Master OOP in PHP 5

Marcus Börger

PHP|tek 2006

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.
- ☑ 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?

- ☑ 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.
- ☑ 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. However if another person has the same laptop the internal details would be exactly the same.

Encapsulation: Are Objects Just Dictionaries?

- Classes as dictionaries are a common idiom:

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

A simple Inheritance Example

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

A better Inheritance Example

```
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 snorkel() { /*...*/ }
}
```

Inheritance and 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 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 Object Model

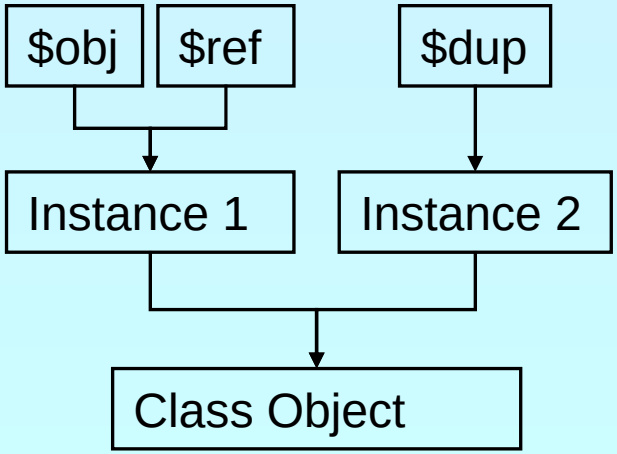
- ☑ PHP 5 has really good OO
  - ☑ Better code reuse
  - ☑ Better for team development
  - ☑ Easier to refactor
  - ☑ Some patterns lead to much more efficient code
  - ☑ Fits better in marketing scenarios

PHP 5 OOP in detail

Objects referenced by identifiers

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

```php
<?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

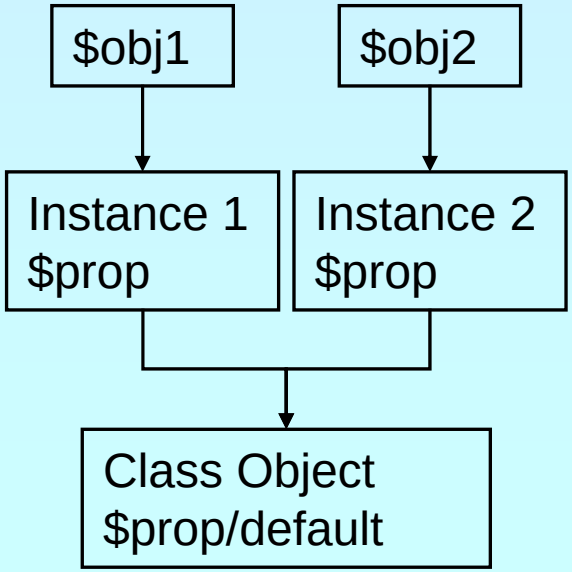
- Parents must be called manually

```
<?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

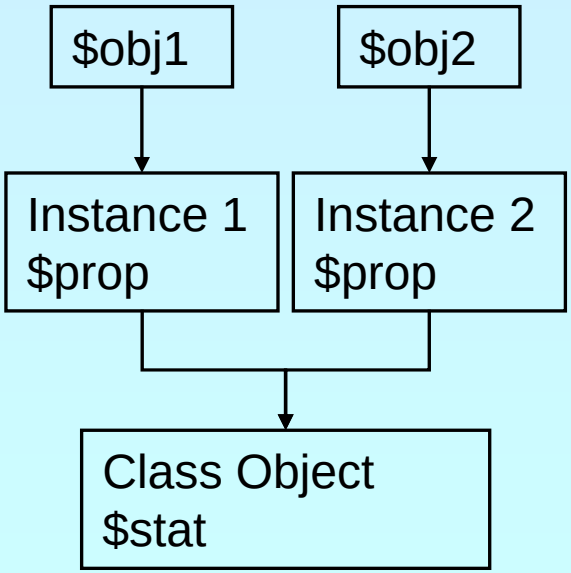
```
<?php

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
    - Only exists once per class rather than per instance
  - 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

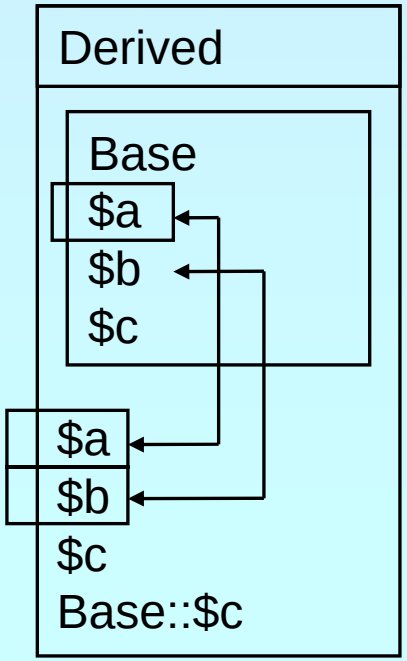
- ☑ **__CLASS__** shows the current class name
- ☑ **__METHOD__** shows class and method or function
- ☑ **self** references the class itself
- ☑ **parent** references the parent class
- ☑ **$this** references the object itself

```
<?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 doesn't know parents private members
  - 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

```
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

```
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;
   }
}
```

Clone visibility

- ☑ A protected __clone prevents external cloning
- ☑ A private final __clone prevents cloning

```
class Base {
   private final 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.

```
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

```
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

- 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
- Properties cannot be made abstract

```
abstract class Base {
  abstract function no_body();
}

class Derived extends Base {
  function no_body() { echo "Body\n"; }
}
```

Final members

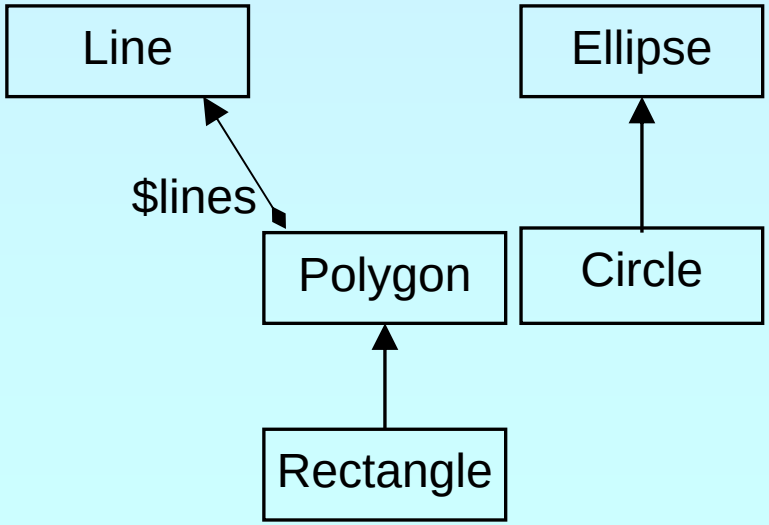
- Methods can be final
  - They cannot be overwritten
  - They are class invariants
- Classes can be final
  - They cannot be inherited

```
class Base {
  final function invariant() { echo "Hello\n"; }
}

class Derived extends Base {
}

final class Leaf extends Derived {
}
```

Different Object same behavior

- Often different objects have the same interface without having the same base class

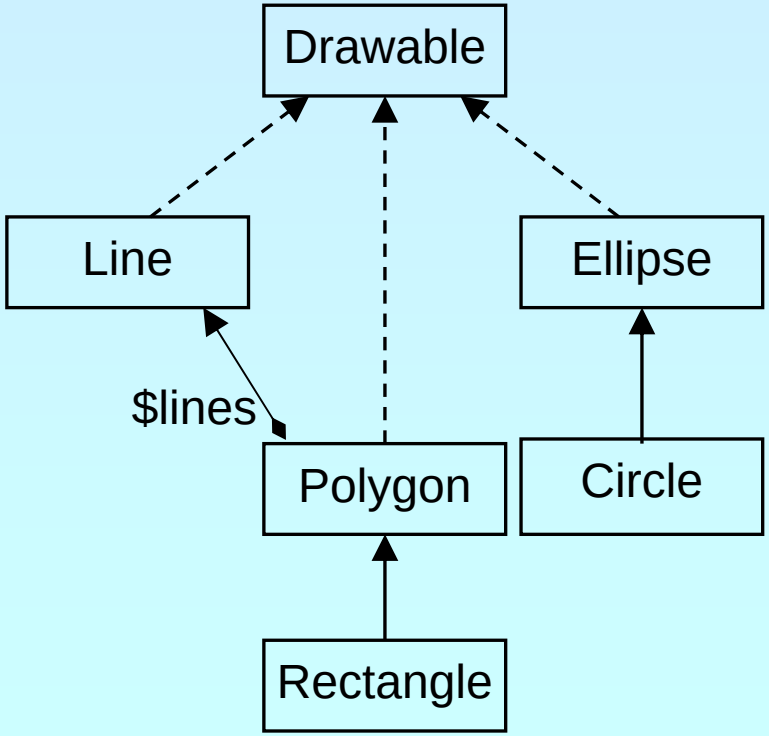
```
class Line {
   function draw() {};
}
class Polygon {
   protected $lines;
   function draw() {
       foreach($this->lines as $line)
            $line->draw();
   }
}
class Rectangle extends Polygon {
   function draw() {};
}
class Ellipse {
   function draw() {};
}
class Circle extends Ellipse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



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 (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
```

Interceptors

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

```
class Object {
   protected $virtual = array();
  function __get($name) {
      return @$this->virtual[$name];
   }
   function __set($name, $value) {
      $this->virtual[$name] = $value;
   }
   function __unset($name, $value) {
      unset($this->virtual[$name]);
   }
   function __isset($name, $value) {
      return isset($this->virtual[$name]);
   }
   function __call($func, $params) {
      echo 'Could not call ' . __CLASS__ . '::' . $func . "\n";
   }
}
```

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
      }
   }
}
```

Dynamic class loading

Dynamic class loading

☑ `__autoload()` is good when you're alone

- ☑ Requires a single file for each class
- ☑ Only load class files when necessary
  - ☑ No need to parse/compile unneeded classes
  - ☑ No need to check which class files to load

☒ Additional user space code

☠ Only one single loader model is possible

__autoload & require_once

- ☑ Store the class loader in an include file
  - ☑ In each script:
    require_once('<path>/autoload.inc')
  - ☑ Use INI option:
    auto_prepend_file=<path>/autoload.inc

```php
<?php
function __autoload($class_name)
{
   require_once(
      dirname(__FILE__) . '/' . $class_name . '.p5c');
}
?>
```

SPL's class loading

- ☑ Supports fast default implementation
  - ☑ Look into path's specified by INI option include_path
  - ☑ Look for specified file extensions (.inc, .php)
- ☑ Ability to register multiple user defined loaders
- ☑ Overwrites ZEND engine's __autoload() cache
  - ☑ You need to register __autoload if using spl's autoload

```
<?php
    spl_autoload_register('spl_autoload');
    if (function_exists('__autoload')) {
        spl_autoload_register('__autoload');
    }
?>
```

SPL's class loading

- ☑ `spl_autoload($class_name,$extensions=NULL)`
  - Load a class from in include path
  - Fast c code implementation
- ☑ `spl_autoload_extensions($extensions=NULL)`
  - Get or set filename extensions
- ☑ `spl_autoload_register($loader_function)`
  - Register a single loader function
- ☑ `spl_autoload_unregister($loader_function)`
  - Unregister a single loader function
- ☑ `spl_autoload_functions()`
  - List all registered loader functions
- ☑ `spl_autoload_call($class_name)`
  - Load a class through registered class loaders
  - Uses `spl_autoload()` as fallback

Exceptions

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
}
?>
```

Practial use of exceptions

- ☑ Constructor failure
- ☑ Converting errors/warnings to exceptions
- ☑ Simplify error handling
- ☑ Provide additional error information by tagging

Constructor failure

- ☑ In PHP 4.4 you would simply `unset($this)`
- ☑ Provide a param that receives the error condition

```
<?php
class Object
{
   function __construct( &$failure)
   {
      $failure = true;
   }
}
$error = false;
$o = new Object($error);
if (!$error) {
   // error handling, NOTE: the object was constructed
   unset($o);
}
?>
```

Constructor failure

- ☑ In 5 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
if (!class_exists(' ErrorException', false)) {
   class ErrorException extends Exception
   {
      protected $severity;
      function __construct($msg,$code,$errno,$file,$line)
      {
         parent::__construct($msg, $code);
         $this->severity = $errno;
         $this->file = $file;
         $this->line = $line;
      }
      function getSeverity() {
         return $this->severity;
      }
   }
}
?>
```

Convert Errors to Exceptions

☑ Implementing the error handler

```
<?php

function ErrorsToExceptions($errno, $msg, $file, $line)
{
  throw new ErrorException($msg, 0, $errno,$file,$line);
}

set_error_handler('ErrorsToExceptions');

?>
```

Simplify error handling

- Typical database access code contains lots of if's

```
<html><body>
<?php
$ok = false;
$db = new PDO('CONNECTION');
if ($db) {
   $res = $db->query('SELECT data');
   if ($res) {
      $res2 = $db->query('SELECT other');
      if ($res2) {
         // handle data
         $ok = true; // only if all went ok
      }
   }
}
if (!$ok) echo '<h1>Service currently unabvailable</h1>';
?>
</body></html>
```

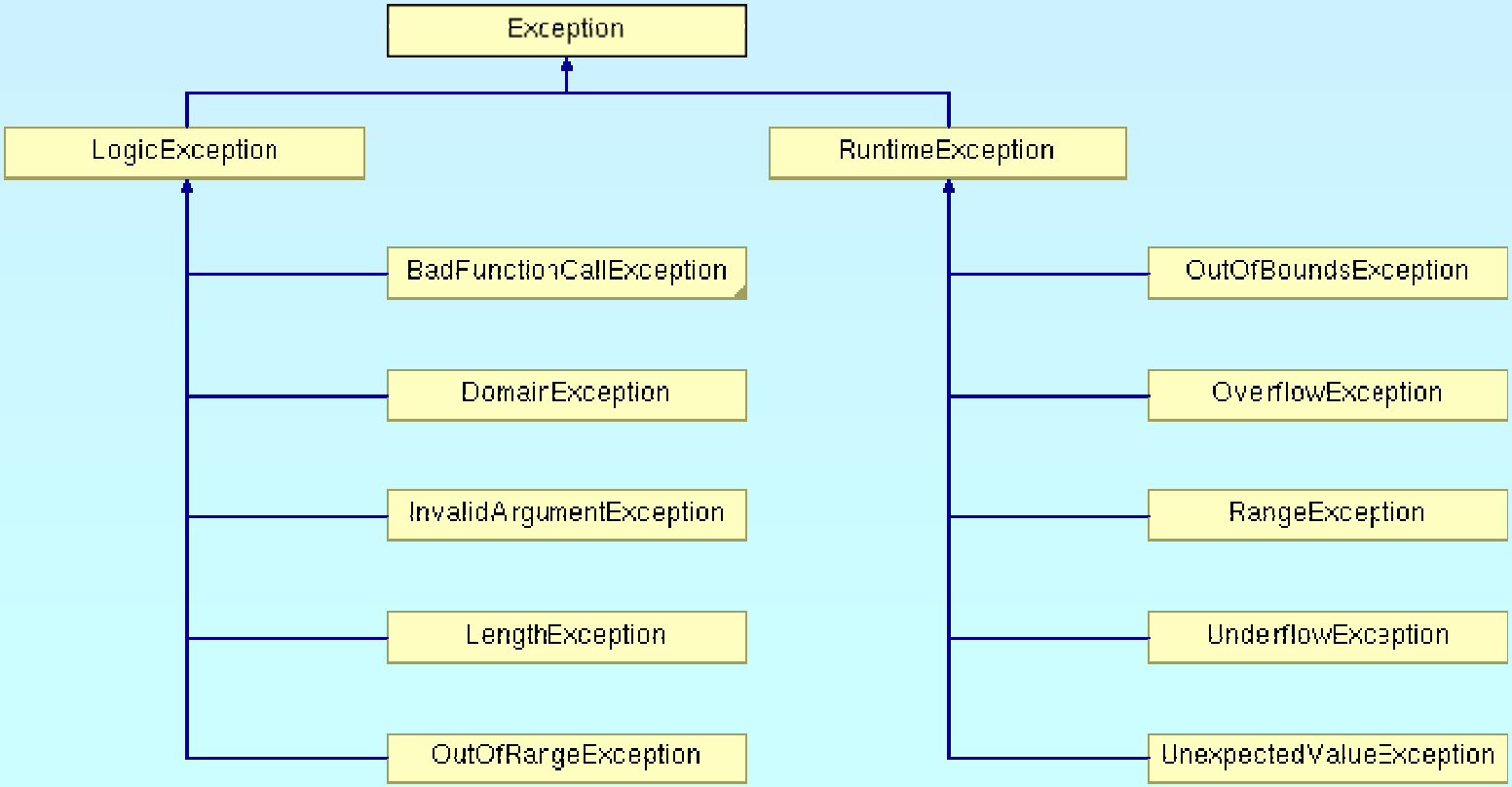
Simplify error handling

- ☑ Trade code simplicity with a new complexity

```
<html><body>
<?php
try {
   $db = new PDO('CONNECTION');
   $db->setAttribute(PDO::ATTR_ERRMODE,
                                PDO::ERRMODE_EXCEPTION);
   $res = $db->query('SELECT data');
   $res2 = $db->query('SELECT other');
   // handle data
}
catch (Exception $e) {
   echo '<h1>Service currently unabvailable</h1>';
   error_log($e->getMessage());
}
?>
</body></html>
```

SPL Exceptions

- ☑ SPL provides a standard set of exceptions
- ☑ Class Exception must be the root of all exceptions



General distinguishing

- ☑ `LogicException`

  ➔ Anything that could have been detected at compile time, during application design or by the good old technology:
  "look precisely"

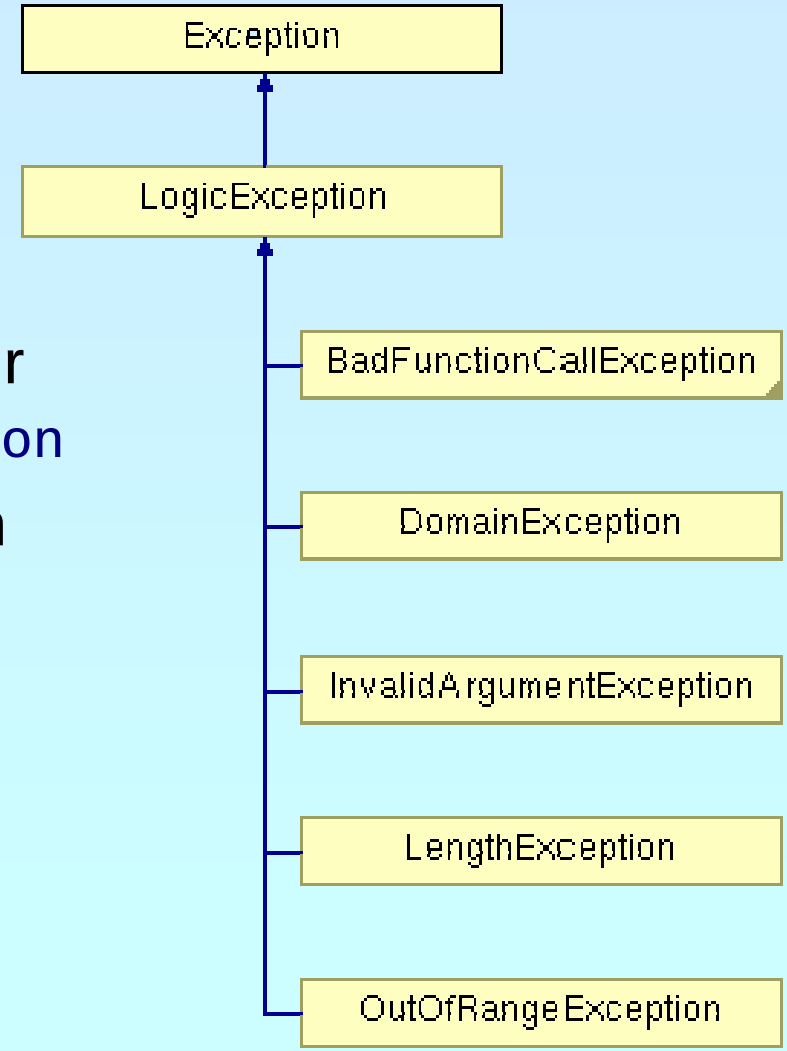
- ☑ `RuntimeException`

  ➔ Anything that is unexpected during runtime

  ➔ Base Exception for all database extensions

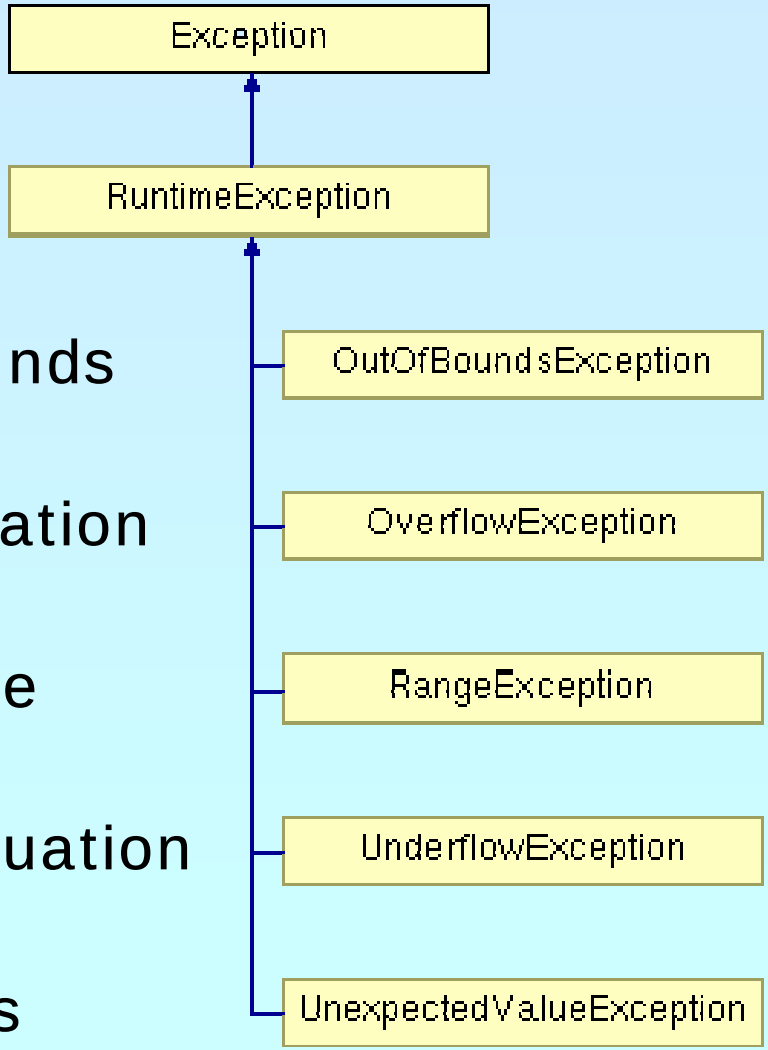
LogicException

- ☑ Function not found or similar
  `BadMethodCallException`
- ☑ Value not in allowed domain
- ☑ Argument not valid
- ☑ Length exceeded
- ☑ Some index is out of range



RunTimeException

- ☑ An actual value is out of bounds
- ☑ Buffer or other overflow situation
- ☑ Value outside expected range
- ☑ Buffer or other underflow situation
- ☑ Any other unexpected values



Overloading __call

☑ If using __call, ensure only valid calls are made

```
abstract class MyIteratorWrapper implements Iterator
{
   function __construct(Iterator $it)
   {
      $this->it = $it;
   }
   function __call($func, $args)
   {
      $callee = array($this->it, $func);
      if (!is_callable($callee)) {
         throw new BadMethodCallException();
      }
      return call_user_func_array($callee, $args);
   }
}
```

Run-Time:

data is different for every execution

Interfaces and __call

- ☑ Interface functions cannot be handled by __call
- ☑ Either mark the class abstract…

```
abstract class MyIteratorWrapper implements Iterator
{
   function __construct(Iterator $it)
   {
      $this->it = $it;
   }
   function __call($func, $args)
   {
      $callee = array($this->it, $func);
      if (!is_callable($callee)) {
         throw new BadMethodCallException();
      }
      return call_user_func_array($callee, $args);
   }
}
```

```
Interface Iterator {
   function rewind();
   function valid();
   function current();
   function key();
   function next();
}
```

Interfaces and __call

- ☑ Interface functions cannot be handled by __call
- ☑ ...or provide the functions (here as proxy/forward)

```
class MyIteratorWrapper implements Iterator
{
   function __construct(Iterator $it)
   {
      $this->it = $it;
   }
   function __call($func, $args)
   {
      $callee = array($this->it, $func);
      if (!is_callable($callee)) {
         throw new BadMethodCallException();
      }
      return call_user_func_array($callee, $args);
   }

   function rewind()  { $this->it->rewind(); }
   function valid()   { return $this->it->valid(); }
   function current() { return $this->it->current(); }
   function key()     { return $this->it->key(); }
   function next()    { $this->it->next(); }
}
```

```
Interface Iterator {
   function rewind();
   function valid();
   function current();
   function key();
   function next();
}
```

Expecting formatted data

- Opening a file for reading

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
```

Run-Time:

File might not be accessible or exist

Expecting formatted data

- ☑ Reading a formatted file line by line

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (/*** CHECK DATA ***/) {
      throw new Exception();
   }
   $data[] = $l;
}
```

Run-Time:

File might not be accessible or exist

Run-Time:

data is different for every execution

- ☑ `!preg_match($regex, $l)` — `UnexpectValueException`
- ☑ `count($l=split(',', $l)) != 3` — `RangeException`
- ☑ `count($data) > 100` — `OverflowException`

Expecting formatted data

- ☑ Cehcking data after pre-processing

Run-Time:

Filemight not be accessible or exist

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (!preg_match('/\d,\d/', $l)) {
      throw new UnexpectedValueException();
   }
   $data[] = $l;
}
// Checks after the file was read entirely
if (count($data) < 10) throw new UnderflowException();
if (count($data) > 99) throw new OverflowException();
if (count($data) < 10 || count($data) > 99)
                        throw new OutOfBoundsException();
```

Run-Time:

data is different for every execution

Expecting formatted data

☑ Processing pre-checked data

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (!preg_match('/\d,\d/', $l)) {
      throw new UnexpectedValueException();
   }
   $data[] = $l;
}
if (count($data) < 10) throw new UnderflowException();
// maybe more precessing code
foreach($data as &$v) {
   if (count($v) == 2) {
      throw new DomainException();
   }
   $v = $v[0] * $v[1];
}
```

Run-Time:

Filemight not be accessible or exist

Run-Time:

data is different for every execution

Compile-Time:

exception signales failed precondition

Reflection

Reflection API

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

```
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 allows dynamic object creation

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

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

Built-in Interfaces

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 --re SPL
php --rc ArrayAccess
```

ArrayAccess

- ☑ Allows for creating objects that can be transparently accessed by array syntax.
- ☑ When combined with the iterator interface, it allows for creating ‘arrays with special properties’.

```
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

- ☑ ArrayAccess does not allow references (the following is an error)

```
class MyArray extends ArrayAccess {
   function &offsetGet($offset) { /* ... */ }
   function offsetSet($offset, &$value) { /* ... */ }
   function offsetExists($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
  if (!class_exists('DbaReader', false)) {
        require_once ‘dbadeader.inc’;
  }
  $_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.
- ☑ Iterators allow to encapsulate algorithms

What are Iterators

- Iterators are a concept to iterate anything that contains other things. Examples:
  - Values and Keys in an array — `ArrayObject`, `ArrayIterator`
  - Text lines in a file — `SplFileObject`
  - Files in a directory — [`Recursive`]`DirectoryIterator`
  - XML Elements or Attributes — ext: SimpleXML, DOM
  - Database query results — ext: PDO, SQLite, MySQLi
  - Dates in a calendar range — PECL/date (?)
  - Bits in an image — ?
- Iterators allow to encapsulate algorithms

What are Iterators

- Iterators are a concept to iterate anything that contains other things. Examples:
  - Values and Keys in an array — `ArrayObject`, `ArrayIterator`
  - Text lines in a file — `SplFileObject`
  - Files in a directory — `[Recursive]DirectoryIterator`
  - XML Elements or Attributes — ext: SimpleXML, DOM
  - Database query results — ext: PDO, SQLite, MySQLi
  - Dates in a calendar range — PECL/date (?)
  - Bits in an image — ?

- Iterators allow to encapsulate algorithms
  - Classes and Interfaces provided by SPL:

    `AppendIterator`, `CachingIterator`, `LimitIterator`, `FilterIterator`, `EmptyIterator`, `InfiniteIterator`, `NoRewindIterator`, `OuterIterator`, `ParentIterator`, `RecursiveIterator`, `RecursiveIteratorIterator`, `SeekableIterator`, `SplFileObject`, . . .

Array vs. Iterator

- ☑ An array in PHP — `$ar = array()`
  - ☑ can be rewound: `reset($ar)`
  - ☑ is valid unless it's key is NULL: `!is_null(key($ar))`
  - ☑ have current values: `current($ar)`
  - ☑ have keys: `key($ar)`
  - ☑ can be forwarded: `next($ar)`

- ☑ Something that is traversable — `$it = new Iterator;`
  - ☑ **may** know how to be rewound: `$it->rewind()`
    `(does not return the element)`
  - ☑ should know if there is a value: `$it->valid()`
  - ☑ **may** have a current value: `$it->current()`
  - ☑ **may** have a key: `$it->key()`
    `(may return NULL at any time)`
  - ☑ can forward to its next element: `$it->next()`

The big difference

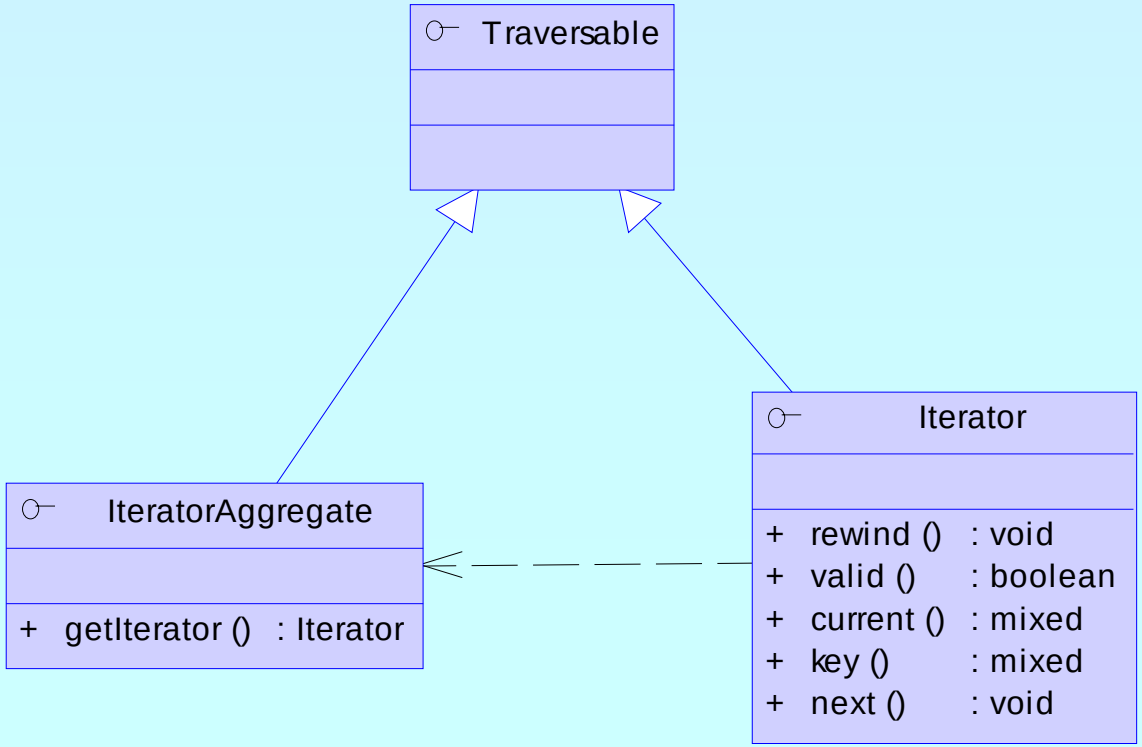
- **Arrays**
  - require memory for all elements
  - allow to access any element directly
- **Iterators**
  - only know one element at a time
  - only require memory for the current element
  - forward access only
  - Access done by method calls
- **Containers**
  - require memory for all elements
  - allow to access any element directly
  - can create external Iterators or are internal Iterators

The basic concepts

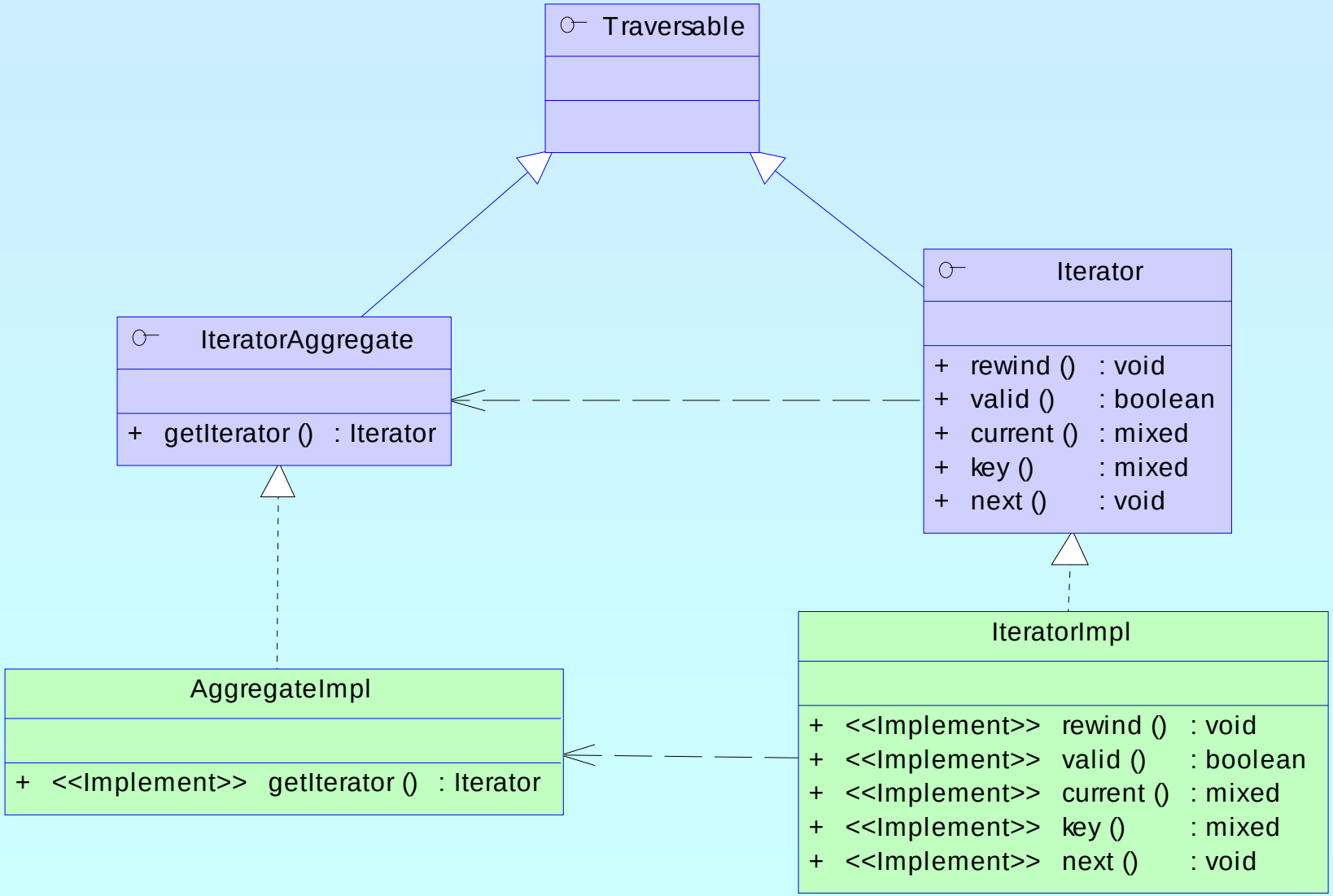
- ☑ Iterators can be internal or external
  also referred to as active or passive
- ☑ An internal iterator modifies the object itself
- ☑ An external iterator points to another object without modifying it
- ☑ PHP always uses external iterators at engine-level
- ☑ Iterators may iterate over other iterators

PHP Iterators

- ☑ Anything that can be iterated implements **`Traversable`**
- ☑ Objects implementing **`Traversable`** can be used in **`foreach`**
- ☑ User classes cannot implement **`Traversable`**
- ☑ **`IteratorAggregate`** is for objects that use external iterators
- ☑ **`Iterator`** is 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
<?php
interface Iterator {
  function rewind();
  function valid();
  function current();
  function key();
  function next();
}
?>
```

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

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

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

Debug Session

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

PHP 5.1

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

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

Aren't Iterators Pointless in PHP?

- Why not just use arrays:

```
foreach($some_array 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

```
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);
   }
   private function fetch_data($key) {
      if (($this->key = $key) !== false)
         $this->val = dba_fetch($this->key, $this->db);
   }
   function rewind() {
      $this->fetch_data(dba_firstkey($this->db));
   }
   function next() {
      $this->fetch_data(dba_nextkey($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 $rx;

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

Getting only the groups

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

class IniGroups extends KeyFilter {
   function __construct($file) {
      parent::__construct(
         new DbaReader($file,'inifile'),'^\[.*\]$');
   }
   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', false)) {
   require_once('keyfilter.inc');
}
if (!class_exists('IniGroups', false)) {
   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";
}

?>
```

Conclusion so far

- ☑ Iterators require a new way of programming
- ☑ Iterators allow to implement algorithms abstracted from data
- ☑ Iterators promote code reuse
- ☑ Some things are already in SPL
  - ☑ Filtering
  - ☑ Handling recursion
  - ☑ Limiting

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
- ☑ Factory 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 perfoms 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.

Object handling side notes

- ☑ You cannot access the object identifier/handle

```
$observers[] = $observer;
```

- ☑ YOU need to prevent double insertion/execution

```
foreach($observers as $o) {
   if ($o === $observer) return;
}
$observers[] = $observer;
```

- ☑ No easy way to delete an object from an array

```
foreach($observers as $k => $o) {
   if ($o === $observer) {
      unset($observer[$k]);
      break;
   }
}
```

Object Storage

```
class ObjectStorage {
   protected $storage = array();

   function attach($obj) {
      foreach($this->storage as $o) {
         if ($o === $obj) return;
      }
      $this->storage[] = $obj;
   }

   function detatch($obj) {
      foreach($this->storage as $k => $o) {
         if ($o === $obj) {
            unset($this->storage[$k]);
            return;
         }
      }
   }
}
```

Observer Pattern Implementation

```
class MySubject implements Subject {
   protected $observers;
   public function __construct() {
      $this->observer = new ObjectStorage;
   }
   public function attach(Observer $o) {
      $this->observers->attach($o);
   }
   public function detach(Observer $o) {
      $this->observers->detach($o);
   }
   public function notify() {
      foreach($this->observers as $o) $o->update($this);
   }
}
class MyObserver implements Observer {
   public function update(Subject $s) {
      // do logging or some other action
   }
}
```

- ☑ Concrete Examples: logging facilities: email, debugging, SOAP message notifications.

Reference

- ☑ Everything 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