

Webbutveckling med PHP

DEPENDENCY INJECTION

Utbildare: Mikael Olsson

mikael.olsson@emmio.se

076-174 90 43

NACKADEMIN

OOP

- Abstract class
- Interface

Abstract class

- PHP 5 introduces abstract classes and methods. Classes defined as abstract cannot be instantiated, and any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature - they cannot define the implementation.
- When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private. Furthermore the signatures of the methods must match, i.e. the type hints and the number of required arguments must be the same. For example, if the child class defines an optional argument, where the abstract method's signature does not, there is no conflict in the signature.

Abstract class - example 1

```
<?php
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_') . "\n";

$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') . "\n";
```

Abstract class - example 2

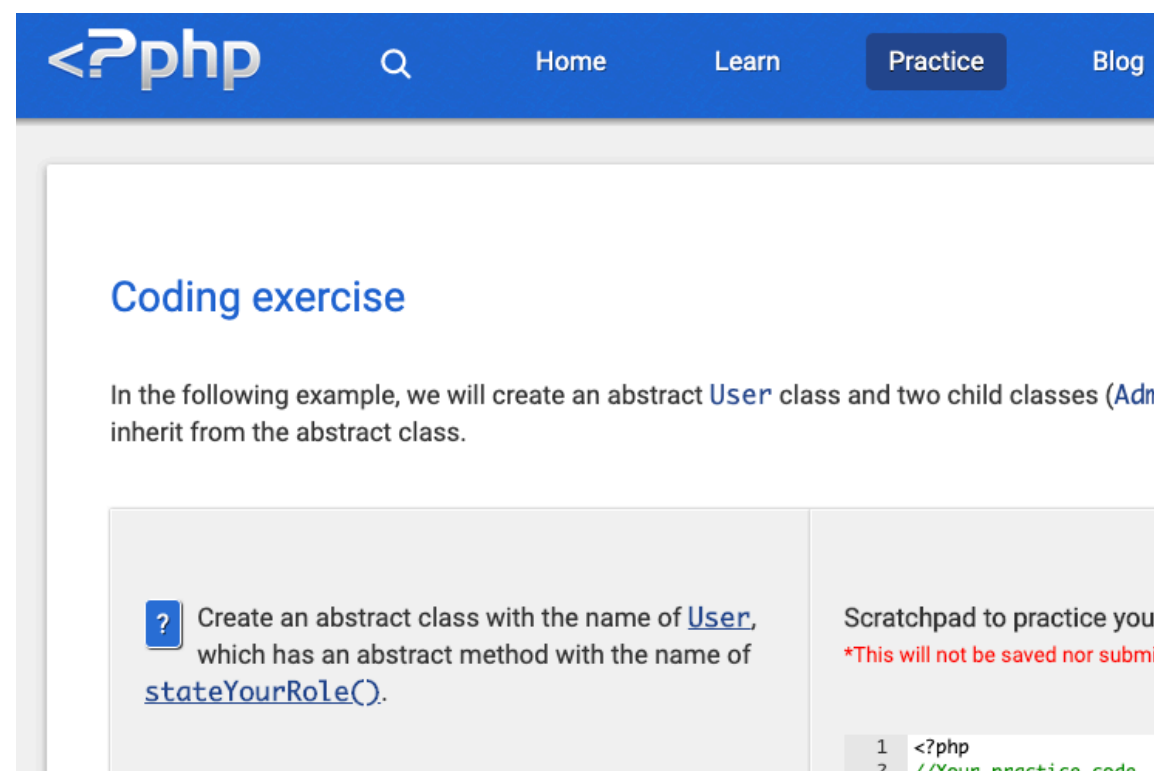
```
<?php
abstract class AbstractClass
{
    // Our abstract method only needs to define the required arguments
    abstract protected function prefixName($name);
}

class ConcreteClass extends AbstractClass
{
    // Our child class may define optional arguments not in the parent's signature
    public function prefixName($name, $separator = ".") {
        if ($name == "Pacman") {
            $prefix = "Mr";
        } elseif ($name == "Pacwoman") {
            $prefix = "Mrs";
        } else {
            $prefix = "";
        }
        return "{$prefix}{$separator} {$name}";
    }
}

$class = new ConcreteClass;
echo $class->prefixName("Pacman"), "\n";
echo $class->prefixName("Pacwoman"), "\n";
```

Abstract classes

- <https://www.youtube.com/watch?v=Eu7l8utquGY>
- Övningar
- <https://phpenthusiast.com/object-oriented-php-tutorials/abstract-classes-and-methods/practice>



The screenshot shows the 'Practice' section of the phpenthusiast.com website. The top navigation bar is blue with the '<?php' logo, a search icon, and links for 'Home', 'Learn', 'Practice' (highlighted), and 'Blog'. Below the navigation bar, the page title 'Coding exercise' is displayed in blue. The main content area contains a paragraph explaining the exercise: 'In the following example, we will create an abstract `User` class and two child classes (`Admin` and `Guest`) which inherit from the abstract class.' Below this text, there are two side-by-side boxes. The left box contains a question icon and the instruction: 'Create an abstract class with the name of `User`, which has an abstract method with the name of `stateYourRole()`.' The right box is titled 'Scratchpad to practice you' and contains a red warning message: '*This will not be saved nor submitted'. Below the warning, there is a code editor with the following code:

```
1 <?php
2 //Your practice code
```

Interface

- Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are implemented.
- Interfaces are defined in the same way as a class, but with the interface keyword replacing the class keyword and without any of the methods having their contents defined.
- All methods declared in an interface must be public; this is the nature of an interface.

Interface - example 1

<?php

```
// Declare the interface 'iTemplate'
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}
```

```
// Implement the interface
// This will work
```

```
class Template implements iTemplate
{
    private $vars = array();

    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }

    public function getHtml($template)
    {
        foreach($this->vars as $name => $value) {
            $template = str_replace('{ ' . $name . ' }', $value, $template);
        }

        return $template;
    }
}
```

```
// This will not work
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
    private $vars = array();

    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
}
```

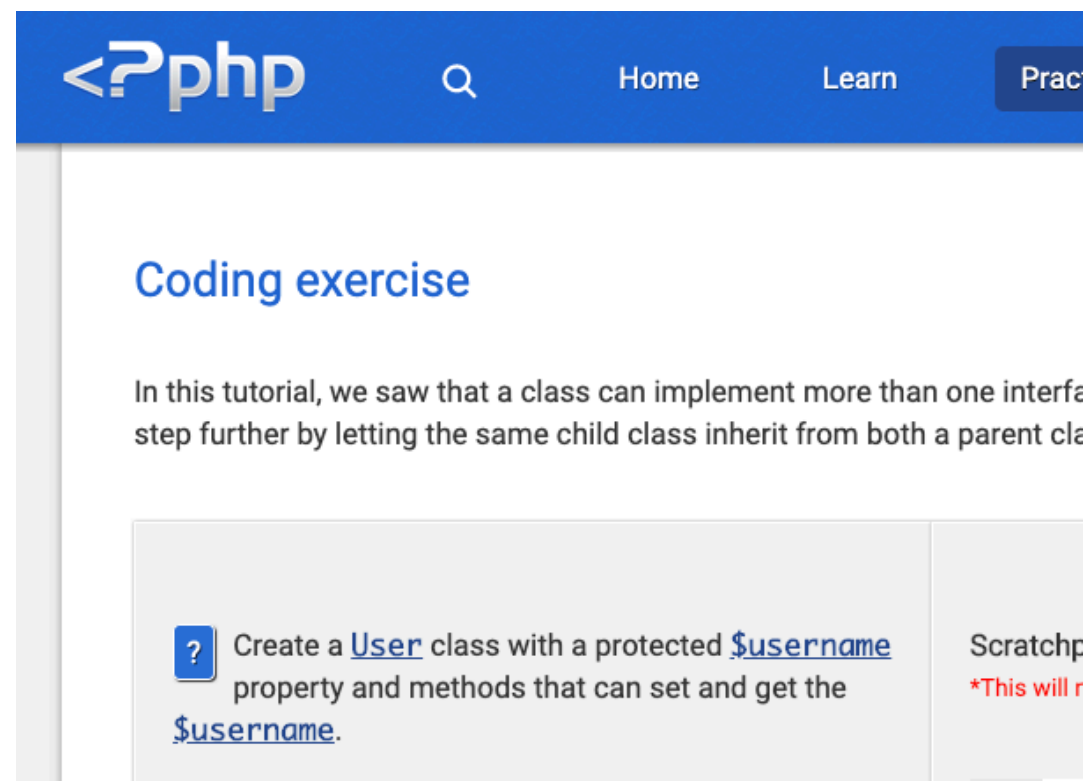

Abstract class vs interface

| Interface | Abstract class |
|--|--|
| Interface support multiple inheritance | Abstract class does not support multiple inheritance |
| Interface does'n Contains Data Member | Abstract class contains Data Member |
| Interface does'n contains Cunstructors | Abstract class contains Cunstructors |
| An interface Contains only incomplete member (signature of member) | An abstract class Contains both incomplete (abstract) and complete member |
| An interface cannot have access modifiers by default everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties |
| Member of interface can not be Static | Only Complete Member of abstract class can be Static |

<https://codeinphp.github.io/post/abstract-class-vs-interface/>

Interface

- <https://www.youtube.com/watch?v=jHjjsgRNFVE>
- Övningar
- <https://phpenthusiast.com/object-oriented-php-tutorials/interfaces/practice>



Type hinting

- I PHP kan vi om vi vill ange vilken typ av data en funktion förväntar sig.
- Type declarations allow functions to require that parameters are of a certain type at call time. If the given value is of the incorrect type, then an error is generated: in PHP 5, this will be a recoverable fatal error, while PHP 7 will throw a TypeError exception.

```
<?php
```

```
function test (String $string, Int $int) {  
    echo $string . PHP_EOL;  
    echo $int . PHP_EOL;  
}
```

```
test("hello", 4);
```

Dependency Injection

- Dependency Injection is providing a component with its dependencies either through constructor injection, method calls or the setting of properties.
- <https://phptherightway.com/>

Dependency Injection

- Here we have a Database class that requires an adapter to speak to the database. We instantiate the adapter in the constructor and create a hard dependency. This makes testing difficult and means the Database class is very tightly coupled to the adapter.

```
<?php
namespace Database;

class Database
{
    protected $adapter;

    public function __construct()
    {
        $this->adapter = new MySqlAdapter;
    }
}

class MySqlAdapter {}
```

Dependency Injection

- This code can be refactored to use Dependency Injection and therefore loosen the dependency.

```
<?php
namespace Database;
```

```
class Database
{
    protected $adapter;
```

```
    public function __construct(MySqlAdapter $adapter)
    {
        $this->adapter = $adapter;
    }
}
```

```
class MySqlAdapter {}
```

Skickar med databasobjektet
som en parameter istället.

- Now we are giving the Database class its dependency rather than creating it itself. We could even create a method that would accept an argument of the dependency and set it that way, or if the `$adapter` property was public we could set it directly.

Complex Problem

- If you have ever read about Dependency Injection then you have probably seen the terms “Inversion of Control” or “Dependency Inversion Principle”. These are the complex problems that Dependency Injection solves.

Inversion of Control

- Inversion of Control is as it says, “inverting the control” of a system by keeping organizational control entirely separate from our objects. In terms of Dependency Injection, this means loosening our dependencies by controlling and instantiating them elsewhere in the system.
- Dependency Injection allows us to more elegantly solve this problem by only injecting the dependencies we need, when we need them, without the need for any hard coded dependencies at all.

S.O.L.I.D.

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

- The Single Responsibility Principle is about actors and high-level architecture. It states that “A class should have only one reason to change.” This means that every class should only have responsibility over a single part of the functionality provided by the software. The largest benefit of this approach is that it enables improved code reusability. By designing our class to do just one thing, we can use (or re-use) it in any other program without changing it.
- https://en.wikipedia.org/wiki/Single_responsibility_principle

Open/Closed Principle

- The Open/Closed Principle is about class design and feature extensions. It states that “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” This means that we should design our modules, classes and functions in a way that when a new functionality is needed, we should not modify our existing code but rather write new code that will be used by existing code. Practically speaking, this means that we should write classes that implement and adhere to interfaces, then type-hint against those interfaces instead of specific classes.
- The largest benefit of this approach is that we can very easily extend our code with support for something new without having to modify existing code, meaning that we can reduce QA time, and the risk for negative impact to the application is substantially reduced. We can deploy new code, faster, and with more confidence.
- https://en.wikipedia.org/wiki/Open/closed_principle

Liskov Substitution Principle

- The Liskov Substitution Principle is about subtyping and inheritance. It states that “Child classes should never break the parent class’ type definitions.” Or, in Robert C. Martin’s words, “Subtypes must be substitutable for their base types.”
- For example, if we have a `FileInterface` interface which defines an `embed()` method, and we have `Audio` and `Video` classes which both implement the `FileInterface` interface, then we can expect that the usage of the `embed()` method will always do the thing that we intend. If we later create a `PDF` class or a `Gist` class which implement the `FileInterface` interface, we will already know and understand what the `embed()` method will do. The largest benefit of this approach is that we have the ability to build flexible and easily-configurable programs, because when we change one object of a type (e.g., `FileInterface`) to another we don’t need to change anything else in our program.
- https://en.wikipedia.org/wiki/Liskov_substitution_principle

Interface Segregation Principle

- The Interface Segregation Principle (ISP) is about business-logic-to-clients communication. It states that “No client should be forced to depend on methods it does not use.” This means that instead of having a single monolithic interface that all conforming classes need to implement, we should instead provide a set of smaller, concept-specific interfaces that a conforming class implements one or more of.
- For example, a `Car` or `Bus` class would be interested in a `steeringWheel()` method, but a `Motorcycle` or `Tricycle` class would not. Conversely, a `Motorcycle` or `Tricycle` class would be interested in a `handlebars()` method, but a `Car` or `Bus` class would not. There is no need to have all of these types of vehicles implement support for both `steeringWheel()` as well as `handlebars()`, so we should break-apart the source interface.
- https://en.wikipedia.org/wiki/Interface_segregation_principle

Dependency Inversion Principle

- The Dependency Inversion Principle is about removing hard-links between discrete classes so that new functionality can be leveraged by passing a different class. It states that one should “Depend on Abstractions. Do not depend on concretions.”. Put simply, this means our dependencies should be interfaces/contracts or abstract classes rather than concrete implementations. We can easily refactor the above example to follow this principle.

```
<?php
namespace Database;

class Database
{
    protected $adapter;

    public function __construct(AdapterInterface $adapter)
    {
        $this->adapter = $adapter;
    }
}

interface AdapterInterface {}

class MysqlAdapter implements AdapterInterface {}
```

- https://en.wikipedia.org/wiki/Dependency_inversion_principle

Benefits

- There are several benefits to the `Database` class now depending on an interface rather than a concretion.
- Consider that we are working in a team and the adapter is being worked on by a colleague. In our first example, we would have to wait for said colleague to finish the adapter before we could properly mock it for our unit tests. Now that the dependency is an interface/contract we can happily mock that interface knowing that our colleague will build the adapter based on that contract.
- An even bigger benefit to this method is that our code is now much more scalable. If a year down the line we decide that we want to migrate to a different type of database, we can write an adapter that implements the original interface and injects that instead, no more refactoring would be required as we can ensure that the adapter follows the contract set by the interface.

Containers

- The first thing you should understand about Dependency Injection Containers is that they are not the same thing as Dependency Injection. A container is a convenience utility that helps us implement Dependency Injection, however, they can be and often are misused to implement an anti-pattern, Service Location. Injecting a DI container as a Service Locator in to your classes arguably creates a harder dependency on the container than the dependency you are replacing. It also makes your code much less transparent and ultimately harder to test.
- Most modern frameworks have their own Dependency Injection Container that allows you to wire your dependencies together through configuration. What this means in practice is that you can write application code that is as clean and de-coupled as the framework it is built on.

Dependency Injection

- In simple terms, Dependency Injection is a design pattern that helps avoid hard-coded dependencies for some piece of code or software.
- The dependencies can be changed at run time as well as compile time. We can use Dependency Injection to write modular, testable and maintainable code:
 - **Modular:** The Dependency Injection helps create completely self-sufficient classes or modules
 - **Testable:** It helps write testable code easily eg unit tests for example
 - **Maintainable:** Since each class becomes modular, it becomes easier to manage it
- <https://codeinphp.github.io/post/dependency-injection-in-php/>

Dependency Injection

- <https://www.youtube.com/watch?v=pDxVRtPXpqY>
- <https://www.youtube.com/watch?v=IKD2-MAkXyQ>
- SOLID principles:
 - https://www.youtube.com/watch?v=DfBqZCA8_Zg&list=PLrIm-p2rpV0EMU_OsbH8RekBNp9buhSr-

Dagens projekt

- I valfria grupper, bygg en applikation som implementerar en studentklass och en lärarklass som båda ärver från en personklass som använder sig av en databasklass.
- Använd så mycket som möjligt av era nya kunskaper om Dependency Injection.

Utvärdering

- Prata i grupper om 2-3 personer i två minuter.
- Vad har varit bra idag?
- Vad skulle kunna förbättras?