School of Information and Computer Technology
Sirindhorn International Institute of Technology
Thammasat University

ITS351 Database Programming Laboratory

Laboratory #2: PHP II

Objective:
- To learn commonly used functions in the standard PHP library.
- To introduce basic concept of object-oriented programming
- To learn how to create classes and objects in PHP

Resources:
- http://th.php.net/manual/en/language.oop5.php
- Nixon, Robin. Learning PHP, MySQL & Javascript. O'Reilly, 2009

1 More on PHP

In the previous lab, we have already covered basic constructs in PHP i.e., branching, statements, iteration, and arrays. Using these basic constructs, you have already created a fairly simple dynamically generated web page. To facilitate the creation of an even more complicated system, in this lab, we will dive deeper into some more commonly used functions in the standard PHP library. We will also introduce the basic concept of object-oriented programming (OOP) in PHP, as well as how to define classes and create objects in PHP. Understanding of classes and objects in PHP will allow you to quickly reuse many more 3rd-party libraries written by other people.

1.1 User-defined Functions

A function can be defined using the following syntax.

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

Any valid PHP code may appear inside a function, even other functions and class definitions. Function names follow the same rules as other labels in PHP. A valid function name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. Any type of values can be returned with the **return** keyword. Also, only one value can be returned. If multiple values are needed to be returned, they can be put in an array first, then return the array instead.

1.2 Optional Arguments

PHP also supports optional arguments when defining a function. Here is an example.

```php
<?php

function plus($x, $y=5){ // $y is default to 5
      return $x + $y;
}

echo plus(1, 2); // This prints 3
echo '<br>';
echo plus(1); // This prints 6
?>
```

There are two arguments needed to call plus(), namely $x and $y. When both of them are specified, the specified values are used for $x and $y. If only one argument is present, it is used as the value for $x, and $y is set to 5. Note that optional arguments must come after mandatory arguments. So, in general, you may want to position variables that have default values towards the end of the argument list.

1.3 Including Files

Breaking a big system into its smaller components, and implementing them with functions are common practice in procedural programming. To promote reusability, it is useful to group these functions into a file and include them when needed. This can be done in PHP with **require** directive.

```php
<?php
      // require "lib/library.php";
      require_once "lib/library.php";

      // Your code goes here
      // ...
?>
```

In the example, "lib/library.php" refers to the file named "library.php" which is in the folder named "lib". The folder "lib" is in the same folder as the current file. Using the **require** directive has the same effect as if the content of library.php is defined in the place where **require** is used. If PHP fails to load "lib/library.php", an error will be produced, and the execution will stop.

One problem with **require** is that if the same file is included more than once, an error message will be produced since the same functions are redefined with the exact same names. To solve this problem, **require_once** may be used instead. With **require_once**, subsequent **require** or **require_once** of the same file will be ignored, and no errors will be produced.

include and **include_once** are variants of the above directives, except that only a warning is produced (not an error) on loading failure. A warning does not stop an ongoing execution. Therefore, the use of **require_once** is safer and more preferable.

1.4 Single and Double Quoted Strings

As you might have already known, PHP allows both " (double quotes) and ' (single quote) to be used to specify a string. However, there are some subtle differences between the two in interpreting characters in the string. The following code will help you in understanding the differences.

```php
<?php
echo '1. String with single quote <br>';
echo "2. String with double quotes <br>";
echo 'There are
two new lines
in HTML source <br>';

echo '3. Backslash is needed for \' <br>';
echo "4. Backslash is not needed here ' <br>";

echo '5. Print one backslash \ <br>';
echo "6. Also one backslash \\ <br>";

echo 'No newline. Really see \n <br>';
echo "A newline in HTML source \n <br>";

echo '"Same" as next one <br>';
echo "\"Same\" as previous one <br>";
?>
```

The code will give the following output (already rendered by a web browser).

1. String with single quote
2. String with double quotes
There are two new lines in HTML source
3. Backslash is needed for '
4. Backslash is not needed here '
5. Print one backslash \
6. Also one backslash \
No newline. Really see \n
A newline in HTML source
"Same" as next one
"Same" as previous one

In short, double quotes "interpret" special characters a single quote does not. Note that in the output above, some characters like new lines cannot be seen after the browser renders them, because they collapsed to a single space. You are advised to try the code by yourself. Make sure to check the raw output of the produced result (not just rendered result).

Another difference of single and double quotes is variable expansion. With single quote, variables are not expanded, while double quoted strings will have all variable names replaced with their values.

```php
<?php
      $x = 'here';
      echo "Expand with \" like $x <br>";
      echo 'Will not expand with \' like $x';
?>
```

This code will produce

Expand with " like here
Will not expand with ' like $x

.

More complicated expressions like accessing an array with an index can also be expanded in a double quoted string. However, we omit the detail here since it is not used very often. Expansion of such complicated expression can actually be quite confusing.

For more information, see http://php.net/manual/en/language.types.string.php .

1.5 Debugging in PHP

There are many useful functions which can help you locate bugs quickly.

error_reporting([int $level])

error_reporting() sets the error reporting directive at runtime. Normally, changing configuration is done at php.ini. This function allows the change to be made at runtime. PHP has many levels of reporting errors. Here is some examples taken from http://www.php.net/manual/en/function.error-reporting.php .

```php
<?php

// Turn off all error reporting
error_reporting(0);

// Report simple running errors
error_reporting(E_ERROR | E_WARNING | E_PARSE);

// Reporting E_NOTICE can be good too (to report uninitialized
// variables or catch variable name misspellings ...)
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);

// Report all errors except E_NOTICE
// This is the default value set in php.ini
error_reporting(E_ALL ^ E_NOTICE);

// Report all PHP errors.
error_reporting(E_ALL);

// Report all PHP errors
error_reporting(-1);

?>
```

When you really cannot catch bugs, putting the following code at the beginning of the main file may help.

```php
// Turn on displaying errors
ini_set('display_errors', 'On');
error_reporting(E_ALL | E_STRICT);
```

This will result in more details of errors and notices (potential bugs) produced.

print_r($expression)

print_r() displays information about a variable in a way that is readable by humans. This is useful for checking whether or not a variable is properly assigned, as the whole structure will be printed (work even for a nested array).

```php
<?php
$a = array ('a' => 'apple', 'b' => 'banana',
  'c' => array ('x', 'y', 'z'));
print_r ($a);
?>
```


The code will produce

```
Array
(
    [a] => apple
    [b] => banana
    [c] => Array
        (
            [0] => x
            [1] => y
            [2] => z
        )
)
```

You may need to wrap the output in a <pre>..</pre> tag to preserve spaces and newlines. **var_dump($expression)** is also another function with similar functionality.

1.6 Passing by Reference

Primitive-typed (int, string) variables in PHP are passed to a function by value, meaning that their values are copied to the receiving function. Unlike in C, arrays in PHP are also passed by value. Locally changing a value of the variable which was passed by value does not have any effect once the function returns. Here is a demonstration.

```php
<?php
	$x = 1;
	fake_change($x);
	echo "After fake_change: ". $x . "<br>";
	real_change($x); # Caller side does not need &
	echo "After real_change: ". $x . "<br>";

	function fake_change($n){
		$n = 100;
	}

	function real_change(&$n){
		# There is no return here.
		$n = 100;
	}
?>
```

The code will produce

```
After fake_change: 1
After real_change: 100
```

.

In order to use a pass by reference, the receiving side must have & prepended to the variable (&$n in this case). The use of & to denote a variable's reference can also be used without a function like this.

```php
$a = &$b; // same as $b = &$a;
$b = 20;
echo $a; // print 20
```

In short, the statement `$a = &$b` simply means "$a and $b can be used interchangeably", as opposed to `$a = $b` which means "assign the value of $b to $a".

1.7 Variable Variables, and Variable Functions

Since PHP is an interpreted language, variable interpretation is done at runtime. This allows a dynamic variable reference (variable variables). Variable variables refer to act of accessing a variable by using the value of another one. Here is a demonstration

```php
<?php
	$x = 'hello';
	$hello = 'PHP';
	echo $x; // print: hello
	echo '<br>';
	echo $$x; //print: PHP
?>
```

Here **$$x** is not the same as **$x**. While just $x simply gives 'hello', with $$x, PHP will first interpret $x to be 'hello', which leaves us with $hello. Then, the value in $hello is obtained, which is 'PHP' in this case.

$$x → $hello → 'PHP'

The same trick can actually be used to call a function using a variable's value to specify the function name to call.

```php
<?php
	$a = 'foo';
	$a(); //Same as directly writing foo()

	function foo(){
		echo 'foo is called!';
	}
?>
```

Dynamically calling functions like this may save us from writing a lot of if-else statements because the value obtained can be used to directly call a function.

2 Classes and Objects

Until now, statements are grouped together into a so called "function" or "procedure". Grouping statements into such individual unit has many advantages over inline code. This includes:

- The semantic of the underlying sequence of statements is labeled with the function name.
- Reusability and maintainability are promoted.
- Functions act as an abstraction of complicated operations. Users of a well-written function do not need to care much about how it works as long as it works.

Objects take this concept a step further by incorporating both functions (called **"methods"**) and the data (called **"properties"**) they use, into a single structure called a class. A class is the template from which objects are created. Programming by organizing real-world entities into classes, and modeling their actions as methods is referred to as "Object-oriented Programming (OOP)".

For example, a class "Human" may have eat(), sleep(), work() as their associated methods. This basically says that all objects (people) of this class can at least do the action "eat",

"sleep", and "work". In OOP, although objects from the same class behave similarly, each object is treated as a unique existence i.e., just like people. In this case, the class "Human" may have a property "name" as its identity.

2.1 Class Definition

Class definitions can be put anywhere in the code, before or after the statements that use them. Defining a class in PHP is done with the **class** keyword as follows. The class definition syntax is similar to that of many popular programming languages e.g., Java, C++.

```
<?php
$p = new Person('Smith');
print_r($p); // Output: Person Object ( [name:private] => Smith )
//echo $p->name; //Error: Cannot access private property

class Person{
	private $name;

	function __construct($n){
		// __construct() is a special function for object constructor
		$this->set_name($n);
		// $this->name = $n; // is also fine.
	}
	function get_name(){
		return $this->name;
	}

	function set_name($n){
		$this->name = $n;
	}
}?>
```

Here we defined a class **Person** which has "name" as its property. The property is marked with the keyword **private** so that directly accessing the property from outside the class will give an error. Marking properties as **private** and providing a getter-setter method (in this case get_name() and set_name()) is considered a good practice (encapsulation).

__construct() is a special function representing the constructor of the class. In this case, the constructor of class Person also takes one argument, and assigns the value to the property *name* using the set_name() method. Notice that the use of **$this** keyword in $this->set_name($n) and $this->name is necessary to access the properties or methods defined in the class scope.

As mentioned before, a class is just a template for objects. To actually use it, an object has to be created with the **new** keyword, followed by the class name. If the class constructor requires parameters, they must be enclosed in parentheses () i.e., new Person('Smith'). If the constructor does not require any parameter, then parentheses are optional. Say, the class **Apple**'s constructor does not require a parameter. Then, both new Apple or new Apple() are valid for creating an Apple object.

As a side note, any function in PHP starting with two underscores is considered special, and is referred to as a **magic method**. Prefixing a newly created function name with two underscores is completely valid, but is discouraged since it is misleading. Also, in the past, the constructor function can be named with the class name, just like in Java. However, in the future version of PHP, naming a method with the class name will not be treated as a constructor anymore. It will instead be treated as a usual method. So, to create a constructor, the magic method **__constructor()** should be used.

2.2 Static Properties and Methods

Usual properties (instance properties) belong to objects not classes. In the case of the Person class, each object has its separated "name" property. That is, changing the value of "name" property of one object does not affect the value in another object. However, occasionally we may need to maintain data about a whole class. Specifically, we want all objects of a class to share the same piece of data. This can be achieved through the use of **static** keyword.

For example, let's say we would like to count the number of Person objects created in the program. This can be done as follows.

```php
<?php
$people = array(new Person('Smith'), new Person('Yamada'));
$c = Person::person_count(); // $c is 2

class Person{
	private static $count = 0;
	private $name;

	function __construct($n){
		$this->set_name($n);
		//Person::$count++  // is also fine
		self::$count++;
	}
	static function person_count(){
		//return Person::$count // is also fine
		return self::$count;
	}

	function get_name(){
		return $this->name;
	}

	function set_name($n){
		$this->name = $n;
	}
}
?>
```

In the class definition, we have added a static property called $count (`private static $count = 0`) initialized to 0, to keep track of how many Person objects are created. Every time a new Person object is created, the count is incremented by one (`self::$count++;`). In order to access a static method/property, the method/property has to be prefixed with its class name followed by :: (called **scope resolution operator**) e.g., Person::. So, `Person::$count` would mean the static variable $count defined in Person class. In the case of accessing such static methods/properties within the same class, the class name may be replaced with the **self** keyword, which literally means the class in the current context.

In the code, two Person objects have been created in the first line into an array. Then, `Person::person_count();` is used to query for the count. person_count() is a static method which returns the value of $count. So, at the end, $c will contain the value 2.

Notice that if $count is not declared static, then counting Person objects in this way will not work because each object will have its own $count which will always contain the value 1 (i.e., each object counts itself only once). Apart from being **static**, $count is also declared **private** to prevent code outside the class to change its value.

2.3 Class Inheritance

There are many occasions in which we need another class which is very similar to an existing one. Instead of copying the whole class definition, and modifying it, it is better to create the new class by extending from the original one. Creating a new class in this way is called "sub-classing", and is done with **extends** keyword. There are many advantages with this approach over redefining the whole thing from scratch.

- Besides additional properties/methods defined in the sub-class, the non-private properties/methods of the super-class (parent) are automatically inherited.
- The change of definition of the super-class is automatically propagated to its sub-classes.
- Less coding is involved.

Here is an example of an extended version of Person class which can keep "age".

```
class AgePerson extends Person{
      private $age;

      function __construct($n, $a){
            //Person::__construct($n); // is also fine
            parent::__construct($n);
            $this->set_age($a);
      }

      function get_age(){
            return $this->age;
      }

      function set_age($a){
            $this->age = $a;
      }
}
```

The new class AgePerson extends the Person class by adding the property $age. Sub-classing Person automatically gives AgePerson the methods get_name(), set_name(), as well as person_count(). The constructor of AgePerson class explicitly calls the constructor of Person class by using `parent::__construct($n);` which also passes $n (name) along. The `parent` refers to the super-class (parent class) of the current class, and is the counterpart of **self** keyword. Without the statement `parent::__construct($n);` the $name property will not be initialized with $n i.e., $name will be blank. Since the constructor of Person is also called, creation of an AgePerso object will also increment the count of Person objects. This actually makes sense since "An AgePerson is also a Person". So, it should also be counted. The method person_count() is also inheritied to AgePerson class. In this case, `AgePerson::person_count()` is exactly the same as `Person::person_count()`.

2.4 Visibility

PHP provides three keywords to control access to properties or methods of a class. They are **public**, **protected**, and **private**.

- **public** allows a direct access from anywhere, inside or outside a class. Methods are declared as public by default. Sub-classes also inherit public properties/methods from its parent.

- **protected** properties/methods can be referenced by methods within the same class, or by its subclass.
- **private** keyword is the most restricted. Private properties/methods can be referenced only by methods within the same class (not by subclasses).

As a general guideline, most of the time, properties are declared private, and interfacing methods are declared public. **protected** is used when inheritance is involved.

2.5 Passing Objects

Object passing works the same way as most other programming languages i.e. Java. A variable which contains an object is merely a reference to the object. Assigning a variable containing an object to another variable just results in the same object having two references. That is, no new object is created or cloned. Here is a code to demonstrate this.

```
$a = new Person('John');
$b = $a;
print_r($b); //Output: Person Object ( [name:private] => John )
$b->set_name('Dave');
echo '<br>';
print_r($a); //Output: Person Object ( [name:private] => Dave )
```

It can be seen in the code that $a refers to a Person object. The statement `$b = $a;` simply means "assign to $b the same object that $a contains". You can imagine the object as an apple, and imagine variables $a, and $b as your hands. In the beginning, one of your hand ($a) grabs the apple (`$a = new Person('John');`). We make the other hand ($b) grab the same apple with the statement $b = $a;. Since the two hands are grabbing the exact same apple, if the apple is changed in some ways, both hands should feel it. That is why in this code, even though we change the name through $b with `$b->set_name('Dave');`, the change is also seen in $a.

In the case of passing objects to a function, the change made to the objects inside the function will retain even after the function returns. To prevent the original object from being changed in a function, we can just create a new object and pass it to the function.

3 Frequently Used Functions

This section lists a few useful functions in PHP. There are many other functions offered in the standard library.

explode()

array explode (string $delimiter , string $string)

Split $string into many substrings. $delimiter is a string to be used as the delimiter (splitting points). $delimiter will not be included in the returned array.

```
$str = 'Paul,24,A+';
// $pieces = explode(',', $str); // $pieces is an array of 3 elements
list($name, $age, $grade) = explode(',', $str);
echo "Name: $name <br>";
echo "Age: $age <br>";
echo "Grade: $grade <br>";
```

The code will output

```
Name: Paul
Age: 24
Grade: A+
```

The **list** construct is a useful trick to assign multiple values from an array into a list of variables.

join(), implode()

string implode (string $glue , array $pieces)

join() is an alias of implode(). They do the same thing. implode() joins array elements $pieces with $glue. The function can be considered as a reverse operation of explode().

```php
$array = array('lastname', 'email', 'phone');
$comma_separated = implode(",", $array);
echo $comma_separated; // lastname,email,phone
```

trim()

string trim (string $str)

Return a string with whitespace (including newlines) stripped from the beginning and end of $str. This is useful in processing form input as users may accidentally put some spaces before of after, say, their username.

serialize()

string serialize (mixed $value)

Generates a storable string representation of a value. This is useful for storing or passing PHP values (e.g. integers, arrays, objects) around without losing their type and structure. To make the serialized string into a PHP value again, use **unserialize()**.

unserialize()

mixed unserialize (string $str)

unserialize() takes a single serialized variable and converts it back into a PHP value.

sprintf()

string sprintf (string $format [, mixed $args [, mixed $...]])

Returns a string produced according to the formatting string $format.

Each conversion specification consists of a percent sign (%), followed by one or more of these elements, in order:

1. An optional sign specifier that forces a sign (- or +) to be used on a number. By default, only the - sign is used on a number if it's negative. This specifier forces positive numbers to have the + sign attached as well.
2. An optional padding specifier that says what character will be used for padding the results to the right string size. This may be a space character or a 0 (zero character). The default is to pad with spaces. An alternate padding character can be specified by prefixing it with a single quote (').
3. An optional alignment specifier that says if the result should be left-justified or right-justified. The default is right-justified; a - character here will make it left-justified.

4. An optional number, a width specifier that says how many characters (minimum) this conversion should result in.
5. An optional precision specifier in the form of a period ('.') followed by an optional decimal digit string that says how many decimal digits should be displayed for floating-point numbers. When using this specifier on a string, it acts as a cutoff point, setting a maximum character limit to the string.
6. A type specifier that says what type the argument data should be treated as. Possible types:
   - % - a literal percent character. No argument is required.
   - b - the argument is treated as an integer, and presented as a binary number.
   - c - the argument is treated as an integer, and presented as the character with that ASCII value.
   - d - the argument is treated as an integer, and presented as a (signed) decimal number.
   - e - the argument is treated as scientific notation (e.g. 1.2e+2). The precision specifier stands for the number of digits after the decimal point.
   - E - like %e but uses uppercase letter (e.g. 1.2E+2).
   - u - the argument is treated as an integer, and presented as an unsigned decimal number.
   - f - the argument is treated as a float, and presented as a floating-point number (locale aware).
   - F - the argument is treated as a float, and presented as a floating-point number (non-locale aware). g - shorter of %e and %f.
   - G - shorter of %E and %f.
   - o - the argument is treated as an integer, and presented as an octal number.
   - s - the argument is treated as and presented as a string.
   - x - the argument is treated as an integer and presented as a hexadecimal number (with lowercase letters).
   - X - the argument is treated as an integer and presented as a hexadecimal number (with uppercase letters).

sprintf() works exactly the same way as printf(). With printf() the result is printed instead of returning as a string. Here are some examples.

```
$s = 'monkey';
$t = 'many monkeys';

printf("[%s]<br>",       $s); // standard string output
printf("[%10s]<br>",     $s); // right-justification with spaces
printf("[%-10s]<br>",    $s); // left-justification with spaces
printf("[%010s]<br>",    $s); // zero-padding works on strings too
printf("[%'#10s]<br>",   $s); // use the custom padding character '#'
printf("[%10.10s]<br>", $t); // left-justification but with a cutoff of
10 characters
```

The code gives


```
[monkey]
[    monkey]
[monkey    ]
[0000monkey]
[####monkey]
[many monke]
```

Here is another example specifically for numbers.

```
$n =  4395;
$u = -4395;
$c = 65; // ASCII 65 is 'A'

// notice the double %%, this prints a literal '%' character
printf("%%b = '%b'<br>", $n); // binary representation
printf("%%c = '%c'<br>", $c); // print the ascii character, same as chr()
function
printf("%%d = '%d'<br>", $n); // standard integer representation
printf("%%e = '%e'<br>", $n); // scientific notation
printf("%%u = '%u'<br>", $n); // unsigned integer representation of a
positive integer
printf("%%u = '%u'<br>", $u); // unsigned integer representation of a
negative integer
printf("%%f = '%f'<br>", $n); // floating point representation
printf("%%o = '%o'<br>", $n); // octal representation
printf("%%s = '%s'<br>", $n); // string representation
printf("%%x = '%x'<br>", $n); // hexadecimal representation (lower-case)
printf("%%X = '%X'<br>", $n); // hexadecimal representation (upper-case)

printf("%%+d = '%+d'<br>", $n); // sign specifier on a positive integer
printf("%%+d = '%+d'<br>", $u); // sign specifier on a negative integer
```

The code outputs

```
%b = '1000100101011'
%c = 'A'
%d = '4395'
%e = '4.395000e+3'
%u = '4395'
%u = '4294962901'
%f = '4395.000000'
%o = '10453'
%s = '4395'
%x = '112b'
%X = '112B'
%+d = '+4395'
%+d = '-4395'
```

in_array()

bool in_array (mixed $v , array $arr)

Check if the value $v exists in the array $arr. in_array()'s search is case-sensitive.

file_exists()

bool file_exists (string $filename)

Checks whether a file or directory exists

htmlentities()

string htmlentities (string $string)

Convert all applicable characters to HTML entities. This function is useful when you need to display HTML's special characters like '<', '>' as is. For example, when outputting '<b>' from PHP, the web browser would interpret '<b>' as being a tag to make text bold. However, in the case that you really want to print out '<b>' as is, you should use

```
htmlentities('<b>')
```

This will output `<b>` which renders as '<b>' on the screen.

4 File Manipulation

Reading and writing to a file is an essential part in any system. Although a MySQL database can replace files in most cases, it is still essential to know how to read and write file in PHP.

In this very brief introduction, we will use only four functions to read and write files. Specifically, we use **fopen()** to open a file for mainly for writing. The writing is done with **fwrite()**. After we finish writing (presumably each line represents one record), the file is closed with **fclose()**. Data in the file is read back with **file()** function, which automatically reads all the content into an array. Each element of the array corresponds to a line in the file, with the newline still attached.

Here is a general coding pattern of how an array can be written to a file, with each element on one line.

```
$records = array(
      array('John', 24),
      array('Steve', 25)
);
// 'w' places the file pointer at the beginning of the file (replace).
// 'a' places the file pointer at the end of the file (append).
$f = fopen('testfile.txt', 'w');
foreach($records as $rec){
      $line = implode(',', $rec) . "\n";
      fwrite($f, $line); //write: name,age\n
}
fclose($f);
```

In the code, $records is the array we want to write to the file. The file 'testfile.txt' is opened with **fopen()** with the mode 'w'. The mode 'w' is used to write to a new file (or replace an existing one). **fopen()** then returns a file pointer $f which we can use with **fwrite()** for an actual writing. Note that we have to append "\n" for a newline to be written. At the end, we close the file pointer with **fclose()**. After the code is executed, the file 'testfile.txt' will be created with the content shown as follows.

```
John,24
Steve,25
```

To read the content back to an array, the following code can be used.

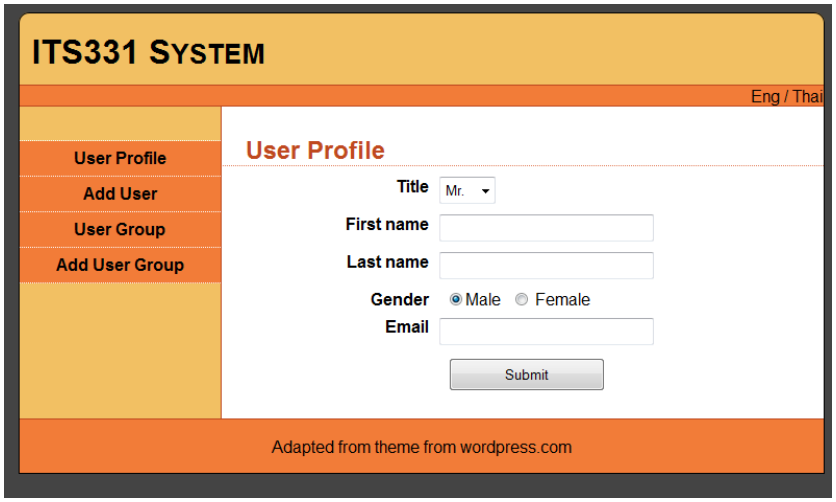
```php
$lines = file('testfile.txt');
foreach($lines as $line){
	list($name, $age) = explode(',', $line);
	// $age contains newline at the end.
	// Remove it.
	$age = trim($age);
	$records[] = array($name, $age);
}
```

As you can see, writing to a file in this way requires **implode()**, and reading the data back requires **explode()**. Make sure you understand them along with **fopen()**, **fwrite()**, **fclose()**, and **file()**.

Worksheet 1

Extend add_user1.html in the previous lab so that it allows the user to select the language (Thai and English) to display.

Here is a screenshot when "Eng" is selected



Here is a screenshot when "Thai" is selected

Partially completed files are already provided.

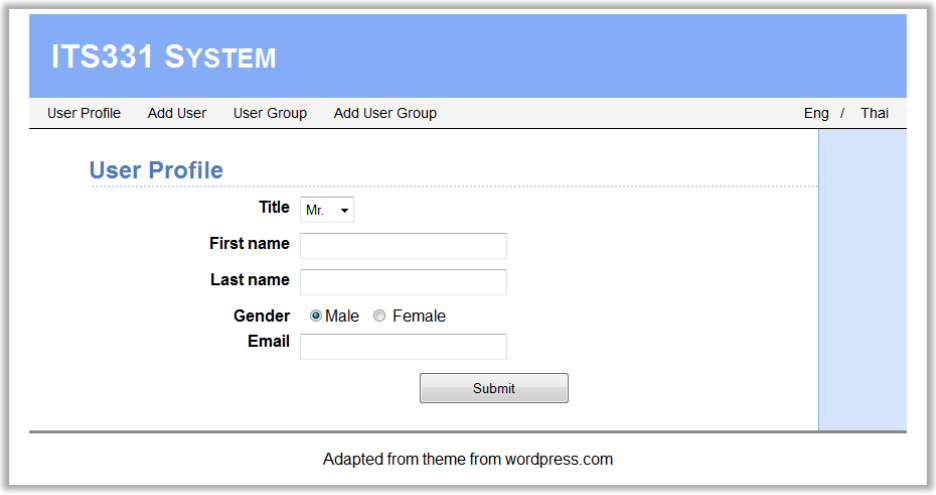
- add_user1.php – The main page. Create an object of either ITS331Eng or ITS331Thai (depending on the user's choice). Call an appropriate method in the place where a string needs to be replaced. For example, at the heading, instead of "ITS331 System", the value of the method text_heading() should be echoed out.
- bilingual.inc – PHP file containing definitions of classes (ITS331, ITS331Eng, and ITS331Thai classes). Complete the ITS331Eng and ITS331Thai classes. All methods of ITS331Eng and ITS331Thai simply return strings to output to add_user1.php.
- default1.css – CSS file for add_user1.php. No need to modify.

Translating the page by putting all translated values in a class like this eases an addition of new languages. If a new language needs to be supported, we simply create a subclass of ITS331 class. Since all subclasses share the same method names, we do not need to change the way we call methods on the created object. The only thing needed is an object of the right class for the selected language when add_user1.php is loaded.

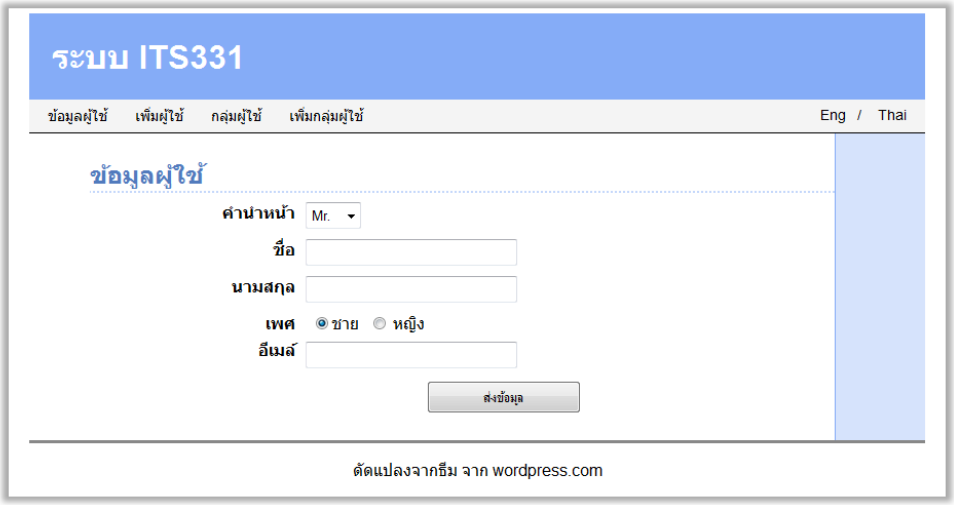
Worksheet 2

Extend add_user2.html in the previous lab so that it allows the user to select the language (Thai and English) to display.

Here is a screenshot when "Eng" is selected



Here is a screenshot when "Thai" is selected



Partially completed files are already provided.

- add_user2.php – The main page. Create an object of either ITS331Eng or ITS331Thai (depending on the user's choice). Call an appropriate method in the place where a string needs to be replaced. For example, at the heading, instead of "ITS331 System", the value of the method text_heading() should be echoed out.
- bilingual.inc – PHP file containing definitions of classes (ITS331, ITS331Eng, and ITS331Thai classes). Complete the ITS331Eng and ITS331Thai classes. All methods of ITS331Eng and ITS331Thai simply return strings to output to add_user2.php.
- default2.css – CSS file for add_user2.php. No need to modify.

Translating the page by putting all translated values in a class like this eases an addition of new languages. If a new language needs to be supported, we simply create a subclass of ITS331 class. Since all subclasses share the same method names, we do not need to change the way we call methods on the created object. The only thing needed is an object of the right class for the selected language when add_user2.php is loaded.

Worksheet 3

Extend add_user3.html in the previous lab so that it allows the user to select the language (Thai and English) to display.

Here is a screenshot when "Eng" is selected

Here is a screenshot when "Thai" is selected

Partially completed files are already provided.

- add_user3.php – The main page. Create an object of either ITS331Eng or ITS331Thai (depending on the user's choice). Call an appropriate method in the place where a string needs to be replaced. For example, at the heading, instead of "ITS331 System", the value of the method text_heading() should be echoed out.
- bilingual.inc – PHP file containing definitions of classes (ITS331, ITS331Eng, and ITS331Thai classes). Complete the ITS331Eng and ITS331Thai classes. All methods of ITS331Eng and ITS331Thai simply return strings to output to add_user3.php.
- default3.css – CSS file for add_user3.php. No need to modify.

Translating the page by putting all translated values in a class like this eases an addition of new languages. If a new language needs to be supported, we simply create a subclass of ITS331 class. Since all subclasses share the same method names, we do not need to change the way we call methods on the created object. The only thing needed is an object of the right class for the selected language when add_user3.php is loaded.

