**OOP**

**ADVANCED**

**Advanced theories**

Inheritanceis far & away the most important aspect in advanced object-oriented programming.

Object inheritance is where one class is derived from another, just as humans inherit qualities from their parents.

In UML, inheritance is represented using an arrow between the 2 classes. The arrow should point to the parent:

|  |
| --- |
| **ParentClass** |
| Attr1  Attr2 |
| Method1()  Method2()  Method3() |

|  |
| --- |
| **ChildClass** |
| \*attr1  \*attr2  \*attr3 |
| \*method1()  \*method2()  \*method3()  Method4()  Method5() |

\* = inherited from the ParentClass.

As you can see, the ChildClass has its own members, separating itself from the ParentClass.

Inheritance isn’t necessarily a simple one-to-one relationship, there is no limit to how many times inheritance can occur: multiple classes can inherit from the same parent, or a class can be a child of a child.

E.g. 1

ChildClass

ParentClass

ChildClass

E.g. 2

ParentClass ChildClass GrandChildClass

But what if you wanted to change the behaviour of the ParentClass methods for one of the ChildClass? It would be wrong to change the classes of the parent as this would affect the other children, instead, you can override a parent’s class’s method to customise if for the new class.

**Inheriting classes**

|  |
| --- |
| **User** |
| Username  userID  email  password |
| Login()  Logout() |

|  |
| --- |
| **Admin** |
| \*Username  \*userID  \*email  \*password  AccessLevel |
| \*Login()  \*Logout()  editUser() |

The syntax to inherit from a class is:

Class ChildClass extends ClassName{ }

e.g. class Pet {

public $name;

function \_\_construct($pet\_name) {

$this->name = $pet\_name;

}

Function sleep() {

Echo “<p>$this->name is sleeping</p>”;

}

}//end of parent class

Class Cat extends Pet {

Function climb() {

Echo “<p>$this->name is climbing</p>”;

}

}

Class Dog extends Pet {

Function fetch() {

Echo “<p>$this->name is fetching</p>”;

}

}

//create instances

$dog = new Dog(‘Satchel’);

$cat = new Cat(‘Bucky’);

//now test out the methods!

The *instanceof* keyword can be used to see if a particular object is of a certain class type:

If($obj instanceof SomeClass) {…

**Inheriting constructors & destructors**

When you create an object of a child class, the parent’s class constructor is not automatically called.

As a rule, PHP will always call the constructor for the class just instantiated, if it has one.

E.g. using the Rectangle example from the ‘oop-basics’ doc:

//include class definition:

Require(‘Rectangle.php’);

Class Square extends Rectangle {

Function \_\_constructor($side = 0) {

$this->width = $side;

$this->height = $side;

}

}//end of square class

//rectangle dimensions

$width = 21;

$height = 98;

$r = new Rectangle($width, $height);

//square

$side = 60;

$s = new Square($side);

# now test the methods for both objects!

Unset($r, $s); //delete objects

A general OOP recommendation is that all classes have a constructor, inherited or otherwise.

When deciding where to place methods (parent or child), include constructors & destructors, you have to think about whether that functionality is universal or specific.

**Overriding methods**

To override a method in PHP, the subclass must define a method with the exact same name & number of arguments as the parent class.

E.g. (using Pet example above)

//add play method to Pet class

Function play() {

Echo “<p>$this->name is playing</p>”;

}

//now change the method names (fetching & climbing) in dog and cat instances to play(), and keep code the same

**Access control**

Access control, AKA *visibility*, dictates how accessible a class’s properties and methods are, i.e. where they can and cannot be referenced.

3 levels: public, protected, private

In the Pets example, because $name is public, you can do the following to change the name: $pet = new Pet(“Charlie”); $pet->name = “Fungo”;

It may seem odd for a class to have attributes, or members, that are inaccessible, but this is a valuable, important OOP concept, called *encapsulation*. Simply put, encapsulation is the hiding of information that does not need to be available outside of the class.

Looking at the Pet example, its $name attribute should be protected, it makes sense for the class and subclasses to be able to access the name, but you shouldn’t be able to change the name outside of the class. The same applies for the $width and $height attributes of the Rectangle class.

To indicate visibility in UML, use:

+ **public**

- **private**

# **protected**

e.g. #name:string

E.g. class Test {

Public $public = ‘public’;

Protected $protected = ‘protected’;

Private $\_private = ‘private’;

Function print\_var($var) {

Echo “<p>In test, \$$var: ‘{$this->$var}’ </p>”’

}

}

Class LittleTest extends Test {

Function print\_var($var) {

Echo “<p>In little test, \$$var: ‘{$this->$var}’.</p>”;

}

}

//end of classes

//create objects

$parent = new Test();

$child = new LittleTest();

Echo “<h1>Public</h1>”;

Echo “<h2>Initially…</h2>”;

$parent->print\_var(‘public’);

$child->print\_var(‘public’);

//also echo $parent->public or echo $child->public;

//do same for private and protected

//between each one, attempt to change value like this…

$parent->public = ‘modified’;

$parent->print\_var(‘public’);

$child->print\_var(‘public’);

Many programmers would argue that all attributes should be protected or private, so they are never directly accessible outside of the class. You would then write ‘get’ and ‘set’ methods as an interface for accessing them when needed.

**Using the scope resolution operator**

OOP has some of its own operators, e.g. ->

Another, is the scope resolution operator :: (2 colons).

It is used to access members through classes, not objects:

ClassName::methodName();

ClassName::propertyName;

There are 2 common places in which this is used:

1. Within classes, to avoid confusion when inherited classes have the same attribute and methods
2. Outside of classes, to access members without first creating objects

E.g. using the Pet example

//in Pet class

Function \_\_construct($pet\_name) {

$this->name = $pet\_name;

Self::sleep();

}

*this* is a keyword, which is a reference to the current object instance.

*self* is a keyword, which is a reference to the current class (now, this method is called as soon as an object is created).

/// in the Dog and Cat class; in function play();

Function play() {

Parent::play();

Echo “<p>$this->name is climbing/fetching</p>”;

}

*Parent* is a keyword, to refer to a member of a parent class.

//create instances

$dog = new Dog(‘Satchel’);

$cat = new Cat(‘Bucky’);

$pet = new Pet(‘Rob’);

//run the eat, sleep and play methods

#note that the sleep method runs when created

Unset($dog, $cat, $pet); //delete objects

**Creating static members**

A static function variable remembers its value each time a function is called:

Function test() {

Static $n = 1;

Echo “$n <br />”;

$n++;

}

Test(); //1

Test(); //2

Test(); //3

//as opposed to 1,1,1 if not static

To declare a static attribute, use the *static* keyword after the visibility indicator:

Class SomeClass{

Public *static* $var = “value”;

}

\*side note – class constants are accessible to all instances of that class (or derived classes), but as with any other constant, the value can never change:

Const PI = 3.14; //3.14 is the value, the value cannot be based on another variable, or be a result of an expression or function call

Constants, like static attributes, cannot be accessed through the object, i.e. $obj->PI; or $obj::PI;

However, you can access them via ClassName::constantName, e.g. SomClass::PI;

\*

You must use *self::* to access static variables in a class:

Class SomeClass {

Public static $counter = 0;

Function \_\_construct() {

Self::counter++;

}

}//creates counter for how many objects of this class exist

Static methods are created in much the same way:

Class SomeClass {

Public static function doThis(){

//code

}

}

E.g. Class Pet {

Protected $name;

Private static $\_count = 0;

//constructor increments the counter:

Function \_\_construct($pet\_name) {

$this->name = $pet\_name;

Self::$\_count++;

}

//destructor decrements the counter:

Function \_\_destruct() {

Self::$\_count--;

}

Public static function getCount() {

Return self::$\_count;

}

}//end of pet class

Class Dog extends Pet {

}

Class Cat extends Pet {

}

Class Ferret extends Pet {

}

//now create numerous instances and echo the value of the number of pets via:

Echo Pet::getCount();

//now delete some instances and run function again

If you did want to have overridden constructors and destructors in the derived classes in this example, you would need them to call the Pet constructor and destructor in order to properly manage the page count. You would do so by adding: parent::\_\_construct() and parent::\_\_destruct() to them.

Static methods are almost always public because they can’t be called through an object.

The special variable *$this*, which always refers to the current object, is not available inside a static method, because static methods are meant to be invoked without using an object.

**Quiz**

1. What is meant by inheritance?
2. Give example of inheritance using multiple children of a parent class
3. Give example of inheritance using grandparent < parent < child
4. Give example of inheriting constructors and destructors
5. What is meant by override – give example
6. What is meant by access control/visibility
7. Define the 3 levels of visibility and give example
8. What is the scope resolution operator and when would you use it
9. Use *self* and *parent* in an example, using the scope resolution operator
10. What is a *static* member – give example