CpSc 462/662: Database Management Systems (DBMS) (TEXNH Approach)

PHP Programming

James Wang

Foreword

- This lecture will only discuss the basic concepts of PHP programming language.
- How to access MySQL database through PHP script will also be introduced.
- However, you need advanced knowledge on PHP and its MySQL extension to complete your project.
- If you encounter difficulties in your project, please consult the PHP document at:

  http://www.php.net
- The instructor will guide you to solve the problems in your project but will not help you to write the program.

What is PHP?

- PHP ("*P*HP: *H*ypertext *P*reprocessor") is a widely-used Open Source general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. PHP is a powerful server-side scripting language for creating dynamic and interactive websites.
- Example:

```
<html>
    <head>
      <title>Example</title>
    </head>
    <body>
      <?php
        echo "Hi, I'm a PHP script!";
      ?>
    </body>
</html>
```


PHP Syntax

- PHP is a server-side scripting language. Therefore, you cannot see the PHP code in your web browser. Instead, you will see the HTML code generated by your PHP code.
- Basic PHP Syntax:
  - A PHP scripting block always starts with **<?php** and ends with **?>**. A PHP scripting block can be placed anywhere in the HTML document.
- Comments in PHP:
  - In PHP, we use // to make a single-line comment or /* and */ to make a large comment block.
- Alernative PHP Syntax:

```
<script language="php">
    echo 'some editors (like FrontPage) don\'t like processing
    instructions'
</script>
```


PHP Data Types

- PHP supports eight primitive types.
  - Four scalar types: boolean, integer, float (double), string.
  - Two compound types: array, object.
  - Two special types: resource, NULL.
- The type of a variable is usually not set by the programmer; rather, it is decided at runtime by PHP depending on the context in which that variable is used.
- If you would like to force a variable to be converted to a certain type, you may either cast the variable or use the settype() function on it.

PHP Data Types (cont.)

```
<?php
    $a_bool = TRUE;   // a boolean
    $a_str  = "foo";  // a string
    $a_str2 = 'foo';  // a string
    $an_int = 12;     // an integer

    echo gettype($a_bool); // prints out:  boolean
    echo gettype($a_str);  // prints out:  string

    // If this is an integer, increment it by four
    if (is_int($an_int)) {
        $an_int += 4;
    }

    // If $bool is a string, print it out
    // (does not print out anything)
    if (is_string($a_bool)) {
        echo "String: $a_bool";
    }
?>
```

PHP Variables

- **Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.**
- **Variable names follow the same rules as other labels in PHP.**
  - A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

```
<?php
        $var = 'Bob';
        $Var = 'Joe';
        echo "$var, $Var";      // outputs "Bob, Joe"

        $4site = 'not yet';     // invalid; starts with a number
        $_4site = 'not yet';    // valid; starts with an underscore
        $täyte = 'mansikka';    // valid; 'ä' is (Extended) ASCII 228.
?>
```

7

PHP Variables (cont.)

- **To assign by reference, simply prepend an ampersand (&) to the beginning of the variable which is being assigned (the source variable).**

```
<?php
    $foo = 'Bob';              // Assign the value 'Bob' to $foo
    $bar = &$foo;              // Reference $foo via $bar.
    $bar = "My name is $bar";  // Alter $bar...
    echo $bar;
    echo $foo;                 // $foo is altered too.
?>
```

- **If a variable references another variable, change one variable will affect another.**
- **For instance, the above code snippet outputs 'My name is Bob' twice.**

8

PHP Constants

- **A constant is an identifier (name) for a simple value, which won't change during the execution of the script**
- **A constant is case-sensitive. By convention, constant identifiers are always uppercase.**

```
<?php

// Valid constant names
define("FOO",     "something");
define("FOO2",    "something else");
define("FOO_BAR", "something more");

// Invalid constant names
define("2FOO",    "something");

// This is valid, but should be avoided:
// PHP may one day provide a magical constant
// that will break your script
define("__FOO__", "something");

?>
```

9

PHP Constants (cont.)

- **You can define a constant by using the define()-function. Once a constant is defined, it can never be changed or undefined.**
- **Only scalar data (boolean, integer, float and string) can be contained in constants. Do not define resource constants.**
- **You can get the value of a constant by simply specifying its name. Unlike with variables, you should *not* prepend a constant with a *$*.**
- **You can also use the function constant() to read a constant's value if you wish to obtain the constant's name dynamically.**

10

PHP Expressions

- **In PHP, almost anything you write is an expression, or "anything that has a value" is an expression.**

```
<?php
    function double($i)
    {
       return $i*2;
    }
    $b = $a = 5;        /* assign the value five into the variable $a and $b */
    $c = $a++;          /* post-increment, assign original value of $a (5) to $c */
    $e = $d = ++$b;     /* pre-increment, assign the incremented value of
            $b (6) to $d and $e . at this point, both $d and $e are equal to 6 */

    $f = double($d++);  /* assign twice the value of $d before
                        the increment, 2*6 = 12 to $f */
    $g = double(++$e);  /* assign twice the value of $e after
                        the increment, 2*7 = 14 to $g */
    $h = $g += 10;      /* first, $g is incremented by 10 and ends with the
            value of 24. the value of the assignment (24) is then assigned
            into $h, and $h ends with the value of 24 as well. */
?>
```

11

Operators and Operator Precedence

Associativity	Operators
non-associative	new
left	[
non-associative	++ --
non-associative	~ - (int) (float) (string) (array) (object) @
non-associative	instanceof
right	!
left	* / %
left	+ - .
left	<< >>
non-associative	< <= > >=

Associativity	Operators
non-associative	== != === !==
left	&
left	^
left	|
left	&&
left	||
left	? :
right	= += -= *= /= .= %= &= |= ^= <<= >>=
left	and
left	xor
left	or
left	,

12

Arithmetic Operators

Example	Name	Result
-$a	Negation	Opposite of $a.
$a + $b	Addition	Sum of $a and $b.
$a - $b	Subtraction	Difference of $a and $b.
$a * $b	Multiplication	Product of $a and $b.
$a / $b	Division	Quotient of $a and $b.
$a % $b	Modulus	Remainder of $a divided by $b.

13

Assignment Operators

- **The basic assignment operator is "=". It means that the left operand gets set to the value of the expression on the right.**

```
<?php
    $a = ($b = 4) + 5; // $a is equal to 9 now, and $b has been set to 4.
?>
```

- **In addition to the basic assignment operator, there are "combined operators" for all of the binary arithmetic, array union and string operators that allow you to use a value in an expression and then set its value to the result of that expression.**

```
<?php
    $a = 3;
    $a += 5; // sets $a to 8, as if we had said: $a = $a + 5;
    $b = "Hello ";
    $b .= "There!"; // sets $b to "Hello There!", just like $b = $b . "There!";
?>
```

14

Bitwise Operators

Example	Name	Result
$a & $b	And	Bits that are set in both $a and $b are set.
$a | $b	Or	Bits that are set in either $a or $b are set.
$a ^ $b	Xor	Bits that are set in $a or $b but not both are set.
~ $a	Not	Bits that are set in $a are not set, and vice versa.
$a << $b	Shift left	Shift the bits of $a $b steps to the left (each step means "multiply by two")
$a >> $b	Shift right	Shift the bits of $a $b steps to the right (each step means "divide by two")

15

Comparison Operators

Example	Name	Result
$a == $b	Equal	**TRUE** if $a is equal to $b.
$a === $b	Identical	**TRUE** if $a is equal to $b, and they are of the same type. (introduced in PHP 4)
$a != $b	Not equal	**TRUE** if $a is not equal to $b.
$a <> $b	Not equal	**TRUE** if $a is not equal to $b.
$a !== $b	Not identical	**TRUE** if $a is not equal to $b, or they are not of the same type. (introduced in PHP 4)
$a < $b	Less than	**TRUE** if $a is strictly less than $b.
$a > $b	Greater than	**TRUE** if $a is strictly greater than $b.
$a <= $b	Less than or equal to	**TRUE** if $a is less than or equal to $b.
$a >= $b	Greater than or equal to	**TRUE** if $a is greater than or equal to $b.

16

Error Control Operators

- **PHP supports one error control operator: the at sign (@). When prepended to an expression in PHP, any error messages that might be generated by that expression will be ignored.**
- **If the *track_errors* feature is enabled, any error message generated by the expression will be saved in the variable $php_errormsg. This variable will be overwritten on each error, so check early if you want to use it.**

```
<?php
    /* Intentional file error */
    $my_file = @file ('non_existent_file') or
        die ("Failed opening file: error was '$php_errormsg'");

    // this works for any expression, not just functions:
    $value = @$cache[$key];
    // will not issue a notice if the index $key doesn't exist.

?>
```

17

Execution Operators

- **PHP supports one execution operator: backticks (``).**
  - Note that these are not single-quotes!
- **PHP will attempt to execute the contents of the backticks as a shell command; the output will be returned (i.e., it won't simply be dumped to output; it can be assigned to a variable).**
- **Use of the backtick operator is identical to shell_exec().**

```
<?php
    $output = `ls -al`;
    echo "<pre>$output</pre>";
?>
```

18

Incrementing/Decrementing Operators

Example	Name	Effect
++$a	Pre-increment	Increments $a by one, then returns $a.
$a++	Post-increment	Returns $a, then increments $a by one.
--$a	Pre-decrement	Decrements $a by one, then returns $a.
$a--	Post-decrement	Returns $a, then decrements $a by one.

- **PHP follows Perl's convention when dealing with arithmetic operations on character variables and not C's. For example, in Perl 'Z'+1 turns into 'AA', while in C 'Z'+1 turns into '[' (ord('Z') == 90, ord('[') == 91).**
  - Note that character variables can be incremented but not decremented and even so only plain ASCII characters (a-z and A-Z) are supported.

```
<?php
    $i = 'W';
    for ($n=0; $n<6; $n++) {
        echo ++$i . "\n";
    }
?>
```

```
Output:
X
Y
Z
AA
AB
AC
```

19

Logical Operators

Example	Name	Result
$a and $b	And	**TRUE** if both $a and $b are **TRUE**.
$a or $b	Or	**TRUE** if either $a or $b is **TRUE**.
$a xor $b	Xor	**TRUE** if either $a or $b is **TRUE**, but not both.
! $a	Not	**TRUE** if $a is not **TRUE**.
$a && $b	And	**TRUE** if both $a and $b are **TRUE**.
$a || $b	Or	**TRUE** if either $a or $b is **TRUE**.

20

String Operators

- **There are two string operators. The first is the concatenation operator ('.'), which returns the concatenation of its right and left arguments.**
- **The second is the concatenating assignment operator ('.='), which appends the argument on the right side to the argument on the left side.**

```
<?php
    $a = "Hello ";
    $b = $a . "World!"; // now $b contains "Hello World!"

    $a = "Hello ";
    $a .= "World!";     // now $a contains "Hello World!"
?>
```

21

Array Operators

Example	Name	Result
$a + $b	Union	Union of $a and $b.
$a == $b	Equality	**TRUE** if $a and $b have the same key/value pairs.
$a === $b	Identity	**TRUE** if $a and $b have the same key/value pairs in the same order and of the same types.
$a != $b	Inequality	**TRUE** if $a is not equal to $b.
$a <> $b	Inequality	**TRUE** if $a is not equal to $b.
$a !== $b	Non-identity	**TRUE** if $a is not identical to $b.

- **The + operator appends elements of remaining keys from the right handed array to the left handed, whereas duplicated keys are NOT overwritten.**

22

Control Structures

- **if, else, elseif:**

```
<?php
    if ($a > $b) {
        echo "a is bigger than b";
    } elseif ($a == $b) {
        echo "a is equal to b";
    } else {
        echo "a is smaller than b";
    }
?>
```

- **While:**

```
<?php
    $i = 1;
    while ($i <= 10) {
        echo $i++;  /* the printed value would be
                       $i before the increment (post-increment) */
    }
?>
```

23

Control Structures (cont.)

- **do … while:**

```
<?php
    $i = 0;
    do {
        echo $i++;
    } while ($i < 5);
?>
```

- **While:**

```
<?php
    $i = 1;
    while ($i <= 10) {
        echo $i++;  /* the printed value would be
                       $i before the increment (post-increment) */
    }
?>
```

24

for

for (expr1; expr2; expr3) statement

- **The first expression (*expr1*) is evaluated (executed) once unconditionally at the beginning of the loop.**
- **In the beginning of each iteration, *expr2* is evaluated. If it evaluates to TRUE, the loop continues and the nested statement(s) are executed. If it evaluates to FALSE, the execution of the loop ends.**
- **At the end of each iteration, *expr3* is evaluated (executed).**

25

foreach

foreach (array_expression as $value)
  statement
foreach (array_expression as $key => $value)
  statement

- **The first form loops over the array given by *array_expression*. On each loop, the value of the current element is assigned to *$value* and the internal array pointer is advanced by one (so on the next loop, you'll be looking at the next element).**
- **The second form does the same thing, except that the current element's key will be assigned to the variable *$key* on each loop.**

```
<?php
    $arr = array(1, 2, 3, 4);
    foreach ($arr as &$value) {
        $value = $value * 2;
    }
    // $arr is now array(2, 4, 6, 8)
    unset($value); // break the reference with the last element
?>
```

26

break

- ***break* ends execution of the current *for*, *foreach*, *while*, *do-while* or *switch* structure. *break* accepts an optional numeric argument which tells it how many nested enclosing structures are to be broken out of.**

```
<?php
    $arr = array('one', 'two', 'three', 'four', 'stop', 'five');
    while (list(, $val) = each($arr)) {
        if ($val == 'stop') {
            break;    /* You could also write 'break 1;' here. */
        }
        echo "$val<br />\n";
    }

    $i = 0;
    while (++$i) {
        switch ($i) {
        case 5:
            echo "At 5<br />\n";
            break 1;  /* Exit only the switch. */
        case 10:
            echo "At 10; quitting<br />\n";
            break 2;  /* Exit the switch and the while. */
        default:
            break;
        }
    }
?>
```

27

switch

- **The *switch* statement is similar to a series of IF statements on the same expression. With switch, you can compare the same variable (or expression) with many different values, and execute a different piece of code depending on which value it equals to.**

```
<?php
    if ($i == 0) {
        echo "i equals 0";
    } elseif ($i == 1) {
        echo "i equals 1";
    } elseif ($i == 2) {
        echo "i equals 2";
    }

    switch ($i) {
    case 0:
        echo "i equals 0";
        break;
    case 1:
        echo "i equals 1";
        break;
    case 2:
        echo "i equals 2";
        break;
    }
?>
```

28

continue

- ***continue* is used within looping structures to skip the rest of the current loop iteration and continue execution at the condition evaluation and then the beginning of the next iteration. *continue* accepts an optional numeric argument which tells it how many levels of enclosing loops it should skip to the end of.**

```
<?php
    while (list($key, $value) = each($arr)) {
        if (!($key % 2)) { // skip odd members
            continue;
        }
        do_something_odd($value);
    }
    $i = 0;
    while ($i++ < 5) {
        echo "Outer<br />\n";
        while (1) {
            echo "  Middle<br />\n";
            while (1) {
                echo "  Inner<br />\n";
                continue 3;
            }
            echo "This never gets output.<br />\n";
        }
        echo "Neither does this.<br />\n";
    }
?>
```

29

return

- **If called from within a function, the return() statement immediately ends execution of the current function, and returns its argument as the value of the function call. return() will also end the execution of an eval() statement or script file.**
- **If called from the global scope, then execution of the current script file is ended. If the current script file was include()ed or require()ed, then control is passed back to the calling file. Furthermore, if the current script file was include()ed, then the value given to return() will be returned as the value of the include() call.**
- **If return() is called from within the main script file, then script execution ends. If the current script file was named by the auto_prepend_file or auto_append_file configuration options in *php.ini*, then that script file's execution is ended.**

30

require() and include()

- **The require() or include() statement includes and evaluates the specific file.**
- **require() and include() are identical in every way except how they handle failure.**
  - They both produce a Warning, but require() results in a Fatal Error. In other words, don't hesitate to use require() if you want a missing file to halt processing of the page.
  - include() does not behave this way, the script will continue regardless. Be sure to have an appropriate include_path setting as well.

```
<?php
    require 'prepend.php';
    require $somefile;
    require ('somefile.txt');
    include 'vars.php';
?>
```

31

require_once() and include_once()

- **The require_once() or include_once() statement includes and evaluates the specific file during the execution of the script.**
- **This is a behavior similar to the require() or include() statement, with the only difference being that if the code from a file has already been included, it will not be included again.**

```
<?php
    require_once("a.php"); // this will include a.php
    require_once("a.php"); // this will not include a.php again
    include_once("b.php"); // this will include a.php
    include_once("b.php"); // this will not include a.php again
?>
```

32

Functions

- **User defined functions:**

```
<?php
    function foo($arg_1, $arg_2, /* ..., */ $arg_n)
    {
        echo "Example function.\n";
        return $retval;
    }
?>
```

- **Function arguments:**
  - Information may be passed to functions via the argument list, which is a comma-delimited list of expressions.

33

Passing by reference

- **By default, function arguments are passed by value (so that if you change the value of the argument within the function, it does not get changed outside of the function).**
- **If you wish to allow a function to modify its arguments, you must pass them by reference.**
- **If you want an argument to a function to always be passed by reference, you can prepend an ampersand (&) to the argument name in the function definition.**

```
<?php
    function add_some_extra(&$string)
    {
        $string .= 'and something extra.';
    }
    $str = 'This is a string, ';
    add_some_extra($str);
    echo $str;    // outputs 'This is a string, and something extra.'
?>
```

34

Returning values

- **Values are returned by using the optional return statement. Any type may be returned, including lists and objects. You can't return multiple values from a function, but similar results can be obtained by returning a list.**

```
<?php
    function small_numbers()
    {
        return array (0, 1, 2);
    }
    list ($zero, $one, $two) = small_numbers();
?>
```

- **To return a reference from a function, you have to use the reference operator & in both the function declaration and when assigning the returned value to a variable.**

```
<?php
    function &returns_reference()
    {
        return $someref;
    }
    $newref =& returns_reference();
?>
```

35

Variable functions

- **PHP supports the concept of variable functions. This means that if a variable name has parentheses appended to it, PHP will look for a function with the same name as whatever the variable evaluates to, and will attempt to execute it.**

```
<?php
    function foo() {
        echo "In foo()<br />\n";
    }

    function bar($arg = '')
    {
        echo "In bar(); argument was '$arg'.<br />\n";
    }

    $func = 'foo';
    $func();        // This calls foo()

    $func = 'bar';
    $func('test');  // This calls bar()
?>
```

36

Variable functions (cont.)

- Among other things, variable functions can be used to implement callbacks, function tables, and so forth.
- Variable functions won't work with language constructs such as echo(), print(), unset(), isset(), empty(), include(), require() and the like. You need to use your own wrapper function to utilize any of these constructs as variable functions.

```
<?php
    // This is a wrapper function around echo
    function echoit($string)
    {
        echo $string;
    }

    $func = 'echoit';
    $func('test');  // This calls echoit()
?>
```

- You can also call an object's method by using the variable functions feature.

Internal (built-in) functions

- PHP comes standard with many functions and constructs. There are also functions that require specific PHP extensions compiled in otherwise you'll get fatal "undefined function" errors.
  - For example, to use image functions such as imagecreatetruecolor(), you'll need your PHP compiled with GD support. Or, to use mysql_connect() you'll need your PHP compiled in with MySQL support.
- There are many core functions that are included in every version of PHP like the string and variable functions.
  - A call to phpinfo() or get_loaded_extensions() will show you which extensions are loaded into your PHP.
- Also note that many extensions are enabled by default and that the PHP manual is split up by extension. See the configuration, installation, and individual extension chapters, for information on how to setup your PHP.

Exceptions

- PHP 5 has an exception model similar to that of other programming languages.
- An exception can be *thrown*, and caught ("*catched*") within PHP. Code may be surrounded in a *try* block, to facilitate the catching of potential exceptions.
- Each *try* must have at least one corresponding *catch* block. Multiple *catch* blocks can be used to catch different classes of exeptions.
- Normal execution (when no exception is thrown within the *try* block, or when a *catch* matching the thrown exception's class is not present) will continue after that last catch block defined in sequence.
- Exceptions can be *thrown* (or re-thrown) within a *catch* block.

Throwing an Exception

```
<?php
    function inverse($x) {
        if (!$x) {
            throw new Exception('Division by zero.');
        }
        else return 1/$x;
    }

    try {
        echo inverse(5) . "\n";
        echo inverse(0) . "\n";
    } catch (Exception $e) {
        echo 'Caught exception: ',  $e->getMessage(), "\n";
    }

    // Continue execution
    echo 'Hello World';

?>
```


Classes and Objects

- PHP introduced the object-oriented programming model in version 4 and revised the model entirely in version 5.
- A class is a collection of variables and functions working with these variables.
- For more information, please read:
  http://us.php.net/manual/en/language.oop.php
  http://us.php.net/manual/en/language.oop5.php

PHP $_GET

- The $_GET variable is used to collect values from a form with method="get".
  - The $_GET variable is an array of variable names and values sent by the HTTP GET method.
  - The $_GET variable is used to collect values from a form with method="get". Information sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and it has limits on the amount of information to send (max. 100 characters).
- Example:

```
<form action="welcome.php" method="get">
    Name: <input type="text" name="name" />
    Age: <input type="text" name="age" />
    <input type="submit" />
</form>
```

- When use click on submit button, address bar will have:
  http://webserver/welcome.php?name=Peter&age=37

PHP $_GET (cont.)

- **Your server side scripting file, "welcome.php", now must use the $_GET variable to catch the form data.**

```
Welcome <?php echo $_GET["name"]; ?>.<br />
You are <?php echo $_GET["age"]; ?> years old!
```

- **Notice that the names of the form fields will automatically be the ID keys in the $_GET array.**
- **Pros and Cons:**
  - When using the $_GET variable all variable names and values are displayed in the URL. So this method should not be used when sending passwords or other sensitive information!
  - However, because the variables are displayed in the URL, it is possible to bookmark the page. This can be useful in some cases.
  - The HTTP GET method is not suitable on large variable values; the value cannot exceed 100 characters.

PHP $_POST

- **The $_POST variable is an array of variable names and values sent by the HTTP POST method.**
- **The $_POST variable is used to collect values from a form with method="post". Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send.**
- **For example, for the same form, your script will look like:**

```
Welcome <?php echo $_POST["name"]; ?>.<br />
You are <?php echo $_POST["age"]; ?> years old!
```

- **Pros and Cons:**
  - Variables sent with HTTP POST are not shown in the URL
  - Variables have no length limit
  - However, because the variables are not displayed in the URL, it is not possible to bookmark the page.

PHP $_REQUEST

- **The PHP $_REQUEST variable contains the contents of both $_GET, $_POST, and $_COOKIE.**
- **The PHP $_REQUEST variable can be used to get the result from form data sent with both the GET and POST methods.**
- **For the same example, your script will look like:**

```
Welcome <?php echo $_REQUEST["name"]; ?>.<br />
You are <?php echo $_REQUEST["age"]; ?> years old!
```


References

- **http://us.php.net/manual/en/**
- **http://www.w3.org/TR/REC-html40/interact/forms.html**
- **http://www.w3schools.com/php/**