PHP Hypertext Preprocessor

Written by Levente Hunyadi[1]
Translated by Levente Erős

1. Introduction

Web pages generated dynamically on the server side are widely used nowadays. It's also quite typical that a database management system (DBMS) can be found behind this dynamic content. PHP (abbreviating Personal HomePage) is a freely accessible and universal command language, specialized mostly for web-related development projects. PHP is accessible for many different platforms (including Linux, Windows, Mac OS X and several other Unix-variants), it's easy to

[1] This handout is based on the previous work of Bálint Laczay, Gábor Salamon, and Gábor Surányi. I'd like to thank Levente Erős, István Madari and Gábor Zavarkó for their comments on this work.

learn and it's organically embedded into markup language sources of web pages. With all these properties PHP is ideal for web-related development. The Apache web server (serving nearly 60% of active websites) has a built-in PHP module in more than 35% of all the cases.[2] PHP can cooperate with 8 further well-known web server applications and 22 DBMSs, which is another reason for its popularity beyond its simplicity. PHP includes several tools for text processing, ranging from regular expressions to XML tools. Even more complex tasks, like generating images, PDF documents or compressing files can be carried out by PHP's different function libraries.

The steps of running a PHP script are as follows:

1. The user's browser sends an HTTP-request to the web server addressing a page with the extension `php`.
2. This request – given the settings are appropriate – results in the web server giving the control to the PHP interpreter.
3. The interpreter processes the page regarding it as a PHP script and returns the output of the script to the web server.
4. The web server forwards this output to the user's web browser.

Database access is carried out by the previously mentioned function libraries. Most database managers have a so-called native function library. Unfortunately, these libraries differ DBMS by DBMS; each of them has to be used on a different way. There is however a library, the standard ODBC (Open Database Connectivity) library that makes it possible to access the database in a uniform way. Its major drawback is its low efficiency compared to native function libraries. Furthermore, ODBC is unable to exploit the special features of different DBMSs.

Our PHP application is going to connect to an Oracle database server through the native Oracle8 interface.[3] The PHP interpreter provides this interface through the OCI8 (Oracle Call Interface) library, thus OCI8 has to be installed on the web server as well.

To learn this new aspect of database usage it's necessary to learn a whole new programming language. Thus, we illustrate the usage of the different methods used during the workshop on a sample PHP application. This application is accessible on the website of the course.

2. PHP basics

In the following few sections we give a basic insight into PHP. Our goal is to deliver a minimal knowledge on the subject that is enough to succeed in the entry test on the one hand and which, on the other hand makes it easier to understand the entire PHP documentation. For a complete review on the features of PHP and all its function libraries, see the website *http://www.php.net/manual/en/*. Nonetheless, after understanding the sample script by the end of the section you should be able to fulfill the requirements of the workshop.

We assume that you already have a basic knowledge of the (X)HTML and C languages, since their presentation is far beyond the scope of this handout. To get to know the basics of HTML,

[2] According to reports *Web server survey* (netcraft.com) and *Apache module report* (securityspace.com) published in December 2007.

[3] There is also a function library called Oracle, which is a predecessor of Oracle8 with way less capabilities than Oracle8.

XHTML, and web page forms see the documentation on *http://www.w3.org/TR/html4*, and *http://www.w3.org/TR/xhtml1*.

XHTML and HTML are almost identical from the perspicuity and automatic processing points of view, but XHTML is more favorable, since it is well formed (in XML means), i.e. the restrictions of XML apply to XHTML codes. The main differences between HTML and XHTML are as follows:

- While HTML allows overlapping tags, XHTML does not. For example, in XHTML, `<b>bold <i>bold italic</i></b><i> italic</i>` has to be used instead of `<b>bold <i>bold italic</b> italic</i>`.
- Self-closing tags like `<br>` are illegal in XHTML, for example, `<br/>` has to be used instead of `<br>`.[4]
- XHTML tags always have to be closed (for example, after `<p>` there has to be a `</p>` somewhere in the code).
- XHTML tags are written by lowercase letters (`<html>` instead of `<HTML>`).

2.1 PHP scripts

PHP pages are text documents and have to be placed on the web server the same way as regular HTML documents. Due to their php extensions the server interprets these files as PHP pages. A PHP page consists of an alternating series of script parts (to be executed) and HTML parts.[5]

HTML parts don't have to be indicated explicitly, since HTML is default. These parts may contain arbitrary markup language elements which will be included word-by-word in the output webpage returned to the web server application. PHP commands can be written either by capitals or by lowercase letters. These commands have to be placed between `<?php` and `?>` marks. Commands are separated by `;` (semicolons) just like in PERL or C. Comments can be typed between the marks `/*` and `*/` or after mark `//` or `#` until the end of the line. Blocks of commands are introduced by `{` and closed by `}`.

In the following, we introduce a simple PHP page for listing the multiples of seven:

[4] In XHTML, we can also use the long forms of self-closing tags, like `<br></br>`, but these forms confuse browsers that can only interpret HTML and not XHTML. Thus in practice, short forms are used just like in this handout.

[5] Notepad++ (http://notepad-plus.sourceforge.net/) is a universal and easy-to-use tool for editing PHP pages (and other kinds of text documents). For completing the assignments given you in class we highly recommend the usage of a software that supports PHP and XHTML editing.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Multiples of Seven</title>
</head>
<body>
  <table>
    <tr><th>x</th><th>7*x</th></tr>
    <?php
      for ($i = 1; $i <= 10; $i++) {
        print "<tr><td>";
        print $i;
        print "</td><td>";
        print 7*$i;
        print "</td></tr>\n";
      }
    ?>
  </table>
</body>
</html>
```

2.2 Types

The most commonly used types in PHP are boolean, integer, float, string, and array.[6] Logical values can be defined by constants `TRUE` and `FALSE` (or `true` and `false`), numbers are defined by their decimal values on the easiest way, while strings are defined between apostrophes (`'`) or quote marks (`"`). When using apostrophes the interpreter doesn't changes the value of the string,[7] while in the case of quote marks the references to variables (marked by `$`) will be substituted and a few escape sequences will also be interpreted. Thus

- The value of `$a = '\n'` will be the sequence of a backslash and letter n instead of a line break,
- The value of `$b = "{$a}\n{$a}"` is going to be two sequences of a backslash and letter n (substitution of the value of the variable) divided by a line break (interpreting the escape sequence).

The type of a value or a variable is (almost) never given explicitly, since types are decided by the interpreter based on the environment in which the value or variable is used. Type conversion is carried out automatically when necessary. In the following, we give a list of the most common type conversion cases:

- Number to string: the string is going to be the decimal form of the number, with exponential notation in the case of a large number. For example, `134.7` is converted to `"134.7"`, while `10000000*10000000` is converted to `"1E+014"`.

[6] PHP has three further types, namely object (the notion of class known from object-oriented languages is also known in PHP, the object type is the overall name of the different class instances), resource (which is for storing references to exterior resources (e.g. a database connection)), and the NULL type (one instance of which is the special NULL value known from SQL).

[7] If we'd like to insert an apostrophe or a quote mark in our string, sequence `\'` or `\"` has to be used. Similarly, to include a `\` sequence `\\` has to be used. Other characters, however, don't have any special meaning.

- String to number: just like when using `strtod()` in C, the value of the number is going to be the number found at the beginning of the string, for example `"3 little pigs"` is converted to `3`, while `"route 66"` is converted to `0`, since there is no number at the beginning of the string.
- Boolean to string: `FALSE` is converted to an empty string, while `TRUE` is converted to the string `"1"`.
- String to boolean: `""` and `"0"` are converted to `FALSE`, everything else is converted to `TRUE`. This means that `"true"` and `"false"` are both converted to `TRUE`.
- Number to boolean: `0` and `0.0` are converted to `FALSE`, all the other numbers are converted to `TRUE`.
- Boolean to number: `FALSE` is converted to `0`, `TRUE` is converted to `1`.
- Array to boolean: an empty array is converted to `FALSE`, while an array containing at least one element is converted to `TRUE`.

Of course we can carry out type conversions explicitly as well, by the cast operator known from C: the value of `$i = (int) 1.7` is thus `1` while the value of `$s = (string) 2.5` is `"2.5"`

2.3 Operators

The most important operators in PHP are:

- Arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division of float numbers), `%` (division remainder), `-` with a single operand (negation): the operands of these operators are numbers. Furthermore, operator `++` is used for pre-incrementing and post-incrementing, and `--` is used for pre-decrementing and post-decrementing values of variables, respectively. Division of integers can be implemented by a division remainder calculation and a subtraction or by a truncation.
- Assignment operator: `=`. On the left side there has to be a reference to a variable. Its forms contracted with other operators (`+=`, `-=`, etc, just like in C) can also be used.
- Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. If the types of the operands are different an automatic type conversion is carried out. If one of the values is a logical value the interpreter converts both of the operands to logical values. When comparing a number to a string the string is converted to a number. Two strings containing numbers are converted on the same way. The comparison of two strings can be done lexically or by a numeric comparison depending on the values stored in the strings.
- Comparison without type conversion: `===`. It works the same way as operator `==` but without making type conversions: if the types of the operands are different it returns `FALSE`. Its negation is operator `!==`.
- Concatenation of strings: `.` (dot). It returns the concatenation of its two string operands. The result of `$a . $b` is the same as the result of `"{$a}{$b}"`.

Some examples of using these operators:

- The value of `4/8` is `0.5` (number).
- The value of `4/"6hours"` is `0.666667` (number), and the value of `5 + "3 little pigs"` is `8` (number), since these operators convert their operands to numbers.
- The value of `"FALSE" != FALSE` is `TRUE` (boolean), since one of the operands of `!=` is a boolean value, so the interpreter converts both operands to booleans.

- The value of `(10 * "1 dalmatian") . "1 dalmatian" . "s"` is `"101 dalmatians"`.

2.4 Variables

The (case sensitive) name of a variable in PHP consists of letters, numbers and underscores (_). Variable names must not begin with a number. Referencing a variable is in the form `$variableName`. Variables can be used without a declaration, since their types are decided when being assigned a value. For example, after `$b = 4` variable `$b` is going to be a number and then after `$b = $b . "2 is the answer"` variable `$b` is going to be a string with a value of `"42 is the answer"`. At this point the value of the expression `$b + 3` is `45`, since the value of `$b` becomes a number again. It's important to note that the conversion is only used for evaluating the operands and it does *not* change the type nor the value of the variable. So the value of `$b` remains `"42 is the answer"`.

2.5 Arrays

PHP arrays are associative, which means their indices can be arbitrary values (not necessarily non negative integers). The easiest way to create an array is to use the keyword `array`, for example: `$fruit = array("pineapple", "avocado", "orange")`. As a result of this expression array `$fruit` will have three elements with element `"pineapple"` belonging to the index value `0`, element `"avocado"` belonging to the index value `1`, and element `"orange"` belonging to the index value `3`. If we don't want to use numbers as index values, we can, for example, do the following:

```
$vegetable = array(
  "c"=>"carrot", "t"=>"turnip", "b"=>"beet"
);
```

We can also use multi-dimensional arrays, like:

```
$data = array(
  "fruits" => array("pineapple", "orange", "plum"),
  "colors" => array("yellow", "orange", "blue")
);
```

To get the value of a certain element of an array the corresponding index value has to be put into brackets. The value of the following expression is thus `"orange"`:

```
$data["fruits"][1]
```

This way we can also insert new elements into the array, for example:

```
$vegetable["p"]="pea"
```

or simply:

```
$fruit[] = "red currant"
```

In this case PHP generates the new index automatically based on the highest numerical index in the array.

2.6 Writing the output

It's pretty common that we would like to insert simple text into the page returned to the browser. In this case using marks `<?php` and `?>` continuously is likely to result in a hard readable source . Let's see an example for this. (As you might assume, function `getdate` returns an associative array, with its index `"hours"` containing the current time in hours.)

```
<?php $date = getdate(); ?>
<p>Welcome, good
<?php if ($date["hours"]<12) { ?>morning<?php } else { ?>afternoon<?php } ?>
!</p>
```

Our other option for writing the output is to use the PHP commands `print` or `echo`. The expression after the keyword is converted to a string. Our previous example by using `echo`:

```
<?php $date = getdate(); ?>
<p>Welcome, good
  <?php
    if ($date["hours"] < 12) {
      echo("morning");
    } else {
      echo("afternoon");
    }
  ?>!
</p>
```

2.7 Control structures

Control structures of PHP might remind one to those of other languages. Structures `if ... else ...`, `while`, `do ... while`, `for`, `break`, `continue`, `switch` are used exactly the same way as in C. Structure `elseif` works the same way as the sequence `else if`.

Structure `foreach` is used for walking through the values of all the elements stored in the first dimension of an array. The parameters of this structure are an array and one or two variables. As an example, let's look at the following code:

```
foreach ($fruit as $name) echo("I like {$name}<br />\n");
```

and its output:

```
I like pineapple<br />
I like avocado<br />
I like orange<br />
I like red currant<br />
```

The following code copies the contents of array `normal` to array `reversed` swapping the keys and values. (Of course, it only works exactly this way if the values of the original array are all unique.)

```
foreach ($normal as $key => $value) $reversed[$value] = $key;
```

2.8 Referencing external files

Keywords `require` and `include` are both used for including external files. `require` works the same way as the C preprocessor, that is the included file is extracted before its execution, while in the case of `include` the file is only extracted when the execution reaches the command. As a consequence, by `include` even variable names can be used. What might be surprising (and that's why we call your attention to it) is that PHP commands in the included files have to be put between the marks `<?php` and `?>` regardless of whether being called from inside or outside the PHP context, otherwise the interpreter will not process these commands.

2.9 Function definitions

The definition of a function begins with the keyword `function`. After that come the name of the function and a list of its formal parameters. As we mentioned earlier there's no need to declare the types of any parameters and the type of the return value. Parameter names have to be preceded by `$` (dollar sign) like in the case of variable names. In PHP parameters can be passed by value and by reference as well. To pass a parameter by reference an `&` (ampersand) has to be put right before the dollar sign of the variable name. Keyword `return` is used to return from the function, optionally followed by the expression generating the return value. More than one value can be returned by arrays or by output parameters, for example:

```
function operations($a, $b, &$c) {
  $c = $a + $b;
  $d = $a - $b;
  return array($c, $d);
}
```

The main program of the PHP script consists of the commands outside of function bodies. Variables defined outside of function bodies are invisible inside functions (by default). If a function still needs such a variable, the variable can be passed to the function as a parameter or imported to the body of the function using the keyword `global`.

2.10 Predefined variables

PHP provides plenty of predefined variables for the programmer. Many of them contain data about the system and the script, but those we are interested in the most are the ones supporting communication between web pages. These variables have three types:

1. values of parameters passed explicitly in the URL of the request
2. values related to fields of the form on the page that induces the request
3. values of cookies[8]

[8] HTTP is a stateless protocol. This means that after responding a request the server doesn't store any information it could use later (e.g. for responding the next request). In database management and other applications, however, we might need to store some information (e.g. the contents of the shopping cart) among the many requests. Basically, there are two solutions for solving this problem: using sessions and using client-side applications. In the case of sessions the information is stored on the server and is reached by a key linked to every single request, while in the case of client-side applications all the information is stored on the client and only the necessary pieces of information

In the following, we only describe the first two types, as we are only going to use these two types of variables.

When the browser sends an HTTP request to the server it has the opportunity to pass variables. These parameters are accessible for the application running on the server (in our case the PHP script). This way the server is able to generate its response dynamically, depending on parameters of the request. The parameters of an HTTP request consist of a name and a value belonging to it.

When submitting a form on an HTML page its fields appear as the parameters of the request. The name of the parameter is stored in the `name` attribute of the field, while its value is stored in the `value` attribute of the field, which in some cases (e.g. an input field) is set by the user. Checkboxes only appear among the parameters of the request if checked when submitting the form. Similarily, from all the elements of a select list (form element `select`) the only element the value of which appears among the parameters of the HTTP request is the one selected. Form element `hidden` does not appear in the browser, but its value is sent to the server as a parameter of the request just like other parameters. This way hidden variables are suitable for passing identifiers for example.

There are two ways of passing form data, GET and POST. When using GET mode the form data sent appears in the URL of the request (after the `?` in the URL). In POST mode the form data appears in the body (payload) of the request. Accordingly, all the data from the form is stored in the global arrays `$_GET` and `$_POST`, the keys of which are the values defined as the values of `name` attributes of form elements. Since `$_GET` stores the values of all the parameters in the URL parameters linked to the request can be retrieved from this array. For example, in the case of the URL `index.php?id=82` array `$_GET` will contain the value `82` (as a string) at key `id`.[9] Besides `$_GET` and `$_POST` there is a third array called `$_REQUEST` by which we can access our data without knowing whether it comes from a GET or a POST kind of source.

PHP makes two changes to names of form variables. First, if a name contains an array indexing operator (`[]`) the value will be stored in `$_GET` or `$_POST` as an element of an array corresponding to the name. This makes it possible, for example to process more than one selected elements of a list. Second, as `.` (dot) is not allowed in variable names, PHP changes its occurrences to _ characters (underscores).

To understand the above let's take a look at the following complex example. Let the HTML source of a registration web page (`register.html`) be the following:

---

are linked to the request. Java applets are typical cases of client-side applications, while sessions can be implemented even in PHP.

[9] Since the processing page references elements of `$_GET` and `$_POST` pretty often, PHP can provide the programmer with the values of the first dimension of the arrays as values of variables with the names of corresponding keys. This however, emerges security-related questions, thus this feature is switched off in newer PHP versions by default (and during the workshop as well).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Registration page</title>
</head>
<body>
<form action="register.php" method="post">
  <table>
  <tr>
    <td>Name:</td>
    <td><input type="text" name="name" /></td>
  </tr>
  <tr>
    <td>E-mail</td>
    <td><input type="text" name="email" value="yourname@example.com" /></td>
  </tr>
  <tr>
    <td>Beer:</td>
    <td>
      <select multiple="multiple" name="beer[]">
        <option value="Miller">Miller</option>
        <option value="Guinness">Guinness</option>
        <option value="Stuttgarter">Stuttgarter Schwabenbräu</option>
      </select>
    </td>
  </tr>
  <tr>
    <td><input type="submit" /></td>
  </tr>
  </table>
</form>
</body>
</html>
```

Let the processing page `register.php` be the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Successful registration</title>
</head>
<body>
<p><?php
  print "Dear {$name}, you\'ve registered with the e-mail address {$email}\n";
  $drinks = implode(', ', $beer);
  print "The beers you\'ve ordered are the following: {$drinks}."
?></p>
</body>
</html>
```

If the user enters `Sir Arthur Conan Doyle` as his name and `arthur@example.com` as his address on `register.html`, checks the beers `Miller` and `Guinness` and submits the form `register.php` will generate the following HTML code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Successful registration</title>
</head>
<body>
<p>Dear Sir Arthur Conan Doyle, you've registered with the e-mail address
arthur@example.com.
The beers you've ordered are the following: Miller, Guinness.</p>
</body>
</html>
```

As you can see, if the user selects more than one element of the form field `select` at the same time, in the HTTP request a name-value pair is going to be generated for every single element (e.g. `beer[]="Miller"` and `beer[]="Guinness"`). When the PHP script gets a request including these parameters, it creates the array `beer` and puts the strings `"Miller"` and `"Guinness"` in it as elements indexed by `0` and `1`. Command `implode` concatenates the elements of its input array.

3. The main Oracle8 functions

In this section, we roughly introduce the main Oracle8 functions. Although there are no type names in a PHP code, in our handout for each input parameter, we will note the type the parameter is handled as by its function. If an input parameter's type is different from the type expected by the function, it is converted to the desired input type automatically. In our list of functions we also note the return types of functions; however the same function might produce return values of different types. Most of the present functions return a special value in case of a successful database operation, and a boolean value (`FALSE`) otherwise. According to our syntax parameters in brackets ([]) are optional, they can be omitted. For a more detailed description of Oracle8 functions see the online documentation.

3.1 oci_connect, oci_pconnect

Syntax:
```
resource oci_connect(string $username, string $password
[, string $db])
resource oci_pconnect(string $username, string $password
[, string $db])
```
Function: connects the database
Return value: descriptor of the connection or `FALSE` in case of an error

These two functions only differ at one point: function `oci_connect` always opens a new connection between the web server and the database which has to be closed by function `oci_close`, while `oci_pconnect` only opens a connection if there is no open connection with the same parameters. This kind of connection can't be closed down by PHP, it will remain open even after the script in the page stops running, and therefore it's not recommended to use `oci_pconnect` in systems under high stress level.

3.2 oci_close

Syntax: `bool oci_close(resource $connection)`
Function: disconnects from database

3.3 oci_parse

Syntax: `Resource oci_parse(resource $conn, string query)`
Function: prepares an SQL statement for execution
Return value: descriptor of the executable statement, `FALSE` in case of an error

3.4 oci_bind_by_name

Syntax:
```
bool oci_bind_by_name(resource $statement, string $ph_name,
mixed &$variable[, int $length[, int $type]])
```
Function: binds a PHP variable to an Oracle value
Return value: `TRUE` or `FALSE` depending on the success of the operation

Parameter `variable` is the PHP variable; `ph_name` defines what to bind the variable to in the PHP statement. Parameter `length` is the maximal length if the given variable, if `-1`, the actual length will be the maximal length.

3.5 oci_execute

Syntax: `bool oci_execute(resource $statement[, int $mode]);`
Function: executes a prepared statement
Return value: `TRUE` or `FALSE` depending on the success of the operation

If `mode` is the default `OCI_COMMIT_ON_SUCCESS`, this function automatically commits after a successful execution of the statement. If we'd like to use Oracle's default transactional behavior `OCI_DEFAULT` has to be declared as the mode *explicitly*. In this case `oci_commit` and `oci_rollback` can control transactions.

3.6 oci_free_statement

Syntax: `bool oci_free_statement(resource $stmt)`
Function: frees resources related to a prepared statement
Return value: `TRUE` or `FALSE` depending on the success of the operation

3.7 oci_fetch_array

Syntax: `int oci_fetch_array(resource $stmt[, int $mode])`
Function: reads the next row of the result of the executed (`SELECT`) statement
Return value: `FALSE` if the operation is unsuccessful, otherwise an array corresponding to the actual row of the result

If mode is `OCI_ASSOC` the return value is an associative array with the column names as keys. If mode is `OCI_NUM` the return value is an array indexed by integers (starting with zero). The default

value is `OCI_BOTH`, that is, the data in the returned array can be reached by both the column names and the column indices. `OCI_RETURN_NULLS` can be combined with all of the above mentioned modes (by character +) resulting in the returned array including empty values, that is, even empty cells will be included in the returned array with a value of NULL. (Comparison to a NULL value can be carried out by operator `===` or by function `is_null`.)

3.8 oci_num_fields

Syntax: `int oci_num_fields(resource $stmt)`
Function: gets the number of columns of the executed (`SELECT`) statement
Return value: number of columns

3.9 oci_field_name

Syntax: `string oci_field_name(resource $stmt, int $col)`
Function: gets the name of the column corresponding to the given column number from the result of the executed (`SELECT`) statement (the first column number is 1)
Return value: name of column

3.10 oci_fetch_all

Syntax: `int oci_fetch_all(resource $stmt, array &$variable)`
Function: puts all the rows of the result of the executed (`SELECT`) statement into a two-dimensional array. The first index is always the name of the column (written by capitals) while the second is the row number (starting from zero).
Return value: `FALSE` in case of an unsuccessful operation, otherwise the number of rows in the result.

As a side effect of automatic type conversion `0` (zero) and `FALSE` can be confused with each other, as both of these values are valid return values of the function. Thus, it's worth using comparison operator `===`, since it requires type matching.

3.11 oci_error

Syntax: `array oci_error(resource $stmt|$conn)`
Function: gets the properties of the last error
Return value: an array of error properties

If we'd like to get the properties of an error made by `oci_parse` the parameter should be the descriptor of the database connection, while in the case of an error made by an `oci_execute` command the parameter is the descriptor of the prepared statement.

4. Sample application

In this section, we introduce some parts of the web pages of an imaginary transport company. The web pages are not public; they are used by the company for registering and tracing their orders. In our example, we will view a list of vehicles (*vehicles.php*) and some more detailed

information on the vehicles (*vehicle-details.php*). The example is not complete since our goal is to illustrate how to get the information we need from a database and then publish it on a web page. We will also show how to do this using HTML form parameters. See comments of the source code for explanations on the most important constructs.

As you might notice data access and presentation are separated in the application. While *services.php* (section 4.3) contains general services for database access, *queries.php* (section 4.4) contains the queries to be used. Both of these two files are placed outside of the publicly accessible *public_html* directory structure. This way if the PHP interpreter is down and an attacker is able to view the PHP source, the database-access information and database queries will still be hidden from him. The files for presenting the queried data (sections 4.1, 4.2, and 4.5) are, however, placed inside the *public_html* directory structure importing the files for database access by the appropriate PHP commands. Separating data access and presentation assures that the modifications made to the data access or presentation part stay local and don't affect the whole application.

The directory structure of the sample application is the following:

```
php_include
    services.php
    queries.php
public_html
    layout.php
    vehicles.php
    vehicle-details.php
```

4.1 Listing (vehicles.php)

This web page lists the vehicles of the company in a table. Each registration number in the table is a link to a page containing detailed information on the corresponding vehicle. Above the list of vehicles the page has a list containing the days of the week. From this list more items can be selected at the same time. When submitting the page the web page filters the table and only lists the vehicles purchased on the selected days of the week.

Two outer files are imported by PHP structure **require**.

```
<?php

require("../../php_include/services.php");
require("../../php_include/queries.php");
```

Function **get_contents** queries all the data needed by the viewing page from the database. Its return value is an associative array with its keys indexing different kinds of data from queries and HTML forms. Under the key *vehicles* for example the array stores information on the queried vehicles where variable $vehicles is an array itself, with keys of numbers and values of records storing the properties of vehicles. The query itself is executed by function **get_vehicles**. If we'd like to find out whether the query was successful or not we can use function **print_r** which prints a complex variable (e.g. an array the elements of which are also arrays, etc.)

```
function get_contents() {
   // getting parameter days_of_week from POST source
   $days_of_week = get_array_value_of_param('days_of_week');

   // connecting the database before database operations
   $conn = db_connect();
   // querying data
   $vehicles = get_vehicles($conn, $days_of_week);
   // closing the connection after all the database operations
   db_disconnect($conn);

   $contents = array(
      'days_of_week' => $days_of_week,
      'vehicles' => $vehicles
   );
   // print_r($contents);
   return $contents;
}
```

The function on the right gets the multiple values (e.g. all the selected elements of a list) of the name attribute of a form sent by method POST. Function **array_key_exists** checks whether the value of the first parameter is a key of the associative array passed in the second parameter.

```
function get_array_value_of_param($name, $default = array()) {
   if (array_key_exists($name, $_POST) &&
         is_array($_POST[$name])) {
      return $_POST[$name];
   } else {
      return $default;
   }
}
```

Function **print_contents** formats the data returned by **get_contents**. Function **print_days** generates a list of the days of the week highlighting the days the keys of which can be found in array $days_of_week. (This variable is filled up by function **get_contents** based on which list elements were selected by the user on the HTML form sent)

As you can see, instead of using commands echo and print we step out of PHP mode in the body of the function and use simple HTML for a while before re-entering PHP mode. This solution is practical if there are many HTML commands in a row and relatively few PHP commands.

It's important to note some problems emerged by special characters. There are a few characters resulting in documents producing non well-formed and often errorneous outputs when directly embedded into HTML. One of these is *&* (ampersand) character used for introducing so-called HTML entities like *–* (dash). The character set of URLs are also limited. If the data to be presented contains special characters these have to be converted by functions **htmlspecialchars** (in case of embedding data in HTML output) and **urlencode** (in case of embedding data in a URL).

```
function print_contents($contents) {
   $days_of_week = $contents['days_of_week'];
   ?>
   <form method="post">
   <table>
   <tr>
      <td>Show vehicles purchased on</td>
      <td><select name="days_of_week[]" multiple="multiple">
         <?php print_days($days_of_week); ?>
      </select></td>
   </tr>
   <tr>
      <td colspan="2">
         <input type="submit" />
      </td>
   </tr>
   </table>
   </form>
   <table>
      <tr>
         <th>Numberplate</th>
         <th>Date of purchase</th>
         <th>Date of last maintenance check</th>
      </tr>
   <?php
   foreach ($contents['vehicles'] as $vehicle) {
      print "<tr>";
      $identifier = urlencode($vehicle['NUMBERPLATE']);
      print "<td>
      <a href='vehicle-details.php?numberplate={$identifier}'>
         {$vehicle['NUMBERPLATE']}</a></td>";
      print "<td>{$vehicle['PURCHASE_DATE']}</td>";
      print "<td>{$vehicle['MAINTENANCE_DATE']}</td>";
      print "</tr>";
   }
   ?>
   </table>
   <?php
}

function print_days($days_of_week) {
   for ($i = 1; $i <= 7; $i++) {
      print "<option ";
      if (array_search($i, $days_of_week) !== FALSE) {
         print "selected='selected'";
      }
      $day_name = get_name_of_day($i);
      print "value='{$i}'>{$day_name}</option>\n";
   }
}

require("layout.php");

?>
```

Data presentation is defined in a separate file using functions **get_contents** and **print_contents** for viewing data.

4.2 Vehicle details page (vehicle-details.php)

This page lists the properties of vehicles in a table. Property names appear on the left side while the properties themselves appear on the right.

The first step is to import the functions for database access and executing queries.

```
<?php

require("../../php_include/services.php");
require("../../php_include/queries.php");
```

The function on the right returns the numerical value of attribute *name* obtained from the URL (that is, passed by method GET). If there's no such attribute it returns a predefined default value.

```
function get_integer_value_of_parameter($name, $default =
FALSE) {
   if (array_key_exists($name, $_GET)) {
      $value = $_GET[$name];
      if (is_numeric($value)) {
         return (int) $value;
      }
   }
   return $default;
}
```

This function is similar to the previous one. It returns the string value of a parameter passed in the URL.

```
function get_string_value_of_parameter($name, $default =
FALSE) {
   if (array_key_exists($name, $_GET)) {
      return $_GET[$name];
   } else {
      return $default;
   }
}
```

Similarly to the page listing all the vehicles function **get_contents** queries all the data needed from the database. Function **get_contents** of the listing page had a return value of an associative array with key *vehicles* storing an array (indexed by numbers) of associative arrays containing information on different vehicles. In the present case however, the page has to view the properties of a single vehicle, thus our string keys directly index the properties of that single vehicle.

```
function get_contents() {
   $conn = db_connect();
   $numberplate =
get_string_value_of_parameter('numberplate');
   if ($numberplate !== FALSE) {
      $details = get_vehicle_details($conn, $numberplate);
      if (count($details) > 0) {
         $details = $details[0];
      } else {
         $details = FALSE;
      }
   } else {
      $details = FALSE;
   }
   db_disconnect($conn);
   $contents = array('details' => $details);
   return $contents;
}
```

In order to prevent users from meaningless codes used in the database these codes have to be resolved and the corresponding names have to be returned.

```
function get_type_string($type) {
   switch ($type) {
      case 'g': return 'gasoline tank';
      case 'c': return 'container chassis';
      case 'd': return 'dump';
      case 'v': return 'dry freight van';
      case 'l': return 'livestock trailer';
      case 'a': return 'auto carrier';
      default: return 'not specified';
   }
}
```

Just like in the earlier cases output is generated by a separate function based on the data returned by function **get_contents**.
As you might have noticed there's a long series of HTML commands in the function. Unlike the previous case like this, in this function we use the so-called *heredoc* syntax instead of leaving PHP mode. In this syntax the rules of substituting variables are the same as in the case of using quote marks but instead of a quote mark the beginning of the string is indicated by <<< followed by an arbitrary token. The string lasts until PHP finds this token again at the beginning of a line. If the token is not at the beginning of a line it becomes a part of the string instead of terminating it. The token can only be followed by an optional semicolon terminating the command (like in our example).

```
function print_contents($contents) {
   $vehicle = $contents['details'];
   if ($vehicle !== FALSE) {
      $type = get_type_string($vehicle['VEHICLE_TYPE']);
      $maintenance = $vehicle['MAINTENANCE_DATE'];
      $maintenance = $maintenance === NULL ?
         "&lt; field not specified &gt;" : $maintenance;
      print <<<TABLE
      <table>
      <tr>
         <td>Numberplate:</td>
         <td>{$vehicle['NUMBERPLATE']}</td>
      </tr>

      <tr>
         <td>Type of vehicle:</td>
         <td>{$type}</td>
      </tr>

      <tr>
         <td>Date of purchase:</td>
         <td>{$vehicle['PURCHASE_DATE']}</td>
      </tr>

      <tr>
         <td>Date of last maintenance check:</td>
         <td>{$maintenance}</td>
      </tr>
      </table>
TABLE;
   } else {
      print "<p>No such vehicle, invalid identifier.</p>";
   }
}

require("layout.php");

?>
```

4.3 Database access (services.php)

To make our application portable, the functions of database access are stored in a separate file. Packed in each of these functions there's a database specific function. This packing approach makes it possible to adjust the parameters of functions centrally, masking the differences of different database manager versions and ensuring some kind of an independence from different manufacturers: questions related to database access are all concentrated here.

An important question is that of character coding. In order to make our data readable on the output, the coding of data queried from the database has to be the same as the coding of the displayed page.[10] The desired behavior can be achieved by an appropriate parameterization of the database driver.

[10] The file structure generated in the beginning of the workshop guarantees this.

User name, password, and the coding of result sets and the web page are passed by constants. According to our former issues of security it's important not to store this file inside the *public_html* directory.

```
<?php

define(DB_USER, 'levente_hunyadi');
define(DB_PASSWORD, 'password');
define(DB_CHARSET, 'utf8');
define(XHTML_CHARSET, 'utf-8');
```

Function **db_connect** connects the database and returns a connection identifier.

```
function db_connect() {
   $connection = oci_connect(DB_USER, DB_PASSWORD, "szglab",
DB_CHARSET);
   if ($connection === FALSE) {
      db_fatal_error();
   } else {
      return $connection;
   }
}
```

Function **db_disconnect** terminates the connection corresponding to the given connection identifier.

```
function db_disconnect($connection) {
   return oci_close($connection);
}
```

Function **db_create_statement** creates a new SQL statement. Like in the case of other languages learned in this course instead of inserting parameters directly into the SQL query we use parameterized queries with the values of parameters bound to the statement later.

```
function db_create_statement($connection, $query) {
   $statement = oci_parse($connection, $query);
   if ($statement === FALSE) {
      print "<p>Illegal query:<br/>{$query}</p>";
      db_fatal_error();
   } else {
      return $statement;
   }
}
```

Function **db_destroy_statement** frees up the memory used by the SQL query.

```
function db_destroy_statement($statement) {
   return oci_free_statement($statement);
}
```

Function **db_execute** executes an SQL statement with no default COMMIT.

```
function db_execute($statement) {
   return oci_execute($statement, OCI_DEFAULT);
}
```

Function **db_bind** binds a variable of a parameterized SQL statement to a value.

```
function db_bind($statement, $name, $value) {
   return oci_bind_by_name($statement, $name, $value);
}
```

Function **db_fetch_resultset** gets a series of rows of the result set. Its return value is an array with keys corresponding to column names of the result set. Parameters *skip* and *maxrows* adjust the number of rows to be omitted from the result set (starting with the first row) and the maximal number of rows to be returned. They can be used for turning listing pages.

```
function db_fetch_resultset($statement,$skip=0,$maxrows=-1) {
   $array = array();
   if (oci_execute($statement, OCI_DEFAULT)) {
      oci_fetch_all($statement, $array, $skip, $maxrows,
         OCI_FETCHSTATEMENT_BY_ROW + OCI_ASSOC);
      return $array;
   } else {
      return FALSE;
   }
}
```

Function **db_fatal_error** stops running the PHP script in case of a critical database error and prints an error message.

```
function db_fatal_error() {
   $error = oci_error();
   print htmlentities($error['message']);
   exit;
}
```

Function **set_content_type** sets the content type (e.g. xhtml) and character coding (e.g. utf-8) used when generating the output. If content type is xhtml+xml some of the browsers are able to check whether the page is well-formed (in xml means), while other browsers rather get confused.

```
function set_content_type() {
   $subpatters = array();
   if (preg_match('|Firefox/(\d+(?:[.]\d+)+)|ui',
         $_SERVER['HTTP_USER_AGENT'], $subpatterns)) {
      header('Content-type: application/xhtml+xml; charset=' .
         XHTML_CHARSET);
   } else {
      header('Content-type: text/html; charset=' .
         XHTML_CHARSET);
   }
}

set_content_type();

?>
```

Since an HTTP response consists of a header and a payload, a call of the function **set_content_type** has to be the first printing command. It's important that even space characters on the top of the page source (before <?php) are considered to be printing commands. The restriction stands for the PHP script referencing the page as well, of course.

4.4 Queries (queries.php)

Function **get_vehicles** returns a list of vehicles. Its return value is an array containing records indexed by numbers. Each one of these records corresponds to a vehicle. Thus, the keys of the associative array are numbers referencing associative arrays storing properties of vehicles as pairs of field names and field values. (The return value of **get_vehicles** will be used as parameter *$vehicles* of function **get_contents** in *vehicles.php*.) Unfortunately, in this case it's not possible to use a fully parameterized query, since the selected days of the week make up a list with variable length, and handling this kind of a list is beyond the possibilities of parameterized queries.

```
<?php

function get_vehicles($conn, $days_of_week = array()) {
   $values = array();
   foreach ($days_of_week as $day) {
      if ($day > 0 && $day <= 7) {
         $values[] = $day;
      }
   }
   if (count($values) > 0) {
      $value_list = implode(', ', $values);
      $condition =
         "TO_CHAR(date_of_purchase, 'd') IN ({$value_list})";
   } else {
      $condition = '1 = 1';
   }

   $statement = db_create_statement($conn, "
      SELECT numberplate,
         TO_CHAR(date_of_purchase, 'mm/dd/yyyy') AS
            purchase_date,
         TO_CHAR(last_checked, 'mm/dd/yyyy') AS
            maintenance_date
      FROM vehicles
      WHERE {$condition}");
   $resultset = db_fetch_resultset($statement);
   return $resultset;
}
```

Function **get_vehicle_details** returns the detailed properties of a vehicle in an associative array. The registration number is present in the output (although it is used as the search key). The main benefit of this solution is that it helps us to generate the output, since the value of the registration number doesn't have to be retrieved from the URL, as it is stored in the associative array containing the result set.

```
function get_vehicle_details($conn, $numberplate) {
   $statement = db_create_statement($conn, "
      SELECT numberplate, vehicle_type,
         TO_CHAR(date_of_purchase, 'mm/dd/yyyy') AS
            purchase_date,
         TO_CHAR(last_checked, 'mm/dd/yyyy') AS
            maintenance_date
      FROM vehicles
      WHERE numberplate = :numberplate");
   db_bind($statement, ':numberplate', $numberplate);
   return db_fetch_resultset($statement);
}

?>
```

4.5 Output (layout.php)

Finally, without any comments, we show the page *layout.php* for generating the output:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=<?php print XHTML_CHARSET; ?>" />
<title>Lakemacher and Co. Shipping International</title>
<link rel="stylesheet" type="text/css" href="style.css" />
</head>

<body>

<!-- place of header -->
<p>Lakemacher &amp;amp; Co. Shipping International</p>
<hr />

<!-- place of body -->
<?php
       $data = get_contents();
       print_contents($data);
?>

<!-- place of footer -->
<hr />

</body>
</html>
```