю), `center`

(по центру) и justify (по ширине). При использовании выравнивания по ширине следует учитывать, что пока браузеры не умеют переносить текст по слогам, что может привести к существенному ухудшению внешнего вида абзаца. Кроме выравнивания можно задать абзацный отступ; для этого используется свойство text-indent, значение которого может быть выражено с использованием единиц измерения длины либо в процентах от ширины элемента. Примеры:

```
text-align: justify; /* выравнивание по ширине */
text-indent: 1cm;    /* абзацный отступ размером в 1 см. */
```

2.6. Цвет и фоновое изображение

Для задания цвета текста и цвета фона используются свойства color и background-color соответственно. Значения этих свойств могут задаваться следующими способами:

• с помощью ключевых слов aqua (голубой), black (черный), blue (синий), fuchsia (фуксия), gray (серый), green (зеленый), lime (лайм), maroon (бордовый), navy (темно-синий), olive (оливковый), orange (оранжевый), purple (пурпурный), red (красный), silver (серебряный), teal (сине-зеленый), white (белый), и yellow (желтый). Отметим, что помимо перечисленных некоторые браузеры поддерживают и другие ключевые слова для задания цвета;

• на основе цветовой модели RGB путем задания трех цветовых компонентов. Числовые значения компонентов могут быть заданы в десятичной нотации в целыми числами от 0 до 255 либо в процентах от 0% до 100% либо в шестнадцатеричной нотации целыми числами от 0 до F либо 00 до FF. При использовании десятичной нотации компоненты представляются как аргументы функции rgb (т. е. пишутся после ключевого слова rgb будучи взятыми в круглые скобки и разделенными запятыми). При использовании шестнадцатеричной нотации компоненты пишутся без разделения в виде трех либо шести шестнадцатеричных цифр, которым предшествует решетка (#); если используются три цифры, то считается, что каждая из них задает как старший, так и младший полубайт.

Примеры:

```
background-color: #77bbff; /* светло-голубой */
background-color: #7bf;    /* то же самое */
```

В качестве фона можно использовать графическое изображение — для этого используется свойство `background-image`. Графическое изображение задается своим адресом, который записывается в виде аргумента функции `url`, причем сам адрес может быть заключен в одинарные или двойные кавычки. Так же, как в HTML, адрес может быть как абсолютным, так и относительным. Пример:

```
background-image: url("img/sky.jpg"); /* изображение,
    находящееся в файле sky.jpg в подкаталоге img */
```

Если размер фонового изображения меньше размера элемента, то оно может быть повторено вдоль горизонтальной или вертикальной оси либо вдоль обоих осей. Для этого используется свойство `background-repeat`, возможные значения которого представлены следующими ключевыми словами: `repeat` — повтор вдоль обеих осей, `repeat-x` — повтор только вдоль горизонтальной оси, `repeat-y` — повтор только вдоль вертикальной оси и `no-repeat` — без повтора.

Отметим, что если размер фонового изображения меньше размера элемента и повторение не используется, то оставшиеся области заполняются цветом фона. Кроме этого, цвет фона используется в процессе загрузки фонового изображения, а также если последнее загрузить не удается; по этой причине рекомендуется, чтобы цвет фона соответствовал преобладающему цвету в фоновом изображении.

Для задания выравнивания фонового изображения относительно элемента используется свойство `background-position`, у которого может быть одно или два значения, разделенных пробелом; эти значения определяют положение изображения вдоль вертикальной и горизонтальной осей. Каждое значение может быть представлено следующими способами:

- в виде смещения верхнего (левого) края изображения относительно верхнего (левого) края элемента вниз (вправо), выраженного с использованием некоторой единицы измерения длины;
- в процентах, причем 0% соответствует выравниванию по верхнему (левому) краю, 100% — выравниванию по нижнему (правому) краю, а 50% — центрированию.

• с использованием следующих ключевых слов: `left` — выравнивание по левому краю, `top` — выравнивание по верхнему краю, `center` — центрирование, `right` — выравнивание по правому краю и `bottom` — выравнивание по нижнему краю.

Пример:

```
background-position: top center; /* вдоль верхнего края
                                    по центру */
background-repeat: no-repeat;    /* без повторения */
```

Свойство `background-position` можно использовать для оптимизации загрузи изображений: дело в том, что несколько мелких изображений загружаются существенно медленней, нежели одно большее, содержащее такое же количество пикселей. Оптимизация состоит в том, что несколько изображений собирают в одно, а для выбора нужной части составного изображения используют свойство `background-position`. Пример:

```
/* Предположим, что имеется три блока с ссылками,
   расположенные один по другим, каждый размером
   80 × 30 пикселей.
   Используется одно изображение размером
   160 × 120 пикселей, состоящее из шести частей
   (2 столбца × 3 строки).
   Строки соответствуют блокам, а столбцы состояниям. */
ul.links li {
    display: block;
    width: 80px;
    height: 30px;
    background-image: url('img/links.png')
}
/* первый блок имеет идентификатор home-link */
#home-link {
    background-position: 0 0;
}
#home-link:hover {
    background-position: -80px 0;
}
/* второй блок имеет идентификатор products-link */
#products-link {
    background-position: 0 -30px;
}
#products-link:hover {
    background-position: -80px -30px;
```

```
}
/* третий блок имеет идентификатор support-link */
#support-link {
    background-position: 0 -60px;
}
#support-link:hover {
    background-position: -80px -60px;
}
```

По умолчанию фоновое изображение прокручивается вместе с самим элементом. Однако это поведение можно изменить, воспользовавшись свойством `background-attachment`, возможные значения которого представлены ключевыми следующими словами: `fixed` — фоновое изображение зафиксировано, `scroll` — фоновое изображение прокручивается.

Для одновременного задания всех параметров фона можно использовать псевдосвойство `background`, значениями которого могут быть цвет и параметры фонового изображения. Пример:

```
background: #5592e4 url(img/sky.jpg) no-repeat
```

2.7. Границы и поля

У всех элементов, включая строчные, могут быть границы. Каждая граница может характеризоваться следующими параметрами, причем каждый из них можно задать независимо для верхнего (top), правого (right), нижнего (bottom) и левого (left) края:

- толщина — задается свойствами `border-top-width`, `border-right-width`, `border-bottom-width` и `border-left-width`. Значение может быть выражено в одной из единиц измерения длины либо представлено одним из следующих ключевых слов: `thin` — тонкая, `medium` — средняя и `thick` — толстая;
- стиль — задается свойствами `border-top-style`, `border-right-style`, `border-bottom-style` и `border-left-style`. Значение должно быть одним из следующих ключевых слов: `none` — отсутствует, `hidden` — невидимая, `dotted` — точки, `dashed` — штрихи, `solid` — сплошная одинарная, `double` — сплошная двойная, `grove`, `ridge`, `inset` и `outset` — задают псевдотрехмерный эффект;
- цвет — задается свойством `border-top-color`, `border-right-color`, `border-bottom-color` и `border-left-color`. Исполь-

зуется то же представление цвета, что и в свойствах `color` и `background-color`.

Пример:

```
border-bottom-style: dotted; /* стиль нижней границы — точки */
border-bottom-color: red;    /* цвет нижней границы — красный */
```

Вместо задания параметров для каждой из границ в отдельности можно задать их все сразу. Для этого можно воспользоваться псевдосвойствами `border-width`, `border-solid` и `border-color`. Для каждого из этих свойств можно задавать до четырех значений, причем действуют следующие правила:

- если указано одно значение, то оно применяется ко всем границам;
- если указано два значения, то первое применяется к верхней и нижней границе, а второе — к правой и левой;
- если указано три значения, то первое применяется к верхней границе, второе — к правой и левой, а третье — к нижней;
- если указано четыре значения, то они применяются начиная с верхней границе по часовой стрелке, т. е. первое — к верхней границе, второе — к правой, третье — к нижней, а четвертое — к левой.

Примеры:

```
border-width: 5pt;         /* толщина всех границ 5 пт */
border-style: solid none; /* верняя и нижняя граница
                             сплошная, а правая и левая
                             отсутствуют */
```

Кроме задания индивидуальных свойств границ одновременно для всех сторон можно также задавать одновременно задавать все свойства (толщину, стиль и цвет) как индивидуально для каждой стороны, так и для всех сторон сразу. Для этого используются псевдосвойства `border-top`, `border-right`, `border-bottom`, `border-left` и `border`. У всех этих псевдосвойств может быть до трех значений, задающих толщину, стиль и цвет; подобно псевдосвойству `background` значения разделяются пробелами и могут быть перечислены в произвольном порядке. Пример:

```
border: 1pt solid #777; /* сплошная серая граница
                           толщиной в 1 пт */
```

У каждого элемента могут быть два вида полей: внешние (margins), размещающиеся снаружи границы, и внутренние (padding), размещающиеся между границей и содержанием. Внешние поля задаются свойствами `margin-top`, `margin-right`, `margin-bottom` и `margin-left`, а внутренние — свойствами `padding-top`, `padding-right`, `padding-bottom` и `padding-left`. Значения всех этих свойств могут быть заданы в виде числа с единицей измерения длины либо в виде числа со знаком процента; последнее определяет размер поля через ширину включающего элемента. Помимо некоторого числа значение внешнего поля может быть задано ключевым словом `auto`; этот вариант обычно используют для центрирования.

Подобно свойствам, задающим параметры границы, свойства, задающие поля также могут быть заданы одновременно для всех сторон элемента. Для этого используются псевдосвойства `margin` и `padding`, для каждого из которых можно указать от одного до четырех значений. Пример:

```
margin: 0 auto; /* центрировать элемент по горизонтали */
```

2.8. Размеры и положение

Таблицы стилей можно использовать для явного задания размеров элементов, а также и окружающих их рамок и отступов. Для задания ширины и высоты элемента используются свойства `width` и `height`. Значения этих свойств могут быть выражены в единицах измерения длины или в процентах от соответствующего размера включающего элемента либо представлены ключевым словом `auto` (в этом случае размер элемента определяется содержимым). Необходимо заметить, что свойства `width` и `height` задают размер элемента без учета полей и границ; если у элемента есть поля и/или границы, то его действительный размер будет больше того, который задан свойствами `width` и `height`. Пример:

```
width: 50%;    /* ширина составляет половину
                  от ширины включающего элемента */
height: 100px; /* высота равна 100 пикселям */
```

Отметим, что для строчных элементов настройка высоты и ширины недоступна.

Положение элемента задается свойствами `position`, `left`, `right`, `top` и `bottom`. Значения свойств `left`, `right`, `top` и `bottom` могут быть представлены числом с единицей измерения длины либо знаком процента либо ключевым словом `auto`, соответствующим положению по умолчанию. Интерпретация значений этих свойств зависит от используемого браузером алгоритма определения положения, задаваемым значением свойства `position`, которым может быть одно из следующих ключевых слов:

- `static` — положение элемента на странице полностью определяется порядком следования элементов в исходном коде страницы, а размер элемента учитывается при размещении следующих за ним фрагментов. При использовании этого алгоритма значения свойств `left`, `right`, `top` и `bottom` игнорируются. Алгоритм `static` используется по умолчанию;
- `relative` — отличается от предыдущего тем, что позволяет смещать элемент относительного его положения по умолчанию по вертикали или горизонтали на заданное расстояние. Смещение по горизонтали задается свойством `left` или `right`, а по вертикали — свойством `top` или `bottom`. При использовании свойств `left` (`top`) смещение вправо (вниз) задается положительной величиной, а влево (вверх) — отрицательной. При использовании свойств `right` и `bottom` интерпретация положительных и отрицательных величин меняется на противоположную. При использовании алгоритма `relative`, так же, как и при использовании алгоритма `static`, размер элемента учитывается при размещении следующих за ним фрагментов страницы, однако само смещение при этом не учитывается (т. е. следующие фрагменты страницы всегда располагаются так, как будто у данного элемента никакого смещения нет);
- `absolute` — положение элемента на странице определяется явно на основе значений свойств `left`, `right`, `top` и `bottom`, задающих соответственно смещение левой, правой, верхней и нижней стороны ограничивающего прямоугольника элемента относительно соответствующих сторон включающего его блока. Включающим блоком считается ближайший предок, для расположения которого используется алгоритм, отличный от `static` (в этом еще одно раз-

личие алгоритмов `static` и `relative`). Отметим, что если не задано свойство width (`height`), но одновременно задана пара свойства `left` и `right` (`top` и `bottom`), то ширина (высота) элемента определяется на основе этой пары. Если же, наоборот, значения свойств `left` и `right` (`top` и `bottom`) одновременно отсутствуют, то в качестве положения элемента по горизонтали (по вертикали) используется положение по умолчанию. Алгоритм `absolute` отличается от алгоритмов `static` и `relative` тем, что размер элемента не учитывается при расположении следующих фрагментов страницы (т. е. следующие фрагменты страницы располагаются так, как будто данного элемента вовсе не существует);

• `fixed` — отличается от предыдущего тем, что положение элемента привязывается к окну браузера, в результате чего элемент остается на одном и том же месте вне зависимости от состояния полос прокрутки.

Пример:

```
position: fixed; /* элемент привязан к окну браузера */
right: 20px;     /* правый край элемента на расстоянии
                    20 пикселей от правого края
                    окна браузера */
bottom: 50px;    /* нижний край элемента на расстоянии
                    50 пикселей от нижнего края
                    окна браузера */
```

Свойство `position` (иногда в сочетании со свойствами `left`, `right`, `top` и `bottom`) часто используют при создании раскрывающихся меню и появляющихся подсказок.

Помимо использования свойств position, `top`, `right`, `bottom` и `left` еще одним способом, позволяющим изменить взаимное расположение элементов страницы, является использование свойств `float` и `clear`. Свойство `float` позволяет сделать элемент «плавающим» (вроде врезки в газете или журнале) и может задаваться следующими ключевыми словами:

• `left` — элемент «плавает» слева от основного текста, т. е. последний обтекает его справа и снизу. Положение самого «плавающего» элемента при этом не меняется. Если в исходном коде страницы встречается подряд несколько «плавающих» слева эле-

ментов, то они располагаются слева направо насколько хватает ширины включающего элемента;

• `right` — элемент «плавает» справа, т. е. остальной текст обтекает его слева и снизу. Сам «плавающий» элемент при этом выравнивается по правому краю включающего блока. Если в исходном коде страницы встречается подряд несколько «плавающих» справа элементов, то они располагаются справа налево насколько хватает ширины включающего элемента;

• `none` — элемент не «плавает».

Свойство `clear` позволяет запретить размещение элемента слева или справа от плавающего элемента, заставляя браузер разместить элемент ниже (если элемент и так расположен ниже плавающего, то это свойство никакого эффекта не оказывает). Значения свойства `clear` задаются следующими ключевыми словами:

• `left` — запрещает размещение элемента справа от элементов, «плавающих» слева;

• `right` — запрещает размещение элемента слева от элементов, «плавающих» справа;

• `both` — запрещает размещение элемента слева или справа от любых «плавающих» элементов (комбинация двух предыдущих вариантов);

• `none` — не накладывает никаких ограничений.

Помимо основного назначения (создания врезок) свойства `float` и `clear` часто используют для размещения содержимого страницы в виде нескольких колонок. Пример:

```
#column-1 { /* первая колонка */
    width: 30%;
    float: left; /* колонки расположены слева направо */
}
#column-2 { /* вторая колонка */
    width: 70%;
    float: left;
}
#footer { /* подвал страницы (под колонками) */
    clear: left;
}
```

Контрольные вопросы

1. Для чего применяются таблицы стилей? В чем их преимущество?
2. Какова структура таблицы стилей?
3. Каким образом можно подключить таблицы стилей к веб-странице?
4. Что такое селектор? Какие виды селекторов бывают?
5. Что называют сложным селектором? Какие способы комбинирования селекторов бывают?
6. Как можно задать расстояние в таблице стилей?
7. Как можно задать цвет в таблице стилей?
8. Какие параметры шрифта поддерживаются таблицами стилей?
9. Какие параметры абзаца поддерживаются таблицами стилей?
10. Как можно настроить фон элемента?
11. Как можно настроить поля и границы элемента?
12. Как можно настроить положение элемента?

Глава 3. PHP

3.1. Общие сведения

Языки, наиболее часто используемые в настоящее время для разработки Web-приложений, можно условно разделить на две группы:

- PHP, Python, Ruby и Perl — интерпретируемые языки с динамической типизацией, называемые иногда «скриптовыми». Некоторые из этих языков (Python и Ruby) изначально были спроектированы как универсальные объектно-ориентированные, другие же возникли как специализированные (PHP и Perl) и приобрели объектно-ориентированные возможности в ходе своего развития. Особенностью, выделяющей PHP, является то, что этот язык с самого начала предназначался именно для применения в Web;
- Java, C# и Visual Basic .NET — языки со статической типизацией, компилируемые в независимый от платформы двоичный код, выполняемый виртуальной машиной. Все эти языки также являются объектно-ориентированными.

Из перечисленных языков наиболее широко используемым безусловно является PHP. Следует, однако, признать, что несмотря на это в отношении PHP, можно услышать самые разнообразные, в том числе крайне негативные мнения. Действительно, язык и среда выполнения PHP далеко не свободны от недостатков. Из наиболее серьезных, пожалуй, можно перечислить следующие:

- наличие небезопасных конструкций, неумелое использование которых ведет к появлению кода, характеризующегося, по меньшей мере, низкой надежностью и неудобством изменения и сопровождения, а чаще, в дополнение к перечисленному, еще и содержащему серьезные уязвимости для разного рода атак;
- наличие морально устаревших конструкций, поддерживаемых для сохранения совместимости с предыдущими версиями, самим фактом своего существования препятствующих освоению начинающими разработчиками более эффективных приемов программирования;
- малая степень совместимости новых версий с предыдущими, несмотря на предыдущий пункт.

Наверное, после прочтения предыдущего абзаца у читателя невольно возникает вопрос: как же получилось так, что при всех перечисленных недостатках PHP остается самым популярным средством разработки Web-приложений? Ответ, видимо, заключается в следующем:

- низкий порог вхождения — для тех, кто никогда не программировал, начать программировать на PHP легче, чем на любом из перечисленных языков. Этот низкий порог породил множество разработчиков, популяризующих достоинства PHP (иногда — действительные, иногда — мнимые) и способствующих, таким образом, еще большему распространению PHP;
- C-подобный синтаксис — облегчает изучение PHP для тех, кто имеет опыт программирования на C, C++, Java, C# или JavaScript;
- как следствие предыдущего пункта, поддержка PHP хостинг-провайдерами — хостинги, позволяющие размещать на них Web-приложения, написанные на других языках, встречаются реже и стоят дороже. В сущности поддержка PHP столь широка, что можно утверждать, что она является практически обязательным пунктом в списке услуг, предоставляемых хостинг-провайдером;
- многочисленное сообщество разработчиков постоянно вырабатывает и совершенствует эффективные и безопасные приемы программирования, следование которым позволяет преодолеть некоторые из вышеотмеченных недостатков языка. Этому способствует также использование фреймворков, среди которых можно назвать, в частности, Zend Framework, Symfony, CodeIgniter и CakePHP;
- команда разработчиков PHP совершенствует язык, устраняя по возможности недостатки, оставшиеся в наследство от предыдущих версий, и дополняя его возможностями, имеющимися в таких современных объектно-ориентированных языках, как Java и Ruby.

Перечисленное не только объясняет нынешнюю почти беспрецедентную популярность и распространенность PHP, но и позволяет предположить, что сложившаяся ситуация не претерпит существенных изменений и в ближайшем будущем. Из этого следует, что изучение PHP является необходимым компонентом в подготовке профессионального разработчика Web-приложений.

3.2. Схема выполнения сценария

Программа, отвечающая за обработку входящих HTTP-запросов, называется Web-сервером или HTTP-сервером;[1] как правило Web-сервер выполняется в режиме демона (Unix) или сервиса (Window). На момент написания наиболее популярными Web-серверами являются Apache, Microsoft IIS (Internet Information Services), nginx и Google GWS.

В простейшем случае Web-содержание располагается в некотором каталоге на диске, называемом корневым каталогом Web-сервера,[2] и повторяет структуру соответствующего сайта. Предположим, что корневой каталог называется C:\WebRoot, тогда при получении от клиента запроса к ресурсу abc/def/xyz.html сервер вернет клиенту содержимое файла C:\WebRoot\abc\def\xyz.html. Web-сервер может использовать несколько корневых каталогов, выбирая между ними на основании данных, содержащихся в запросе; как правило выбор производится на основе начальных компонентов имени ресурса (виртуальный каталог) и/или заголовка Host (виртуальный сервер).

Для генерации динамического содержимого Web-сервер вместо того, чтобы передавать клиенту содержимое соответствующего файла в неизменном виде, выполняет этот файл как программу и передает клиенту вывод, генерируемый этой программой. Файл, генерирующий динамическое содержимое, вовсе не обязательно должен быть настоящей двоичной программой. Он может быть текстовым файлом, содержащим сценарий; в этом случае Web-сервер использует для выполнения такого файла соответствующий интерпретатор.

Наиболее распространенным и исторически первым стандартом взаимодействия Web-сервера с программой, генерирующей динамическое содержимое, является CGI — Common Gateway Interface. Согласно нему содержание выводится в стандартный поток вывода (например, в программе на языке C для этого можно использовать функции `printf` и `puts`). Отметим, что вывод должен

---

[1] Впрочем, также называется и компьютер, на котором выполняется Web-сервер.

[2] Не путать с корневым каталогом диска.

содержать не только тело сообщения, но также начальную строку и заголовки. Параметры запроса, а также информация о самом Web-сервере передаются в программу, генерирующую содержание, через переменные окружения (в программах на языке C для получения значений переменных окружения используется функция `getenv`). Определены следующие переменные окружения:

- `REQUEST_METHOD` — HTTP-метод, например GET или POST;
- `QUERY_STRING` — строка запроса, т. е. часть имени ресурса после первого знака вопроса. Например, при запросе www.myserver.ru/my.cgi/info?page=13&sort=1 эта переменная будет содержать page=13&sort=1;
- `SCRIPT_NAME` — часть пути, которая была использована Web-сервером для нахождения программы. Например, если предположить, что имя программы — my.cgi, то для вышеприведенного запроса эта переменная будет содержать /my.cgi;
- `PATH_INFO` — оставшаяся часть пути. Например, для вышеприведенного запроса с учетом сделанного предположения эта переменная будет содержать /info;
- `PATH_TRANSLATED` — `PATH_INFO` преобразованная в имя файла;
- `REMOTE_ADDR` — сетевой адрес клиента;
- `SERVER_NAME` — доменное имя сервера;
- `SERVER_PORT` — порт сервера;
- `HTTP_*` — заголовок запроса. Например, заголовок `Accept-Language` будет представлен переменной `HTTP_ACCEPT_LANGUAGE`.

В том случае, если запрос содержит тело, то считать его программа может, используя стандартный входной поток (например, в программе на языке для этого можно использовать функции `gets` и `scanf`).

Недостатком CGI является то, что новый процесс запускается Web-сервером для выполнения каждого нового запроса (этот недостаток особенно чувствителен на Windows-платформе). Улучшением CGI является FastCGI; в этом случае один процесс используется многократно, а обмен данными осуществляется с использованием сокетов.

3.3. Структура исходного файла

При анализе исходного файла интерпретатор PHP разбирает в соответствии с синтаксисом языка PHP только текст, заключенный между открывающим и закрывающим тегами PHP, имеющими вид **<?php** и **?>**, весь остальной текст интерпретатор воспринимает так, как если бы на его месте имелся бы оператор вывода. Рассмотрим пример:

```
<!DOCTYPE html>
<html>
    <head><title>Hello</title></head>
    <body><p><?php echo 'Hello World' ?></p></body>
</html>
```

В результате выполнения этого файла интерпретатор PHP выведет следующий текст:

```
<!DOCTYPE html>
<html>
    <head><title>Hello</title></head>
    <body><p>Hello World</p></body>
</html>
```

3.4. Выражения и операторы

Для вывода в языке PHP используются операторы **echo** и **print**, являющиеся синонимами.

Поддерживаются следующие примитивные типы данных:

- числовые (целый и вещественный);
- логический — представлен литералами **TRUE** и **FALSE**;
- строковый — литералы заключаются в одинарные или двойные кавычки;
- пустой — представлен литералом **NULL**.

Кроме примитивных типов язык PHP поддерживает ресурсы (никак не интерпретируемые значения, возвращаемые многими встроенными функциями), а также массивы и объекты (о них будет сказано далее).

В отличие от значений переменные не имеют типа, т. е. одна и та же переменная может хранить значения любого типа. Также в языке PHP нет предварительного описания переменных, т. е. переменная создается при первом присваивании. Перед именем пере-

менной всегда ставится знак доллара. Операторы присваивания в PHP используют привычный для С-подобных языков синтаксис. Пример:

```
$x = '10';
$x += 5; // 15
```

Язык PHP позволяет удалять переменные и проверять их существование; для этого используются соответственно ключевые слова **unset** и **isset**, записываемые как функции. Отметим, что при обращении к несуществующей переменной генерируется предупреждение notice и возвращается значение **null**.

Арифметические и логические операторы в языке PHP записываются с использованием того же синтаксиса и используют тот же приоритет и ассоциативность, что и в других С-подобных языках. При необходимости PHP автоматически преобразует тип данных. Например:

```
'10' + '5' // 15
27.5 % '7' // 6
```

При преобразовании в строковый тип **false** и **null** дают пустую строку, а **true** — '1'. При преобразовании в логический тип **null**, 0, 0.0, пустая строка и '0' дают **false**, остальные — **true**. При необходимости можно использовать явное преобразование типа. Например:

```
(string) 123  // '123'
(float) '123' // 123.0
(int) 5.6     // 5
(bool) '0'    // false
```

Ключевое слово **empty**, записанное как функция, возвращает **true**, если переменная существует и содержит истинное значение.

Для логических операторов в языке PHP предусмотрен альтернативный синтаксис с использованием ключевых слов **and**, **or**, **xor** и **not**; при использовании альтернативного синтаксиса логические операторы обладают наименьшим приоритетом. Например:

```
$fh = @fopen('nosuchfile.txt', 'r')
        or trigger_error('No such file');
```

В дополнение к операторам, традиционным для С-подобных языков, язык PHP предусматривает оператор сцепления строк, например:

```
$x = '10' . 0; // '100'
$x .= 100;     // '100100'
```

Отметим, что в отличие от языков Java и JavaScript оператор сложения не приводит к сцеплению строк, но преобразовывает строки в числа и складывает их как числа.

В языке PHP поддерживается два вида сравнений: нестрогое, сравнивающее значения без учета типа данных, обозначаемое традиционными операторами == и !=, и строгое, учитывающее тип данных и обозначаемое операторами === и !==. Пример:

```
100 == '100'  // true
100 === '100' // false
```

Нестрогое сравнение не является транзитивным и, вообще, ведет себя весьма причудливым образом:

```
'' == 0       // true
0 == '0'      // true
'' == '0'     // false
'12' == '12x' // false
12 == '12x'   // true
```

Не лучше ситуация с другими операторами сравнения, при том, что строгой версии у них нет:

```
'11' < '2'  // false
'11' < '2x' // true
```

Для того, чтобы избежать неоднозначности в операторах сравнения можно использовать операторы явного преобразования типа.

Язык PHP поддерживает интерполяцию строк, т. е. возможность включения значений переменных в строковые литералы. Интерполируемые литералы, в отличие от обычных, записываются в двойных кавычках. Например:

```
$x = 123;
$s = "The value of x is $x"; // 'The value of x is 123'
```

Помимо переменных, интерполируемые строки могут включать некоторые выражения, а именно, обращения к элементам массивов и полям объектов, а также вызов методов объектов.

Операторы ветвления и цикла в языке PHP могут записываться с использованием синтаксиса, принятого в других C-подобных языках. Однако в дополнение к этому в языке PHP доступен альтернативный синтаксис, например:

```
if (4 == date('w')):
    echo 'Sunday!';
endif;
```

Поскольку текст, находящийся вне тегов PHP, в сущности является оператором вывода, возможны следующие конструкции:

```
<?php if (in_array(date('w'), [0, 6])): ?>
    <p>Today is a weekend!</p>
<?php endif; ?>
```

В языке PHP имеется (очень) большое количество встроенных функций. В частности:

- математические: `sin`, `tan`, `exp` и т. д.;
- проверки типов: `is_int`, `is_float` и т. д.;
- `is_numeric` — истина, если аргумент является числом либо строкой, представляющей собой число;
- строковые: `strtolower` и `strtoupper` — преобразование всей строки к соответствующему регистру, `ucfirst` и `lcfirst` — преобразование первого символа строки к соответствующему регистру, `strlen` — длина строки, `trim`, `ltrim` и `rtrim` — отсечение пробелов, `strcmp` — двоичное сравнение, `strcoll` — сравнение с учетом региональных настроек, `strnatcmp` и `strnatcasecmp` — «естественное» сравнение (как при сортировке файлов в Explorer), `ord` — ASCII-код первого символа, `chr` — символ по ASCII-коду, `htmlspecialchars` и `htmlentities` — экранирует символы, имеющие специальное значение в HTML (второй вариант более агрессивным), `strpos`, `stripos`, `strrpos`, `strripos` — поиск первого или последнего вхождения второй строки в первую с учетом или без учета регистра (в отличие от языка C при отсутствии вхождения возвращает false, а не -1), `md5` и `sha1` — хеш-функция на основе соответствующего алгоритма.

Кроме переменных PHP позволяет определять константы; для этого используется функция `define`. В отличие от переменных константы записываются без знака доллара. Например:

```
define('GREETING', 'Hello');
echo GREETING;
```

При попытке повторного определения константы ее значение не меняется, но выдается предупреждение (впрочем, иногда нет и предупреждения). Также с константами можно использовать функции `defined` и `constant`.

3.5. Массивы

Особенность массивов в PHP состоит в том что они являются ассоциативными, т. е. в отличие массивов, представленных в большинстве других языков программирования, массивы PHP представляют собой не коллекцию элементов, последовательно индексированных целыми числами, а коллекцию ключей и связанных с ними значений. В пределах массива ключи уникальны и с каждым из них связано в точности одно значение; в то же время значения могут быть неуникальны и несколько ключей могут быть связаны с одним значением. В PHP ключи массива должны быть строками или целыми числами; элементы же могут быть любого типа.

Для создания массива можно использовать конструктор массива, представляющий собой заключенное в квадратные скобки перечисление элементов; элементы массива перечисляются через запятую, а ключ и значение разделяются стрелкой. В предыдущих версиях PHP вместо квадратных скобок использовалось ключевое слово **`array`** и круглые скобки. Пример:

```
$a1 = ['ru' => 'Russian', 'en' => 'English'];
$a2 = [1 => 'one', 3 => 'three', 5 => 'five'];
$a3 = array(1 => 'one', 3 => 'three', 5 => 'five');
```

Если в качестве ключа указывается строка, представляющая собой десятичную запись целого числа, то эта строка автоматически преобразовывается в целое число. Пример:

```
$a = ['1' => 'one', '2' => 'two'];
// a = [1 => 'one', 2 => 'two']
```

В конструкторе массива ключи всех или некоторых элементов можно не указывать; в этом случае ключом будет целое число, на единицу большее максимального целого числа, использованного в качестве ключа среди предшествующих элементов, либо ноль, если все предшествующие элементы имели строковые ключи. Пример:

```
$a4 = ['Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su'];
      // [0 => 'Mo', 1 => 'Tu', …]
$a5 = [10 => 'ten', 'eleven',
       20 => 'twenty', 'twenty one'];
      // [10 => 'ten', 11 => 'eleven', …]
```

Поскольку на тип элементов массива не накладывается никаких ограничений, ими, в свою очередь, также могут быть массивы, называемые в этом случае вложенными. Вложенные массивы позволяют создавать весьма сложные структуры данных:

```
$users = [
    'vasya' => [
        'name' => 'Вася',
        'password' => '123',
        'friends' => ['masha' => true,
                      'vova' => true],
        'photos' => [
            [ 'file' => 'vacations.jpg',
              'width' => 600,
              'height' => 400,
              'caption' => 'На отдыхе']]],
    'masha' => [
        'name' => 'Маша']];
```

Как и в других С-подобных языках для обращения к элементам массива в PHP используются квадратные скобки. В том случае, если в качестве ключа указывается строка, являющаяся десятичной записью целого числа, эта строка автоматически преобразовывается в число. Пример:

```
$a1['ru']       // 'Russian'
$a2[1]          // 'one'
$a2['3']        // 'three'
$users['vasya'] // ['name' => 'Вася']
$users['vasya']['name']             // 'Вася'
$users['vasya']['friends']['masha'] // true
$users['vasya']['photos'][0] // ['file' => 'vacations.jpg']
$users['vasya']['photos'][0]['width'] // 600
```

Поведение PHP при обращении к несуществующему элементу массива такое же, как при обращении к несуществующей переменной — возвращается значение **null** и генерируется предупреждение.

Если при присваивании элементу массива опустить ключ (но оставить квадратные скобки), то в качестве ключа будет использовано целое число, на единицу большее максимального целочисленного ключа, либо ноль (т. е. произойдет то же, что и в том случае, когда опущен ключ в конструкторе массива). Пример:

```
$a = [];
$a[] = 'zero'; $a[] = 'one'; $a[] = 'two'; $a[] = 'three';
// [0 => 'zero', 1 => 'one', 2 => 'two', 3 => 'three']
```

Для проверки наличия в массиве элемента с определенным ключом и для удаления элемента по ключу можно использовать уже упоминавшиеся операторы **isset**, **empty** и **unset** соответственно. Пример:

```
$a = [100, 200, 300];
isset($a[1])  // true
unset($a[1]); // $a = [0 => 100, 2 => 300]
isset($a[1])  // false
```

Обратите внимание, что удаление некоторого элемента не изменяет ключей оставшихся элементов; из-за этого в сплошной нумерации элементов массива могут возникать пропуски.

В отличие от языков C и C++ в PHP вполне допустимо присваивание целых массивов, причем в отличие от Java и C# в PHP при присваивании массива не создается псевдонима — для прикладной программы присваивание выглядит как копирование всего содержимого присваиваемого массива, включая вложенные массивы, если таковые есть. Примеры:

```
$a = ['Russia' => 'Moscow', 'China' => 'Beijing'];
$b = $a;
$a['Japan'] = 'Tokyo'; unset($a['China']);
$b['Ukraine'] = 'Kiev'; unset($b['Russia']);
// стало: a = ['Russia' => 'Moscow', 'Japan' => 'Tokyo']
//        b = ['China' => 'Beijing', 'Ukraine' => 'Kiev']
```

Уточним, что в действительности PHP не копирует весь массив при присваивании — вместо этого копируется ссылка и увели-

чивается на единицу значение счетчика блокировок массива. При попытке изменения массива счетчик блокировок проверяется и, если его значение оказывается больше единицы, то создается копия, в которую и вносятся изменения; у копии счетчик ссылок устанавливается равным единице, а у исходного массива уменьшается на единицу. Такой механизм называется ленивым копированием; благодаря этому механизму фактического копирования часто удается избежать (например, при возврате массива из функции).

Массивы можно сравнивать с использованием как строгого, так и нестрогого равенства. Если используется нестрогое равенство, то два массива считаются равными, если они содержат одинаковый набор ключей и каждый ключ в обоих массивах связан со значениями, которые также нестрого равны между собой. Если же используется строгое равенство, то помимо наличия одинакового набора ключей и (теперь уже) строгого равенства значений для того, чтобы массивы считались равными, необходим одинаковый порядок элементов в каждом массиве. Порядок элементов определяется очередностью их добавления в массив (или же очередностью их указания в конструкторе массива); после добавления элементов их порядок может быть изменен некоторыми встроенными функциями. Примеры:

```
$a = ['three' => 3];
$b = ['three' => '3'];
$a == $b  // true
$a === $b // false
$a = ['three' => 3, 'five' => 5];
$b = ['five' => 5, 'three' => 3];
$a == $b  // true
$a === $b // false
```

Для организации цикла по элементам массива в PHP есть специальный оператор цикла, задаваемый ключевым словом **`foreach`**. Пример:

```
$a = [10, 20, 30]; $s = 0;
foreach ($a as $n) {
    $s += $n * $n;
}
echo $s; // 1400
```

После ключевого слова **as** указывается переменная, которой на каждой итерации присваивается значение, хранящееся в очередном элементе массива. В теле оператора **foreach** можно использовать не только значение, но и ключ соответствующего элемента массива; для этого после ключевого слова **as** указывают две переменные, разделяя их стрелкой. Пример:

```
$a = [2 => 3, 3 => 1, 5 => 2]; $n = 1;
foreach ($a as $p => $q) {
    $n *= pow($p, $q);
}
echo $n; // 600
```

Наконец, оператор **foreach** позволяет изменять значения элементов; для этого переменную, получающую значение элемента массива, следует предварить амперсандом.[3] Пример:

```
$a = ['abc', 'DEF', 'Ghi'];
foreach ($a as &$s) {
    $s = strtoupper($s);
}
// $a = ['ABC', 'DEF', 'GHI']
```

Подобно другим управляющим конструкциям у оператора **foreach** есть альтернативный синтаксис. Пример:

```
<ul>
    <?php foreach ($menu as $label => $url): ?>
        <li>
            <a href="<?php echo htmlspecialchars($url) ?>">
                <?php echo htmlspecialchars($label) ?>
            </a>
        </li>
    <?php endforeach; ?>
</ul>
```

В PHP встроено большое количество функций для работы с массивами; далее рассмотрим некоторые из них.

Функция `count` возвращает количество элементов в массиве.

Функции `in_array` и `array_search` ищут значение, заданное первым параметром, в массиве, заданном вторым параметром. В

---

[3] Следует понимать это так, что в этом случае значение очередного элемента массива передается в тело оператора **foreach** не по значению, а по ссылке. Подробнее о ссылках будет рассказано в разделе, посвященном функциям.

случае успеха первая функция возвращает **true**, а вторая — ключ того элемента, в котором нашлось искомое значение (если искомое значение встречается несколько раз, то возвращается ключ, соответствующий первому по порядку элементу). В случае неудачи обе функции возвращают **false**. Также обе функции могут принимать третий параметр, задающий вид сравнения: если этот параметр истинен, то используется строгое сравнение, иначе — нестрогое (это же является вариантом по умолчанию). Следует помнить, что в отличие от поиска ключа, характеризующегося сложностью $O(1)$, поиск значения выполняется путем просмотра всех элементов, и, стало быть, характеризуется сложностью $O(n)$. Пример:

```
$a = [10, 30, '30', '50'];
in_array(50, $a)              // вернет true
in_array(50, $a, true)        // вернет false
array_search(30, $a)          // вернет 1
array_search('30', $a)        // вернет 1
array_search('30', $a, true) // вернет 2
array_search(50, $a, true)    // вернет false
```

Для сортировки элементов массива можно использовать функции `sort`, `rsort`, `asort`, `arsort`, `ksort` и `krsort`. Функции `sort`, `asort` и `ksort` сортируют массив по возрастанию, а функции `rsort`, `arsort` и `krsort` — по убыванию. Функциями `sort`, `rsort`, `asort` и `arsort` сортировка производится на основе значения элементов, а функциями `ksort` и `krsort` — но основе ключей элементов. Наконец, функции `asort` и `arsort` отличаются от функций `sort` и `rsort` тем, что первые сохраняют ключи элементов массива без изменений, а вторые заменяют их последовательными целыми числами начиная с нуля (функции `ksort` и `krsort`, разумеется тоже сохраняют ключи элементов). Примеры:

```
$a = ['Russian ' => 'ru', 'English' => 'en',
      'German' => 'de', 'Chinese' => 'zh'];
$c = $b = $a;
sort($a);  // стало: [0 => 'de', 1 => 'en',
           //         2 => 'ru', 3 => 'zh']
asort($b); // стало: ['German' => 'de', 'English' => 'en',
           // 'Russian ' => 'ru', 'Chinese' => 'zh']
ksort($c); // стало: ['Chinese' => 'zh', 'English' => 'en',
           // 'German' => 'de', 'Russian ' => 'ru']
```

Все шесть функций сортировки кроме первого параметра, задающего массив, могут принимать второй параметр, который задает способ сравнения, используемый при сортировке. Возможные значения этого параметра представлены следующими константами: `SORT_REGULAR` — как в операторе сравнения (это вариант по умолчанию), `SORT_NUMERIC` — с преобразованием в числовой тип, `SORT_STRING` и `SORT_LOCALE_STRING` — с преобразованием в строковый тип (соответственно без учета и с учетом региональных настроек), `SORT_NATURAL` — в т. н. естественном порядке (как в Проводнике Windows), `SORT_FLAG_CASE` — без учета регистра (эту константу можно комбинировать с использованием оператора двоичного «или» с константами `SORT_STRING` и `SORT_NATURAL`). Все функции сортировки характеризуются сложностью $O(n \log n)$.

Функции `array_keys` и `array_values` — возвращают массив, значениями элементов которого являются соответственно ключи и значения исходного массива, заданного первым параметром, а ключами — целые числа начиная с нуля в соответствии с порядком исходного массива. Примеры:

```
$a = ['zero' => 'rei', 'one' => 'ichi', 'two' => 'ni'];
array_keys($a)    // возвращает: [0 => 'zero', 1 => 'one',
                  //               2 => 'two']
array_values($a)  // возвращает: [0 => 'rei', 1 => 'ichi',
                  //               2 => 'ni']
```

Функция `array_keys` помимо исходного массива может принимать вторым параметром некоторое значение; в этом случае функция возвращает только те ключи, значения которых совпадают с этим значением. Наконец, третий параметр может использоваться для задания вида сравнения: если он истинен, то используется строгое сравнение, иначе — нестрогое (это же является вариантом по умолчанию).

```
$a = ['one' => 1, 'ichi' => '1', 'two' => 2, 'ni' => '2'];
array_keys($a, 2)       // возвращает: [0 => 'two',
                        //               1 => 'ni']
array_keys($a, 2, true) // возвращает: [0 => 'two']
```

Функция `array_combine` выполняет действие, обратное совместному действию функций `array_keys` и `array_values` — она принимает два параметра, которые должны быть массивами одинако-

вого размера, и возвращает массив, ключи которого берутся из значений первого исходного массива, а значения — из значений второго. Функция `array_combine` учитывает порядок элементов в исходных массивах и игнорирует их ключи. Пример:

```
array_combine(['one', 'two', 'three'],
              ['ichi', 'ni', 'san'])
// возвращает: ['one' => 'ichi', 'two' => 'ni',
//              'three' => 'san']
```

Функция `explode` возвращает массив, полученный путем разбиения строки, заданной вторым параметром, на части с использованием разделителя, заданного первым параметром; максимальное количество частей может быть указано третьим параметром (по умолчанию количество частей не ограничено). Пример:

```
explode(' ', 'one two three')
// возвращает: [0 => 'one', 1 => 'two', 2 => 'three']
explode(' ', 'one two three', 2)
// возвращает: [0 => 'one', 1 => 'two three']
explode(' ', ' a  b ')
// возвращает: [0 => '', 1 => 'a', 2 => '', 3 => 'b',
//              4 => '']
```

Функция `implode` является обратной по отношению к функции `explode`. Она возвращает строку, полученную путем склеивания значений массива, заданного вторым параметром, с использованием строки, заданной первым параметром. Примеры:

```
implode(' ', ['one', 'two', 'three'])
// возвращает: 'one two three'
```

В PHP поддерживается специальная конструкция, позволяющая присвоить значения элементов массива сразу нескольким переменным. Конструкция задается ключевым словом **`list`**, за которым в круглых скобках следуют переменные (или, что-либо еще, чему можно присвоить значение). Конструкция **`list`** записывается слева от знака присваивания; выражение справа должно быть массивом. В результате присваивания первая из указанных переменных получает значение элемента массива с ключом 0, вторая — элемента массива с ключом 1 и т. д. Пример:

```
list($x, $y) = [10, 20]; // стало: x = 10, y = 20
```

3.6. Предопределенные переменные

Еще до начала выполнения программы среда выполнения PHP присваивает значениям некоторым переменным. Такие переменные называются предопределенными. Рассмотрим некоторые из них.

В массив `$_SERVER` записываются переменные окружения. Если интерпретатор PHP выполняется под управлением Web-сервера (внешнего или встроенного в PHP), то содержимое этого массива соответствует стандарту CGI. Пример:

```
// запрос GET example.ru:8000/test.php/info?id=100&sort=asc
$_SERVER['REQUEST_METHOD'] // GET
$_SERVER['SERVER_NAME']    // example.ru
$_SERVER['SERVER_PORT']    // 8000
$_SERVER['SCRIPT_NAME']    // /test.php
$_SERVER['PATH_INFO']      // /info
$_SERVER['QUERY_STRING']   // id=100&sort=asc
```

В массив `$_GET` записываются параметры запроса, переданного через строку запроса.[4] В простейшем случае ключи массива представляют собой имена параметров, а значения — декодированные значения параметров. Если строка запроса содержит несколько одноименных параметров, то по умолчанию в массиве `$_GET` оказывается записанным только последнее значение параметра. Примеры:

```
// строка запроса: id=100&sort=asc
$_GET // ['id' => '100', 'sort' => 'asc']
// строка запроса: id=100&id=200
$_GET // ['id' => '200']
// строка запроса: text=Hello%20World
$_GET // ['text' => 'Hello World']
```

Если имя параметра заканчивается одной или несколькими парами квадратных скобок, то в качестве ключа используется только часть имени параметра перед первой квадратной скобкой; остальные части имени, заключенные в квадратные скобки, интерпретируются как ключи вложенного массива, представляющего собой значение параметра. Возможно использование нескольких уровней вложенности, а так же пустых пар скобок, которые интерпретиру-

[4] Этот массив доступен при любом методе запроса, не обязательно GET.

ются также, как аналогичная конструкция при присваивании значения элементу массива. Примеры:

```
// строка запроса: name[first]=John&name[last]=Smith
$_GET // ['name' => ['first' => 'John', 'last' => 'Smith']]
// строка запроса: ids[]=100&ids[]=200&ids[]=300
$_GET // ['ids' => [0 => '100', 1 => '200', 2 => '300']]
```

В массив `$_POST` записываются параметры, считанные из стандартного входного потока (т. е. переданные в теле запроса). Как правило эти параметры представляют собой значения, введенные в поля формы, имеющей атрибут `method`, равный `post`. При разборе входного потока PHP принимает во внимание тип содержимого, который передается в заголовке Content-Type.[5] Если имена параметров содержат квадратные скобки, то они обрабатываются так же, как для строки запроса.

3.7. Функции

Как и в других C-подобных языках, в PHP именованные фрагменты кода называются функциями; набор операторов, составляющих тело функции, заключается в фигурные скобки,[6] а для возврата значения из функции используется оператор **`return`**. Приведем пример:

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

Вызов функции, определенной в программе, ничем не отличается от вызова встроенной функции. Например:

```
cube(11) // 1331
```

Отметим, что в коде вызов функции вполне может предшествовать ее определению.

Подобно языкам C++ и C# язык PHP поддерживает задание для параметров значений по умолчанию. Значение по умолчанию

---

[5] Для задания типа содержимого запроса можно использовать атрибут формы `enctype`. Впрочем, явное задание этого атрибута необходимо только, если форма содержит поля для загрузки файлов; в этом случае в качестве типа содержимого следует указать multipart/form-data.

[6] Альтернативного синтаксиса у функций нет.

используется в том случае, когда в вызове функции соответствующие фактический параметр отсутствует. Параметры, для которых задано значение по умолчанию, называются необязательными. Пример:

```
function normal_density($x, $scale = 1, $location = 0) {
    $x -= $location;
    return exp(-$x * $x / (2 * $scale * $scale))
            / ($scale * sqrt(2 * pi()));
}
normal_density(0.5)         // вернет 0.3520653267643
normal_density(0.5, 2)      // вернет 0.19333405840142
normal_density(0.5, 2, -1) // вернет 0.1505687160774
```

Если в вызове функции отсутствует значение какого-либо обязательного (т. е. не имеющего значения по умолчанию) параметра, то генерируется предупреждение, но выполнение продолжается, причем в теле функции параметр, оставшийся без значения, считается несуществующим — обращение к нему вернет **`null`** и сгенерирует предупреждение так, как если бы это было обращение к несуществующей переменной.

Передача в функцию лишних параметров ни предупреждений, ни, тем более, ошибок, не генерирует. В теле функции можно использовать функцию `func_get_args` для того, чтобы получить значения всех фактических параметров в виде массива, либо функцию `func_get_arg` для того, чтобы получить значение одного фактического параметра по целочисленному индексу (первый параметр имеет индекс 0); помимо этого можно получить количество фактических параметров, воспользовавшись функцией `func_num_args`. Перечисленные встроенные функции дают возможность создавать пользовательские функции с переменным количеством аргументов. Например:

```
function sum() {
    $sum = 0;
    foreach (func_get_args() as $arg) {
        $sum += is_array($arg) ? array_sum($arg) : $arg;
    }
    return $sum;
}
sum()                   // вернет 0
sum(1, 2, 3)            // вернет 6
```

```
sum(1, 2, [3, 4, 5]) // вернет 15
```

Наряду с принятой по умолчанию передачей параметров по значению возможна передача параметров по ссылке; в объявлении функции, параметры, передаваемые по ссылке, предваряются амперсандом. Например:

```
function swap(&$x, &$y) {
    list($x, $y) = [$y, $x];
}
$a = 10; $b = 20;
swap($a, $b);          // стало: $a = 20, $b = 10
$c = [100, 200, 300];
swap($c[0], $c[2]);    // стало: $c = [300, 200, 100]
```

По сути при передаче параметра по ссылке вместо копирования значения фактического параметра создается псевдоним этого значения.[7] Создать псевдоним можно не только при передаче параметра в функцию, но также в операторе присваивания; для этого амперсанд ставится справа от оператора присваивания (можно представлять себе происходящее так, что амперсанд подавляет копирования присваиваемого значения). Пример:

```
$x = 10;
$y = &$x;
$y = 12;               // $x = 12
$z = &$y;
$z = 14;               // $x = 12
$a = [100, 200, 300];
$e = &$a[1];
$e = 400;              // $a = [100, 200, 300]
$a = [];
$a[] = &$a;            // $a = [[[...]]]
```

Помимо вызова функции и присваивания, псевдоним может быть создан при возврате значения из функции; в этом случае амперсанд ставится перед именем функции (такую ситуацию можно назвать возвратом значения по ссылке). Пример:

```
function &iif($cond, &$then, &$else) {
    return $cond ? $then : $else;
```

[7] В отличие от языка Perl в PHP нет такого типа данных как ссылка. Также в PHP нет эффективного способа узнать, имеет ли данное значение псевдонимы.

```
}
$a = 1; $b = 2;
$c = &iif(true, $a, $b);
$c = 3; // стало: $a = 3
```

Следует помнить, что, если функция возвращает значение по ссылке, то в операторе присваивания этого значения также необходим амперсанд, поскольку в противном случае произойдет копирование возвращенного значения.

Отметим, что использовать присваивание с созданием псевдонимов, а также возврат по ссылке следует экономно, поскольку обилие псевдонимов сильно усложняет отладку программы; в частности, псевдонимы не следует использовать с целью повышения быстродействия в расчете на снижение затрат времени на копирование данных (никакого прироста быстродействия не будет, поскольку в PHP, как сообщалось выше, и так используется ленивое копирование).

Одной из немногих особенностей PHP, препятствующих написанию плохого кода, является то, что по умолчанию в теле функции не видны глобальные (т. е. определенные за пределами функции) переменные. Для использования глобальной переменной программист должен явно объявить ее в теле функции; например:

```
function increment_counter() {
    global $counter;
    ++$counter;
}
```

Особый случай представляют т. н. суперглобальные переменные, к которым относятся предопределенные переменные `$_SERVER`, `$_GET` и `$_POST`, а также некоторые другие. В отличие от обычных глобальных переменных суперглобальные переменные всегда видны как вне тел функций, так и внутри них и не требуют объявления с помощью **`global`**.

В PHP поддерживаются локальные статические переменные. Так же как в C такие переменные, в отличие от обычных локальных переменных, сохраняют свое значение между вызовами функции. Подобно глобальным переменным, используемым в теле функции, статические переменные требуют объявления, которое, однако, в отличие от объявления глобальных переменных, может включать

инициализирующее выражение. В инициализирующем выражении допускаются только константы (включая пустой конструктор массива) — использовать переменные и функции нельзя; благодаря такому ограничению начальное значение статической переменной не зависит от того, в какой момент содержащая ее функция была вызвана в первый раз. Пример:

```
function increment_counter() {
    static $count = 0;
    return ++$count;
}
```

В PHP поддерживается косвенный вызов функции; для этого в операторе вызова функции достаточно указать на месте имени функции переменную, содержащую имя функции в виде строки. Например:

```
$f = 'cube';
$f(12);       // 1728
```

Перед вызовом функции по имени можно проверить, существует ли она, воспользоваться встроенной функцией `function_exists`, которая в качестве параметра принимает имя функции и возвращает **`true`**, если функция с указанным именем существует, и **`false`** — в противном случае.

Косвенный вызов функций часто позволяет избежать громоздких операторов выбора. Например:

```
function render_index_page() {
    echo 'Here we render the index page';
}
function render_registration_page() {
    echo 'Here we render the registration page';
}
$pageName = @$_GET['p'] or $pageName = 'index';
$pageFunction = "render_{$pageName}_page";
if (function_exists($pageFunction)) {
    $pageFunction();
} else {
    echo 'No such page';
}
```

Отметим, что добавление к имени страницы, извлеченному из строки запроса, какого-либо префикса и/или суффикса необходимо

по соображениям безопасности: в противном случае у посетителя сайта появится возможность вызвать любую функцию.

В PHP много встроенных функций, использующих косвенный вызов. Рассмотрим некоторые из них.

Функция `array_map` вызывает переданную ей первым параметром функцию для значений каждого из элементов массивов, переданных ей параметрами, начиная со второго, и возвращает результат в виде нового массива, сформированного из значений, возвращенных переданной функцией. Количество параметров, принимаемых функцией, переданной первым параметром в `array_map`, должно совпадать с количество переданных в `array_map` массивов. Пример:

```
array_map('strtoupper', ['ru', 'en', 'jp'])
// вернет [0 => 'RU', 1 => 'EN', 2 => 'JP']
array_map('pow', [2, 3, 5], [3, 2, 1])
// вернет [0 => 8, 1 => 9, 2 => 5]
```

Если в функцию `array_map` передан только один массив, то возвращаемый массив сохраняет ключи исходного, иначе в качестве ключей возвращаемого используются последовательные целые числа, начиная с нуля, а ключи исходных массивов игнорируются.

Функция `array_filter` возвращает массив, полученный из исходного, переданного первым параметром, путем исключения из него тех элементов, для которых переданная вторым параметром функция вернула ложное значение. Функция `array_filter` не меняет исходный массив. Если при вызове `array_filter` второй параметр (задающий функцию) не указан, то из исходного массива исключаются ложные значения. Пример:

```
array_filter([1, '2', '', 'x', 0], 'is_numeric')
// вернет [0 => 1, 1 => '2', 4 => 0]
array_filter([1, '2', '', 'x', 0])
// вернет [0 => 1, 1 => '2', 3 => 'x']
```

Для косвенного вызова можно использовать встроенные функции `call_user_func` и `call_user_func_array`. Первая из них принимает произвольное количество параметров (но, не меньше одного), первый из которых задает вызываемую функцию, а остальные — ее параметры; в сущности функция `call_user_func` ничего не добавляет по сравнению с ранее рассмотренным способом

косвенного вызова. Вторая функция принимает два параметра: первый из которых, как и в предыдущем случае, задает функцию, а второй —все параметры вызова в виде массива; благодаря заданию параметров в виде массива функция `call_user_func_array` позволяет выполнять косвенный вызов в случае переменного количества параметров. Пример:

```
function benchmark_array($func, $args) {
    $startTime = microtime(true);
    call_user_func_array($func, $args);
    return microtime(true) - $startTime;
}
function benchmark($func) { // обеспечивает более
                            // «чистый» вызов
    return benchmark_array($func,
            array_slice(func_get_args(), 1));
}
function factorize($num) { // разложение на множители
    $divs = [];
    for ($d = 2; $num > 1; ++$d) {
        $c = 0;
        for (; 0 == $num % $d; $num /= $d) { ++$c; }
        if ($c) { $divs[$d] = $c; }
    }
    return $divs;
}
printf("%.5f s.", benchmark('factorize', 391937));
// выведет что-то типа 0.00003 s.
printf("%.5f s.", benchmark('factorize', 391939));
// выведет что-то типа 0.03851 s.
printf("%.5f s.", benchmark_array('factorize', [391939]));
// выведет что-то типа 0.03683 s.
```

Подобно многим современных языкам в PHP поддерживаются λ-функции. λ-функция — это выражение, значением которого является функция.[8] В отличие от обычной функции λ-функция является безымянной. Приведем пример:

```
$f = function($x) { return $x * $x; };
echo $f(23);                              // 529
```

[8] Если быть более точным, то оператор создания λ-функции возвращает объект класса Closure, который можно использовать как функцию.

В отличие от многих других языков, поддерживающих λ-функции, в PHP λ-функции не обеспечены замыканием; это означает, что по умолчанию в PHP в теле λ-функции не видны переменные, определенные вне самой λ-функции. Для того, чтобы сделать внешние переменные видимыми внутри функции их следует включить в контекст, который обозначается ключевым словом **use** и записывается после списка параметров но перед телом λ-функции. По умолчанию в момент создания λ-функции значения переменных, включенных в контекст, копируются в созданную функцию; в дальнейшем всякая связь между включенными в контекст переменными и λ-функцией теряется. Пример:

```
$x = 100;
$getX = function() use($x) { return $x; };
$x = 200;
echo $getX(); // напечатает 100
$setX = function($newX) use($x) { $x = $newX; };
$setX(300);
echo $x;      // напечатает 200
```

В теле λ-функции значения контекстных переменных можно изменять, однако эти изменения не сохраняются в промежутке между вызовами.

Если необходимо, чтобы λ-функция могла изменять значение какой-либо внешней переменной, то такую переменную следует включить в контекст по ссылке, предварив ее амперсандом. Пример:

```
$z = 1000;
$getZ = function() use(&$z) { return $z; };
$z = 2000;
echo $getZ(); // напечатает 2000
$setZ = function($newZ) use(&$z) { $z = $newZ; };
$setZ(3000);
echo $z;      // напечатает 3000
```

λ-функции можно использовать везде, где допустимо имя функции.

3.8. Работа с файлами

Функции для файлового ввода-вывода в PHP напоминают таковые в стандартной библиотеке языка C. Для открытия файла ис-

пользуется функция `fopen`. В случае успеха эта функция возвращает ресурс, представляющий поток, который следует использовать для чтения из файла и записи в него. Функция `fopen` принимает до четырех параметров, из которых первые два являются обязательными: первый параметр задает имя файла, а второй — тип доступа. Для задания типа доступа используется строка, причем собственно тип доступа задается первым символом:

- `r` — чтение, если файла с указанным именем не существует, то функция `fopen` возвращает **`false`**;
- `w` — запись, если файл с указанным именем уже существует, то прежнее содержание файла теряется;
- `a` — дописывание в конец, если файла с указанным именем не существует, то такой файл автоматически создается; сразу после открытия указатель устанавливается на конец файла;
- `x` — как `w`, но, если файл с указанным именем уже существует, то функция возвращает **`false`**, а прежнее содержание файла сохраняется;
- `c` — как `a`, но сразу после открытия указатель устанавливается на начало файла.

После буквы, задающей режим доступа, можно указать знак `+`, означающий, что открываемый файл можно будет как читать, так и записывать.

Вместо имени файла можно указать URL; в частности поддерживаются следующие схемы:

- `http` — доступ на чтение к указанному ресурсу по протоколу HTTP;
- `ftp` — доступ к указанному удаленному файлу по протоколу FTP;
- `php` — чтение или запись указанного встроенного потока PHP. Определены следующие потоки: `output` — выходной поток, используемый операторами **`echo`** и **`print`**; `input` — входной поток, можно использовать для чтения тела HTTP-запроса; `memory` — временный поток в память (поддерживает как запись, так и чтение).

Для чтения из потока используется функция `fread`, которая принимает два параметра: поток (представленный ресурсом) и число, задающее максимальное количество считываемых данных (в

байтах). В случае успеха функция возвращает считанные данные в виде строки; противном случае возвращается **`false`**. Если поток, из которого производится чтение, соответствует обычному файлу и если не достигнут конец потока, то количество фактически считанных данных всегда равно заданному максимальному количеству. Если же поток соответствует чему-либо иному (например ресурсу, доступному по протоколу HTTP, или удаленному файлу, доступному по протоколу FTP), то количество считанных данных может быть меньше максимального количества даже в том случае, когда конец потока не достигнут; гарантируется лишь то, что будет считан по крайней мере один байт.[9] При достижении конца потока (вне зависимости от его вида) функция `fread` возвращает пустую строку.

Если файл является текстовым, то для его чтения удобно использовать функцию `fgets`. Эта функция отличается от `fread` тем, что считывает из потока не более одной строки. Считается, что строки разделяются разделителем строк, представленным встроенной константой `PHP_EOL`, значение которой зависит от платформы (в Windows используется `'\r\n'`, в Unix — `'\n'`). Разделитель строк включается в возвращаемую строку. Другое отличие функции `fgets` от функции `fread` состоит в том, что второй параметр, задающий максимальное количество считываемых данных, является необязательным; если он отсутствует то количество считываемых данных не ограничивается.

Для записи в поток используется функция `fwrite`, которая принимает три параметра, два из которых обязательные: поток и строка, представляющая данные, которые следует записать. В случае успеха функция возвращает количество фактически записанных байтов; в противном случае возвращается `false`. Если поток, в который производится запись, соответствует обычному файлу, то количество фактически записанных данных всегда равно количеству переданных. Если же поток соответствует чему-либо иному, то количество фактически записанных данных может быть меньше ко-

---

[9] Такое поведение — особенность не PHP, а соответствующей системной функции ввода-вывода. К сожалению, начинающие программисты часто забывают об этой особенности при программировании сетевых приложений.

личества переданных (для записи оставшихся данных необходимо вызвать функцию `fwrite` еще один или несколько раз). Пример:

По аналогии с функцией `fgets` в PHP поддерживается функция `fputs`, являющаяся синонимом функции `fwrite`.

После окончания работы с потоком его можно закрыть, воспользовавшись функцией `fclose`, принимающей один параметр — поток. Несмотря на то, что по окончании выполнения программы все открытые потоки закрываются автоматически, явное закрытие всех потоков является рекомендуемой практикой, поскольку позволяет сократить вероятность конфликтов при совместном доступе к файлу, а также избежать утечки памяти, связанной с наличием неиспользуемых потоков. Явное закрытие всех потоков особенно важно тогда, когда предполагается, что программа будет выполняться в течение продолжительного времени.

В том случае, когда размер файла невелик, все его содержимое в виде строки можно получить, воспользовавшись функцией `file_get_contents`. Единственным обязательным параметром этой функции является имя файла. Аналогом функции `file_get_contents` для записи является функция `file_put_contents`. Эта функция принимает до четырех параметров, из которых два первых — обязательные: имя файла и записываемые данные.

Необходимо помнить, что web-сервер может одновременно обслуживать несколько запросов; это значит, что может оказаться так, что один экземпляр программы может попытаться получить доступ к файлу в то время, пока этот файл используется другим экземпляром. По умолчанию при работе с файлами никаких блокировок не накладывается. Это приводит к тому, что в один и тот же файл могут одновременно записывать данные несколько программ (или несколько экземпляров одной программы); последствия этого предсказуемы. Простейший способ избежать этого — записать данные во временный файл с уникальным именем, после чего этот файл переименовать. Для генерации уникального имени временного файла можно использовать функцию `tempnam`, которой передается два параметра: имя каталога, в котором должен находиться временный файл, и префикс имени файла. В случае успеха функция возвращает сгенерированное имя файла.

Другой способ организовать совместный доступ к файлу — это использование функции `flock`, которая устанавливает режим блокировки при совместном доступе. Эта функция принимает два параметра: ресурс, представляющий открытый поток, и устанавливаемый режим блокировки, который может быть задан следующими константами: `LOCK_SH` — устанавливает разделяемую блокировку (блокировку на чтение); `LOCK_EX` — устанавливает неразделяемую блокировку (блокировку на запись); `LOCK_UN` — снимает ранее установленную блокировку. В каждый момент времени на одном и том же файле возможно наличие либо произвольного количества разделяемых блокировок, либо наличие не более одной неразделяемой блокировки. В случае успеха функция возвращает значение **`true`**. В случае, если в момент вызова требуемая блокировка не может быть установлена из-за того, что файл уже имеет несовместимую блокировку, наложенную другим процессом, функция `flock` не возвращает управление до тех пор, пока затребованная блокировка не будет установлена. Следует помнить, что действие функции `flock` состоит только в упорядочении доступа к файлу; сама по себе эта функция не блокирует ни запись ни чтение.

Помимо универсальных функций чтения из потока и записи в поток в PHP имеются функции, которые удобно применять в том случае, когда данные в файле находятся в некотором распространенном формате. Например, для чтения и записи данных в формате CSV можно использовать функции `fgetcsv` и `fputcsv`. Эти функции отличаются от функций `fgets` и `fputs` тем, что для представления данных используется массив.

Все вышеперечисленные функции позволяют считывать и записывать только данные некоторых, заранее определенных типов. Если в файл необходимо записать значение произвольного типа, причем так, чтобы после считывания можно было бы в точности восстановить записанное значение, то можно воспользоваться функцией `serialize`, преобразующей значение произвольного типа в строку. Эта функция принимает один параметр, задающий преобразуемое значение, и возвращает результат преобразования. Для обратного преобразования используется функция `unserialize`. Эта функция также принимает один параметр, который должен быть

строкой, сформированной функцией `serialize`, и возвращает исходное значение. Пример:

```
$albums = [
    ['title' => 'Angels Fall First',
     'artist' => 'Nightwish',
     'year' => 1997],
    ['title' => 'Oceanborn',
     'artist' => 'Nightwish',
     'year' => 1998]];
echo serialize($albums);
// выведет (одной строкой без пробелов)
// a:2:{
//   i:0;
//   a:3:{
//     s:5:"title";
//     s:17:"Angels Fall First";
//     s:6:"artist";
//     s:9:"Nightwish";
//     s:4:"year";
//     i:1997;}
//   i:1;
//   a:3:{
//     s:5:"title";
//     s:9:"Oceanborn";
//     s:6:"artist";
//     s:9:"Nightwish";
//     s:4:"year";
//     i:1998;}}
```

Функции `serialize` и `unserialize` позволяют точно восстановить исходное значение, но используют формат, специфичный для PHP. В том случае, когда требуется организовать обмен данными с программами, написанными на других языках (например на JavaScript) удобно использовать формат JSON.[10] Для преобразования значения в формат JSON используется функция `json_encode`, первый параметр которой задает преобразовываемое значение, а возвращаемое значение — результат преобразования. Для обратного преобразования используется функция `json_decode`, принимающая до четырех параметров, первый из которых является обязательным — он задает строку в формате JSON. Пример:

[10] JavaScript Object Notation; читается как «джейсон».

```
echo json_encode($albums);
// выведет (одной строкой без пробелов)
// [{"title":"Angels Fall First",
//   "artist":"Nightwish",
//   "year":1997},
//  {"title":"Oceanborn",
//   "artist":"Nightwish",
//   "year":1998}]
```

Кроме функций, предназначенных для чтения и записи файлов, в PHP поддерживается довольно много функций, предназначенных для других действий с файлами на диске, а также с именами файлов.

Для выделения из пути к файлу собственно имени файла используется функция `basename`, а для выделения пути к родительскому каталогу файла — функция `dirname`. Обе функции принимают первым параметром путь к файлу, а возвращают соответствующий компонент пути. Отметим, что как функция `basename`, так и функция `dirname` манипулируют лишь строкой символов, представляющей путь к файлу и не проверяют фактическое существование соответствующего файла. Пример:

```
$file = 'dir1\\dir2\\dir3\\file.ext';
echo basename($file);          // выведет file.ext
echo dirname($file);           // выведет dir1\dir2\dir3
echo dirname(dirname($file)); // выведет dir1\dir2
```

Нормализация имени файла может быть выполнена функцией `realpath`. Эта функция принимает исходное имя файла в качестве параметра и возвращает абсолютный путь к файлу, не содержащий ссылок на родительский каталог. В отличие от функций `basename`, и `dirname` функция `realpath`, проверяет фактическое существование файла и возвращает `false`, если заданного файла нет. Пример:

```
$file = @$_GET['f'];
if (null !== $file) {
    $dataDir = realpath('data');
    // проверить, не пытается ли посетитель сайта
    // получить доступ к файлу,
    // находящемуся за пределами подкаталога data
    // текущего каталога.
    if ($file = realpath($dataDir . DIRECTORY_SEPARATOR
            . $file)
            and 0 === strpos($file, $dataDir)) {
```

```
        // предоставить доступ к файлу
    } else {
        // отказать в доступе к файлу
    }
}
```

Для поиска файлов удобно использовать функцию `glob`, принимающую два параметра, первый из которых обязателен и задает маску поиска. В маске можно использовать символы `?` и `*` для обозначения соответственно одного любого символа и последовательности любых символов произвольной длины. Функция `glob` возвращает массив, содержащий имена файлов, соответствующих маске. Возвращаемые имена включают путь, только если таковой был указан в маске. Второй параметр функции `glob`, если присутствует, должен быть целым числом, биты которого задают дополнительные опции; в частности, определены следующие опции: `GLOB_NOSORT` — не производится сортировка возвращаемого списка (т. е. файлы возвращаются в том порядке в котором хранятся в соответствующей структуре на диске), `GLOB_BRACE` — позволяет указать несколько альтернативных масок, перечислив изменяющиеся части через запятую и заключив их в фигурные скобки, `GLOB_ONLYDIR` — в возвращаемые список включаются только имена каталогов. Пример:

```
// поиск в каталоге images файлов,
// соответствующих маскам *.jpeg, *.jpg и *.png
foreach (glob('images\\*.{jpg,jpeg,png}',
              GLOB_BRACE | GLOB_NOSORT) as $file) {
    // что-то сделать с файлом $file
}
```

Для того, чтобы отличить файл, являющийся каталогом, можно воспользоваться функцией `is_dir`, принимающей имя файла и возвращающей **`true`**, если соответствующий файл является каталогом, и **`false`** — в противном случае. Соответственно, для того, чтобы отличить обычный файл, можно воспользоваться функций `is_file`, которая аналогична ранее рассмотренной функции `is_dir`. Пример:

```
$paths = [''];
do {
    foreach (glob(current($paths) . '*',
             GLOB_NOSORT) as $file) {
```

```
        if (is_dir($file)) {
            $paths[] = $file . DIRECTORY_SEPARATOR;
        } elseif (is_file($file)) {
            // что-то сделать с файлом $file
        }
    }
} while (next($paths));
```

Время последнего изменения файла, а также время последнего обращения к файлу можно узнать с помощью функций `filemtime` и `fileatime` соответственно. Обе функции принимают первым параметром имя файла и возвращают целое число, представляющее собой количество секунд, прошедшее с полночи 1 января 1970 года (это стандартный способ представления даты и времени в PHP, использующийся, например, функцией `date`, которая преобразует дату и/или время в строку символов).

Переименовать и/или переместить файл (или каталог) можно воспользовавшись функцией `rename`, принимающей два параметра: первый задает текущее имя файл, а второй — новое имя. Для копирования можно использовать функцию `copy`, которая так же, как и функция `rename`, принимает два параметра. Наконец, чтобы удалить файл надо воспользоваться функцией `unlink`, принимающей один параметр, задающий удаляемый файл. В случае успеха все три функции возвращают значение **`true`**; в противном случае возвращается значение **`false`.**

Для создания каталога используется функция `mkdir`, принимающая до четырех параметров, первый из которых является обязательным — он задает путь к создаваемому каталогу. Второй параметр задает режим доступа к создаваемому каталогу в соответствии с соглашениями Unix (в Window этот параметр игнорируется). Третий параметр, если задан и равен **`true`,** задает рекурсивное создание каталога, т. е. помимо указанного каталога функция `mkdir` автоматически создаст все несуществующие родительские каталоги. В случае успеха функция возвращает **`true`**; в противном случае — **`false`**. Для удаления каталога используется функция `rmdir`, принимающая до двух параметров, первый из которых является обязательным и задает имя удаляемого каталога. Удаляемый каталог должен быть пустым; иначе удаление невозможно. Функция

rmdir так же, как и функция mkdir, в случае успеха возвращает **true**; в противном случае — **false.**

В массив $_FILES записывается информация о файлах, загруженных с использованием предназначенного для этого поля ввода. Ключами массива являются имена полей ввода, а значениями вложенные массивы, каждый из которых содержит следующие элементы: name — исходное имя файла на клиенте (как оно показывалось в браузере), type — тип содержимого (например, image/jpeg), size — размер файла в байтах, tmp_name — имя временного файла на Web-сервере, в котором хранится содержимое загруженного файла, error — код ошибки.

3.9. Сессии

Сессия — это набор данных, сохраняемых на web-сервере в промежутке между выполнением запросов. Поскольку одновременно с web-сервером могут работать несколько пользователей, поддерживается одновременное существование нескольких сессий, каждая из которых обладает уникальным идентификатором. Идентификатор сессии хранится на клиенте и включается в каждый запрос. По умолчанию включение реализуется на основе cookies, однако возможна реализация путем автоматического добавления параметра в строку запроса.

Перед обращением к данным сессии ее необходимо открыть вызовом встроенной функции session_start. Эта функция не принимает параметров и возвращает **true** в случае успеха и **false** в противном случае. Если запрос содержит идентификатор ранее созданной сессии, функция session_start пытается загрузить ее. Если в запросе идентификатора сессии нет либо если загрузить ранее созданную сессию не удается, создается новая сессия, для которой генерирует новый идентификатор. Функция session_start может модифицировать набор заголовков отклика, поэтому ее вызов должен предшествовать выводу страницы.

После открытия сессии ее данные помещаются в суперглобальный массив $_SESSION. Пример:

```
$currentUser = $_SESSION['user'];
```

Для изменения какого-либо элемента данных сессии достаточно обновить значение соответствующего элемента массива `$_SESSION`. Пример:

```
$userId = authorize($username, $password);
// какая-то функция для авторизации
if (FALSE !== $userId) {
    $_SESSION['user'] = ['id' => $userId,
                         'name' => $username];
}
```

Идентификатор текущей сессии возвращает функция `session_id`. Эту же функцию можно использовать для установки идентификатора, передавая его ей в качестве параметра (установка идентификатора возможна только перед открытием сессии).

После завершения работы с сессией ее можно закрыть вызовом функции `session_write_close`. Эта функция не принимает параметров. В ходе закрытия сессии ее данные сохраняются. Впрочем, обычно явный вызов функции `session_write_close` не нужен, поскольку незакрытая сессия автоматически закрывается после завершения работы программы.

По умолчанию данные сессий сохраняются в обычных файлах. Путь к этим файлам в конфигурационном файле можно задать директивой `session.save_path`. Во время исполнения значение этой директивы доступно с использованием функции `session_save_path` (разумеется, изменять путь к файлам сессии можно только до открытия сессии).

3.10. Модули

Исходный код приложения может располагаться в нескольких файлах. Для включения файла с исходным кодом используются операторы включения, представляющие собой ключевое слово **`require`**, за которым следует выражение, задающее имя включаемого файла в виде строки. Пример:

```
require 'config.php';
```

Если заданное в операторе **`require`** имя файла не содержит пути (т. е. состоит из собственно имени файла, которому, возможно, предшествуют имена одного или нескольких каталогов, отделенные от имени файла и друг от друга прямой или обратной косой

чертой), то поиск файла производится в каталогах, список которых задан директивой `include_path` в файле настроек,[11] а также в текущем каталоге и каталоге, в котором расположен файл, содержащий оператор включения. Отметим, что значение, заданное директивой `include_path` во время работы программы можно изменить, воспользовавшись функцией `set_include_path`, передав ей единственным параметром список каталогов, который должен представлять собой строку, содержащую имена каталогов, разделенные символом, заданным встроенной константой `PATH_SEPARATOR`.[12] Текущее значение директивы `include_path` можно получить, воспользовавшись встроенной функцией `get_include_path`. Пример:

Если имя файла, указанного в операторе **`require`**, содержит абсолютный или относительный (относительно текущего каталога) путь, то поиск в каталогах, перечисленных в директиве `include_path`, а также в каталоге, в котором расположен включающий файл, не производится.

Отметим, что выражение, задающее имя файла, не обязательно должно быть статическим и может содержать переменные.

Оператор **`require`** можно использовать внутри управляющих конструкций. Более того, оператор включения подобно оператору вызова функции возвращает значение, благодаря чему его можно записывать в правой части оператора присваивания и в списке фактических параметров при вызове функции. Значение, возвращаемое оператором **`require`**, задается во включаемом файле оператором **`return`**, находящимся вне тела функции.

Код, размещенный во включаемом файле, выполняется в том же контексте, что и операторы включения; т. е., если, например, оператор включения расположен в теле функции, то во включаемом файле будут доступны локальные переменные и параметры.

Если включаемый файл содержит определения функций, то повторное включение приведет к ошибке, вызванной попыткой переопределения этих функций. Чтобы избежать этого вместо опера-

---

[11] Обычно этот файл называется php.ini и находится в том же каталоге, что и интерпретатор php.

[12] Значение этой константы зависит от платформы: в Windows используется точка с запятой, а в Unix — двоеточие.

тора **require** следует использовать оператор **require_once**; в том случае, если указанный файл ранее уже был включен, оператор **require_once** ничего не делает.

3.11. Классы

В PHP объектно-ориентированные возможности впервые были реализованы в четвертой версии, однако многие разработчики отнеслись к этой реализации весьма критически из-за крайней ограниченности использованной объектно-ориентированной модели (она немного напоминала таковую из Turbo Pascal). По этой причине в пятой версии объектная модель PHP подверглась коренной ревизии — была взята за основу объектная модель языка Java. На взгляд авторов, это решение также вряд ли можно назвать удачным: Java в отличие от PHP является строго типизированным языком и многие аспекты объектной модели Java являются прямыми следствиями этого факта; например наличие интерфейсов и абстрактных классов объясняется необходимостью контроля типов при компиляции. В таком динамическом языке как PHP эти аспекты не только выглядят чужеродными, но и фактически не способны выполнять те функции, которые были закреплены за ними изначально; например, те же интерфейсы, столь необходимые в Java, в PHP в сущности бесполезны вследствие того, что никакого контроля типов на этапе компиляции в PHP нет. Неудачность выбора образца для подражания становится вполне очевидна, если взглянуть на такие динамические языки, как Python и Ruby — создатели этих языков не пошли по пути заимствования популярных, но чужеродных идей, и, в результате, объектные модели этих языков не только лучше соответствуют принципам, изначально заложенным в соответствующие языки, но, при этом одновременно проще в освоении и шире в отношении предоставляемых возможностей.

Все же, сколь бы уязвимой для критики не была бы объектная модель PHP, именно ее наличие является тем фактором, который позволяет рассматривать PHP в качестве инструмента для разработки серьезных проектов, включающих сложную бизнес-логику и требующих больших объемов программного кода. Сказанное означает, что изучение объектной модели и объектно-ориентированных

возможностей PHP является совершенно необходимым для эффективного применения этого языка на практике.

Не только объектная модель, но и сам синтаксис объявления класса скопирован с языка Java. Определение класса может включать в себя объявления методов, свойств (которые, впрочем, правильнее было бы называть полями) и констант. Подобно многим объектно-ориентированным языкам для элементов класса в PHP предусмотрено три области видимости:

- **`private`** (частная) — элемент виден только в том классе, в котором объявлен;
- **`protected`** (защищенная) — элемент виден в том классе, в котором объявлен, а также во всех потомках данного класса;
- **`public`** (общедоступная) — элемент виден везде.

В случае нарушения области видимости возникает фатальная ошибка.

Объявление свойства состоит из имени, которому должно предшествовать указание области видимости, и может включать задание начального значения. Выражение, задающее начальное значение, должно быть вычисляемым на этапе компиляции (подобно начальному значению статической переменной). Если выражение, задающее начальное значение, отсутствует, то начальным значением свойства будет **`null`**. Пример:

```
class Book {
    public $title;
    public $year = 2012;
    private $_price = 37.26;
}
```

Подобно другим C-подобным языкам в PHP для создания объекта класса используется оператор **`new`**. Этот оператор возвращает ссылку[13] на объект. Для обращения к свойству между выражением, задающим объект, и именем свойства пишется стрелка.[14] Пример:

---

[13] Не путать со ссылками, рассматривавшимися в разделе, посвященном функциям.

[14] Во-первых, это напоминает синтаксис языка C, а, во вторых, используемая для тех же целей в Java и JavaScript точка в PHP занята под операцию сцепления строк.

```
$b = new Book();
var_export($b->title); // выведет NULL
var_export($b->year);  // выведет 2012
//var_export($b->_price); будет Fatal error
$b->title = 'Advanced PHP Programming';
$b->year = 2004;
```

Вообще говоря, объявление свойства не является обязательным: по умолчанию подобно тому, как это происходит с переменными, ранее не существовавшее свойство автоматически создается при первом присваивании ему значения, а попытка получения значения несуществующего свойства приводит к замечанию (notice) и возвращает **null**. Аналогично переменным, свойство можно удалить, а также проверить его наличие. Пример:

```
var_export(isset($b->authors));        // выведет false
$b->authors = ['George Schlossnagle'];
var_export(isset($b->authors));        // выведет true
unset($b->year);
var_export(isset($b->year));           // выведет false
```

Несмотря на то, что язык PHP допускает использование необъявленных свойств, следует стараться всегда объявлять в классе все используемые свойства, поскольку это избавляет от необходимости проверять их наличие, а также позволяет задать область видимости (необъявленные свойства имеют область видимости public).

Объявление метода может начинаться с указания области видимости (по умолчанию принимается **public**) и, в остальном, мало отличается от объявления обычной функции. Для вызова метода так же, как и для обращения к свойству, используется стрелка. В теле метода текущий объект (т. е. тот, который в вызове метода являлся результатом выражения, записанного перед стрелкой) представлен встроенной переменной **$this**. В отличие от Java в PHP использование **$this**[15] для доступа к свойствам и методам текущего объекта обязательно (если забыть написать **$this** и стрелку перед именем свойства или метода, то они будут трактоваться как гло-

[15] Для методов можно вместо **$this** с последующей стрелкой использовать ключевое слово **static** с последующим двойным двоеточием, однако авторы не рекомендуют такой вариант.

бальная константа и глобальная функция соответственно). Попытка вызова несуществующего метода приводит к фатальной ошибке. Рассмотрим пример:

```
class User {
    private $_passwordHash;
    public function changePassword($pwd) {
        $this->_passwordHash = md5($pwd);
    }
    public function authorize($pwd) {
        return $this->_passwordHash === md5($pwd);
    }
}
$u = new User();
$u->changePassword('123');
//$u->nonexistentMethod(); будет Fatal error
```

Как и в других С-подобных объектно-ориентированных языках в PHP методы и свойства могут быть статическими. Для обращения к ним вместо объектной ссылки используется имя класса, которое отделяется от имени свойства или метода двойным двоеточием.[16] Внутри класса вместо имени можно использовать ключевое слово **self**.

```
class User {
    // ранее представленные свойства
    private static $_instances = [];
    // ранее представленные методы
    public static function forId($id) {
        $user = @self::$_instances[$id];
        if (empty($user)) {
            // имитация загрузки из БД
            $user = new self();
            switch ($id) {
                case 101: $user->changePassword('123');
                    break;
                case 102: $user->changePassword('321');
                    break;
                default: return null;
            }
            self::$_instances[$id] = $user;
```

[16] В PHP двойное двоеточие обозначается как paamayim nekudotayim, что является названием двойного двоеточия на иврите (язык PHP был создан израильскими программистами Энди Гутмансом и Зеевом Сураски).

```
        }
        return $user;
    }
}
// за пределами класса User
$u = User::forId(101);
```

Константы задаются ключевым словом **const** и всегда общедоступны (явно указать область видимости для них невозможно).

```
const DEFAULT_SERVER = 'example.ru';
```

Класс может иметь конструктор, т. е. метод, который автоматически вызывается при создании нового объекта и обычно используется для инициализации его свойств. В PHP конструктор должен называться `__construct`. При вызове он получает в качестве параметров значения выражений, указанных в операторе **new**. Пример:

```
class Album {
    private $_title;
    private $_artist;
    public function __construct($title, $artist) {
        $this->setTitle($title);
        $this->setArtist($artist);
    }
    // методы setTitle и setArtist
    // для установки соответствующих свойств
}
$a = new Album('Angels Fall First', 'Nightwish');
```

Конструктор представляет собой один из магических методов. В PHP магическими называют методы, которые автоматически вызываются в определенные моменты жизненного цикла объекта. Названия всех магических методов начинаются с удвоенного подчеркивания; по соглашению двойное подчеркивание не должно использоваться в качестве префикса обычных методов.

При присваивании объектной ссылки копируется только сама ссылка, в то время, как копирования состояния объекта (т. е. значений его свойств) не происходит. Пример:

```
$u = new User();
$u->changePassword('123');
$u->authorize('123'); // вернет true
$v = $u;
$u->changePassword('321');
```

```
$v->authorize('123'); // вернет false
```

Для каждого объекта поддерживается счетчик ссылок, представляющий собой количество элементов данных (переменных, элементов массивов и свойств объектов), хранящих ссылку на данный объект. Счетчик ссылок изменяется при присваивании, а также при удалении элемента данных. Когда счетчик ссылок достигает нуля, объект автоматически удаляется. В то же время, возможности явно удалить объект, счетчик ссылок которого отличен от нуля, нет. Для задания действий, выполняемых в момент удаления объекта, можно определить в соответствующем классе метод `__destruct`; этот метод не получает никаких аргументов и не должен возвращать никакого значения.

По умолчанию в ходе выполнения программы сборка мусора на основе анализа достижимости не производится. Из-за этого может возникнуть ситуация, при которой из-за наличия циклических ссылок более недоступные объекты не будут удалены вплоть до окончания работы программы. Пример:

```
class Undestructible {
    private $_me;
    public function __construct() {
        $this->_me = $this; // сделаем циклическую ссылку
    }
    public function __destruct() {
        echo 'Destructing!';
    }
}
$u = new Undestructible();
$u = null;
// сообщение Destructing появится
// только после выполнения всей программы.
```

Поскольку обычно программы на PHP должны выполняться в течение небольшого времени, появление мусора не считается серьезной проблемой. Если же некоторая программа предназначена для продолжительного выполнения (это может быть программа, запускаемая из командной строки), то выполнить сборку мусора можно, воспользовавшись функцией `gc_collect_cycles`, не принимающей параметров. Можно также включать и выключать автоматическую сборку мусора, для чего используются функции `gc_enable` и

gc_disable соответственно; эти функции также не принимают параметров.

Иногда копирования ссылки на объект недостаточно, поскольку необходима копия целого объекта. В этом случае можно использовать операцию клонирования, задаваемую ключевым словом clone. При клонировании конструктор объекта не вызывается. Пример:

```
$a1 = new Album('Angels Fall First', 'Nightwish');
$a2 = clone $a1;
$a2->setTitle('Oceanborn');
echo $a1->getTitle(); // выведет Angels Fall First
echo $a2->getTitle(); // выведет Oceanborn
```

Собственно клонирование выполняется путем копирования в новый объект всех свойств, определенных в исходном объекте. В следствие этого по умолчанию клонирование является неглубоким (shallow), т. е. если значением некоторого свойства является объект, то этот объект не клонируется. В том случае, если некоторый объект с логической точки зрения владеет какими-либо из объектов, хранящихся в его свойствах, при его клонировании следовало бы клонировать также и объекты, находящиеся в его владении. Этого можно достичь с помощью магического метода __clone, который вызывается для нового объекта сразу после клонирования. Этому методу не передается никаких аргументов и он не должен возвращать никакого значения. Пример:

```
class Album {
    private $_title;
    private $_artist;
    private $_tracks = [];
    // конструктор и соответствующие методы…
    public function addTrack($track) {
        $this->_tracks[$track->getNumber()] = $track;
    }
    public function __clone() {
        foreach ($this->_tracks as &$track) {
            $track = clone $track;
        }
    }
}
class Track {
    private $_number;
```

```
    private $_title;
    private $_duration;
    // конструктор и соответствующие методы…
}
$a1 = new Album('Angels Fall First', 'Nightwish');
$a1->addTrack(new Track(1, 'Elvenpath', 280));
$a2 = clone $a1;
$a2->getTrack(1)->setTitle('Stargazers');
echo $a1->getTrack(1)->getTitle(); // выведет Elvenpath
echo $a2->getTrack(1)->getTitle(); // выведет Stargazers
```

Метод `__clone` можно использовать для запрещения клонирования. Пример:

```
class Unique {
    public function __clone() {
        trigger_error('Don\'t clone this object',
                      E_USER_ERROR);
    }
}
```

Подобно косвенному вызову функций в PHP поддерживается косвенное обращение к свойствам и косвенный вызов методов, а также косвенное создание объектов. Во всех этих случаях на месте имени свойства, метода или класса указывается переменная, содержащая строку с соответствующим именем. При косвенном обращении к свойству и при косвенном вызове метода можно также вместо переменной записать заключенное в фигурные скобки выражение, значением которого должно быть имя свойства или метода соответственно. Как уже отмечалось, косвенные операции часто могут заменить оператор выбора. Пример:

```
class IndexPage {
    public function doGet() {
        echo 'You are getting the index page...';
    }
    public function doPost() {
        echo 'You are posting something'
             . ' to the index page...';
    }
}
$pageClass = ucfirst(@$_GET['p'] ?: 'index') . 'Page';
// по умолчанию показываем страницу index
if (!class_exists($pageClass)) { throw new
PageNotFoundException(); }
```

```
$page = new $pageClass();
$page->{'do' .
ucfirst(strtolower($_SERVER['REQUEST_METHOD']))}();
```

Методы классов (в том числе статические) можно использовать в качестве функций. Для этого необходимо создать массив, в котором по ключу 0 будет храниться ссылка на объект (если нужно вызвать обычный метод), либо строка с именем класса (если нужно вызвать статический метод), а по ключу 1 — строка с именем метода. Такой массив можно использовать точно так же, как имя функции или λ-функцию. Пример:

```
$album = new Album('Angels Fall First', 'Nightwish');
$getAlbumTitle = [$album, 'getTitle'];
echo $getAlbumTitle();   // выведет Angels Fall First
call_user_func_array([$album, 'setTitle'], ['Oceanborn']);
echo $album->getTitle(); // выведет Oceanborn
```

Отметим, что простого способа превратить в функцию создание объекта нет, что создает сложности при необходимости создавать объекты, передавая в конструкторы соответствующих классов переменное количество параметров.

Магические методы `__get`, `__set`, `__isset` и `__unset` вызываются при попытке обратиться к несуществующему свойству (принимается во внимание фактическое существование свойства в объекте, а не наличие объявления свойства в определении класса). Все четыре метода первым параметром принимают имя свойства. Метод `__get` вызывается при попытке получения значения и должен возвратить некоторое значение; метод `__set` вызывается при попытке установки значения и принимает устанавливаемое значение в качестве второго параметра; метод `__isset` вызывается для проверки существования свойства; наконец, метод `__unset` вызывается для удаления свойства. Пример:

```
// «ленивая» инициализация свойств на основе
// заданного массива фабрик.
class Lazy {
    private $_factories; // фабрики
    public function __construct(array $factories) {
        $this->_factories = $factories;
    }
    public function __get($name) {
        $factory = @$this->_factories[$name];
```

```
        if (!is_callable($factory)) {
            trigger_error("No valid factory for $name",
                          E_USER_NOTICE);
            return null; // вернем что-нибудь…
        }
        return $this->$name = $factory();
    }
    public function __isset($name) {
        return is_callable(@$this->_factories[$name]);
    }
}
$lazy = new Lazy([
    'connection' => function () {
            return new PDO('mysql:host=localhost',
                           'root', '123'); }]);
$connection = $lazy->connection;
// фактическое создание соединения с БД
```

Магические методы `__call` и `__callStatic` вызываются при попытке вызвать несуществующий метод. Оба метода принимают два параметра: первый параметр задает имя вызванного метода, а второй — массив значений переданных параметров. Метод `__callStatic` отличается от метода `__call` тем, что вызывается в случае вызова неопределенного статического метода; разумеется, сам метод `__callStatic` также должен быть статическим. Пример:

```
class Guard { // реализует шаблон Декоратор
    private $_guarded; // охраняемый объект
    private $_rights;  // разрешенные методы
    public function __construct($guarded, array $rights) {
        assert(is_object($guarded));
        $this->_guarded = $guarded;
        $this->_rights = $rights;
    }
    public function __call($name, array $arguments) {
        $this->_validateCall($name);
        return call_user_func_array([$this->_guarded,
                $name], $arguments);
    }
    private function _validateCall($name) {
        // проверяет, разрешен ли вызов
        if (!@$this->_rights[$name]) {
            trigger_error("It's forbidden to call $name",
                          E_USER_ERROR);
        }
    }
```

```
}
class Account {
    private $_balance = 0;
    public function __construct($balance = 0) {
        $this->_balance = $balance;
    }
    public function getBalance() {
        return $this->_balance;
    }
    public function withdraw($amount) {
        $this->_balance -= $amount;
    }
    public function deposit($amount) {
        $this->_balance += $amount;
    }
}
$a = new Account(100);
$a->deposit(50);
$g = new Guard($a, ['getBalance' => true]);
echo $g->getBalance();
// $g->deposit(200); будет Fatal error
```

Метод `__invoke` позволяет вызывать объект как функцию.

3.12. Иерархии классов

Включение в один класс всех свойств и методов другого класса называется *наследованием*; при этом включающий класс называют *производным*, а включаемый — *базовым*. В производном классе можно заново определить один или несколько методов, определенных в базовом классе; такое повторное определение называется *замещением*.

Наследование представляет собой вид отношения между классами. Класс называется *предком* данного класса, если он является его базовым классом либо предком его базового класса и, наоборот, класс называется потомком данного класса, если он является его производным классом либо потомком его производного класса. Говорят, что совокупность классов, связанных между собой отношениями наследования, называется *иерархией*. Основным критерием, принимаемым во внимание при введении между двумя классами отношения наследования, является связанность выражаемых этими классами понятий по принципу «общее–частное».

Объектная модель языка PHP, будучи, как отмечалось выше, во многом позаимствована у языка Java, поддерживает только *единичное наследование классов*. Это означает, что у каждого класса может быть не более одного *базового* класса. В определении класса для задания базового класса служит ключевое слово **extends**, за которым следует имя базового класса. Если конструкция **extends** пропущена, то соответствующий класс не наследует ни от какого класса (этим PHP отличается от Java: в последнем при отсутствии явно заданного базового класса в качестве такового принимается класс `Object`, являющийся благодаря этому предком всех остальных классов). Пример:

```
class Product {
    private $_name;
    private $_price;
    public function getName() {
        return $this->_name;
    }
    public function setName($name) {
        $this->_name = $name;
    }
    // методы getPrice и setPrice
}
class Book extends Product {
    private $_pageCount;
    public function getPageCount() {
        return $this->_pageCount;
    }
    public function setPageCount($pageCount) {
        $this->_pageCount = $pageCount;
    }
}
$book = new Book();
$book->setName('Advanced PHP Programming');
$book->setPrice(37.26);
$book->setPageCount(672);
```

Для вызова методов, определенных в базовом классе, используется ключевое слово **parent**, после которого записывается двойное двоеточие и название соответствующего метода.

В отличие от C++ и Java в PHP конструктор ничем, кроме названия, не отличается от обычных методов. В частности, так же, как и другие методы конструктор наследуется производными классами

от базового; это позволяет во многих случаях избежать повторного определения конструктора в производном классе. С другой стороны, тело конструктора не включает неявно вызова конструктора базового класса; поэтому, как правило, в конструкторе производного класса требуется явный вызов унаследованного конструктора. Пример:

```
class Product {
    public function __construct($name, $price) {
        $this->setName($name);
        $this->setPrice($price);
    }
    // все свойтсва и остальные методы
    // как в предыдущем примере
}
class Book extends Product {
    public function __construct($name, $price,
                                $pageCount) {
        parent::__construct($name, $price);
        $this->setPageCount($pageCount);
    }
    // все свойтсва и остальные методы
    // как в предыдущем примере
}
```

Если некоторый класс должен использоваться только в качестве предка для других классов и не должен иметь собственных объектов, то такой класс можно объявить абстрактным. Если некоторый метод класса таков, что обязательно должен быть замещен в каждом из потомков (не считая тех потомков, которые сами являются абстрактными классами), то такой метод подобно классу можно объявить абстрактным, а тело метода опустить (т. е. написать вместо него точку с запятой). Поскольку абстрактный метод подлежит обязательному замещению, класс, содержащий хотя бы один абстрактный метод, сам должен быть абстрактным. Для объявления классов и методов абстрактными в самом начале их определения записывается ключевое слово **abstract.** В PHP есть интересная особенность: абстрактные методы могут быть статическими. Наконец, отметим, что при замещении абстрактного метода набор параметров замещающего метода должен совпадать с таковым у замещающего (в случае, если замещаемый метод не абстрактный, совпадения наборов параметров не требуется).

Помимо того, чтобы сделать наследование класса и замещение метода обязательными, можно их наоборот запретить; для этого следует объявить соответственно класс или метод конечными, воспользовавшись для этого ключевым словом **final**, записываемым подобно **abstract** в начале соответствующего определения. Ни класс ни метод не могут быть одновременно абстрактными и конечными. Пример:

```
abstract class Predicate {
    private $_col;
    public final function toSql(&$p) {
        return "$this->_col {$this->_condToSql($p)}";
    }
    protected function __construct($c) {
        $this->_col = $c;
    }
    protected abstract function _condToSql(&$p);
}
abstract class SimplePredicate extends Predicate {
    private $_val;
    public function __construct($c, $v) {
        parent::__construct($c);
        $this->_val = $v;
    }
    protected final function _condToSql(&$p) {
        $p[] = $this->_val;
        return "{$this->_opToSql()} ?";
    }
    protected abstract function _opToSql();
}
final class GreaterThanPredicate extends SimplePredicate {
    protected function _opToSql() { return '>'; }
}
abstract class InversiblePredicate extends Predicate {
    private $_inv;
    protected function __construct($c, $i) {
        parent::__construct($c);
        $this->_inv = $i; }
    protected final function _condToSql(&$p) {
        $s = $this->_coreCondToSql($p);
        return $this->_inv ? "NOT $s" : $s;
    }
    protected abstract function _coreCondToSql(&$p);
}
final class BetweenPredicate extends InversiblePredicate {
```

```
    private $_lBnd, $_uBnd;
    public function __construct($c, $lb, $ub, $i = false) {
        parent::__construct($c, $i);
        $this->_lBnd = $lb;
        $this->_uBnd = $ub;
    }
    protected function _coreCondToSql(&$p) {
        $p[] = $this->_lBnd;
        $p[] = $this->_uBnd;
        return 'BETWEEN ? AND ?';
    }
}
final class LikePredicate extends InversiblePredicate {
    private $_pat;
    public function __construct($c, $p, $i = false) {
        parent::__construct($c, $i);
        $this->_pat = $p;
    }
    protected function _coreCondToSql(&$p) {
        $p[] = $this->_pat;
        return 'LIKE ?';
    }
}
$predicates = [];
$predicates[] = new LikePredicate('title', '%Z%');
$predicates[] = new BetweenPredicate('year',
        2000, 2010, true);
$predicates[] = new GreaterThanPredicate('budget',
        1000000);
$parameters = [];
echo implode(' AND ', array_map(function (Predicate $p)
                                    use (&$parameters) {
    return $p->toSql($parameters);
}, $predicates));
// выведет
// title LIKE ? AND year NOT BETWEEN ? AND ? AND budget > ?
var_export($parameters);
// выведет [0 => '%Z%', 1 => 2000, 2 => 2010, 3 => 1000000]
```

Подобно языку Java в PHP поддерживаются интерфейсы. Интерфейс является дальнейшим развитием концепции абстрактного класса и представляет собой по сути набор абстрактных методов, которые должны быть замещены в каждом классе, реализующем данный интерфейс. Для интерфейсов поддерживается множественное наследование; это значит, что у каждого интерфейса может

быть произвольное количество базовых интерфейсов. В свою очередь, класс также может реализовывать произвольное количество интерфейсов. Объявление интерфейса отличается от объявления класса заменой ключевым словом **interface** ключевого слова **class**, а также тем, что интерфейс не может содержать свойств, а все его методы — абстрактные и общедоступные (хотя ключевое слово **abstract** в описании интерфейса не используется). Так же, как абстрактные методы классов, методы интерфейсов могут быть статическими. В объявлении интерфейса базовые интерфейсы перечисляются после ключевого слова **extends**, а в реализуемые интерфейсы в объявлении класса — после ключевого слова **implements**. Пример:

```
interface Logger {
    const INFO = 1, WARN = 2, ERR = 3;
    public function log($lvl, $msg);
}
interface Loggable {
    public function getLogger();
    public function setLogger(Logger $l);
}
class Authenticator implements Loggable {
    private $_logger;
    public function authenticate($usr, $pwd) {
        $r = $this->_checkPassword($usr, $pwd)
                or $this->getLogger()->log(Logger::INFO,
                        "Auth. failed for $usr.");
        return $r;
    }
    public function getLogger() {
        return $this->_logger;
    }
    public function setLogger(Logger $l) {
        $this->_logger = $l;
        return $this;
    }
    private function _checkPassword($usr, $pwd) {
        // каким-то образом проверяет пароль
    }
}
class Mailer implements Loggable {
    private $_account;
    private $_logger;
    public function __construct($a) {
```

```
        $this->_account = $a;
    }
    public function mail($subj, $recip, $content,
                         $type = 'text/plain') {
        $this->getLogger()->log(Logger::INFO,
                "Sending a mail to $recip.");
        // каким-то образом отправляет почту
    }
    public function getLogger() {
        return $this->_logger;
    }
    public function setLogger(Logger $l) {
        $this->_logger = $l;
        return $this;
    }
}
class EchoLogger extends Logger {
    public function log($lvl, $msg) {
        echo "[{$this->_levelToString($lvl)}] $msg\n";
        // простейший реализация
    }
    public static function _levelToString($lvl) {
        static $strings = [self::INFO => 'Info',
                           self::WARN => 'Warning',
                           self::ERR => 'Error'];
        return $strings[$lvl];
    }
}
$logger = new EchoLogger();
$a = new Authenticator();
$a->setLogger($logger);
$a->authenticate('vasya', '123');
// будет выведено [Info] Authentication failed for vasya.
$user = ['name' => 'Vasya', 'email' => 'vasya@example.ru'];
$password = uniqid();
$m = new Mailer('our-site@example.ru');
$m->setLogger($logger);
$m->mail('Recovering password', $user['email'], <<<END
Hello, $user[name]!
Your new password: $password
END
); // будет выведено Sending a mail to vasya@example.ru.
```

Необходимость замещать все методы интерфейсов часто приводит к появлению повторяющегося кода. Избавиться от повторяющегося кода в классе могут помочь *особенности*. Особен-

ность — это набор элементов класса.[17] В отличие от интерфейсов особенности могут содержать свойства, а методы могут не быть абстрактными. Класс может включать произвольное количество особенностей. Между особенностями не может быть отношения наследования, однако особенность может подобно классу включать произвольное количество других особенностей. Объявление особенности отличается от объявления класса только заменой ключевым словом **trait** ключевого слова **class** и отсутствие конструкций **extends** и **implements**. Подобно интерфейсам и абстрактным классам особенности не могут использоваться для создания объектов. Во включающем классе или в другой особенности имена включаемых особенностей записываются внутри тела (т. е. внутри фигурных скобок) после ключевого слова **use** (в одном классе может быть несколько конструкций **use**). Пример:

```
trait LoggableSupport {
    private $_logger;
    public function getLogger() {
        return $this->_logger;
    }
    public function setLogger(Logger $l) {
        $this->_logger = $l;
        return $this;
    }
    protected function log($lvl, $msg) {
        $this->_logger->log($lvl, $msg);
    }
}
class Mailer implements Loggable {
    use LoggableSupport;
    private $_account;
    public function __construct($account) {
        $this->_account = $account;
    }
    public function mail($subj, $recip, $content,
                         $type = 'text/plain') {
        $this->log(Logger::INFO,
                   "Sending a mail to $recip.");
        // каким-то образом отправляет почту
    }
```

[17] Среди других языков программирования наиболее близкой аналогией являются модули в языке Ruby.

```
}
```

Особенности не являются полноценной заменой интерфейсам, однако их удобно использовать в качестве готовой реализации интерфейсов. В определении особенности ключевое слово self задает не саму особенность, а класс, в который особенность включена. Пример:

```
trait Singleton {
    private static $_instance = null;
    public static function getInstance() {
        if (!self::$_instance) {
            self::$_instance = new self();
        }
        return self::$_instance;
    }
}
final class CreditCardGateway {
    use Singleton;
    public function authorize($number) {
        /* как-то авторизует кредитную карту */
    }
}
CreditCardGateway::getInstance()->authorize('123');
```

Разумеется особенности вполне могут содержать магические методы; это, например, позволяет реализовать в PHP настоящие свойства. Пример:

```
trait MagicProperties {
    public function __get($name) {
        return $this->{'get' . ucfirst($name)}();
    }
    public function __set($name, $value) {
        $this->{'set' . ucfirst($name)}($value);
    }
}
class Album {
    use MagicProperties;
    // все свойства и методы как в предыдущем примере
}
$a = new Album('Oceanborn', 'Nightwish');
echo "$a1->title\n"; // выведет Oceanborn
```

Отметим интересную черту особенностей: в их методах видны частные свойства и методы включающего класса или особенности.

Кроме встроенной переменной **$this**, представляющей текущий объект, в PHP определено ключевое слово **static**, представляющее текущей класс, а также не имеющая параметров функция get_called_class, возвращающая имя текущего класса в виде строки. Ключевое слово **static** и функция get_called_class отличаются от ключевого слова **self** и вызванной без параметров функции get_class тем, что возвращают не тот класс, в пределах определения которого находится соответствующий фрагмент кода, а тот, которому принадлежит текущий объект либо, в случае вызова статического метода, тот, который указан в вызове метода перед двойным двоеточием. Пример:

```
abstract class ActiveRecord {
    public static function select(PDO $c) {
        $s = $c->query(self::_makeSelectSql());
        return array_map(['static', '_load'],
                         $s->fetchAll());
    }
    private static function _makeSelectSql() {
        $tn = static::$_tableName;
        return "SELECT * FROM $tn";
    }
    private static function _load($r) {
        $o = new static();
        foreach (static::$_columnNames as $p => $c) {
            $o->$p = $r[$c];
        }
        return $o;
    }
}
class Album extends ActiveRecord {
    protected static $_tableName = 'album';
    protected static $_columnNames =
            ['_title' => 'a_title',
             '_artist' => 'a_artist'];
    protected $_title;
    protected $_artist;
    public function getTitle() {
        return $this->_title;
    }
    public function getArtist() {
        return $this->_artist;
    }
}
```

```
$c = new PDO('mysql:host=localhost;dbname=album_data',
             'root', '123');
foreach (Album::select($c) as $a) {
    echo "[{$a->getArtist()}] {$a->getTitle()}";
}
// Предположим, что запрос
// SELECT a_title, a_album FROM album возвращает строки
// +------------------+-----------+
// | a_title          | a_artist  |
// +------------------+-----------+
// | Angels Fall First | Nightwish |
// | Oceanborn         | Nightwish |
// +------------------+-----------+
// тогда, вышеприведенный код выведет строки
// [Nightwish] Angels Fall First
// [Nightwish] Oceanborn
```

3.13. Работа с базами данных

Несмотря на то, что в PHP встроено большое количество функций, предназначенных для работы с базами данных различных производителей, стандартным способом является использование универсального интерфейса доступа к данным, представленного PDO (PHP Data Objects).

Взаимодействие с базой данных с использованием PDO начинается с установления соединения, для чего используется конструктор класса `PDO`, принимающий до четырех параметров, из которых первый, задающий имя источника данных, является обязательным. Имя источника данных задается строкой и в большинстве случаев следует формату URI, причем в качестве схемы используется имя драйвера, задающего используемую СУБД; формат остальной части имени источника данных зависит от драйвера и как правило включает параметры, задающие имя сервера и имя базы данных. Перечислим некоторые драйверы в соответствии с используемыми ими схемами:

- `mysql` — используется для серверной СУБД MySQL. Имя источника данных (остающееся после отбрасывания схемы и следующего за ней двоеточия) представляет собой набор пар, каждая из которых состоит из имени и значения параметра соединения. В составе пары имя отделяется от значения знаком равенства; пары разделяются между собой точкой с запятой. Поддерживаются сле-

дующие параметры (все необязательные): `dbname` — имя базы данных; `charset` — имя набора символов, например `utf8` (следует указывать тот набор символов, который используется для вывода страницы); `host` — имя сервера, по умолчанию устанавливается соединение с локальным компьютером; `port` — номер порта сервера, по умолчанию используется порт 3306.

• `pgsql` — используется для серверной СУБД PostgreSQL. Формат имени источника данных отличается от такового у драйвера MySQL только тем, что параметр `charset` не поддерживается. В том случае, если набор символов, используемый в базе данных, не совпадает с тем, который используется для вывода страницы, полученные строки придется перекодировать, воспользовавшись, например, функцией `mb_convert_encoding`;

• `sqlite` — используется для встроенной СУБД SQLite. Имя источника данных представляет собой полное имя файла базы данных либо ключевое слово `:memory:` для создания временной базы данных в оперативной памяти.

Если для доступа к базе данных требуется указание имени учетной записи и пароля, то эти сведения задаются соответственно вторым и третьим параметрами конструктора. Четвертым параметром можно передать в конструктор массив, задающий опции используемого драйвера (ключи массива должны соответствовать названиям опций). Пример:

```
// соединение с сервером MySQL
$c = new PDO('mysql:dbname=album_data;charset=utf8',
'album_site', '123');
// соединение с сервером PostgreSQL
$c = new
PDO('pgsql:host=localhost;port=5432;dbname=album_data',
'album_site', '123');
// «соединение» с SQLite
$c = new PDO('sqlite:D:\\Data\\Albums.db');
// соединение с сервером MySQL с задание опции,
// заставляющей выбрасывать исключение при каждой ошибке;
// по умолчанию всего лишь возвращается false
$c = new PDO('mysql:dbname=album_data;charset=utf8',
'album_site', '123',
             [PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION]);
```

В том случае, если установить соединение не удается, конструктор класса `PDO` выбрасывает исключение `PDOException`.

После установления соединения программа должна создать оператор; для этого используется метод `prepare` класса `PDO`. Этот метод принимает до двух параметров, первый из которых задает текст запроса на языке SQL и является обязательным. Текст запроса может содержать метки параметров запроса, причем допускаются как позиционные параметры, отмечаемые знаком вопроса, так и именованные, отмечаемые двоеточием за которым следует имя параметра. Второй параметр метода `prepare` можно использовать для задания массива опций. В случае успеха метод возвращает объект класса `PDOStatement`; в противном случае по умолчанию возвращается **`false`**.

Для выполнения подготовленного оператора используется метод `execute` класса `PDOStatement`. Этот метод может принимать один необязательный параметр, который, если присутствует, должен быть массивом, задающим фактические значения параметров запроса. Те значения массива, которые соответствуют числовым ключам, задают значения позиционных параметров (индексация начинается с нуля), а те, которые соответствуют строковым ключам — значения именованных параметров. В случае успеха метод `execute` возвращает **`true`**; в противном случае по умолчанию возвращается **`false`**. Пример:

```
$s = $c->prepare('DELETE FROM album WHERE a_id = ?');
$s->execute([101]);
```

В том, случае, когда запрос не содержит параметров для создания оператора вместо метода `prepare` можно использовать метод `query`, который так же, как метод `prepare`, принимает в качестве параметра строку и возвращает объект класса `PDOStatement`, отличаясь, однако, тем, что возвращаемый методом `query` оператор является уже выполненным.

Если в результате выполнения оператора было сформировано результирующее множество, то выбрать все его строки можно с использованием оператора `foreach`, указав оператор перед ключевым словом **`as`**. Для доступа к очередной строке в теле цикла следует использовать переменную, указанную в операторе **`foreach`** после

ключевого слова **as**. По умолчанию строка представляется массивом, в котором каждому столбцу результирующего множества соответствует два элемента: один с числовым ключом, равным порядковому номеру столбца (считая с нуля), а другой со строковым ключом, представляющим собой имя столбца; значения обоих элементов равны значению, находящемуся в строке результирующего множества в соответствующем столбце. Пример:

```
$s = $c->prepare('SELECT * FROM album'
                 . ' WHERE a_year > :year');
$s->execute(['year' => 2000]);
foreach ($s as $row) {
    echo "[$row[a_artist]] $row[a_title]\n";
}
```

Способ представления выбираемых строк можно изменить воспользовавшись методом `setFetchMode`, принимающим до трех параметров, первый из которых обязательный и задает режим выборки с использованием одной из следующих констант (все они определены в классе `PDO`):

- `FETCH_ASSOC`, `FETCH_NUM` и `FETCH_BOTH` — строка представляется массивом, в котором значения доступны соответственно по строковым ключам, представляющим собой имена столбцов, числовым ключам, представляющим собой порядковые номера столбцов либо одновременно по строковым и числовым ключам;
- `FETCH_OBJ`, `FETCH_CLASS` и `FETCH_INTO` — строка представляется объектом, в котором значения доступны как одноименные столбцам свойства. Между собой эти режимы различаются следующем: при использовании режимов `FETCH_OBJ` и `FETCH_CLASS` для каждой строки создается новый объект, причем в первом случае используется предопределенный класс `stdClass`, а во втором — класс, имя которого задано вторым параметром метода `setFetchMode` (в том случае, когда конструктор класса требует параметры, их можно задать третьим параметром, поместив в массив); при использовании режима `FETCH_INTO` для всех строк повторно используется один и тот же объект, также задаваемый вторым параметром метода `setFetchMode`;

• FETCH_COLUMN — строка представляется единственным значением выбираемым из столбца, номер которого задан вторым параметром метода setFetchMode;

• FETCH_BOUND — в случае успешной выборки возвращается true. Этот режим предназначен для использования с методом bindColumn.

Пример:

```
$s = $c->query('SELECT a_title AS title'
                  . ', a_artist AS artist'
              . ' FROM album');
$s->setFetchMode(PDO::FETCH_CLASS, 'Album');
foreach ($statement as $row) {
    echo "[$row->artist] $row->title\n";
}
```

Вместо того, чтобы использовать для выборки строк цикл **foreach**, можно выбрать все строки сразу, воспользовавшись методом fetchAll. Этот метод возвращает массив и может принимать те же аргументы, что и метод setFetchMode (если метод fetchAll вызывается без параметров, то используется режим выборки, заданный последним вызовом setFetchMode либо режим выборки по умолчанию). Пример:

```
echo implode(PHP_EOL, $c
                ->query('SELECT a_title, a_artist'
                      . ' FROM album')
                ->fetchAll(PDO::FETCH_FUNC,
function($title, $artist) {
                            return "[$artist] $title";
                        }));
```

Может оказаться, что приложению требуется больший контроль за порядком выборки строк нежели тот, который предоставляет оператор **foreach** и метод fetchAll.

Если в результате выполнения запроса не было выбрано результирующего множества, но были вставлены, обновлены или удалены строки некоторой таблицы, то количество таких строк можно получить, вызвав метод rowCount класса PDOStatement; этот метод не принимает параметров.

Многие СУБД поддерживают автоматическую генерацию последовательных значений; обычно эта возможность используется

при вставке новых строк для присвоения значения столбцу, являющемуся первичным ключом. Для того, чтобы получить значение, которое было автоматически сгенерировано в результате выполнения запроса, следует использовать метод `lastInsertId` класса `PDO`, который может принимать один параметр. Если параметр опущен, то метод возвращает последнее значение, сгенерированное в результате вставки в таблицу, для которой в схеме базы данных задана автоматическая генерация. Если параметр присутствует, то он должен быть именем последовательности, определенной в схеме базы данных.

3.14. Пространства имен

Если исходный код приложения содержит большое количество классов и функций или же, если над ним совместно работают несколько разработчиков, то велика вероятность возникновения конфликта имен. Традиционным средством от этого является использование префиксов, добавляемых к именам классов и функций таким образом, что у тех классов и функций, которые принадлежат к одному архитектурному слою или модулю префикс одинаковый, а тех, которые принадлежат разным — разный. Префикс представляет собой неотъемлемую часть имени и поэтому не может быть опущен даже тогда, когда неоднозначности нет.

Более удобным способом выражения в именах структуры приложения является использование пространств имен. Подобно каталогам и подкаталогам они позволяют заменить плоский список имен классов и функций иерархическим. В пределах одного пространства имен классы и функции могут ссылаться друг на друга с использованием коротких имен, используя длинные только для обращения к классам и функциям, входящим в другие пространства имен.

В PHP элементами пространства имен могут быть функции, константы, классы, интерфейсы и особенности; глобальные переменные в пространства имен не входят. Имена самих пространств имен образуют иерархию, корнем которой является безымянное глобальное пространство имен.

Пространства имен объявляются в исходных файлах. Одно и то же пространство имен может быть объявлено в нескольких ис-

ходных файлах и, наоборот, в одном исходном файле может быть объявлено несколько пространств имен, причем одно и то же пространство имен в одном и тот же файле может быть объявлено несколько раз.

Для объявления пространства имен используется специальный оператор, причем если в исходном файле есть хотя бы одно такое объявление, то каждый оператор (кроме, быть может, **declare**) должен находится в пределах действия какого-либо из этих объявлений. Другим словами перед первым оператором объявления пространства имен других оператором (кроме **declare**) быть не должно.

Оператор объявления пространства имен может иметь две эквивалентные формы. Обе формы начинаются ключевым словом **namespace**, за которым следует имя пространства имен, для разделения компонентов которого используется обратная косая черта. В первой форме за именем пространства имен следует точка с запятой. Область действия такого оператора простирается до следующего оператора объявления пространства имен либо до конца файла. Пример:

```
namespace One;
class C { } // полное имя: One\C
namespace Another;
class C { } // полное имя: Another\C
namespace Yet\Another;
class C { } // полное имя: Yet\Another\C
```

Во второй форме за именем пространства имен следует открывающаяся фигурная скобка. В этом случае область действия оператора простирается до парной закрывающейся фигурной скобки. Вложенные объявления пространств имен не допускаются. Особенностью второй формы является то, что после ключевого слова **namespace** имя пространства имен можно опустить; в этом случае содержимое пространства имен будет помещено в глобальное пространство имен. В одном исходной файле можно использовать только одну форму оператора **namespace**. Пример:

```
namespace One {
    class C { } // полное имя: One\C
}
namespace Another {
```

```
    class C { } // полное имя: Another\C
}
namespace Yer\Another {
    class C { } // полное имя: Yet\Another\C
}
namespace {
    class C { } // полное имя: C
}
```

Область действия оператора объявления пространства имен распространяется только на файл, непосредственно содержащий соответствующее определение; в нее не входят файлы, включаемые операторами **require** и **include.**

В составе имени элемента имя пространства имен отделяется от собственного имени элемента обратной косой чертой (так же, как компоненты внутри имени самого пространства имен). По умолчанию имена элементов программы разрешаются относительно текущего пространства имен. Если же необходимо, чтобы имя разрешалось относительно глобального пространства имен, то необходимо поставить обратную наклонную черту в начале имени. Пример:

```
namespace One;
class C1 { }
namespace One\Two;
class C2 { }
namespace Another;
class C3 { }
namespace One;
$c1 = new C1();
$c2 = new Two\C2();
$c3 = new \Another\C3();
```

Избежать длинных имен позволяет оператор использования, который включает используемый элемент в текущее пространство имен. В простейшем случае этот оператор состоит из ключевого слова **use** и перечисления через запятую имен используемых элементов относительно глобального пространства имен (но без начальной наклонной черты), за которыми следует точка с запятой. При применении данной формы каждый используемый элемент включается под своим собственным именем. Область действия оператора использования ограничена областью действия вклю-

чающего его оператора **namespace**. Операторы **use** не могут располагаться в пределах других конструкций. Пример:

```
namespace One;
use One\Two\C2, Another\C3;
$c2 = new C2();
$c3 = new C3();
```

Оператор use позволяет переименовать используемые элементы. Для этого после имени элемента надо записать ключевое слово **as** и новое имя. Пример:

```
namespace One;
use One\Two\C2 as TC, Another\C3 as AC;
$c2 = new TC();
$c3 = new AC();
```

Отметим, что с помощью оператора **use** можно использовать не только элементы пространств имен, но и сами пространства имен.

Для ссылки на текущее пространство имен можно использовать ключевое слово **namespace** и предопределенную константу `__NAMESPACE__`.

3.15. Буферизация вывода

Буферизация вывода позволяет выводить HTML-фрагменты не в том порядке, в котором они должны оказаться в окончательно сформированной странице, а в том, который наилучшим образом соответствует логике приложения.

В PHP поддерживается многоуровневая буферизация, причем буферы организованы в виде стека. Если стек не пуст, то вывод производится в буфер, находящийся в вершине стека; в противном случае весь вывод направляется в стандартный выходной поток, что в случае web-приложения означает вывод тела отклика. Буферизация распространяется как на режим разметки, так и на операторы **echo** и **print**.

Чтобы поместить буфер в вершину стека используется функция `ob_start`, которую можно вызвать без параметров. Извлечь буфер из вершины стека можно воспользовавшись функцией `ob_get_clean` или `ob_get_flush`. Обе функции не принимают параметров и возвращают текущее содержание буфера в виде строки.

Функция `ob_get_flush` отличается от функции `ob_get_clean` тем, что записывает содержимое текущего буфера в нижележащий буфер либо в стандартный выходной поток, если текущий буфер единственный. Пример:

```
ob_start();
?>
<div>Some text</div>
<?
$div = ob_get_clean(); // будет присвоено значение
'<div>Some text</div>'
```

Если содержимое буфера не нужно, то для вместо `ob_get_clean` и `ob_get_flush` можно использовать `ob_end_clean` и `ob_end_flush`. Для очистки буфера без извлечения можно воспользоваться функциями `ob_clean` и `ob_flush`. Наконец, получить содержимое буфера или его длину позволяют функции `ob_get_contents` и `ob_get_length` соответственно.

Буферизацию часто используют для реализации кэширования.

Контрольные вопросы

1. Какие языки применяются для разработки Интернет-приложений?
2. Что описывает стандарт CGI?
3. Какова структура исходного файла на языке PHP?
4. Какие типы данных поддерживаются языком PHP?
5. В чем разница между строгим (===) и нестрогим (==) равенством?
6. Какие встроенные функции PHP используются для работы со строками?
7. Как объявить константу?
8. Какие управляющие конструкции поддерживаются языком PHP?
9. Что представляют собой массивы PHP как структуры данных?
10. Что такое литерал массива?
11. Как организовать цикл по элементам массива?
12. Как отсортировать массив?
13. Что такое ленивое копирование и каковы его преимущества?

14. Какие встроенные переменные используются для представления информации об HTTP-запросе?

15. Как определить функцию? Какие параметры называют необязательными?

16. В чем отличие локальных, глобальных и суперглобальных переменных?

17. Что такое локальная статическая переменная?

18. Что такое ссылка? Как организовать передачу параметров по ссылке?

19. Что такое косвенный вызов функции?

20. Что такое λ-функция? Как определить контекст λ-функции?

21. Какие встроенные функции PHP используются для чтения и записи файлов?

22. Какие предосторожности необходимо соблюдать при работе с файлами в Интернет-приложении?

23. Что называют упорядочением (serialization)?

24. Какие встроенные функции PHP используются для работы с файловой системой?

25. Как извлечь файлы, переданные в HTTP-запросе?

26. Что такое сессия? Как они реализуются в PHP?

27. Как подключить модуль? В чем разница между операторами **`require`** и **`include`**?

28. Какие элементы может включать определение класса?

29. Какие области видимости поддерживаются в PHP?

30. В чем особенность статических элементов класса?

31. Как создать экземпляр класса?

32. В чем разница между встроенной переменной **`$this`** и ключевым словом **`self`**?

33. Какие методы называют магическими? Для чего они используются?

34. Какие свойства и методы называют виртуальными?

35. Что называют клонированием объекта класса?

36. Что такое сборка мусора? Как она осуществляется в PHP?

37. Как создать объект класса по имени? Как обращаться к элементам класса по имени?

38. Как вызвать метод как функцию?

39. Как вызвать метод базового класса?

40. Какие методы называются абстрактными? Для чего они применяются?

41. Что такое интерфейс?

42. Что такое особенность (trait)?

43. В чем разница между ключевыми словами **`self`** и **`static`**?

44. Что такое пространство имен?

45. Как импортировать идентификаторы из пространства имен?

46. Что называют автозагрузкой классов? Как ее включить?

47. Как установить соединение с базой данных с использованием PDO? Что такое строка соединения?

48. Как выполнить SQL-запрос с использованием PDO?

49. Как извлечь результаты выполнения SQL-запроса?

50. Что такое буферизация вывода?

Глава 4. JavaScript

4.1. Общие сведения

Если PHP в своем классе является лишь одной из альтернатив, пусть и наиболее популярной, то JavaScript в классе языков программирования Web-страниц является стандартом, так же, как HTML является стандартом в классе языков разметки.

Подобно тому, как разные браузеры для отображения HTML используют разные HTML-движки, так же и для выполнения сценариев JavaScript разные браузеры используют разные виртуальные машины; например Firefox использует SpiderMonkey, а Google Chrome — V8. Разнообразие виртуальных машин порождает проблемы совместимости, подобные тем, которые существуют с HTML. Web-разработчик не может ориентироваться на какую-то одну виртуальную машину, поскольку это может создать проблемы для посетителей сайта, использующих браузеры, в которые встроены другие виртуальные машины; вместо этого web-разработчик обязан так программировать сценарии, чтобы они выполнялись одинаково во всех широко распространенных браузерах.

Для достижения единообразного поведения сценария в различных браузерах разработчикам приходится комбинировать два принципа:

- использовать только те возможности, которые одинаково реализованы во всех виртуальных машинах (использовать «наибольший общий делитель»);
- при необходимости воспользоваться возможностью, которой может не быть в какой-либо виртуальной машине, проверить тип браузера и предусмотреть обходной путь на тот случай, если окажется, что возможности действительно нет.

Все это существенно усложняет разработку надежных и эффективных сценариев, однако были разработаны библиотеки (например jQuery), скрывающие особенности браузеров за универсальным интерфейсом и тем самым помогающие прикладным разработчикам сосредоточиться на логике создаваемого приложения.

4.2. Сценарии

Фрагменты кода на языке JavaScript, выполняемые браузером, называют сценариями. Сценарии могут быть внешними и встроенными. Внешний сценарий размещается в отдельном ресурсе. Для его подключения к Web-странице используют пустой элемент `script`, который должен содержать атрибут `src`, задающий идентификатор ресурса. Внимание: несмотря на то, что при подключении внешнего сценария элемент `script` должен быть пустым, наличие закрывающего тега обязательно.

Встроенный сценарий размещается непосредственно в самой Web-странице в элементе `script` между открывающим и закрывающим тегом; атрибут `src` в этом случае не указывается.

Элемент `script` может располагаться как в заголовке страницы (т. е. в элементе `head`), так и в ее теле (т. е. в элементе `body`). Часто рекомендуют располагать элементы `script` в конце страницы, чтобы отображение страницы в окне браузера не задерживалось выполнением сценария.

Внешний сценарий обладает рядом преимуществ:

- его можно использовать в нескольких страницах;
- загрузка страниц ускоряется, если сценарий уже находится в кэше браузера (однако начальная загрузка может, наоборот, несколько замедлиться);
- отделение сценария от представления упрощает кодирование и верстку, поскольку код на JavaScript не мешает читать разметку на HTML и наоборот;
- упрощается распределение обязанностей в команде разработчиков.

Преимуществом встроенного сценария является возможность разместить его рядом с обрабатываемыми элементами страницы.

4.3. Выражения и операторы

В JavaScript поддерживаются следующие примитивные типы:

- `Number` — вещественное число двойной точности (т. е. 64-разрядное). Специального типа данных для целых чисел в JavaScript нет. Числовые литералы записываются по обычным для C-подобных языков правилам.

• `String` — строка символов. Литералы могут записываться с использованием как одинарных, так и двойных кавычек; в отличие от PHP интерполяция строк не поддерживается и никакой разницы между двумя видами кавычек нет. Внутри виртуальной машины для строк используется кодировка UTF-16.

• `Boolean` — логическое значение. Возможные значения — ложь и истина, представленные ключевыми словами **`false`** и **`true`**.

• `Null` — пустое значение, представленное ключевым словом **`null`**.

• `Undefined` — неопределенное значение, представленное ключевым словом **`undefined`**.

Тип значения можно получить, воспользовавшись унарным оператором **`typeof`**, возвращающим строку, представляющую собой название типа данных.

Интерпретация многих конструкций зависит от того, включен ли строгий режим. По умолчанию строгий режим выключен. Для его включения в начале программы или функции следует поместить строку `"use strict"`. Рекомендуется использовать строгий режим, поскольку в нем становится возможно раньше диагностировать многие ошибке.

В языке JavaScript переменные нуждаются в объявлении. Объявление переменной состоит из ключевого слова **`var`**, за которым следуют через запятую имена переменных, каждое из которых может сопровождаться инициализирующим выражением, отделяемым от имени переменной знаком равенства. Пример:

```
var z = 202;
```

Отметим, что объявление переменной вовсе не обязательно должно предшествовать ее первому использованию.

4.4. Объекты

Помимо примитивных типов в JavaScript поддерживается объектный тип; значением объектного типа является ссылка на объект. В JavaScript объект — это не более чем набор свойств. Каждое свойство представляет собой пару, состоящую из имени и значения. Имена свойств являются строками символов; значения могут быть произвольного (в том числе объектного) типа. Наиболее близкой

аналогией объектов языка JavaScript являются массивы языка PHP. Объект может быть задан объектным литералом. Этот литерал представляет собой последовательность определений свойств, разделенную запятыми и заключенную в фигурные скобки. Определение свойства может состоять из имени и значения, отделенных друг от друга двоеточием. Для задания имени можно использовать строковый или числовой литерал либо идентификатор (независимо от способа задания имя свойство хранится как строка символов).[18] Значение свойства может быть задано произвольным выражением. Пример:

```
var x = {p: 10, 2: 100, '-->': 1000};
// эквивалентно {'p': 10, '2': 100, '-->': 1000}
```

Для обращения к значению свойства объекта в JavaScript предусмотрено две формы. Если для задания имени свойства используется выражение, то оно должно записываться в квадратных скобках после выражения, задающего ссылку на объект. Если же имя задается идентификатором, то он отделяется от предшествующего выражения, задающего ссылку, точкой. Пример:

```
console.log(x.p);
console.log(x['p']); // то же, что предыдущее
console.log(x[2]);
```

Для организации цикла по свойствам объекта используется специальный оператор цикла: после ключевого слова **for** в скобках записывается выражение либо объявление переменной, задающее переменную, в которую перед каждым выполнением тела цикла будет помещаться имя свойства; затем пишется ключевое слово **in**, за которым следует выражение, задающее объектную ссылку. Пример:

```
for (var p in x) {
    console.log(x[p]);
    // последовательно выведет 10, 100 и 1000
}
```

Отметим, что, вообще говоря, оператор **for** перечисляет не все свойства объекта, а только те, для которых это разрешено.

---

[18] Будучи использован для задания имени свойства идентификатор интерпретируется как особая форма строкового литерала, а не как имя переменной.

4.5. Массивы

В JavaScript каждый массив — это объект. Массив может быть задан литералом массива. Пример:

```
var a = ['one', 'two', 'three'];
```

Также можно использовать конструктор массива `Array`. Если конструктор вызван с одним параметром, представляющим собой целое число, то это число трактуется как размер создаваемого массива; значения всех элементов в этом случае неопределенные. В противном случае параметры конструктора трактуются как элементы массива (как при использовании литерала). Пример:

```
var a2 = Array(3);    // [undefined, undefined, undefined]
var a3 = Array(1, 2); // [1, 2]
```

Длина массива представлена свойством `length`. Это свойство допускает присваивание. Если новая длина больше текущей, то массив дополняется справа неопределенными значениями; если же, наоборот, меньше текущей, то справа удаляется соответствующие лишние элементы. Пример:

```
a3.length = 4; // [1, 2, undefined, undefined]
a3.length = 1; // [1]
```

Элементы массива также представлены свойствами, причем в качестве имен используются индексы элементов, записанные в десятичной системе счисления. Индекс первого элемента равен нулю. Пример:

```
console.log(a3[0]); // выведет 1
```

4.6. Функции

Подобно массивам каждая функция — это тоже объект. Объявление функции начинается с ключевого слова **`function`**, за которым следует имя функции, список параметров в круглых скобках и тело в фигурных скобках. Для возвращения значения из функции используется оператор **`return`**. По умолчанию возвращаемое значение не определено. Пример:

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

```
console.log(cube(5)); // выведет 125
```

Результатом объявления функции является присваивание ссылки на объект, представляющий функцию переменной, имя которой совпадает с именем, указанным после ключевого слова **function**. Несмотря на это в пределах блока кода объявление функции может предшествовать использованию, поскольку все объявления функций обрабатываются до начала выполнения операторов.

В строгом режиме не разрешается объявлять функций внутри операторов, однако вложения объявлений функций друг в друга допустимы.

Внутри функции доступны все переменные включающего лексического контекста. Иными словами, во всякой функции доступны переменные, объявленные на верхнем уровне (т. е. вне какой-либо функции), кроме этого во вложенной функции доступны переменные всех включающих функций. Контекст, использовавшийся при создании объекта вложенной функции, не разрушается при возврате из включающей функции; благодаря этому вложенную функцию можно вызывать после завершения выполнения включающей. Пример:

```
function createCounter(step) {
    var currentValue = 0;
    return counter;
    function counter() {
        var value = currentValue;
        currentValue += step;
        return value;
    }
}
var c = createCounter(3);
console.log(c()); // выведет 0
console.log(c()); // выведет 3
console.log(c()); // выведет 6
```

Как и многие современные языки язык JavaScript поддерживает λ-функции, т. е. безымянные функции, определяемые в составе выражений. Синтаксис λ-функции отличается от синтаксиса объявления функции только тем, что имя функции необязательно. Как нетрудно догадаться, значением λ-выражения является объект, представляющий собой функцию. λ-функции часто используются для задания параметров других функций. Пример:

```
var a = [1, 2, 3].map(function (p) {
    return Math.pow(5, p);
}); // в a будет [5, 25, 125]
```

Вызов безымянной функции можно использовать для того, чтобы поместить некоторый код в отдельный лексический контекст. Пример:

```
(function () {
    var f = 1;
    for (var i = 2; i < 7; ++i) {
        f *= i;
    }
    console.log(f);
})();
// здесь переменных f и i уже нет
```

4.7. Методы

Как и любые другие значения, функции могут быть значениями свойств объектов. Если в операторе вызова функция задается значением свойства некоторого объекта, то этот объект доступен в теле функции с использованием ключевого слова **this**. Условимся называть такой вызов вызовом метода. Пример:

```
function distance2d(that) {
    return Math.sqrt(Math.pow(this.x - that.x, 2) +
Math.pow(this.y - that.y, 2));
}
var p = {x: 1, y: 2}, q = {x: 3, y: 4};
p.distance = distance2d;
console.log(p.distance(q));
```

Функцию можно вызывать как метод (т. е. определив значение **this**), даже если эта функция не является свойством объекта; для этого можно воспользоваться методом `call`, определенным в самом объекте функции. Этому методу передается произвольное количество параметров, причем первый из них задает значение **this**, а остальные —значения обычных параметров. Пример:

```
distance2d.call({x: 1, y: 2}, {x: 3, y: 4})
```

Иногда вместо метода `call` удобно использовать метод `apply`; он отличается от метода `call` только тем, что параметры функции передаются ему в виде массива.

Функция может использоваться для инициализации свойств нового объекта. Для этого используют конструктор, представляющий собой выражение, начинающееся с ключевого слова **new**, за которым следует вызов функции. Если функция вызвана в составе конструктора, то конструируемый объект будет доступен в ней с использованием ключевого слова **this**. Конструктор возвращает сконструированный объект. Пример:

```
function Point2d(x, y) {
    this.x = x;
    this.y = y;
}
var r = new Point2d(5, 6);
console.log(r);
```

Помимо явно определенных свойств каждый объект содержит ссылку на прототип. Прототип используется следующим образом: если при попытке прочитать некоторое свойство объекта обнаруживается, что такого свойства в объекте нет, то предпринимается попытка прочитать это же свойство в прототипе. Таким образом, прототип удобно использовать для хранения свойств, совместно используемых некоторым классом объектов. Обычно значениями свойств прототипа являются функции. Прототип объектов, создаваемых с использованием конструктора, представлен свойством prototype функции, использовавшейся при конструировании. Пример:

```
Point2d.prototype.distance = distance2d;
console.log(r.distance(q));
```

В JavaScript нет специальных конструкций, предназначенных для реализации наследования. Однако наследование вполне возможно реализовать на основе прототипов, определив прототип прототипа. В этом случае может оказаться полезной функция `Object.create`, создающая пустой объект с использованием прототипа, заданного первым параметром. Пример:

```
function Figure(x, y) {
    console.log('Creating a figure...');
    this.x = x;
    this.y = y;
}
Figure.prototype.draw = function () {
```

```
    console.log('Drawing a figure...');
};
function Circle(x, y, radius) {
    console.log('Creating a circle...');
    Figure.call(this, x, y);
    this.radius = radius;
}
Circle.prototype = Object.create(Figure.prototype);
Circle.prototype.draw = function () {
    Figure.prototype.draw.call(this);
    console.log('Drawing a circle...');
};
var c = new Circle();
c.draw();
```

4.8. DOM

Помимо объектов, встроенных в JavaScript, в распоряжении сценария обычно находятся объекты, предоставляемые средой, включающей сценарий. В случае сценария, подключенного к Web-странице, эта среда представлена окном браузера, отображающим эту страницу. Состав объектов, а также их свойств и методов (свойств, значениями которых являются функции), доступных в контексте окна браузера, определяется стандартом DOM, что расшифровывается как Document Object Model.

Согласно DOM верхний уровень представлен самим окном браузера. Для него определены следующие свойства:

- `document` — документ (web-страница), отображаемый в окне в данный момент. Свойства документа будут более подробно рассмотрены далее;
- `location` — содержимое строки адреса;
- `name` — строка с заголовком окна (или вкладки);
- `screen` — сведения о разрешении экрана.

Для выбора элемента документа по идентификатору (т. е. по значению атрибута `id`) используется метод `getElementById`; метод возвращает объект, представляющий элемент с заданным идентификатором, либо **`null`**, если искомого элемента в документе нет. Пример:

```
var resultDiv = document.getElementById('resultDiv');
```

Кроме идентификатора в качестве критерия для выбора можно использовать тэг и имя (значение атрибута `name`) или классы (значение атрибута `class`). В этом случае используются методы `getElementsByTagName`, `getElementsByName` и `getElementsByClassName`; все эти методы возвращают объект `NodeList`, представляющий список узлов. Количество узлов в списке хранится в свойстве `length`, а выбрать элемент по индексу можно воспользовавшись методом `item`. Также список узлов можно использовать как массив. Пример:

```
var nodes = document.getElementsByClassName('a');
for (var i = 0; i < nodes.length; ++i) {
    var e = nodes[i];
    // сделать что-то с элементом e
};
```

Поскольку в самом списке узлов нет метода `forEach` можно воспользоваться методом `forEach`, определенном для массивов. Пример:

```
Array.prototype.forEach.call(
    document.getElementsByClassName('a'),
    function (e) {
    // сделать что-то с элементом e
});
```

Наибольшие возможности предоставляют методы `querySelector` и `querySelectorAll`; в качестве параметра этим методам передается строка, задающая один или несколько CSS-селекторов, разделенных запятыми. Метод `querySelector` возвращает первый по порядку элемент, соответствующий хотя бы одному из селекторов, а метод `querySelectorAll` — все соответствующие элементы в виде объекта `NodeList`. Пример:

```
Array.prototype.forEach.call(
    document.querySelectorAll('div.a, div.b'),
    function (e) {
    // сделать что-то с элементом e
});
```

Методы `getElementById`, `getElementsByTagName`, `getElementsByClassName`, `querySelector` и `querySelectorAll` можно вызвать не только для объекта, представляющего весь документ,

но и для элементов документа; в этом случае поиск будет проводиться в соответствующем поддереве.

Кроме выбора элементов по некоторому критерию возможна навигацию по узлам дерева документа; при использовании навигации можно выбрать не только элементы, но и узлы других типов (например: текстовые узлы). Для навигации по узлам и их инспектирования используются следующие свойства:

- `firstChild` и `lastChild` — первый или последний по порядку дочерний узел данного узла. Если дочерних узлов нет, то значением этих свойств будет **`null`**;
- `nextSibling` и `previousSibling` — узел, следующий за данным или предшествующий ему либо **`null`**;
- `childNodes` — список дочерних узлов;
- `nodeType` — тип узла. Представлен числовыми константами `Node.ELEMENT_NODE` (элемент), `Node.TEXT_NODE` (текстовый узел), `Node.COMMENT_NODE` (комментарий) и т. д.;
- `nodeName` — имя узла. Для элементов в этом свойстве хранится тэг;
- `nodeValue` — значение узлов. Для элементов в этом свойстве храниться **`null`**, а для текстовых узлов — соответствующий текст. Присваивание нового значения данному свойству модифицирует документ и приводит к обновлению его представления в окне браузера;
- `textContent` — текст, содержащийся в узле без HTML-разметки. В отличие от предыдущего свойства для элементов в этом свойстве хранится текст дочерних текстовых узлов. Присваивание данному свойству также модифицирует документ.

Пример:

```
for (var n = document.getElementById('textDiv').firstChild;
         n; n = n.nextSibling) {
    if (Node.TEXT_NODE === n.nodeType) {
        n.nodeValue = n.nodeValue.toUpperCase();
    }
}
```

При работе с текстовыми узлами следует помнить, что браузер вовсе не обязан представлять весь подряд идущий текст одним тек-

стовым узлом; браузер вполне может разбить длинный (по его мнению) фрагмент текста между несколькими текстовыми узлами.

Список всех атрибутов узла хранится в свойстве `attributes`; этот список представляет собой объект, в котором атрибуты узла представлены одноименными свойствами. Каждый атрибут узла, в свою очередь, также представляет собой объект со свойствами `name` и `value`, в которых хранятся соответственно имя и значение атрибута. Также список атрибутов можно использовать как список узлов (т. е. в нем есть свойство `length` и поддерживается доступ к атрибутам по числовому индексу). Если у узла не может быть атрибутов, в свойстве `attributes` хранится **`null`**.

Некоторые атрибуты элементов представляются отдельными свойствами:

- идентификатор хранится в свойстве `id`;
- набор классов хранится в свойстве `classList`. Он представлен объектом, поддерживающим, в частности, следующие методы: `contains` — возвращает true, если заданный параметром класс присутствует в наборе, `add` и `remove` — соответственно добавляет в набор или удаляет из набора указанный класс, `toggle` — если указанный класс отсутствует в наборе, то добавляет его, иначе — удаляет;
- стиль хранится в свойстве `style`. Он также представлен объектом, каждое свойство которого соответствует CSS-свойству (имена CSS-свойств модифицируются, так, например, CSS-свойство `background-color` представляется в указанном объекте как свойство `backgroundColor`).

Модифицировать структуру дерева документа можно путем создания, добавления и удаления узлов. Для создания новых узлов используются следующие методы документа:

- `createElement` — создает новый элемент. Метод принимает один параметр, задающий тэг элемента, и возвращает объект, представляющий созданный элемент;
- `createTextNode` — создает новый текстовый узел. Содержимое узла задается первым параметром.

Методы создают лишь структуру данных `createElement` и `createTextNode`, не добавляя созданный узел в дерево документа.

Чтобы добавить узел в дерево документа (и сделать его видимым в окне браузера) используются следующие методы:

- `appendChild` — добавляет узел, заданный параметром, в конец списка дочерних узлов данного узла;
- `insertBefore` — вставляет узел, заданный параметром, перед данным узлом.

Пример:

```
var div1 = document.getElementById('div1');
for (var i = 0; i < 5; ++i) {
    var newButton = document.createElement('button');
    newButton.textContent = 'Button #' + i;
    newButton.number = i;
    newButton.addEventListener('click', function () {
        alert('Button #' + this.number + ' clicked!');
    });
    div1.appendChild(newButton);
}
```

Для удаления узла из списка дочерних узлов используется метод `removeChild`; удаляемый узел передается ему в качестве параметра. Также дочерний узел можно заменить; для этого используется метод `replaceChild`, которому передается два параметра: заменяемый узел и тот узел, которым его следует заменить.

Другой способ модифицировать структуру документа — использовать свойства `innerHTML` и `outerHTML`. Для элементов документа в этих свойствах хранится HTML-разметка в виде текстовой строки. Присваивание значений этим свойствам вызывает обновление представления документа в окне браузера. Свойства `innerHTML` и `outerHTML` отличаются друг от друга тем, что первое представляет разметку без тэгов самого элементов, а второе — с тэгами. Пример:

```
div1.innerHTML = '<em>Hello</em> <strong>World!</strong>';
```

4.9. События DOM

Помимо наличия свойств и методов многие объекты DOM являются источниками событий. Чтобы подключить к какому-либо объекту обработчик события для этого объекта можно вызвать метод `addEventListener`, передав ему первым параметром строку, задающую тип события, а вторым слушатель, т. е. функцию, которая

будет вызываться при наступлении события заданного типа. Пример:

```
var button = document.getElementById('helloButton');
button.addEventListener('click', function () {
    alert('Hello from ' + this.textContent);
});
```

Подключив слушатель к событию `onreadystatechange`, источником которого является документ, сценарий может задать действия, выполняемые в момент окончания загрузки страницы. Эта возможность весьма полезна, поскольку браузер выполняет сценарий сразу же, как только встречает его в тексте документа, что может привести к тому, что в момент выполнения сценария некоторые элементы могут оказаться еще незагруженными. Пример:

```
document.addEventListener('readystatechange', function () {
    if('complete' === document.readyState) {
        console.log('Initializing...');
    }
})
```

Альтернативный вариант — использовать событие `load`, источником которого является окно браузера.

4.10. AJAX

AJAX — это технология, позволяющая организовать обмен данными с web-сервером без перезагрузки всей страницы. AJAX расшифровывается как Asynchronous JavaScript and XML (асинхронный JavaScript и XML). Впрочем, в настоящее время термин AJAX часто предпочитают не расшифровывать и писать как Ajax, поскольку расшифровка не совсем точно отражает суть технологии; так, в частности, форматом данных, передаваемых с помощью этой технологии, как правило является на XML, а JSON или HTML.

Собственно обмен выполняется под управлением сценария с использованием объекта, создаваемого конструктором `XMLHttpRequest`,[19] представляющего запрос. Работа с запросом начинается с его открытия, выполняемого методом `open`, которому первыми двумя параметрами передаются строки, задающие соот-

[19] Однако иногда можно обойтись и без этого объекта.

ветственно HTTP-метод и идентификатор ресурса; последний может быть относительным. После открытия запроса можно добавить к нему HTTP-заголовки; для этого используется метод `setRequestHeader`, которому передаются две строки: имя заголовка и его значение. Поскольку обмен данными выполняется асинхронно (т. е. после окончания выполнения текущего сценария) необходимо задать обработчик события, который будет вызываться каждый раз при изменении состояния запроса; это можно сделать, присвоив соответствующее значение свойству `onreadystatechange` либо воспользовавшись методом `addEventListener` (указав в качестве первого параметра строку `readystatechange`, а в качестве второго — обработчик).

Собственно выполнение запроса начинается после вызова метода `send`, который можно вызвать как без параметров, так и с одним параметром, задающим тело запроса; тело запроса может быть представлено строкой, моделью документа, данными формы (используется объект `FormData`) либо двоичными данными (используются объекты `Blob` или `ArrayBuffer`). В ходе выполнения состояние запроса будет меняться, что приведет к вызовам обработчика события `onreadystatechange`. Текущее состояние запроса хранится в свойстве `readyState` и представлено целым числом; по окончании обмена значением этого свойства будет `XMLHttpRequest.DONE`.

После окончания обмена числовой код статуса будет храниться в свойстве `status`, а код, соответствующий статусу текст, — в свойстве `statusText`. Также можно получить заголовки отклика, воспользовавшись методом `getResponseHeader`, который принимает имя заголовка и возвращает его значение либо **`null`**.

Тело отклика храниться в свойстве `response`; по умолчанию оно представлено строкой. Тип данных, который будет использоваться для представления тела отклика можно изменить, если перед вызовом метода `send` задать значение свойства `responseType`; тип должен быть одной из следующих строк:

- `json` — тело будет представлено обычным объектом JavaScript. Требуется, чтобы исходное (полученное от web-сервера) тело отклика соответствовало спецификацией JSON;

• `document` — тело будет представлено моделью документа (объектом Document). Требуется, чтобы исходное тело отклика было в формате XML;

• `blob` или `arraybuffer` — тело будет представлено объектами `Blob` и `ArrayBuffer` соответственно. Этот вариант следует использовать, если тело отклика содержит бинарные данные;

• `text` или пустая строка — тело будет представлено строкой. Это вариант по умолчанию.

Пример:

```
document.getElementById('queryDataButton').addEventListener
('click', function () {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'data/data.json');
    xhr.responseType = 'json';
    xhr.addEventListener('readystatechange', function () {
        if (XMLHttpRequest.DONE === xhr.readyState) {
            if (200 === xhr.status) {
                // сделать что-то с полученными данными
            } else {
                // сообщить об ошибке
            }
        }
    });
    xhr.send();
});
```

Выполнение запроса может быть прервано вызовом метода `abort`.

Получить данные с сервера можно и без использования `XMLHttpRequest`. Для это можно использовать, например, элемент `script`. Пример:

```
// эта функция должна вызываться динамически загружаемыми
сценариями
function dataReceived(data) {
    console.log(data);
}
(function () {
    var script = null;
    var buttons = document.querySelectorAll(
            '#loadButtons button');
    for (var i = 0; i < buttons.length; ++i) {
        buttons[i].addEventListener('click', function () {
```

```
            if (script) {
                document.body.removeChild(script);
            }
            script = document.createElement('script');
            document.body.appendChild(script);
            if (this.dataset) {
                script.src = this.dataset.src;
            } else {
                script.src = this.attributes[
                        'data-src'].value;
            }
        });
    }
})();
```

Контрольные вопросы

1. Как можно подключить сценарий к веб-странице?
2. Какие типы данных поддерживаются языком JavaScript?
3. Как объявить переменную?
4. Что представляет собой объект JavaScript как структура данных?
5. Что называется объектным литералом?
6. Как обратиться к свойству объекта?
7. Как организовать цикл по свойствам объекта?
8. Как в языке JavaScript реализованы массивы?
9. Как объявить функцию?
10. Что такое лексический контекст?
11. Что такое замыкание?
12. Что в JavaScript называют методом? Для чего используется ключевое слово `this`?
13. Что в JavaScript называют конструктором? Что такое прототип?
14. Какие преимущества дает метод `Object.create`?
15. Что такое DOM?
16. Как найти элемент в дереве документа?
17. Какие способы навигации по дереву документа поддерживаются в DOM?
18. Как изменить содержимое и/или атрибуты элемента?
19. Что называют событием DOM?
20. Как задать обработчик события?

21. Какие события поддерживаются в DOM?

22. Что такое AJAX? Какие ограничения на использование AJAX накладываются браузером?

23. Как использовать объект `XMLHttpRequest`?

24. Какие форматы данных поддерживает объект `XMLHttpRequest`?

25. Как использовать AJAX без объекта `XMLHttpRequest`?

Заключение

В предшествующих четырех главах сделана попытка представить основы тех технологий, которые используются в настоящее время в Web-приложениях. Разумеется, изложение не является исчерпывающим и многие темы остались за рамками пособия. Среди таковых автор считает необходимым назвать следующие:

- шаблоны проектирования, представляющие собой выработанные практикой и признанные сообществом разработчиков достаточно четко сформулированные решения часто возникающих проблем. К числу важнейших шаблонов, используемых в Web-приложениях, можно отнести Model-View-Controller (модель–вид–контроллер), Front Controller (фронт-контроллер), Master Page (главная страница), Object Relation Mapping (объектно-реляционное отображение) и Session Object (объект сессии);
- фреймворки, избавляющие разработчика от необходимости заново реализовывать многие часто встречающиеся задачи. В отличие от обычных библиотек, предоставляющих средства для решения одной или нескольких обособленных задач, фреймворки, как правило, задают архитектуру всего приложения. Среди наиболее популярных фреймворков для языка PHP можно назвать Zend Framework, Symfony, CakePHP и CodeIgniter, а для языка JavaScript — jQuery, Prototype, script.aculo.us и Dojo. Отметим, что фреймворки, как правило, содержат реализации наиболее часто используемых шаблонов проектирования;
- вопросы, связанные с масштабированием и безопасностью Web-приложений.

Каждая из перечисленных тем заслуживает отдельной книги, в следствие чего остается адресовать читателя к источникам, указанным в списке литературы.

Библиографический список

1. Zend Framework: разработка приложений на PHP / В. Васвани, СПб.: Питер, 2012. 432 с.

2. Велихов, С. Справочник по HTML 4.0 / С. Велихов, М.: Бук пресс, 2006. 412 с.

3. Веллинг, Л., Разработка Web-приложений с помощью PHP и MySQL / Л. Веллинг, Л. Томсон, 3-е издание. М.: Издательский дом «Вильямс», 2008. 880 с.

4. Зандстра, М. PHP: объекты, шаблоны и методики программирования / М. Зандстра, 3-е изд.. М.: ООО «И.Д. Вильямс», 2011. 560 с.

5. Кастро, Э. HTML и CSS для создания Web-страниц / Э. Кастро, М.: НТ Пресс, 2006. 144 с.

6. Квинт, И. HTML, XHTML и CSS на 100 % / И. Квинт, СПб.: Питер, 2010. 384 с.

7. Кингсли-Хью, Э. JavaScript в примерах / Э. Кингсли-Хью, К. Кингсли-Хью, М.: ДМК Пресс, 2009. 272 с.

8. Коггзолл, Дж. PHP 5. Полное руководство / Дж. Коггзолл, М.: Издательский дом «Вильямс», 2006. 752 с.

9. Колисниченко, Д. Н. Профессиональное программирование на PHP / Д. Н. Колисниченко, СПб.: БХВ-Петербург, 2007. 416 с.

10. Котеров, Д. В. PHP 5 / Д. В. Котеров, А. Ф. Костарев, 2-е изд., перераб. и доп. СПб.: БХВ-Петербург, 2008. 1104 с.

11. PHP 5 для профессионалов / Э. Леки-Томпсон, А. Коув, С. Новицки, Х. Айде-Гудман, М.: ООО «И.Д. Вильямс», 2006. 608 с.

12. Маккоу, А. Веб-приложения на JavaScript / А. Маккоу, СПб.: Питер, 2012. 288 с.

13. Макфарланд, Д. Большая книга CSS / Д. Макфарланд, 2-е изд. СПб.: Питер, 2012. 560 с.

14. PHP 5 для начинающих / Д. Мерсер, А. Кент, С. Новицки, Д. Мерсер и др. М.: ООО «И.Д. Вильямс», 2006. 848 с.

15. Моррисон, М. Изучаем JavaScript / М. Моррисон, СПб.: Питер, 2012. 608 с.

16. Никсон, Р. Создаем динамические веб-сайты с помощью PHP, MySQL и JavaScript / Р. Никсон, СПб.: Питер, 2011. 496 с.

17. Прохоренок, Н. А. jQuery. Новый стиль программирования на JavaScript / Н. А. Прохоренок, М.: ООО «И.Д. Вильямс», 2010. 272 с.

18. Симдянов, И. В. PHP. Практика создания Web-сайтов / И. В. Симдянов, М. В. Кузнецов, 2-е изд., перераб. и доп. СПб.: БХВ-Петербург, 2009. 1264 с.

19. Стефанов, С. JavaScript. Шаблоны / С. Стефанов, СПб.: Символ-Плюс, 2011. 272 с.

20. Ташков, П. А. Веб-мастеринг на 100 %: HTML, CSS, JavaScript, PHP, CMS, AJAX, раскрутка / П. А. Ташков, СПб.: Питер, 2010. 512 с.

21. Хеник, Б. HTML и CSS: путь к совершенству / Б. Хеник СПб.: Питер, 2011. 336 с.

22. Хольцнер, С. PHP в примерах / С. Хольцнер, М.: ООО «Бином-Пресс», 2007. 352 с.

23. Шмитт, К. CSS. Рецепты программирования / К. Шмитт, 3-е изд. СПб.: БХВ-Петербург, 2011. 672 с.

24. Эндрю, Р. CSS: 100 и 1 совет / Р. Эндрю, 3-е изд. СПб.: Символ-Плюс, 2010. 336 с.

Приложение А. Протокол HTTP

Протокол HTTP (Hypertext Transfer Protocol — Протокол передачи гипертекста) используется для передачи текстов, изображений, видео и т. п.

Общие сведения

После установки соединения обмен соединениям, начинается с запроса клиента. Одному запросу клиента может соответствовать один или два отклика сервера (в последнем случае первый отклик является предварительным и посылается до приема всего запроса). Хотя в рамках одного соединения может быть передано несколько запросов (и соответственно несколько откликов), каждый запрос обслуживается сервером изолированно, вне связи с предшествующими запросами. Благодаря этому протокол HTTP называют протоколом без состояния (stateless protocol).

Синтаксис сообщения

С точки зрения синтаксиса сообщения HTTP состоят из начальной строки, набора заголовков и необязательного тела сообщения. Начальная строка и заголовки являются текстовыми данными.

В запросе начальная строка задает действие, которое сервер должен выполнить; это строка состоит из трех разделенных пробелами частей: имени метода, идентификатора ресурса и номера версии протокола. Метод задает вид действия, а идентификатор ресурса — ресурс над которым это действие должно быть выполнено. Синтаксис версии:

```
HTTP/ <Старший номер>.<Младший номер>
```

Пример начальной строки:

```
GET /index.html HTTP/1.1
```

В отклике начальная строка описывает результат выполнения действия и также состоит из трех разделенных пробелами частей: номера версии протокола, кода статуса и объясняющей фразы. Код статуса является трехзначным десятичным числом, а объясняющая фраза — произвольным текстом. Код статуса предназначен для ав-

томатической обработки клиентом, а объясняющая фраза — для отображения пользователю. Пример:

```
HTTP/1.1 200 OK
```

В зависимости от старшего разряда коды статута подразделяют на следующие классы:

- Информация (1??). — используется для предварительных откликов и означает, что начальная строка и заголовки успешно приняты. После приемки тела запроса и выполнения запрошенного действия сервер передаст еще один отклик. Обычно сервер посылает предварительные отклики, только если клиент специально этого потребовал.
- Успех (2??) — запрошенное действие успешно выполнено.
- Перенаправление (3??) — действие не выполнено. Для выполнения действия клиент должен послать новый запрос к другому ресурсу.
- Ошибка клиента (4??) — действие не выполнено из-за ошибки клиента. Клиент должен изменить запрос.
- Ошибка сервера (5??) — действие не выполнено из-за ошибки сервера. Клиент может попытаться снова послать этот же запрос.

Заголовки содержат дополнительную информацию. Список заголовков состоит из нескольких строк, причем последняя строка является пустой и служит признаком конца заголовка. Каждый заголовок состоит из имени и значения, которые разделяются двоеточием. Значение заголовка может располагаться на нескольких строках, в этом случае каждая следующая строка должна начинаться с одного или нескольких пробелов (которые игнорируются). Порядок заголовков может быть произвольным. Один и тот же заголовок может встречаться несколько раз, только если его значение семантически является списком. В этом случае старое и новое значения заголовка сцепляются.

С точки зрения семантики различают четыре вида заголовков:

- заголовки запроса характеризуют запрашиваемое действие и могут быть только в запросе;
- заголовки отклика характеризуют результат выполнения действия и могут быть только в отклике;

• общие заголовки могут характеризовать как действия, так и их результаты, и потому могут быть как в запросе, так и в отклике;

• заголовки тела характеризуют тело сообщения, и также могут быть как в запросе, так и в отклике.

Тело сообщения

Тело сообщения может содержать как текстовые, так и бинарные данные. Запрос включает тело, если и только если он содержит заголовок тела *Content-Length* (длина содержания) или общий заголовок *Transfer-Encoding* (кодирование при передаче). Значением заголовка *Content-Length* является десятичное число, представляющее длину тела запроса в байтах. Значением заголовка *Transfer-Encoding* является разделенный запятым список способов кодирования примененных при передаче к данным, составляющим тело сообщения. Определены два способа кодирования: *identity* (без кодирования) и *chunked* (блочное кодирование).

Отклик НЕ включает тело, если и только если:

• он вызван запросом с методом *HEAD*, или

• является информационным, или

• является успешным с кодом 204 (нет содержания), или

• является перенаправлением с кодом 304 (не был изменен).

Во всех остальных случаях (в том числе и в случае ошибки) отклик включает тело. (В случае ошибки телом является описание произошедшей ошибки.)

При наличии тела, его длина определяется:

• явно, значением заголовка *Content-Length*, или

• неявно, путем использования блочного кодирования, или

• неявно, путем закрытия соединения сразу после передачи тела сообщения. В этом случае в сообщение надо включить общий заголовок *Connection* (соединение) со значением *close*. Значение *close* является единственным возможным значением этого заголовка.

Для описания данных, находящихся в теле сообщения, используются следующие заголовки:

• Content-*Type* (тип содержимого) задает тип передаваемых данных. Тип состоит из собственно типа, подтипа и необязательных параметров. (Тип и подтип разделяются наклонной чертой, а

параметры отделяются от подтипа и разделяются между собой точкой с запятой; имя параметра отделяется от значения знаком равенства.) Примеры:

```
Content-Type: image/gif
Content-Type: text/html; charset=windows-1251
```

Значение по умолчанию — *application/octet-stream*;

• Content-*Language* (язык содержания) задает язык. Возможные значения: *ru*, *en*, *de* и т. п.

• Content-*Encoding* (кодирование содержания) задает метод сжатия тела сообщения. Возможные значения: *gzip*, *compress* и *deflate*. По умолчанию сжатия нет. Передающая сторона сначала сжимает данные, а затем кодирует их для передачи. Принимающая сторона должна действовать в обратном порядке.

• Expires (утрачивает силу) задает дату и время, когда данные, составляющие тело сообщения, утратят силу (т. е. устареют). Для задания даты и времени в протоколе HTTP может использоваться один из следующих трех форматов:

```
Expires: Sun, 06 Nov 1994 08:49:37 GMT
Expires: Sunday, 06-Nov-94 08:49:37 GMT
Expires: Sun Nov  6 08:49:37 1994
```

• *Last*-Modified (последнее изменение) задает дату и время последнего изменения ресурса.

Также, для описания тела сообщения может использоваться заголовок отклика *ETag* (тег). Значением этого заголовка является тег ресурса. Тег ресурса — это алфавитно-цифровая последовательность, изменяющаяся при каждом изменении ресурса.

Блочное кодирование

При использовании блочного кодирования данные, составляющие тела сообщения, передаются в виде последовательности блоков переменной длины. Каждый блок начинается строкой, содержащей шестнадцатеричное число, представляющее количество байтов данных в этом блоке. После строки с размером следуют собственно данные, которые могут быть как символьными, так и бинарными. Блок заканчивается пустой строкой (ее размер не учитывается в размере данных). После всех блоков с данными следует

специальный последний блок, не содержащий данных (т. е. количество байтов данных в нем равно 0). В отличие от обычных блоков данных, последний блок не заканчивается пустой строкой. После последнего блока следует (еще один) набор заголовков, заканчивающийся, как обычно, пустой строкой.

Методы

Вне зависимости от метода любой запрос должен иметь заголовок запроса *Host* (компьютер), содержащий доменное имя или адрес компьютера, а также, возможно, его порт. Клиент может включить в запрос следующие заголовки запроса:

- *From* (откуда) — адрес электронной почты пользователя;
- *Referer* (кто ссылается) — идентификатор ресурса, содержащего ссылку на запрашиваемый ресурс;
- *User-Agent* (агент пользователя) — наименование и версия клиента.

Сервер может включить в отклик заголовок отклика *Server* (сервер), содержащий наименование и версию сервера, например:

```
Server: CERN/3.0 libwww/2.17
```

Обычно при успехе все методы передают отклик с кодом 200 (OK). Если ресурса с указанным идентификатором нет, но есть другой ресурс, которым надо пользоваться вместо него, то передается отклик с кодом 301 (перемещен навсегда) или с кодом 307 (временное перенаправление). В обоих случаях отклик должен иметь заголовок *Location* (расположение), содержащий новый (постоянный или временный) идентификатор ресурса.

Если ресурса нет и неизвестно чем его заменить, передается отклик с кодом 404 (не найден).

При ошибке в синтаксисе запроса сервер передает отклик с кодом 400 (плохой запрос), а в случае собственной ошибки — отклик с кодом 500 (внутренняя ошибка сервера). Если клиент запрашивает возможность, не поддерживаемую сервером, передается отклик с кодом 501 (не реализовано).

Теперь перечислим основные методы:

- *GET* (Получить) Передает клиенту в теле отклика представление указанного ресурса. Выполнение метода *GET* не должно из-

менять ресурс. Запрос с методом *GET* обычно не имеет тела. Реализация метода *GET* является обязательной.

Ресурс может иметь несколько представлений. Чтобы выбрать среди них клиент может использовать следующие заголовки запроса:

• *Accept* (принимает) задает приемлемые типы содержимого. Список типов разделяется запятыми. Вместо типа и подтипа можно указать звездочку «*». Сервер по возможности выбирает наиболее специфичный тип содержимого. Пример:

```
Accept: text/*, text/html, */*
```

Типам содержимого можно сопоставить приоритет от 0 до 1. По умолчанию приоритет равен 1. Если приоритет равен 0, то такой тип содержимого является для клиента неприемлемым. Пример:

```
Accept: image/jpeg, image/gif;q=0.3, image/*;q=0.5
```

• *Accept-Charset* (принимает кодировку), *Accept-Encoding* (принимает сжатие), *Accept-Language* (принимает язык) — аналогично *Accept*.

Имена заголовков, которые сервер принял во внимание при выборе представления, сервер может указать в заголовке отклика *Vary* (вариант). Если выбранное представление имеет собственный идентификатор, то он может быть указан в заголовке отклика *Content-Location*.

Если сервер не может предоставить клиенту требуемое представление он передает отклик с кодом 406 (не применимо).

Если не требуются передавать весь ресурс, а достаточно лишь его части, можно использовать частичную форму метода *GET*, для чего в запрос включается заголовок запроса *Range* (диапазон). Значение заголовка состоит из единицы измерения и списка диапазонов. Единица измерения отделяется от списка диапазонов знаком равенства «=», а диапазоны в составе списка разделяются запятыми. Каждый диапазон записывается в виде пары десятичных чисел, разделенных запятыми. Диапазоны считаются закрытыми. Примеры:

```
Range: bytes=0-199,300-599
Range: bytes=900-
Range: bytes=-100
```

Как нижние, так и верхние границы диапазонов могут превосходить размер ресурса. Если хотя бы один диапазон полностью или частично содержится в ресурсе, передается отклик с кодом 206 (частичное содержание), в противном случае с кодом 416 (неудовлетворительный диапазон в запросе). Если запрос содержал только один диапазон, то соответствующий отклик должен иметь заголовок тела *Content-Range* (диапазон содержания), содержащий границы передаваемой части и общий размер ресурса, а заголовок *Content-Length* — длину передаваемой части. Пример:

```
Content-Range: bytes 0-199/1000
Content-Length: 200
```

Если в заголовке *Range* диапазонов было несколько, то отклик содержит данные типа *multipart/byteranges*.

• *HEAD* (Заголовок) То же, что *GET*, но отклик не содержит тела (но содержит заголовки тела). Реализация метода *HEAD* так же является обязательной.

• *POST* (Послать) Передает серверу в теле запроса данные для включения их в указанный ресурс в качестве составной части. Необязательное тело отклика может содержать данные, описывающие произошедшее действие. Если же тело отсутствует, то отклик должен содержать код 204. Если результат выполнения метода может быть описан каким-либо ресурсом, то отклик может иметь код 201 (создан) или 303 (смотри другой) и заголовок *Location* содержащий идентификатор этого ресурса. В этом случае клиент должен отправить к этому ресурсу запрос с методом *GET*. Хотя метод *POST* может изменять содержимое ресурса, но его реализация должна быть такова, чтобы выполнение последовательности из нескольких одинаковых запросов с методом *POST* переводило бы ресурс в то же состояние, что и единственный запрос (т. е. другими словами, реализация должна исключать гонки). Обычно метод *POST* используют для редактирования информации в базах данных, отправки почты и сообщений в группы новостей и др. подобных задач.

• *PUT* (Поместить) Передает серверу в теле запроса новое содержимое указанного ресурса. Отличается от *POST* тем, что тело запроса не добавляется в ресурс, а заменяет его.

• *DELETE* (Удалить) Удаляет ресурс.

Перечисленные методы могут быть условными. Условный метод выполняется только в том случае, если выполняется условие, указанное в запросе. Условия задаются следующими заголовками запроса:

• *If-Modified-Since* (если был изменен) метод выполняется только, если ресурс изменился с момента, указанного в этом заголовке. В противном случае, т. е. если начиная с указанного времени и до настоящего момента ресурс не менялся, при использовании с методами *GET* и *HEAD*, передает отклик с кодом 304, а при использовании с остальными вышеперечисленными методами — отклик с кодом 412 (предусловие не выполнено).

• *If-None-Match* (если ни один не совпадает) метод выполняется только, если текущий тег ресурса не совпадает ни с одним из тегов, указанных в этом заголовке. В противном случае в зависимости от метода передает отклик с кодом 304 или с кодом 412. Вместо списка тегов в заголовке *If-None-Match* может быть звездочка «*», означающая, что метод выполняется, если ресурс не существует.

• *If-Unmodified-Since* (если не был изменен) метод выполняется только если ресурс НЕ изменялся, начиная с даты, указанной в заголовке. В противном случае передает отклик с кодом 412.

• *If-Match* (если совпадает) метод выполняется только если тег ресурса совпадает с одним из тегов, указанных в этом заголовке. В противном случае передает отклик с кодом 412.

Учебное издание

Бородин Михаил Владимирович

Титенко Евгений Анатольевич

ИНТЕРНЕТ-ТЕХНОЛОГИИ

Учебное пособие

Редактор Е. А. Припачкина
Компьютерная верстка и макет

Подписано в печать . Формат 60 × 84 1/16. Бумага офсетная.
Усл. печ. л. . Уч.-изд. л. . Тираж 250 экз. Заказ .
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94.

Отпечатано в ЮЗГУ.