

PHP 5

and the new OO features

Marcus Börger
George Schlossnagle

OSCON 2004

Overview

- ☑ What is OOP?

- ☑ PHP and OOP
 - ☑ PHP 5 vs. PHP 4
 - ☑ Is PHP 5 revolutionary?

- ☑ PHP 5 OOP in detail

- ☑ Using PHP 5 OOP by example



What is OOP



What does OOP aim to achieve?

- ☑ Allow compartmentalized refactoring of code.
- ☑ Promote code re-use.
- ☑ Promote extensibility, flexibility and adaptability.
- ☑ Better for team development.
- ☑ 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.



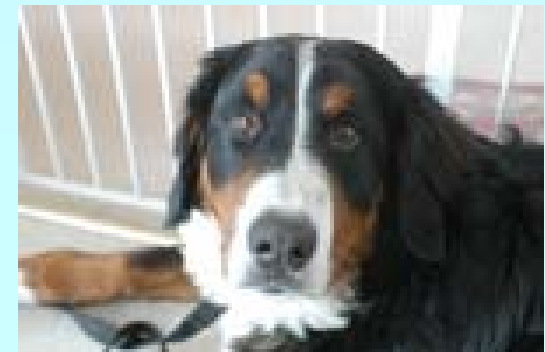
What are the features of OOP?

- ☑ Group data with functionality
- ☑ Encapsulation
- ☑ Inheritance
- ☑ Polymorphism



Encapsulation

- ☑ Encapsulation is about grouping of 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. For example 'Dog' may be a class (it's a type of thing), while Grendel (my dog) is an instance of that class.



Encapsulation: Are Objects Just Dictionaries?

- ☑ Classes as dictionaries are a common idiom, seen in C:

```
typedef struct _entry {
    time_t date;
    char *data;
    char *(*display)(struct _entry *e);
} entry;
entry *e = (entry*)malloc(sizeof(entry));
// initialize e
e->display(e);
```

- ☑ You can see this idiom in Perl and Python, both of which prototype class methods to explicitly grab \$this (or their equivalent).



Encapsulation: Are Objects Just Dictionaries?

- ☑ PHP is somewhat different, since PHP functions aren't really first class objects. Still, PHP4 objects were little more than arrays.
- ☑ The difference is coherency. Classes can be told to automatically execute specific code on object creation and destruction.

```
<?php
class Simple {
    function __construct() { /* ... */ }
    function __destruct() { /* ... */ }
}
?>
```



Data Hiding

- ☑ Another difference between objects and arrays are 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



A Simple Inheritance Example

```
class Dog {
    public function __construct($name) {
        /* ... */
    }
    public function bark() { /* ... */ }
    public function sleep() { /* ... */ }
    public function eat() { /* ... */ }
}
class Rottweiler extends Dog {
    public function guard($person) {
        /* ... */
    }
}
```



Inheritance and Code Duplication

- ☑ Code duplication is a major problem for 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 Foo extends Base {  
    public function toXML() {  
        parent::toXML();  
        // foo specific stuff  
    }  
}
```



Polymorphism



Suppose we have a calendar that is a collection of entries. Procedurally displaying 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 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.



Polymorphism

the other way round

- ☑ Unlike other languages PHP does not and will not offer polymorphism for method calling. Thus the following will never be available 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)



PHP and OOP



PHP 4 and OOP ?



Poor Object model

✓ Methods

- ✗ No visibility
- ✗ No abstracts, No final
- ✗ Static without declaration

✓ Properties

- ✗ No default values
- ✗ No static properties
- ✗ No constants

✓ Inheritance

- ✗ No abstract, final inheritance, no interfaces
- ✗ No prototype checking, no types

✓ 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



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 5 OOP in detail

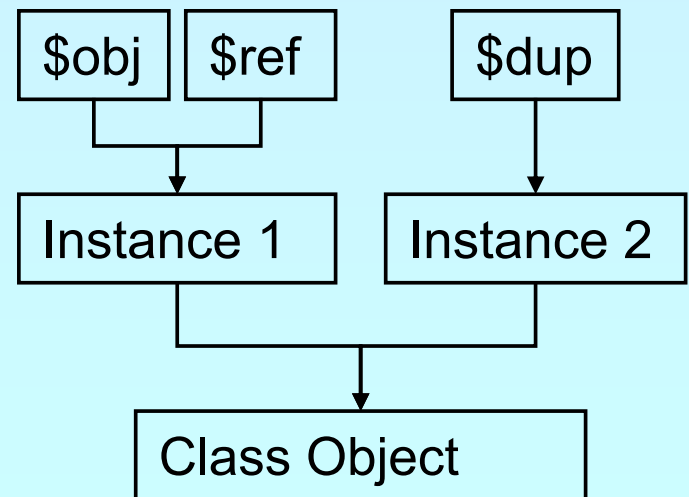


Objects referenced by identifiers

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

```
<?php
```

```
class Object {};  
  
$obj = new Object();  
  
$ref = $obj;  
  
$dup = clone $obj;  
  
?>
```

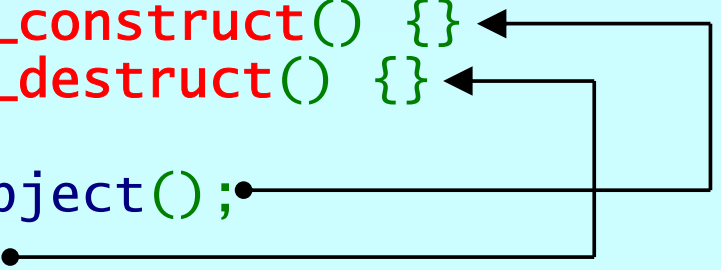


Constructors and Destructors

- ☑ Constructors/Destructors control object lifetime
 - ☑ Constructors may have both new OR old style name
 - ☑ New style constructors are preferred
 - ☑ Constructors must not use inherited protocol
 - ☑ Destructors are called when deleting the last reference
 - ☑ No particular or controllable order during shutdown
 - ☑ Destructors cannot have parameters
 - ☑ Since PHP 5.0.1 destructors can work with resources

<?php

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



Constructors and Destructors



Parents must be called manually

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



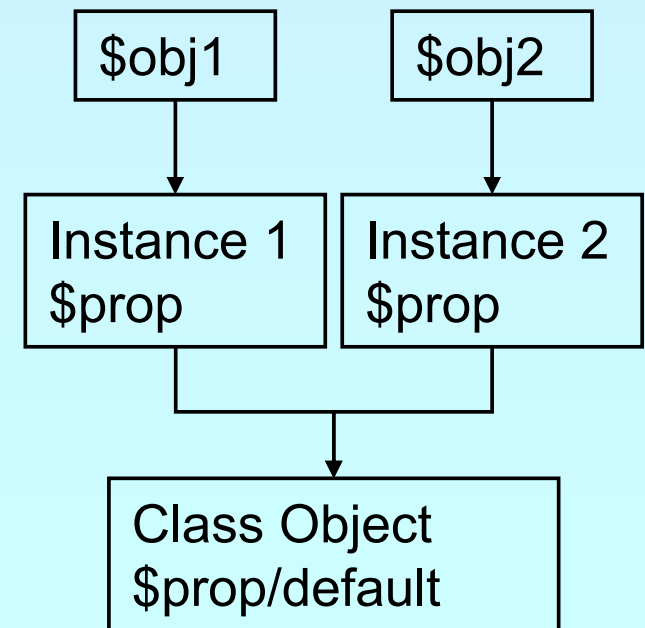
Default property values

- ☑ 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
```

```
?>
```



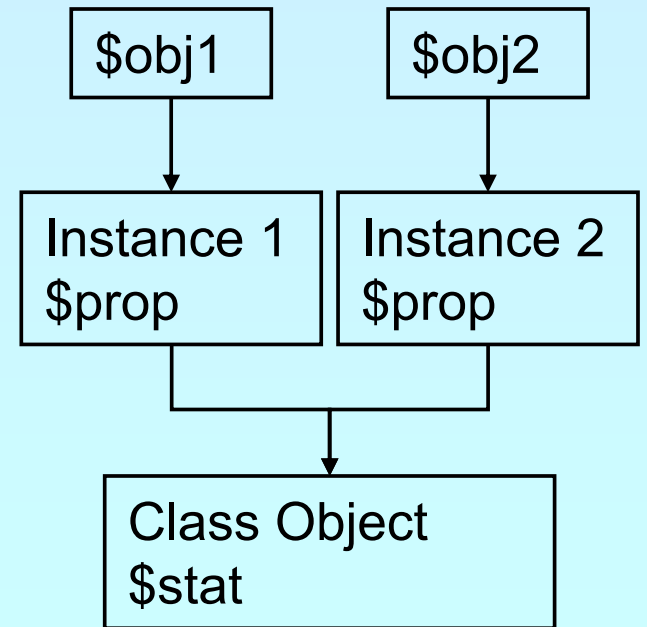
Static members



Static methods and properties

- ✓ Bound to the class not to the object
- ✓ Can be initialized

```
<?php
class Object {
    var $prop;
    static $stat = "Hello\n";
    static function test() {
        echo self::$stat;
    }
}
Object::test();
$obj1 = new Object;
$obj2 = new Object;
?>
```



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

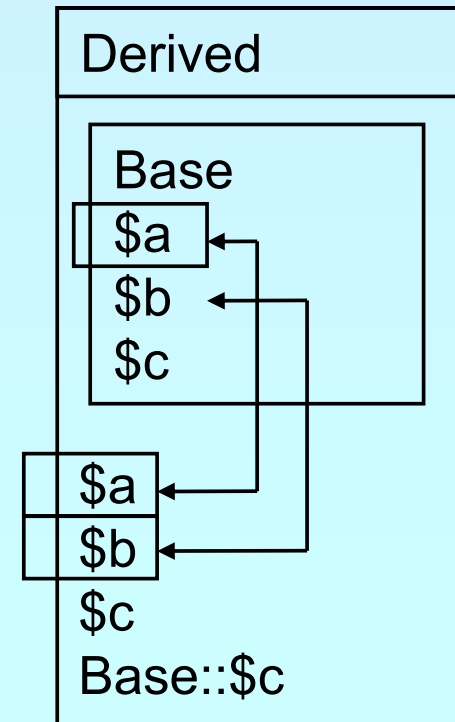
```
<?php
class Base {
    static function Show() {
        echo __FILE__.'('.__LINE__.'):'.__METHOD__."\n";
    }
}
class Object extends Base {
    static function Use() {
        self::Show();
        parent::Show();
    }
    static function Show() {
        echo __FILE__.'('.__LINE__.'):'.__METHOD__."\n";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



Visibility

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

```
<?php
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

```
<?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 {
    public function __construct() {
    }
}
?>
```



Clone visibility

- ✓ A protected `__clone` prevents external cloning
- ✓ A private final `__clone` prevents cloning

```
<?php
class Base {
    protected function clone() {
    }
}
class Derived extends Base {
    public function clone($that) {
        // // some object cloning code
    }
    public static function copyBase($that) {
        // return clone $that;
    }
}
?>
```



The Singleton pattern

- ☑ Sometimes you want only a single instance of any object to ever exist.
 - ☑ DB connections
 - ☑ An object representing the requesting user or connection.

```
<?php
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";
?>
```



Constants

- ✓ Constants are read only static properties
- ✓ Constants are always public

```
<?php
class Base {
    const greeting = "Hello\n";
}
class Derived 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 don't have a body
 - ✓ A class with an abstract method must be abstract
- ✓ Classes can be made abstract
 - ✓ The class cannot be instantiated

```
<?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

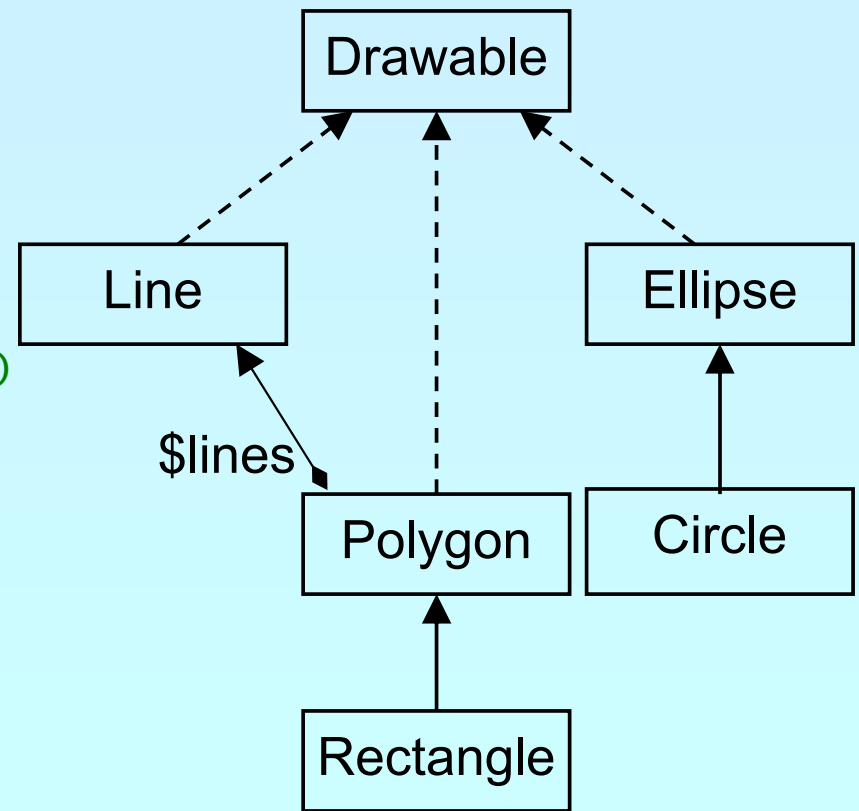
```
<?php
class Base {
    final function invariant() { echo "Hello\n"; }
}
class Derived extends Base {
}
final class Leaf extends Derived {
}
?>
```



Interfaces

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

```
<?php
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();
    };
}
?>
```



Property kinds

- ☑ 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
 - ☑ Bound to the class rather than to the instance



Object to String conversion



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

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

$o = new Object;

echo $o;

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



Interceptors

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

```
<?php
class Object {
    protected $virtual;
    function __get($name) {
        return @$this->virtual[$name];
    }
    function __set($name, $value) {
        $this->virtual[$name] = $value;
    }
    function __call($func, $params) {
        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

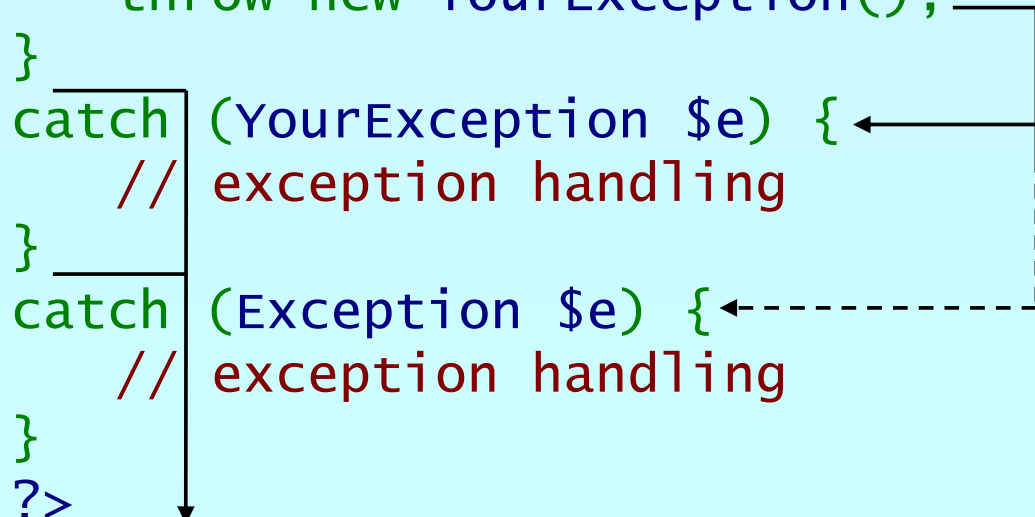
```
<?php
try {
    // your code
    throw new Exception();
}
catch (Exception $e) {
    // exception handling
}
?>
```



Exception specialization

- ☑ Exceptions should be specialized
- ☑ Exceptions should inherit built in class exception

```
<?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 re thrown

```
<?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
- ☑ 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";
}
?>
```



Convert Errors to Exceptions

☑ Implementing PHP 5.1 class `ErrorException`

```
<?php
class ErrorException extends Exception {
    protected $severity, $message;
    function __construct($message, $code, $severity){
        parent::__construct($message, $code);
        $this->severity = $severity;
    }
    function getSeverity() {
        return $this->severity;
    }
}
function ErrorsToExceptions($severity, $message) {
    throw new ErrorException($message, 0, $severity);
}
set_error_handler('ErrorsToExceptions');
?>
```



Typehinting

- ☑ PHP 5 allows to easily force a type of a parameter
 - ☑ Beta 1 and beta 2 allowed NULL with typehints
 - ☑ Starting with Beta 3 NULL is disallowed for typehints
 - ☑ Typehints must be inherited as given in base class
 - ☑ PHP 5.1 will offer a syntax to explicitly allow NULL
 - ☑ PHP 5.1 will offer typehinting with arrays

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



Reflection API

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

```
<?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');
?>
```



Dynamic object creation



Reflection API allows to dynamically create objects

```
<?php
class Test {
    function __construct($x, $y = NULL) {
        $this->x = $x;
        $this->y = $y;
    }
}
function new_object_array($class, $parameters = NULL) {
    return call_user_func_array(
        array(new ReflectionClass($class), 'newInstance'),
        $parameters);
}

new_object_array('stdClass');
new_object_array('Test', array(1));
new_object_array('Test', array(1, 2));
?>
```



__autoload



__autoload supports automatic class file loading

- ✓ It's a good idea to have __autoload in an auto_prepend_file
- ✗ You can only have one __autoload function

```
<?php
function __load_class($classname, $dir) {
    $file = $dir . '/' . $classname . '.inc';
    if (file_exists($file)) {
        require_once($file);
        return true;
    }
    return false;
}

function __autoload($classname) {
    $classname = strtolower($classname);
    $inc = split(':', ini_get('include_path'));
    $inc[] = '.';
    $inc[] = dirname($_SERVER['PATH_TRANSLATED']);
    foreach($inc as $dir) {
        if (__load_class($classname, $dir)) {
            return;
        }
    }
    error_log('Class not found (' . $classname . ')');
}
?>
```



Using PHP 5 OOP by example



Built-in Interfaces

- ☑ PHP 5 contains built-in interfaces that allow you to change the way the engine treats objects.
 - ☑ `ArrayAccess`
 - ☑ `Iterator`
 - ☑ `IteratorAggregate`

- ☑ Built-in extension SPL provides more Interfaces and Classes
 - ☑ `ArrayObject`, `ArrayIterator`
 - ☑ `FilterIterator`
 - ☑ `RecursiveIterator`

 - ☑ Use CLI: `php -r 'ReflectionExtension::export("SPL");'`



Array Access Interception

- ✓ Allows for creating objects that can be transparently accessed as arrays.
- ✓ When combined with the iterator interface, it allows for creating 'arrays with special properties'.

```
<?php
interface ArrayAccess {
    // @return whether $offset is valid (true/false)
    function offsetExists($offset);

    // @return the value associated with $offset
    function offsetGet($offset);

    // associate $value with $offset (store the data)
    function offsetSet($offset, $value);

    // unset the data associated with $offset
    function offsetUnset($offset);
}
?>
```



ArrayAccess Example

- ☑ We want to create variables which can be shared between processes.
- ☑ We will set up interception so that access attempts on the variable are actually performed through a DBM file.



Binding Access to a DBM

```
<?php
class Dbareader implements ArrayAccess {
    protected $db = NULL;
    function __construct($file, $handler) {
        if (!$this->db = dba_open($file, 'cd', $handler))
            throw new exception('could not open file ' . $file);
    }
    function __destruct() { dba_close($this->db); }
    function offsetExists($offset) {
        return dba_exists($offset, $this->db);
    }
    function offsetGet($offset) {
        return dba_fetch($offset, $this->db);
    }
    function offsetSet($offset, $value) {
        return dba_replace($offset, $value, $this->db);
    }
    function offsetUnset($offset) {
        return dba_delete($offset, $this->db);
    }
}
?>
```



A Trivial Example

```
<?php
    include_once 'TiedArray.php';
    $_SHARED = new Dbareader("/tmp/.counter", "flatfile");
    $_SHARED['counter'] += 1;
    printf("PID: %d\nCOUNTER: %d\n", getmypid(),
           $_SHARED['counter']);
?>
```



Iterators

- ☑ Normal objects behave like arrays when used with the `foreach` construct
- ☑ Specialized Iterator objects can be iterated differently

```
<?php
```

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

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

```
?>
```



What are Iterators

- ☑ Iterators are a concept to iterate anything that contains other things. Examples:
 - ☑ Values and Keys in an array
 - ☑ Text lines in a file
 - ☑ Database query results
 - ☑ Files in a directory
 - ☑ Elements or Attributes in XML
 - ☑ Bits in an image
 - ☑ Dates in a calendar range

- ☑ Iterators allow to encapsulate algorithms
 - ☑ Code re-use
 - ☑ Functional programming



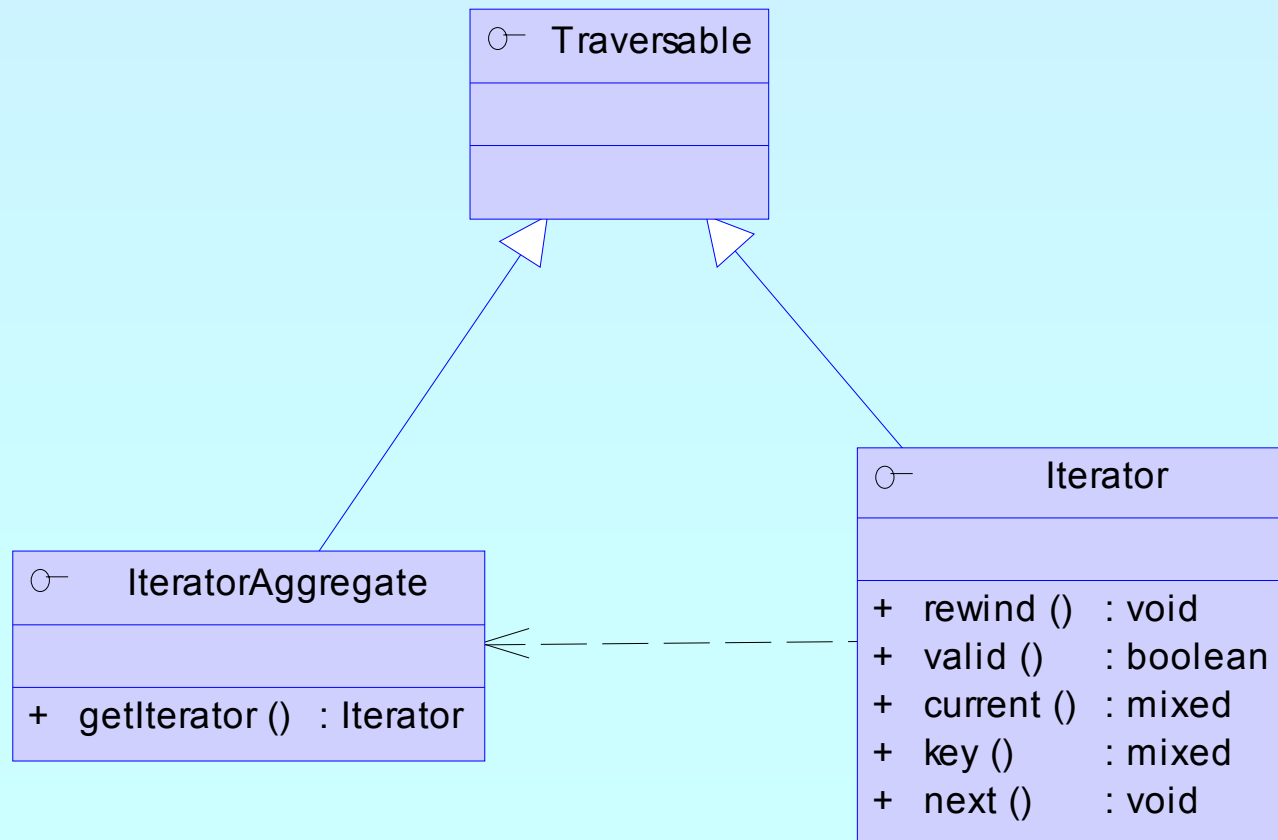
The basic Iterator concepts

- ☑ Iterators can be internal or external also referred to as active or passive
- ☑ An internal iterator modifies the object itself
- ☑ An external iterator points to another object without modifying it
- ☑ PHP always uses external iterators at engine-level

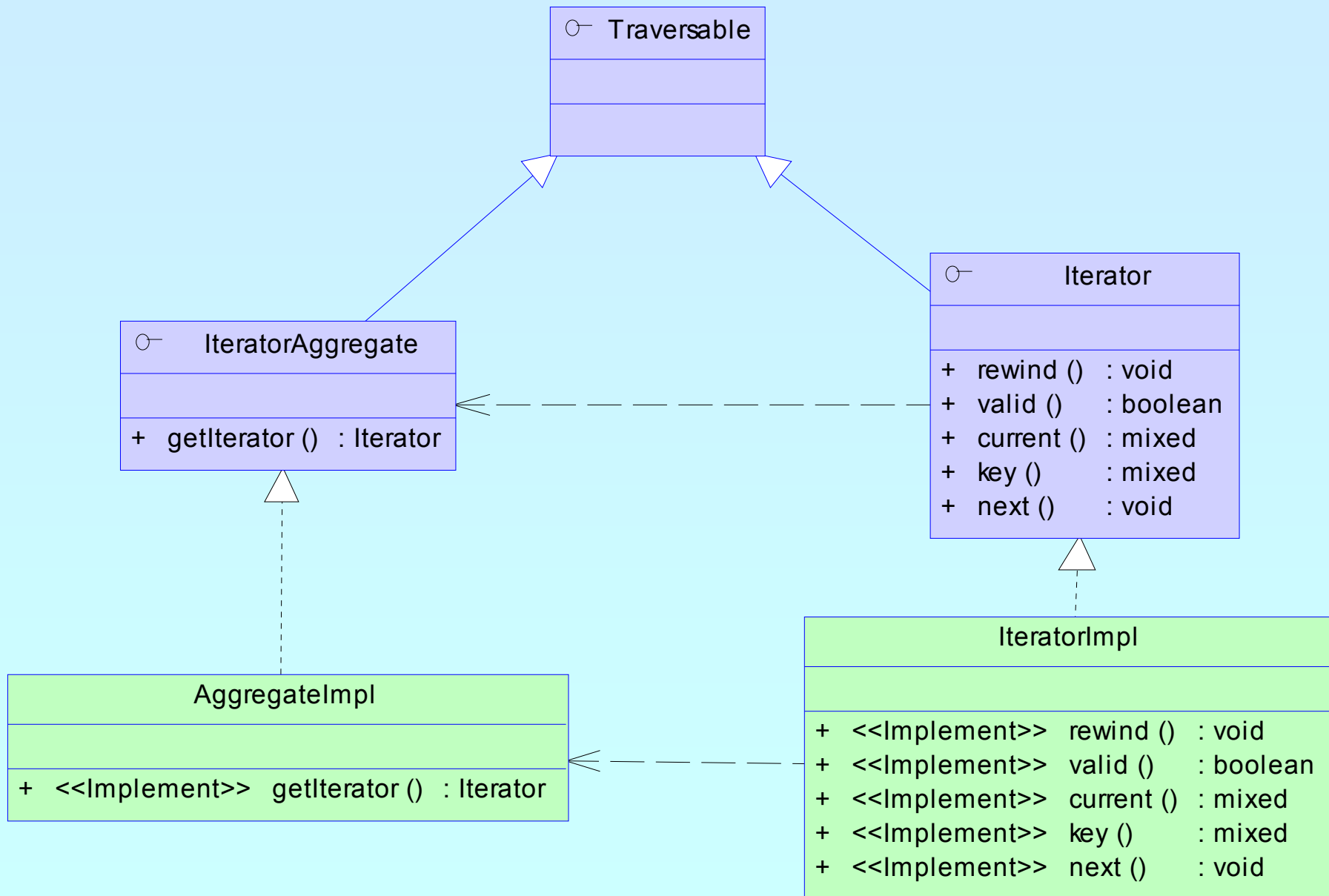


PHP Iterators

- ☑ Anything that can be iterated implements **Traversable**
- ☑ User classes cannot implement **Traversable**
- ☑ **Aggregate** is used for objects that use external iterators
- ☑ **Iterator** is used for internal traversal or external iterators



Implementing Iterators



How Iterators work

- ✓ Iterators can be used manually
- ✓ Iterators can be used implicitly with **foreach**

```
<?php
$o = new ArrayIterator(array(1, 2, 3));
$o->rewind();
while ($o->valid()) {
    $key = $o->key();
    $val = $o->current();
    // some code
    $o->next();
}
?>
```

```
<?php
$o = new ArrayIterator(array(1, 2, 3));
foreach($o as $key => $val) {
    // some code
}
?>
```



How Iterators work

- ☑ Internal Iterators
- ☑ User Iterators

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

```
<?php
class FilterIterator implements Iterator {
    function __construct(Iterator $input)...
    function rewind()...
    function accept()...
    function valid()...
    function get_resource()...
    function __construct($key=>$val) {
        // access data...
    }
?>
```

```
<?php
$it = get_resource();
for($it = new Filter($it, $filter); $it->valid(); $it->next()) {
    // access filtered data only $key = $it->key();
}
?>
```



Debug Session

```
<?php
class ArrayIterator {
    protected $ar;
    function __construct(Array $ar) {
        $this->ar = $ar;
    }
    function rewind() {
        rewind($this->ar);
    }
    function valid() {
        return !is_null(key($this->ar));
    }
    function key() {
        return key($this->ar);
    }
    function current() {
        return current($this->ar);
    }
    function next() {
        next($this->ar);
    }
}
?>
```

PHP 5.1



```
<?php
$a = array(1, 2, 3);
$o = new ArrayIterator($a);
foreach($o as $key => $val) {
    echo "$key => $val\n";
}
?>
```

```
0 => 1
1 => 2
2 => 3
```



Aren't Iterators Pointless in PHP?

- ☑ Why not just use arrays:

```
foreach($aggregate as $item) { /*...*/ }
```
- ☑ Aren't we making life more difficult than need be?
- ☑ No! For simple aggregations the above works fine (though it's slow), but not everything is an array.
What about:
 - ☑ Buffered result sets
 - ☑ Lazy Initialization
 - ☑ Directories
 - ☑ Anything not already an array



Iterators by example

- ☑ Using Iterators you can efficiently grab all groups from INI files

- ☑ The building blocks:
 - ☑ A class that handles INI files
 - ☑ An abstract filter Iterator
 - ☑ A filter that filters group names from the INI file input
 - ☑ An Iterator to read all entries in the INI file
 - ☑ Another filter that allow to search for specific groups



INI file abstraction

```
<?php
class Dbareader implements Iterator {
    protected $db = NULL;
    private $key = false, $val = false;

    function __construct($file, $handler) {
        if (!$this->db = dba_open($file, 'r', $handler))
            throw new exception('Could not open file ' . $file);
    }
    function __destruct() {
        dba_close($this->db);
    }
    function rewind() {
        $this->key = dba_firstkey($this->db);
        fetch_data();
    }
    function next() {
        $this->key = dba_nextkey($this->db);
        fetch_data();
    }
    private function fetch_data() {
        if ($this->key !== false)
            $this->val = dba_fetch($this->key, $this->db);
    }
    function current() { return $this->val; }
    function valid() { return $this->key !== false; }
    function key() { return $this->key; }
}
```



Filtering Iterator keys



FilterIteraor is an abstract class

- ✓ Abstract accept() is called from rewind() and next()
- ✓ When accept() returns false next() will be called automatically

```
<?php
class KeyFilter extends FilterIterator
{
    private $regex;

    function __construct(Iterator $it, $regex) {
        parent::__construct($it);
        $this->regex = $regex;
    }
    function accept() {
        return ereg($this->regex, $this->getInnerIterator()->key());
    }
    function getRegex() {
        return $this->regex;
    }
    protected function __clone($that) {
        // disallow clone
    }
}
?>
```



Getting only the groups

```
<?php
if (!class_exists('KeyFilter')) {
    require_once('keyfilter.inc');
}

class IniGroups extends KeyFilter {
    function __construct($file) {
        parent::__construct(
            new Dbareader($file, 'infile'), '^\\[.*\\]$');
    }
    function current() {
        return substr(parent::key(), 1, -1);
    }
    function key() {
        return substr(parent::key(), 1, -1);
    }
}
?>
```



Putting it to work

```
<?php
if (!class_exists('KeyFilter')) {
    require_once('keyfilter.inc');
}
if (!class_exists('IniGroups')) {
    require_once('inigroups.inc');
}

$it = new IniGroups($argv[1]);

if ($argc > 2) {
    $it = new KeyFilter($it, $argv[2]);
}

foreach($it as $group) {
    echo $group . "\n";
}

?>
```



Let's Talk About Patterns

- ☑ Patterns catalog solutions to categories of problems
- ☑ They consist of
 - ☑ A name
 - ☑ A description of their problem
 - ☑ A description of the solution
 - ☑ An assessment of the pros and cons of the pattern



What do patterns have to do with OOP?

- ☑ Not so much. Patterns sources outside OOP include:
- ☑ Architecture (the originator of the paradigm)
- ☑ User Interface Design (wizards, cookie crumbs, tabs)
- ☑ Cooking (braising, pickling)



Patterns We've Seen So Far

- ☑ Singleton Pattern
- ☑ Iterator Pattern



Aggregator Pattern

- ✓ Problem: You have collections of items that you operate on frequently with lots of repeated code.
- ✓ Remember our calendars:

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

- ✓ Solution: Create a container that implements the same interface, and performs the iteration for you.



Aggregator Pattern

```
✓ class EntryAggregate extends Entry {
    protected $entries;
    ...
    public function display() {
        foreach($this->entries as $entry) {
            $entry->display();
        }
    }
    public function add(Entry $e) {
        array_push($this->entries, $e);
    }
}
```

✓ By extending Entry, the aggregate can actually stand in any place that entry did, and can itself contain other aggregated collections.



Proxy Pattern

- ☑ Problem: You need to provide access to an object, but it has an interface you don't know at compile time.
- ☑ Solution: Use accessor/method overloading to dynamically dispatch methods to the object.
- ☑ Discussion: This is very typical of RPC-type facilities like SOAP where you can interface with the service by reading in a definitions file of some sort at runtime.



Proxy Pattern in PEAR SOAP

```
<?php
class SOAP_Client {
    public $wsdl;
    public function __construct($endpoint) {
        $this->wsdl = WSDLManager::get($endpoint);
    }
    public function __call($method, $args) {
        $port = $this->wsdl->getPortForOperation($method);
        $this->endpoint=$this->wsdl->getPortEndpoint($port);
        $request = SOAP_Envelope::request($this->wsdl);
        $request->addMethod($method, $args);
        $data = $request->saveXML();
        return SOAP_Envelope::parse($this->endpoint,$data);
    }
}
?>
```



Observer Pattern

- ☑ Problem: You want an object to automatically notify dependents when it is updated.
- ☑ Solution: Allow 'observer' to register themselves with the observable object.
- ☑ Discussion: An object may not apriori know who might be interested in it. The Observer pattern allows objects to register their interest and supply a notification method.



Observer Pattern

```
<?php
class Observable {
    protected $observers;
    public function attach(Observer $o) {
        array_push($this->observers, $o);
    }
    public function notify() {
        foreach($this->observers as $o) {
            $o->update();
        }
    }
}
interface Observer {
    public function update();
}
?>
```

☑ Concrete Examples: logging facilities: email, debugging, SOAP message notifications. NOT Apache request hooks.



New extensions



New extensions

| | |
|---|----------|
| <input checked="" type="checkbox"/> Date | PECL |
| <input checked="" type="checkbox"/> DOM | 5.0 |
| <input checked="" type="checkbox"/> FFI | PECL |
| <input checked="" type="checkbox"/> MySQLi | 5.0 |
| <input checked="" type="checkbox"/> PDO | PECL/5.1 |
| <input checked="" type="checkbox"/> PIMP | ? |
| <input checked="" type="checkbox"/> SimpleXML | 5.0 |
| <input checked="" type="checkbox"/> SPL | 5.0 |
| <input checked="" type="checkbox"/> SQLite | 5.0 |
| <input checked="" type="checkbox"/> Tidy | 5.0 |
| <input checked="" type="checkbox"/> XML + XSL | 5.0 |



Reference

- ☑ Everything about PHP
<http://php.net>

- ☑ These slides
<http://somabo.de/talks>

- ☑ SPL Documentaion & Examples
<http://php.net/~helly/php/ext/spl>
<http://cvs.php.net/php-src/ext/spl/examples>

- ☑ George's Book ([Advanced PHP Programming](#))

