Relational Databases and Web Integration

Week 15

c.j.pulley@hud.ac.uk

PHP Script Structure

- PHP commands are embedded in standard HTML.

```
<p> This is plain HTML.
<?php
      //Print something out
      echo “<br /> This is generated by PHP.”;
?>
<br /> And this plain old HTML again!
</p>
```

PHP Script Structure

- PHP statements are marked with start tag "<?php" and end tag "?>".
- Single line comments indicated by "//".
  - PHP ignores comments.
  - Multi-line comments enclosed between "/*" and "*/".
- Statements are terminated by a semi-colon ";".

What The Browser Sees:

- The web server pre-processes the above HTML, executing the PHP statement `echo`.
  - The `echo` statement prints out the text:

```
<br /> This is generated by PHP
```

- So web browser is sent the following HTML:

```
<p> This is plain HTML.
<br /> This is generated by PHP.
<br /> And this plain old HTML again!
</p>
```

String Literals

- String literals can be enclosed in single or double quotes.
  - `echo 'single quoted string';`
  - `echo "double quoted string";`
- Can be useful:
  - `echo 'She said "Hello"';`
  - `echo "He wouldn't open it";`
- Or you can use escape sequence:
  - `echo "She said \"Hello\" ";`

Variables

- Variables names start with the dollar sign: “$”.
- Variable names are *case sensitive*.
- Variables don’t need to be declared in advance.
- Variable types are assigned on instantiation.

Variable Types

- PHP has four scalar types:
  - boolean, float, integer and string
- PHP has two compound types:
  - array
  - object

Variables - Example

- Define a variable and give it a value.

```
/*    Create a variable
            name:  $var
            type:  integer
           value: 15  */
$var = 15;
```

- Variables can change type:

```
$var = 15;                    //type integer
$var = “fifteen”;  //type string
```

Pitfalls Of Automatic

- Automatic variable declaration allows very flexible coding, but can lead to problems.
- eg. following is fine, but nothing is printed out!

```
//Print a welcome message

$welcomeMsg = “Hello ”.$userName;

echo $welcomemsg;
```

Constants

- Constants are simple, scalar values.

```
//Define a constant:  max
define(“max”, 100);
```

- Once defined, you can use a constant anywhere in the script.

Arithmetic Expressions

- PHP has the usual assignment and arithmetic operators.

```
//Assign the value 4 + 7 to $var
$var = 4 + 7;
```

- eg. temperature conversion.

```
$F = 84; //Fahrenheit
$C = 5*(($F-32)/9); //Celsius
```

Arithmetic Expressions and Operators

- Also has increment operators.

```
//All add one to $var
$var = $var + 1;
$var += 1;
$var++;
```

- Similarly for other arithmetic operators (see the PHP manual).

String Operators

- Can concatenate (join together) strings.

```
$name = “Fred”;

echo “Hello ”. $name;
```

- Outputs the following:

```
Hello Fred
```

Type Conversion

- PHP will automatically convert between types if the code requires.
  - eg. output an integer as part of a string.

```
$var = 15;
echo “The number is: ” . $var;
```

  - The integer variable `$var` is automatically converted to a string. The output will be:

```
The number is: 15
```

Type Conversion

- You can also explicitly *cast* a variable from one type to another.
  - eg. cast an integer to a string.

```
$var = 15;

echo “Number is: ”.(string) $var;
```

- Type conversion usually follows common sense, but see PHP Manual for details.

Checking Variable Type

- Because variables can change type in PHP, it's useful to test their current type before using them.

```
boolean is_int(mixed variable)

boolean is_float(mixed variable)

boolean is_bool(mixed variable)

boolean is_string(mixed variable)
```

- Return `true` if `variable` is the correct type.

Basic Debugging

- To print the type, and contents, of a variable you can use the functions.

```
print_r(mixed expression)
```

```
var_dump(mixed expression
                [, mixed expression …])
```

- Print out a readable description of *any* type of variable, including arrays and objects.

If … else Statement

- Follows fairly standard format:

```
if ($var > 15)
{
        echo “Greater than 15”;
}
else
{
        echo “Not greater than 15”;
}
```

`If ... else` Statement

- There is no “end if”.
- The `else` part is optional.
- The braces are only really needed to enclose multiple statements. For example, this is OK:

```
if ($var > 15)
      echo “Greater than 15”;
else
      echo “Not greater than 15”;
```

If … elseif … Statement

- Consecutive conditions can be tested:

```
if ($var > 15)
        echo “Greater than 15”;
elseif ($var == 15)
        echo “Equal to 15”;
else
        echo “Less than 15”;
```

Multiple Conditions

- The condition may have several parts, joined by Boolean operators.
  - "Or" is written: ||
  - "And" is written: &&

```
if ($var >= 15) && ($var <= 20)
        echo "Between 15 and 20";
```

- See PHP manual for details of conditional expressions.

Loops

- PHP has four loop statements:
  - `while`
  - `do … while`
  - `for`
  - `foreach`
- `foreach` introduced in PHP4 to make it easier to iterate through an array (later lecture …).

`while` Loops

- Loops through a block of statements until the "end-loop" condition is met.
  - eg. list the integers from 1 to 10.

```
$counter = 1;
while ($counter < 11) {
      echo $counter . " ";
      $counter++;  //add 1 to value of $counter
}
```

  - Prints out: `1 2 3 4 5 6 7 8 9 10`

`while` and `do … while` Loops

- The end-loop condition is evaluated *first*, so if it is not met, the loop body is never executed.
- The `do … while` loop executes the loop body *before* testing the end-loop condition.
  - See Text book OR PHP online manual for details.

`for` Loops

- Loops through a block of statements a fixed number of times.
  - eg. list the integers from 1 to 10.

```
for($counter=1; $counter<11; $counter++)
{
        echo $counter . “ ”;
}
```

- Same result as the `while` loop above, but more compact.

The `break` Statement

- You can stop a loop *before* the end-loop condition is met using the `break` statement.
  - eg. stop listing the integers at 6.

```
for($count=1, $stop=6; $count<11; $count++)
{     if ($count==$stop)
            break;
      echo $count . " ";
}
```

- Also illustrates multiple initialization statements.

The `continue` Statement

- You can also skip to the next iteration using the `continue` statement.
  - eg. list the integers from 1 to 10, except 6.

```
for($count=1, $skip=6; $count<11; $count++)
{
        if ($count==$skip)
            continue;
        echo $count . " ";
}
```

Formatting Strings With `printf()`

- The `echo` statement provides a basic method for outputting text.
- To provide more sophisticated formatting, use the `printf` function:

```
integer printf(string format
                    [, mixed args …])
```

- `printf()` sends its output direct to the PHP output buffer used to build the HTTP response.

`printf()` Example

- Accept a floating point number `$var` and print out its value.

```
printf(“Result: %f”, $var);
```

- The `%f` symbol is a *conversion specification*. It tells `printf` to expect an argument of type float as the second parameter (our `$var`) and to output the value of this parameter as part of the string.

`printf()` Conversion Specifications

- As well as `%f` for float we have:
  - `%b` for boolean.
  - `%c` for single character.
  - `%d` for integer.
  - `%s` for string.
- We can also include a *width specifier*.
  - `%5.2f` means display the number with five digits before and two after the decimal point.

printf() Example

- Now print out $var to two decimal places.

```
$var = 3.14159;
printf(“Result: %5.2f”, $var);
```

- The actual output will be:

```
Result:     3.14
```

- There are an additional 4 spaces (for clarity, indicated by the ~ character) before the 3!

```
Result: ~~~~3.14
```

Comparing Strings
`strcmp()`

- Using the equality operator "== " will not always correctly compare two strings.
- Use the function `strcmp()` to do this.

```
integer strcmp(string str1,
                        string str2)
```

- There is also a `strncmp()` to compare the first *n* characters of the strings - see the PHP online manual!

`strcmp()` Example

- `strcmp()` returns and integer:
  - 0 if the strings are the same;
  - -1 if str1 precedes str2 alphabetically;
  - and +1 if str2 precedes str1 alphabetically.
- For example

```
strcmp(“apple”, “pear”); //-1
strcmp(“apple”, “apple”); //0
strcmp(“pear”, “apple”); //+1
```

Other Functions To Manipulate Strings

- PHP has lots of string manipulation functions. There are functions to:
  - Pad strings; change case; trim white space;
  - Find one string in another
  - …and lots more. See the PHP manual for details!

Passing Parameters To PHP Scripts

- There are three ways of passing a parameter to a PHP script.
  - Type the URL manually, and add the parameter and its value to the end.
  - Embed a hyperlink in your HTML that includes the parameter and its value.
  - Use a HTML form to capture the parameter's value.

Passing Parameters With The URL

- Consider this URL.

  `http://localhost/example1.php?user="Fred"`

  - Specifies a HTTP request,
  - For a file on the local host called `example.php`
    - The '`?`' indicates the start of the parameter list.
- When PHP sees '`user= "Fred"`', it automatically adds an element `$user` with value "`Fred`" to the associative array `$_GET` (see later lectures).

Using Parameter Inside The PHP Script

- Consider this fragment of a script file called with URL:

```
http://localhost/example1.php?user="Fred"
```

```
<body>
<p>
  <?php   echo "Hello ". $_GET[$user] . "\n"; ?>
</p>
```

- To use the value of a URL parameter, we just need to know its name.

Passing Multiple Parameters

- Consider this URL.

```
http://localhost/example2.php?user="Fred"&age="36"
```

```
<body>
<p>
<?php
        echo "Hello ".$_GET[$user]."\n";
        echo "You are ".$_GET[$age]." years old";
?>
</p>
```

Embedded Hyperlinks

- The same basic technique can be used to generate as hyperlink with a parameter list added to the URL.
- Simply use the `printf` or `echo` statements to generate the URL.

Class structure in PHP 5

```
class Classname {
   var $attribute1;

   function __construct($param) {
      echo "Constructor";
   } // end of constructor function

   function __destruct() {
   } // end of destructor function

   function operation1() {
   } // end of method operation1
} // end of class Classname
```

Instantiating classes

```
class Classname {
   function __construct($p) {
      echo "Constructor $p\n";
   } // end of constructor
} // end of class Classname

$a = new Classname('First');
$b = new Classname('Second');
```

Using class attributes and operations

```
class Classname {
    var $foo;

    function bar($p) {
        $this->foo = $p;
        echo $this->foo."\n";
    } // end of method bar

} // end of class Classname

$b = new Classname();
$b->bar('wow');
echo $b->foo."\n";
```

Implementing inheritance

```
class B extends A
{
   var $attr2;

   function op2()
   {
   } // end of method op2

} // end of class B

$b = new B();
$b->op1();  // defined in A
```

Overriding

- Can override operations in parent
- No way to access original in PHP 4
- PHP 5 provides access with `parent` keyword:

```
parent::op1();
```

- PHP 5 adds `final` keyword
- Operations declared as final cannot be overridden

```
final function op2() {
}
```

Multiple Inheritance

- PHP does **not** allow multiple inheritance
- Can define interfaces in PHP 5
  - Similar to interfaces in Java

```
interface Display {
  function display();
} // end of interface Display
class DisplayImpl implements Display {
  function display() {
    ..
  } // end of method display
} // end of class DisplayImpl
```

New features in PHP 5

- `__get` and `__set`
- `public`, `protected`, `private`
- Per-class constants
- Static methods
- `instanceof` and type hinting
- Cloning and `__clone`
- Abstract classes
- `__toString` and other special functions

__get and __set

```
class Classname {
   var $att1;
   function __get($name) {
      return $this->$name;
   } // end of getter
   function __set($name, $value) {
      $this->$name = $value;
   } // end of setter
} // end of class Classname

$a = new Classname();
$a->att1 = 5;
```

instanceof and type hinting

```
($b instanceof B)
```

True if $b is an object of class B

```
function check_hint(B $obj)
{
    // …
}

check_hint($a);
```

Error if $a is not an object of class B

PHP 5 vs. PHP 4

Two notes

- Objects are passed by **reference** in PHP 5
- Passed by **value** in PHP 4
- Pass by reference is more efficient
- PHP 4 has "difficulty dereferencing objects that were returned from functions"
- eg. `select_object()->display();`
- This problem is fixed in PHP 5.

Example

- Recall the *Generalized Hello World* web application from week 13
  - can add new messages into database
  - select a greeting for display
- Let's provide a *mock-up* implementation in PHP for this application.

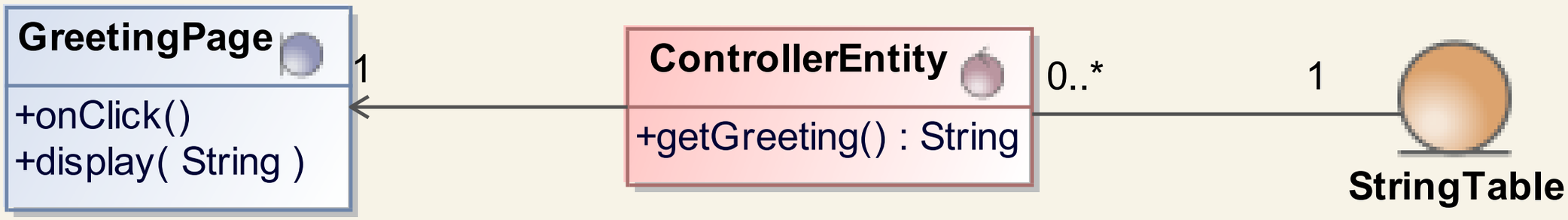
Mock-up Pages

- Used to *prototype* the user interface
  - so, not necessary to provide all implementation details
- Mock-up is intended to provide a client with an idea as to the look and feel of an end application
  - so, not necessary to implement all controller entities in MVC model

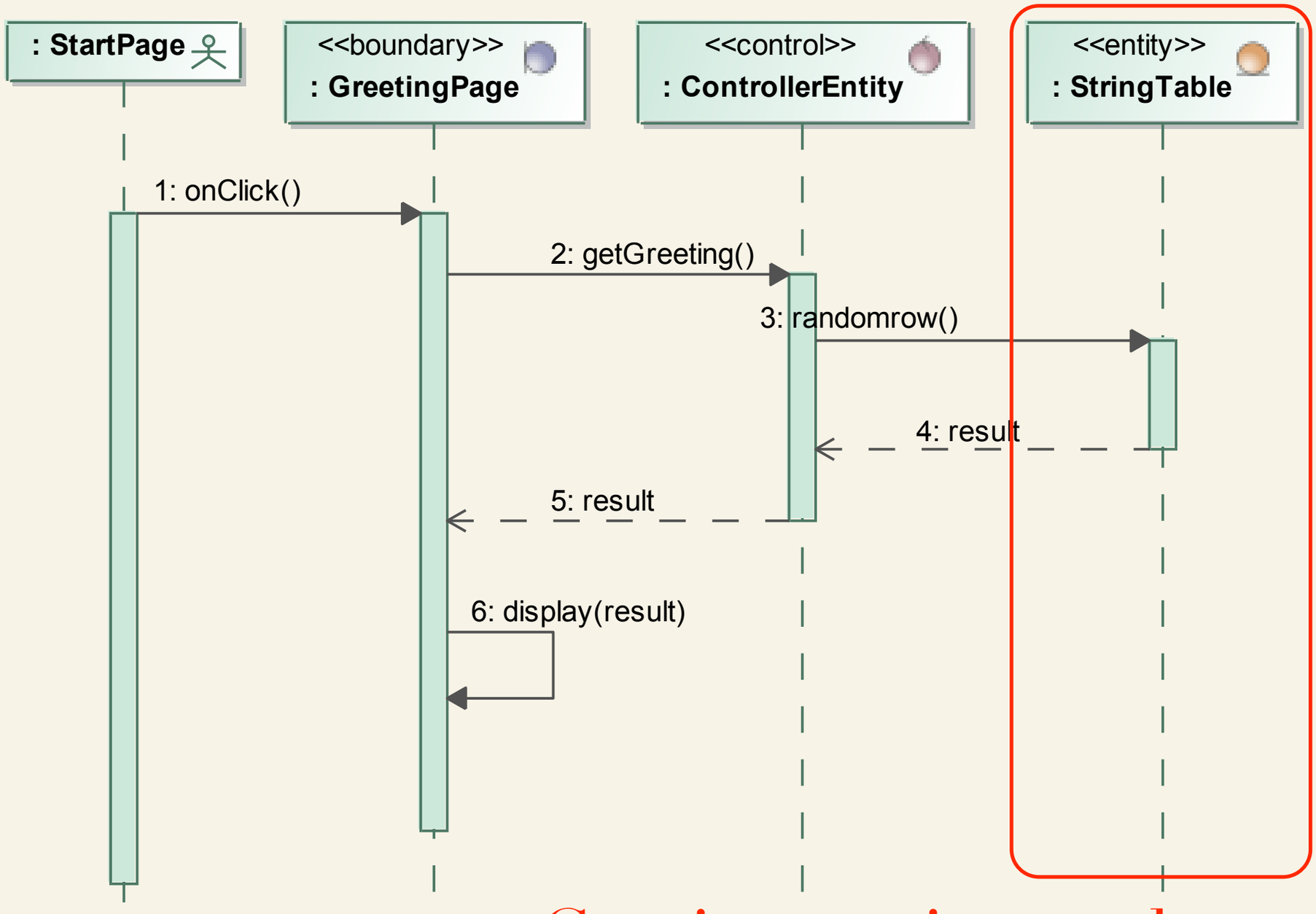
Example Continued

- With the mock-up, we can *ignore* database interactions (ie. our MVC entity elements)
  - database interactions can be *simulated* by **hard-coding** in database values
- We'll just focus on the boundary elements of our MVC UML model and implement sufficient detail in our controller elements

Example Continued



Example Continued



Can ignore in mock-up

Example Continued

```
class GreetingPage {
  public function onClick() {
    $message = ControllerEntity::getGreeting();
    display($message);
  } // end of method onClick

  private function display($message) {
?>
<html>
  <head><title>Greeting Application: GreetingPage</title></head>
  <body>
    <h1><?php echo “    $message\n”; ?></h1>
  </body>
</html>
<?php
  } // end of method display
} // end of class GreetingPage
```


This class solution assumes that only one instance of GreetingPage will be called at any one time. Since our server may have many simultaneous incoming page requests, this assumption will be invalid.

Example Revised

```
class GreetingPage {
  var $controller;

  public function __construct($controller) {
    $this->controller = $controller;
  } // end of constructor function

  public function onClick() {
    $message = $controller->getGreeting();
    display($message);
  } // end of method onClick

  private function display($message) {
    ..
  } // end of method display
} // end of class GreetingPage
```

Example Continued

```
class ControllerEntity {
  public function getGreeting() {
    // Fake a database interaction
    $result = “Hello World!”;
    return $result;
  } // end of method getGreeting
} // end of class ControllerEntity
```

Example Continued

```
<html>
  <head><title>Greeting Application: StartPage</title></head>
  <body>
    <a href=”GreetingPage.phtml?action=onClick”></a>
  </body>
</html>
```

```
if ($_REQUEST[‘action’] == “onClick”) {
  GreetingPage::onClick();
} // end of if-then
```

Part of GreetingPage.phtml

Again, we’re incorrectly assuming that only one incoming page request is handled by the server at any one time.
Moreover, we’re comparing string values incorrectly!!

Example Revised

```
<html>
  <head><title>Greeting Application: StartPage</title></head>
  <body>
    <a href=”GreetingPage.phtml?action=onClick”></a>
  </body>
</html>
```

```
if (strcmp($_REQUEST[‘action’], “onClick”)) {
  $controller = new ControllerEntity();
  $boundary = new GreetingPage($controller);
  $boundary->onClick();
} // end of if-then
```

Part of GreetingPage.phtml

We could use $_GET here. However, for simplicity and generality we’ve used $_REQUEST. This means that both GET and POST requests can be used to pass the **onClick** string value as part of the **action** parameter name.