LOCALIZATION OF E-GOVERNANCE PROJECT

PHP 5 Classes and Objects

Prepared by:

Project Team,

The e-platform model,

DIT, MoIC, RGoB

Contents

Introduction

In PHP, a class is simply a set of program statements which perform a specific task. A typical class definition contains both variables and functions, and serves as the template from which to spawn specific instances of that class.

These specific instances of a class are referred to as objects. Every object has certain characteristics, or properties, and certain pre-defined functions, or methods. These properties and methods of the object correspond directly with the variables and functions within the class definition.

Once a class has been defined, PHP allows you to spawn as many instances of the class as you like. Each of these instances is a completely independent object, with its own properties and methods, and can therefore be manipulated independently of other objects. This comes in handy in situations where you need to spawn more than one instance of an object - for example, two simultaneous database links for two simultaneous queries, or two shopping carts.

Classes also help you keep your code modular - you can define a class in a separate file, and include that file only in the scripts where you plan to use the class - and simplify code changes, since you only need to edit a single file to add new functionality to all your spawned objects.

Sample Exercise (Class)

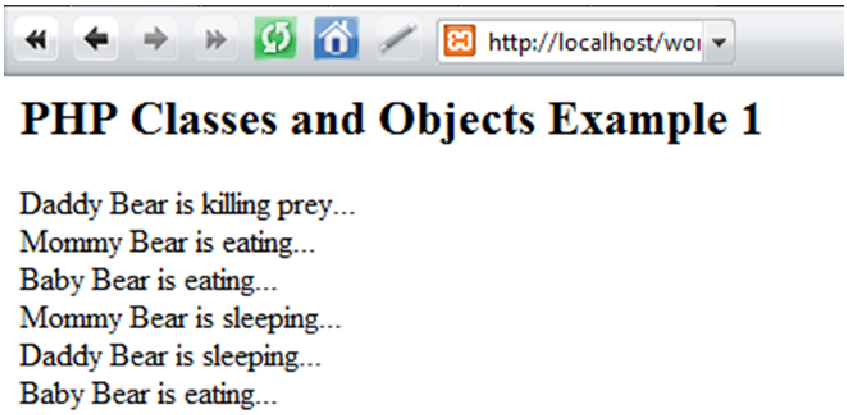
To understand this better, pick an animal, any animal. For example, we can take bear. every bear has certain characteristics - age, weight, sex - which are equivalent to object properties. And every bear can perform certain activities - eat, sleep, walk, run, mate - all of which are equivalent to object methods.

Let's take it a little further. Since all bears share certain characteristics, it is possible to conceive of a template `Bear()`, which defines the basic characteristics and abilities of every bear on the planet. Once this `Bear()` ("class") is used to create a new `$bear` ("object"), the individual characteristics of the newly-created Bear can be manipulated independently of other Bears that may be created from the template.

Now, if you sat down to code this class in PHP 5, it would probably look something like this:

```
<?php
	// PHP 5 // class definition
	class Bear {
		// define properties
		public $name;
		public $weight;
		public $age;
		public $sex;
		public $colour;

		// define methods
		public function eat() {
			echo $this->name." is eating...\n";
		}
		public function run() {
			echo $this->name." is running...\n";
		}
		public function kill() {
			echo $this->name." is killing prey...\n";
		}
		public function sleep() {
			echo $this->name." is sleeping...\n";
		}
	}
?>
<?php
	// my first bear
	$daddy = new Bear;
	// give him a name
	$daddy->name = "Daddy Bear";
	// how old is he
	$daddy->age = 8;
	// what sex is he
	$daddy->sex = "male";
	// what colour is his coat
	$daddy->colour = "black";
	// how much does he weigh
	$daddy->weight = 300;
```

```
	// give daddy a wife
	$mommy = new Bear;
	$mommy->name = "Mommy Bear";
	$mommy->age = 7;
	$mommy->sex = "female";
	$mommy->colour = "black";
	$mommy->weight = 310;

	// and a baby to complete the family
	$baby = new Bear;
	$baby->name = "Baby Bear";
	$baby->age = 1;
	$baby->sex = "male";
	$baby->colour = "black";
	$baby->weight = 180;

	print("<h2>PHP Class and Object Example 1</h1>");
	// a nice evening in the Bear family
	// daddy kills prey and brings it home
	$daddy->kill();
	echo "<br/>";
	// mommy eats it
	$mommy->eat();
	echo "<br/>";
	// and so does baby
	$baby->eat();
	echo "<br/>";
	// mommy sleeps
	$mommy->sleep();
	echo "<br/>";
	// and so does daddy
	$daddy->sleep();
	echo "<br/>";
	// baby eats some more
	$baby->eat();
	echo "<br/>";
?>
```



Using Constructors

It's also possible to automatically execute a function when the class is called to create a new object. This is referred to in geek lingo as a constructor and, in order to use it, your PHP 5 class definition must contain a special function, `__construct()`.

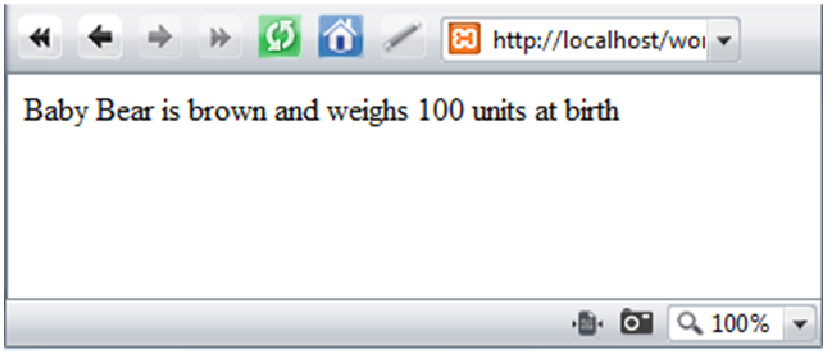
For example, if you'd like all newly born bears to be brown and weigh 100 units, you could add this to your class definition:

```
<?php
	// PHP 5 // class definition
	class Bear {
		// define properties
		public $name;
		public $weight;
		public $colour;

		public function __construct() {
			$this->age = 0;
			$this->weight = 100;
			$this->colour = "brown";
		}
	}
?>
<?php
	// create instance
```

```
	$baby = new Bear;
	$baby->name = "Baby Bear";
	echo $baby->name." is ".$baby->colour." and weighs ".$baby->weight." units
at birth";
?>
```

Here, the constructor automatically sets default properties every time an object of the class is instantiated. Therefore, when you run the script above, you will see this:

Baby Bear is brown and weighs 100 units at birth



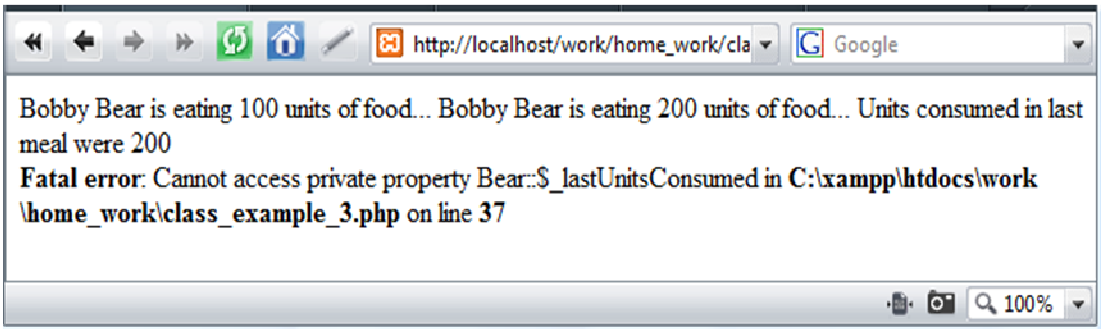
Using `private`

PHP 5 makes it possible to mark class properties and methods as private, which means that they cannot be manipulated or viewed outside the class definition. This is useful to protect the inner workings of your class from manipulation by object instances. Consider the following example, which illustrates this by adding a new `private` variable, `$_lastUnitsConsumed`, to the `Bear()` class:

```
<?php
	// class definition
	class Bear {
		// define properties
		public $name;
		public $age;
		public $weight;
		private $_lastUnitsConsumed;

		// constructor

		public function __construct() {
			$this->age = 0;
```

```
                $this->weight = 100;
                $this->_lastUnitsConsumed = 0;
        }

        // define methods
        public function eat($units) {
                echo $this->name." is eating ".$units." units of food...\n";
                $this->weight += $units;
                $this->_lastUnitsConsumed = $units;
        }

        public function getLastMeal() {
                echo "Units consumed in last meal were ".$this->_lastUnitsConsumed."\n";
        }
    }
?>
<?php
    $bob = new Bear;
    $bob->name = "Bobby Bear";
    $bob->eat(100);
    $bob->eat(200);
    echo $bob->getLastMeal();
    // the next line will generate a fatal error
    $bob->_lastUnitsConsumed = 1000;
?>
```

Now, since the `$_lastUnitsConsumed` variable is declared as private, any attempt to modify it from an object instance will fail. Here is the output:



In a similar way, class methods can also be marked as `private`.

Inheritance Concept

Two of the best things about OOP, whether in PHP 4 or in PHP 5, are extensibility and inheritance. Very simply, this means that you can create a new class based on an existing class, add new features (read: properties and methods) to it, and then create objects based on this new class. These objects will contain all the features of the original parent class, together with the new features of the child class.

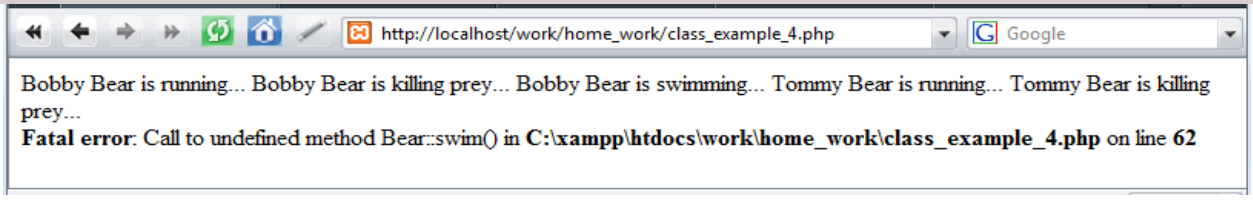
As an illustration, consider the following `PolarBear()` class, which extends the `Bear()` class with a new method.

```
<?php
	// class definition
	class Bear {
		// define properties
		public $name;
		public $weight;
		public $age;
		public $sex;
		public $colour;

		public function __construct() {
			$this->age = 0;
			$this->weight = 100;
		}
		// define methods
		public function eat($units) {
			echo $this->name." is eating ".$units." units of food...\n";
			$this->weight += $units;
		}
		public function run() {
			echo $this->name." is running...\n";
		}
		public function kill() {
			echo $this->name." is killing prey...\n";
		}
		public function sleep() {
			echo $this->name." is sleeping...\n";
		}
	}
```

```
	// extended class definition
	class PolarBear extends Bear {
		// constructor
		public function __construct() {
			parent::__construct();
			$this->colour = "white";
			$this->weight = 600;
		}
		// define methods
		public function swim() {
			echo $this->name." is swimming...\n";
		}
	}
?>
```

The `extends` keyword is used to extend a parent class to a child class. All the functions and variables of the parent class immediately become available to the child class. This is clearly visible in the following code snippet:

```
<?php
	// create instance of Bear()
	$tom = new Bear;
	$tom->name = "Tommy Bear";

	// create instance of PolarBear()
	$bob = new PolarBear;
	$bob->name = "Bobby Bear";

	// $bob can use all the methods of Bear() and PolarBear()
	$bob->run();
	$bob->kill();
	$bob->swim();

	// $tom can use all the methods of Bear() but not PolarBear()
	$tom->run();
	$tom->kill();
	$tom->swim();
?>
```



In this case, the final call to `$tom->swim()` will fail and cause an error, because the `Bear()` class does not contain a `swim()` method. However, none of the calls `to $bob->run()` or `$bob->kill()` will fail, because as a child of the `Bear()` class, `PolarBear()` inherits all the methods and properties of its parent.

Note how the parent class constructor has been called in the `PolarBear()` child class constructor - it's a good idea to do this so that all necessary initialization of the parent class is carried out when a child class is instantiated. Child-specific initialization can then be done in the child class constructor. Only if a child class does not have a constructor, is the parent class constructor automatically called.

Using `final`

To prevent a class or its methods from being inherited, use the final keyword before the class or method name (this is new in PHP 5 and will not work in older versions of PHP). Here's an example, which renders the `Bear()` class un-inheritable (if that's actually a word):

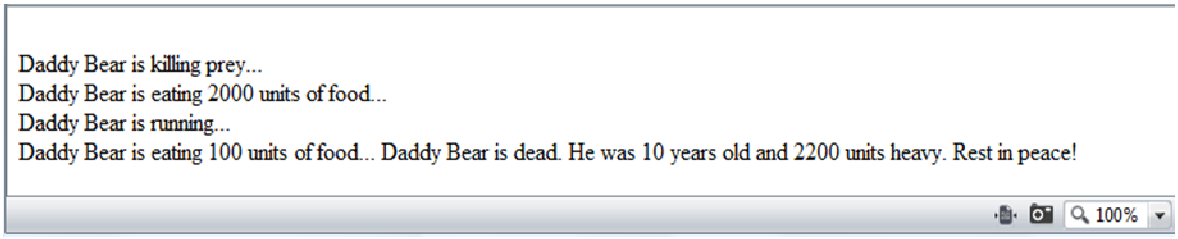
```
<?php
	// class definition
	final class Bear {
		// define properties
		// define methods
	}
	// extended class definition
	// this will fail because Bear() cannot be extended
	class PolarBear extends Bear {
		// define methods
	}
	// create instance of PolarBear()
	// this will fail because Bear() could not be extended
	$bob = new PolarBear;
	$bob->name = "Bobby Bear";
	echo $bob->weight;
?>
```

Using Destructors

Just as there are constructors, so also are there destructors. Destructors are object methods which are called when the last reference to an object in memory is destroyed, and they are usually tasked with clean-up work - for example, closing database connections or files, destroying a session and so on. Destructors are only available in PHP 5, and must be named `__destruct()`. Here's an example:

```
<?php
	// class definition
	class Bear {
		// define properties
		public $name;
		public $weight;
		public $age;
		public $sex;
		public $colour;

		// constructor
		public function __construct() {
			$this->age = 0;
			$this->weight = 100;
			$this->colour = "brown";
		}

		// destructor
		public function __destruct() {
			echo $this->name." is dead. He was ".$this->age." years old and
".$this->weight." units heavy. Rest in peace! \n";
		}

		// define methods
		public function eat($units) {
			echo $this->name." is eating ".$units." units of food...\n";
			$this->weight += $units;
		}
		public function run() {
			echo $this->name." is running...\n";
		}

		public function kill() {
```

```
                    echo $this->name." is killing prey...\n";
            }
      }
      // create instance of Bear()
      $daddy = new Bear;
      $daddy->name = "Daddy Bear";
      $daddy->age = 10;
      echo "<br/>";
      $daddy->kill();
      echo "<br/>";
      $daddy->eat(2000);
      echo "<br/>";
      $daddy->run();
      echo "<br/>";
      $daddy->eat(100);
?>
```



`get_class()` and `get_parent_class()`

PHP 4 and PHP 5 come with a bunch of functions designed to let you discover object properties and methods, and find out which class an object belongs to. The first two of these are the `get_class()` and `get_parent_class()` functions, which tell you the name of the classes which spawned a particular object. Consider the following class definition:

```
<?php
      class DOG{
            public $name;
            public $sex;
            public $colour;
            public $weight;

            public function eat(){
                  echo $this->name. " with ".$this->colour." colour is eating...\n";
```

```
        }
        public function weightt(){
            echo $this->name. " is ".$this->weight." Kilo Gram.";
        }
    }
?>
```

You can view all the properties exposed by a class with `get_class_vars()`, and all its methods with `get_class_methods()` function. To view properties of the specific object instance, use `get_object_vars()` instead of `get_class_vars()`. Here is an example:

```
<?php
    $daddy = new DOG;
    $daddy->name="Daddy Dog";
    $daddy->weight=120;
    $daddy->sex="Male";
    $daddy->colour="Blue";

    print("<br/>");
    $class_name = get_class($daddy);
    print_r(get_class_vars($class_name));
    print("<br/>");
    print_r(get_class_methods($class_name));
    print("<br/>");
    print_r(get_object_vars($daddy));
?>
```

```
Array ( [name] => [sex] => [colour] => [weight] => )
Array ( [0] => eat [1] => weightt )
Array ( [name] => Daddy Dog [sex] => Male [colour] => Blue [weight] => 120 )
```

As noted in one of the previous segments of this tutorial, the `print_r()` function allows you to look inside any PHP variable, including an object.

Autoloading Objects

In PHP 5, it is no longer necessary to create one PHP source file per-class definition while writing object-oriented application. You may define an `__autoload` function which is automatically called in case you are trying to use a `class/interface` which hasn't been defined yet. By calling this function the scripting engine is given a last chance to load the class before PHP fails with an error.

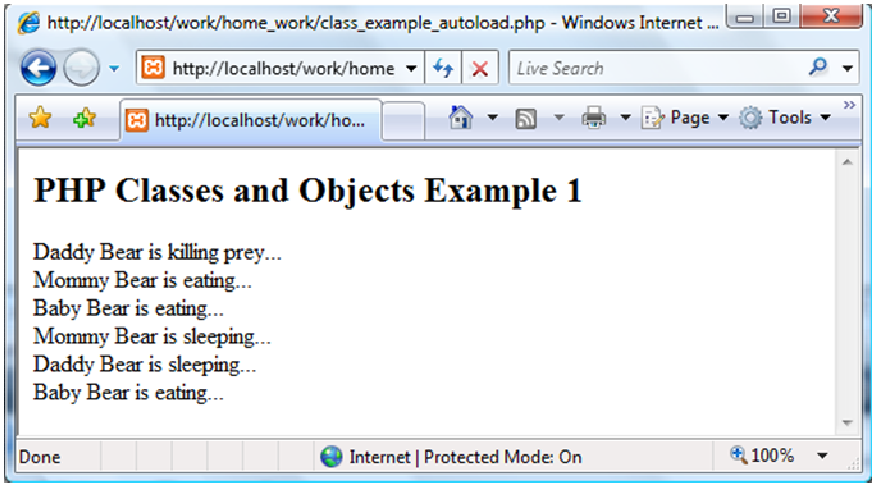
Note: Exceptions thrown in __autoload function cannot be caught in the catch block and results in a fatal error.

Note: Autoloading is not available if using PHP in CLI interactive mode.

Note: If the class name is used e.g. in call_user_func() then it can contain some dangerous characters such as ../. It is recommended to not use the user-input in such functions or at least verify the input in __autoload().

The example below attempts to load the class Bear from the files Bear.php.

```
<?php
	function __autoload($class_name) {
		require_once $class_name . '.php';
	}
	$obj  = new Bear();
?>
```



Visibility

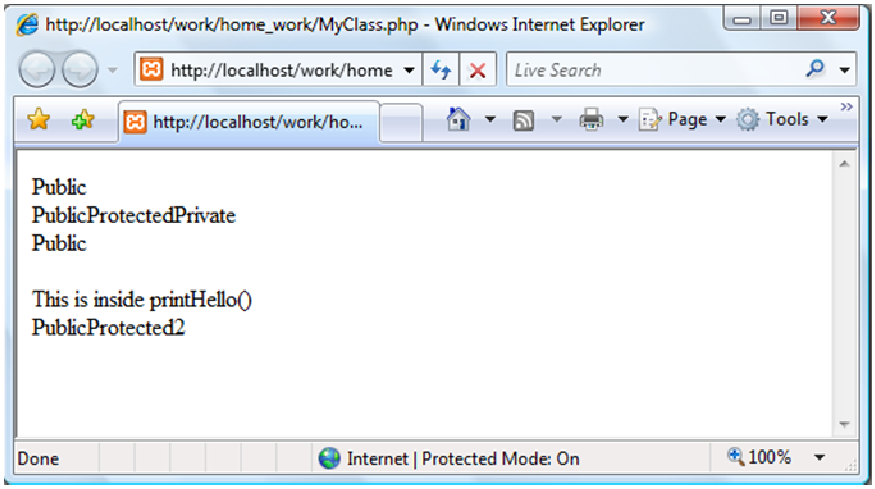
The visibility of a property or method can be defined by prefixing the declaration with the keywords: `public`, `protected` or `private`. `Public` declared items can be accessed everywhere. `Protected` limits access to inherited and parent classes (and to the class that defines the item). `Private` limits visibility only to the class that defines the item.

Members Visibility

Class members must be defined with public, private, or protected.

```
<?php
	class MyClass
	{
		public $public = 'Public';
		protected $protected = 'Protected';
		private $private = 'Private';

		function printHello()
		{
			echo $this->public;
			echo $this->protected;
			echo $this->private;
		}
	}
	$obj = new MyClass();
	echo $obj->public; // Works
	echo "<br/>";
	//echo $obj->protected; // Fatal Error
	//echo $obj->private; // Fatal Error
	$obj->printHello(); // Shows Public, Protected and Private
	echo "<br/>";

	class MyClass2 extends MyClass
	{
		// We can redeclare the public and protected method, but not private
		protected $protected = 'Protected2';

		function printHello()
		{
```

```
                    echo "This is inside printHello()";
                    echo "<br/>";
                    echo $this->public;
                    echo $this->protected;
                    echo $this->private;
            }
      }

      $obj2 = new MyClass2();
      echo $obj2->public; // Works
      echo "<br/>";
      echo $obj2->private; // Undefined
      echo "<br/>";
      //echo $obj2->protected; // Fatal Error
      $obj2->printHello(); // Shows Public, Protected2, Undefined
      echo "<br/>";
?>
```



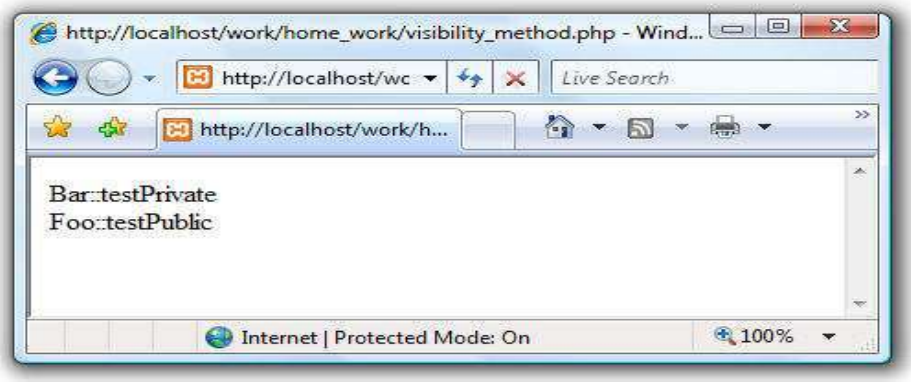
Method Visibility

Class methods must be defined with public, private, or protected. Methods without any declaration are defined as public.

```
<?php
      class MyClass
      {
            // Declare a public constructor
            public function __construct() { }
```

```
        // Declare a public method
        public function MyPublic() { }

        // Declare a protected method
        protected function MyProtected() { }

        // Declare a private method
        private function MyPrivate() { }

        // This is public
        function Foo()
        {
                $this->MyPublic();
                $this->MyProtected();
                $this->MyPrivate();
        }
}

$myclass = new MyClass;
$myclass->MyPublic(); // Works
//$myclass->MyProtected(); // Fatal Error
//$myclass->MyPrivate(); // Fatal Error
$myclass->Foo(); // Public, Protected and Private work

/**
 * Define MyClass2
 */
class MyClass2 extends MyClass
{
        // This is public
        function Foo2()
        {
                $this->MyPublic();
                $this->MyProtected();
                //$this->MyPrivate(); // Fatal Error
        }
}

$myclass2 = new MyClass2;
$myclass2->MyPublic(); // Works
$myclass2->Foo2(); // Public and Protected work, not Private
```

```
	class Bar
	{
		public function test() {
			$this->testPrivate();
			$this->testPublic();
		}

		public function testPublic() {
			echo "Bar::testPublic\n";

		}

		private function testPrivate() {
			echo "Bar::testPrivate\n";
			echo "<br/>";
		}
	}
	class Foo extends Bar
	{
		public function testPublic() {
			echo "Foo::testPublic\n";
		}

		private function testPrivate() {
			echo "Foo::testPrivate\n";
		}
	}
	$myFoo = new foo();
	$myFoo->test(); // Bar::testPrivate
				// Foo::testPublic
?>
```

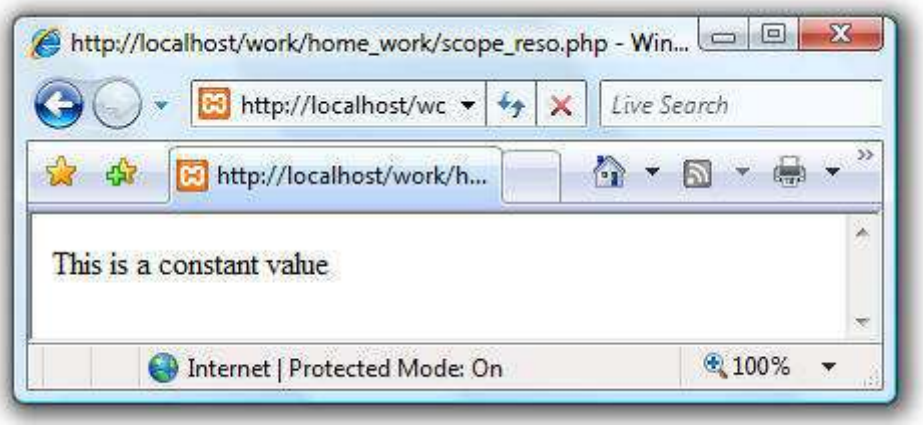


Scope Resolution Operator (::)

The Scope Resolution Operator, the double colon, is a token that allows access to static, constant, and overridden members or methods of a class.

Example #1 :: from outside the class definition

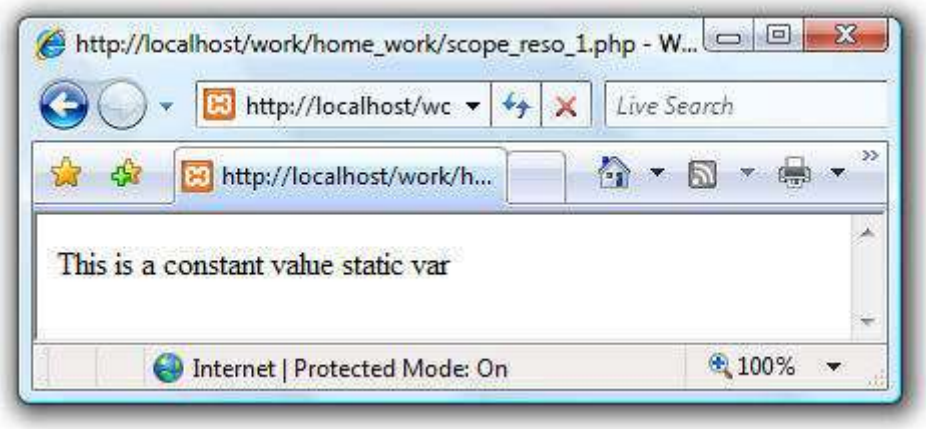
```
<?php
	class MyClass {
		const CONST_VALUE = 'This is a constant value';
	}
	echo MyClass::CONST_VALUE;
?>
```



Example #2 :: from inside the class definition

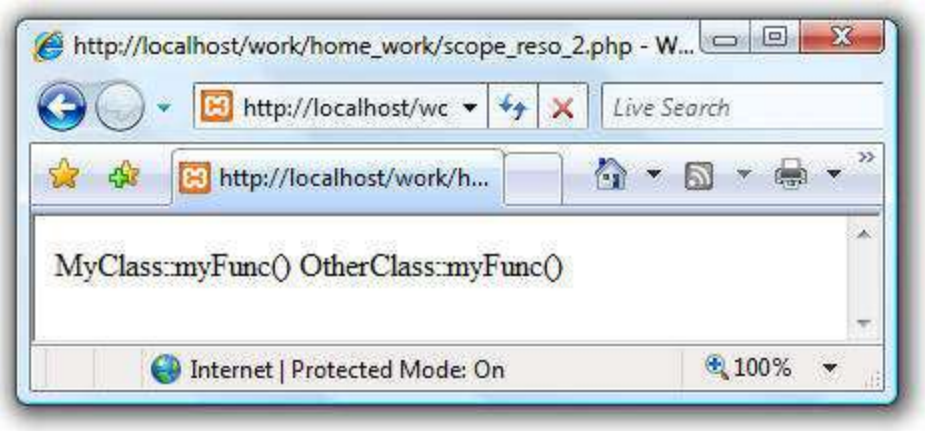
```
<?php
	class MyClass {
		const CONST_VALUE = 'This is a constant value';
	}
?>
<?php
class OtherClass extends MyClass
{
    public static $my_static = 'static var';

    public static function doubleColon() {
        echo parent::CONST_VALUE . "\n";
```

```
        echo self::$my_static . "\n";
    }
}
OtherClass::doubleColon();
?>
```



Example #3 Calling a parent's method

```
<?php
	class MyClass
	{
		protected function myFunc() {
			echo "MyClass::myFunc()\n";
		}
	}

	class OtherClass extends MyClass
	{
		// Override parent's definition
		public function myFunc()
		{
			// But still call the parent function
			parent::myFunc();
			echo "OtherClass::myFunc()\n";
		}
	}

	$class = new OtherClass();
	$class->myFunc();
?>
```



Static Keyword

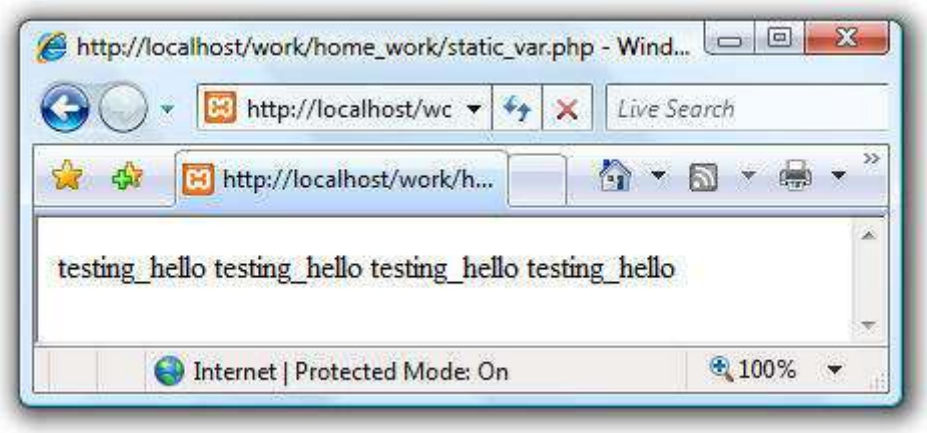
Declaring class members or methods as `static` makes them accessible without needing an instantiation of the class. A member declared as static cannot be accessed with an instantiated class object (though a static method can).

Because static methods are callable without an instance of the object created, the pseudo variable $this is not available inside the method declared as static.

Note : Static properties cannot be accessed through the object using the arrow operator ->.

```
<?php
	class Hello
	{
		public static $my_static = 'testing_hello';

		public function staticValue() {
			return self::$my_static;
		}
	}

	class Bar extends Hello
	{
		public function testing_helloStatic() {
			return parent::$my_static;
```

```
            }
        }

        print Hello::$my_static . "\n";

        $testing_hello = new Hello();
        print $testing_hello->staticValue() . "\n";
        print $testing_hello->my_static . "\n";      // Undefined "Property" my_static

        print Bar::$my_static . "\n";
        $bar = new Bar();
        print $bar->testing_helloStatic() . "\n";
?>
```



Class Abstraction

PHP 5 introduces abstract classes and methods. It is not allowed to create an instance of a class that has been defined as `abstract`. Any class that contains at least one abstract method must also be abstract. Methods defined as abstract simply declare the method's signature they cannot define the implementation.

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same (or a less restricted) visibility. For example, if the abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private.

```
<?php
        abstract class AbstractClass
        {
```

```
        // Force Extending class to define this method
        abstract protected function getValue();
        abstract protected function prefixValue($prefix);

        // Common method
        public function printOut() {
            print $this->getValue() . "\n";
        }
    }

    class ConcreteClass1 extends AbstractClass
    {
        protected function getValue() {
            return "ConcreteClass1";
        }

        public function prefixValue($prefix) {
            return "{$prefix}ConcreteClass1";
        }
    }

    class ConcreteClass2 extends AbstractClass
    {
        public function getValue() {
            return "ConcreteClass2";
        }

        public function prefixValue($prefix) {
            return "{$prefix}ConcreteClass2";
        }
    }

    $class1 = new ConcreteClass1;
    $class1->printOut();
    echo $class1->prefixValue('FOO_') ."\n";

    $class2 = new ConcreteClass2;
    $class2->printOut();
    echo $class2->prefixValue('FOO_') ."\n";
?>
```

OUTPUT

```
ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2
```

A Real time Example for Abstract Class

```
<?php
	abstract class person {
		abstract protected function write_info();

		public $LastName;
		public $FirstName;
		public $BirthDate;

		public function get_Age($today=NULL){
			//age computation function
		}
	}

	final class employee extends person{
		public $EmployeeNumber;
		public $DateHired;

		public function write_info(){
		    echo "Writing ". $this->LastName . "'s info to emloyee dbase table";
			//ADD unique mandatory checking unique to EMPLOYEE ONLY
			//actual sql codes here
		}
	}

	final class student extends person{
		public $StudentNumber;
		public $CourseName;

		public function write_info(){
		   echo "Writing ". $this->LastName . "'s info to student dbase table";
			//ADD unique mandatory checking unique to STUDENT ONLY
			//actual sql codes here
		}
	}
```

```
        $personA = new employee;
        $personB = new student;

        $personA->FirstName="Joe";
        $personA->LastName="McDonalds";

        $personB->FirstName="Ben";
        $personB->LastName="Dover";

        $personA->write_info();
?>
```

OUTPUT FOR THE ABOVE PROGRAM IS

```
Writing McDonalds's info to emloyee dbase table
```

Object Iteration

PHP 5 provides a way for objects to be defined so it is possible to iterate through a list of items, with, for example a `foreach` statement. By default, all visible properties will be used for the iteration.

```
<?php
class MyClass
{
    public $var1 = 'value 1';
    public $var2 = 'value 2';
    public $var3 = 'value 3';

    protected $protected = 'protected var';
    private   $private   = 'private var';

    function iterateVisible() {
       echo "MyClass::iterateVisible:\n";
       foreach($this as $key => $value) {
           print "$key => $value\n";
       }
    }
}

$class = new MyClass();
```

```
foreach($class as $key => $value) {
    print "$key => $value\n";
}
echo "\n";


$class->iterateVisible();

?>
```

The above example will output:

```
var1 => value 1
var2 => value 2
var3 => value 3
MyClass::iterateVisible:
var1 => value 1
var2 => value 2
var3 => value 3
protected => protected var
private => private var
```

Magic Methods

__toString

```
<?php
// Declare a simple class
class TestClass
{
    public $foo;

    public function __construct($foo) {
        $this->foo = $foo;
    }
    public function __toString() {
        return $this->foo;
    }
}
```

```
$class = new TestClass('Hello');
echo $class;
?>
```

The above example will output:

```
Hello
```

__clone

```
<?php
	class Car
	{
		private $gas = 0;
		private $color = "red";
		function addGas($amount){
			$this->gas = $this->gas + $amount;
			echo "$amount gallons added to gas tank";
		}
		function __clone()  {
			$this->gas = 0;
		}
	}
	$firstCar = new Car;
	$firstCar->addGas(12);
	$secondCar=clone $firstCar;
	$secondCar->addGas(17);
?>
```

The above example will output:

```
12 gallons added to gas tank17 gallons added to gas tank
```

Patterns

Patterns are ways to describe best practices and good designs. They show a flexible solution to common programming problems.

Factory

The Factory pattern allows for the instantiation of objects at runtime. It is called a Factory Pattern since it is responsible for "manufacturing" an object. A Parameterized Factory receives the name of the class to instantiate as argument.

Example #1 Parameterized Factory Method

```
<?php
class Example
{
    // The parameterized factory method
    public static function factory($type)
    {
        if (include_once 'Drivers/' . $type . '.php') {
            $classname = 'Driver_' . $type;
            return new $classname;
        } else {
            throw new Exception ('Driver not found');
        }
    }
}
?>
```

Defining this method in a class allows drivers to be loaded on the fly. If the Example class was a database abstraction class, loading a MySQL and SQLite driver could be done as follows:

```
<?php
// Load a MySQL Driver
$mysql = Example::factory('MySQL');

// Load a SQLite Driver
$sqlite = Example::factory('SQLite');
?>
```

Singleton

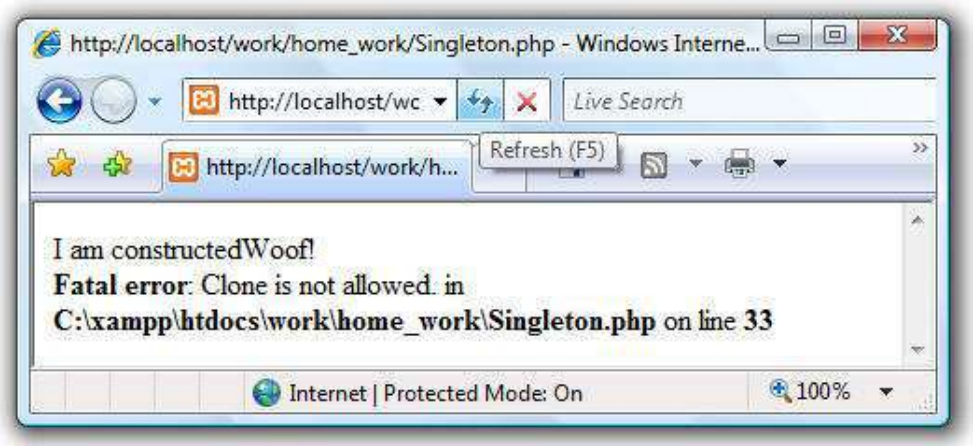
The Singleton pattern applies to situations in which there needs to be a single instance of a class. The most common example of this is a database connection. Implementing this pattern allows a programmer to make this single instance easily accessible by many other objects.

Example #2 Singleton Function

```
<?php
class Example
{
    // Hold an instance of the class
    private static $instance;

    // A private constructor; prevents direct creation of object
    private function __construct()
    {
        echo 'I am constructed';
    }

    // The singleton method
    public static function singleton()
    {
        if (!isset(self::$instance)) {
            $c = __CLASS__;
            self::$instance = new $c;
        }
        return self::$instance;
    }

    // Example method
    public function bark()
    {
        echo 'Woof!';
    }

    // Prevent users to clone the instance
    public function __clone()
    {
        trigger_error('Clone is not allowed.', E_USER_ERROR);
    }
}
?>
```

This allows a single instance of the Example class to be retrieved.

```
<?php
// This would fail because the constructor is private
//$test = new Example;

// This will always retrieve a single instance of the class
$test = Example::singleton();
$test->bark();

// This will issue an E_USER_ERROR.
$test_clone = clone $test;
?>
```



Late Static Bindings

This feature was named "late static bindings" with an internal perspective in mind. "Late binding" comes from the fact that `static::` will no longer be resolved using the class where the method is defined but it will rather be computed using runtime information. It was also called a "static binding" as it can be used for (but is not limited to) static method calls.

Limitations of self::

Static references to the current class like `self:: or __CLASS__` are resolved using the class in which the function belongs, as in where it was defined:

Example #1 self:: usage

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        self::who();
    }
}

class B extends A {
    public static function who() {
         echo __CLASS__;
    }
}

B::test();
?>
```

The above example will output:

```
A
```

Example #2 static:: simple usage

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        static::who(); // Here comes Late Static Bindings
    }
}

class B extends A {
    public static function who() {
         echo __CLASS__;
    }
}
B::test();
?>
```

The above example will output:

```
B
```

Example #3 static:: usage in a non-static context

```
<?php
class TestChild extends TestParent {
    public function __construct() {
        static::who();
    }

    public function test() {
        $o = new TestParent();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class TestParent {
    public function __construct() {
        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}
$o = new TestChild;
$o->test();

?>
```

The above example will output:

```
TestChild
TestParent
```

Example #4 Forwarding and non-forwarding calls

```
<?php
class A {
    public static function foo() {
        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class B extends A {
    public static function test() {
        A::foo();
        parent::foo();
        self::foo();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}
class C extends B {
    public static function who() {
        echo __CLASS__."\n";
    }
}

C::test();
?>
```

The above example will output:

```
A
C
C
```

Example #5 Late static bindings inside magic methods

```
<?php
class A {
```

```
   protected static function who() {
        echo __CLASS__."\n";
   }

   public function __get($var) {
       return static::who();
   }
}

class B extends A {

   protected static function who() {
        echo __CLASS__."\n";
   }
}
$b = new B;
$b->foo;
?>
```

The above example will output:

```
B
```

Objects and references

One of the key-point of PHP5 OOP that is often mentioned is that "objects are passed by references by default" This is not completely true. This section rectifies that general thought using some examples.

Example #1 References and Objects

```
<?php
class A {
    public $foo = 1;
}

$a = new A;
$b = $a;     // $a and $b are copies of the same identifier
             // ($a) = ($b) = <id>
$b->foo = 2;
echo $a->foo."\n";
```

```
$c = new A;
$d = &$c;    // $c and $d are references
             // ($c,$d) = <id>

$d->foo = 2;
echo $c->foo."\n";


$e = new A;

function foo($obj) {
    // ($obj) = ($e) = <id>
    $obj->foo = 2;
}

foo($e);
echo $e->foo."\n";

?>
```

The above example will output:

```
2
2
2
```

Interfaces

Object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled.

Interfaces are defined using the interface keyword, in the same way as a standard class, but without any of the methods having their contents defined.

All methods declared in an interface must be `public`, this is the nature of an interface.

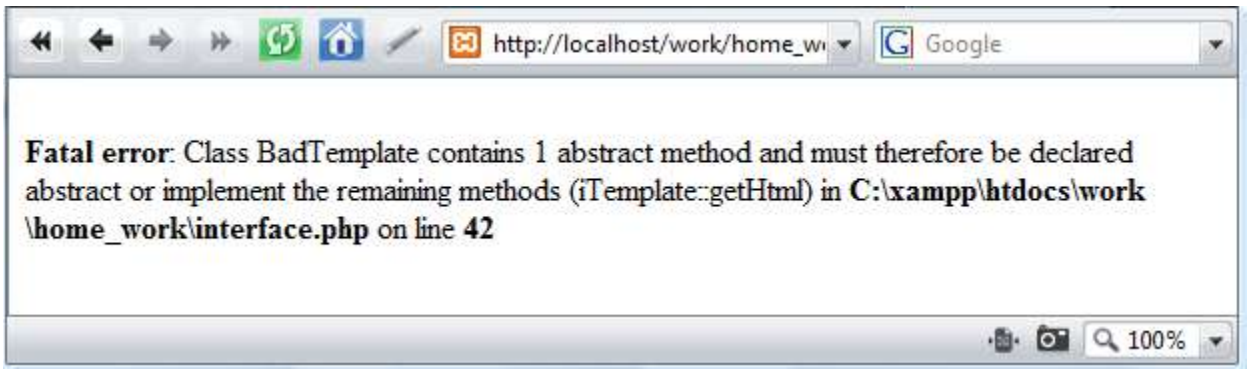
implements

To implement an interface, the implements operator is used. All methods in the interface must be implemented within a class; failure to do so will result in a fatal error. Classes may implement more than one interface if desired by separating each interface with a comma.

Example # Interface example

```
<?php
	// Declare the interface 'iTemplate'
	interface iTemplate
	{
		public function setVariable($name, $var);
		public function getHtml($template);
	}

	// Implement the interface
	// This will work
	class Template implements iTemplate
	{
		private $vars = array();

		public function setVariable($name, $var)
		{
			$this->vars[$name] = $var;
		}

		public function getHtml($template)
		{
			foreach($this->vars as $name => $value) {
			   $template = str_replace('{' . $name . '}', $value, $template);
			}
			return $template;
		}
	}

	// This will not work
	// Fatal error: Class BadTemplate contains 1 abstract methods
	// and must therefore be declared abstract (iTemplate::getHtml)
	class BadTemplate implements iTemplate
	{
		private $vars = array();

		public function setVariable($name, $var)
```

```
		{
			$this->vars[$name] = $var;
		}
	}
?>
```



Exceptions

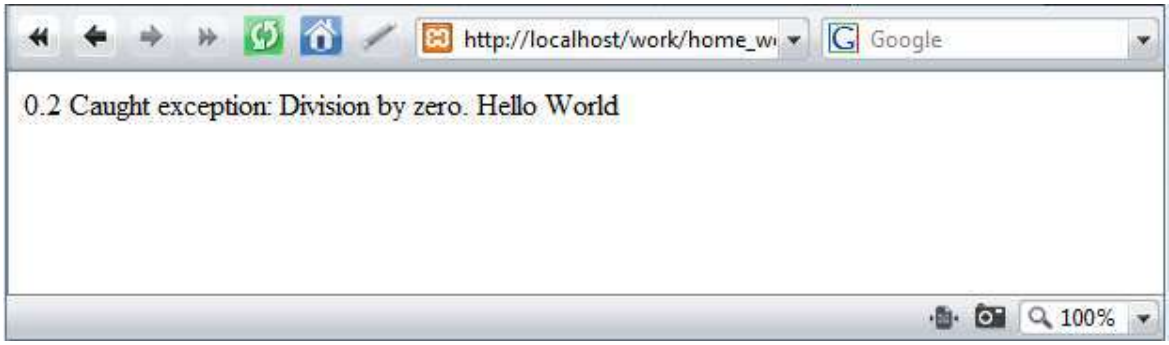
PHP 5 has an exception model similar to that of other programming languages. An exception can be `thrown`, and caught ("`catched`") within PHP. Code may be surrounded in a `try` block, to facilitate the catching of potential exceptions. Each try must have at least one corresponding catch block. Multiple catch blocks can be used to catch different classes of exeptions.

When an exception is thrown, code following the statement will not be executed, and PHP will attempt to find the first matching catch block. If an exception is not caught, a PHP Fatal Error will be issued with an "Uncaught Exception ..." message, unless a handler has been defined with `set_exception_handler()`.

Example #1 Throwing an Exception

```
<?php
	function inverse($x) {
		if (!$x) {
			throw new Exception('Division by zero.');
		}
		else return 1/$x;
	}

	try {
```

```
            echo inverse(5) . "\n";
            echo inverse(0) . "\n";
      } catch (Exception $e) {
            echo 'Caught exception: ',  $e->getMessage(), "\n";
      }

      // Continue execution
      echo 'Hello World';
?>
```



Example #2 Nested Exception

```
<?php
      class MyException extends Exception { }
      class Test {
            public function testing() {
                  try {
                        try {
                              throw new MyException('foo!');
                        } catch (MyException $e) {
                              /* rethrow it */
                              throw $e;
                        }
                  } catch (Exception $e) {
                        var_dump($e->getMessage());
                  }
            }
      }
      $foo = new Test;
      $foo->testing();
?>
```

The above example will output:

```
string(4) "foo!"
```