eZ publish
conference 2006
Skien, Norway, June 21 - 23

Introduction to object oriented PHP

Marcus Börger

eZ publish Conference

Overview

- What is OOP?
- PHP and OOP

What is OOP

```
class Useless extends Nonsense
{
    abstract function blaBla();
}
```

?

What does OOP aim to achieve?

- Allow compartmentalized refactoring of code
- Promote code re-use
- Promote extensibility, flexibility and adaptability
- Better for team development
- Many patterns are designed for OOP
- Some patterns lead to much more efficient code

- Do you need to use OOP to achieve these goals?
  - ☑ Of course not
  - ☑ It’s designed to make those things easier though

What are the features of OOP?

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

- Encapsulation is about grouping of functionality (operations) and related data (attributes) together into a coherent data structure (classes).

Encapsulation

- Encapsulation is about grouping of functionality (operations) and related data (attributes) together into a coherent data structure (classes).
- Classes represent complex data types and the operations that act on them. An object is a particular instance of a class.

Encapsulation

- Encapsulation is about grouping of functionality (operations) and related data (attributes) together into a coherent data structure (classes).
- Classes represent complex data types and the operations that act on them. An object is a particular instance of a class.
- The basic idea is to re-code real life.

  For instance if you press a key on your laptop keyboard you do not know what is happening in detail. For you it is the same as if you press the keyboard of an ATM. We say the interface is the same. If another person has the same laptop the internal details would be exactly the same.

Encapsulation

- Encapsulation is about grouping of functionality (operations) and related data (attributes) together into a coherent data structure (classes).
- Classes represent complex data types and the operations that act on them. An object is a particular instance of a class.
- The basic idea is to re-code real life.

  For instance if you publish a text that is not really different from publishing a picture. Both are content types and you might want to encapsulate the details on how to do the actual publishing in a class. And once you have that you can easily have contend that consists of both pictures and text and yet use the same operations for publishing.

Encapsulation: Are Objects Just Dictionaries?

- In PHP 4 objects were little more than arrays.
- In PHP 5 you get much more control by visibility, interfaces, type hints, interceptors and more.
- Another difference is coherency. Classes can be told to automatically execute specific code on object creation and destruction.

```
class Simple {
    function __construct() { /*...*/ }
    function __destruct() { /*...*/ }
}
```

Data Hiding

- Another difference between objects and arrays is that objects permit strict visibility semantics. Data hiding eases refactoring by controlling what other parties can access in your code.

☑	public	anyone can access it
☑	protected	only descendants can access it
☑	private	only you can access it
☑	final	no one can re-declare it
☑	abstract	someone else will implement this

Why have these in PHP?

Because sometimes self-discipline isn’t enough.

Inheritance

- Inheritance allows a class to specialize (or extend) another class and inherit all its methods, properties and behaviors.

- This promotes
  - ☑ Extensibility
  - ☑ Reusability
  - ☑ Code Consolidation
  - ☑ Abstraction
  - ☑ Responsibility

The Problem of Code Duplication

- Code duplication contradicts maintainability.
  You often end up with code that looks like this:

```
function foo_to_xml($foo) {
    // generic stuff
    // foo-specific stuff
}

function bar_to_xml($bar) {
    // generic stuff
    // bar specific stuff
}
```

The Problem of Code Duplication

- You could clean that up as follows

```
function base_to_xml($data) { /*...*/ }
function foo_to_xml($foo) {
   base_to_xml($foo);
   // foo specific stuff
}
function bar_to_xml($bar) {
    base_to_xml($bar);
    // bar specific stuff
}
```

- But it’s hard to keep base_to_xml() working for the disparate foo and bar types.

The Problem of Code Duplication

- In an OOP style you would create classes for the Foo and Bar classes that extend from a base class that handles common functionality.
- Sharing a base class promotes sameness.

```
class Base {
   public function toXML()
   {
      /*...*/
   }
}
```

```
class Foo extends Base {
   public function toXML()
   {
      parent::toXML();
      // foo specific stuff
   }
}
```

```
class Bar extends Base {
   public function toXML()
   {
      parent::toXML();
      // bar specific stuff
   }
}
```

Polymorphism?

- Suppose a calendar that is a collection of entries.
  Procedurally dislpaying all the entries might look like:

```
foreach($entries as $entry) {
    switch($entry['type']) {
    case 'professional':
        display_professional_entry($entry);
        break;
    case 'personal':
        display_personal_entry($entry);
        break;
    }
}
```

Simplicity through Polymorphism

- In an OOP paradigm this would look like:

```
foreach($entries as $entry) {
    $entry->display();
}
```

- The key point is we don't have to modify this loop to add new types. When we add a new type, that type gets a display() method so that it knows how to display itself, and we're done.

- Also this is much faster because we do not have to check the type for every element.

Simplicity through Magic?

- Actually in PHP you might want this:

```
foreach($entries as $entry) {
    echo $entry;
}
```

- A class can have a `__tostring()` method which defines how its objects are converted into a textual representation.

- PHP 5.2 supports this in all string contexts.

Another example

```
class Humans {
    public function __construct($name) {
        /*...*/
    }
    public function eat() { /*...*/ }
    public function sleep() { /*...*/ }
    public function snore() { /*...*/ }
    public function wakeup() { /*...*/ }
}
```

Some Inheritance

```
class Humans {
    public function __construct($name) { /*...*/ }
    public function eat() { /*...*/ }
    public function sleep() { /*...*/ }
    public function snore() { /*...*/ }
    public function wakeup() { /*...*/ }
}
class Women extends Humans {
    public function giveBirth() { /*...*/ }
}
```

eZ publish
conference 2006
Skien, Norway, June 21 - 23

Inheritance+Polymorphism

```
class Humans {
    public function __construct($name) { /*...*/ }
    public function eat() { /*...*/ }
    public function sleep() { /*...*/ }
    public function wakeup() { /*...*/ }
}
class Women extends Humans {
    public function giveBirth() { /*...*/ }
}
class Men extends Humans {
    public function snore() { /*...*/ }
}
```

A little abstraction

```
abstract class Humans {
    public function __construct($name) { /*...*/ }
    abstract public function gender();
    public function eat() { /*...*/ }
    public function sleep() { /*...*/ }
    public function wakeup() { /*...*/ }
}
class Women extends Humans {
    public function gender() { return 'female'; }
    public function giveBirth() { /*...*/ }
}
class Men extends Humans {
    public function gender() { return 'male'; }
    public function snore() { /*...*/ }
}
```

Overloading or Polymorphism the other way round

- Unlike other languages PHP does not and will not offer overloading polymorphism for method calling. Thus the following will never work in PHP

```
<?php
class Test {
    function toXML(Personal $obj) //...
    function toXML(Professional $obj) //...
}
?>
```

- To work around this
  - ☑ Use the other way round (call other methods from a single toXML() function in a polymorphic way)
  - ☑ Use switch/case (though this is not the OO way)

Constructor visibility

- A protected constructor prevents instantiation

```
class Base {
   protected function __construct() {
   }
}

class Derived extends Base {
   // constructor is still protected
   static function getBase() {
      return new Base; // Factory pattern
   }
}

class Three extends Derived {
   public function __construct() {
   }
}
```

The Singleton pattern

Sometimes you want only a single instance of aclass to ever exist.

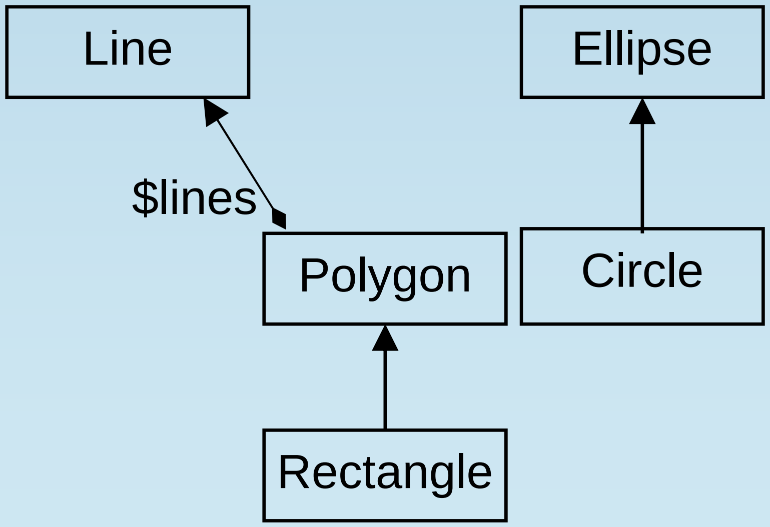
- ☑ DB connections
- ☑ An object representing the user or connection.

```
class Singleton {
    static private $instance;
    protected function __construct() {}
    final private function __clone() {}
    static function getInstance() {
       if(!self::$instance)
          self::$instance = new Singleton();
       return self::$instance;
    }
}
$a = Singleton::getInstance();
$a->id = 1;
$b = Singleton::getInstance();
print $b->id."\n";
```

Different Object same behavior

Often different objects have the some equal functionality without sharing the same base class

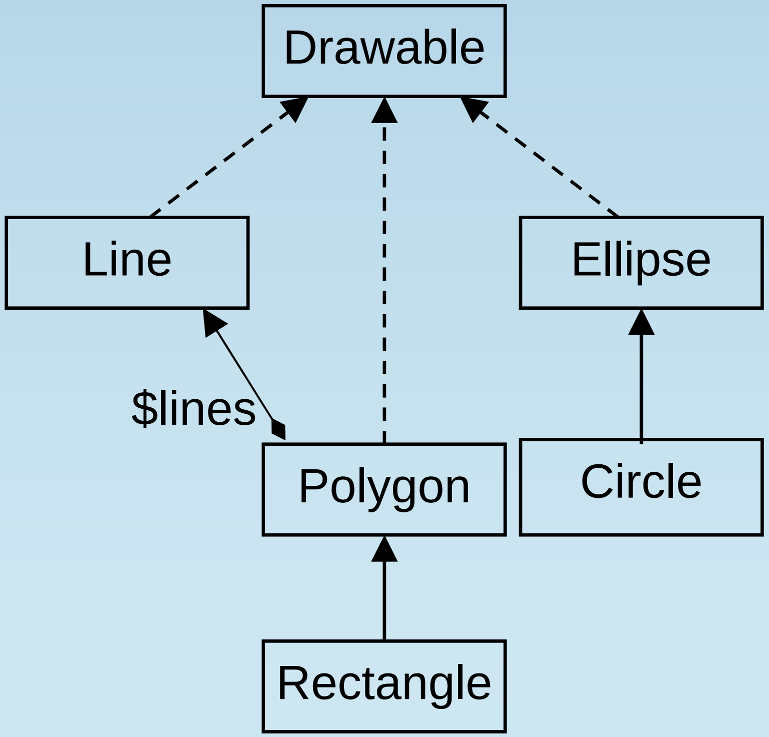
```
class Line {
   function draw() {};
}
class Polygon {
   protected $lines;
   function draw() {
       foreach($this->lines as $line)
            $line->draw();
   }
}
class Rectangle extends Polygon {
   function draw() {};
}
class Ellipse {
   function draw() {};
}
class Circle extends Ellipse {
   function draw() {
       parent::draw();
   }
}
```



Interfaces

- Interfaces describe an abstract class protocol
- Classes may inherit multiple Interfaces

```
interface Drawable {
   function draw();
}
class Line implements Drawable {
   function draw() {};
}
class Polygon implements Drawable {
   protected $lines;
   function draw() {
       foreach($this->lines as $line)
            $line->draw();
   }
}
class Rectangle extends Polygon {
   function draw() {};
}
class Ellipse implements Drawable {
   function draw() {};
}
class Circle extends Ellipse {
   function draw() {
       parent::draw();
   }
}
```



Object to String conversion

__toString(): semi-automatic object to string conversion with echo and print (automatic starting with 5.2)

```
class Object {
    function __toString() {
        return 'Object as string';
    }
}

$o = new Object;

echo $o;

$str = (string) $o; // does NOT call __toString
```

Typehinting

PHP 5 allows to easily force a type of a parameter

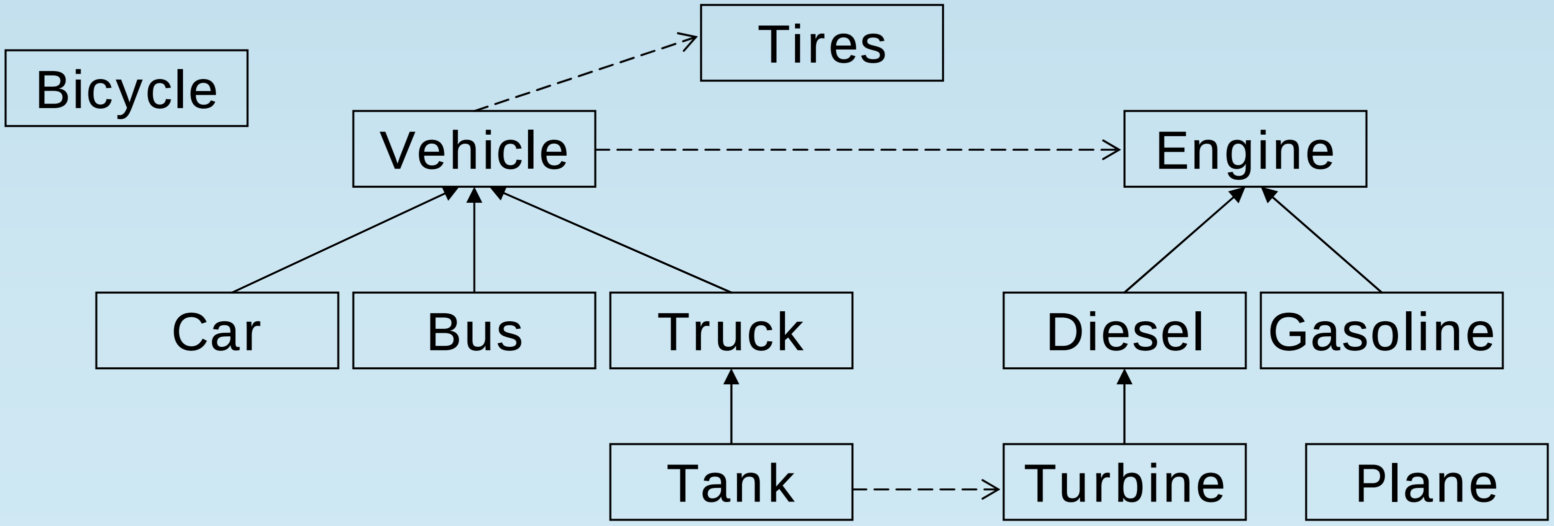
- ☑ PHP does not allow NULL for typehints
- ☑ Typehints must be inherited as given in base class
- ☑ PHP 5.1 offers typehinting with arrays
- ☑ PHP 5.2 offers optional typhinted parameters (= NULL)

```
class Object {
   public function compare(Object $other) {
      // Some code here
   }
   public function compare2($other) {
      if (is_null($other) || $other instanceof Object) {
         // Some code here
      }
   }
}
```

eZ publish conference 2006 Skien, Norway, June 21 - 23

Class Design

- It is important to think about your class hierarchy
- Avoid very deep or broad inheritance graphs
- PHP only supports is-a and has-a relations



Reference

- Everythining about PHP
  - http://php.net
- These slides
  - http://talks.somabo.de
- SPL Documentaion & Examples
  - http://php.net/~helly/php/ext/spl
  - http://cvs.php.net/php-src/ext/spl/examples
  - http://cvs.php.net/php-src/ext/spl/internal
- George Schlossnagle
  - Advanced PHP Programming
- Andi Gutmans, Stig Bakken, Derick Rethans
  - PHP 5 Power Programming