This lecture continues coverage of HTML forms and introduces some more PHP constructs.

Menus

One of the most useful controls is the *menu* whereby users are presented with a scroll-down list of options.

A menu is implemented with the HTML `<select>..</select>` element, containing a number of `<option>..</option>` elements listing the available options.

```html
<select name="sandwich_type">
  <option>Sandwich</option>
  <option>Long Roll</option>
  <option selected>Round Roll</option>
</select>
```

The user is presented with a drop-down menu where the initially selected option is shown.

One option should always be explicitly selected because the menu's initial appearance is undefined otherwise.
At the script

The PHP script will receive the choice in the variable

$_POST[‘sandwich_type’]

or

$_GET[‘sandwich_type’]

where the value of that variable will be a string equal to the text that they selected — in this case "Round Roll".

Menus can be designed to accept multiple selections, and in this case, the name of the control should be an array type, just as with checkboxes.

Text Areas

The final control we will look at is the

<textarea> .. </textarea>

element that allows the user to enter a number of lines of text. Textareas are widely used for user reviews, comments, blog postings, company feedback and so on.

<textarea name="comment" rows="10" cols="60">
Please enter your comments here...
</textarea>

The attributes rows and cols indicate the initial size of the displayed text area, which will initially contain the specified text (in this case “Please enter your comments here...”).

What the user sees

The user can overtype the initially displayed text with their own comment and/or feedback.

The PHP script receives the entire content of the area as a single string in

$_POST[‘comment’].

PHP functions

Certain tasks (for example, connecting to a DB, constructing menus) occur frequently in PHP development. Rather than repeating the code for this everywhere it is needed, it is far better to separate it out into a single reusable function.

PHP has a fairly intuitive function syntax that should cause no problems to programmers familiar with Java or C.

```php
function factorial($n) {
    if ($n == 0) {
        return 1;
    } else {
        return $n * factorial($n-1);
    }
}
```
Two function examples

```php
<?php
require_once "MDB2.php";

function getDSN() {
    $driver = "mysqli";
    $user = "db3240";
    $database = "db3240";
    $host = "localhost";
    $pass = "XXXXXXX";
    return "$driver://$user:$pass@$host/$database";
}

function getConnection() {
    $conn =& MDB2::connect(getDSN());
    if (MDB2::isError($conn)) {
        die($conn->getUserInfo());
    }
    return $conn;
}
?>
```

Using these functions

The first of these two functions creates a *data source name* from its various components, and then the second function uses the first to get a valid (error-checked) connection to the database.

If these two functions are stored in a file, say `db3240.php` then *all* of the scripts that need database connectivity can simply require_once "db3240.php"; and then call the `getConnection()` function to get a connection.

Because the error checking code is contained in the function itself no additional effort is needed to ensure that the connection is valid.

Advantages of this modularization

There are numerous advantages of modularizing the code like this:

- If the database is moved or changed, or the username or password changed, then just *one file* needs to be altered.
- There is no temptation to skip the error-checking code on the connection object, because it happens automatically.
- This file can be moved out of the Apache document tree, thus keeping the DB password one-step-removed from the script in case for some reason, the server delivers the script to the user without the PHP being interpreted.

Convenience functions

Another important reason to use PHP functions is to automate some of the more tedious HTML generation tasks, or to incorporate repetitive elements (such as common headers and/or footers) into your HTML pages.

As an example, we'll write a function that will produce a valid HTML table from *any* result set obtained from a database connection. This can be used for debugging purposes when you are unsure what the result set contains, or for a production website when you are sure that the query is precisely correct.
Tabulate a result set

Our function will be called

```php
function tabulateResultSet($result) {
    // produces an HTML table
}
```

and it will simply print out a table containing the contents of the result set.

(More pedantic programmers may observe that this is not a function at all because it does not return a value, but a procedure. We could rewrite it as a function that constructs one huge string and returns it.)

First output the table element

```php
function tabulateResultSet($result) {
    $nr = $result->numRows();
    $nc = $result->numCols();
    print '<table border="1">';
    // Print each row here
    print '</table>);
}
```

This uses the result object methods `numRows()` and `numCols()` to find out how many rows and columns it contains (although we won’t need the number of rows).

Notice the use of single quotes so that the resulting HTML has its quotes intact.

Next fetch each row in turn

```php
print '<table border="1">';
while ($row =& $result->fetchRow()) {
    // Print out a single row
}
print '</table>";
```

This uses the standard while-loop mechanism for fetching each row in turn from the result set.

Mark up the row’s elements

```php
while ($row =& $result->fetchRow()) {
    print "<tr>
    for ($i=0; $i<$nc; $i++) {
        if ($row[$i] == NULL) {
            print "<td>NULL</td>";
        } else {
            print "<td>$row[$i]</td>";
        }
    }
    print "</tr>
```

We know that each row has $nc entries and so a for-loop can be used to wrap each one into a <td>..</td> pair of tags.
What about NULLs

If the database table contained a NULL value, then the corresponding variable in the PHP array $row will not be set.

This can be checked in a number of ways:
- if ($row[$i] == NULL)
- if is_null($row[$i])
- if (!isset($row[$i]))

and a suitable value printed into the table.

Maintaining state

Recall that HTTP is a stateless protocol and thus every client/server transaction is a separate entity.

This means that the webserver has no automatic mechanism to “remember” information about an individual browser as they navigate through the site.

However many web-based applications need to maintain state information — for example to allow a user to browse throughout a site adding various items to a “shopping cart” before proceeding to the “checkout”.

Sessions

Fortunately PHP provides a simple and transparent mechanism to allow scripts to maintain state information — this mechanism is called support for sessions.

A session can be viewed as a sequence of HTTP requests from a particular browser, and the challenge is to recognise that the second and subsequent requests originate from the same place.

For example, a user may enter a web-site at a log-in page, and then expect that subsequent browsing within that site recognizes that they are logged in.

Session Management

PHP’s session management is handled through the use of the superglobal array $_SESSION.

One PHP script can create a variable in this array and it will automatically be available to all other scripts that are requested in the same session.

Thus, for example a log-in script can validate a user’s details and then set a variable indicating who the user is, and all other scripts can check whether the variable is set or not.
### Initiating a session

Any script wishing to use sessions must first call `session_start`. After this the script can either write to or read from the array `$_SESSION`.

```php
<?php
    session_start();
    // validate user’s log in details

    $_SESSION['user_id'] = "gordon@csse.uwa.edu.au";
?>
```

This might be a portion of a login page that first validates details and then sets a session variable.

### How do sessions work?

Sessions primarily work through the use of cookies. When the session is first created, PHP generates a session id which is sent to the browser as a cookie — this is a small amount of information that the browser stores.

Any variables created in the `$_SESSION` array are stored on the server in an area identified by the session id.

On every subsequent HTTP request, the browser automatically sends the cookie back to the webserver, which can then locate the stored values of the `$_SESSION` array and make them available to the new script.

### What if cookies are turned off?

Some web-browsers have a setting where they will not accept cookies from a web server. If this is the case, then PHP automatically passes the session id along with the URL.

For example, if the session id is `1234510813018230123` then PHP will automatically append this to every link on the page so that a link like `cart.php` would become `cart.php?PHPSESID=1234510813018230123`

Then when the user clicks through, the session id is passed back to the server along with the URL.