

# Object Oriented Programming

# Introduction

- ❖ PHP support for object oriented programming (OOP) is one of the major changes introduced by PHP 5.

# Declaring a Class

- ❖ The basic syntax used when declaring a class is:

```
class [class name]
{
    ...
}
```

Example:

```
class Rectangle
{
    ...
}
```

# Class Instantiation

- ❖ Instantiating a class is done by using the `new` construct.

```
$var = new [class name]();
```

Example:

```
$myRectangle = new Rectangle();
```

# Object by Reference

- ❖ Starting with PHP 5, an object is always treated using its reference rather than its value.

Example:

```
$rec1 = new Rectangle();
```

```
$rec2 = $rec1;
```

Both `$rec1` and `$rec2` point to the same object. Both `$rec1` and `$rec2` hold the same reference for the same object.

# The '->' Operator

- ❖ Calling a method on a specific object is done using the '->' operator.

```
class Xyz
{
    function foo() { echo "xyz"; }
}
```

```
$xyz = new Xyz();
```

```
$xyz -> foo();
```

# The '->' Operator

- ❖ Calling a method from within another method (on the same object) should be done using `'$this'`. Unlike Java, C++ and C# PHP doesn't allow calling another method without using the `'$this'` keyword.

# The '->' Operator

- ❖ Accessing a variable within an object is done using the '->' operator.

```
class Xyz
{
    var $num;
}
```

```
$xyz = new Xyz();
$xyz -> num = 9;
```



# Constructor

- ❖ A constructor is a special function called when the class is instantiated.
- ❖ The constructor name should be either `__construct` or a name identical to the name of the class.

# Constructor

```
<?php
```

```
class Rectangle
{
    var $width;
    var $height;
    function Rectangle($numA, $numB)
    {
        $this->width=$numA;
        $this->height=$numB;
    }
    function area() { return $this->width*$this->height; }
}
```

# Constructor

```
$obj = new Rectangle(5,2)  
echo $obj->area();
```

?>

# Destructor

- ❖ The destructor is a special function called when the object ends its life.
- ❖ We can place within the destructor commands to free resources the object used.
- ❖ The destructor name must be `__destruct`.

# Destructor

```
<?php

class Rectangle
{
    var $width;
    var $height;
    function __destruct()
    {
        ...
    }
}

?>
```

# The `$this` Keyword

- ❖ Within the scope of every method we can refer the current object using `$this`.
- ❖ Trying to access object's variables should be done using the `$this` keyword and the arrow `->` operator. When doing so there is no need to specify `$` before the variable name.

...

```
function setWidth($val)
{
    $this->width = $val;
}
```

...

# The Scope

- ❖ PHP 5 allows us defining each one of the class variables and each one of the class functions with a scope:

`public` ...can be accessed from any scope (default).

`protected` ...can be accessed from within the class and its descendants.

`private` ...can be accessed from within the class only.

# The Scope

```
<?php
class Rectangle
{
    private $width;
    private $height;
    function __construct($wval, $hval)
    {
        $this->set_width($wval);
        $this->set_height($hval);
    }
    function area()
    {
        return $this->width*$this->height;
    }
}
```



# The Scope

```
function set_width($val)
{
    if($val>0)
    {
        $this->width = $val;
    }
}
function set_height($val)
{
    if($val>0)
    {
        $this->height = $val;
    }
}
```

# The Scope

```
function details()  
{  
    echo "width=";  
    echo $this->width;  
    echo "<BR>";  
    echo "height=";  
    echo $this->height;  
    echo "<BR>";  
    echo "area=";  
    echo $this->area();  
}  
  
}  
  
$rec = new Rectangle(5, 3);  
$rec->details();
```

?>

# Inheritance

- ❖ Declaring a class that extends another is done using the `extends` construct.

```
class Aaa
{
    ...
}
class Bbb extends Aaa
{
    ...
}
```

# Overriding Methods

- ❖ Declaring a class extending another allows adding new method and new properties as well as declaring methods that already exist (overriding).

```
class Aaa
{
    function doSomething() {echo "Aaa something";}
}

class Bbb extends Aaa
{
    function doSomething() {echo "Bbb something";}
}
```

# Overriding Methods

```
<?php
class Aaa
{
    function doSomething()
    {
        echo "a something";
    }
}
class Bbb extends Aaa
{
    function doSomething()
    {
        echo "b something";
    }
}
$obj_b = new Bbb();
$obj_b->doSomething();
?>
```

# The 'parent::' Construct

- ❖ Using the 'parent::' construct it is possible to access the parent class' method version.

```
class Aaa
{
    function doSomething() {echo "Aaa something";}
}

class Bbb extends Aaa
{
    function doSomething() {parent::doSomething(); echo "Bbb";}
}
```

# The 'parent::' Construct

```
<?php
class Aaa
{
    function doSomething()
    {
        echo "a something";
    }
}
class Bbb extends Aaa
{
    function doSomething()
    {
        parent::doSomething();
    }
}
$obj_b = new Bbb();
$obj_b->doSomething();
?>
```

# The 'final' Keyword

- ❖ Adding 'final' to our class definition will ensure that we won't be able to extend that class.
- ❖ Adding 'final' to our method definition will ensure that we won't be able to override that method.



# The 'final' Keyword

```
<?php
class Person
{
    private $name;
    private $id;

    function Person($name_val,$id_val)
    {
        $this->name = $name_val;
        $this->$id = $id_val;
    }

    final function set_id($val)
    {
        if($val>0 && $val<1000)
        {
            $this->id = $val;
        }
    }
}
?>
```

# The 'final' Keyword

```
<?php
final class Person
{
    private $name;
    private $id;

    function Person($name_val,$id_val)
    {
        $this->name = $name_val;
        $this->id = $id_val;
    }

    function set_id($val)
    {
        if($val>0 && $val<1000)
        {
            $this->id = $val;
        }
    }
}
?>
```

# Properties Initialization

- ❖ When defining a class property (variable) it is possible to initialize it with a value.
- ❖ That value can not be an expression.

```
class Circle
{
    var $radius = 8;
    function details()
    {
        echo "radius=";
        echo $this->radius;
    }
}
```

# Static Methods & Static Properties

- ❖ Adding the 'static' keyword to our method / variable definition will turn it into a static one.
- ❖ Static method doesn't work on a specific object.
- ❖ Static variable is not duplicated for each one of the instantiated objects.
- ❖ Calling a static method should be done by writing the class name following “: :” preceding the static method we call.

# Static Methods & Static Properties

```
<?php
class Utils
{
    static $interest = 5.4;
    static function sum($a,$b)
    {
        return $a+$b;
    }
    static function multiply($a,$b)
    {
        return $a*$b;
    }
}
echo Utils::sum(4,5);
echo "<BR>";
echo Utils::multiply(4,5);
echo "<BR>";
echo Utils::$interest;
?>
```

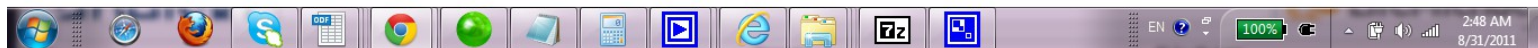


# Static Methods & Static Properties



9  
20  
5.4

abelski



# Class Constants

- ❖ Class constants are constants (as any other constant) except for the fact they are scoped within a class.

```
class [class name]
{
    const [constant name] = [constant value];
}
```

```
class Something
{
    const SCHOOL_NAME = "De Shalit";
}
```

# Class Constants

- ❖ Accessing a class constant is done by writing the class name + “::” preceding the constant name.

```
echo [class name]::[constant name];
```

```
echo Something::SCHOOL_NAME;
```



# Abstract Class

- ❖ Adding the 'abstract' keyword to the class definition and include within that class the definition for one (or more) abstract methods will turn that class into an abstract one.
- ❖ An abstract method is a method with the 'abstract' keyword in its declaration and without a body.
- ❖ It is impossible to instantiate an abstract class. There is a need to extend it and override each one of the abstract methods in order to be able to instantiate the new class.

# Abstract Class

- ❖ When we define an abstract method we cannot use the private access modifier. It is impossible to define a private abstract method.

# Abstract Class

```
<?php
```

```
abstract class Shape
{
    abstract function area();
}

class Rectangle extends Shape
{
    private $width;
    private $height;
    public function __construct($wval,$hval)
    {
        $this->width = $wval;
        $this->height = $hval;
    }
    public function area()
    {
        return $this->width * $this->height;
    }
}
```



# Abstract Class

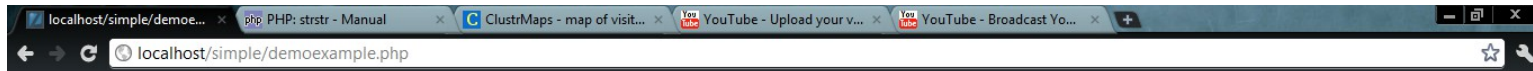
```
class Circle extends Shape
{
    private $radius;
    public function __construct($num)
    {
        $this->radius = $num;
    }
    public function area()
    {
        return $this->radius * $this->radius * 3.14;
    }
}
```

```
$rec = new Rectangle(5,2);
echo $rec->area();
echo "<BR>";
```

```
$circ = new Circle(4);
echo $circ->area();
echo "<BR>";
```

```
?>
```

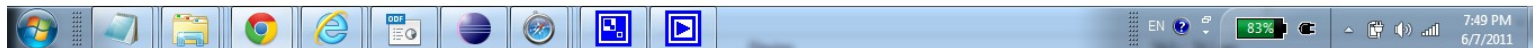
# Abstract Class



10  
50.24

abelski

 LifeMichael  
Haim Michael Blog



# Interface

- ❖ We use the `'interface'` keyword in order to define an interface. Similarly to defining a class. The differences are:
  1. Instead of using the `'class'` keyword we use `'interface'`.
  2. Within the interface we can define abstract methods only.
  3. Within the interface we cannot define neither a constructor or a destructor.
- ❖ We can define a class and mention that it implements an interface. To do so, we use the `'implements'` keyword.
- ❖ Interfaces can be used to abstract the behavior of an expected component.

# Interface

- ❖ Unlike a class that can extend one other class only, it is possible to define a class that implements more than one interface. We should write the names of each one of the interfaces separated with a commas.

```
class Something implements Driveable, Cloneable, Printable
{
    .
    .
    .
}
```

# Interface

```
interface Printable
{
    function print_details();
}

class Rectangle implements Printable
{
    private $width;
    private $height;
    public function Rectangle($w_val,$h_val)
    {
        $this->width = $w_val;
        $this->height = $h_val;
    }
    public function print_details()
    {
        echo "rectangle... width=".$this->width." height=".$this->height;
    }
}
```



# Interface

```
class Person implements Printable
{
    private $name;
    private $id;
    public function Person($n_val,$id_val)
    {
        $this->name = $n_val;
        $this->id = $id_val;
    }
    public function print_details()
    {
        echo "person... name=$this->name id=$this->id";
    }
}
```

# Interface

```
class Car implements Printable
{
    private $name;
    private $id;
    public function Car($n_val,$id_val)
    {
        $this->name = $n_val;
        $this->id = $id_val;
    }
    public function print_details()
    {
        echo "car... name=$this->name id=$this->id";
    }
}
```

# Interface

```
$vec = array();  
$vec[0] = new Car("Toyota",233423);  
$vec[1] = new Car("Ford",2435434);  
$vec[2] = new Rectangle(8,4);  
$vec[3] = new Rectangle(10,8);  
$vec[4] = new Person("John",46354);  
$vec[5] = new Person("Moshe",463445);  
  
foreach($vec as $ob)  
{  
    $ob->print_details();  
    echo "<br>";  
}
```

# The 'instanceof' Operator

- ❖ Using the 'instanceof' operator it is possible to determine whether a given object is an instance of a specific class or of a class that implements a specific interface.

```
if ([object variable] instanceof [class or interface name])  
{  
    ...  
}
```

The value of this boolean expression is true if one of the following is true:

- (1) The object was instantiated from a class that implements the specified interface.
- (2) The object was instantiated from the specified class.
- (3) The object was instantiated from a class that extends the specified class

# Objects Serialization

- ❖ Similarly to Java, PHP allows us to serialize objects into a storable representation.

We can later store it into a file or send it over the network to another application.

- ❖ The `serialize` function receives an object and returns its storable representation.

...

```
$obj = new Car();
```

```
$obj_ser = serialize($obj);
```

...

# Objects Serialization

- ❖ The unserialize function can receives a storable representation of a given object and creates a new object based on it.

...

```
$another_ob = unserialize($ob_ser);
```

...

# Objects Serialization

- ❖ It is possible to change the default behavior of the `serialize` function by defining the `__sleep` and the `__wakeup` magic functions within the class from which the objects were instantiated.
- ❖ The `__sleep` function should return an array that its values are the names of the object's variables we want to include in its storeable representation.

# Objects Serialization

- ❖ The `__wakeUp` function should include the code we want to be executed when a new object is created based on a storable representation.



# Objects Serialization

```
<?php
class Trip
{
    private $id;
    private $name;
    private $participants;
    private $trip_time;

    ...

    public function __sleep()
    {
        return array('id', 'name');
    }

    public function __wakeup()
    {
        $this->trip_time = date("F j, Y, g:i a");
    }
}
?>
```

# The `__toString` Function

- ❖ Defining the `__toString` method within our class we can set the behavior when objects instantiated from our class are converted to string.
- ❖ The `__toString` should return a string. That string will be the outcome when converting an object into a string.

# The `__toString` Function

```
<?php
class Person
{
    private $id;
    private $name;
    function Person($name_val,$id_val)
    {
        $this->id = $id_val;
        $this->name = $name_val;
    }
    function __toString()
    {
        $id_var = $this->id;
        $name_var = $this->name;
        return "## ".$id_var." ".$name_var." ##";
    }
}

$obj = new Person("David",123123);
echo $obj;
?>
```

# The `__invoke` Function

- ❖ Defining `__invoke` within our class we can set the behavior when trying to call an object as if it was a function.  
This magic function is available since PHP 5.3.0.

# The `__invoke` Function

```
<?php
class Student
{
    private $id;
    private $name;
    private $average;
    function Student($name_val,$id_val,$average_val)
    {
        $this->id = $id_val;
        $this->name = $name_val;
        $this->average = $average_val;
    }
    function __toString()
    {
        $id_var = $this->id;
        $average_var = $this->average;
        return "## ".$id_var." ".$average_var." ##";
    }
}
```

# The `__invoke` Function

```
function __invoke($var)
{
    $this->average=$var;
}

$obj = new Student("David",123123,94);
$obj(100);
echo $obj;
?>
```

# The `__invoke` Function

- ❖ We can use the `__invoke` magic method as if we were using delegates in C#.



# The `__invoke` Function

```
<?php
class Account
{
    private $id;
    private $balance;
    function __construct($idVal, $balanceVal)
    {
        $this->setId($idVal);
        $this->setBalance($balanceVal);
    }
    function setId($num)
    {
        if ($num > 0)
        {
            $this->id = $num;
        }
    }
}
```



# The `__invoke` Function

```
function setBalance($sum)
{
    $this->balance = $sum;
}
function deposit($sum)
{
    $this->balance += $sum;
}
function __invoke($sum)
{
    $this->deposit($sum);
}
function __toString()
{
    return "[id=$this->id balance=$this->balance]";
}
}
```

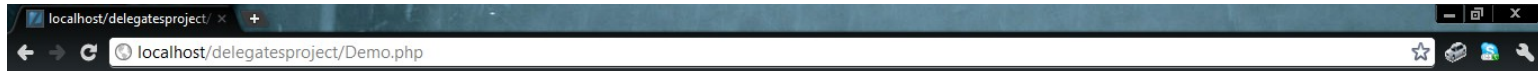
# The `__invoke` Function

```
<?php
class Utils
{
    static function transfer($sum, $from, $to)
    {
        $from(-$sum);
        $to($sum);
    }
}
?>
```

# The `__invoke` Function

```
<?php
include "Account.php";
include "Utils.php";
$accountA = new Account(1,200);
$accountB = new Account(2,300);
//echo "<br>$accountA";
//$accountA(33);
//echo "<br>$accountA";
echo "<br>$accountA  $accountB";
Utils::transfer(50,$accountA,$accountB);
echo "<br>$accountA  $accountB";
?>
```

# The `__invoke` Function



```
[id=1 balance=200] [id=2 balance=300]  
[id=1 balance=150] [id=2 balance=350]
```



# The `__autoload` Function

- ❖ Defining the `__autoload` function we can specify the code we want to execute when a required class is loaded into the memory.

# The `__autoload` Function

```
<?php
```

```
function __autoload($classname)
{
    echo "inside __autoload<p>";
    include $classname.".php";
}
```

```
$obj = new Student("mosh");
echo "just text<p>";
echo $obj;
```

```
?>
```

demo.php



# The `__autoload` Function

```
<?php
class Student
{
    var $firstname;
    function __construct($str)
    {
        $this->firstname = $str;
    }
    function __toString()
    {
        return $this->firstname;
    }
}
?>
```

————— Student.php

# The `__autoload` Function



before everything

inside `__autoload`

just text

mosh





# Type Hinting

- ❖ When defining a function we can force its parameters to be of a specific class type.
- ❖ We do it by defining the parameters preceding with a specific name of a class or an interface.

When passing a value to such parameter it must be a reference for object of the specified type (or a type that extends it... or a type that implements it - when the specified type is the name of a specific interface).

# Type Hinting

```
<?php
class Line
{
    var $p1, $p2;
    function Line(Point $ob_1, Point $ob_2)
    {
        $this->setP1($ob_1);
        $this->setP2($ob_2);
    }

    function setP1(Point $ob)
    {
        $this->p1 = $ob;
    }

    function setP2(Point $ob)
    {
        $this->p2 = $ob;
    }
}
```

# Type Hinting

```
function length()
{
    return sqrt(pow($this->p1->y-$this->p2->y,2)
                +pow($this->p1->x-$this->p2->x,2));
}

class Point
{
    var $x,$y;
    function Point($x_val,$y_val)
    {
        $this->x = $x_val;
        $this->y = $y_val;
    }
}

$line_1 = new Line(new Point(3,3),new Point(7,6));
echo $line_1->length();
?>
```

# Type Hinting

- ❖ It is also possible to specify that a specific parameter must be of an array type.
- ❖ Doing so, when passing a value to that parameter the value must be a valid array.

# Type Hinting

```
<?php
class Line
{
    var $p1, $p2;
    function Line(array $vec)
    {
        $this->setP1(new Point($vec[0], $vec[1]));
        $this->setP2(new Point($vec[2], $vec[3]));
    }
    function setP1(Point $ob)
    {
        $this->p1 = $ob;
    }
    function setP2(Point $ob)
    {
        $this->p2 = $ob;
    }
    function length()
    {
        return sqrt(pow($this->p1->y-$this->p2->y, 2)
            +pow($this->p1->x-$this->p2->x, 2));
    }
}
```

# Type Hinting

```
class Point
{
    var $x,$y;

    function Point($x_val,$y_val)
    {
        $this->x = $x_val;
        $this->y = $y_val;
    }
}

$line_1 = new Line(array(3,3,7,6));
echo $line_1->length();
?>
```

# Traits

- ❖ Defining a trait is very similar to defining a class. Instead of using the keyword `class` we use the keyword `trait`.
- ❖ The purpose of traits is to group functionality in a fine grained and consistent way.
- ❖ It is not possible to instantiate a trait. The trait servers as an additional capability that provides us with additional capabilities when using inheritance in our code.

# Traits

- ❖ The trait provides us with an horizontal composition of behavior.
- ❖ In order to use a trait we should place the `use` keyword within the body of our class.



# Traits

```
<?php
trait Academic
{
    function think()
    {
        echo "i m thinking!";
    }
}
```



# Traits

```
class Person
{
    private $id;
    private $name;
    function __construct($idValue, $nameValue)
    {
        $this->id = $idValue;
        $this->name = $nameValue;
    }
    function __toString()
    {
        return "id=".$this->id." name=".$this->name;
    }
}
```

# Traits

```
class Student extends Person
{
    use Academic;
    private $avg;
    function __construct($idVal, $nameVal, $avgVal)
    {
        parent::__construct($idVal, $nameVal);
        $this->avg = $avgVal;
    }
    function __toString()
    {
        $str = parent::__toString();
        return "avg=" . $this->avg . $str;
    }
}
```

# Traits

```
class Lecturer extends Person
{

    private $degree;
    function __construct($idVal,$nameVal,$degreeVal)
    {
        parent::__construct($idVal,$nameVal);
        $this->degree = $degreeVal;
    }
    function __toString()
    {
        $str = parent::__toString();
        return "degree=".$this->degree.$str;
    }
    use Academic;
}
```

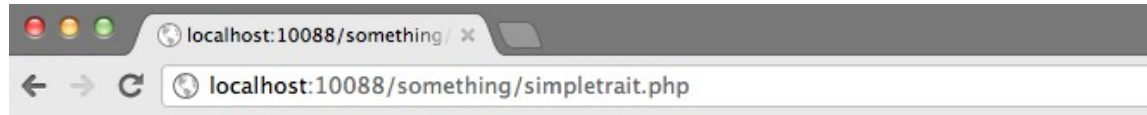
# Traits

```
$student = new Student(123123, "mosh", 98);  
$lecturer = new Lecturer(42343, "dan", "mba");
```

```
$student->think();  
echo "<hr/>";  
$lecturer->think();
```

```
?>
```

# Traits



i m thinking!

---

i m thinking!

# Traits Precedence

- ❖ Methods of the current class override methods we inserted using the trait.

# Traits Precedence

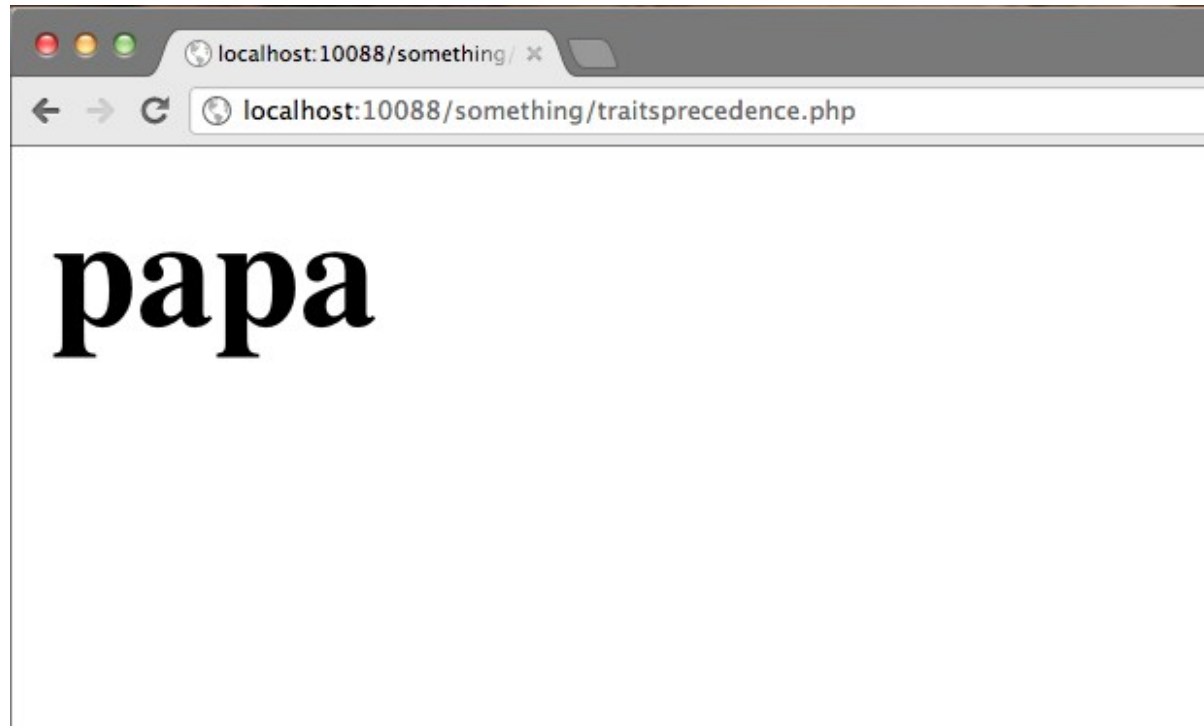
```
<?php
trait Gamer
{
    function play()
    {
        echo "<h1>gaga</h1>";
    }
}

class Person
{
    use Gamer;
    function play()
    {
        echo "<h1>papa</h1>";
    }
}

$obj = new Person();
$obj->play();
```



# Traits Precedence



# Traits Precedence

- ❖ Methods inserted by the trait override methods inherited from a base class.

# Traits Precedence

```
<?php
trait Gamer
{
    function play()
    {
        echo "<h1>gaga</h1>";
    }
}

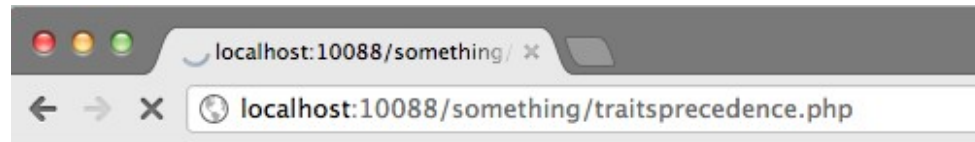
class Person
{
    function play()
    {
        echo "<h1>papa</h1>";
    }
}
```

# Traits Precedence

```
class Student extends Person
{
    use Gamer;
}

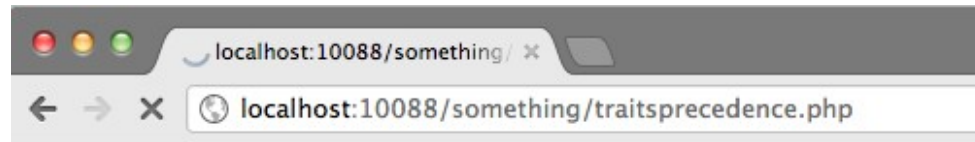
$obj = new Student();
$obj->play();
```

# Traits Precedence



**gaga**

# Traits Precedence



**gaga**

# Multiple Traits

- ❖ We can insert multiple traits into our class by listing them in the use statement separated by commas.

# Multiple Traits

```
<?php
trait Gamer
{
    function play()
    {
        echo "<h1>play</h1>";
    }
}

trait Painter
{
    function paint()
    {
        echo "<h1>paint</h1>";
    }
}
```



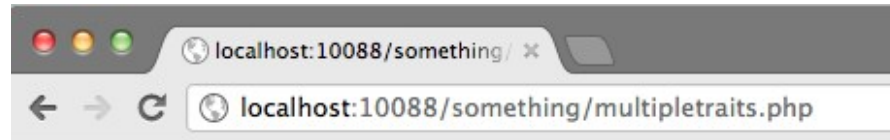
# Multiple Traits

```
class Person
{
    use Painter, Gamer;
}
```

```
$obj = new Person();
$obj->play();
$obj->paint();
```

```
?>
```

# Multiple Traits



**play**

**paint**

# Traits Conflicts

- ❖ If two traits (or more) insert two methods with the same name then a fatal error is produced.
- ❖ We can use the `insteadof` operator in order to choose the exact method we want to use.
- ❖ We can use the `as` operator in order to include a conflicting method under another name.

# Traits Conflicts

```
<?php
trait Player
{
    function play()
    {
        echo "<h1>whoo-a</h1>";
    }
    function printdetails()
    {
        echo "<h1>player...</h1>";
    }
}
```

# Traits Conflicts

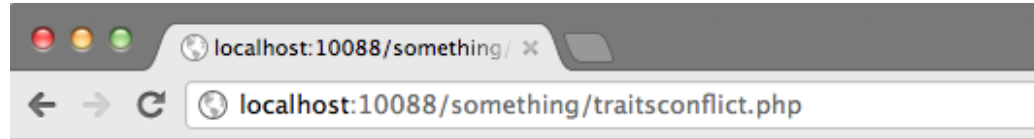
```
trait Gamer
{
  function play()
  {
    echo "<h1>shoooo</h1>";
  }
  function printdetails()
  {
    echo "<h1>gamer...</h1>";
  }
}

class Person
{
  use Gamer, Player
  {
    Gamer::printdetails insteadof Player;
    Player::play insteadof Gamer;
    Gamer::play as xplay;
  }
}
```

# Traits Conflicts

```
$ob = new Person();  
$ob->xplay();  
$ob->play();  
$ob->printdetails();  
  
?>
```

# Traits Conflicts



**shoooo**

**whoo-a**

**gamer...**

# Changing Trait's Method Visibility

- ❖ We can change the visibility of a method a trait inserts into our class. We do it using the `as` operator.

```
use [trait name] {[method name] as [visibility];}
```



# Changing Trait's Method Visibility

```
<?php
trait Academic
{
    function think()
    {
        echo "i m thinking!";
    }
}

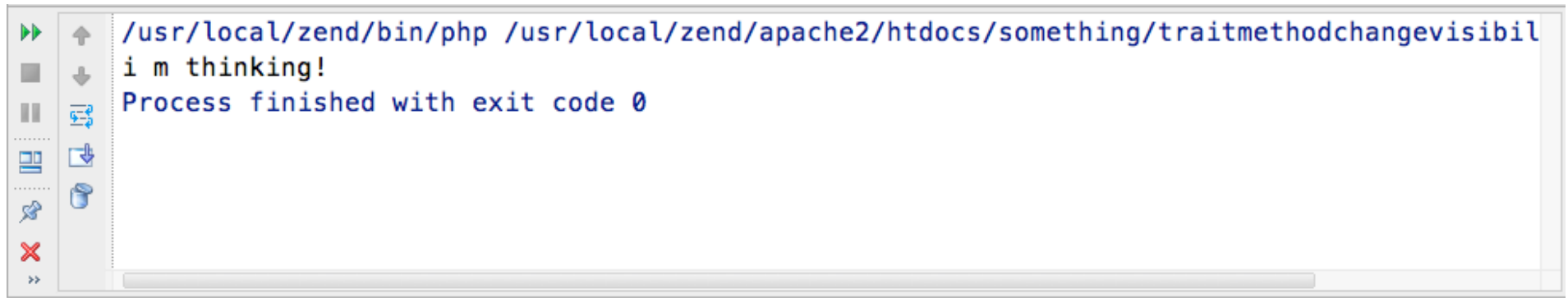
class Person
{
    use Academic {think as protected;}
    private $id;
    private $name;
}
```

# Changing Trait's Method Visibility

```
function __construct($idValue,$nameValue)
{
    $this->id = $idValue;
    $this->name = $nameValue;
}
function __toString()
{
    return "id=".$this->id." name=".$this->name;
}
function xthink()
{
    //do something here
    $this->think();
}
}

$obj = new Person(123123,"mosh");
$obj->xthink();
?>
```

# Changing Trait's Method Visibility



A terminal window showing the execution of a PHP script. The command is `/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitmethodchangevisibil`. The output is `i m thinking!` followed by `Process finished with exit code 0`. The terminal has a standard toolbar on the left with icons for running, stopping, and refreshing.

```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitmethodchangevisibil  
i m thinking!  
Process finished with exit code 0
```

# Traits Composed of Other Traits

- ❖ We can define a trait composed of others. Doing so we can put together separated traits into one.

# Traits Composed of Other Traits

```
<?php
trait Gamer
{
    function play()
    {
        echo "play...";
    }
}

trait Gambler
{
    function gamble()
    {
        echo "gamble...";
    }
}
```

# Traits Composed of Other Traits

```
trait GamblingGamer
{
    use Gambler, Gamer;
}
```

```
class User
{
    use GamblingGamer;
}
```

```
$obj = new User();
$obj->gamble();
$obj->play();
?>
```

# Traits Composed of Other Traits



A terminal window with a light gray border and a vertical toolbar on the left. The toolbar contains icons for running, pausing, and other terminal functions. The terminal text is as follows:

```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitcomposedoftraits.ph  
gamble...play...  
Process finished with exit code 0
```

# Using Class Variables & Methods

- ❖ The code in our trait can access variables and methods that were defined in the class that uses our trait.



# Using Class Variables & Methods

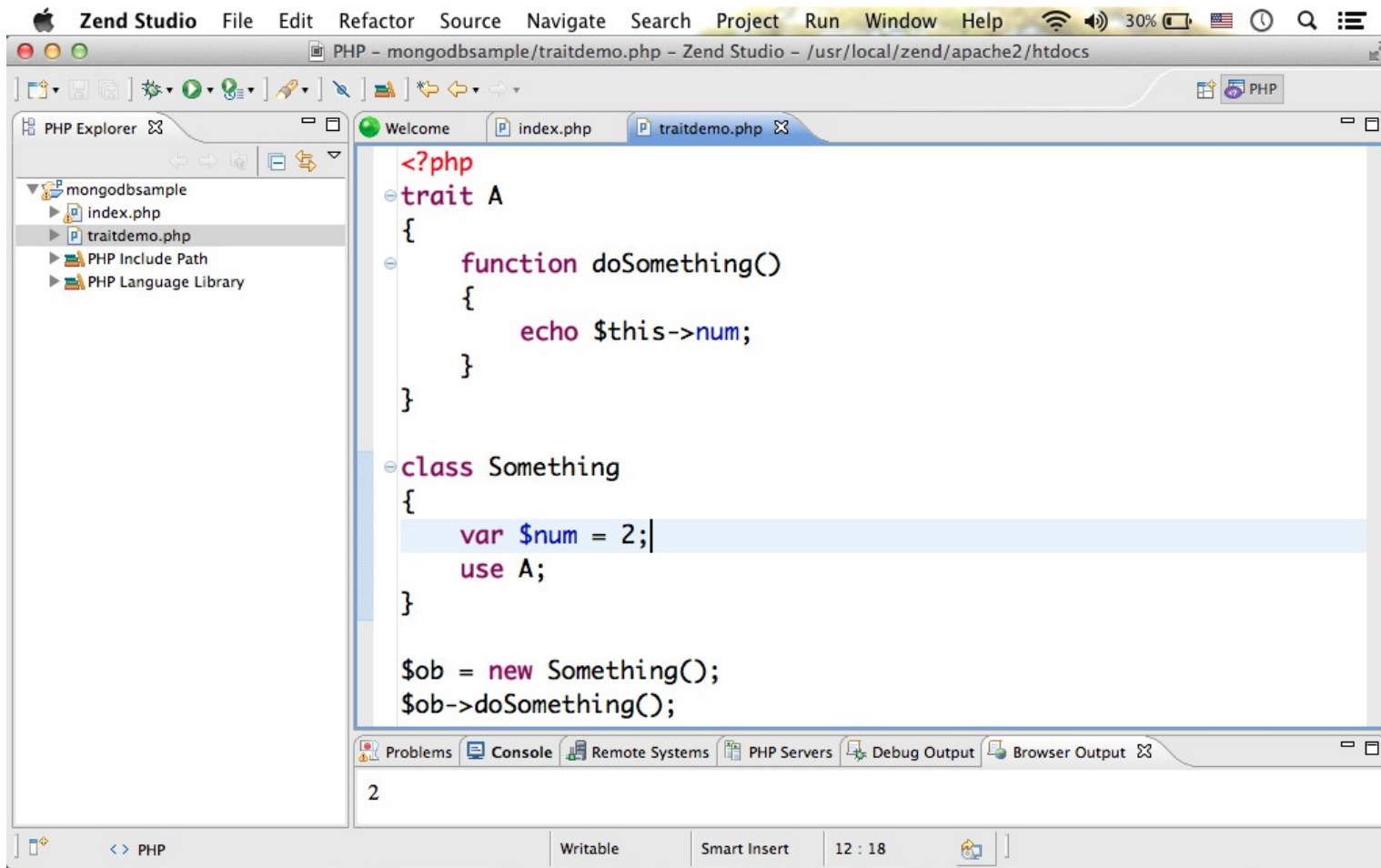
```
<?php
trait A
{
    function doSomething()
    {
        echo $this->num;
    }
}

class Something
{
    var $num = 2;
    use A;
}

$obj = new Something();
$obj->doSomething();
?>
```



# Using Class Variables & Methods



The screenshot shows the Zend Studio IDE interface. The main editor window displays the following PHP code:

```
<?php
trait A
{
    function doSomething()
    {
        echo $this->num;
    }
}

class Something
{
    var $num = 2;
    use A;
}

$obj = new Something();
$obj->doSomething();
```

The code defines a trait `A` with a `doSomething()` method that echoes the value of `$this->num`. A class `Something` uses trait `A` and has a class variable `$num` set to 2. An instance `$obj` of `Something` is created, and the `doSomething()` method is called, resulting in the output '2' shown in the console window at the bottom.

# Trait with Abstract Members

- ❖ We can define a trait that includes the definition for abstract methods.
- ❖ Doing so, we can use the trait to impose requirements upon the classes that uses our trait.

# Trait with Abstract Members

```
<?php
trait Learner
{
    abstract function learn();
}

class Student
{
    use Learner;
    function learn()
    {
        echo "i learn...";
    }
}

$obj = new Student();
$obj->learn();

?>
```

# Trait with Abstract Members



```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitwithabstractmethods  
i learn...  
Process finished with exit code 0
```

The image shows a terminal window with a light gray border and a vertical toolbar on the left. The toolbar contains icons for running, pausing, and other terminal functions. The terminal text is displayed in a blue monospaced font. The first line is a full path to a PHP script. The second line is the output of the script, and the third line is the message from the IDE indicating the process has finished successfully.

# Trait with Static Method


- ❖ It is possible to define a static method within our trait. Doing so, it will be possible to call that static method from anywhere in our code.

# Trait with Static Method

```
<?php
trait Learner
{
    static function announce_learning()
    {
        echo "quite please. we learn.";
    }
}

Learner::announce_learning();
?>
```

# Trait with Static Method



A terminal window showing the execution of a PHP script. The command is `/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitwithstaticmethod.php`. The output consists of two lines: `quite please. we learn.` and `Process finished with exit code 0`. The terminal has a standard toolbar on the left with icons for back, forward, search, and other navigation functions.

```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitwithstaticmethod.php  
quite please. we learn.  
Process finished with exit code 0
```



# Trait with Static Variable


- ❖ It is possible to define a static variable within our trait. Doing so, it will be possible to refer that static variable from anywhere in our code.

# Trait with Static Variable

```
<?php
trait Learner
{
    static $str = "quite please. we learn.";
}

echo Learner::$str;
?>
```

# Trait with Static Variable

A terminal window with a light gray border and a vertical toolbar on the left. The toolbar contains icons for running, pausing, refreshing, copying, pasting, and closing. The terminal text is as follows:

```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitwithstaticvariable.  
quite please. we learn.  
Process finished with exit code 0
```

# Trait with Properties

- ❖ It is possible to define our trait with properties. When instantiating a class that uses our trait we will be able to refer those properties in the new created object.
- ❖ If the class that uses our trait includes the definition for a property with the same name we will get an error.

# Trait with Properties

```
<?php
trait Teacher
{
    var $subject;
}

class Lecturer
{
    use Teacher;
}

$obj = new Lecturer();
$obj->subject = "math";
echo $obj->subject;
?>
```

# Trait with Properties



A terminal window with a light gray border and a vertical toolbar on the left. The toolbar contains icons for running, pausing, and other actions. The terminal text is as follows:

```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/traitwithproperties.php  
math  
Process finished with exit code 0
```

# Members Access on Instantiation

- ❖ PHP 5.4 allows us to access class members on the object instantiation. It is useful in those cases when we need to access a single member of an object and don't need the object.

# Members Access on Instantiation

```
<?  
class Utils  
{  
    function calc($numA,$numB)  
    {  
        return $numA+$numB;  
    }  
}  
  
$temp = (new Utils)->calc(3,4);  
  
echo $temp;  
  
?>
```



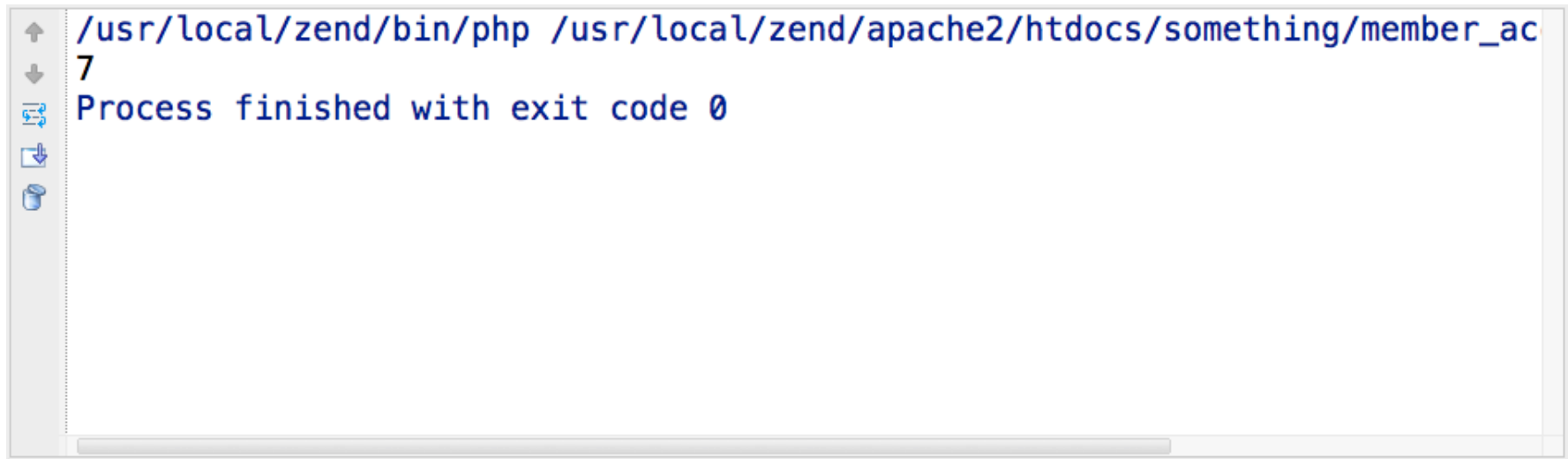


# Members Access on Instantiation

```
<?  
class Utils  
{  
    function calc($numA,$numB)  
    {  
        return $numA+$numB;  
    }  
}  
  
$temp = (new Utils)->calc(3,4);  
  
echo $temp;  
  
?>
```



# Members Access on Instantiation



A terminal window with a light gray background and a vertical toolbar on the left. The toolbar contains icons for back, forward, refresh, and search. The terminal text is as follows:

```
↑ /usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/member_ac  
↓ 7  
Process finished with exit code 0
```

# The `Class:: {expr} ()` Syntax

- ❖ PHP 5.4 allows us to call a static function defined within a class using the following unique syntax:

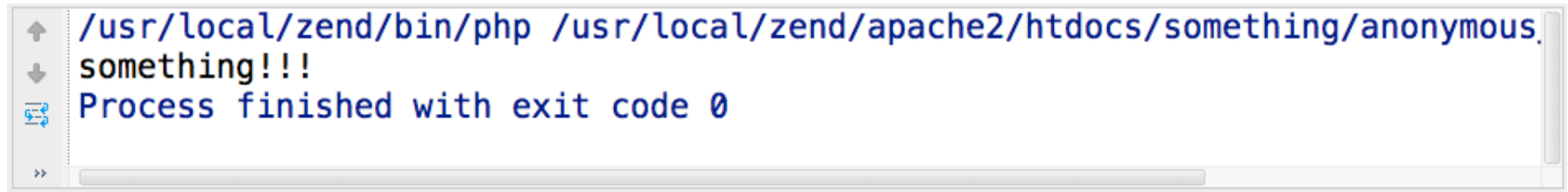
```
[class name]:: {[function name] ()
```

# The Class:: {expr} () Syntax

```
<?  
class GoGo  
{  
    public static function do_something()  
    {  
        echo "something!!!";  
    }  
}  
  
GoGo::{'do_something'}()  
  
?>
```



# The `Class:: {expr} ()` Syntax



```
/usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/anonymous.  
something!!!  
Process finished with exit code 0
```

# The `__get()` Magic Function

- ❖ When trying to get a value of a variable that doesn't exist the `__get()` magic function will be invoked. The name of the variable we try to access will be passed over to this magic function.

# The `__set()` Magic Function

- ❖ When trying to assign a value to a variable that doesn't exist the `__set()` magic function will be invoked. The name of the variable will be passed over as the first argument. The value will be passed over as the second argument.

# Sample

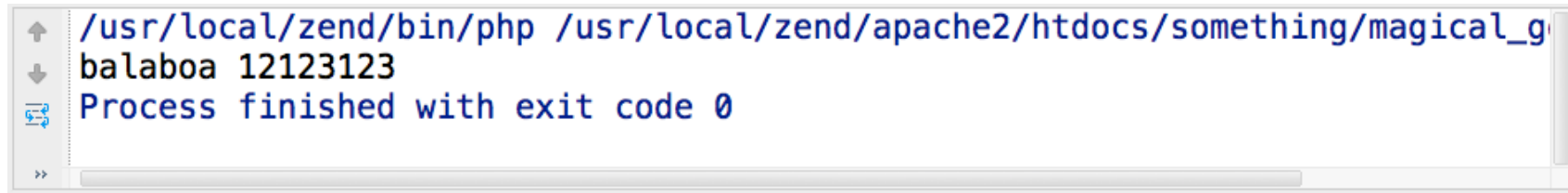
```
<?
class Bongo
{
    var $vec;
    function __construct()
    {
        $this->vec = array();
    }
    function __get($str)
    {
        return $this->vec[$str];
    }
    function __set($var_name, $var_value)
    {
        $this->vec[$var_name] = $var_value;
    }
}

$obj = new Bongo();
$obj->name="balaboa";
$obj->id=12123123;
echo $obj->name." ".$obj->id;
?>
```





# Sample



```
↑ /usr/local/zend/bin/php /usr/local/zend/apache2/htdocs/something/magical_g  
↓ balaboa 12123123  
⏪ Process finished with exit code 0  
⏩
```

# The `__PHP_Incomplete_Class` Object

- ❖ When storing an object in `$_SESSION` trying to retrieve it in another page we will get an error if the class itself is not available when the `session_start()` function builds the `$_SESSION` array.
- ❖ In order to avoid this problem we better make sure that the class definition is available before we call the `session_start()` function.

# The `__PHP_Incomplete_Class` Object

- ❖ Similar scenarios include calling the `unserialize()` function while the class definition is not available.



# The `__debugInfo()` Magic Function

- ❖ The `__debugInfo()` magic function was introduced in PHP 5.6. It allows us to specify which properties will be presented together with their values when passing over an object to the `var_dump()` function.

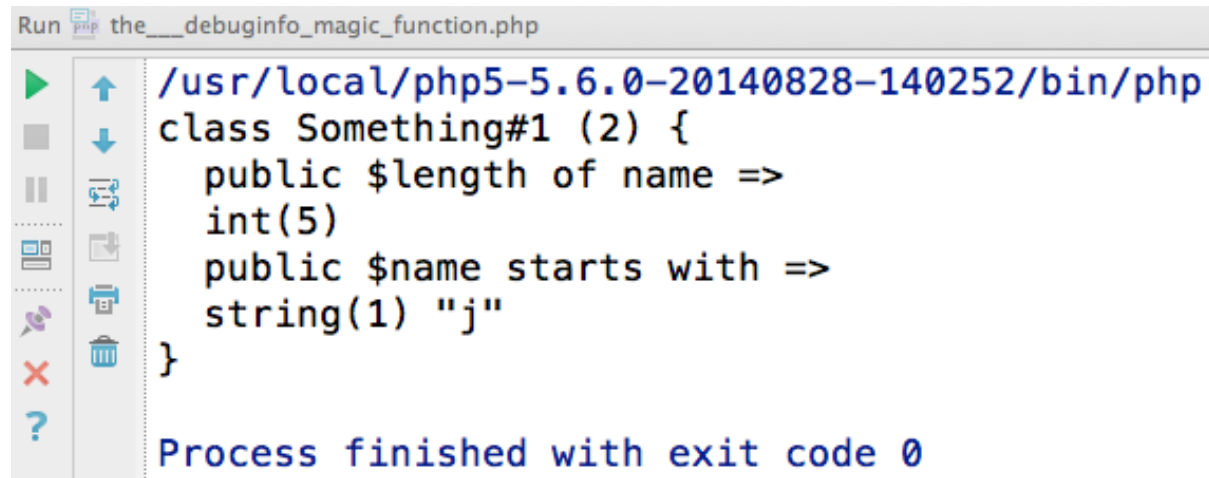
# The `__debugInfo()` Magic Function

```
<?php
class Something
{
    private $name;
    public function __construct($str)
    {
        $this->name = $str;
    }
    public function __debugInfo()
    {
        return [
            'length of name' => strlen($this->name),
            'name starts with' => substr($this->name,0,1)
        ];
    }
}

$obj = new Something("james");
var_dump($obj);
?>
```



# The `__debugInfo()` Magic Function



```
Run the__debuginfo_magic_function.php
/usr/local/php5-5.6.0-20140828-140252/bin/php
class Something#1 (2) {
    public $length of name =>
        int(5)
    public $name starts with =>
        string(1) "j"
}
Process finished with exit code 0
```

# Anonymous Class

- ❖ As of PHP 7, we can define an anonymous class. It is highly useful when in a need for one object only.
- ❖ The new anonymous class can extend another class and implements as many interfaces as we want.
- ❖ When defining an anonymous class within the scope of another class we won't get any access to any of the private or protected properties of the outer class.

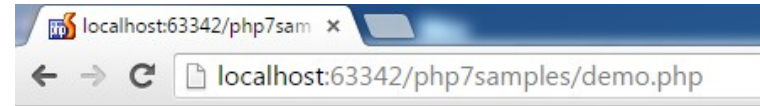
# Anonymous Class

```
<?php
class C {
    public function doSomething() {
        echo "<h1>something</h1>";
    }
}
interface I {}
trait T {}
$obj = new class(10) extends C implements I {
    private $num;
    public function __construct($num)
    {
        $this->num = $num;
    }
    use T;
};
$obj->doSomething();
?>
```





# Anonymous Class



**something**

# The Filtered `unserialize()` Function

- ❖ When we unserialize an object, as of PHP 7 we can specify the names of the classes that can be unserialized.
- ❖ Specifying the names of the classes that can be unserialized improves the security of our code. When unserializing untrusted data using this function we prevent possible code injections.

# The Filtered unserialize() Function

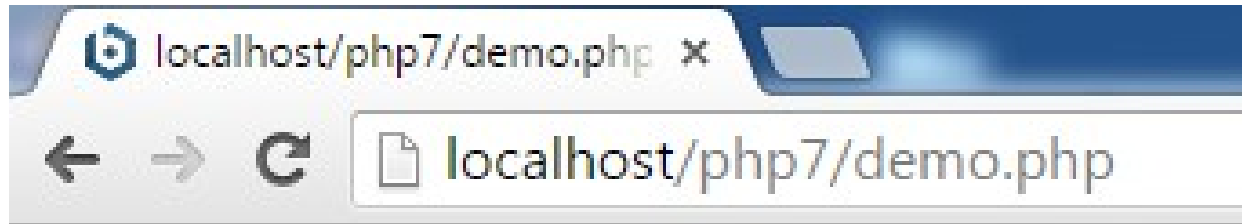
```
<?php
class A {
    private $magic;
    function __construct($number) {
        $this->setMagic($number);
    }
    function setMagic($number) {
        if($number>0) {
            $this->magic = $number;
        }
    }
    function getMagic() {
        return $this->magic;
    }
}
```



# The Filtered `unserialize()` Function

```
$obj1 = new A(5);  
$data = serialize($obj1);  
$obj2 = unserialize(  
    $data,  
    ["allowed_classes" => ["A", "Rectangle"]]);  
echo $obj2->getMagic();  
  
?>
```

# The Filtered unserialize () Function



5