Trait in PHP

PHP implements a method of code reuse called traits. Traits are a mechanism for code reuse in single inheritance language such as PHP.

Minimizing code duplication through better organization and code reuse is an important goal of Object Oriented Programming. But in PHP it can sometimes be difficult because of the limitations of the single inheritance model it uses; you might have some methods that you would like to use in multiple classes but they may not fit well into the inheritance hierarchy.

Languages like C++ and Python allow us to inherit from multiple classes which solves this problem to some extent, and mixins in Ruby allows us to mix the functionality of one or more classes without using inheritance. But multiple inheritance has issues such as the [Diamond Problem](http://en.wikipedia.org/wiki/Diamond_problem) problem, and mixins can be a complex mechanism to work with.

In this article I will discuss traits, a new feature introduced in PHP 5.4 to overcome such issues. The concept of traits itself is nothing new to programming and is used in other languages like Scala and Perl. They allows us to horizontally reuse code across independent classes in different class hierarchies.

**What a Trait Looks Like**

A trait is similar to an abstract class which cannot be instantiated on its own (though more often it’s compared to an interface). The PHP documentation defines traits as follows:

Traits is a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies.

Let’s consider this example:

<?php

class DbReader extends Mysqli

{

}

class FileReader extends SplFileObject

{

}

It’d be a problem if both classes needed some common functionality, for example making both of them singletons. Since PHP doesn’t support multiple inheritance, either each class will have to implement the necessary code to support the Singleton pattern or there will be an inheritance hierarchy that doesn’t make sense. Traits offer a solution to exactly this type of problem.

<?php

trait Singleton

{

private static $instance;

public static function getInstance() {

if (!(self::$instance instanceof self)) {

self::$instance = new self;

}

return self::$instance;

}

}

class DbReader extends ArrayObject

{

use Singleton;

}

class FileReader

{

use Singleton;

}

The trait Singleton has a straight forward implementation of the Singleton pattern with a static method getInstance() which creates an object of the class using this trait (if it’s not already created) and returns it.

Let’s try to create the objects of these classes using the method getInstance().

<?php

$a = DbReader::getInstance();

$b = FileReader::getInstance();

var\_dump($a); //object(DbReader)

var\_dump($b); //object(FileReader)

We can see that $a is an object of DbReader and $b is an object of FileReader, but both are now behaving as singletons. The method from Singleton has been horizontally injected to the classes using it.

Traits do not impose any additional semantics on the class. In a way, you can think of it as a compiler-assisted copy and paste mechanism where the methods of the trait is copied into the composing class.

If we were simply subclassing DbReader from a parent with a private $instance property, the property wouldn’t be shown in the dump of ReflectionClass::export(). And yet with traits, there it is!

Class [ class FileReader ] {

@@ /home/shameer/workplace/php54/index.php 19-22

- Constants [0] {

}

- Static properties [1] {

Property [ private static $\_instance ]

}

- Static methods [1] {

Method [ static public method instance ] {

@@ /home/shameer/workplace/php54/index.php 6 - 11

}

}

- Properties [0] {

}

- Methods [0] {

}

}

**Multiple Traits**

So far we have used only one trait with a class, but in some cases we may need to incorporate the functionality of more than one trait.

<?php

trait Hello

{

function sayHello() {

echo "Hello";

}

}

trait World

{

function sayWorld() {

echo "World";

}

}

class MyWorld

{

use Hello, World;

}

$world = new MyWorld();

echo $world->sayHello() . " " . $world->sayWorld(); //Hello World

Here we have two traits, Hello and World. Trait Hello is only able to say “Hello” and trait World can say “World”. In the MyWorld class we have applied Hello and World so that the MyWorld object will have methods from both traits and be able to say “Hello World”.

**Traits Composed of Traits**

As the application grows, it’s quite possible that we will have a set of traits which are used across different classes. PHP 5.4 allows us to have traits composed of other traits so that we can include only one instead of a number of traits in all these classes. This lets us rewrite the previous example as follows:

<?php

trait HelloWorld

{

use Hello, World;

}

class MyWorld

{

use HelloWorld;

}

$world = new MyWorld();

echo $world->sayHello() . " " . $world->sayWorld(); //Hello World

Here we have created the trait HelloWorld, using traits Hello and World, and included it in MyWorld. Since the HelloWorld trait has methods from the other two traits, it’s just the same as if we had including the two traits in the class ourselves.

**Precedence Order**

As I’ve already mentioned, traits work as if their methods have been copied and pasted into the classes using them and they are totally flattened into the classes’ definition. There may be methods with the same name in different traits or in the class itself. You might wonder which one will be available in the object of child class.

The precedence order is:

1. the methods of a trait override inherited methods from the parent class
2. the methods defined in the current class override methods from a trait

This is made clear in the following example:

<?php

trait Hello

{

function sayHello() {

return "Hello";

}

function sayWorld() {

return "Trait World";

}

function sayHelloWorld() {

echo $this->sayHello() . " " . $this->sayWorld();

}

function sayBaseWorld() {

echo $this->sayHello() . " " . parent::sayWorld();

}

}

class Base

{

function sayWorld(){

return "Base World";

}

}

class HelloWorld extends Base

{

use Hello;

function sayWorld() {

return "World";

}

}

$h = new HelloWorld();

$h->sayHelloWorld(); // Hello World

$h->sayBaseWorld(); // Hello Base World

We have a HelloWorld class derived from Base, and both classes have a method named sayWorld() but with different implementations. Also, we have included the trait Hello in the HelloWorld class.

We have two methods, sayHelloWorld() and sayBaseWorld(), the former of which calls sayWorld() which exists in both classes as well as in the trait. But in the output, we can see the one from the child class was invoked. If we need to reference the method from the parent class, we can do so by using the parent keyword as shown in the sayBaseWorld() method.

**Conflict Resolution and Aliasing**

When using multiple traits there may be a situation where different traits use the same method names. For example, PHP will give a fatal error if you try to run the following code because of conflicting method names:

<?php

trait Game

{

function play() {

echo "Playing a game";

}

}

trait Music

{

function play() {

echo "Playing music";

}

}

class Player

{

use Game, Music;

}

$player = new Player();

$player->play();

Such trait conflicts aren’t resolved automatically for you. Instead, you must choose which method should be used inside the composing class using the keyword insteadof.

<?php

class Player

{

use Game, Music {

Music::play insteadof Game;

}

}

$player = new Player();

$player->play(); //Playing music

Here we have chosen to use the play() method of the Music trait inside the composing class so the class Player will play music, not a game.

In the above example, one method has been chosen over the other from two traits. In some cases you may want to keep both of them, but still avoiding conflicts. It’s possible to introduce a new name for a method in a trait as an alias. An alias doesn’t rename the method, but offers an alternate name by which it can be invoked. Aliases are created using the keyword as.

<?php

class Player

{

use Game, Music {

Game::play as gamePlay;

Music::play insteadof Game;

}

}

$player = new Player();

$player->play(); //Playing music

$player->gamePlay(); //Playing a game

Now any object of class Player will have a method gamePlay(), which is the same as Game::play(). Here it should be noted that we have resolved any conflicts explicitly, even after aliasing.

**Reflection**

The Reflection API is one of the powerful features of PHP to analyze the internal structure of interfaces, classes, and methods and reverse engineer them. And since we’re talking about traits, you might be interested to know about the Reflection API’s support for traits. In PHP 5.4, four methods have been added to ReflectionClass to get information about traits in a class.

We can use ReflectionClass::getTraits() to get an array of all traits used in a class. The ReflectionClass::getTraitNames() method will simply return an array of trait names in that class. ReflectionClass::isTrait() is helpful to check if something is a trait or not.

In the previous section we discussed having aliases for traits to avoid collisions due to traits with the same name. ReflectionClass::getTraitAliases() will return an array of trait aliases mapped to its original name.

**Other Features**

Apart from the above mentioned, there are other features that makes traits more interesting. We know that in classical inheritance the private properties of a class can’t be accessed by child classes. Traits can access the private properties or methods of the composing classes, and vice versa! Here is an example:

<?php

trait Message

{

function alert() {

echo $this->message;

}

}

class Messenger

{

use Message;

private $message = "This is a message";

}

$messenger = new Messenger;

$messenger->alert(); //This is a message

As traits are completely flattened into the class composed of them, any property or method of the trait will become a part of that class and we access them just like any other class properties or methods.

We can even have abstract methods in a trait to enforce the composing class to implement these methods. For example:

<?php

trait Message

{

private $message;

function alert() {

$this->define();

echo $this->message;

}

abstract function define();

}

class Messenger

{

use Message;

function define() {

$this->message = "Custom Message";

}

}

$messenger = new Messenger;

$messenger->alert(); //Custom Message

Here we have a trait Message with an abstract method define(). It requires all classes which use this trait to implement the method. Otherwise, PHP will give an error saying there is an abstract method which has not been implemented.

Unlike traits in Scala, traits in PHP can have a constructor but it must be declared public (an error will be thrown if is private or protected). Anyway, be cautious when using constructors in traits, though, because it may lead to unintended collisions in the composing classes.

**Summary**

Traits are one of the most powerful features introduced in PHP 5.4, and I’ve discussed almost all their features in this article. They let programmers reuse code fragments horizontally across multiple classes which do not have to be within the same inheritance hierarchy. Instead of having complex semantics, they provide us with a light weight mechanism for code reuse. Though there are some drawbacks with traits, they certainly can help improve the design of your application removing code duplication and making it more DRY.