PHP Traits

aka mixins
aka interface with implementations
aka compile time copy + paste

@MrDanack

Danack@basereality.com

What are Traits?

Interfaces with implementation.
Syntax similar to classes.

```
trait Hello {
	public function sayHello() {
		echo 'Hello ';
	}
}

class HelloWorld {
	use Hello;
}

$o = new HelloWorld();
$o->sayHello();
```

Traits can have abstract methods

Same as abstract class, error if not implemented

```
trait Greet {
	abstract public function getName();

	function greet(){
		echo "Hello there ".$this->getName()."\n";
	}
}

class AnonymousUser{
	use Greet;
}
```

```
PHP Fatal error:  Class AnonymousUser contains 1 abstract method and must therefore be declared
abstract or implement the remaining methods (AnonymousUser::getName) in
/documents/projects/traits/intro/example2_requiringMethods.php on line 35
```

Traits really are* copied and pasted into each class

```
trait Counter {
    public function inc() {
        static $count = 0;
        $count = $count + 1;
        echo $count."\n";
    }

    public static $classCount = 0;

    public function incClassVar(){
        self::$classCount += 1;
        echo  self::$classCount."\n";
    }
}
```

* small implementation details aside - see bonus slide.

Traits really are* copied and pasted into each class

```
class C1 {   use Counter; }

class C2 {   use Counter; }

$a = new C1();
$a->inc(); // 1

$b = new C2();
$b->inc(); // 1

$a->incClassVar(); // 1
$b->incClassVar(); // 1
```

Trait inheritance

Almost the same as classes

```
class ParentClass {
    public static function test(){ echo "Parent class"; }
}

trait TestTrait {
     public static function test(){ echo "A trait!"; }
}

class TestClass extends ParentClass {
    use TestTrait;
}

TestClass::test(); //outputs "A trait!"
```

Trait inheritance

Almost the same as classes

```
trait TestTrait {
     public static function test(){ echo "A trait!"; }
}

class TestClass {
    use TestTrait;
    public static function test(){ echo "Child class!"; }
}

TestClass::test(); //outputs "Child class!"
```

Test Inheritance - When Traits collide

Remove trait method names to avoid collision.

```
trait A {
      public function smallTalk() { echo 'a';  }
      public function bigTalk()    { echo 'A';  }
}

trait B {
      public function smallTalk()  { echo 'b';  }
      public function bigTalk()     { echo 'B';  }
}

class Aliased_Talker {
      use A, B {
            B::smallTalk insteadof A;   // Throw away A::smallTalk
            A::bigTalk insteadof B;      // Throw away B::BigTalk
            B::bigTalk as talk;             // Alias B::bigTalk to talk
      }
}
```

New keyword - static()

Late static binding - because 'static' isn't used enough already.

```
class A {
	public static function get_self()    {         return new self();    }
	public static function get_static() {       return new static(); }
}

class B extends A { }

echo get_class(B::get_self());      // A
echo get_class(B::get_static());   // B

echo get_class(A::get_static());   // A
```

New keywords - __TRAIT__

```
trait    TestTrait{
     function  compileTimeJoy(){
          echo "Using trait [". __TRAIT__."] in class [".__CLASS__."]";
     }
}

class  TestClass{
     use TestTrait;
}

//Using trait [TestTrait] in class [TestClass]
```

Why would you use Traits?

Not that many good uses - probably more bad uses than good. The good ones are:

- Interfaces that don't need dependency injecting.
- Horizontal code duplication in classes.
- Helper functions that aren't really related to the class.

Interface that doesn't need injection

```
trait Singleton{
	private static $instance = null;
	/** This type hinting works correctly in PHPStorm 6
	 * @return static  */
	public static function getInstance(){
		if(static::$instance == null){
			$newInstance = new static();
			static::$instance = $newInstance;
		}
		return static::$instance;
	}
};
class	TestClass{
	use Singleton;
}

$testClass = TestClass::getInstance();
```

Helper class not related to hierarchy

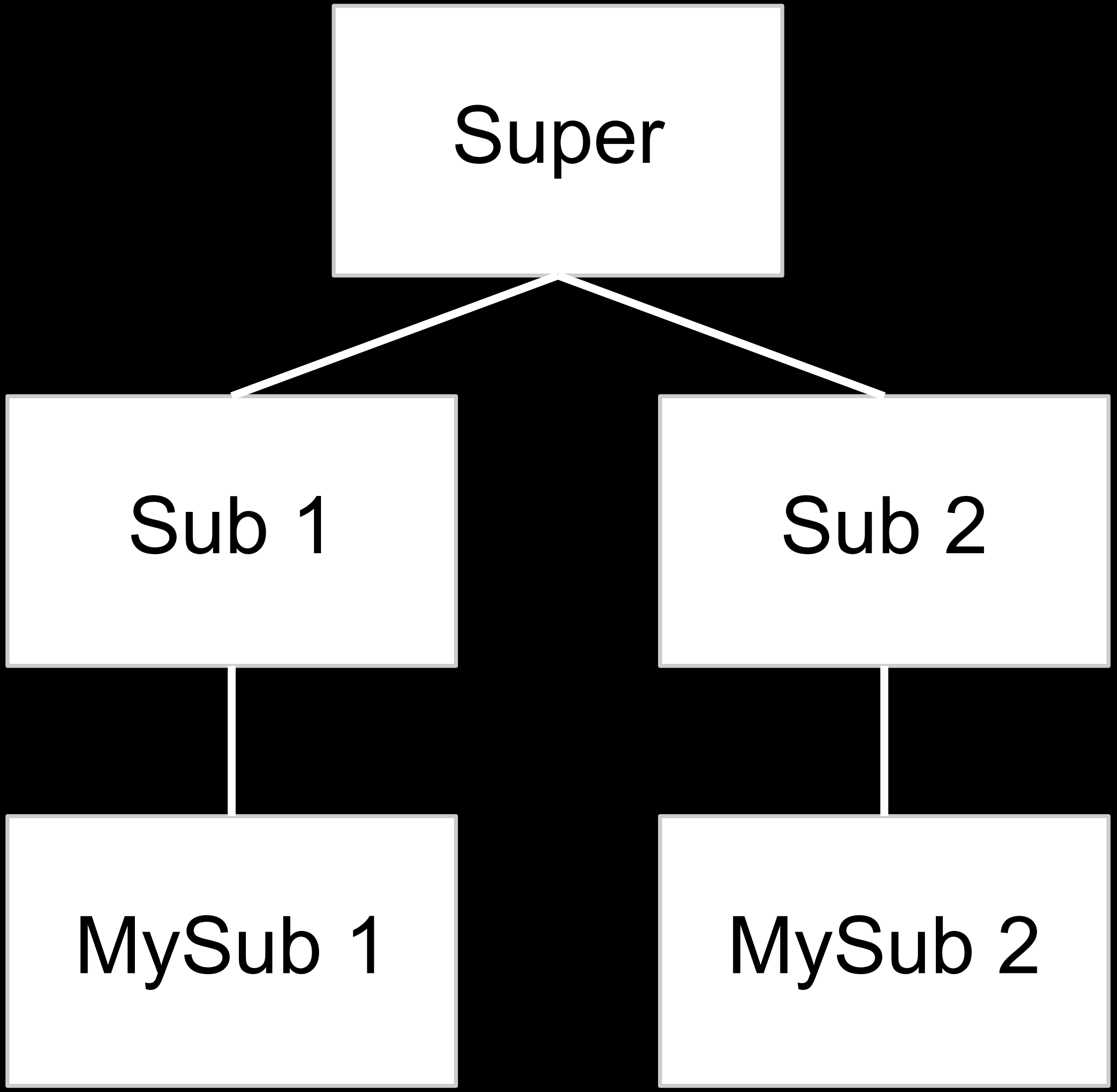
aka removing one of the worst features of PHP

```
trait SafeAccess {
	public function __set($name, $value) {
		throw new \Exception("Property [$name] doesn't exist for class
["._CLASS_."] so can't set it");
	}
	public function __get($name) {
		throw new \Exception("Property [$name] doesn't exist for class
["._CLASS_."] so can't get it");
	}
}
```

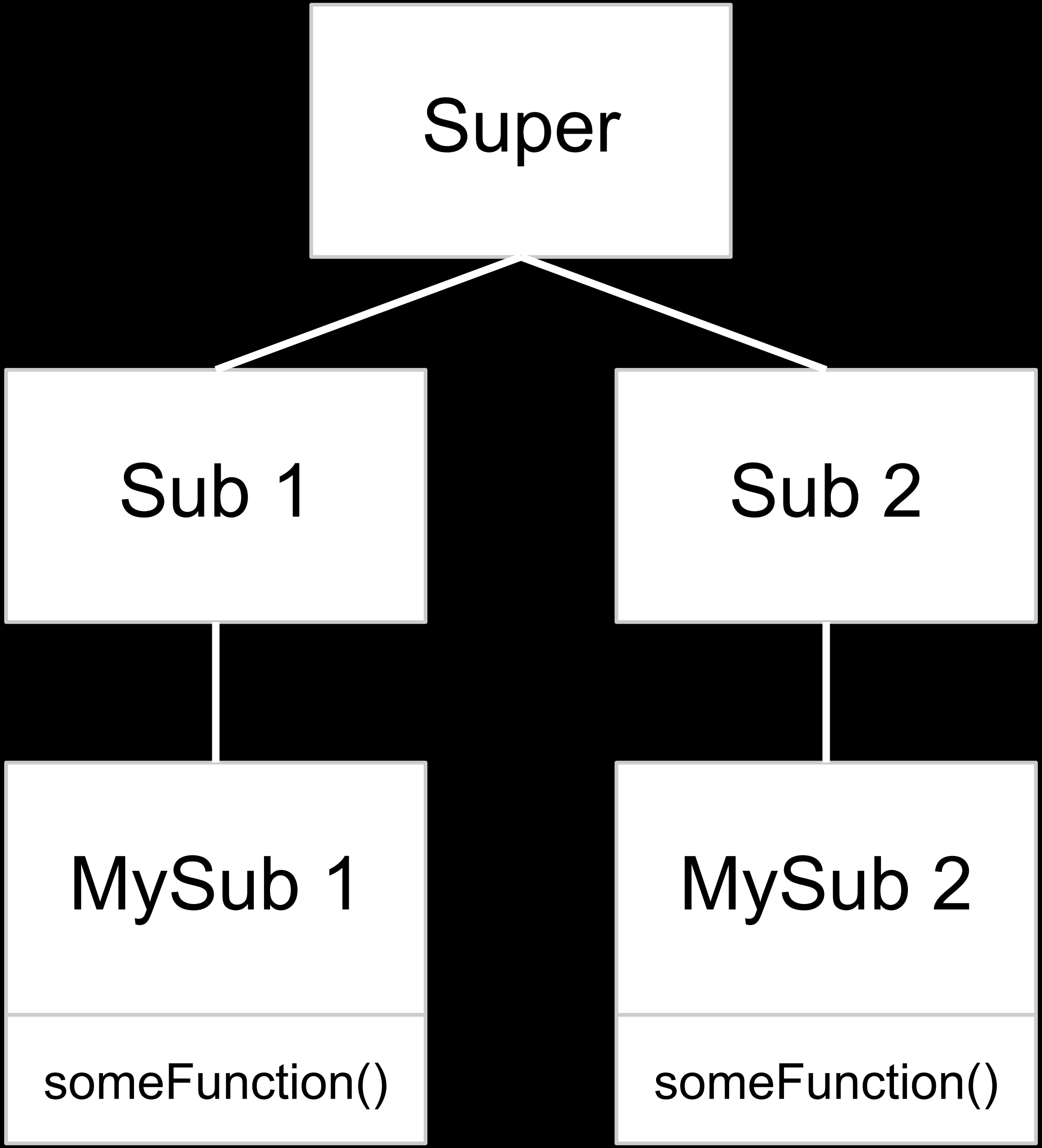
Helper class not related to hierarchy

```
class DangerClass{
	private $value;
	function  setValue($newValue){ 	$this->vallue = $newValue;	}
}

class SafeClass{
	use SafeAccess;
	private $value;
	function  setValue($newValue){ $this->vallue = $newValue;	}
}

$dangerClass = new DangerClass();
$dangerClass->setValue(5);   // Works ?!?

$safeClass = new SafeClass();
$safeClass->setValue(5);   //Exception Property [vallue] doesn't exist for class
```

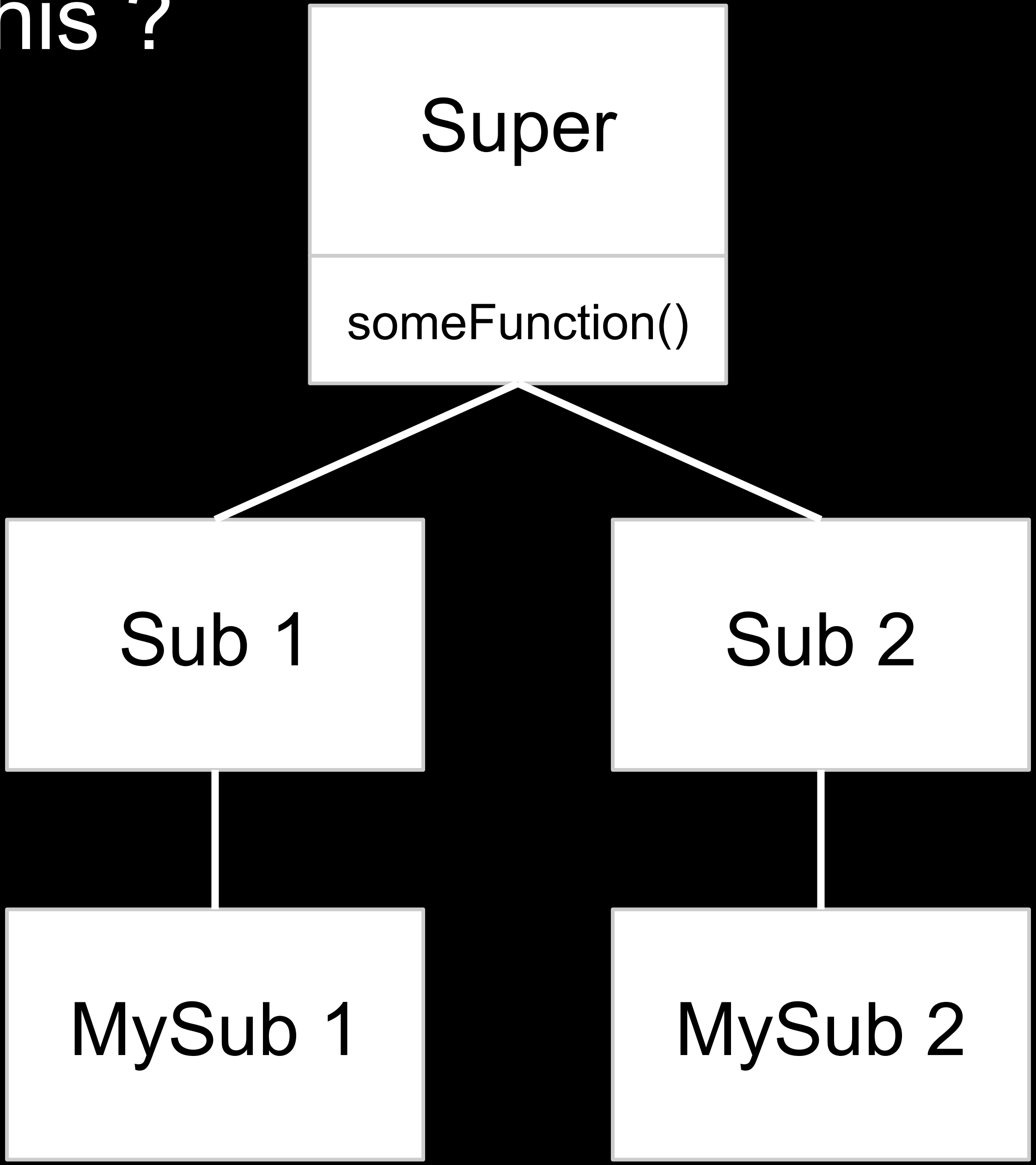
Horizontal code reuse



Two subclasses have duplicate code



Fix like this ?



How to solve?

- Extend a class
  - How deep do you want your classes to be?
  - Root class might be in a library, so not possible.
  - Bad class design - Why is a root class aware of sub-classes?

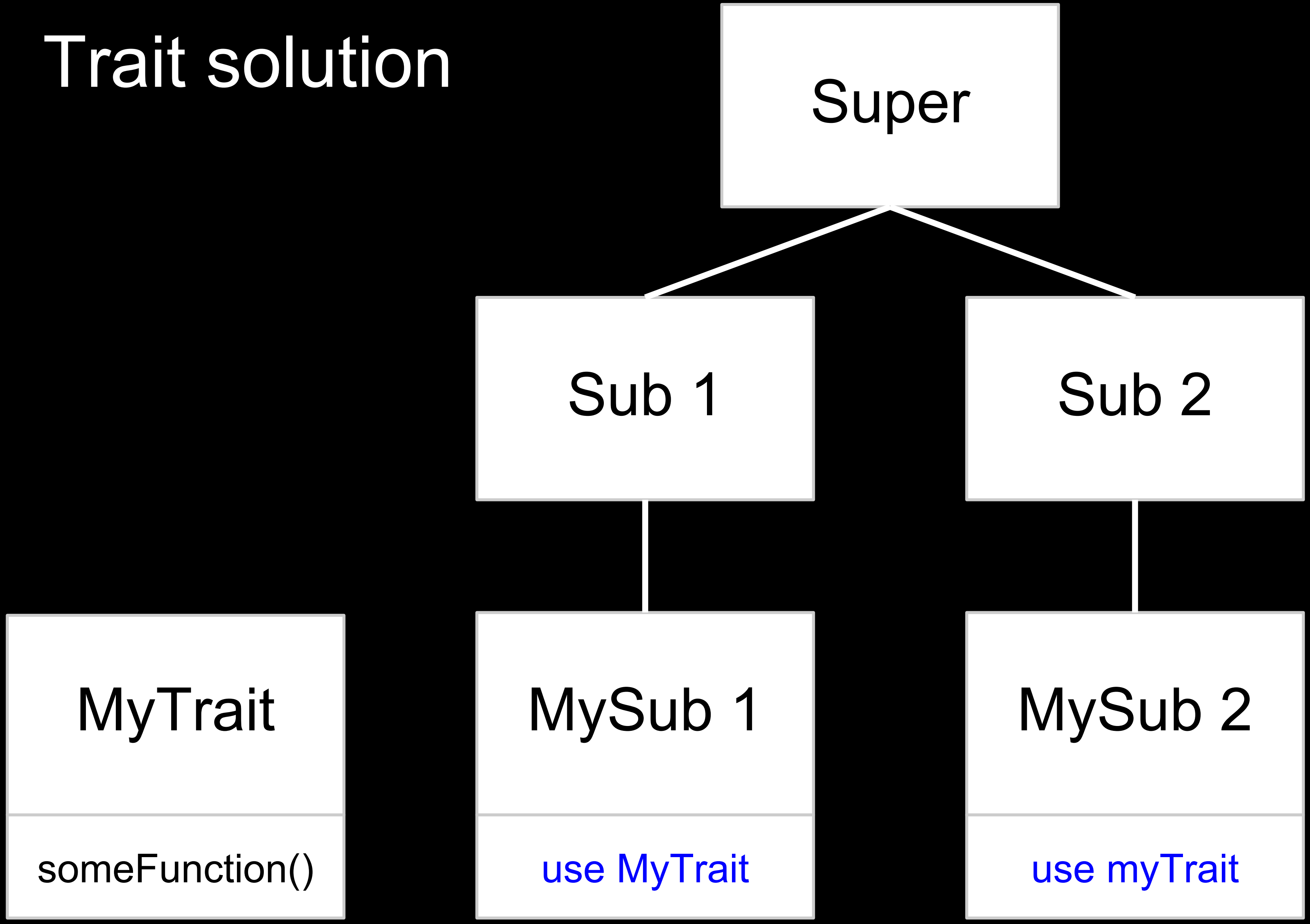
- Multiple inheritance - introduces diamond problem.

- Global functions - no.

- Function/closure passed as parameter - can't access 'this'.

So a trait then?

Trait solution



Zend example of horizontal reuse

- I am not a zend coder.
- Please be gentle.

```
namespace Zend\View;
class TemplatePathStack implements TemplateResolver
{
    public function setOptions($options = array()) {
        if (!is_array($options) && !$options instanceof Traversable) {
            throw new Exception\InvalidArgumentException(
                __METHOD__ . ' expects an array or Traversable'
            );
        }

        foreach ($options as $key => $value) {
            $this->setOption($key, $value); }
            return $this;
        }

        public function setOption($key, $value) {
            switch (strtolower($key)) {
            case 'lfi_protection': $this->setLfiProtection($value); break;
            case 'script_paths': $this->addPaths($value); break;
            default: break;
        }
    }
```

```
namespace Zend\Mvc\Router;
class RouteBroker implements Broker
{
    public function setOptions($options) {
        if (!is_array($options) && !$options instanceof \Traversable) {
            throw new Exception\InvalidArgumentException(sprintf(
            'Expected an array or Traversable; received "%s"',
            (is_object($options) ? get_class($options) :
                gettype($options))
            ));
        }

        foreach ($options as $key => $value) {
            switch (strtolower($key)) {
                case 'class_loader': // handle this case
                Default: break;// ignore unknown options
            }
        }
        return $this;
    }
}
```

Problem: Code Duplication

```
if (!is_array($options) && !$options instanceof \Traversable) {
        throw new Exception\InvalidArgumentException(sprintf(
        'Expected an array or Traversable; received "%s"',
        (is_object($options) ? get_class($options) : gettype($options))
        ));
}

foreach ($options as $key => $value) {
    //handle each case
}

return $this;
```

Trait Time!

```
trait Options
{
      public function setOptions($options)
      {
            if (!is_array($options) && !$options instanceof \Traversable)
            {
                  throw new Exception\InvalidArgumentException(sprintf(
                  'Expected an array or Traversable; received "%s"',
                  (is_object($options) ? get_class($options) : gettype
($options))
                  ));
            }

            foreach ($options as $key => $value) {
                  $this->setOption($key, $value);
            }
            return $this;
      }
}
```

```
namespace Zend\View;
class TemplatePathStack implements TemplateResolver
{
    use Options;

    public function setOption($key, $value) {
        switch (strtolower($key)) {
            case 'lfi_protection':
                $this->setLfiProtection($value);
            break;
            case 'script_paths':
                $this->addPaths($value);
            break;
            default:
            break;
        }
    }
}

$templateStack = new TemplatePathStack();
$templateStack->setOptions(['lfi_protection' => true]);
```

```
namespace Zend\Mvc\Router;

class RouteBroker implements Broker
{
    use Options;

    public function setOption($key, $value) {

        switch (strtolower($key)) {

            case 'class_loader':
                // handle this case
            default:
                // ignore unknown options
            break;
        }
    }
}

$routeBroker = (new RouteBroker)->setOptions
(['class_loader'=>'SomeLoader']);
```

Summary

- Not going to be a huge change in how to write code.
- Main use is to remove code duplication.
- Could be used for badness.

FIN

Links + more slides:

http://zuttonet.com/articles/php-class-traits/

http://hounddog.github.io/blog/using-traits-in-zend-framework-2/

http://www.slideshare.net/NickBelhomme/php-traits-treat-or-threat-11354185

http://blog.everymansoftware.com/2012/09/interfaces-and-traits-powerful-combo.html - badnes

Bonus stuff follows - here be dragons.

Inception

```
trait Trait1{
        function testFunction(){            echo "Hello!";   }
        abstract function required();
}

trait Trait2{
        use Trait1;
}

class TestClass{
        use Trait2;
        function    required()   {  echo "There tiger."; }
}

$class = new TestClass();
$class->testFunction();

//Why would you do this? Seriously - don't do this.
```

Abstract trait + renaming = badness

```
namespace Zend\View;
trait Options {
    abstract public function setOptions($options);
}

class TemplatePathStack {
    //This is where the 'compile time copy + paste' analogy breaks
    use Options {Options::setOptions as setConfig;}

    public function setConfig($options)
    {  echo 'implementation enforced by trait';  }
}

// Fatal error: Class Zend\View\TemplatePathStack contains 1 abstract
method and must therefore be declared abstract or implement the
```

Can be used for evil

```
interface Addressable {
 public function setAddress(Address $address);
 public function getAddress();
}
trait AddressAccessor {
 protected $address;
 public function setAddress(Address $address) {   $this->address = $address;   }
 public function getAddress() {   return $this->address;  }
}
class User implements Addressable {
 use AddressAccessor;
}
class Company implements Addressable {
 use AddressAccessor;
}
```

powerful but, ewww.....this really smells.

Detecting traits

class_uses() - Return the traits used by the given class, but doesn't check class hierarchy, so use this:

```
function class_uses_deep($class, $autoload = true) {
    $traits = [];
    do {
        $traits = array_merge(class_uses($class, $autoload), $traits);
    } while($class = get_parent_class($class));
    foreach ($traits as $trait => $same) {
        $traits = array_merge(class_uses($trait, $autoload), $traits);
    }
    return array_unique($traits);
}
```