Module 11: Objects

CS 85:
PHP PROGRAMMING

Santa Monica College
Computer Science &
Information Systems Dept.

Introduction to Object Oriented Programming

The concept of Object Oriented Programming is the idea of a structure composed of related items such as multiple variables and functions. The term Object is a programming term used to describe the unit that contains these related items. The items within an Object can include members (also known as properties or variables) and methods (also known as functions). For example, a person is not limited to their name. A normal person has many charactistics or action the person can take. For example, properties could include the person $firstName, $lastName, $address, $cellNumber, etc. A person can take many different actions that can be coded as functions. When programming a function within an object, the function is referred to as a method. A person method() could include getMarried($sponseName); buyHome($loanAmt, $address), newJob($salary, $jobTitle).

Encapsulation

In order to understand the purpose of encapsulation you first need to understand the purpose of classes. Classes are the blueprints for objects, but more than that they are like the building blocks and tools used to make your application. Your classes should be designed to represent specific characteristics and functionality. In order for them to be used properly as they were designed, you will need to limit how users of the class can interact with those characteristics and functionality. When I speak of 'users of the class' I am talking about anyone who uses the class within their application, including you. After all these 'tools' and 'materials' were designed to help you build an application, and that is how they should be thought of when you design them. Encapsulation is the process of creating that interface for users of your class to use. This is done through the use of visibility modifiers, and interface functions that strictly define how a user can interact with your class. By doing this you ensure your class is being used as you intend for it, and limit the possibilities of miss-use and unexpected run-time errors. Below we will go over the process of encapsulating a class and explain in detail the purpose of each step with examples of use.

Classes

A class in PHP (as in most modern programming languages) is a way to group related functions (or methods) and variables (or members) together that create the Object. The class is a blueprint that is used to define the structure of each new Object. When a ***new*** Object is created, you essentially are creating a new instance of the class. The new instance inherits all the methods and properties of the class.

Each property or method declared can specific it's visibility that is either explicit (from "private", "protected" and "public") or implicit (that is, omission of a declaration, only allowed for methods for which the default value is "public").

An object is an instance of a class, in which its class's methods (if any) can be called, and its class's members (if any) can be provided with new values overriding the default ones.

Sample Person Class:

```php
class Person
{
    //Class Person, two private properties $_name & $_age
    //Note $_ in the property name. The underscore
    // used for private variables/properties
    private $_name;
    private $_age;
}
```

Sample Html Class:

```
<?php

/* Start the class */
class Html {

    /*
      Declare a public member named "br" with a default value of
"&lt;br /&gt;" (the HTML line break)
    */
    public $br = '&lt;br /&gt;';

    /*
      Declare a public (by default) method named "printLn" requiring
an argument
      named "message"
    */
    function printLn ($message) {

        /*
          Echo the "message" argument followed by the value of the
"br" member (accessed through the use of the "$this" keyword (that
represents the current object), the "->" operator (used to access
members) and the member's name ''without'' the "$" character.
        */
        echo $message . $this->br;

    }
}
?>
```

However, just because you've created a class doesn't mean you get to use anything in it. A class is an abstract entity. It is a way of saying "here is how to create Html-type objects, and what they do." Before you can do anything with the members of a class, you must make an *instance* of that class.

That is, you must actually make an "Html" object and work with that.

Objects

Once you have defined a class, you will want to try it out. Objects are instances of a class. create objects in PHP is as follows, by using the ***new*** operator to create the Object.

```
<?php
$myPage = new Html; // Creates an instance of the Html class
$myPage->printLn('This is some text'); // Prints "This is some
```

```
text&lt;br /&gt;"
?>
```

The new concepts in that piece are:

- The `new` keyword — Used to make a new instance of an object, in this case an object of the class `Html`.
- The `->` operator — Used to access a member or method of an instantiated object, in this case, the method `printLn()`

Now, the fact that the representation of a line break is defined within a member of the class allows you to change it. Note that the changes will only concern an instance, not the whole class. Here is an example (note that "<br>" is an HTML line break and "<br />" is an XHTML line break):

```
<?php
// Creates an instance of the Html class
$myPage = new Html;

//Creates another instance of the Html class
$myOldHtmlPage = new Html;

/* Changes the value of the "br" member of the "$myOldHtmlPage"
instance to the old HTML value *
$myOldHtmlPage->br = "&lt;br&gt;";

/* Prints "This is some text&lt;br&gt;", the new value of the member
is used */
$myOldHtmlPage->printLn('This is some text');

/* Prints "This is some text&lt;br /&gt;", the default value of the
member is kept */
$myPage->printLn('This is some text');
?>
```

However, note that it is better to avoid modifying an instance's members: to allow the instance for possible constraint validation, one should instead add to the class (and use while dealing with instances) "get_br" and "set_br" methods to get and set the value of the member while allowing the instance to perform tests and possibly reject the change.

Scope

The scope of an item of a class defines who is allowed either to call it (for methods) or to read / change its value (for properties).

Scope definition is compulsory for properties and facultative for methods where "public" is the default value if nothing is specified.

Public

Unless you specify otherwise, properties and methods of a class are public. That is to say, they may be accessed in three possible situations –

- From outside the class in which it is declared
- From within the class in which it is declared
- From within another class that implements the class in which it is declared

Till now we have seen all members as public members. If you wish to limit the accessibility of the members of a class then you define class members as **private** or **protected**.

Properties

Even though declaring class properties as public makes their use easier, it is generally not recommended. Should you decide in future versions of the class to perform any kind of check on a public property, or store it in another additional property for caching purposes, you couldn't, since the rest of the code would still use the public property without any check.

Therefore, one should prefer (except in special cases) not allowing public access to properties, and provide simple "get_foo" and "set_foo" methods to return or overwrite them, leaving open the possibility of adding complexity if required.

Methods

Since the methods provide the actual interface between an object and the rest of the world, it makes sense for almost all of your methods to be in public access.

More precisely, any method intended to be called from the outside should be public. However, methods only meant for internal use and that aren't part of the interface should not be public. A typical example being an object abstracting a connection with a database while providing a cache system. Different public methods should exist to allow interaction with the outside world, but, for example, a method sending a raw SQL request to the database should not be public, as any foreign code calling it would bypass the cache and potentially cause useless load or (worse) loss of data as the database's data may be outdated (if the latent changes have taken place in the cache).

Visibility Modifiers

In PHP there are three visibility modifiers; private, protected, and public. These modifiers are used to define what level of visibility a variable or function within your class has. Each of the modifier keywords represent a level at which the variable or function can be seen outside and within the class. When we talk about ‘seen outside’ the class, I am referring to the interaction with an object that was instantiated from your class. If your class represents a dog and I create a new dog object from your class, I can then interact with the dog characteristics that are visible outside of the class. If you think of it just like in the real world, if you have a dog in front of you, you can interact with that dog. You can clearly see the dogs visible characteristics such as size, color, and probably even breed. You can also perform actions with the dog such as petting it, giving it a command, playing fetch, etc. Other characteristics and activity is not so visible however such as the dogs temperament or health, or the actions that keep the dogs heart beating and lungs working. They are very important to the makeup of the dog, however your interaction with the dog is limited only to what is visible to you

Public

Public keyword modifier is used to define a variable or function as being visible to everyone, everywhere. When this modifier is used the variable or function following is visible to the users using your class as well as all the inner workings of your class. This modifier is the only modifier that allows interaction outside of the class

Protected

keyword modifier is similar to the Private keyword modifier in that it is visible inside of the class but not to users outside the class. In addition to that however, the protected keyword also allows child classes that extend the class to see the characteristics as well. This is useful when inheriting characteristics from a more generic class. See Inheritance.

Private

Private keyword modifier is used to define a variable or function as being visible only to the inside of the class. When this modifier is used the variable or function following is invisible to the users using your class, but all the code that you write inside your class will be able to see it.

Practical use

This example doesn't make the case for classes and objects. Why go to the bother of creating all that extra structure just to access functions?

Let's give an example that can show how this sort of thing would be useful:

```php
<?php

class Html {
    private $source = "";
    public function printLn ($message) {
        echo $this->source .= $message . "&lt;br /&gt;";
    }
    public function show () {
        echo $this->source;
    }
}

$elvis = new Html();
$goth = new Html();

$elvis->printLn("Welcome to my Elvis Fan Page! Uh-huh, uh-huh, uh-
huh.");
$goth->printLn("Entree the Goth Poetry Labyrinth of Spoooky
Doooommmm...");
$elvis->show();

?>
```

Some things to note:

- The statement echo $this->source .= $message ."&lt;br /&gt;"; first changes the value of the private property $source and then prints the changed value. The changed value is saved.
- A different copy of the $source property exists within each instance of the Html class.
- The printLn and show functions refer to the same copy of $source that was called for their object. (When I call $elvis's printLn function, it changes $elvis's $source variable and the statement $elvis->show(); prints the changed value of $source).

- Using standard variables and methods, there would be a greater risk of sending the wrong content to the wrong page. Which could be horribly confusing for my website visitors.

Now we can start to see how we could save some time and trouble using classes and objects. We can now easily manage two potential web pages at once, printing the correct content to each. When done, we can merely call a page's show function to send that page's html source to the output stream.

I could add more features to all of the objects just by adding variables and functions to the governing class, and I could add more objects at will just by declaring them. In fact, the more complicated my program gets, the easier everything is to manage using object-oriented programming.

Inheritance

Inheritance is the extension of a class. A child class has all the properties and methods of its parent class in additional to the properties and methods of its own class blueprint. Inheritance is one of the core concepts in object oriented programming. PHP supports inheritance like other object oriented language supports inheritance.

Example 1: pets

For example, pets generally share similar characteristics, regardless of what type of animal they are. Pets eat, and sleep, and can be given names. However, the different types of pet also have their own methods: dogs bark and cats meow. Below is an implementation of this:

```php
<?php

class Pet
{
    var $_name;

    function Pet($name) {
        $this->_name = $name;
    }

    function eat() {
    }

    function sleep() {
    }
}

class Dog extends Pet
{
    function bark() {
    }
}

class Cat extends Pet
{
    function meow() {
    }
}
```

```
$dog = new Dog("Max");
$dog->eat();
$dog->bark();
$dog->sleep();

$cat = new Cat("Misty");
$cat->eat();
$cat->meow();
$cat->sleep();
?>
```

Likewise, we could use the PHP5 syntax for our inherited class:

```
<?php
class Pet
{
    var $_name

   public function __construct($name) {
       $this->_name = $name;
   }

   function eat() {
   }

   function sleep() {
   }
}
?>
```

Example 2: persons

Consider two persons one the parent and his child. By definition the child would have inherited certain properties from the parent. So the child might have all the characteristics of the parent and in addition to that the child might have additional characteristics. With this analogy in mind consider two classes Person and programmer the base class has the following code.

```
<?php
class Person{
    var $legs=2;
    var $head=1;

    function walk(){
        echo "Walk";
     }
}
?>
```

The Person class has two attributes $legs and $head and it has one method walk. Suppose there is another class programmer. The programmer class has all this attributes and methods so you can define it in again or you can just inherit from the Person class. So you can define the programmer class as follows.

```php
<?php
class programmer extends Person{
    var $programmingSkill;
}
?>
```

No this that all we have to do is just use the keyword **extends** and then followed by base class then the all the properties of the base class are inherited which means we can do this.

```php
<?php

$jedai = new programmer();
echo $jedai->legs;
echo $jedai->head;
echo $jedai->walk();

?>
```

Class Constructor

When working with Objects, its useful to initialize the properties for example the Person object name or a Bank object account balance. PHP uses a special method called the constructor. The constructor method is called __constructor(). When any new Object is created, the __constructor() method will automatically be called.

The constructor method below takes two parameters, $name and $age, where age is a default parameter, meaning if the user does not specify an $age, it will use the default value of 0. The if statement checks to make sure $age argument is an integer before setting it. If it is not an integer value then throw an error, forcing our users to use the class the way we intend for them to use it.

Like a constructor function you can define a destructor function using function __destruct(). You can release all the resources with-in a destructor.

```php
<?php
class Person
{
    private $_name;
    private $_age;

    function __construct($name, $age = 0)
    {
        if (!is_int($age))
        {
            throw new Exception("Cannot assign non integer value to integer field,
'Age'");
        }

        $this->_age = $age;
        $this->_name = $name;
    }
}
?>
```

Constructor Overloading

Constructor overloading allows you to create multiple constructors with the same name __construct() but different parameters. Constructor overloading enables you to initialize object's properties in various ways. The following example demonstrates the idea of constructor overloading:

```
<?php
class BankAccount{

    private $accountNumber;
    private $totalBalance;

    public function __construct(){
    }

    public function __construct($accountNo){
        $this->accountNumber = $accountNo;
    }

    public function __construct($accountNo, $initialAmount){
        $this->accountNumber = $accountNo;
        $this->totalBalance = $initialAmount;
    }
}
?>
```

We have three constructors:

- The first constructor is empty, it does nothing.
- The second one only initializes account number.
- The third one initializes both account number and initial amount.

PHP have not yet supported constructor overloading, Oops. Fortunately, you can achieve the same constructor overloading effect by using several PHP functions.

Encapsulate Example

Encapsulate $_age private property is the process of giving the user a way to change or set the person's age through an interface method. This is done simply be creating a public method the user can call. The setAge() method only requires a single parameter, $age. We know age must be an integer so we check the user's input to make sure it is an integer first. If it isn't we throw an error, forcing the user to use our interface function the way we intend it be used. If the condition passes then our private age variable is set to the value of the user's input $age.

```
public function setAge($age )
{
    if (!is_int($age))
    {
        throw new Exception("Cannot assign non integer value to
integer field, 'Age'");
```

```
    }

    $this->_age = $age;
}
```

Traits

Traits are a mechanism for code reuse in single inheritance languages such as PHP. A Trait is intended to reduce some limitations of single inheritance by enabling a developer to reuse sets of methods freely in several independent classes living in different class hierarchies. The semantics of the combination of Traits and classes is defined in a way which reduces complexity, and avoids the typical problems associated with multiple inheritance and Mixins.

A Trait is similar to a class, but only intended to group functionality in a fine-grained and consistent way. It is not possible to instantiate a Trait on its own. It is an addition to traditional inheritance and enables horizontal composition of behavior; that is, the application of class members without requiring inheritance.

Example:

```
<?php
trait MyTrait1
{
  function Hello() {
    print 'Hello';
  }
}

trait MyTrait2
{
  function World() {
    print 'World';
  }
}



class MyClasse1
{
    use MyTrait1;
    use MyTrait2;

    function __construct() {
        $this->Hello();
        $this->World();
        }
}
```