**More Advanced**

**OOP**

**Abstract class and methods**

Abstract classes are template versions of a parent class. By defining an abstract class, you can indicate the general behaviour that subclasses should have. Put another way, an abstract class defines the interfaces: How derived classes of this base type are to be used. The subclasses are then responsible for defining the actual implementations of those interfaces.

Attempting to create an object of an abstract class’s type results in a fatal error. Abstract classes are meant to be extended, and then you create an instance of that extended class.

Abstract classes normally have abstract methods:

Abstract class ClassName {

Abstract function methodName();

Abstract function methodName($var\_1, $var\_2);

}

You can add visibility to the class methods after the *abstract* keyword, e.g. *abstract public function getName();*

Example

Abstract class Pet {

Protected $\_name;

Abstract public function getName();

}

Class Cat extends Pet {

Function getName() {

Return $this->name;

}

}

You would never make an abstract method *private* as private methods cannot be inherited.

The implemented version of the method must have the same number of arguments as the abstract definition.

\*Note that if a class has even one abstract method, the class must be an abstract class, however, an abstract class can have non-abstract members. \*

Abstract class vs interface

Abstract classes are meant to be extended by a more specific class. An interface is not inherited by a class, an interface establishes a contract for the functionality that a class must have, regardless of the class type.

**Interfaces**

Interfaces, like abstract classes, identify the functionality (i.e. the methods) that must be defined by a specific class:

Interface iSomething {

Public function someFunction($var);

}

//conventionally, interface names often begin with a lower case ‘i’.

Note that all methods in an interface must be public. Also, interfaces only identify methods; they never include attributes.

To associate a class with an interface, use the *implements* operator in the class definition:

Class SomeClass implements iSomething { }

The class must then define all of the methods listed in the interface, or a fatal error will occur.

Example

Interface iCrud {

Public function create($data);

Public function read();

Public function update($data);

Public function delete();

}

Class User implements iCrud {

Private $\_userId = NULL;

Private $\_username = NULL;

Function \_\_construct($data) {

$this->\_userId = uniqid();

$this->\_username = $data[‘username’];

}

Function create($data) {

Self::\_\_construct($data);

}

Function read() {

Return [‘userId’ => $this->\_userId, ‘username’ => $this->\_username];

}

Function update($data) {

$this->\_username = $data[‘username’];

}

Function delete() {

$this->\_userId = NULL;

$this->\_username = NULL;

}

}

$user = array(‘username’, ’rezaa91’)’

$me = new User($user);

$info = $me->read(); //array

$me->update(‘username’ => ‘ali’);

$me->delete();

Unset($me);

**Typehinting**

Typehinting is the programming act of indicating what type of valid is expected:

Public function update(array $data) { }

**Namespaces**

Namespaces provided a solution to a common OOP problem: as you begin utilising more and more classes, including those defined by other developers and in third-party libraries, conflicts can occur if multiple classes have the same name. Namespaces prevent these conflicts by letting you organise your code into groups. This has the effect of allowing you to safely use descriptive names without concern for conflicts.

There are limits to what you can place in a namespace:

* Classes
* Interfaces
* Functions
* Constants

Note that the *namespace* should be the first line of PHP code in a file.

Namespace SomeNamespace;

Class SomeClass { }

Namespaces can have sub-namespaces, just as you’d have levels of directories on your computer. To do that, indicate a sub-namespace using backslash:

Namespace MyUtilities\UserManagement;

Example

Company.php

<?php

Namespace MyNamespace\Company

Class Department {

Private $\_name;

Private $\_employees;

Function \_\_construct($name) {

$this->\_name = $name;

$this->\_employees = array();

}

Function addEmployee(Employee $e) {  
 $this->\_employees[] = $e;

Echo “<p>{$e->getName()} has been added to the {$this->\_name} department.</p>”;

}

}

Class Employee {

Private $\_name;

Function \_\_construct($name) {

$this->\_name = $name;

}

Function getName() {

Return $this->\_name;

}

}

?>

Namespace.php

<?php

Require(‘MyNamespace/Company/Company.php’);

$hr = new \MyNamespace\Company\Department(‘Accounting’);

$e1 = new \MyNamespace\Company\Employee(‘Ali Issaee’);

$e2 = new \MyNamespace\Company\Employee(‘Lauren Foster’);

//add employees to the department

$hr->addEmployee($e1);

$hr->addEmployee($e2);

Unset($hr, $e1, $e2);

?>

You can use the same namespace in multiple files, which will allow you to put multiple classes, each defined in separate scripts, within the same namespace.

PHP allows you to more quickly reference a namespace by bringing it into current scope via the *use* keyword:

Use MyNamespace\Company;

Having done that, you can now create an object by just referencing classes within the Company namespace:

$obj = new Department();