

COMP284 Scripting Languages

Lecture 13: PHP (Part 5)

Handouts (8 on 1)

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Classes

Defining and Instantiating a Class

A Closer Look at Class Definitions

- The pseudo-variable `$this` is available when a method is called from within an object context and is a reference to the calling object
- Inside method definitions, `$this` can be used to refer to the properties and methods of the calling object
- The **object operator** `->` is used to access methods and properties of the calling object

```
class Rectangle {
    protected $height;
    protected $width;

    function __construct($height,$width) {
        $this->width = $width;
        $this->height = $height;
    }
}
```

COMP284 Scripting Languages

Lecture 13

Slide L13 - 4

Contents

- 1 Classes
 - Defining and Instantiating a Class
 - Visibility
 - Class Constants
 - Static Properties and Methods
 - Destructors
 - Inheritance
 - Interfaces
 - Introspection Functions
- 2 The PDO Class
 - Introduction
 - Connections
 - Queries and Processing of Results
 - Prepared Statements

COMP284 Scripting Languages

Lecture 13

Slide L13 - 2

Classes

Visibility

Visibility

- Properties and methods can be declared as
 - public** accessible everywhere
 - private** accessible only within the same class
 - protected** accessible only within the class itself and by inheriting and parent classes

- For **properties**, a **visibility** declaration is required
- For **methods**, a **visibility** declaration is optional
 - ~ by default, **methods** are **public**
 - Accessing a **private** or **protected** property / method outside its visibility is a **fatal error**

```
class Vis {
    public $public = 1;
    private $private = 2;
    protected $protected = 3;
    protected function proFc() {}
    private function priFc() {}
}

echo $v->public; # prints 1
echo $v->private; # Fatal Error
echo $v->protected; # Fatal Error
echo $v->priFc(); # Fatal Error
echo $v->proFc(); # Fatal Error
```

COMP284 Scripting Languages

Lecture 13

Slide L13 - 5

COMP284 Scripting Languages

Lecture 13

Slide L13 - 5

Defining and Instantiating a Class

- PHP is an object-oriented language with **classes**
- A **class** can be defined as follows:

```
class identifier {
    property_definitions
    function_definitions
}
```

 - The **class name** *identifier* is **case-sensitive**
 - The body of a class consists of **property definitions** and **function definitions**
 - The function definitions may include the definition of a **constructor**
- An **object** of a class is created using

```
new identifier(arg1,arg2,...)
```

where `arg1,arg2,...` is a possibly empty list of arguments passed to the constructor of the class *identifier*

COMP284 Scripting Languages

Lecture 13

Slide L13 - 2

Classes

Defining and Instantiating a Class

Constants

- Classes can have their own **constants** and **constants** can be declared to be **public**, **private** or **protected**
 - ~ by default, **class constants** are **public**

```
vis const identifier = value;
```

- Accessing a **private** or **protected** constant outside its visibility is a **fatal error** ~ execution of the script stops
- Class constants are allocated once per class, and not for each class instance
- Class constants are accessed using the **scope resolution operator** `::`:

```
class MyClass {
    const SIZE = 10;
}

echo MyClass::SIZE; # prints 10
$o = new MyClass();
echo $o::SIZE; # prints 10
```

COMP284 Scripting Languages

Lecture 13

Slide L13 - 6

Classes

Defining and Instantiating a Class

A Closer Look at Class Definitions

In more detail, the definition of a **class** typically looks as follows

```
class identifier {
    # Properties
    vis $attrib1
    ...
    vis $attribN = value

    # Constructor
    function __construct(p1,...) {
        statements
    }

    # Methods
    vis function method1(p1,...) {
        statements
    }
    vis function methodN(p1,...) {
        statements
    }
}
```

- Every instance obj of this class will have **attributes** `attrib1,...` and **methods** `method1(),...` accessible as `obj->attrib1` and `obj->method1(a1,...)`
- **__construct** is the **constructor** of the class and will be called whenever **new identifier(a1,...)** is executed
- **vis** is a declaration of the **visibility** of each attribute and method

COMP284 Scripting Languages

Lecture 13

Slide L13 - 3

Classes

Static Properties and Methods

Static Properties and Methods

- **Class properties** or **methods** can be declared **static**
- Static class properties and methods are accessed (via the class) using the **scope resolution operator** `::`
- Static class **properties** cannot be accessed via an instantiated class object, but **static class methods** can
- Static class **method** have no access to `$this`

```
class Employee {
    static $totalNumber = 0;
    public $name;

    function __construct($name) {
        $this->$name = $name;
        Employee::$totalNumber++;
    }
}

$e1 = new Employee("Ada");
$e2 = new Employee("Ben");
echo Employee::$totalNumber # prints 2
```

COMP284 Scripting Languages

Lecture 13

Slide L13 - 7

Classes	Destructors
Destructors	
<ul style="list-style-type: none">A class can have a destructor method <code>__destruct</code> that will be called as soon as there are no other references to a particular object	
<pre>class Employee { static \$totalNumber = 0; public \$name; function __construct(\$name) { \$this->name = \$name; Employee::\$totalNumber++; } function __destruct() { Employee::\$totalNumber--; } } \$e1 = new Employee("Ada"); \$e2 = new Employee("Ben"); echo Employee::\$totalNumber # prints 2 \$e1 = null; echo Employee::\$totalNumber # prints 1</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 8

Classes	Inheritance
Inheritance	
<ul style="list-style-type: none">In a class definition it is possible to specify one parent class from which a class inherits constants, properties and methods: <pre>class identifier1 extends identifier2 { ... }</pre>The constructor of the parent class is not automatically called it must be called explicitly from the child classInherited constants, properties and methods can be overridden by redeclaring them with the same name defined in the parent classThe declaration final can be used to prevent a method from being overriddenUsing parent:: it is possible to access overridden methods or static properties of the parent classUsing self:: it is possible to access static properties and methods of the current class	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 9

Classes	Inheritance
Inheritance: Example	
<pre>class Rectangle { protected \$height; protected \$width; function __construct(\$height,\$width) { \$this->width = \$width; \$this->height = \$height; } function area() { return \$this->width * \$this->height; } } class Square extends Rectangle { function __construct(\$size) { parent::__construct(\$size,\$size); } } \$r1 = new Rectangle(3,4); echo "\\$r1 area = ",\$r1->area(),"\n"; \$s1 = new Square(5); echo "\\$s1 area = ",\$s1->area(),"\n"; \$r1 area = 12 \$s1 area = 15</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 10

Classes	Interfaces
Interfaces	
<ul style="list-style-type: none">Interfaces specify which methods a class must implement without providing an implementationInterfaces are defined in the same way as a class with the keyword class replaced by interfaceAll methods in an interface must be declared publicA class can declare that it implements one or more interfaces using the implements keyword	
<pre>interface Shape { public function area(); } class Rectangle implements Shape { ... }</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 11

Classes	Introspection Functions
Introspection Functions	
<p>There are functions for inspecting objects and classes:</p>	
<pre>bool class_exists(string class) returns TRUE iff a class class exists class_exists('Rectangle') # returns TRUE string get_class(object obj) returns the name of the class to which an object belongs get_class(\$sq1) # returns 'Square' bool is_a(object obj, string class) returns TRUE iff obj is an instance of class named class is_a(\$sq1,'Rectangle') # returns TRUE bool method_exists(object obj,string method) returns TRUE iff obj has a method named method method_exists(\$sq1,'area') # returns TRUE</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 12

Classes	Introspection Functions
Introspection Functions	
<p>There are functions for inspecting objects and classes:</p>	
<pre>bool property_exists(object obj,string property) returns TRUE iff object has a property named property property_exists(\$sq1,'size') # returns FALSE get_object_vars(object) returns an array with the accessible non-static properties of object mapped to their values get_object_vars(\$e2) # returns ["name" => "Ben"] get_class_methods(class) returns an array of method names defined for class get_class_methods(Square) # returns ["__construct", "area"]</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 13

The PDO Class	Introduction
The PDO Class	
<ul style="list-style-type: none">The PHP Data Objects (PDO) extension defines an interface for accessing databases in PHPVarious PDO drivers implement that interface for specific database management systems<ul style="list-style-type: none">PDO_MYSQL implements the PDO interface for MySQL 3.x to 5.xPDO_SQLSRV implements the PDO interface for MS SQL Server and SQL Azure	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 14

The PDO Class	Connections
Connections	
<ul style="list-style-type: none">Before we can interact with a DBMS we need to establish a connection to itA connection is established by creating an instance of the PDO classThe constructor for the PDO class accepts arguments that specify the database source (DSN), username, password and additional options <pre>\$pdo = new PDO(dsn,username,password,options);</pre>Upon successful connection to the database, the constructor returns an instance of the PDO classThe connection remains active for the lifetime of that PDO objectAssigning NULL to the variable storing the PDO object destroys it and closes the connection <pre>\$pdo = NULL</pre>	
COMP284 Scripting Languages	Lecture 13 Slide L13 – 15

<div> <div>The PDO Class</div> <div>Connections</div> <div>Connections: Example</div> <div> <pre># Connection information for the Departmental MySQL Server \$host = "mysql"; \$user = "ullrich"; \$password = "-----"; \$db = "ullrich"; \$charset = "utf8mb4"; \$dsn = "mysql:host=\$host;dbname=\$db;charset=\$charset"; # Useful options \$opt = array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC, PDO::ATTR_EMULATE_PREPARES => false); try { \$pdo = new PDO(\$dsn,\$user,\$password,\$opt); } catch (PDOException \$e) { echo 'Connection failed: ', \$e->getMessage(); } }</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 16</div> </div> </div>	<div> <div>The PDO Class</div> <div>Queries and Processing of Results</div> <div>Processing Result Sets</div> <div> <ul style="list-style-type: none"> Using <code>bindColumn()</code> we can bind a variable a particular column in the result set from a query <ul style="list-style-type: none"> columns can be specified by number (starting with 1!) columns can be specified by name (matching case) Each call to <code>fetch()</code> and <code>fetchAll()</code> will then update all the variables that are bound to columns The binding needs to be renewed after each query execution </div> <div> <pre>\$result->bindColumn(1, \$slot); # bind by column no \$result->bindColumn(2, \$name); \$result->bindColumn('email', \$email); # bind by column name while (\$row = \$result->fetch(PDO::FETCH_BOUND)) { echo "Slot: ", \$slot, "
\n"; echo "Name: ", \$name, "
\n"; echo "Email: ", \$email, "

\n"; }</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 20</div> </div> </div>
<div> <div>The PDO Class</div> <div>Queries and Processing of Results</div> <div>Queries</div> <div> <ul style="list-style-type: none"> The <code>query()</code> method of PDO objects can be used to execute an SQL query <pre>\$result = \$pdo->query(\$statement) \$result = \$pdo->query("SELECT * FROM meetings")</pre> <code>query()</code> returns the result set (if any) of the SQL query as a PDOStatement object The <code>exec()</code> method of PDO objects executes an SQL statement, returning the number of rows affected by the statement <pre>\$rowNum = \$pdo->exec(\$statement) \$rowNum = \$pdo->exec("DELETE * FROM meetings")</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 17</div> </div> </div>	<div> <div>The PDO Class</div> <div>Prepared Statements</div> <div>Prepared Statements</div> <div> <ul style="list-style-type: none"> The use of parameterised prepared statements is preferable over queries Prepared statements are parsed, analysed, compiled and optimised only once Prepared statements can be executed repeatedly with different arguments Arguments to prepared statements do not need to be quoted and binding of parameters to arguments will automatically prevent SQL injection PDO can emulate prepared statements for a DBMS that does not support them <ul style="list-style-type: none"> MySQL supports prepared statements natively, so PDO emulation should be turned off <pre>\$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, FALSE);</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 21</div> </div> </div>
<div> <div>The PDO Class</div> <div>Queries and Processing of Results</div> <div>Processing Result Sets</div> <div> <ul style="list-style-type: none"> To get a single row as an array from a result set stored in a PDOStatement object, we can use the <code>fetch()</code> method By default, PDO returns each row as an array indexed by the column name and 0-indexed column position in the row <pre>\$row = \$result->fetch() array('slot' => 1, 'name' => 'Michael North', 'email' => 'M.North@student.liverpool.ac.uk', 0 => 1, 1 => 'Michael North', 2 => 'M.North@student.liverpool.ac.uk')</pre> After the last call of <code>fetch()</code> the result set should be released using <pre>\$rows = \$result->closeCursor()</pre> The get all rows as an array of arrays from a result set stored in a PDOStatement object, we can use the <code>fetchAll()</code> method <pre>\$rows = \$result->fetchAll()</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 18</div> </div> </div>	<div> <div>The PDO Class</div> <div>Prepared Statements</div> <div>Prepared Statements: SQL Templates</div> <div> <ul style="list-style-type: none"> An SQL template is an SQL query (as a string) possibly containing either <ul style="list-style-type: none"> named parameters of the form :name, where name is a PHP identifier, or question marks <code>?</code> for which values will be substituted when the query is executed <pre>\$tpl1 = "select slot from meetings where name=:name and email=:email"; \$tpl2 = "select slot from meetings where name=?";</pre> The PDO method <code>prepare()</code> turns an SQL template into prepared statement (by asking the DBMS to do so) <ul style="list-style-type: none"> on success, a PDOStatement object is returned on failure, FALSE or an error will be returned <pre>\$stmt1 = \$pdo->prepare(\$tpl1); \$stmt2 = \$pdo->prepare("select * from fruit where col=?");</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 22</div> </div> </div>
<div> <div>The PDO Class</div> <div>Queries and Processing of Results</div> <div>Processing Result Sets</div> <div> <ul style="list-style-type: none"> We can use a while-loop together with the <code>fetch()</code> method to iterate over all rows in a result set <pre>while (\$row = \$result->fetch()) { echo "Slot: ", \$row["slot"], "
\n"; echo "Name: ", \$row["name"], "
\n"; echo "Email: ", \$row["email"], "

\n"; }</pre> Alternatively, we can use a foreach-loop <pre>foreach(\$result as \$row) { echo "Slot: ", \$row["slot"], "
\n"; echo "Name: ", \$row["name"], "
\n"; echo "Email: ", \$row["email"], "

\n"; }</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 19</div> </div> </div>	<div> <div>The PDO Class</div> <div>Prepared Statements</div> <div>Prepared Statements: Binding</div> <div> <ul style="list-style-type: none"> We can bind the parameters of a PDOStatement object to a value using the <code>bindValue()</code> method <ul style="list-style-type: none"> Named parameters are bound by name Question mark parameters are bound by position (starting from 1!) the datatype of the value can optionally be declared (to match that of the corresponding database field) the value is bound to the parameter at the time <code>bindValue()</code> is executed </div> <div> <pre>\$stmt1->bindValue(':name', 'Ben', PDO::PARAM_STR); \$email = 'bj10liv.ac.uk'; \$stmt1->bindValue(':email', \$email); \$stmt2->bindValue(1, 20, PDO::PARAM_INT);</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 23</div> </div> </div>

<div> <div>The PDO Class</div> <div>Prepared Statements</div> <div>Prepared Statements: Binding</div> <div> <ul style="list-style-type: none"> We can bind the parameters of a PDOStatement object to a variable using the bindParam() method <ul style="list-style-type: none"> Named parameters are bound by name Question mark parameters are bound by position (starting from 1!) the datatype of the value can optionally be declared (to match that of the corresponding database field) the variable is bound to the parameter as a reference a value is only substituted when the statement is executed <pre>\$name = 'Ben'; \$stmt1->bindParam(':name',\$name,PDO::PARAM_STR); \$stmt1->bindParam(':email',\$email); \$email = 'bj1@liv.ac.uk'; \$slot = 20; \$stmt2->bindParam(1,\$slot,PDO::PARAM_INT);</pre> <ul style="list-style-type: none"> It is possible to mix bindParam() and bindValue() </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 24</div> </div> </div>	<div> <div>The PDO Class</div> <div>Transactions</div> <div>Transactions: Example</div> <pre>\$pdo = new PDO('mysql:host=...;dbname=...','...','...', array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION, PDO::ATTR_EMULATE_PREPARES => false)); \$pdo->beginTransaction(); try{ \$userId = 1; \$paymentAmount = 10.50; //Query 1: Attempt to insert a payment record \$sql = "INSERT INTO payments (user_id, amount) VALUES (?, ?)"; \$stmt = \$pdo->prepare(\$sql); \$stmt->execute(array(\$userId,\$paymentAmount)); //Query 2: Attempt to update the user's account \$sql = "UPDATE accounts SET balance = balance + ? WHERE id = ?"; \$stmt = \$pdo->prepare(\$sql); \$stmt->execute(array(\$paymentAmount,\$userId)); // Commit the transaction \$pdo->commit(); }catch(Exception \$e){ echo \$e->getMessage(); //Rollback the transaction \$pdo->rollBack(); }</pre> <p>Based on http://thisintertastme.com/php-pdo-transaction-example/</p> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 28</div> </div>
<div> <div>The PDO Class</div> <div>Prepared Statements</div> <div>Prepared Statements: Execution</div> <ul style="list-style-type: none"> Prepared statements are executed using execute() method Parameters must <ul style="list-style-type: none"> previously have been bound using bindValue() or bindParam(), or be given as an array of values to execute <ul style="list-style-type: none"> take precedence over previous bindings are bound using bindValue() execute() returns TRUE on success or FALSE on failure On success, the PDOStatement object stores a result set (if appropriate) <pre>\$stmt1->execute(); \$stmt1->execute(array(':name'=>'Eve', ':email' => \$email)); \$stmt2->execute(array(10));</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 25</div> </div>	<div> <div>The PDO Class</div> <div>Transactions</div> <div>Revision</div> <p>Read</p> <ul style="list-style-type: none"> Language Reference: Classes and Objects http://php.net/manual/en/language.oop5.php The PDO Class http://php.net/manual/en/class.pdo.php <p>of M. Achour, F. Betz, A. Dovgal, et al: PHP Manual. The PHP Group, 2017. http://uk.php.net/manual/en [accessed 07 Dec 2017]</p> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 29</div> </div>
<div> <div>The PDO Class</div> <div>Transactions</div> <div>Transactions</div> <ul style="list-style-type: none"> There are often situations where a single 'unit of work' requires sequence of database operations <ul style="list-style-type: none"> e.g., bookings, transfers By default, PDO runs in "auto-commit" mode <ul style="list-style-type: none"> successfully executed SQL statements cannot be 'undone' To execute a sequence of SQL statements whose changes are <ul style="list-style-type: none"> only committed at the end once all have been successful or rolled back otherwise, PDO provides the methods <ul style="list-style-type: none"> beginTransaction() commit() rollBack() </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 26</div> </div>	
<div> <div>The PDO Class</div> <div>Transactions</div> <div>Transactions</div> <p>To support transactions, PDO provides the methods</p> <pre>beginTransaction() – turns off auto-commit mode; changes to the database are not committed until commit() is called – returns TRUE on success or FALSE on failure – throws an exception if another transaction is already active</pre> <pre>commit() – changes to the database are made permanent; auto-commit mode is turned on – returns TRUE on success or FALSE on failure – throws an exception if no transaction is active</pre> <pre>rollBack() – discard changes to the database; auto-commit mode is restored – returns TRUE on success or FALSE on failure – throws an exception if no transaction is active</pre> </div> <div> <div>COMP284 Scripting Languages</div> <div>Lecture 13</div> <div>Slide L13 – 27</div> </div>	

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder