Abstract Classes

Abstract classes are declared using the ***abstract*** word and they are used to enforce a certain behavior for its sub-classes. Also, abstract classes cannot be instantiated.

abstract class Shape

{

    //adds encapsulation

    protected $color;

    public function \_\_construct($color)

    {

        $this->color = $color;

    }

}

class Square extends Shape

{

    protected $length = 4;

    public function getArea()

    {

        return pow($this->length, 2);

    }

}

class Triangle extends Shape

{

    protected $base = 4;

    protected $height = 7;

    public function getArea()

    {

        return .5 \* $this->base \* $this->height;

    }

}

class Circle extends Shape

{

}

(new Square('red'));

\*in the example above, the shapes must have a color being passed as argument when they are instantiated (this is enforced through the abstract class);

\*abstract class also adds encapsulation by declaring $color as ***protected***;

\*we can also add a default color to the abstract class (so that when its subclasses are being instantiated, if they aren’t being passed a color, the default one is being appied):

abstract class Shape

{

    //adds encapsulation

    protected $color;

    //this makes sure that 'red' is the default color for any subclasses if they don't pass in another color to overwrite the default

    public function \_\_construct($color = 'red')

    {

        $this->color = $color;

    }

}

class Square extends Shape

{

    protected $length = 4;

    public function getArea()

    {

        return pow($this->length, 2);

    }

}

Now we can instantiate the subclass without having to pass a color:

(new Square);// has color red

(new Square('blue')); //has color blue

Furthermore, we can add a method in the abstract parent class, to access the private property $color:

abstract class Shape

{

    //adds encapsulation

    protected $color;

    //this makes sure that 'red' is the default color for any subclasses if they don't pass in another color to overwrite the default

    public function \_\_construct($color = 'red')

    {

        $this->color = $color;

    }

    //shared behaviour for all the subclasses (this method allows us to access the protected color from the subclasses)

    public function getColor()

    {

        return $this->color;

    }

}

Now we can do:

echo (new Square('blue'))->getColor(); //prints blue

\*we can access a private property through a ‘getter’ method;

Abstract Methods

abstract class Shape

{

    //adds encapsulation

    protected $color;

    //this makes sure that 'red' is the default color for any subclasses if they don't pass in another color to overwrite the default

    public function \_\_construct($color = 'red')

    {

        $this->color = $color;

    }

    //shared behaviour for all the subclasses (this method allows us to access the protected color from the subclasses)

    public function getColor()

    {

        return $this->color;

    }

    //abstract method (abstract template design patter)

    abstract protected function getArea();

    //this abstract function does not require a body (it will only enforce the behaviour that all its subclasses have a getArea() method defined)

}

\*an abstract method enforces the behavior that its subclasses have the method defined;

\*so now any class that extends the abstract class (which has an abstract method) must have a method defined after the ‘blueprint’ of the abstract one (so in this case any class that extends Shape must have a getArea() method defined);

\*now if I define a circle class without the getArea() method and instantiate it, program will throw an error:

class Circle extends Shape

{

}

(new Circle);//throws an error if we don’t define getArea() in Circle class