COMP284 Scripting Languages

Lecture 13: PHP (Part 5)

Handouts

Department of Computer
School of Electrical Engineering, Electronics, an
University of Liverpo

Contents

1. Classes
   - Defining and Instantiating a Class
   - Visibility
   - Class Constants
   - Static Properties and Methods
   - De
   - Inh
   - Int
   - Introspection Functions
2. The PDO Class
   - Introduction
   - Connections
   - Queries and Processing of Results
   - Prepared Statements
   - Transactions

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 c
  - The body of a class consists of property definitions and function definitions
  - The function definitions may include the definition

- 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*

A Closer Look at Class Definitions

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

```
class identifier {
  # Properties
  vis $attrib1
  ...
  vis $a

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

- Every instance obj of this class will have attributes *attrib1*,... and methods

  ible

  1 and

  1 ...)

  ,...)

  is executed
- *vis* is a declaration of the visibility of each attribute and method

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 ob
  callin

```
class Rectan
  protected $height;
  protected $width;

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

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 pro
  declar

- 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

```
 protected $protected = 3;
 protected fu
 private  functio
}
$v = new Vis();
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
```

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;
```

- Acces
  error

- Class c
  and not for each class instance
- Class constants are accessed using the scope resol ::

```
class MyClass {
  const SIZE = 10;
}
echo MyClass::SIZE;   # prints 10
$o = new MyClass();
echo $o::SIZE;        # prints 10
```

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
- Static c [illegible]

```
class Employ
  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
```

Destructors

- A class can have a destructor method `__destruct` that will be called as soon as there are no other references to a particular object

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

  functi
    $th
    Emp
  }
  function __destruct() {
    Employee::$totalNumber--;
  }
}
$e1 = new Employee("Ada");
$e2 = new Employee("Ben");
echo Employee::$totalNumber  # prints 2
$e1 = null;
echo Employee::$totalNumber  # prints 1
```

Inheritance

- In a class definition it is possible to specify one parent class from which a class inherits constants, properties and methods:

```
class identifier1 extends identifier2 { ... }
```

- The constructor of the parent class is not automatically called it must be calle
- Inheri redecl

- The declaration final can be used to prevent overriden
- Using parent:: it is possible to access overrid properties of the parent class

- Using self:: it is possible to access static properties and methods of the current class

Inheritance: Example

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

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

class Square extends Rectangle {
  function __construct($size) {
    parent::__construct($size,$size);
} }

$rt1 = new Rectangle(3,4);
echo "\$rt1 area = ",$rt1->area(),"\n";
$sq1 = new Square(5);
echo "\$sq1 area = ",$sq1->area(),"\n";
```

```
$rt1 area = 12
$sq1 area = 15
```

Interfaces

- Interfaces specify which methods a class must implement without providing an implementation
- Interfaces are defined in the same way as a class with the keyword `class` replaced by `interface`

- All met
- A class c
  `impl`

```
interface Shape {
  public function area();
}
class Rectangle implements Shape {
  ...
}
```

Introspection Functions

There are functions for inspecting objects and classes:

```
bool class_exists(string class)
```

returns TRUE iff a class *class* exists

```
class_exists('Rectangle')      # returns TRUE
```

```
strin
```

returns t

```
get_c
```

```
bool
```

returns TRUE iff *obj* is an instance of class name

```
is_a($sq1,'Rectangle')         # re
```

```
bool method_exists(object obj,st
```

returns TRUE iff *obj* has a method named *method*

```
method_exists($sq1,'area')     # returns TRUE
```

Introspection Functions

There are functions for inspecting objects and classes:

```
bool property_exists(object obj,string property)
```

returns TRUE iff object has a property named property

```
property_exists($sq1,'size') # returns FALSE
```

```
get_o
```

returns bject

mappe

```
get_o
# returns ["name" => "Ben"]
```

```
get_class_methods(class)
```

returns an array of method names defined for

```
get_class_methods('Square')
# returns ["__construct", "area"]
```

The PDO Class

- The PHP Data Objects (PDO) extension defines an interface for access
- Vario datab

  - PDO_MYSQL implements the PDO interface for M
  - PDO_SQLSRV implements the PDO interface for Azure

Connections

- Before we can interact with a DBMS we need to establish a connection to it
- A connection is established by creating an instance of the PDO class

- The constructor for the PDO class accepts arguments that specify the datab

```
$pdo = new PDO
```

- Upon s
  instance of the PDO class
- The connection remains active for the lifetime of th

- Assigning NULL to the variable storing the PD
  closes the connection

```
$pdo = NULL
```

Connections: Example

```
# Connection information for the Departmental MySQL Server
$host    = "mysql";
$user    = "ullrich";
$passwd  = "--------";
$db      = "ullrich";
$charset = "utf8mb4";
$dsn     = "mysql:host=$

# 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,$user,$passwd,$opt);
} catch (PDOException $e) {
 echo 'Connection failed: ',$e->getMessage();
}
```

Queries

- The query() method of PDO objects can be used to execute an SQL query

```
$result = $pdo->query(statement)
$result = $pdo->query("SELECT␣*␣FROM␣meetings")
```

- quer
  PDO

- The e
  returning the number of rows affected by the statem

```
$rowNum = $pdo->exec(statement)
$rowNum = $pdo->exec("DELETE␣*␣FRO
```

Processing Result Sets

- To get a single row as an array from a result set stored in a PDOStatement object, we can use the fetch() method
- By default, PDO returns each row as an array indexed by the column name and 0-indexed column position in the row

```
$row = $result->fetch()
array('
[illegible]
     1 => 'Michael North',
     2 => 'M.North@student.liverpool.ac.uk')
```

- After the last call of fetch() the result se [illegible]

```
$rows = $result->closeCursor()
```

- The get all rows as an array of arrays from a result set stored in a PDOStatement object, we can use the fetchAll() method

```
$rows = $result->fetchAll()
```

Processing Result Sets

- We can use a while-loop together with the fetch() method to iterate over all rows in a result set

```
while ($row = $result->fetch()) {
  echo "Slot:  ",$row["slot"], "<br>\n";
  echo "Name:  ",$row["name"], "<br>\n";
  ech  "
}
```

- Altern

```
foreach($result as $row) {
  echo "Slot:  ",$row["slot"], "<br>\n";
  echo "Name:  ",$row["name"], "<br>\n";
  echo "Email: ",$row["email"],"<br><br>\n";
}
```

Processing Result Sets

- Using bindColumn() we can bind a variable a particular column in the result set from a query
  - columns can be specified by number (starting with 1!)

  - columns can be specified by name (matching case)
- Each c ables that ar

- The bin

```
$result->bindColumn(1, $slot);
$result->bindColumn(2, $name);
$result->bindColumn(’email’, $email);
while ($row = $result->fetch(PDO::FETCH_BOUND)) {
  echo "Slot:  ",$slot, "<br>\n";
  echo "Name:  ",$name, "<br>\n";
  echo "Email: ",$email,"<br><br>\n";
}
```

Prepared Statements

- The use of parameterised prepared statements is preferable over queries
- Prepared statements are are parsed, analysed, compiled and optimised only once
- Prepared statements can be executed repeatedly with different argu
- Argu bindin injection
- PDO can emulate prepared statements for a DBM support them
- MySQL supports prepared statements nativel should be turned off

```
$pdo->setAttribute(PDO::ATTR_EMULATE_PREPARES, FALSE);
```

Prepared Statements: SQL Templates

- An SQL template is an SQL query (as a string) possibly containing either
  - named parameters of the form :*name* where *name* is a PHP identifier, or
  - question marks ?

  for whi

```
$tpl1 = "se

$tpl2 = "se
```

- The PDO method prepare() turns an SQ statement (by asking the DBMS to do so)
  - on success, a PDOStatement object is returned
  - on failure, FALSE or an error will be returned

```
$stmt1 = $pdo->prepare($tpl1);
$stmt2 = $pdo->prepare("select * from fruit where col=?");
```

Prepared Statements: Binding

- We can bind the parameters of a PDOStatement object to a value using the bindValue() method
  - Named parameters are bound by name

  - Question mark parameters are bound by position (starting from 1!)
  - the d [illegible] (to m [illegible]
  - the v [illegible] is executed

```
$stmt1->bindValue(':name','Ben',PDO::PARAM_STR);
$email = 'bj1@liv.ac.uk';
$stmt1->bindValue(':email',$email);
$stmt2->bindValue(1,20,PDO::PARAM_INT);
```

Prepared Statements: Binding

- We can bind the parameters of a PDOStatement object to a variable using the bindParam() method
  - Named parameters are bound by name
  - Question mark parameters are bound by position (starting from 1!)
  - the d
    (to m
  - the v
  - a val

```
$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);
```

- It is possible to mix bindParam() and bindValue()

Prepared Statements: Execution

- Prepared statements are executed using execute() method
- Parameters must
  - previously have been bound using bindValue() or bindParam(), or
  - be given as an array of values to execute
    - ⇝ t
    - ⇝ a
- exec
- On success, the PDOStatement object stores a result set (if appropriate)

```
$stmt1->execute();
$stmt1->execute(array(':name' => 'Eve', ':email' => $email));
$stmt2->execute(array(10));
```

Transactions

- There are often situations where a single ‘unit of work’ requires a sequence of database operations

  ⤳ e.g., bookings, transfers

- By default, PDO runs in ”auto-commit” mode

  ⤳ su

- To exe

  - only

  - rolled back otherwise,

  PDO provides the methods

  - `beginTransaction()`
  - `commit()`
  - `rollBack()`

Transactions

To support transactions, PDO provides the methods

`beginTransaction()`

- turns off auto-commit mode; changes to the database are not committed until `commit()` is called
- returns
- throws a

`commi`

- change
  auto-commit mode is turned on
- returns `TRUE` on success or `FALSE` on fa
- throws an exception if no transaction is active

`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

Transactions: Example

```
$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 = "INSER  I
  $stmt = $pdo-
  $stmt->e

  //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();
}
```

Based on `http://thisinterestsme.com/php-pdo-transaction-example/`

Revision

Read

- Language Reference: Classes and Objects
  `http://php.net/manual/en/language.oop5.php`
- The PDO Class
  `http`

of M. Acho
2017. h