Polymorphism in PHP

In object oriented programming, Polymorphism is a powerful and fundamental tool. It can be used to create a more organic flow in your application. This tutorial will describe the general concept of polymorphism, and how it can easily be deployed in PHP.

What is Polymorphism?

Polymorphism is a long word for a very simple concept.

Polymorphism describes a pattern in object oriented programming in which classes have different functionality while sharing a common interface.

The beauty of polymorphism is that the code working with the different classes does not need to know which class it is using since they're all used the same way.

A real world analogy for polymorphism is a button. Everyone knows how to use a button: you simply apply pressure to it. What a button "does," however, depends on what it is connected to and the context in which it is used -- but the result does not affect how it is used. If your boss tells you to press a button, you already have all the information needed to perform the task.

In the programming world, polymorphism is used to make applications more modular and extensible. Instead of messy conditional statements describing different courses of action, you create interchangeable objects that you select based on your needs. That is the basic goal of polymorphism.

Interfaces

An integral part of polymorphism is the common interface. There are two ways to define an interface in PHP: interfaces and abstract classes. Both have their uses, and you can mix and match them as you see fit in your class hierarchy.

Interface

An interface is similar to a class except that it cannot contain code. An interface can define method names and arguments, but not the contents of the methods. Any classes implementing an interface must implement all methods defined by the interface. A class can implement multiple interfaces.

An interface is declared using the 'interface' keyword:

|  |  |
| --- | --- |
|  | **interface MyInterface {**  **// methods**  **}** |

and is attached to a class using the 'implements' keyword (multiple interfaces can be implemented by listing them separated with commas):

Methods can be defined in the interface just like in a class, except without the body (the part between the braces):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **interface MyInterface {**  **public function doThis();**  **public function doThat();**  **public function setName($name);**  **}**  All methods defined here will need to be included in any implementing classes exactly as described. (read the code comments below)   |  |  | | --- | --- | |  | class MyClass implements MyInterface {      protected $name;      public function doThis() {          // code that does this      }      public function doThat() {          // code that does that      }      public function setName($name) {          $this->name = $name;      }  } |   First examples  Math examle:  **interface math {**  **public function calculate();**  **}**  **class Square implements math{**  **public $number;**  **public function \_\_construct($value) {**  **$this->number = $value;**  **}**  **public function calculate() {**  **echo "Square's surface area is: " . $this->number \* $this->number ."</br>";**  **}**  **}**  **class Cube implements math{**  **public $number;**  **public function \_\_construct($value) {**  **$this->number = $value;**  **}**  **public function calculate() {**  **echo "Cube's surface area is: " . $this->number \* $this->number \* 6;**  **}**  **}**  **$square = new Square(2);**  **$square->calculate();**  **$cube = new Cube(3);**  **$cube->calculate();**  Second example, Shapes:  **interface Shape{**  **public function getArea();**  **}**  **class Rect implements Shape{**  **public $width;**  **public $height;**    **public function \_\_construct($width, $height) {**  **$this->width = $width;**  **$this->height = $height;**  **}**  **public function getArea(){**  **return $this->width \* $this->height ;**  **}**  **}**  **class Circle implements Shape{**  **public $radius;**    **public function \_\_construct($radius) {**  **$this->radius = $radius;**  **}**  **public function getArea(){**  **return $this->radius \* $this->radius \* 3.14156;**  **}**  **}**  **function getPrice (Shape $shape){**  **return $shape->getArea() \* 0.25;**  **}**  **$rect = new Rect(5,7);**  **$circle = new Circle(3);**    **echo "Price for rectangular carpet in $ is: ". getPrice($rect) . "<br />";**  **echo "Price for circled carpet in $ is : " . getPrice($circle) . "<br />";**  Abstract Class  An abstract class is a mix between an interface and a class. It can define functionality as well as interface  (in the form of abstract methods). Classes extending an abstract class **must** implement all of the abstract methods  defined in the abstract class. Also you can instantiate abstract class!  Abstract class:   * May contain properties or static variables * May contain methods and static methods * May contain a constructor but it can not be called directly since abstract classes can not be instantiated. * May contain abstract methods. * Any class that contains an abstract method is itself an abstract class regardless of how it is defined.   An abstract class is declared the same way as classes with the addition of the 'abstract' keyword:   |  |  | | --- | --- | |  | **abstract class MyAbstract {**  **// methods**  **}** |   and is attached to a class using the 'extends' keyword:   |  |  | | --- | --- | |  | **class MyClass extends MyAbstract {**  **// class methods**  **}** |   Regular methods can be defined in an abstract class just like in a regular class, as well as any abstract methods  (using the 'abstract' keyword). Abstract methods behave just like methods defined in an interface, and must be  implemented exactly as defined by extending classes.   |  |  | | --- | --- | |  | **abstract class MyAbstract {**  **public $name;**  **public function doThis() {**  **// do this**  **}**  **abstract public function doThat();**  **abstract public function setName($name);**  **}** |   Example:  **abstract class test {**  **public function demo\_function(){**  **echo "This is normal function";**  **}**  **abstract public function set\_value($val);**  **abstract public function get\_value();**  **}**  **/\* Can't instantiate this class**  **\* $obj = new test;**  **$obj->demo\_function();**  **\*/**  **class demo extends test{**  **public $num;**    **public function set\_value($val) {**  **$this->num = $val;**  **}**  **public function get\_value() {**  **echo "<br> Value is: " . $this->num;**  **}**  **}**  **$obj = new demo();**  **$obj->set\_value(10);**  **$obj->demo\_function();**  **$obj->get\_value();** |

Example 2

**abstract class AbstractClass**

**{**

**// Force Extending class to define this method**

**abstract protected function getValue();**

**abstract protected function prefixValue($prefix);**

**// Common method**

**public function printOut() {**

**print $this->getValue() . "\n";**

**}**

**}**

**class ConcreteClass1 extends AbstractClass**

**{**

**protected function getValue() {**

**return "ConcreteClass1";**

**}**

**public function prefixValue($prefix) {**

**return "{$prefix}ConcreteClass1";**

**}**

**}**

**class ConcreteClass2 extends AbstractClass**

**{**

**public function getValue() {**

**return "ConcreteClass2";**

**}**

**public function prefixValue($prefix) {**

**return "{$prefix}ConcreteClass2";**

**}**

**}**

**$class1 = new ConcreteClass1;**

**$class1->printOut();**

**echo $class1->prefixValue('FOO\_') ."<br />";**

**$class2 = new ConcreteClass2;**

**$class2->printOut();**

**echo $class2->prefixValue('FOO\_') ."<br />";**