Титульный лист методических рекомендаций и указаний; методических рекомендаций; методических указаний

Форма
Ф СО ПГУ 7.18.3/40

Министерство образования и науки Республики Казахстан

Павлодарский государственный университет имени С.Торайгырова

Кафедра информатики и информационных систем

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ И УКАЗАНИЯ

к выполнению лабораторных работ
по дисциплине Web-технологии
для студентов специальности 050703 – Информационные системы

Павлодар

Лист утверждения методических рекомендаций и указаний; методических рекомендаций; методических указаний

Форма
Ф СО ПГУ 7.18.3/41

УТВЕРЖДАЮ
Декан ФФМиИТ
___________ ___________
(подпись) (Ф.И.О.)
«___»____________20__г.

Составитель: старший преподаватель Науман

Кафедра Информатики и информацис

Методические рекомендации и указания

к выполнению лабораторных работ

по дисциплине Web-технологии
для студентов специальности 050703-Информационные системы

Рекомендовано на заседании кафедры
«___»______________20____г., протокол №__

Заведующий кафедрой _____________ Асаинова А.Ж. «____» ________20__г.
(подпись) (Ф.И.О.)
Одобрено УМС ФФМиИТ
«___»______________20__г., протокол №____
Председатель УМС ____________ Муканова Ж.Г. «____» ________20__г.
(подпись) (Ф.И.О.)
ОДОБРЕНО:
Начальник ОПиМОУП _____________ Варакута А.А. «____» ________20__г.
(подпись) (Ф.И.О.)

Одобрена учебно-методическим советом университета
«___»______________20__г. Протокол №____

Лабораторная работа №1. Формы в HTML-документах.

Краткие теоретические сведения

Формы передают информацию программам-обработчикам в виде пар [имя переменной]=[значение переменной]. Имена переменных следует задавать латинскими буквами.

Значения переменных воспринимаются обработчиками как строки, даже если они содержат только цифры.

Форма открывается тегом <FORM> и заканчивается меткой </FORM>. HTML-документ может содержать в себе несколько форм, однако формы не должны находиться одна внутри другой.

Тег **<FORM>** может содержать три атрибута, один из которых является обязательным:

ACTION Обязательный атрибут. Определяет, где находится обработчик формы.

METHOD Определяет, каким образом (иначе говоря, с помощью какого метода протокола передачи гипертекстов) данные из формы будут переданы обработчику. Допустимые значения: METHOD=POST и METHOD=GET. Если значение атрибута не установлено, по умолчанию предполагается METHOD=GET.

GET: методом "get" HTTP браузер берёт значение action, добавляет `?' к нему, затем присоединяет набор данных формы, кодированный с использованием типа содержимого "application/x-www-form-urlencoded". Затем перенаправляет всё по гиперссылке на этот URL. В этом сценарии данные формы ограничены кодами ASCII (нельзя использовать спецсимволы) и имеют весьма жесткие ограничения на объем вводимой информации.

POST: методом "post" HTTP браузер проводит транзакцию HTTP "post" (в теле HTTP-запроса), используя значение атрибута action и сообщение, созданное в соответствии с типом содержимого, определённым атрибутом enctype.

ENCTYPE Определяет, каким образом данные из формы будут закодированы для передачи обработчику. Если значение атрибута не установлено, по умолчанию предполагается ENCTYPE=application/x-www-form-urlencoded.

"Кнопка", чтобы запустить процесс передачи данных из формы на сервер, создается с помощью тега.

<INPUT TYPE=submit>

Встретив такую строчку внутри формы, браузер нарисует на экране кнопку с надписью Submit, при нажатии на которую все имеющиеся в форме данные будут переданы обработчику, определенному в метке.

Надпись на кнопке можно задать любую путем введения атрибута VALUE="[Надпись]"например:

<INPUT TYPE=submit VALUE="Отправить!">

Надпись, нанесенную на кнопку, можно при необходимости передать обработчику путем введения в определение кнопки атрибута NAME=[имя] например:

<INPUT TYPE=submit NAME=button VALUE="Отправить!">

При нажатии на такую кнопку обработчик вместе со всеми остальными данными получит и переменную button со значением Отправить! (т.е. button=Отправить!, это можно видеть в адресной строке).

В форме может быть несколько кнопок типа submit с различными именами и/или значениями.Обработчик, таким образом, может действовать по-разному в зависимости от того, какую именно кнопку submit нажал пользователь.

Существуют и другие типы элементов **<INPUT>**. Каждый элемент **<INPUT>** должен включать атрибут **NAME=[имя]**, определяющий имя переменной, которая будет передана обработчику. Имя должно задаваться только латинскими буквами. Большинство элементов **<INPUT>** должны включать атрибут **VALUE="[значение]"**, определяющий значение, которое будет передано обработчику под этим именем.

Основные типы элементов <INPUT>:

TYPE=text

Определяет окно для ввода строки текста. Может содержать дополнительные атрибуты SIZE=[число] (ширина поля для ввода, в символах) и MAXLENGTH=[число] (максимально допустимая длина вводимой строки в символах).

Пример:

<INPUT TYPE=text SIZE=30 NAME=student VALUE="Иванов Иван">

Определяет ширину поля в 30 символов, для ввода текста. По умолчанию в окне находится текст Иванов Иван, который пользователь может редактировать. Отредактированный (или неотредактированный) текст передается обработчику в переменной student (student=содержимое_поля). Попробуйте отредактировать поле.

TYPE=password

Определяет окно для ввода пароля. Абсолютно аналогичен типу text, только вместо символов вводимого текста показывает на экране звездочки (*), чтобы посторонний не мог прочесть.

Пример:

<INPUT TYPE=password NAME=pswd SIZE=20 MAXLENGTH=10>

Определяет окно шириной 20 символов для ввода пароля. Максимально допустимая длина пароля — 10 символов. Введенный пароль передается обработчику в переменной pswd (pswd=содержимое_поля). Попробуйте ввести информацию в поле.

TYPE=radio

Определяет радиокнопку. Может содержать дополнительный атрибут checked (показывает, что кнопка помечена). В группе радиокнопок с одинаковыми именами может быть только одна помеченная радиокнопка.

Пример:

<INPUT TYPE=radio NAME=modem VALUE="9600" checked>
<INPUT TYPE=radio NAME=modem VALUE="14400">
<INPUT TYPE=radio NAME=modem VALUE="28800">

Определяет группу из трех радиокнопок, подписанных 9600 бит/с, 14400 бит/с и 28800 бит/с. Первоначально помечена первая из кнопок. Если пользователь не отметит другую кнопку, обработчику будет передана переменная modem со значением 9600 (modem=9600). Если пользователь отметит вторую кнопку, обработчику будет передана переменная modem со значением 14400 (modem=14400).

TYPE=checkbox

Определяет квадрат, в котором можно сделать пометку. Может содержать дополнительный атрибут checked (показывает, что квадрат помечен). В отличие от радиокнопок, в группе квадратов с одинаковыми именами может быть несколько помеченных квадратов.

Пример:

<INPUT TYPE=checkbox NAME=comp VALUE="PC">
<INPUT TYPE=checkbox NAME=comp VALUE="WS" checked>
<INPUT TYPE=checkbox NAME=comp VALUE="LAN">
<INPUT TYPE=checkbox NAME=comp VALUE="IS" checked>

Определяет группу из четырех квадратов. Первоначально помечены второй и четвертый квадраты. Если пользователь не произведет изменений, обработчику будут передана одна переменная comp с двумя значениями (comp=WS и comp=IS).

TYPE=hidden

Определяет скрытый элемент данных, который не виден пользователю при заполнении формы и передается обработчику без изменений. Такой элемент иногда полезно иметь в форме, в него можно спрятать от пользователя служебные данные.

Пример:

<INPUT TYPE=hidden NAME=id VALUE="1">

Определяет скрытую переменную индексную id, которая передается обработчику со значением 1.

TYPE=reset

Определяет кнопку, при нажатии на которую форма возвращается в исходное состояние (обнуляется). Поскольку при использовании этой кнопки данные обработчику не передаются, кнопка типа reset может и не иметь атрибута name. Пример:

<INPUT TYPE=reset VALUE="Очистить поля формы">

Определяет кнопку Очистить поля формы, при нажатии на которую форма возвращается в исходное состояние.

Элемент <SELECT>:

Меню <SELECT> из n элементов выглядит примерно так:

```
<SELECT NAME="[имя]">
<OPTION VALUE="[значение 1]">[текст 1]
<OPTION VALUE="[значение 2]">[текст 2]
…
<OPTION VALUE="[значение n]">[текст n]
</SELECT>
```

Метка <SELECT> содержит обязательный атрибут NAME, определяющий имя переменной. Метка <SELECT> может также содержать атрибут MULTIPLE, присутствие которого показывает, что из меню можно выбрать несколько элементов. Большинство браузеров показывают меню <SELECT MULTIPLE> в виде окна, в котором находятся элементы меню (высоту окна в строках можно задать атрибутом SIZE=[число]). Для выбора нескольких значений одновременно удерживают кнопку "SHIFT" и выбирают значения мышкой.

```
<SELECT MULTIPLE SIZE=3 NAME="[имя]">
<OPTION VALUE="[значение 1]">[текст 1]
<OPTION VALUE="[значение 2]">[текст 2]
<OPTION VALUE="...">[...]
<OPTION VALUE="[значение n]">[текст n]
</SELECT>
```

Метка <OPTION> определяет элемент меню. Обязательный атрибут VALUE устанавливает значение, которое будет передано обработчику, если выбран этот элемент меню. Метка <OPTION> может включать атрибут selected, показывающий, что данный элемент отмечен по умолчанию. Пример:

```
<SELECT NAME="selection">
<OPTION VALUE="option1">Вариант 1
<OPTION VALUE="option2" selected>Вариант 2
<OPTION VALUE="option3">Вариант 3
</SELECT>
```

Такой фрагмент определяет меню из трех элементов: Вариант 1, Вариант 2 и Вариант 3. По умолчанию выбран элемент Вариант 2. Обработчику будет передана переменная selection (selection=...) значение которой может быть option1 (по умолчанию), option2 или option3.

Элемент <TEXTAREA>:

Пример:

```
<TEXTAREA NAME=address ROWS=5 COLS=50>
Поле для ввода большого текста, разбитого на абзацы.
</TEXTAREA>
```

Все атрибуты обязательны. Атрибут **NAME** определяет имя, под которым содержимое окна будет передано обработчику (в примере — address). Атрибут **ROWS** устанавливает высоту окна в строках (в примере — 5). Атрибут **COLS** устанавливает ширину окна в символах (в примере — 50). Текст, размещенный между метками <TEXTAREA> и </TEXTAREA>, представляет собой содержимое окна по умолчанию. Пользователь может его отредактировать или просто стереть

ЗАДАНИЕ

1. Создайте новую страницу. Составьте форму-анкету (используя методом **POST**), включающую в себя следующие поля (**все переменные должны быть, читаемы, например: русский язык - langru, а не C1 или T2, по умолчанию в значениях должны быть ваши данные**):
   - Фамилия
   - Имя
   - Отчество
   - E-mail
   - Выбор страны (обязательно выпадающим **SELECT**, стран не менее 10-ти)
   - Выбор города (обязательно с помощью **radio,** переменные должны быть одинаковыми**,** не менее 5-ти)

- Выбор языка (обязательно с помощью **checkbox**, переменные должны быть разными, не менее 5-ти)
- Выбор профессий (обязательно с помощью **SELECT MULTIPLE,** переменные должны быть разными, не менее 10-ти)
- Пароль
- Дополнительная информация (обязательно с помощью **TEXTAREA**)

2. В скрытом поле (**hidden**), передайте переменную student со значением "Ваше_имя"(student=Ваше_имя).
3. Расположите на форме кнопки: Кнопка для загрузки информации на сервер; Кнопка для очистки формы

4. В исходном коде страницы укажите, в виде комментариев, для чего предназначены используемые для форм теги и их свойства (атрибуты).

5. Создайте новую страницу, скопировав предыдущею страницу. Измените метод передачи на **GET**.

6. Сделайте ссылки с первой страницы, на эти две страницы.

7. Создайте новую страницу. Создайте простые поисковые формы к следующим поисковым системам:
www.yandex.ru
www.rambler.ru
www.aport.ru
www.google.com
www.filesearch.ru
URL обработчика и переменные посмотрите в исходных кодах страниц соответствующих поисковых систем.

Лабораторная работа №2. Бегущая строка, таймер, окна на javascript.

Краткие теоретические сведения

Составленные Вами программы на JavaScript могут выполнять запись в строку состояния - прямоугольник в нижней части окна Вашего браузера. Все, что Вам необходимо для этого сделать - всего лишь записать нужную строку в window.status. В следующем примере создаются две кнопки, которые можно использовать, чтобы записывать некий текст в строку состояния и, соответственно, затем его стирать.

Данный скрипт выглядит следующим образом:

```
<html>
<head>
<script language="JavaScript">
<!-- hide
function statbar(txt) {
window.status = txt;
}
// -->
</script>
</head>
<body>
<form>
<input type="button" name="look" value="Писать!"
onClick="statbar('Привет! Это окно состо\яни\я!');">
<input type="button" name="erase" value="Стереть!"
onClick="statbar('');">
</form>
</body>
</html>
```

Итак, мы создаем форму с двумя кнопками. Обе эти кнопки вызывают функцию statbar(). Вызов от клавиши Писать! выглядит следующим образом:

```
statbar('Привет! Это окно состо\яни\я!');
```

В скобках мы написали строку: 'Привет! Это окно состо\яни\я!'. Это как раз и будет текст, передаваемый функции statbar(). В свою очередь, можно видеть, что функция statbar() определена следующим образом:

```
function statbar(txt) {
window.status = txt; }
```

В заголовке функции в скобках мы поместили слово txt. Это означает, что строка, которую мы передали этой функции, помещается в переменную txt. Передача функциям переменных - прием, часто применяемый для придания функциям большей гибкости. Вы можете передать функции несколько таких аргументов - необходимо лишь отделить их друг от друга запятыми. Строка txt заносится в строку состояния посредством команды window.status = txt. Соответственно, удаление текста из строки состояния выполняется как запись в window.status пустой строки.

Механизм вывода текста в строку состояния удобно использовать при работе со ссылками. Вместо того, чтобы выводить на экран URL данной ссылки, Вы можете просто на словах объяснять, о чем будет говориться на следующей странице. Так link демонстрирует это - достаточно лишь поместить указатель вашей мыши над этой ссылкой: Исходный код этого примера выглядит следующим образом:

```
<a href="dontclck.htm"
onMouseOver="window.status='Don\'t click me!'; return true;"
onMouseOut="window.status=";">link</a>
```

Здесь мы пользуемся процедурами onMouseOver и onMouseOut, чтобы отслеживать моменты, когда указатель мыши проходит над данной ссылкой. Вы можете спросить, а почему в onMouseOver мы обязаны возвращать результат true. На самом деле это означает, что браузер не должен вслед за этим выполнять свой собственный код обработки события MouseOver. Как правило, в строке состояния браузер показывает URL соответствующей ссылки. Если же мы не возвратим значение true, то сразу же после того, как наш код был выполнен, браузер перепишет строку состояния на свой лад - то есть наш текст будет тут же затерт и читатель не сможет его увидеть. В общем случае, мы всегда можем отменить дальнейшую обработку события браузером, возвращая true в своей собственной процедуре обработки события. в JavaScript 1.0 процедура onMouseOut еще не была представлена. И если Вы пользуетесь Netscape Navigator 2.x, то возможно на различных платформах Вы можете получить различные результаты. Например, на платформах Unix текст исчезает даже несмотря на то, что браузер не знает о существовании процедуры onMouseOut. В Windows текст не исчезает. И если Вы хотите, чтобы ваш скрипт был совместим с Netscape 2.x для Windows, то можете, к примеру, написать функцию, которая записывает текст в окно состояния, а потом стирает его через некоторый промежуток времени. Программируется это с помощью таймера timeout. Подробнее работу с таймерами мы рассмотрим в следующем параграфе. В этом скрипте Вы можете видеть еще одну вещь - в некоторых случаях Вам понадобится печатать символы кавычек. Например, мы хотим напечатать текст Don't click me - однако поскольку мы передаем эту строку в процедуру обработки события onMouseOver, то мы используем для этого одинарные кавычки. Между тем, как слово Don't тоже содержит символ одинарной кавычки! И в результате если Вы просто впишете 'Don't ...', браузер запутается в этих символах '. Чтобы разрешить эту проблему, Вам достаточно лишь поставить обратный слэш \ перед символом кавычки - это означает, что данный символ предназначен именно для печати. (То же самое Вы можете делать и с двойными кавычками - ").

Таймеры

С помощью функции Timeout (или таймера) Вы можете запрограммировать компьютер на выполнение некоторых команд по истечении некоторого времени. В следующем скрипте демонстрируется кнопка, которая открывает выпадающее окно не сразу, а по истечении 3 секунд.

Скрипт выглядит следующим образом:

```
<script language="JavaScript">
<!-- hide
function timer() {
setTimeout("alert('Врем\я истекло!')", 3000);}
// -->
```

```
</script>
...
<form>
<input type="button" value="Timer" onClick="timer()"> </form>
```


Здесь setTimeout() - это метод объекта window. Он устанавливает интервал времени - Вы наверно догадываетесь, как это происходит. Первый аргумент при вызове - это код JavaScript, который следует выполнить по истечении указанного времени. В нашем случае это вызов - "alert('Врем\я истекло!')". Обратите, пожалуйста, внимание, что код на JavaScript должен быть заключен в кавычки. Во втором аргументе компьютеру сообщается, когда этот код следует выполнять. При этом время Вы должны указывать в миллисекундах (3000 миллисекунд = 3 секунда).

Прокрутка

Теперь, когда Вы знаете, как делать записи в строке состояния и как работать с таймерами, мы можем перейти к управлению прокруткой. Вы уже могли видеть, как текст перемещается строке состояния. В Интернет этим приемом пользуются повсеместно. Теперь же мы рассмотрим, как можно запрограммировать прокрутку в основной линейке. Рассмотрим также и всевозможные усовершенствования этой линейки. Создать бегущую строку довольно просто. Для начала давайте задумаемся, как вообще можно создать в строке состояния перемещающийся текст - бегущую строку. Очевидно, сперва мы должны записать в строку состояния некий текст. Затем по истечении короткого интервала времени мы должны записать туда тот же самый текст, но при этом немного переместив его влево. Если мы это сделаем несколько раз, то у пользователя создастся впечатление, что он имеет дело с бегущей строкой. Однако при этом мы должны помнить еще и о том, что обязаны каждый раз вычислять, какую часть текста следует показывать в строке состояния (как правило, объем текстового материала превышает размер строки состояния).

Эта кнопка откроет окно и покажет образец прокрутки:

Итак, исходный код скрипта – добавлены еще некоторые комментарии:

```
<html>
<head>
<script language="JavaScript">
<!-- hide
// инициализация текста прокрутки
// объявление переменных и задание им значений
var scrtxt = "Это JavaScript! " +"Это JavaScript! " +"Это JavaScript!";
var len = scrtxt.length;\\вычисляем длину строки
var width = 100;
var pos = -(width + 2);
function scroll() {
// напечатать заданный текст справа и установить таймер
// перейти на исходную позицию для следующего шага
pos++;
// вычленить видимую часть текста
var scroller = "";
if (pos == len) {\\\ конструкция условного оператора
pos = -(width + 2);
}
// если текст еще не дошел до левой границы, то мы должны
// добавить перед ним несколько пробелов. В противном случае мы должны
// вырезать начало текста (ту часть, что уже ушла за левую границу
if (pos < 0) {
for (var i = 1; i <= Math.abs(pos); i++) {\\\конструкция цикла со счетчиком
scroller = scroller + " ";}
scroller = scroller + scrtxt.substring(0, width - i + 1);
}
```

```
else {
scroller = scroller + scrtxt.substring(pos, width + pos);
}
// разместить текст в строке состо\яни\я
window.status = scroller;
// вызвать эту функцию вновь через 100 миллисекунд
setTimeout("scroll()", 100);
}
// -->
</script>
</head>
<body onLoad="scroll()">
Это пример прокрутки в строке состояния средствами JavaScript.
</body>
</html>
```

Большая часть функции scroll() нужна для вычленения той части текста, которая будет показана пользователю.

Рассмотрим использованные функции:

scrtxt.substring(m, n) – копирует из строки scrtxt начиная с символа m ровно n символов и возвращает в качестве результата

Для соединения строк используется символ +.

Функция Math.abs(pos)- используется для вычисления модуля числа pos.

Функция scrtxt.length – определяет длину строки и возвращает ее в качестве результата. Чтобы запустить этот процесс, мы используемся процедурой обработки события onLoad, описанной в тэге <body>. То есть функция scroll() будет вызвана сразу же после загрузки HTML-страницы. Через посредство процедуры onLoad мы вызываем функцию scroll(). Первым делом в функции scroll() мы устанавливаем таймер. Этим гарантируется, что функция scroll() будет повторно вызвана через 100 миллисекунд. При этом текст будет перемещен еще на один шаг и запущен другой таймер. Так будет продолжаться без конца. Данный вызов функции не является рекурсивным! Рекурсию мы получим, если будем вызывать функцию scroll() непосредственно внутри самой же функции scroll(). А этого здесь мы как раз и не делаем. Прежняя функция, установившая таймер, заканчивается еще до того, как начинается выполнение новой функции. Скроллинг используется в Интернет довольно широко. И есть риск, что быстро он станет непопулярным. В большинстве страниц, где он применяется, особенно раздражает то, что из-за непрерывного скроллинга становится невозможным прочесть в строке состояния адрес URL. Эту проблему можно было бы решить, позаботившись о приостановке скроллинга, если происходит событие MouseOver - и, соответственно, продолжении, когда фиксируется onMouseOut. Если Вы хотите попытаться создать скроллинг, то, пожалуйста, не используйте стандартный его вариант - пробуйте привнести в него некоторые приятные особенности. Возможен вариант, когда одна часть текста приходит слева, а другая - справа. И когда они встречаются посередине, то в течение некоторых секунд текст остается неизменным.

Функции даты/времени

Работа с датой и временем, установленными на данном компьютере, в JavaScript осуществляется с помощью объекта Date. Перед его использованием его необходимо создать:

DateObject = new Date()

Данной строкой создается объект Date, содержащий текущее системное время компьютера, на котором он создается. При его создании в него можно поместить не системное время, а произвольное. Для этого применяется конструкция: var DateObject = new Date([значение])

где параметр значение может принимать следующие значения:

- миллисекунды - количество миллисекунд прошедшее от 01/01/70 00:00:00; - год, месяц, день; - год, месяц, день, часы, минуты, секунды; - год, месяц, день, часы : минуты : секунды. Методы объекта Date, для работы с датой и временем:

Метод	Описание
getDate()	число месяца

getDay()	номер дня недели
getHours()	Час
getMinutes()	Минута
getMonth()	номер месяца
getSeconds()	Секунда
getTime()	количество миллисекунд между 1 января 1970 года и текущим временем
getTimezoneOffset()	разница в минутах между местным и гринвичским временем
getYear()	год
setDate()	устанавливает день месяца
setHours()	устанавливает час
setMonth()устанавливает минуту setMinutes()	устанавливает месяц
setSeconds()	устанавливает секунду
setYear()	устанавливает год
toGMTString()	преобразует местное время во время по Гринвичу
toLocaleString()	преобразует время по Гринвичу в местное время
возвращает количество миллисекунд между 01/01/70 00:00:00 и заданным временем в формате объекта Date UTC()	

getDate() Возвращает текущий день месяца:
var D, day; D = new Date(); day= D.getDate(); alert("Сегодня " + day + " число");
getDay() Возвращает день недели. Возвращаемая величина представляет собой целое число от 0 до 6. Поскольку дни недели нумеруются, начиная с нуля, первым днем идет воскресенье.
Таким образом:
Воскресение = 0; Понедельник = 1; Вторник = 2; Среда = 3; Четверг = 4; Пятница = 5;Суббота = 6.
function Day_of_week() { var day, DateObj;
//помещаем в массив названия дней недели: var Days = new Array("Воскресенье", "Понедельник", "Вторник", "Среда", "Четверг", "Пятница", "Суббота");
//создаем объект Date DateObj = new Date();
//получаем номер дня day = DateObj.getDay();
//вызываем из массива название дня по его номеру и выводим результат: alert("Сегодня у нас " + Days[day]); }
getMonth() Показывает номер текущего месяца как целое число от 0 до 11. Как и в днях недели нумерация месяцев начинается с нуля:
Январь = 0; Февраль = 1; … Декабрь = 11.
Для демонстрации метода getMonth() составим сценарий, выводящий название месяца:
function Month_of_Year() { var month, DateObj;
//помещаем в массив названия месяцев: var Months = new Array("Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль","Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь");
//создаем объект Date DateObj = new Date();
//получаем номер месяца: month = DateObj.getMonth();
//вызываем из массива название дня по его номеру и выводим результат: alert("Сейчас идет " + Months[month] + " месяц"); }
getYear() Возвращает количество лет прошедших от 1900–го года, в виде целого двухразрядного числа от 0 до 99. До 2000-го года, чтобы вывести полную дату, нужно было к результату, возвращаемому методом getYear() прибавлять 1900:

... var Year = 1900 + DataObj.getYear(); ...

Особенность использования метода **getYear**() заключается в том, что современные версии броузеров, как Netscape, так и Microsoft, при использовании метода в 2000-х годах выведут текущий год в четырехразрядном числе: 2000, 2001 и т.д. И сценарии, которые годились до наступления 2000 года, будут работать неверно, выдавая результат на 1900 лет больше. Следует также сказать, что некоторые браузеры по-прежнему при использовании метода getYear() считают время от 1900 года. Для избегания подобных недоразумений можно составить подобный сценарий, контролирующий это:

<script>
//создаем объект Date: DateObj = new Date();
//получаем год y = DateObj.getYear();
//если год двухразрядный: if (y < 2000) { year = 1900 + y; } else {
//оставляем год без изменений: year = y; }
alert("Сейчас идет год " + year); </script>

getHours() Указывает текущий час. Возвращает целое число от 0 до 23. **getMinutes**() Выводит значение минут, числом от 0 до 59

getSeconds() Выводит значение текущей секунды, числом от 0 до 59.

Если возвращаемые минуты и секунды находятся в промежутке от 0 до 10, то JavaScript возвращает их без нулей: 2, 3 и т.д.

Для удобства пользователей, желательно добавить к ним нуль:

... tSec = DateObj.getSeconds(); if (tSec < 10) { tSec = "0" + tSec; } ... 3 курс 080801 Лабораторная работа № 18 « Строка состояния, таймеры и время »

Для вывода текущего времени составим следующий сценарий:

function T() { //создаем объект Date: DateObj = new Date();
//получаем текущий час: tHour = DateObj.getHours();
//получаем текущую минуту tMin = DateObj.getMinutes(); if (tMin < 10) { tMin = "0" + tMin; }
//секунды: tSec = DateObj.getSeconds(); if (tSec < 10) { tSec = "0" + tSec; }
alert("Текущее время: " + tHour + ":" + tMin + ":" + tSec); }

Если мы хотим не только выводить текущее время по требованию, а поместить на экран настоящие "тикающие" часы, нужно добавить в функцию метод setTimeout().

Создадим на странице поле ввода, в которые поместим наши часы:

<center> Текущее время: <form> <input type="text" size=8> </form> </center>

Чтобы часы отображались в поле ввода, заменим строку

alert("Текущее время: " + tHour + ":" + tMin + ":" + tSec);

на

//присваиваем переменной clock текущее время: clock = tHour + ":" + tMin + ":" + tSec;
//выводим время в текстовое поле document.forms[0].elements[0].value = clock; //Добавляем таймер setTimeout("T()", 1000); Теперь в тег <BODY> добавим обработчик onLoad, вызывающий нашу функцию:

<body onLoad = "T()"> Если мы захотим, то можем поместить часы в строку состояния броузера. Для этого заменим строку document.forms[0].elements[0].value = clock;

на window.status = "Текущее время: " + clock;

Теперь у нас в строке состояния будут системные часы. **getTime**() Метод **getTime**, показывает, сколько миллисекунд прошло от 1 января 1970 года

getTimezoneOffset() Показывает разницу между местным и Гринвичским временем в минутах.

var DateObj = new Date(); var x = DateObj.getTimezoneOffset() alert("Местное время отличается от Гринвичского на "+ x + " минут");

toGMTString() Метод toGMTString() преобразует местное время в гринвичское. Причем возвращает не только время, но и текущую дату. Отображение инфор-мации зависит от используемой платформы.

var DateObj = new Date(); var x = DateObj.toGMTString() alert("Текущее время по Гринвичу: " + x);

toLocaleString() Переводит время по Гринвичу в местное время и дату. Создавать часы на странице данным методом удобнее, чем, используя методы getHours(), getMinutes(), getSeconds(), но только метод toLocaleString() выводит еще и текущую дату.

Составим сценарий выводящий дату и время в строку состояния:

<SCRIPT> function T() { var DateObj = new Date(); var Now_Time = DateObj.toLocaleString();
window.status = "Сейчас " + Now_Time; setTimeout("T()", 1000); } </SCRIPT>
после этого добавим в тег <BODY> вызов функции: <BODY onLoad="T()"> **setDate**() Устанавливает число месяца для созданного объекта Date.
Синтаксис: DateObject.setDate(новое значение)
где новое значение - целое число от 1 до 31
setHours() Устанавливает часы для объекта Date.
Синтаксис: DateObject.setHours(новое значение)
где новое значение - целое число от 0 до 23
setMinutes() Устанавливает минуты для объекта Date.
Синтаксис: DateObject.setMinutes(новое значение)
setMonth() Устанавливает месяц для объекта Date.
Синтаксис: DateObject.setMonth(новое значение)
где новое значение - целое число от 0 до 11
Проиллюстрируем использование метода **setMonth** сценарием, выводящим название предыдущего месяца. Составим этот сценарий только для примера, поскольку получить номер предыдущего месяца можно отняв от значения возвращенного методом getMonth единицу. <script>
function PreviousMonth() { var month, DateObj; var Months = new Array("Январь", "Февраль", "Март", "Апрель", "Май", "Июнь", "Июль","Август", "Сентябрь", "Октябрь", "Ноябрь", "Декабрь");
//создаем объект Date, содержащий текущее системное время: DateObj = new Date();
//получаем текущий месяц month = DateObj.getMonth();
//если месяц не январь: if (month >0) { //устанавливаем предыдущий месяц: DateObj.setMonth(month-1);
//получаем его: month = DateObj.getMonth();
alert("Прошлый месяц был " + Months[month]); } else
//если текущий месяц январь if (month == 0) {
//устанавливаем месяц Декабрь: DateObj.setMonth(11);
//получаем его: month = DateObj.getMonth();
alert("Прошлый месяц был " + Months[month]); } } </script> **setSeconds**() Устанавливает секунды для объекта Date.
Синтаксис: DateObject.setSeconds(новое значение)
где новое значение - целое число от 0 до 59 **setYear**() Устанавливает год для объекта Date.
Синтаксис: DateObject.setYear(новое значение)
где новое значение - целое число, устанавливающее год

Задание.

Используя полученные знания, создайте страницу, выполняющие следующие операции.

1. Страница содержит 2 фрейма, расположенных друг под другом.
2. В верхнем фрейме посередине рассоложены идущие часы, а в верхнем правом углу текущая дата в формате «1 января 2007 года, понедельник».
3. В нижнем фрейме имитируется движение титров, левая половина текста медленно уползает вверх, а правая вниз.
4. В строке состояния расположена бегущая строка с ФИО автора.

Лабораторная работа №3. Программирование игр на javascript.

Игра «Спички»

```
<script language="JavaScript"><!--
// Copyright (c) 1996-1997 Tomer Shiran. All rights reserved.
function stripQuery() {
var search = location.search
var length = search.length
if (search == "")
return "25" //замените на 26 и Вы всегда будете выигрывать!!!!!!!
var query = search.substring(1, length)
return query
```

```
}
function placeMatches(num) {
for (var i = 1; i <= num; ++i) {
document.write('<A HREF="' + getURL(i, num) + '"><IMG SRC="match.gif" BORDER=0></A>')
}
}
function getURL(pos, num) {
var distance = num - pos + 1
if (distance > 3)
return "javascript:alert('Выберите одну из последних трех спичек')"
return "Game40.htm?" + (num - 4)
}
var num = parseInt(stripQuery())
var instructions = ""
instructions += "Цель игры: вы должны сделать так, чтобы последняя "
instructions += "спичка досталась другому игроку (компьютеру). "
instructions += "За один ход Вы можете взять себе только 1, 2, "
instructions += "или 3 спички, считая с конца. "
instructions += "То есть, кликнув по второй с конца спичке, Вы забираете "
instructions += "две штуки, кликнув по третьей с конца - забираете "
instructions += "три, по первой с конца - одну. Для компьютера правила "
instructions += "теже! Так что играйте с головой а не наобум, "
instructions += "иначе проиграете!"
if (num == 25)
alert(instructions)
if (num == 1) {
document.write('<IMG SRC="match.gif"><BR><BR><A HREF="Game40.htm">Повторим?</A>')
alert("\Я выиграл! Вам надо попрактиковаться. Кстати, не обязательно с компьютером!")
} else
if (num < 1)
alert("Вы выиграли! Поздравляю!!!")
else
placeMatches(num)
// --></script>
```

Игра «Уголки»

```
<SCRIPT LANGUAGE="JavaScript">
<!--
/***************** Exchange Functions ********************/
function initArray() {
this.length = initArray.arguments.length
for (var i = 0; i < this.length; i++)
this[i+1] = initArray.arguments[i]
}
var pos = new initArray(1,1,1,1,1,1,1,1,0,2,2,2,2,2,2,2,2);
var numMoves=0;
function display(pos) {
for (var i=0; i<17; i++) {
document.forms[0].elements[i].value = pos[i];
document.forms[0].elements['start'].value="Re"
}
if (numMoves> 1) win();
}
function move(num) {
```

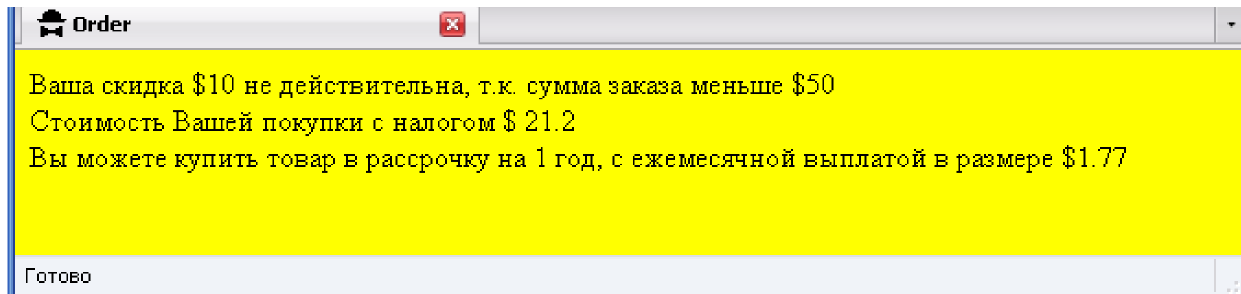
```
// num is the number of the field the user clicked - not the image!
// move '1's to adjacent horizontal space
if ((num > 13 && num < 16 && pos[num]==1 && pos[num+1]==0) ||
(num > 10 && num < 13 && pos[num]==1 && pos[num+1]==0) ||
(num > 5 && num < 10 && pos[num]==1 && pos[num+1]==0) ||
(num > 2 && num < 5 && pos[num]==1 && pos[num+1]==0) ||
(num > -1 && num < 2 && pos[num]==1 && pos[num+1]==0)) {
pos[num]=0;
pos[num+1]=1
numMoves++;
//alert("1");
}
// move '2's to adjacent horizontal space
else if ((num > 0 && num < 3 && pos[num]==2 && pos[num-1]==0) ||
(num > 3 && num < 6 && pos[num]==2 && pos[num-1]==0) ||
(num > 6 && num < 11 && pos[num]==2 && pos[num-1]==0) ||
(num > 11 && num < 14 && pos[num]==2 && pos[num-1]==0) ||
(num > 14 && num < 17 && pos[num]==2 && pos[num-1]==0)) {
pos[num]=0;
pos[num-1]=2
numMoves++;
//alert("2");
}

/**************************** ***************************/

// move '1's to adjacent vertical space
else if ((num < 14 && num > 10 && pos[num]==1 && pos[num+3]==0) ||
(num < 11 && num > 7 && pos[num]==1 && pos[num+3]==0) ||
(num < 6 && num > 2 && pos[num]==1 && pos[num+3]==0) ||
(num < 3 && num > -1 && pos[num]==1 && pos[num+3]==0)) {
pos[num]=0;
pos[num+3]=1;
numMoves++;
//alert("3");
}
// move '2's to adjacent vertical space
else if ((num < 6 && num > 2 && pos[num]==2 && pos[num-3]==0) ||
(num < 9 && num > 5 && pos[num]==2 && pos[num-3]==0) ||
(num < 14 && num > 10 && pos[num]==2 && pos[num-3]==0) ||
(num < 17 && num > 13 && pos[num]==2 && pos[num-3]==0)) {
pos[num]=0;
pos[num-3]=2;
numMoves++;
//alert("4");
}

/********************** *******************************/

// skip '1' over '2' horizontally
else if (pos[num]==1 && pos[num+2]==0 && pos[num+1]==2 &&
(num==14 || num==11 || num==8 || num==7 || num==6 ||
num==3 || num==0)) {
```

```
pos[num]=0;
pos[num+2]=1;
numMoves++;
//alert("5");
}
// skip '2' over '1' horizontally
else if (pos[num]==2 && pos[num-2]==0 && pos[num-1]==1 &&
(num==2 || num==5 || num==8 || num==9 || num==10 ||
num==13 || num==16)) {
pos[num]=0;
pos[num-2]=2;
numMoves++;
//alert("6");
}

/************************* *******************************/

// skip '1' over '2' vertically
else if (pos[num]==1 && pos[num+6]==0 && pos[num+3]==2 &&
(num==10 || num==9 || num==8 || num==5 || num==2 ||
num==1 || num==0)) {
pos[num]=0;
pos[num+6]=1;
numMoves++;
//alert("7");
}
// skip '2' over '1' horizontally
else if (pos[num]==2 && pos[num-6]==0 && pos[num-3]==1 &&
(num==6 || num==7 || num==8 || num==11 || num==14 ||
num==15 || num==16)) {
pos[num]=0;
pos[num-6]=2;
numMoves++;
//alert("8");
}

display(pos);
}

function win() {
if (pos[0]== 2 & pos[1]== 2 & pos[2]== 2 & pos[3]== 2 &
pos[4]== 2 & pos[5]== 2 & pos[6]== 2 & pos[7]== 2 & pos[8]== 0) {
if (confirm('You did it! Do you want to restart?')) newgame();
}
}

function newgame() {
var i;
for (i=0; i<8; i++) pos[i]=1;
pos[8]=0;
for (i=9; i<17; i++) pos[i]=2;
numMoves=0;
display(pos);
```

```
}

// -->
</SCRIPT>
```

Задание: реализовать приведенные игры и снабдить основные моменты кода комментариями.

Лабораторная работа №4. Условный оператор и конструкция выбора в php.




```
<html>
<head>
<title>Order</title>
</head>
<body bgcolor=yellow>
<?PHP
if (!isset($_POST[Submit]))
{
        print("<form action=\"Order.php\" method=POST>");
        print("<table> <tr><td>Quantity</td>");
        print("<td><input type=text name=\"Quantity\" size=20></td></tr>");
        print("<tr><td>Discount</td>");
        print("<td><input type=text name=\"Discount\" size=20></td></tr></table>");
        print("<input type=submit name=\"Submit\" value=\"OK\"> ");
        print("<input type=reset name=\"Reset\" value=\"Cancel\">");
        print("</form>");
}
else
{
        $Cost=20.00;
        $Tax=0.06;
        if ($_POST[Quantity])
                {
                $Quantity=abs($_POST[Quantity]);
                $Discount=abs($_POST[Discount]);
                $TotalCost=$Quantity*$Cost;
                if (($TotalCost<50) and ($Discount))
                {
                print ("Ваша скидка \$$Discount не действительна, т.к. сумма заказа меньше $50 <br>");
                }
                if ($TotalCost>=50)
                        {
                        $TotalCost=$TotalCost-$Discount;
                        }
                $TotalCost=$TotalCost+$TotalCost*$Tax;
                print("Стоимость Вашей покупки с налогом $ $TotalCost <BR> \n");
```

```
		print("Вы можете купить товар в рассрочку на 1 год, с ежемесячной выплатой в размере $");
		$Payments=round($TotalCost/12,2);
		print("$Payments \n");
		}
	else
		{
		print("Вернитесь назад и Введите количество");
}		}
?>
</body>
</html>
```

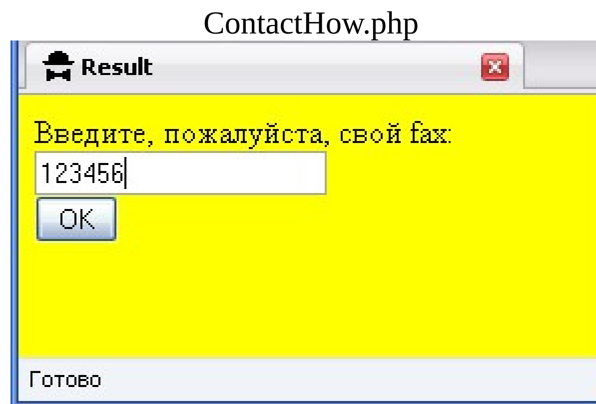
Использование управляющей структуры switch

Contact.html

```
<html>
<head>
<title>Contact</title>
</head>
<body bgcolor=yellow>
<form action="ContactHow.php" method=POST>
<table> <tr><td> First Name</td>
<td><input type=text name="FirstName" size=20></td></tr>
<tr><td>Last Name</td>
<td><input type=text name="LastName" size=20></td></tr>
<tr><td> Comments</td>
<td><textarea name="Comments" rows=5 cols=40></textarea></td></tr></table>
Способ связи
<p><select name="ContactHow" size=1>
<option value="Telephone">Телефон</option>
<option value="Pochta">Почта</option>
<option value="Email">E-mail</option>
<option value="Fax">Fax</option>
</select>
```

```
<p><input type=submit name="Submit" value="OK"> 
<input type=reset name="Reset" value="Cancel">
</form>
</body>
</html>
```

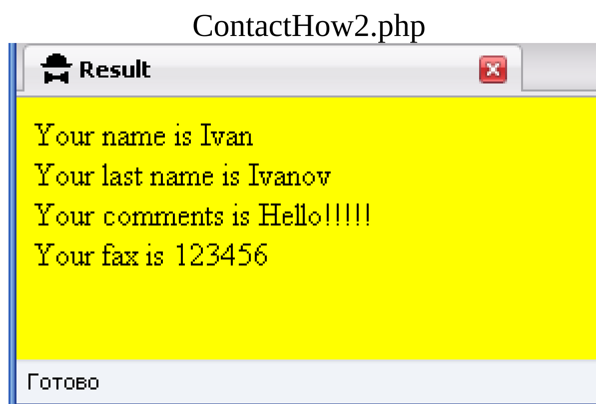
ContactHow.php




```
<html>
<head>
<title>Result</title>
</head>
<body bgcolor=yellow>
<?PHP
print("<form action=\"ContactHow2.php\" method=post>");
print("<input type=hidden name=\"FirstName\" value=$_POST[FirstName]>");
print("<input type=hidden name=\"LastName\" value=$_POST[LastName]>");
print("<input type=hidden name=\"Comments\" value=$_POST[Comments]>");

switch ($_POST[ContactHow])
{
case "Telephone":
        print("Введите, пожалуйста, свой телефон:<br>");
        print("<input type=text name=\"Telephone\" size=20><br>");
        break;
case "Pochta":
        print("Введите, пожалуйста, свой адрес:<br>");
        print("<input type=text name=\"Adress\" size=20><br>");
        break;
case "Email":
        print("Введите, пожалуйста, свой e-mail:<br>");
        print("<input type=text name=\"Email\" size=20><br>");
```

```
        break;
case "Fax":
        print("Введите, пожалуйста, свой fax:<br>");
        print("<input type=text name=\"Fax\" size=20><br>");
        break;
}
print("<input type=submit name=\"Submit\" value=\"OK\"></form>");

?>
</body>
</html>
```

ContactHow2.php

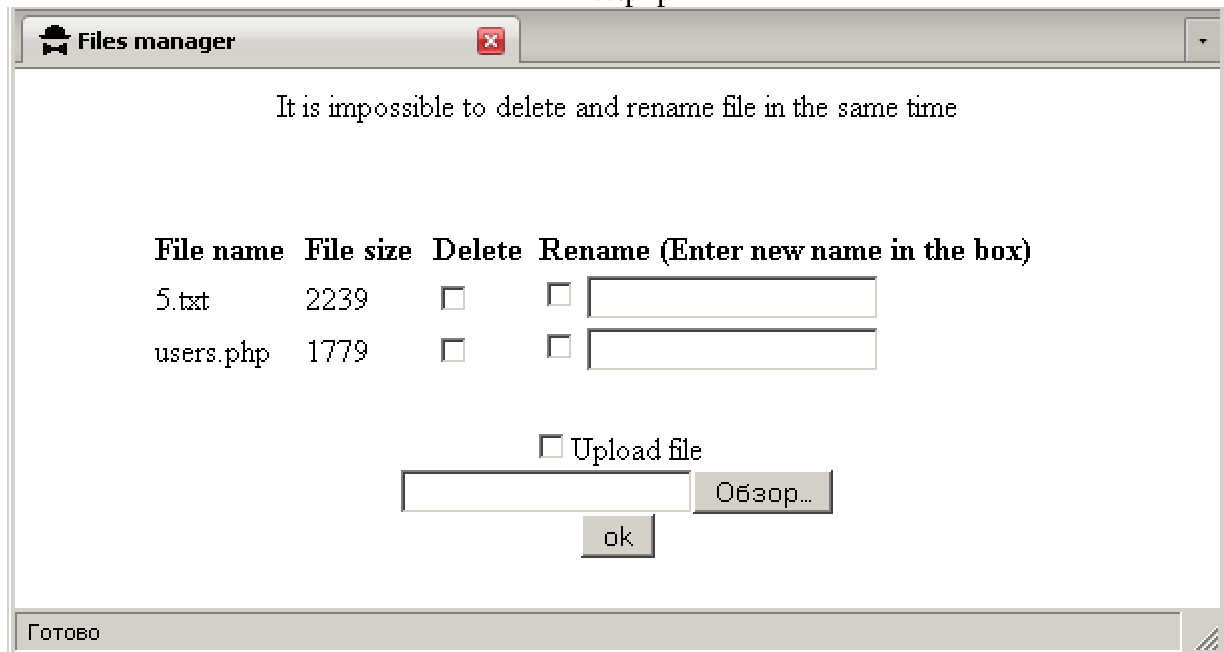


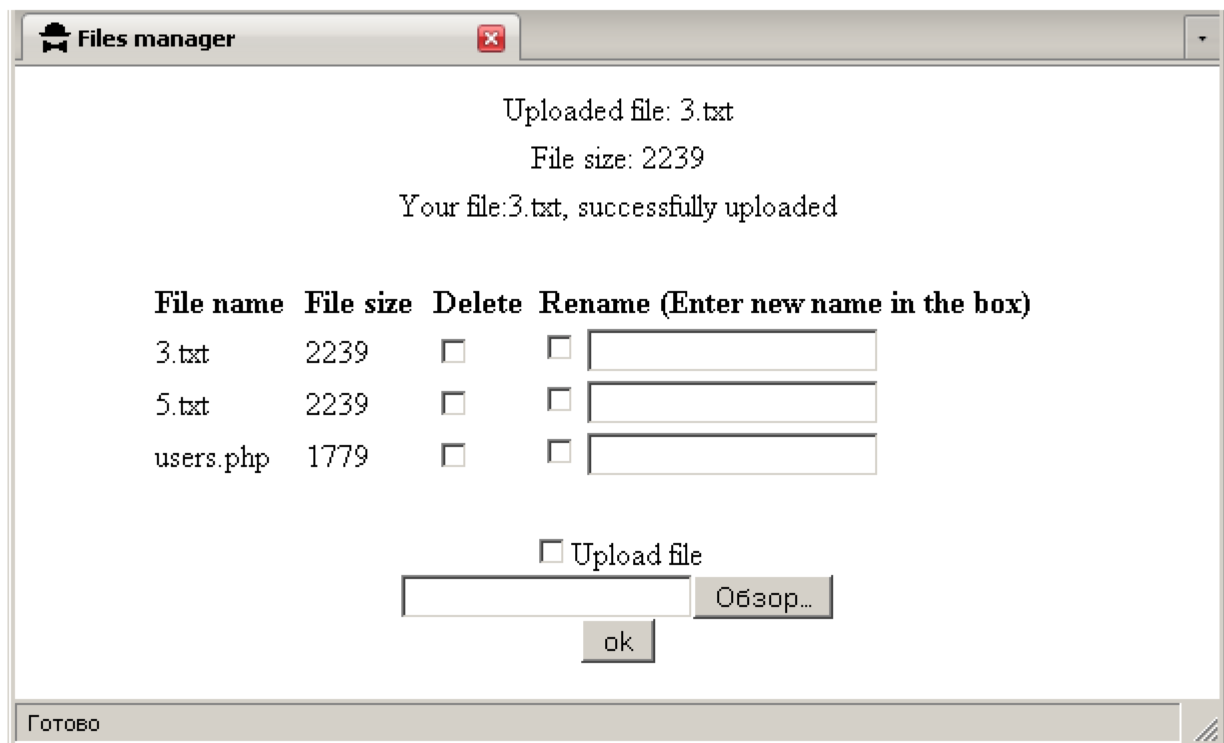

```
<html>
<head>
<title>Result</title>
</head>
<body bgcolor=yellow>
<?PHP
print("Your name is $_POST[FirstName] <br>");
print("Your last name is $_POST[LastName]<br>");
print("Your comments is $_POST[Comments]<br>");

if (isset($_POST[Telephone]))
{print("Your telephone is $_POST[Telephone]");}
if (isset($_POST[Adress]))
{print("Your adress is $_POST[Adress]");}
if (isset($_POST[Email]))
{print("Your e-mail is $_POST[Email]");}
if (isset($_POST[Fax]))
{print("Your fax is $_POST[Fax]");}

?>
</body>
</html>
```

Лабораторная работа №5. Работа с файлами и каталогами в php.

Создание панели управления файлами

files.php






```
<html>
<head>
<title>Files manager</title>
</head>
<body>
<table border=0 width=80% align=center>
<?PHP
if ($_POST["Upload"])
{
```

```
	print("<tr><td colspan=4 align=center>Uploaded file: $File_name</td></tr>");
	print("<tr><td colspan=4 align=center>File size: $File_size</td></tr>");
	if (copy($File,"USERS/$File_name"))
	{
		print("<tr><td colspan=4 align=center>Your file:$File_name, successfully uploaded</td></tr>");
	}
	else
	{
		print("Upload was not success");
	}
	unlink($File);
	print("<tr><td colspan=4> </td></tr>");
}

if ($Delete)
{
	for($i=0;$i<count($Delete);$i++)
	{
	if ($Delete[$i]==$Rename[$i])
	{ print("<center>It is impossible to delete and rename file in the same time</center>");}
	else
	{
		if (unlink("USERS/$Delete[$i]"))
		{
			print("<tr><td colspan=4 align=center> Your file, $Delete[$i],			was
deleted</td></tr>");
		}
		else
		{
			print("<tr><td colspan=4 align=center> Your file, $Delete[$i],			cannot
be deleted</td></tr>");
		}
	}
	}
	print("<tr><td colspan=4 align=center> </td></tr>");
}

if ($Rename)
{
	for($i=0;$i<count($Rename);$i++)
	{
	if ($Delete[$i]==$Rename[$i])
	{}
	else
	{
		$OldName="USERS/".$Rename[$i];
		$Newfile=$Rename[$i];
		$New="USERS/".$NewName[$Newfile];
		if (rename("$OldName","$New"))
		{
			print("<tr><td colspan=4 align=center> Your file, $Rename[$i],			was
renamed in $NewName[$Newfile]</td></tr>");
		}
```

```
			else
			{
					print("<tr><td colspan=4 align=center> Your file, $Rename[$i],
cannot be renamed</td></tr>");
			}
		}
		}
		print("<tr><td colspan=4 align=center> </td></tr>");
}

print("<form action=\"files.php\" method=post enctype=\"multipart/form-data\">\n");
print("<tr><td><b>File name</b></td>
	  <td><b>File size</b></td>
	  <td><b>Delete</b></td>
	  <td><b>Rename (Enter new name in the box)</b></td></tr>");
$Open=opendir("USERS");
while ($Files=readdir($Open))
{
		$FileName="USERS/".$Files;
		if (is_file($FileName))
		{
			$Size=filesize($FileName);
			print("<tr><td>$Files</td>
				  <td>$Size</td>
				  <td><input type=checkbox name=\"Delete[]\"
			value=\"$Files\"</td>
				  <td><input type=checkbox name=\"Rename[]\" value=\"$Files\">
				     <input type=text name=\"NewName[$Files]\"></td></tr>\n");
		}
}
closedir($Open);
print("<tr><td colspan=4 align=center> </td></tr>");
print("<tr><td colspan=4 align=center>
	  <input type=checkbox name=\"Upload\" value=\"yes\">Upload file<br>
	  <input type=file name=\"File\" size=20><br>
	  <input type=submit name=\"Submit\" value=\"ok\"></td></tr>");
print("</form>");
?>
</table>
</body> </html>
```

Лабораторная работа №6. Регулярные выражения.

Регулярные выражения.

Цель работы: Изучить приемы работы с регулярными выражениями в PHP.

Регулярное выражение (**regular expression**, **regexp**, **регэксп**) - механизм, позволяющий задать шаблон для строки и осуществить поиск данных, соответствующих этому шаблону в заданном тексте. Кроме того, дополнительные функции по работе с regexp'ами позволяют получить найденные данные в виде массива строк, произвести замену в тексте по шаблону, разбиение строки по шаблону и т.п. Однако главной их функцией, на которой основаны все остальные, является именно функция поиска в тексте данных, соответствующих шаблону, описанному в синтаксисе регулярных выражений.

Очень часто регулярные выражения используются для того, чтобы проверить, является ли данная строка строкой в необходимом формате. Например следующий regexp предназначен для проверки того, что строка содержит корректный e-mail адрес:

/^\w+([\.\w]+)*\w@\w((\.\w)*\w+)*\.\w{2,3}$/

Регулярные выражения пришли к нам из **Unix** и **Perl**. В **PHP** существует два различных механизма для обработки регулярных выражений: POSIX-совместимые и Perl-совместимые. Их синтаксис во многом похож, однако Perl-совместимые регулярные выражения более мощные и, к тому же, работают намного быстрее (в некоторых случаях до 10 раз быстрее). Поэтому здесь мы будем вести речь только о Perl-совместимых регулярных выражениях.

Кстати, необходимо заметить, что полное описание синтаксиса регулярных выражений, имеющееся в PHP Manual, занимает более 50 килобайт и, естественно, здесь мы не будем рассматривать весь синтаксис. Нам необходимы только основы, которые помогут вам понять, как именно пишутся регулярные выражения.

Сутью механизма регулярных выражений является то, что они позволяют задать шаблон для **нечеткого** поиска по тексту. Например, если перед вами стоит задача найти в тексте определенное слово, то с этой задачей хорошо справляются и обычные функции работы со строками. Однако если вам нужно найти "то, не знаю что", о чем вы можете сказать только то, как **приблизительно** это должно выглядеть - то здесь без регулярных выражений просто не обойтись. Например, вам необходимо найти в тексте информацию, про которую вам известно только то, что это "3 или 4 цифры после которых через пробел идет 5 заглавных латинских букв", то вы сможете сделать это очень просто, воспользовавшись следующим регулярным выражением:

/\d{3,4}\s[A-Z]{5}/

Синтаксис регулярных выражений

Регулярные выражения, как уже было сказано выше, представляют собой строку. Строка всегда начинается с символа разделителя, за которым следует непосредственно регулярное выражение, затем еще один символ разделителя и потом необязятельный список модификаторов. В качестве символа разделителя обычно используется слэш ('/'). Таким образом в следующем регулярном выражении: /\d{3}-\d{2}/m, символ '/' является разделителем, строка '\d{3}-\d{2}' - непосредственно регулярным выражением, а символ 'm', расположенный после второго разделителя - это модификатор.

Основой синтаксиса регулярных выражений является тот факт, что некоторые символы, встречающиеся в строке рассматриваются не как обычные символы, а как имеющие специальное значение (т.н. метасимволы). Именно это решение позволяет работать всему механизму регулярных выражений. Каждый метасимвол имеет свою собственную роль в синтаксисе регулярных выражений. Далее мы рассмотрим все эти метасимволы.

Одним из самых важных метасимволов является символ обратного слэша ('\'). Если в строке встречается этот символ, то парсер рассматривает символ, непосредственно следующий за ним двояко:

- если следующий символ в обычном режиме имеет какое-либо специальное значение, то он теряет это свое специальное значение и рассматривается как обычный символ. Это совершенно необходимо для того, чтобы иметь возможность вставлять в строку специальные символы, как обычные. Например метасимвол '.', в обычном режиме означает "любой единичный символ", а '\.' означает просто точку. Также можно лишить специального значения и сам этот символ: '\\'.
- если следующий символ в обычном режиме не имеет никакого специального значения, то он может получить такое значение, будучи соединенным с символом '\'. К примеру символ 'd' в обычном режиме воспринимается просто как буква, однако, будучи соединенной с обратным слэшем ('\d') становится метасимволом, означающим "любая цифра".

Существует множество символов, которые образуют метасимволы в паре с обратным слэшем. Как правило подобные пары используются для того, чтобы показать, что на этом месте в строке должен находиться символ, с кодом, который не имеет соответствующего ему изображения или же символ, принадлежащий какой-то определенной группе символов. Ниже приведены некоторые наиболее употребительные:

Метасимвол	**Значение**
Метасимволы для задания символов, не имеющих изображения	
\n	Символ перевода строки (код 0x0A)
\r	Символ возврата каретки (код 0x0D)

\t	Символ табуляции (код 0x09)
\xhh	Вставка символа с шестнадцатиричным кодом 0xhh, например \x41 вставит латинскую букву 'A'
Метасимволы для задания групп символов	
\d	Цифра (0-9)
\D	Не цифра (любой символ кроме символов 0-9)
\s	Пустой символ (обычно пробел и символ табуляции)
\S	Непустой символ (все, кроме символов, определяемых метасимволом \s)
\w	"Словесный" символ (символ, который используется в словах. Обычно все буквы, все цифры и знак подчеркивания '_'
\W	Все, кроме символов, определяемых метасимволом \w

Несколько простейших примеров.

Regexp	**Комментарии**
/\d\d\d/	Любое трехзначное число ('123', '719', '001')
/\w\s\d\d/	Буква, пробел (или табуляция) и двузначное число ('A 01', 'z 45', 'S 18')
/\d and \d/	Любая из следующих строк: '1 and 2', '9 and 5', '3 and 4'.

Синтаксис регулярных выражений имеет средства для определения собственных подмножеств символов. Например вам может понадобиться задать условие, что в этом месте строки должна находиться шестнадцатиричная цифра или еще что-то подобное. Для описания таких подмножеств применяются символы квадратных скобок '[]'. Квадратные скобки, встреченные внутри регулярного выражения считаются одним символом, который может принимать значения, перечисленные внутри этих скобок.

Есть небольшая тонкость в том, как работают метасимволы внутри квадратных скобок. Дело в том, что в синтаксисе регулярных выражений существует еще множество метасимволов, но практически все они работают только **вне** секций описаний подмножеств. Единственные метасимволы, которые работают внутри этих секций это:

Обратный слэш ('\'). Т.е. все метасимволы из приведенной ранее таблицы будут работать.

Минус ('-'). Используется для задания набора символов из одного промежутка (например все цифры могут быть заданы как '0-9')

Символ '^'. Если этот символ стоит **первым** в секции задания подмножества символов (и только в этом случае!) он будет рассматриваться как символ отрицания. Т.о. можно задать все символы, которые **не описаны** в данной секции.

Несколько примеров:

Regexp	**Комментарии**
[0-9A-Fa-f]	Цифра в шестнадцатиричной системе счисления
[\dA-Fa-f]	То же самое, но с использованием метасимвола
[02468]	Четная цифра

[\^d]	Все, кроме цифр (аналог метасимвола \D)
[a^b]	Любой из символов 'a', 'b', '^'. Заметьте, что здесь символ '^' не имеет какого-либо специального значения, потому что стоит не на первой позиции внутри квадратных скобок.

Теперь необходимо рассмотреть еще несколько метасимволов. Как уже было сказано ранее, все они работают только вне секций описаний подмножеств символов (вне квадратных скобок).

Символы '^' и '{$}'. Они используются для того, что того, чтобы указать парсеру регулярных выражений на то, чтобы он обратил внимание на положение искомого текста в строке. Символ '^' указывает, что искомый текст должен находиться в начале строки, символ '{$}' наоборот, указывает, что искомый текст должен находиться в конце строки. Посмотрим, как это работает на примере:

Допустим, у нас есть текст:

12 aaa bbb

aaa 27 ccc

aaa aaa 45

И регулярное выражение для поиска чисел в этом тексте: /\d\d/m (не обращайте пока внимания на модификатор). Поиск по этому регулярному выражению вернет нам 3 значения: '12', '27', '45'. Теперь ограничим поиск, указав, где именно внутри строки должен располагаться текст: /^\d\d/m. Здесь результат будет только один - '12', потому что только это число располагается в начале строки. Аналогично, регулярное выражение /\d\d$/m вернет результат '45'.

Символ точки '.'. Этот метасимвол указывает, что на данном месте в строке может находиться любой символ (за исключением символа перевода строки). Очень удобно использовать его, если вам нужно "пропустить" какую-нибудь букву в слове при проверке. Например регулярное выражение /.bc/ найдет в тексте и 'abc' и 'Abc' и 'Zbc' и '5bc'.

Символ вертикальной черты '|'. Используется для задания списка альтернатив. Например регулярное выражение:

/(красное|зеленое) яблоко/

Найдет в тексте все словосочетания 'красное яблоко' и 'зеленое яблоко'.

Символы круглых скобок '(' и ')'. Эти символы позволяют получить из искомой строки дополнительную информацию. Обычно, если парсер регулярных выражений ищет в тексте информацию по заданному выражению и находит ее - он просто возвращает найденную строку. Однако, если он встречает внутри регулярного выражения круглые скобки, то он рассматривает содержимое этих скобок как еще одно регулярное выражение, по которому необходимо произвести поиск. Парсер рекурсивно вызывает сам себя для поиска по новому регулярному выражению и использует результаты поиска для дальнейшей обработки основного регулярного выражения. При этом, если поиск хотя бы по одному из внутренних регулярных выражений не увенчался успехом - поиск по всему регулярному выражению считается безуспешным.

Рассмотрим в качестве примера то, как работает парсер регулярных выражений в случае приведенного выше регулярного выражения о яблоках: /(красное зеленое) яблоко/.

1. Парсер начинает разбор регулярного выражения и встречает выражение в скобках: (красное|зеленое)
2. Парсер вызывает себя для поиска по найденному регулярному выражению.
3. Получив результаты поиска парсер подставляет по очереди каждый из полученных результатов на место выражения в скобках и смотрит, удовлетворяет ли найденный результат всем условиям основного регулярного выражения (в данном случае смотрит, есть ли после найденного слова слово "яблоко").
4. Если все в порядке - результаты поиска по каждому из имеющихся регулярных выражений для этого случая возвращаются, если нет - парсер просто переходит к следующему найденному фрагменту. Результат поиска внутреннего регулярного выражения для этого фрагмента при этом теряется.

В качестве примера возьмем строку:

яблоко красное и зеленое яблоко и еще одно красное

яблоко и еще одно яблоко, зеленое

Поиск по внутреннему регулярному выражению даст 4 результата (выделены жирным шрифтом):

яблоко **красное** и **зеленое** яблоко и еще одно **красное** яблоко и еще одно яблоко, **зеленое**

Однако поиск по всему регулярному выражению даст всего 2 результата, потому как в остальных случаях условия основного регулярного выражения не выполняются:

яблоко красное и **зеленое яблоко** и еще одно **красное яблоко** и еще одно яблоко, зеленое

Необходимо заметить, что для этих двух случаев будет возвращен не только результат поиска по основному регулярному выражению, но и результат поиска по внутреннему регулярному выражению для каждого из найденных фрагментов. В большинстве случаев это полезно (пример - чуть позднее), но иногда наоборот, лучше избавиться от лишних результатов. В этом случае необходимо добавить символы '?:' непосредственно после открывающейся круглой скобки: /(?:красное|зеленое) яблоко/.

Теперь пример, когда получение результатов внутренних регулярных выражений может быть полезным. Допустим, нам необходимо проверить, является ли строка семизначным телефонным номером с указанием кода города и получить из нее код города и номер телефона:

/\((\d{3,5})\)\s+(\d{3}-\d{2}-\d{2})/

Давайте рассмотрим это регулярное выражение подробнее.

Первая круглая скобка здесь теряет свое специальное значение и будет рассматриваться как обычный символ:\(

Далее идет регулярное выражение в скобках (проверка кода города):(\d{3,5})

После этого идет закрывающая круглая скобка, которая также лишена своего специального значения из-за символа обратного слэша, стоящего перед ней:\)

Затем идет пропуск пустого места:\s+

И еще одно регулярное выражение в скобках, которое проверяет номер телефона: (\d{3}-\d{2}-\d{2})

Как видите, здесь есть 3 регулярных выражения - основное и два внутренних. При этом основное выражение позволяет нам проверить, имеет ли строка необходимый нам формат, а два внутренних - получить соответственно код города и номер телефона.

Посмотрим, как работает это регулярное выражение. Пусть у нас есть строка: "My phone is (095) 123-45-67". Результатами поиска будут 3 строки: '(095) 123-45-67', '095' и '123-45-67'.

Нам осталось рассмотреть еще одну группу метасимволов, определяющих количественные показатели (т.н. **quantifiers**). Как вы уже могли заметить ранее - очень часто бывает необходимо указать, что какой-то символ должен повторяться определенное количество раз. Конечно, можно просто указать его необходимое количество раз непосредственно в строке, но это, естественно не выход. Тем более, что очень часто встречаются ситуации, когда точное количество символов неизвестно. Поэтому синтаксис регулярных выражений содержит набор метасимволов, предназначенных именно для решения подобных задач. Каждый из описанных ниже метасимволов определяет количественную характеристику символа который находится **непосредственно** перед ним.

Звездочка '*'. Указывает, что символ должен быть повторен 0 или более раз (т.е. символ может отсутствовать или присутствовать в любых количествах). Пример: выражение /ab*c/ найдет строки 'ac', 'abc', 'abbc' и т.д.

Плюс '+'. Указывает, что символ должен быть повторен 1 или более раз (т.е. символ обязан присутствовать и может присутствовать в любых количествах). Пример: выражение /ab+c/ найдет строки 'abc', 'abbc', 'abbbc' и т.д., но не найдет строку 'ac'.

Знак вопроса '?'. Указывает, что символ моет как присутствовать, так и нет, но при этом не может повторяться более одного раза. Пример: выражение /ab?c/ найдет строки 'ac' и 'abc', но не найдет строку 'abbc'.

Фигурные скобки '{' и '}'. Определяют количественную характеристику символа. Внутри скобок через запятую перечисляются минимальное и максимальное количество повторений символа. При этом любой из параметров может быть опущен, а кроме того можно задать точное количество повторений, указав только одно число. Примеры:

{2,4} - символ должен повториться минимум 2 раза, но не более 4.

{,5} - символ может отсутствовать (т.к. не задано минимальное количество повторений), но если присутствует, то не должен повторяться более 5 раз.

{3,} - символ должен повторяться минимум 3 раза, но может быть и больше.

{4} - символ должен повторяться ровно 4 раза

Есть еще одна тонкость в использовании метасимвола '?'. Посмотрите на такое выражение: /.+a/. Ожидается, что оно вернет нам часть текста до первого вхождения символа 'a' в этот текст. На самом деле оно будет работать несколько не так, как ожидается и результатом поиска будет весь текст до **последнего** вхождения символа 'a'. Дело в том, что по умолчанию количественные метасимволы "жадничают" и пытаются захватить как можно больший кусок текста. Если это не нужно (как а нашем случае), то необходимо "отучить" их от жадности, указав знак '?' после количественного метасимвола: /.+?a/. После этого выражение будет работать так как надо.

Модификаторы регулярных выражений

Механизм регулярных выражений позволяет добавлять модификаторы, влияющие на обработку регулярного выражения. Ниже рассмотрены наиболее употребительные.

i	Включение режима case-insensitive, т.е. большие и маленькие буквы в выражении не различаются.
m	Указывает на то, что текст, по которому ведется поиск, должен рассматриваться как состоящий из нескольких строк. По умолчанию механизм регулярных выражений рассматривает текст как одну строку вне зависимости от того, чем она является на самом деле. Соответственно метасимволы ' ' и '$' указывают на начало и конец всего текста. Если же этот модификатор указан, то они будут указывать соответственно на начало и конец каждой строки текста.
s	По умолчанию метасимвол '.' не включает в свое определение символ перевода строки. Т.е. для многострочного текста выражение /.+/ вернет только первую строку, а не весь текст, как ожидается. Указание этого модификатора снимает это ограничение.
U	Делает все количественные метасимволы "не жадными" по умолчанию (про "жадность" количественных метасимволов см. выше)

Задания к лабораторной работе

1. Найти в документации описание следующих функций работы с регулярными выражениями:

- preg_match()
- preg_match_all()
- preg_replace()
- preg_replace_callback()
- preg_split()
- preg_quote()
- preg_grep()
- preg_quote()

Реализовать примеры их использования.

2. Составьте регулярные выражения для маскирования тегов HTML (20-30 на выбор).
3. Составьте регулярное выражение для проверки значения переменной:

- Быть целым числом.
- Быть вещественным числом.
- Быть идентификатором.
- Быть правильным телефонным номером (например 37-81-40).

- o Быть правильным телефонным номером с кодом города (например (231) 5-94-00).
- o Быть не числом.
- o Не содержать цифр.
- o Не содержать букв.

4. Построить регулярное выражение, возвращающее значение параметров тегов html-документа, содержащих URL.

Лабораторная работа №7. Подключение к базе данных из php-скрипта.

Name	Email	Comments
Ivan Ivanov	12345@mail.ru	Hi!!!!
Kobzar Stanislav	234@mail.ru	Hello!!!

SelectDB.php

```
<html><head>
<title> Create DB</title></head><body>
<?PHP
$Host="localhost";
$User="root";
$Password="";
$DBName="mybase";
$Link=mysql_connect($Host, $User, $Password);
if (mysql_select_db($DBName, $Link))
{
print("DB is connected");
}
else
{
print("Error");
}
mysql_close($Link);
?></body></html>
```

CreateTable.php

```
<html><head>
<title> Create Table</title></head><body>
<?PHP
$Host="localhost";
$User="root";
$Password="mybase";
$DBName="mybase";
$TableName="Table1";
$Link=mysql_connect($Host, $User, $Password);
$query="CREATE TABLE $TableName (id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
            FirstName TEXT,
            LastName TEXT,
            Email TEXT,
            Comments TEXT)";
mysql_db_query($DBName,$query,$Link) or die("Creating table error".mysql_error());
```

```
echo "Таблица успешно создана!";
mysql_close($Link);
?>
</body>
</html>
```

HandleForm.php

```
<html><head>
<title> Create DB</title></head><body>
<?PHP
$Host="localhost";
$User="root";
$Password="";
$DBName="yulia1";
$TableName="mybase";
$FirstName=trim($_POST[FirstName]);
$LastName=trim($_POST[LastName]);
$Email=trim($_POST[Email]);
$Comments=trim($_POST[Comments]);
$Link=mysql_connect($Host, $User, $Password);
$query="INSERT INTO $TableName values(0,'$FirstName','$LastName','$Email','$Comments')";
mysql_db_query($DBName,$query,$Link) or die("Adding data error".mysql_error());
mysql_close($Link);
?></body></html>
```

DisplayDB.php

```
<html><head>
<title> Create DB</title></head><body>
<?PHP
$Host="localhost";
$User="root";
$Password="";
$DBName="mybase";
$TableName="Table1";
$Link=mysql_connect($Host, $User, $Password);
$query="SELECT * FROM $TableName";
$query_result=mysql_db_query($DBName,$query,$Link) or die("Display error".mysql_error());
print("<table border=1 width=80% align=center>\n");
print("<tr><td>Name</td><td>Email</td><td>Comments</td></tr>");
while($Row=mysql_fetch_array($query_result))
{
print("<tr><td>$Row[FirstName]
$Row[LastName]</td><td>$Row[Email]</td><td>$Row[Comments]</td></tr>\n");
}
print("</table>");
mysql_free_result($query_result);
mysql_close($Link);
?>
</body></html>
```

Лабораторная работа №8. Выборка информации из базы данных и представление ее на web-странице.

Базовые сведения:

Основы клиент-серверных технологий

Если речь идет о сервере, невольно всплывает в памяти понятие клиента. Все потому, что эти два понятия неразрывно связаны. Объединяет их компьютерная архитектура клиент-сервер. Обычно, когда говорят «сервер», имеют в виду сервер в архитектуре клиент-сервер, а когда говорят «клиент» – имеют в виду клиент в этой же архитектуре.

Суть клиент-серверной архитектуры в том, чтобы разделить функции между двумя подсистемами: клиентом, который отправляет запрос на выполнение каких-либо действий, и сервером, который выполняет этот запрос. Взаимодействие между клиентом и сервером происходит посредством стандартных специальных протоколов, таких как TCP/IP и z39.50. На самом деле протоколов очень много, они различаются по уровням.

Сервер представляет собой набор программ, которые контролируют выполнение различных процессов. Соответственно, этот набор программ установлен на каком-то компьютере. Часто компьютер, на котором установлен сервер, и называют сервером. Основная функция компьютера-сервера – по запросу клиента запустить какой-либо определенный процесс и отправить клиенту результаты его работы.

Клиентом называют любой процесс, который пользуется услугами сервера. Клиентом может быть как пользователь, так и программа. Основная задача клиента – выполнение приложения и осуществление связи с сервером, когда этого требует приложение. То есть клиент должен предоставлять пользователю интерфейс для работы с приложением, реализовывать логику его работы и при необходимости отправлять задания серверу.

Взаимодействие между клиентом и сервером начинается по инициативе клиента. Клиент запрашивает вид обслуживания, устанавливает сеанс, получает нужные ему результаты и сообщает об окончании работы.

Услугами одного сервера чаще всего пользуется несколько клиентов одновременно. Поэтому каждый сервер должен иметь достаточно большую производительность и обеспечивать безопасность данных. Логичнее всего устанавливать сервер на компьютере, входящем в какую-либо сеть, локальную или глобальную. Однако можно устанавливать сервер и на отдельно стоящий компьютер (тогда он будет являться одновременно и клиентом и сервером).

Существует множество типов серверов. Вот лишь некоторые из них.

Видеосервер

Такой сервер специально приспособлен к обработке изображений, хранению видеоматериалов, видеоигр и т.п. В связи с этим компьютер, на котором установлен видеосервер, должен иметь высокую производительность и большую память.

Поисковый сервер

предназначен для поиска информации в Internet.

Почтовый сервер

предоставляет услуги в ответ на запросы, присланные по электронной почте.

Сервер WWW

предназначен для работы в Internet.

Сервер баз данных

выполняет обработку запросов к базам данных.

Сервер защиты данных

предназначен для обеспечения безопасности данных (содержит, например, средства для идентификации паролей).

Сервер приложений

предназначен для выполнения прикладных процессов. С одной стороны взаимодействует с клиентами, получая задания, а с другой – работает с базами данных, подбирая необходимые для обработки данные.

Сервер удаленного доступа

обеспечивает коллективный удаленный доступ к данным. Файловый сервер обеспечивает функционирование распределенных ресурсов, предоставляет услуги поиска, хранения, архивирования данных и возможность одновременного доступа к ним нескольких пользователей.

Обычно на компьютере-сервере работает сразу несколько программ-серверов. Одна занимается электронной почтой, другая распределением файлов, третья предоставляет web-страницы.

Из всех типов серверов нас в будет интересовать сервер WWW. Часто его называют web-сервером, http-сервером или даже просто сервером.

Web-сервер

Во-первых, это хранилище информационных ресурсов.

Во-вторых, эти ресурсы хранятся и предоставляются пользователям в соответствии со стандартами Internet (такими, как протокол передачи данных HTTP).

Работа с документами web-сервера осуществляется при помощи браузера (например, IE, Opera или Mozilla), который отсылает серверу запросы, созданные в соответствии с протоколом HTTP. В процессе выполнения задания сервер может связываться с другими серверами.

Далее, говоря «сервер», мы будем подразумевать web-сервер.

В качестве примеров web-серверов можно привести сервер Apache группы Apache, Internet Information Server (IIS) компании Microsoft, SunOne фирмы Sun Microsystems,WebLogic фирмы BEA Systems, IAS (Inprise Application Server) фирмы Borland, WebSphere фирмы IBM, OAS (Oracle Application Server).

Протокол HTTP и способы передачи данных на сервер Internet построен по многоуровневому принципу, от физического уровня, связанного с физическими аспектами передачи двоичной информации, и до прикладного уровня, обеспечивающего интерфейс между пользователем и сетью.

HTTP

(HyperText Transfer Protocol, протокол передачи гипертекста) – это протокол прикладного уровня, разработанный для обмена гипертекстовой информацией в Internet.

HTTP предоставляет набор методов для указания целей запроса, отправляемого серверу. Эти методы основаны на дисциплине ссылок, где для указания ресурса, к которому должен быть применен данный метод, используется универсальный идентификатор ресурсов (Universal Resource Identifier) в виде местонахождения ресурса (Universal Resource Locator, URL) или в виде его универсального имени (Universal Resource Name, URN).

Сообщения по сети при использовании протокола HTTP передаются в формате, схожем с форматом почтового сообщения Internet (RFC-822) или с форматом сообщений MIME (Multipurpose Internet Mail Exchange).

HTTP используется для коммуникаций между различными пользовательскими программами и программами-шлюзами, предоставляющими доступ к существующим Internet-протоколам, таким как SMTP (протокол электронной почты), NNTP (протокол передачи новостей), FTP (протокол передачи файлов), Gopher и WAIS. HTTP разработан для того, чтобы позволять таким шлюзам через промежуточные программы-серверы (proxy) передавать данные без потерь.

Протокол реализует принцип запрос/ответ. Запрашивающая программа – клиент инициирует взаимодействие с отвечающей программой – сервером и посылает запрос, содержащий:

- метод доступа;
- адрес URL;
- версию протокола;
- сообщение с информацией о типе передаваемых данных, информацией о клиенте, пославшем запрос, и, возможно, с содержательной частью (телом) сообщения.

Ответ сервера содержит:

- строку состояния, в которую входит версия протокола и код возврата (успех или ошибка);
- сообщение, в которое входит информация сервера, метаинформация (т.е. информация о содержании сообщения) и тело сообщения.

В протоколе не указывается, кто должен открывать и закрывать соединение между клиентом и сервером. На практике соединение, как правило, открывает клиент, а сервер после отправки ответа инициирует его разрыв.

Форма запроса клиента

Клиент отсылает серверу запрос в одной из двух форм: в полной или сокращенной. Запрос в первой форме называется соответственно полным запросом, а во второй форме – простым запросом.
Простой запрос содержит метод доступа и адрес ресурса. Формально это можно записать так:
<Простой-Запрос> := <Метод> <символ пробел>
<Запрашиваемый-URL> <символ новой строки>
В качестве метода могут быть указаны GET, POST, HEAD, PUT, DELETE и другие.
В качестве запрашиваемого URL чаще всего используется URL-адрес ресурса.
Пример простого запроса:
GET http://www.ru/
Здесь GET – это метод доступа, т.е. метод, который должен быть применен к запрашиваемому ресурсу, а http://www.ru/ – это URL-адрес запрашиваемого ресурса.
Полный запрос содержит строку состояния, несколько заголовков (заголовок запроса, общий заголовок или заголовок содержания) и, возможно, тело запроса. Формально общий вид полного запроса можно записать так:
<Полный запрос> := <Строка Состояния>
(<Общий заголовок>|<Заголовок запроса>|
<Заголовок содержания>)
<символ новой строки>
[<содержание запроса>]
Квадратные скобки здесь обозначают необязательные элементы заголовка, через вертикальную черту перечислены альтернативные варианты.
Элемент <Строка состояния> содержит метод запроса и URL ресурса (как и простой запрос) и, кроме того, используемую версию протокола HTTP. Например, для вызова внешней программы можно задействовать следующую строку состояния:
POST http://www.ru/cgi-bin/test HTTP/1.0
В данном случае используется метод POST и протокол HTTP версии 1.0.
В обеих формах запроса важное место занимает URL запрашиваемого ресурса. Чаще всего URL используется в виде URL-адреса ресурса. При обращении к серверу можно применять как полную форму URL, так и упрощенную.
Полная форма содержит тип протокола доступа, адрес сервера ресурса и адрес ресурса на сервере (рисунок 4.2).
В сокращенной форме опускают протокол и адрес сервера, указывая только местоположение ресурса от корня сервера. Полную форму используют, если возможна пересылка запроса другому серверу. Если же работа происходит только с одним сервером, то чаще применяют сокращенную форму.
Методы
Как уже говорилось, любой запрос клиента к серверу должен начинаться с указания метода. Метод сообщает о цели запроса клиента. Протокол HTTP поддерживает достаточно много методов, но реально используются только три: POST, GET и HEAD.
Метод GET

позволяет получить любые данные, идентифицированные с помощью URL в запросе ресурса. Если URL указывает на программу, то возвращается результат работы программы, а не ее текст (если, конечно, текст не есть результат ее работы). Дополнительная информация, необходимая для обработки запроса, встраивается в сам запрос (в строку статуса). При использовании метода GET в поле тела ресурса возвращается собственно затребованная информация (текст HTML-документа, например).

Существует разновидность метода GET – условный GET. Этот метод сообщает серверу о том, что на запрос нужно ответить, только если выполнено условие, содержащееся в поле if-Modified-Since заголовка запроса. Если говорить более точно, то тело ресурса передается в ответ на запрос, если этот ресурс изменялся после даты, указанной в if-Modified-Since.

Метод HEAD

аналогичен методу GET, только не возвращает тело ресурса и не имеет условного аналога. Метод HEAD используют для получения информации о ресурсе. Это может пригодиться, например, при решении задачи тестирования гипертекстовых ссылок.

Метод POST

разработан для передачи на сервер такой информации, как аннотации ресурсов, новостные и почтовые сообщения, данные для добавления в базу данных, т.е. для передачи информации большого объема и достаточно важной. В отличие от методов GET и HEAD, в POST передается тело ресурса, которое и является информацией, получаемой из полей форм или других источников ввода.

Использование HTML-форм для передачи данных на сервер

Для метода GET

При отправке данных формы с помощью метода GET содержимое формы добавляется к URL после знака вопроса в виде пар имя=значения, объединенных с помощью амперсанта &:
action?name1=value1&name2=value2&name3=value3
Здесь action – это URL-адрес программы, которая должна обрабатывать форму (это либо программа, заданная в атрибуте action тега form, либо сама текущая программа, если этот атрибут опущен). Имена name1, name2, name3 соответствуют именам элементов формы, а value1, value2, value3 – значениям этих элементов. Все специальные символы, включая = и &, в именах или значениях этих параметров будут опущены. Поэтому не стоит использовать в названиях или значениях элементов формы эти символы и символы кириллицы в идентификаторах.
Если в поле для ввода ввести какой-нибудь служебный символ, то он будет передан в его шестнадцатеричном коде, например, символ $ заменится на %24. Так же передаются и русские буквы.
Для полей ввода текста и пароля (это элементы input с атрибутом type=text и type=password), значением будет то, что введет пользователь. Если пользователь ничего не вводит в такое поле, то в строке запроса будет присутствовать элемент name=, где name соответствует имени этого элемента формы.
Для кнопок типа checkbox и radio button значение value определяется атрибутом VALUE в том случае, когда кнопка отмечена. Не отмеченные кнопки при составлении строки запроса игнорируются целиком. Несколько кнопок типа checkbox могут иметь один атрибут NAME (и различные VALUE), если это необходимо. Кнопки типа radio button предназначены для одного из всех предложенных вариантов и поэтому должны иметь одинаковый атрибут NAME и различные атрибуты VALUE.
В принципе создавать HTML-форму для передачи данных методом GET не обязательно. Можно просто добавить в строку URL нужные переменные и их значения.
http://www.ru/test.php?id=10&user=pit
В связи с этим у передачи данных методом GET есть один существенный недостаток – любой может подделать значения параметров. Поэтому не советуем использовать этот метод для доступа к защищенным паролем страницам, для передачи информации, влияющей на безопасность работы программы или сервера. Кроме того, не стоит применять метод GET для передачи информации, которую не разрешено изменять пользователю.
Несмотря на все эти недостатки, использовать метод GET достаточно удобно при отладке скриптов (тогда можно видеть значения и имена передаваемых переменных) и для передачи параметров, не влияющих на безопасность.

Для метода POST

Содержимое формы кодируется точно так же, как для метода GET, но вместо добавления строки к URL содержимое запроса посылается блоком данных как часть операции POST. Если присутствует атрибут ACTION, то значение URL, которое там находится, определяет, куда посылать этот блок данных. Этот метод, как уже отмечалось, рекомендуется для передачи больших по объему блоков данных.
Информация, введенная пользователем и отправленная серверу с помощью метода POST, подается на стандартный ввод программе, указанной в атрибуте action, или текущему скрипту, если этот атрибут опущен. Длина посылаемого файла передается в переменной окружения CONTENT_LENGTH, а тип данных – в переменной CONTENT_TYPE.
Передать данные методом POST можно только с помощью HTML-формы, поскольку данные передаются в теле запроса, а не в заголовке, как в GET. Соответственно и изменить значение

параметров можно, только изменив значение, введенное в форму. При использовании POST пользователь не видит передаваемые серверу данные.

Основное преимущество POST запросов – это их большая безопасность и функциональность по сравнению с GET-запросами. Поэтому метод POST чаще используют для передачи важной информации, а также информации большого объема. Тем не менее не стоит целиком полагаться на безопасность этого механизма, поскольку данные POST запроса также можно подделать, например создав html-файл на своей машине и заполнив его нужными данными. Кроме того, не все клиенты могут применять метод POST, что ограничивает варианты его использования.

При отправке данных на сервер любым методом передаются не только сами данные, введенные пользователем, но и ряд переменных, называемых переменными окружения, характеризующих клиента, историю его работы, пути к файлам и т.п. Вот некоторые из переменных окружения:

- REMOTE_ADDR – IP-адрес хоста (компьютера), отправляющего запрос;
- REMOTE_HOST – имя хоста, с которого отправлен запрос;
- HTTP_REFERER – адрес страницы, ссылающейся на текущий скрипт;
- REQUEST_METHOD – метод, который был использован при отправке запроса;
- QUERY_STRING – информация, находящаяся в URL после знака вопроса;
- SCRIPT_NAME – виртуальный путь к программе, которая должна выполняться;
- HTTP_USER_AGENT – информация о браузере, который использует клиент.

Обработка запросов с помощью PHP

Внутри PHP-скрипта существует несколько способов получения доступа к данным, переданным клиентом по протоколу HTTP. До версии PHP 4.1.0 доступ к таким данным осуществлялся по именам переданных переменных. Таким образом, если, например, было передано first_name=Nina, то внутри скрипта появлялась переменная $first_name со значением Nina. Если требовалось различать, каким методом были переданы данные, то использовались ассоциативные массивы \$HTTP_POST_VARS и \$HTTP_GET_VARS, ключами которых являлись имена переданных переменных, а значениями – соответственно значения этих переменных. Таким образом, если пара first_name=Nina передана методом GET, то
$HTTP_GET_VARS["first_name"]="Nina"".

Использовать в программе имена переданных переменных напрямую небезопасно. Поэтому было решено начиная с PHP 4.1.0 задействовать для обращения к переменным, переданным с помощью HTTP-запросов, специальный массив – $_REQUEST. Этот массив содержит данные, переданные методами POST и GET, а также с помощью HTTP cookies. Это суперглобальный ассоциативный массив, т.е. его значения можно получить в любом месте программы, используя в качестве ключа имя соответствующей переменной (элемента формы).

Пример

Figure: Пример внешнего вида формы.

Создана форма для регистрации участников заочной школы программирования. Тогда в файле action.php, обрабатывающем эту форму, можно написать следующее:

```
<?php $str = "Здравствуйте,
  ".$_REQUEST["first_name"]. "
```

```
   ".$_REQUEST["last_name"]."! <br>";
$str .="Вы выбрали для изучения курс по
   ".$_REQUEST["kurs"];
echo $str; ?>
```

Тогда, если в форму мы ввели имя «Вася», фамилию «Петров» и выбрали среди всех курсов курс по PHP, на экране браузера получим такое сообщение:

Здравствуйте, Вася Петров!
Вы выбрали для изучения курс по PHP

После введения массива $_REQUEST массивы $HTTP_POST_VARS и $HTTP_GET_VARS для однородности были переименованы в $_POST и $_GET соответственно, но сами они из обихода не исчезли из соображений совместимости с предыдущими версиями PHP. В отличие от своих предшественников, массивы $_POST и $_GET стали суперглобальными, т.е. доступными напрямую и внутри функций и методов.

Приведем пример использования этих массивов. Допустим, нам нужно обработать форму, содержащую элементы ввода с именами first_name, last_name, kurs (например, форму, приведенную выше). Данные были переданы методом POST, и данные, переданные другими методами, мы обрабатывать не хотим. Это можно сделать следующим образом:

```
<?php $str = "Здравствуйте,
   ".$_POST ["first_name"]."
   ".$_POST ["last_name"] ."! <br>";
$str .= "Вы выбрали для изучения курс по ".
      $_POST["kurs"];
echo $str; ?>
```

Тогда на экране браузера, если мы ввели имя «Вася», фамилию «Петров» и выбрали среди всех курсов курс по PHP, увидим сообщение, как в предыдущем примере:

Здравствуйте, Вася Петров!
Вы выбрали для изучения курс по PHP

Для того чтобы сохранить возможность обработки скриптов более ранних версий, чем PHP 4.1.0, была введена директива register_globals, разрешающая или запрещающая доступ к переменным непосредственно по их именам. Если в файле настроек PHP параметр register_globals=On, то к переменным, переданным серверу методами GET и POST, можно обращаться просто по их именам (т.е. можно писать $first_name). Если же register_globals=Off, то нужно писать

```
$_REQUEST["first_name"]
```

или

```
$_POST["first_name"],
$_GET["first_name"],
$HTTP_POST_VARS["first_name"],
$HTTP_GET_VARS["first_name"].
```

С точки зрения безопасности эту директиву лучше отключать (т.е. register_globals=Off). При включенной директиве register_globals перечисленные выше массивы также будут содержать данные, переданные клиентом.

Иногда возникает необходимость узнать значение какой-либо переменной окружения, например метод, использовавшийся при передаче запроса или IP-адрес компьютера, отправившего запрос. Получить такую информацию можно с помощью функции getenv(). Она возвращает значение переменной окружения, имя которой передано ей в качестве параметра.

```
<? getenv("REQUEST_METHOD");
   // возвратит использованный метод
echo getenv ("REMOTE_ADDR");
   // выведет IP-адрес пользователя,
   // пославшего запрос
?>
```

Как мы уже говорили, если используется метод GET, то данные передаются добавлением строки запроса в виде пар «имя_переменной=значение к URL-адресу ресурса». Все, что записано в URL после знака вопроса, можно получить с помощью команды
getenv("QUERY_STRING");
Благодаря этому можно по методу GET передавать данные в каком-нибудь другом виде. Например, указывать только значения нескольких параметров через знак плюс, а в скрипте разбирать строку запроса на части или можно передавать значение всего одного параметра. В этом случае в массиве $_GET появится пустой элемент с ключом, равным этому значению (всей строке запроса), причем символ «+», встретившийся в строке запроса, будет заменен на подчеркивание «_».
Методом POST данные передаются только с помощью форм, и пользователь (клиент) не видит, какие именно данные отправляются серверу. Чтобы их увидеть, хакер должен подменить нашу форму своей. Тогда сервер отправит результаты обработки неправильной формы не туда, куда нужно. Чтобы этого избежать, можно проверять адрес страницы, с которой были посланы данные. Это можно сделать опять же с помощью функции getenv():
getenv("HTTP_REFERER");
Пример обработки запроса с помощью PHP
Нужно написать обработчки формы (см. выше) для регистрации участников заочной школы программирования и после регистрации отправить участнику сообщение.

```
<h2>Форма для регистрации студентов</h2>
<form action="1.php" method=POST>
Имя <br><input type=text name="first_name"
    value="Введите Ваше имя"><br>
Фамилия <br><input type=text name="last_name"><br>
E-mail <br><input type=text name="email"><br>
<p> Выберите курс, который вы бы хотели посещать:<br>
<input type=checkbox name='kurs[]' value='PHP'>PHP<br>
<input type=checkbox name='kurs[]' value='Lisp'>Lisp<br>
<input type=checkbox name='kurs[]' value='Perl'>Perl<br>
<input type=checkbox name='kurs[]' value='Unix'>Unix<br>
<P>Что вы хотите, чтобы мы знали о вас? <BR>
<textarea name="comment" cols=32 rows=5></textarea>
<input type=submit value="Отправить">
<input type=reset value="Отменить">
</form>
```

Figure: Пример кода формы.

Следует отметить, способ передачи значений элемента checkbox. Когда мы пишем в имени элемента kurs[], это значит, что первый отмеченный элемент checkbox будет записан в первый элемент массива kurs, второй отмеченный checkbox – во второй элемент массива и т.д. Можно, конечно, просто дать разные имена элементам checkbox, но это усложнит обработку данных, если курсов будет много.
Скрипт, который все это будет разбирать и обрабатывать, называется 1.php (форма ссылается именно на этот файл, что записано в ее атрибуте action). По умолчанию используется для передачи метод GET, но мы указали POST. По полученным сведениям от зарегистрировавшегося человека, скрипт генерирует соответствующее сообщение. Если человек выбрал какие-то курсы, то ему выводится сообщение о времени их проведения и о лекторах, которые их читают. Если человек ничего не выбрал, то выводится сообщение о

следующем собрании заочной школы программистов (ЗШП).

```
<?
// создадим массивы соответствий курс-время его
// проведения и курс-его лектор
$times = array("PHP"=>"14.30","Lisp"=>"12.00",
    "Perl"=>"15.00","Unix"=>"14.00");
$lectors = array("PHP"=>"Василий Васильевич",
"Lisp"=>"Иван Иванович", "Perl"=>"Петр Петрович", "Unix"=>"Семен Семенович");
define("SIGN","С уважением, администрация");
    // определяем подпись письма как константу
define("MEETING_TIME","18.00");
    // задаем время собрания студентов
$date = "12 мая"; // задаем дату проведения лекций
    //начинаем составлять текст сообщения
$str = "Здравствуйте, уважаемый " . $_POST["first_name"]
    . " " . $_POST["last_name"]."!<br>";
$str .= "<br>Сообщаем Вам, что ";
$kurses = $_POST["kurs"];    // сохраним в этой переменной
                             // список выбранных курсов
if (!isset($kurses)) { // если не выбран ни один курс
    $event = "следующее собрание студентов";
    $str .= "$event состоится $date ". MEETING_TIME . "<br>";
} else { // если хотя бы один курс выбран
    $event = "выбранные Вами лекции состоятся $date <ul>";
        //функция count вычисляет число элементов в массиве
    for ($i=0;$i<count($kurses);$i++){
                    // для каждого выбранного курса
        $k = $kurses[$i]; // запоминаем название курса
        $lect = $lect . "<li>лекция по $k в $times[$k]";
                    // составляем сообщение
        $lect .= " (Ваш лектор, $lectors[$k])";
    }
    $event = $event . $lect . "</ul>";
    $str .= "$event";
}
$str .= "<br>". SIGN; // добавляем подпись
echo $str; // выводим сообщение на экран
?>
```

Задание:
1. Создать 3 таблицы студенты, группы и список студентов групп.
2. Средствами языка PHP создать html-страницу, содержащую ссылки для просмотра и редактирования каждой таблицы.
3. Создать html-страницу, содержащую список выбора для групп (select, информация заполняется динамически из таблицы базы данных) и отображающую в зависимости от выбранного пункта информацию о студентах группы в виде таблицы.