COMP519 Web Programming
Lecture 25: PHP (Part 7)
Handouts

Ullrich Hustadt

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

Contents

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*

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

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;
  }
}
```

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() {}
}
$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
  $\rightsquigarrow$ by default, class constants are public

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

- Accessing a private or protected constant outside its visibility is a fatal error $\rightsquigarrow$ 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
```

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
```

Destructors

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

```
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
```

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 called explicitly from the child class
- Inherited constants, properties and methods can be overridden by redeclaring them with the same name defined in the parent class
- The declaration `final` can be used to prevent a method from being overriden
- Using `parent::` it is possible to access overridden methods or static properties of the parent class
- Using `self::` it is possible to access static properties and methods of the current class

Inheritance

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

  function __construct($height,$width) {
     $this->height = $height;
     $this->width  = $width;
  }
  function area() {
     return $this->width * $this->height;
} }

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 = 25
```

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 methods in an interface must be declared public
- A class can declare that it implements one ore more interfaces using the `implements` keyword

```
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`
`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`

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_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"]
```

The PDO Class

- The PHP Data Objects (PDO) extension defines an interface for accessing databases in PHP
- Various PDO drivers implement that interface for specific database management systems
  - `PDO_MYSQL` implements the PDO interface for MySQL 3.x to 5.x
  - `PDO_SQLSRV` implements the PDO interface for MS SQL Server and SQL 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 database source (DSN), username, password and additional options

```
$pdo = new PDO(dsn,username,password,options);
```

- Upon successful connection to the database, the constructor returns an instance of the PDO class
- The connection remains active for the lifetime of that PDO object
- Assigning `NULL` to the variable storing the PDO object destroys it and closes the connection

```
$pdo = NULL
```

Connections: Example

```
# Connection information for the Departmental MySQL Server
$host     = "studdb.csc.liv.ac.uk";
$user     = "sgfsurn";  # your University username
$passwd   = "-------";  # your MySQL server account password
$db       = "sgfsurn";  # your University username
$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,$user,$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")
```

- `query()` returns the result set (if any) of the SQL query as a PDOStatement object

- The `exec()` method of PDO objects executes an SQL statement, returning the number of rows affected by the statement

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

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('slot' => 1,
      'name' => 'Michael␣North',
      'email' => 'M.North@student.liverpool.ac.uk',
      0 => 1,
      1 => 'Michael␣North',
      2 => 'M.North@student.liverpool.ac.uk')
```

- After the last call of `fetch()` the result set should be released using

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

- To 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:␣␣",$row["name"], "<br>\n";
  echo "Email:␣",$row["email"],"<br><br>\n";
}
```

- Alternatively, we can use a foreach-loop

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

- `fetch()` uses a cursor that moves through the rows in a result set and does not reset at the end

  ⇝ store the result set in an array first (e.g., using `fetchAll()`), then iterate over the array as often as you want

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 call to `fetch()` and `fetchAll()` will then update all the variables that are bound to columns
- The binding needs to be renewed after each query execution

```
$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, "<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 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
- MySQL supports prepared statements natively, so PDO emulation should be turned off

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

Prepared Statements: SQL Templates

- Using a prepared statement requires three steps
  1. Preparing a statement using a SQL template containing parameters
  2. Binding the parameters to values
  3. Executing the prepared statements

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 which values will be substituted when the query is executed

```
$tpl1 = "select slot from meetings where
         name=:name and email=:email";
$tpl2 = "select slot from meetings where name=?";
```

- The PDO method `prepare()` turns an SQL template into a prepared 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 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 `bindValue()` 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 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

```
$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`
    ⤳ 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)

```
$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
  $\leadsto$ e.g., bookings, transfers
- By default, PDO runs in ”auto-commit” mode
  $\leadsto$ successfully executed SQL statements cannot be ‘undone’
- To execute a sequence of SQL statements whose changes are
  - only committed at the end once all have been successful or
  - 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 `TRUE` on success or `FALSE` on failure
- throws an exception if another transaction is already active

`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

`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 (Part 1)

- We want to transfer £10.50 from one account at a bank to another, but only if the payer has enough money
- Either both subtracting the money from one account and adding it to another should be successful or neither

```
$balance  = array();

// storeBalance($id,$b)
// $id: account id
// $b : balance for the account with id $id
// maps $id to $b in the array $balance
function storeBalance($id,$b) {
  global $balance;
  $balance[$id] = $b;
}

$pdo = new PDO('mysql:host=...;dbname=...','...','...',
       array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
             PDO::ATTR_EMULATE_PREPARES => false));
try {
  // Details of the transaction: payer, payee, amount
  $payerId = 1;  $payeeId = 2;  $paymentAmount  = 10.50;
```

Transactions: Example (Part 2)

```
$pdo->beginTransaction();
// Obtain payee's and payer's account balances
// and lock access to both records
$sql1 = "select id,balance from accounts where id=? or id=? for update";
$stmt = $pdo->prepare($sql1);
$stmt->execute(array($payerId,$payeeId));

// Store the data retrieved from the database in $balance array
$stmt->fetchAll(PDO::FETCH_FUNC,'storeBalance');

// Check whether there is enough money in the payer's account
if ($balance[$payerId] < $paymentAmount) {
  echo "Insufficient funds in payer's account";
} else {
  $sql = "UPDATE accounts SET balance = balance + ? WHERE id = ?";
  $stmt = $pdo->prepare($sql);

  // Increase balance of payee's account by payment ammount
  $stmt->execute(array($paymentAmount,$payeeId));

  // Decrease balance of payer's account by payment ammount
  $stmt->execute(array(-$paymentAmount,$payerId));
}
```

Transactions: Example (Part 3)

```
  // Commit the transaction (whether money was transferred or not)
  $pdo->commit();

} catch (PDOException $e) {
  // Something went wrong at some point
  echo "Error: ",$e->getMessage(),"<br>\n";

  // Roll back the transaction
  $pdo->rollBack();
}
```

Revision and Further Reading

Read

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

of P. Cowburn (ed.): PHP Manual. The PHP Group, 25 Oct 2019.
`http://uk.php.net/manual/en` [accessed 26 Oct 2019]