PHP 5 Object Oriented

Marcus Börger
James Cox

php|cruise 2004

Overview

☑ PHP 5 vs. PHP 4

☑ Is PHP 5 revolutionary?

☑ PHP 5 OO
  - ☑ Why is OO a good thing?

$E = mc^2$

- ☑ PHP 5 is "faster" than PHP 4
  - ☑ Speed by design
  - ☑ Nitty gritty engine improvements
    - ☑ Faster callbacks
    - ☑ Faster comparisons
    - ☑ Faster Harder Stronger
  - ☑ New extensions that eliminate userspace code overhead
    - ☑ PDO
    - ☑ SQLite
- ☑ PHP 4 executes code faster
  - ☑ New execution architecture slows things down
  - ☑ Execution architecture isn't terribly important though

Revamped OO Model

- ☑ PHP 5 has really good OO
  - ☑ Better code reuse
  - ☑ Better for team development
  - ☑ Easier to refactor
  - ☑ Some patterns lead to much more efficient code
  - ☑ Fits better in marketing scenarios

PHP 4 and OO ?

- Poor Object model
  - ☑ Methods
    - ☒ No visibility
    - ☒ No abstracts, No final
    - ☒ Static without declaration
  - ☑ Properties
    - ☒ No default values
    - ☒ No static properties
  - ☑ Inheritance
    - ☒ No abstract, final inheritance, no interfaces
  - ☑ Object handling
    - ☒ Copied by value
    - ☒ No destructors

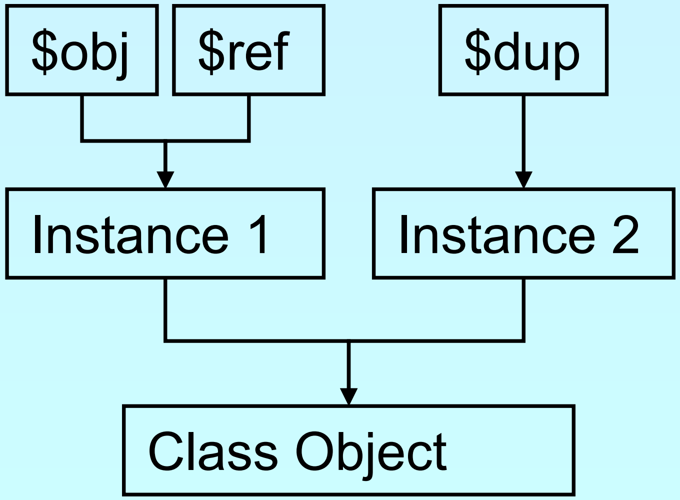
ZE2's revamped object model

- ☑ Objects are referenced by identifiers
- ☑ Constructors and Destructors
- ☑ Static members
- ☑ Default property values
- ☑ Constants
- ☑ Visibility
- ☑ Interfaces
- ☑ Final and abstract members
- ☑ Interceptors
- ☑ Exceptions
- ☑ Reflection API
- ☑ Iterators

Objects referenced by identifiers

- ☑ Objects are no longer copied by default
- ☑ Objects may be copied using __clone()

```php
<?php

class Object {};

$obj = new Object();

$ref = $obj;

$dup = $obj->__clone();

?>
```



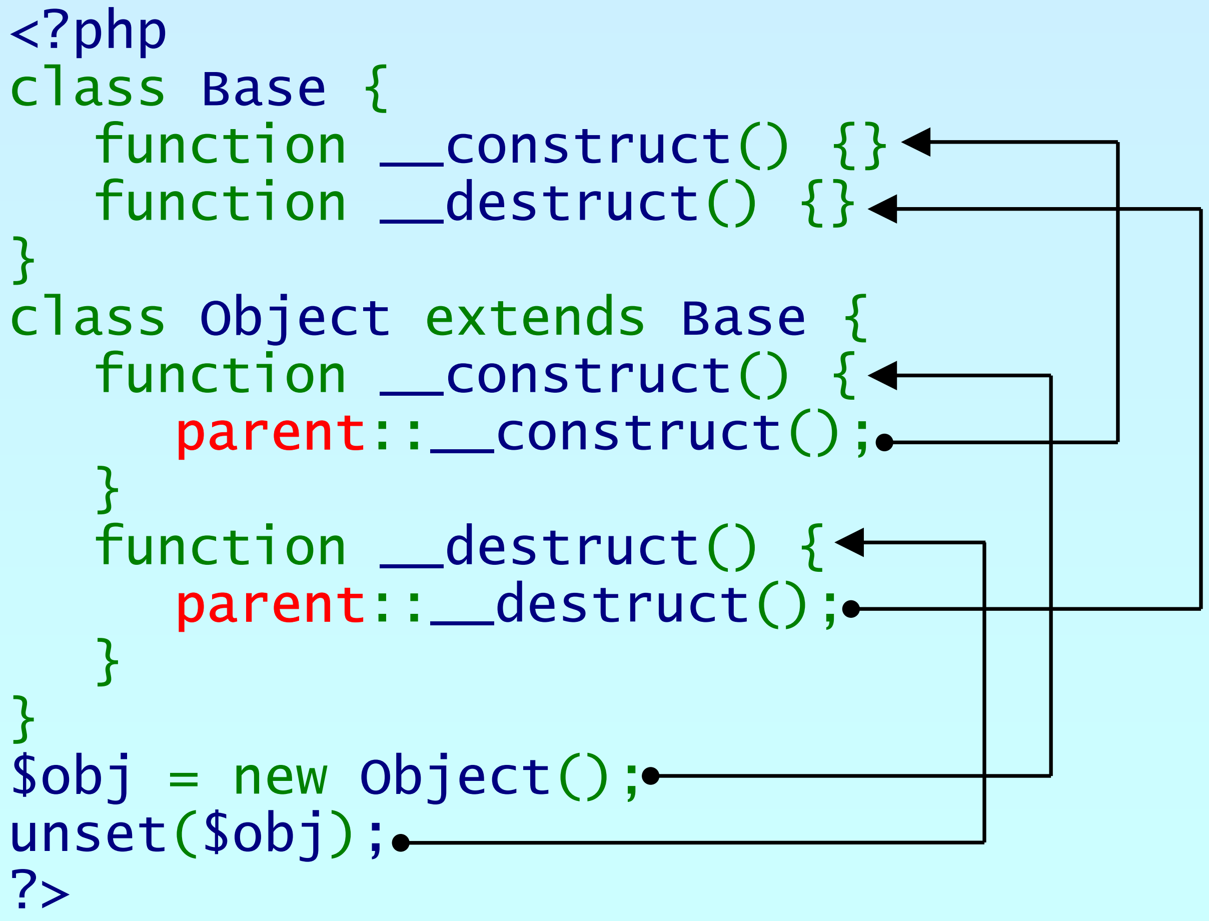
Constructors and Destructors

- ☑ Constructors/Destructors control object lifetime
  - ☑ Constructors may have both new OR old style names
  - ☑ Destructors are called when deleting last reference

```php
<?php

class Object {
  function __construct() {}
  function __destruct() {}
}
$obj = new Object();
unset($obj);

?>
```

Constructors and Destructors

☑ Parents must be called manually

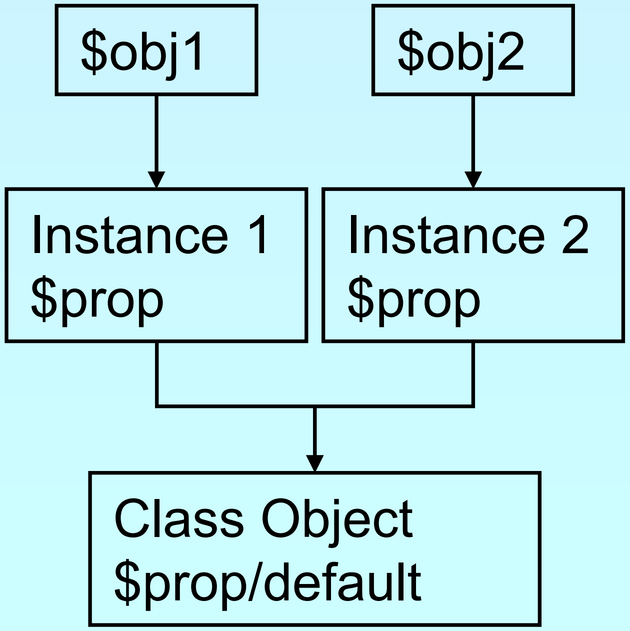


```php
<?php
class Base {
   function __construct() {}
   function __destruct() {}
}
class Object extends Base {
   function __construct() {
      parent::__construct();
   }
   function __destruct() {
      parent::__destruct();
   }
}
$obj = new Object();
unset($obj);
?>
```

Default property values

- ☑ Properties can have default values
  - ☑ Bound to the class not to the object
  - ☑ Default values cannot be changed but overwritten

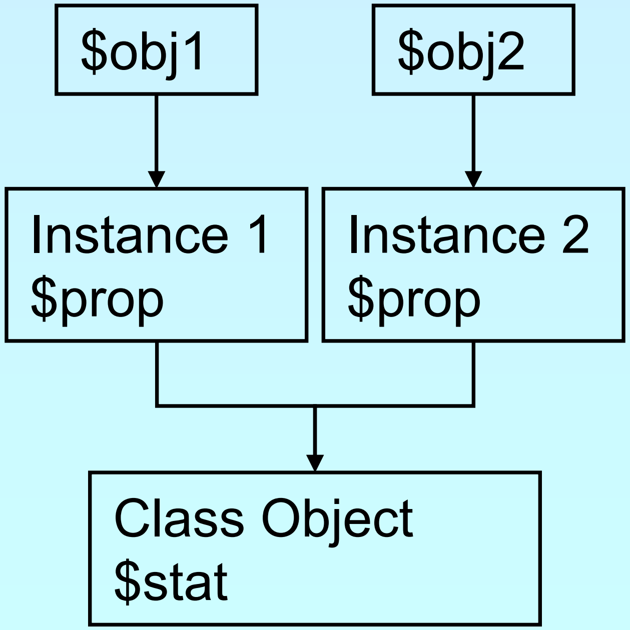
```
<?php

class Object {
  var $prop = "Hello\n";
}

$obj1 = new Object;
$obj1->prop = "Hello World\n";

$obj2 = new Object;
echo $obj2->prop; // Hello

?>
```



Static members

☑ Static methods and properties
- ☑ Bound to the class not to the object
- ☑ Can be initialized

```
<?php

class Object {
  var $pop;
  static $stat = "Hello\n";
  static function test() {
    echo self::$stat;
  }
}

Object::test();
$obj1 = new Object;
$obj2 = new Object;

?>
```



New pseudo constants

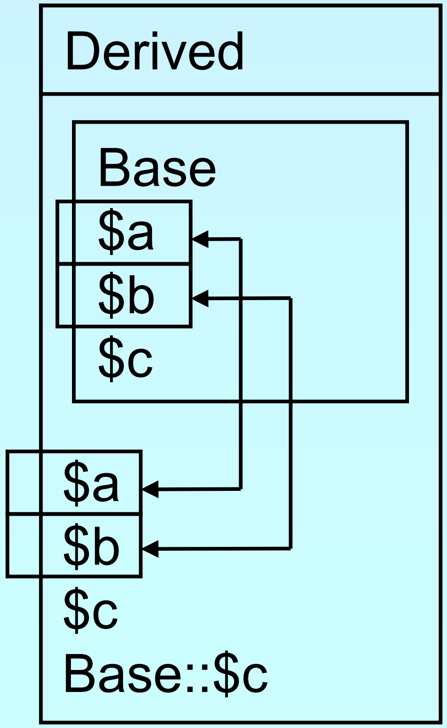
☑	__CLASS__	shows the current class name
☑	__METHOD__	shows class and method or function
☑	Self	references the class itself
☑	Parent	references the parent class
☑	$this	references the object itself

```
<?php
class Base {
    static function Show() {
        echo __FILE__.'('.__LINE__.'):'.__METHOD__."\n";
    }
}
class Object extends Base {
    static function Use() {
        Self::Show();
        Parent::Show();
    }
    static function Show() {
        echo __FILE__.'('.__LINE__.'):'.__METHOD__."\n";
    }
}
?>
```

Visibility

- ☑ Controlling member visibility / Information hiding
  - ☑ A derived class does not know inherited privates
  - ☑ An inherited protected member can be made public

```php
<?php
class Base {
  public $a;
  protected $b;
  private $c;
}
class Derived extends Base {
  public $a;
  public $b;
  private $c;
}
?>
```



Constructor visibility

- ☑ A protected constructor prevents instantiation
- ☑ Adding final prevents instantiation of child classes
- ☑ Static members may call non public constructors

```php
<?php
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 {
   // constructor is public, Three may be instantiated
   public function __construct() {
   }
}
?>
```

Clone visibility

- ☑ A protected __clone prevents external cloning
- ☑ A private final __clone prevents cloning
- ☑ Before __clone is called all properties are copied

```php
<?php
class Base {
  protected function __clone() {
  }
}
class Derived extends Base {
  public function __clone() {
    return new Base;
  }
  public static function copyBase() {
    return Base::__clone();
  }
}
?>
```

```php
<?php
class Base {
  private final function __clone() {
  }
}
class Derived extends Base {
  // public function __clone() {
  //   return new Base;
  // }
  // public static function copyBase() {
  //   return Base::__clone();
  // }
}
?>
```

Constants

- ☑ Constants are read only static members
- ☑ Constants are always public

```
<?php
class Base {
  const greeting = "Hello\n";
}
class Dervied extends Base {
  const greeting = "Hello World\n";
  static function func() {
    echo parent::greeting;
  }
}
echo Base::greeting;
echo Derived::greeting;
Derived::func();
?>
```

Abstract members

- ☑ Properties cannot be made abstract
- ☑ Methods can be abstract
  - ☑ They cannot have a body (aka default implementation)
  - ☑ A class with an abstract method must be abstract
- ☑ Classes can be made abstract
  - ☑ Those classes cannot be instantiated

```php
<?php
abstract class Base {
  abstract function no_body();
}
class Derived extends Base {
  function no_body() { echo "Body\n"; }
}
?>
```

Final members

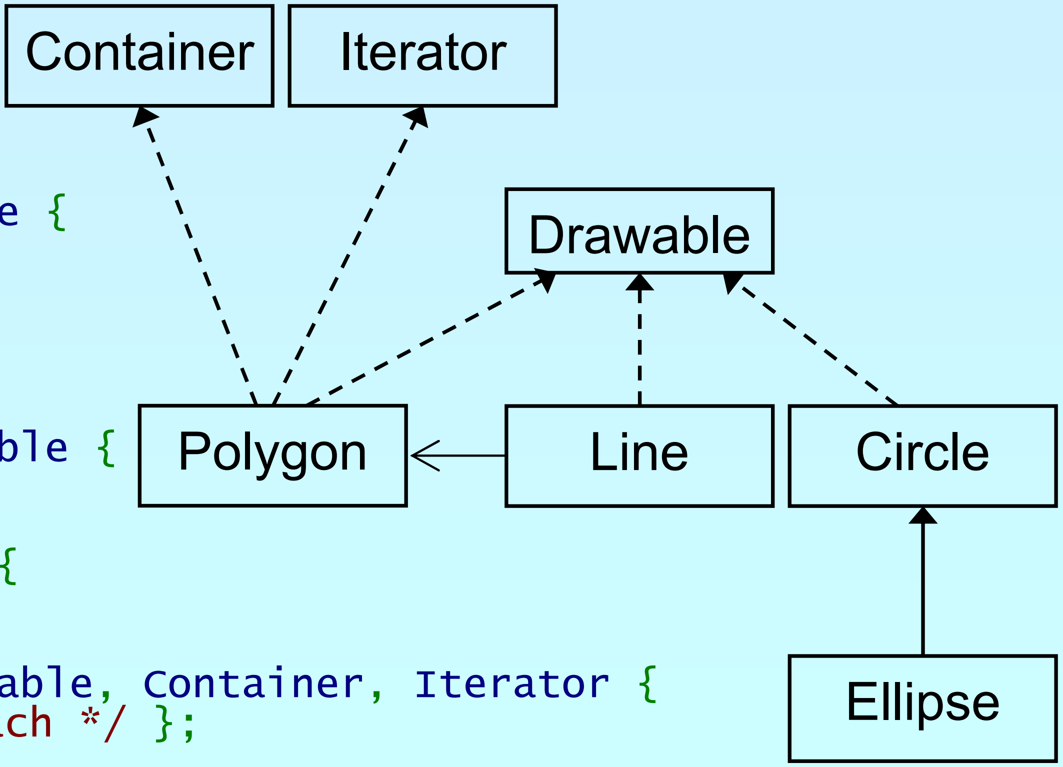
- ☑ Methods can be made final
  - ☑ They cannot be overwritten
  - ☑ They are class invariants
- ☑ Classes can be made final
  - ☑ They cannot be inherited

```php
<?php
class Base {
  final function invariant() { echo "Hello\n"; }
}
class Derived extends Base {
}
final class Leaf extends Derived {
}
?>
```

Interfaces

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

```php
<?php
interface Drawable {
    function draw();
}
class Line implements Drawable {
    function draw() {};
}
interface Container {
    function insert($elem);
}
class Circle implements Drawable {
    function draw() {};
}
class Ellipse extends Circle {
    function draw() {};
}
class Polygon implements Drawable, Container, Iterator {
    function draw() { /* foreach */ };
}
?>
```



Property types

- ☑ Declared properties
  - ☑ May have a default value
  - ☑ Can have selected visibility
- ☑ Implicit public properties
  - ☑ Declared by simply using them in ANY method
- ☑ Virtual properties
  - ☑ Handled by interceptor methods
- ☑ Static properties

Object to String conversion

☑ __toString(): automatic object string conversion

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

$o = new Object;

echo $o;

$str = (string) $o;
?>
```

Interceptors

☑ Allow to dynamically handle non class members
  ☑ Lazy initialization of properties
  ☑ Simulating Object aggregation, Multiple inheritance

```
<?php
class Object {
   protected $virtual = array();
   function __get($name) {
      return @$virtual[$name];
   }
   function __set($name, $value) {
      $virtual[$name] = $value;
   }
   function __call() {
      echo 'Could not call ' . __CLASS__ . '::' . $func . "\n";
   }
}
?>
```

Exceptions

☑ Respect these rules

1. Exceptions are exceptions
2. Never use exceptions for control flow
3. Never ever use exceptions for parameter passing

```
<?php
try {
    // your code
    throw new Exception();
}
catch (Exception $e) {
    // exception handling
}
?>
```

Exception specialization

- ☑ Exception must be derived from class exception
- ☑ Exceptions should be specialized

```php
<?php
class YourException extends Exception {
}
try {
    // your code
    throw new YourException();
}
catch (YourException $e) {
    // exception handling
}
catch (Exception $e) {
    // exception handling
}
?>
```

Exception specialization

- ☑ Exception blocks can be nested
- ☑ Exceptions can be rethrown

```php
<?php
class YourException extends Exception {};
try {
    try {
        // your code
        throw new YourException();
    }
    catch (YourException $e) {
        // exception handling
        throw $e;
    }
    catch (Exception $e) {
        // exception handling
    }
}
catch (YourException $e) {
    // exception handling
}
?>
```

Constructor failure

- ☑ Constructors do not return the created object
  - ➡ Overriding $this as in PHP 4 is no longer possible
- ☑ Exceptions allow to handle failed constructors

```
<?php
class Object {
    function __construct() {
        throw new Exception;
    }
}
try {
    $o = new Object;
}
catch (exception $e) {
    echo "Object could not be instantiated\n";
}
?>
```

Reflection API

- ☑ Can reflect nearly all aspects of your PHP code
  - ☑ Functions
  - ☑ Classes, Methods, Properties
  - ☑ Extensions

```php
<?php
class Foo {
   public $prop;
   function Func($name) {
      echo "Hello $name";
   }
}

ReflectionClass::export('Foo');
ReflectionObject::export(new Foo);
ReflectionMethod::export('Foo', 'func');
ReflectionProperty::export('Foo', 'prop');
ReflectionExtension::export('standard');
?>
```

Why else

☑ Simplify situations where a lot of stuff may fail

```
<?php

if (@$db=sqlite_open($dbname))
{
   if (@$res = sqlite_query())
   {
      // handle result
      if (@$res = sqlite_query())
      {
         // handle result
      }
   }
}
if (sqlite_last_error($db))
{
   // error handling
}
?>
```

```
<?php

try
{
   $db = new sqlite_db($dbname);
   $res = sqlite_query();
   // handle result
   $res = sqlite_query():
   // handle result
}
catch (sqlite_exception $err)
{
   // error handling
}

?>
```

Iterators

- ☑ Some objects can be iterated
- ☑ Others show their properties

```php
<?php

class Object {
    public $prop1 = "Hello";
    public $prop2 = "World\n";
}

foreach(new Object as $prop) {
    echo $prop;
}

?>
```

Typehinting

- ☑ PHP 5 allows to easily force a type of a parameter
  - ☑ NULL is allowed with typehints

```
<?php
class Object {
   public function compare(Object $other) {
      // Some code here
   }
}
?>
```

Iterators

☑ Engine internal Iterator

☑ User Iterators

```
<?php
interface Iterator {
  function rewind();
  function valid();
  function current();
  function key();
  function next();
}
?>
```

```
<?php
$it = get_resource();
foreach($it as $key=>$val) {
  // access data
}
?>
```

```
<?php
abstract class FilterIterator implements Iterator {
  function __construct(Iterator $input)...
  function rewind()...
  abstract function accept();
  function valid()...
  function current()...
  function key()...
  function next()...
}
?>
```

```
<?php
class MyFilter extends FilterIterator {
  function __construct(Iterator $it, $rex) {
    Filter::__construct($it);
    $this->rex = $rex;
  }
  function accept() {
    return preg_match($this->rex,
      $this->current());
  }
}
?>
```

```
<?php
$it = get_resource();
foreach(new MyFilter($it, $regular_expression) as $key=>$val) {
  // access filtered data only
}
?>
```

```
<?php
$it = get_resource();
for ($it->rewind(); $it->valid(); $it->next()) {
  $value = $it->current(); $key = $it->key();
}
?>
```

New extensions

☑ New OO extensions and state/schedule

☑ FFI	PECL / 5.0
☑ Date	PECL / 5.1?
☑ DOM	built-in, default / 5.0
☑ MySQLi	built-in / 5.0
☑ PDO	5.1?
☑ PIMP	5.0?
☑ SimpleXML	built-in, default / 5.0
☑ SOAP	built-in / 5.0
☑ SPL	built-in, default / 5.0
☑ SQLite	built-in, default / 5.0
☑ Tidy	built-in, default / 5.0
☑ XSL	built-in / 5.0

Resources

- ☑ http://php.net
- ☑ http://zend.com