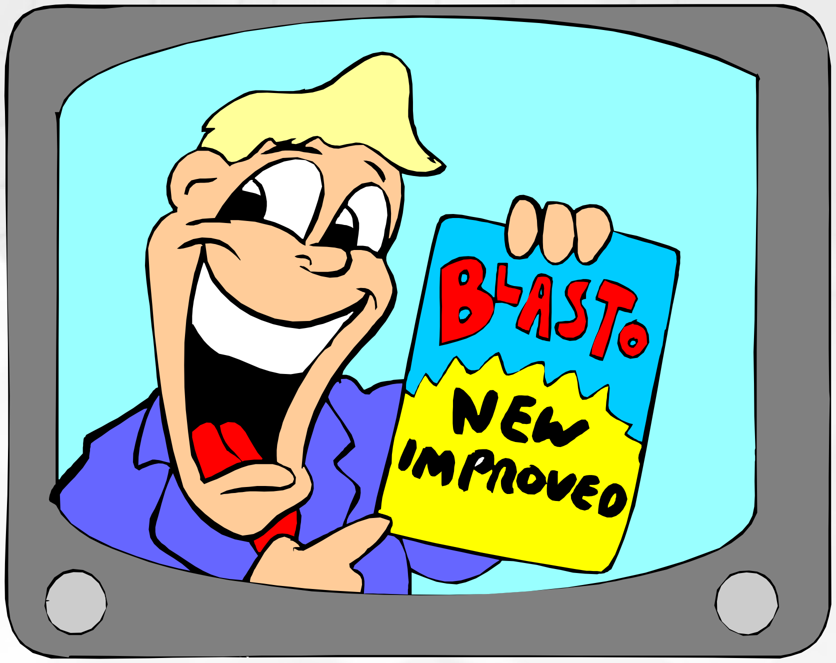
# PHP 5 OOP

By: Ilia Alshanetsky

# New Functionality

- Support for PPP
- Exceptions
- Object Iteration
- Object Cloning
- Interfaces
- Autoload
- And much more,
  and it is faster too!



## The Basics

- The basic object operations have not changed since PHP 4.

```
class my_obj {
    var $foo;
    function my_obj() { // constructor
        $this->foo = 123;
    }
    function static_method($a) {
        return urlencode($a);
    }
}
$a = new my_obj; // instantiate an object
$a->my_obj(); // method calls
my_obj::static_method(123); // static method call
```

## Similar, but not the same.

- While the syntax remains the same, internals are quite different.
- Objects are now always being passed by reference, rather then by value.
  - `PHP 5 $a = new foo(); == PHP4 $a = &new foo();`
- While the old style constructors are supported, new more consistent mechanism is available. `__construct()` method.

## PPP Annoyance

- The `VAR` keyword for identifying class properties became deprecated and will throw an `E_STRICT` warning.

```
PHP Strict Standards:  var: Deprecated. Please
use the public/private/protected modifiers in
obj.php on line 3.
```

- Instead, you should use `PUBLIC`, `PRIVATE` or `PROTECTED` keywords.

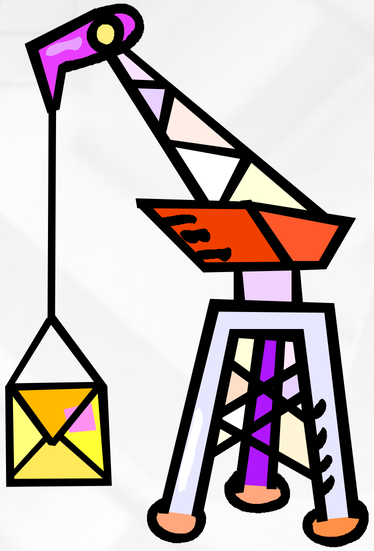
# PHP 5 Ready Code

```
<?php
class my_obj {
        public $foo;

        function __construct() { // constructor
                $this->foo = 123;
        }
        // static methods need to be declared as static
        // to prevent E_STRICT warning messages.
        static function static_method($a) {
                return urlencode($a);
        }
}

$a = new my_obj;
my_obj::static_method("a b");
?>
```

## PHP 5 Constructors

- In PHP 5 `parent::__construct` will automatically determine what parent constructor is available and call it.

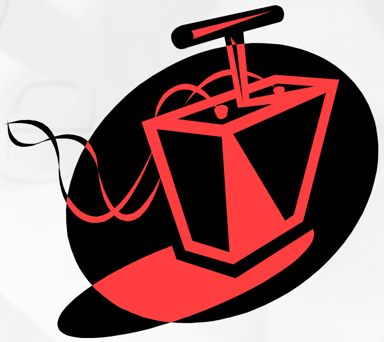
```
class main {
        function main() { echo "Main Class\n"; }
}
class child extends main {
        function __construct() {
                parent::__construct();
                echo "Child Class\n";
        }
}
$a = new child;
```



## Destructors

- Destructor methods specifies code to be executed on object de-initialization.

```
class fileio {
        private $fp;
        function __construct ($file) {
                $this->fp = fopen($file, "w");
        }
        function __destruct() {
                // force PHP to sync data in buffers to disk
                fflush($this->fp);
                fclose($this->fp);
        }
}
```



# Objects by Reference

- No matter how an object is passed in PHP 5+, you always work with the original.

```
function foo($obj) { $obj->foo = 1; }
$a = new StdClass; foo($a);
echo $a->foo; // will print 1

class foo2 {
        function __construct() {
                $GLOBALS['zoom'] = $this;
                $this->a = 1;
        }
}

$a = new foo2();
echo ($a->a == $zoom->a); // will print 1
```

# What If I Want a Copy?

- To copy an object in PHP 5 you need to make use of the `clone` keyword.
- This keyword does the job that `$obj2 = $obj;` did in PHP 4.

## Choices Choices Choices

- Being a keyword, clone supports a number of different, but equivalent syntaxes.

```
class A { public $foo; }

$a = new A;
$a_copy = clone $a;
$a_another_copy = clone($a);

$a->foo = 1; $a_copy->foo = 2; $a_another_copy->foo = 3;

echo $a->foo . $a_copy->foo . $a_another_copy->foo;
// will print 123
```

## Extending Clone

- `__clone()` can be extended to further modify the newly made copy.

```
class A {
        public $is_copy = FALSE;

        public function __clone() {
                $this->is_copy = TRUE;
        }
}
$a = new A;
$b = clone $a;
var_dump($a->is_copy, $b->is_copy); // false, true
```

## PPP

Like in other OO languages, you can now specify the visibility of object properties, for the purposes of restricting their accessibility.

- `PUBLIC` – Accessible to all.
- `PROTECTED` – Can be used internally and inside extending classes.
- `PRIVATE` – For class’ internal usage only.

# PPP in Practice

```
<?php
class sample {
        public $a = 1; private $b = 2; protected $c = 3;
        function __construct() {
                echo $this->a . $this->b . $this->c;
        }
}
class miniSample extends sample {
        function __construct() {
                echo $this->a . $this->b . $this->c;
        }
}
$a = new sample(); // will print 123
$b = new miniSample();
// will print 13 & notice about undefined property miniSample::$b
echo $a->a . $a->b . $a->c;
// fatal error, access to private/protected property
?>
```

# Practical PPP Applications

```
<?php
class registrationData {
        public $Login, $Fname, $Lname, $Address, $Country;
        protected $id, $session_id, $ACL;
}

$a = new registrationData();
foreach ($a as $k => $v) {
        if (isset($_POST[$k])) {
                $a->$k = $_POST[$k];
        }
}
?>
```

**Not all PHP functions/constructs respect, PPP visibility rules**

## Static Properties

- Another new feature of PHP 5 objects, is the ability to contain static properties.

```
<?php
class settings {
        static $login = 'ilia‵, $passwd = '123456';
}
echo settings::$login; // will print "ilia"
$a = new settings();
echo $a->login; // undefined property warning
$a->login = "Local Value"; // parse error? (NOPE!)
echo $a->login; // will print "Local Value"
?>
```

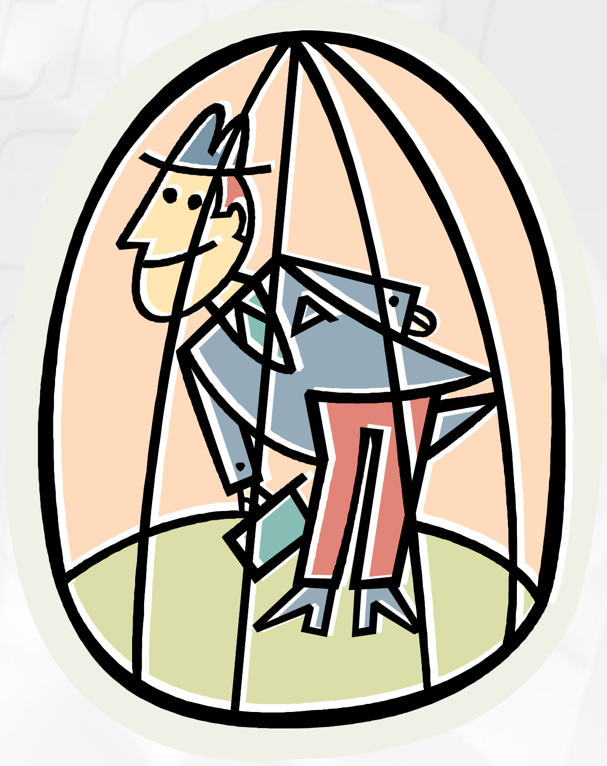
## Class Constants

- PHP 5 also supports class constants, which are very similar to static properties, however their values can never be altered.

```
class cc {
        const value = 'abc 123';

        function print_constant() {
                // access class constants inside of the class
                echo self::value;
        }
}
echo cc::value; // access class constants outside of the class
```

# PPP Applies to Methods Too!

- Method access can also be restricted via PPP.
  - Hide and prevent access to application's internal functionality.
  - Data separation.
  - Increased security.
  - Cleaner Code.



# Practical PPP Methods

```
class mysql {
   private $login, $pass, $host;
   protected $resource, $error, $qp;

   private function __construct() {
          $this->resource = mysql_connect($this->host,
                                      $this->login, $this->pass);
   }
   protected function exec_query($qry) {
         if (!($this->qp = mysql_query($qry, $this->resource))) {
                self::sqlError(mysql_error($this->resource));
         }
   }
   private static function sqlError($str) {
         open_log();
         write_to_error_log($str);
         close_log();
   }
}
```

## Practical PPP Methods

```
class database extends mysql {
        function __construct() {
                parent::__construct();
        }

        function insert($qry) {
                $this->exec_query($qry);
                return mysql_insert_id($this->resource);
        }

         function update($qry) {
                $this->exec_query($qry);
                return mysql_affected_rows($this->resource);
         }
}
```

# Final

- PHP 5 allows classed and methods to be defined a `FINAL`.
  - For methods it means that they cannot be overridden by a child class.
  - Classes defined as final cannot be extended.

## Final Method Example

- By making a method `FINAL` you prevent and extending classes from overriding it.  Can be used to prevent people from re-implementing your `PRIVATE` methods.

```
class main {
        function foo() {}
        final private function bar() {}
}

class child extends main {
        public function bar() {}
}

$a = new child();
```

## Final Class Example

- Classes declared as final cannot be extended.

```
final class main {
        function foo() {}
        function bar() {}
}
class child extends main { }
$a = new child();
```

```
PHP Fatal error:
   Class child may not inherit from final class (main)
```

## Autoload

- Maintaining class decencies in PHP 5 becomes trivial thanks to the `__autoload()` function.

```
<?php
function __autoload($class_name) {
        require_once "/php/classes/{$class_name}.inc.php";
}
$a = new Class1;
?>
```

- If defined, the function will be used to automatically load any needed class that is not yet defined.

# Magic Methods

- Objects in PHP 5 can have 3 magic methods.
  - __sleep() – that allows scope of object serialization to be limited. (not new to PHP)
  - __wakeup() – restore object's properties after deserialization.
  - __toString() – object to string conversion mechanism.

## Serialization

- Serialization is a process of converting a PHP variable to a specially encoded string that can then be used to recreate that variable.
- Needed for complex PHP types such as objects & arrays that cannot simply be written to a file or stored in a database.
- The serialization process is done via `serialize()` and restoration of data via `unserialize()` functions.

# Serialize Example

```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
                $this->bar = $this->foo * 10;
                $this->baz = ($this->bar + 3) / 2;
        }
}
$a = serialize(new test()); // encode instantiated class test
$b = unserialize($a); // restore the class into $b;
```

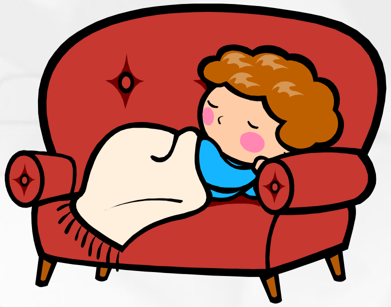
- The encoded version of our object looks like this:

```
O:4:"test":3:{s:3:"foo";i:1;s:3:"bar";i:10;s:3:"baz";d:6.5;}
```

# __sleep()

- The `__sleep()` method allows you to specify precisely which properties are to be serialized.

```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
                $this->bar = $this->foo * 10;
                $this->baz = ($this->bar + 3) / 2;
        }
        function __sleep() { return array('foo'); }
}
```

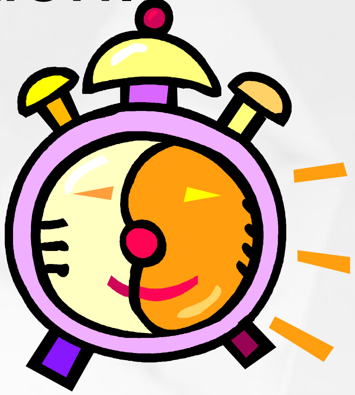


- This makes our serialized data more manageable.

```
O:4:"test":1:{s:3:"foo";i:1;}
```

## __wakeup()

- `__wakeup()`, if available will be called after deserialization. It’s job is to recreate properties skipped during serialization.



```
class test {
        public $foo = 1, $bar, $baz;
        function __construct() {
                $this->bar = $this->foo * 10;
                $this->baz = ($this->bar + 3) / 2;
        }
        function __wakeup() { self::__construct(); }
}
```

## __toString()

- Ever wonder how PHP extensions like `SimpleXML` are able to print objects and output valid data rather then garbage?

```
<?php
$xml =
   simplexml_load_string('<xml>
   <data>Ilia</data></xml>');
var_dump($xml->data);
echo $xml->data;
?>
```

**Output:**

```
object(SimpleXMLElement)#2 (1){
  [0]=> string(4) "Ilia"
}
Ilia
```

## Sample __toString()

```php
<?php
class sample {
        public $foo;

        function __construct() {
                $this->foo = rand();
        }

        function __toString() {
                return (string)$this->foo;
        }
}
echo new Sample();
?>
```

# __toString() Gotchas

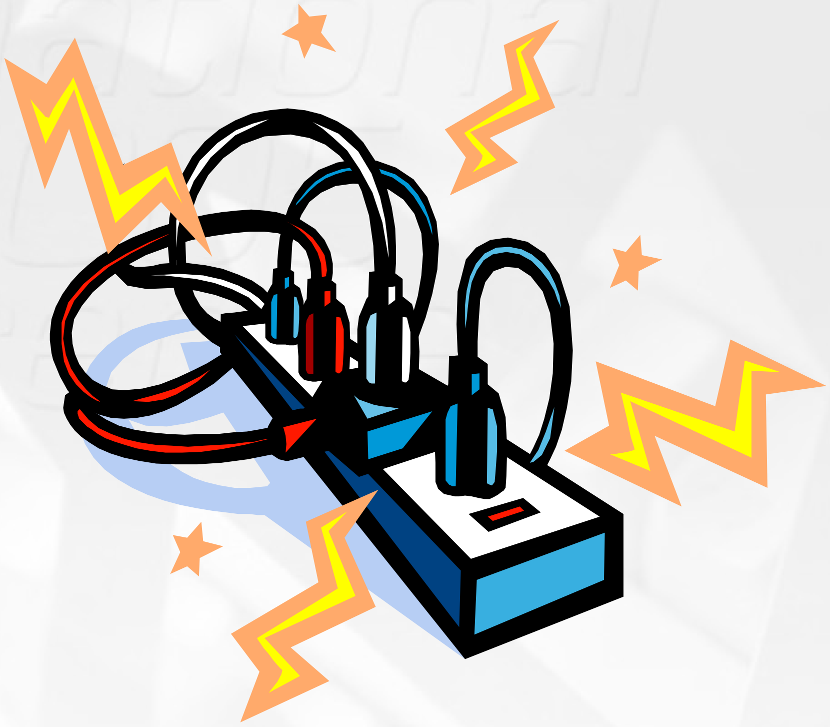
**`Assuming $a = new obj();`**

- `echo "str" . $a;`
- `echo "str {$a}"`
- `echo $a{0}; **`
- `echo (string) $a;`

In all of these instances `__toString()` will not be called.

## Overloading

- Both method calls and member accesses can be overloaded via the `__call`, `__get` and `__set` methods.
- Provide access mechanism to "virtual" properties and methods.



## Getter

- The getter method, `__get()` allows **read** access to virtual object properties.

```
class makePassword {
        function __get($name) {
                if ($name == 'md5')
                        return substr(md5(rand()), 0, 8);
                else if ($name == 'sha1')
                        return substr(sha1(rand()), 0, 8);
                else
                        exit("Invalid Property Name");
        }
}
$a = new makePassword();
var_dump($a->md5, $a->sha1);
```

## Setter

- The setter method, `__set()` allows **write** access to virtual object properties.

```
<?php
class userUpdate {
        public $user_id;

        function __construct() { db_cect() }


        function __set($name, $value) {
                db_update("UPDATE users SET {$name}='{$value}'
                                WHERE id={$user_id}");
        }
}
?>
```

# Dynamic Methods

- The `__call()` method in a class can be used to emulate any non-declared methods.

```
class math {
        function __call($name, $arg) {
                if (count($arg) > 2) return FALSE;
                switch ($name) {
                        case 'add':
                                return $arg[0] + $arg[1]; break;

                        case 'sub':
                                return $arg[0] - $arg[1]; break;
                        case 'div':
                                return $arg[0] / $arg[1]; break;
                }
        }
}
```

## Important Overloading Reminders

- The name passed to `__get`, `__set`, `__call` is not case normalized.

  `$foo->bar != $foo->BAR`
- Will only be called if the method/property does not exist inside the object.
- Functions used to retrieve object properties, methods will not work.
- Use with caution, it takes no effort at all to make code terribly confusing and impossible to debug.

## Object Abstraction

- Abstract classes allow you to create set methods describing the behavior of a to be written class.

# Database Abstraction

- The methods preceded by `abstract` keyword must be implemented by the extending classes.

```
abstract class database {
        public $errStr = '', $errNo = 0;

        // these methods must be provided by extending classes
        abstract protected function init($login,$pass,$host,$db);
        abstract protected function execQuery($qry);
        abstract protected function fetchRow($qryResource);
        abstract protected function disconnect();
        abstract protected function errorCode();
        abstract protected function errorNo();
}
```

# Abstract Implementer

```
class mysql extends database {
        private $c;
        protected function init($login, $pass, $host, $db) {
                $this->c = mysql_connect($host, $login, $pass);
                mysql_select_db($db, $this->c);
        }
        protected function execQuery($qry) {
                return mysql_query($qry, $this->c);
        }
        protected function fetchRow($res) {
                return mysql_fetch_assoc($res);
        }
  protected function errorCode() {return mysql_error($this->c); }
  protected function errorNo() { return mysql_errno($this->c); }
  protected function disconnect() { mysql_close($this->c); }
}
```

## Interfaces

- Object interfaces allows you to define a method “API” that the implementing classes must provide.

# Interface Examples

- Interfaces are highly useful for defining a standard API and ensuring all providers implement it fully.

```
interface webSafe {
        public function encode($str);
        public function decode($str);
}
interface sqlSafe {
        public function textEncode($str);
        public function binaryEncode($str);
}
```

## Implementer

- A class can implement multiple interfaces.

```
class safety Implements webSafe, sqlSafe {
        public function encode($str) {
                return htmlentities($str);
        }
        public function decode($str) {
                return html_entity_decode($str);
        }
        public function textEncode($str) {
                return pg_escape_string($str);
        }
        public function binaryEncode($str) {
                return pg_escape_bytea($str);
        }
}
```

## ArrayAccess Interface

- One of the native interface provided by PHP, allows object to emulate an array.
- The interface requires the following :
  - `offsetExists($key)` - determine if a value exists
  - `offsetGet($key)` - retrieve a value
  - `offsetSet($key, $value)` - assign value to a key
  - `offsetUnset($key)` - remove a specified value

# ArrayAccess in Action

```
class changePassword implements ArrayAccess {
        function offsetExists($id) {
                return $this->db_conn->isValidUserID($id);
        }
        function offsetGet($id) {
                return $this->db_conn->getRawPasswd($id);
        }
        function offsetSet($id, $passwd) {
                $this->db_conn->setPasswd($id, $passwd);
        }
        function offsetUnset($id) {
                $this->db_conn->resetPasswd($id);
        }
}
$pwd = new changePassword;
isset($pwd[123]); // check if user with an id 123 exists
echo $pwd[123]; // print the user's password
$pwd[123] = "pass"; // change user's password to "pass"
unset($pwd[123]); // reset user's password
```

## Object Iteration

- PHP 5 allows an object to implement an internal `iterator` interface that will specify exactly how an object is to be iterated through.

- To use it an object must implement the following methods:
  - `rewind`
  - `current`
  - `key`
  - `next`
  - `valid`

## File Iterator

```
class fileI Implements Iterator {
        private $fp, $line = NULL, $pos = 0;

        function __construct($path) {
                $this->fp = fopen($path, "r");
        }

        public function rewind() {
                rewind($this->fp);
        }

        public function current() {
                if ($this->line === NULL) {
                        $this->line = fgets($this->fp);
                }
                return $this->line;
        }
}
```

## File Iterator Cont.

```
public function key() {
        if ($this->line === NULL) {
                $this->line = fgets($this->fp);
        }
        if ($this->line === FALSE) return FALSE;
        return $this->pos;
}

public function next() {
        $this->line = fgets($this->fp);
        ++$this->pos;
        return $this->line;
}

public function valid() {
        return ($this->line !== FALSE);
}
```

# File Iterator Cont.

```
<?php
function __autoload($class_name) {
        require "./{$class_name}.php";
}
foreach (new fileI(__FILE__) as $k => $v) {
        echo "{$k} {$v}";
}
?>
```

**Output:**

```
0 <?php
1 function __autoload($class_name) {
2         require "./{$class_name}.php";
3 }
4 foreach (new fileI(__FILE__) as $k => $v) {
5         echo "{$k} {$v}";
6 }
7 ?>
```

# Exceptions

- Exceptions are intended as a tool for unifying error handling.
- An entire block of code can be encompassed inside a `try {}` block.
- Any errors, are then sent to the `catch {}` for processing.

# Native Exception Class

```php
class Exception
{
  protected $message = 'Unknown exception';  // exception message
  protected $code = 0; // user defined exception code
  protected $file; // source filename of exception
  protected $line; // source line of exception

  function __construct($message = null, $code = 0);

  final function getMessage(); // message of exception
  final function getCode(); // code of exception
  final function getFile(); // source filename
  final function getLine(); // source line
  final function getTrace(); // backtrace array
  final function getTraceAsString(); // trace as a string

  function __toString(); // formatted string for display
}
```

# Exception Example

```
<?php
try {
        $fp = fopen("m:/file", "w");
        if (!$fp) {
                throw new Exception("Cannot open file.");

        if (fwrite($fp, "abc") != 3)
                throw new Exception("Failed to write data.");

        if (!fclose($fp))
                throw new Exception("Cannot close file.");
} catch (Exception $e) {
        printf("Error on %s:%d %s\n",
                $e->getFile(), $e->getLine(), $e->getMessage());
        exit;
}
?>
```

# Extending Exceptions

```
class iliaException extends Exception {
        public function __construct() {
                parent::__construct($GLOBALS['php_errormsg']);
        }
        public function __toString() {
                return sprintf("Error on [%s:%d]: %s\n",
                       $this->file, $this->line, $this->message);
        }
}

ini_set("track_errors", 1); error_reporting(0);
try {
        $fp = fopen("m:/file", "w");
        if (!$fp) throw new iliaException;
        if (fwrite($fp, "abc") != 3) throw new iliaException;
        if (!fclose($fp)) throw new iliaException;
} catch (iliaException $e) { echo $e; }
```

## Stacking & Alternating Exceptions

```
<?php
try {
  // will go into $try1
  try {
    // will go into $try2
    } catch (Exception $try2) {

    }
   // will go into $try1
} catch (Exception $try1) {

}
?>
```

```
<?php
try {
        $a = new dbConnection();
        $a->execQuery();
        $a->fetchData();
} catch (ConnectException $db) {

} catch (QueryException $qry) {

} catch (fetchException $dt) {

}
?>
```

- PHP Exceptions can be stackable or alternate based on the exception name.

# Exception Handler

- The exception handler function, `set_exception_handler()` allows exceptions to be handled without explicitly listing the `try {} catch () {}` block.

```
function exHndl($e) {
  trigger_error($e->getLine());
}

set_exception_handler('exHndl');

$fp = fopen("m:/file", "w");
if (!$fp)
   throw new iliaException;
if (fwrite($fp, "abc") != 3)
   throw new iliaException;
if (!fclose($fp))
   throw new iliaException;
```

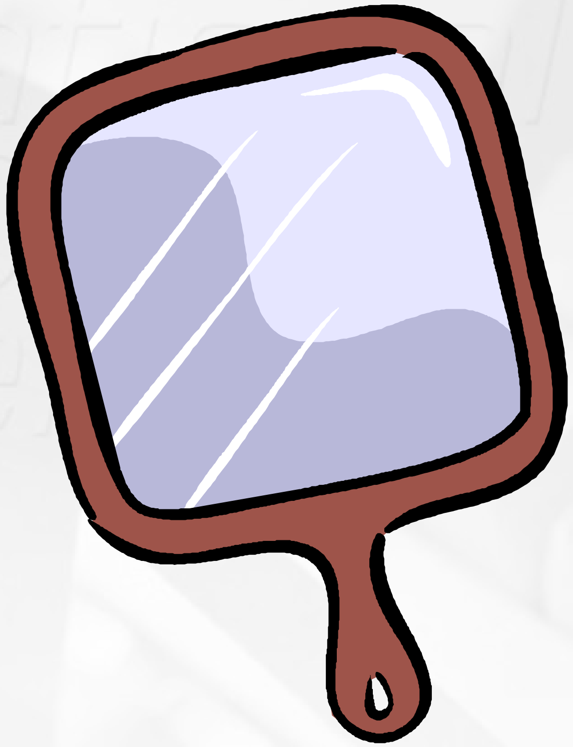
# Type Hinting

- While PHP is still type insensitive, you can now specify what type of objects your functions and methods require.

```php
<?php
class Foo {}
function useFoo(Foo $obj) { /* ... */ }

$a = new Foo;
useFoo($a); // works

$b = new StdClass;
useFoo($b);
// Fatal error: Argument 1 must be an instance of Foo
?>
```

## Reflection API

- Reflection API provides a mechanism for obtaining detailed information about functions, methods, classes and exceptions.
- It also offers ways of retrieving doc comments for functions, classes and methods.



## Reflection Mechanisms

- The API provides a distinct class for study of different entities that can be analyzed.

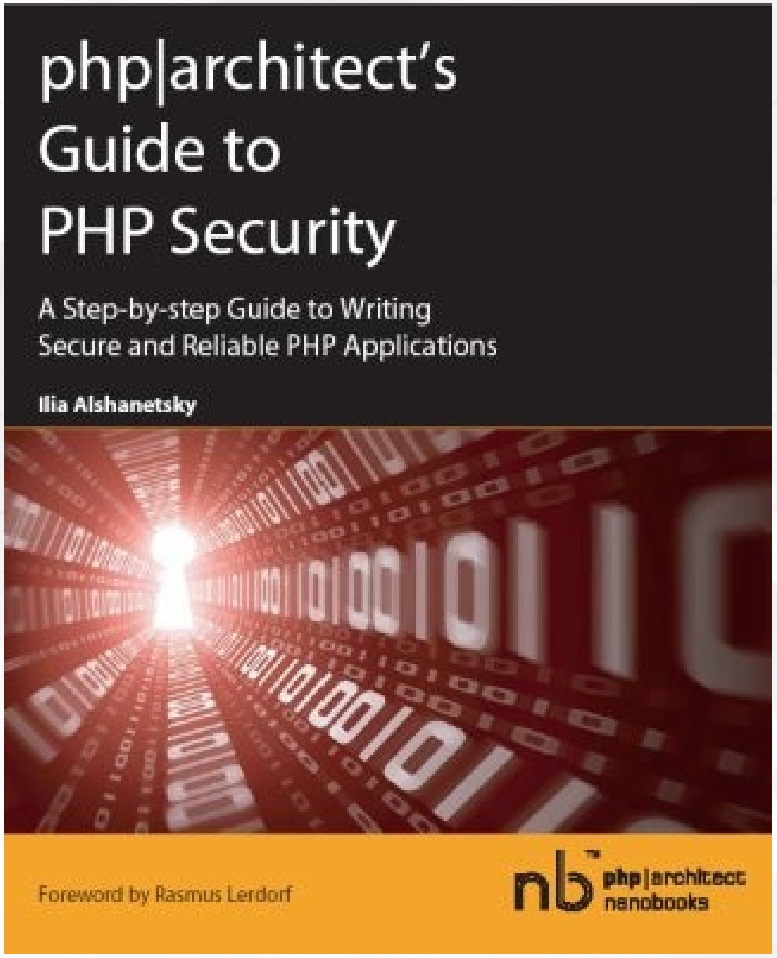
```
<?php
        class Reflection { }
        interface Reflector { }
        class ReflectionException extends Exception { }
        class ReflectionFunction implements Reflector { }
        class ReflectionParameter implements Reflector { }
        class ReflectionMethod extends ReflectionFunction { }
        class ReflectionClass implements Reflector { }
        class ReflectionObject extends ReflectionClass { }
        class ReflectionProperty implements Reflector { }
        class ReflectionExtension implements Reflector { }
?>
```

## Questions

## Resources

- http://ilia.ws/ (Slides will be available here)
- http://ca.php.net/oop5 (PHP5 OOP Docs)

## <?php include “/book/plug.inc”; ?>



php|architect's
Guide to
PHP Security

A Step-by-step Guide to Writing
Secure and Reliable PHP Applications

Ilia Alshanetsky

Foreword by Rasmus Lerdorf

nb php|architect nanobooks