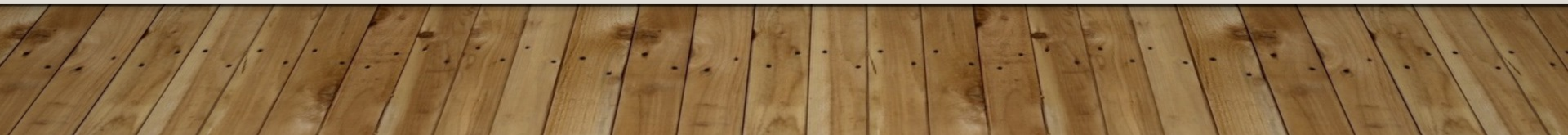
ISIT307 - WEB SERVER PROGRAMMING

LECTURE 6 – PHP: OBJECT-ORIENTED PROGRAMMING



LECTURE PLAN

- Study object-oriented programming concepts
- Use objects in PHP scripts
- Declare data members in classes
- Work with class member functions
- Inheritance
- Polymorphism, Interfaces, Abstract classes, Traits

OBJECT-ORIENTED PROGRAMMING – KEY CONCEPTS

- object,
- encapsulation,
- association,
- aggregation,
- delegation,
- composition,
- dynamic binding,
- polymorphism,
- inheritance,
- hierarchical objects,
- abstract classes

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

- **Object-oriented programming** (OOP) refers the concept of merging related variables and functions into a single interface
- An **object** refers to programming code and data that can be treated as an individual unit or component
- Objects are often also called **components**
- Object orientedness can be considered as co-operative problem solving through objects communicating with one another.

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

- **Data** refers to information contained within variables or other types of storage structures
- The variables that are associated with an object are called **properties** or **attributes**
- The functions associated with an object are called **methods**

UNDERSTANDING ENCAPSULATION

- Objects are **encapsulated** – all code and required data are contained within the object itself
- Encapsulated objects hide all internal code and data
- An object **interface** refers to the methods and properties that are required for a source program to communicate with an object

UNDERSTANDING ENCAPSULATION

- Encapsulated objects allow users to see only the methods and properties of the object that you allow them to see
- Encapsulation reduces the complexity of the code
- Encapsulation prevents other programmers from accidentally introducing a bug into a program, or stealing code

OBJECT-ORIENTED PROGRAMMING AND CLASSES

- The code, methods, attributes, and other information that make up an object are organized into **classes**
- An **instance** is an object that has been created from an existing class
- Creating an object from an existing class is called **instantiating** the object
- An object inherits its methods and properties from a class — it takes on the characteristics of the class on which it is based

USING OBJECTS IN PHP SCRIPTS

- In PHP the declaration of an object is by using the **new** operator
- The syntax for instantiating an object is:

```
$objectName = new ClassName();
```

- After an object is instantiated, a hyphen and a greater-than symbol (->) is used to access the methods and properties contained in the object
- Together, these two characters are referred to as **member selection notation**

DEFINING CUSTOM PHP CLASSES

- Classes:
  - Help make complex programs easier to manage
  - Hide information that users of a class do not need to access or know about
  - Make it easier to reuse code or distribute your code to others for use in their programs
- Inherited characteristics allow you to build new classes based on existing classes without having to rewrite the code contained in the existing one

DEFINING CUSTOM PHP CLASSES

- The functions and variables defined in a class are called **class members**
- Class variables are referred to as **data members** or **member variables** or **attribute** or **property**
- Class functions are referred to as **member functions** or **function members** or **operation** or **method**

CREATING A CLASS DEFINITION

- To create a class in PHP, the `class` keyword is used in a class definition
- A **class definition** contains the data members and member functions that make up the class
- The syntax for defining a class is:

```
class ClassName {
  data member and member function definitions
}
```

CREATING A CLASS DEFINITION

- The **`ClassName`** portion of the class definition is the name of the new class
- Class names usually begin with an uppercase letter to distinguish them from other identifiers
- Example

```
class BankAccount {
      data member and member function definitions
}
$checking = new BankAccount();
```

CREATING A CLASS DEFINITION

- Some useful functions:
  - `get_class()` function can be used to retrieve the name of the class
  - `class_exists()` can be used to determine if a class exists
  - `instanceof` operator can be used to determine whether an object is instantiated from a given class

```
$checking = new BankAccount();
echo 'The $checking object is from' .
     get_class($checking) . " class.</p>";
---
if ($checking instanceof BankAccount) {…}
---
if (class_exists("BankAccount")) {…}
```

STORING CLASSES IN EXTERNAL FILES

- Good programming practice is to store the classes in separate file and include in the code using:
- `include()`,`require()`, `include_once()` or `require_once()` functions

INFORMATION HIDING

- **Information hiding** states that other programmers, sometimes called clients, do not need to access or know about should be hidden
- Helps minimize the amount of information that needs to pass in and out of an object
- Reduces the complexity of the code that clients see
- Prevents other programmers from accidentally introducing a bug into a program by modifying a class's internal workings

USING ACCESS SPECIFIERS

- **Access specifiers** control a client's access to individual data members and member functions
- There are three levels of access specifiers in PHP
  - `public`, `private`, and `protected`
- The **`public` access specifier** allows anyone to call a class's member function or to modify and retrieve a data member

USING ACCESS SPECIFIERS

- The **`private` access specifier** prevents clients from calling member functions or accessing data members and is one of the key elements in information hiding
  - Private access does not restrict a class's internal access to its own members
  - Private access restricts clients from accessing class members
- The **`protected` access specifier** prevents clients from calling member functions or accessing data members, but allows inherited (child) classes to have access

USING ACCESS SPECIFIERS

- The access specifier must be included at the beginning of a data member declaration statement

```
class BankAccount {
    public $balance = 0;
}
```

- A data member should be assigned a value in the declaration

```
class BankAccount {
    public $balance = 0;
}
---
class BankAccount {
    public $balance = 1 + 2; //invalid
}
```

WORKING WITH MEMBER FUNCTIONS

- **public** member functions should be declared all functions that clients need to access
- **private** member functions should be declared all functions that clients do not need to access
- If the member function is not assigned access specifiers, is by default public

USING THE `$THIS` REFERENCE

- Outside of a class, the members of the object can be referred by using the name of the object, the member selection notation (->), and the name of the function or variable
- Within a class function definition, $this must be used to refer to the current object of the class

```
$this->accountNumber = 0;
```

WORKING WITH MEMBER FUNCTIONS

```
class BankAccount {
            public $balance = 958.20;
            public function withdrawal($Amount) {
                $this->balance -= $Amount;
            }
}
if (class_exists("BankAccount"))
   $checking = new BankAccount();
else
   exit("<p>The BankAccount class is not available!</p>");

echo "<p>Your checking account balance is \${$checking->balance}</p>";
$cash = 200;
$checking->withdrawal($cash);
echo "<p>After withdrawing \${$cash}, your checking account balance is
                                        \${$checking->balance}</p>";
```

INITIALIZING WITH CONSTRUCTOR FUNCTIONS

- A **constructor function** is a special function that is called automatically when an object from a class is instantiated

```
class BankAccount {
        private $accountNumber;
        private $customerName;
        private $balance;
        function __construct() {
                $this->accountNumber = 0;
                $this->balance = 0;
                $this->customerName = "";
        }
}
```

INITIALIZING WITH CONSTRUCTOR FUNCTIONS

- Constructor can have input arguments, or optional input arguments
- In the older version of PHP (PHP4) the constructor can be defined with the same name as the class, but it is deprecated in PHP7 and is not recognized as constructor in PHP8
- PHP 8.0 offers Constructor Property Promotion
  - allows class property declaration and constructor assignment right from the constructor

INITIALIZING WITH CONSTRUCTOR FUNCTIONS

- Constructor Property Promotion

```
class BankAccount{
    private float $balance;
    public function __construct(float $balance) {
        $this->balance = $balance;
    }
}
_ _ _
class BankAccount {
    public function __construct(private float $balance) {}
}
- - -
class BankAccount {
    private string $name;
    public function __construct(private float $balance, string $name) {
        $this->name = $name; }
}
```

CLEANING UP WITH DESTRUCTOR FUNCTIONS

- A **destructor** function is called when the object is destroyed
- A destructor function cleans up any resources allocated to an object after the object is destroyed
- A destructor function is commonly called in two ways:
  - When a script ends
  - When you manually delete an object with the `unset()` function
- The destructor syntax is:

```
function __destruct(){...}
```

CLEANING UP WITH DESTRUCTOR FUNCTIONS - EXAMPLE

```
function __construct() {
      $conn = new mysqli("localhost","root","","test");
}
function __destruct() {
      $conn->close();
}
```

WRITING ACCESSOR AND MUTATOR FUNCTIONS

- Accessor/Mutator functions are public member functions that a client can call to retrieve or modify the value of a data member
- These functions often begin with the words "set" or "get"
- Set functions modify data member values (mutator)
- Get functions retrieve data member values (accessor)

WRITING ACCESSOR AND MUTATOR FUNCTIONS - EXAMPLE

```
class BankAccount {
    private $balance=0;
    function __construct($bal=0)
    {     $this->balance = $bal;  }
    public function setBalance($newValue)
    {     if ($newValue >0)
               $this->balance = $newValue;  }
    public function getBalance()
    {     return $this->balance;    }
}
 $checking = new BankAccount();
 $checking->setBalance(100);
 echo "<p>Your checking account balance is " . $checking->getBalance()
                                                    . "</p>\n";
 $newCh = new BankAccount();
 $newCh->setBalance(-50);
 echo "<p>Your checking account balance is " . $newCh->getBalance()
                                                    . "</p>\n";
```

WRITING ACCESSOR AND MUTATOR FUNCTIONS

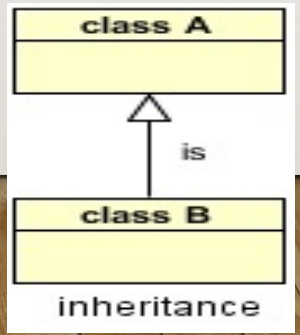
- Also can be used the magic functions:
  - `__set()` – utilized when writing data to inaccessible (protected or private) data member/property
  - `__get()` – utilized for reading data from inaccessible (protected or private) data member/property

```
class MyClass{
     private $myP;
     function __get($name)
     {    return $this->$name;}
     function __set($name, $value)
     {    $this->$name = $value; }
}
```

```
$myV = new MyClass();
$myV->myP = 5;
echo $myV->myP;
```

INHERITANCE

- One of the main advantages of object-oriented programming is the ability to reduce code duplication with inheritance
- Inheritance allows to write the code only once in the parent class, and then use the code in both the parent and the child classes
- By using inheritance, we can create a reusable piece of code that we write only once in the parent class, and use again as much as we need in the child classes



INHERITANCE

- In order to declare that one class inherits the code from another class, we use the `extends` keyword.

```
class Parent {
  // The parent's class code
}

class Child extends Parent {
  // The  child can use the parent's class code

}
```

INHERITANCE

- The child class can make use of all the non-private members (methods and properties) that it inherits from the parent class
- Child class can use the properties and methods of its parent class, but it can have properties and methods of its own as well
- While a child class can use the code it inherited from the parent, the parent class is not allowed to use the child class's code
- If there is need for a property or a method to be approached from both the parent and the child classes (but not to be public), it need to be declared as *protected*

INHERITANCE

- Child class can override the methods of the parent class by rewriting a method that exists in the parent, but assign to it a different code
- In order to prevent the method in the child class to be overridden, the method in the parent should have the prefix *final*
- If the child does not define a constructor or destructor then it may be inherited from the parent class just like a normal class method
- If the child does define a constructor or destructor, parent constructor/destructor are not called implicitly, so a call to `parent::__construct()` or `parent::__destruct()` within the child constructor/destructor is required

INHERITANCE – EXAMPLE (1)

```
<?php
class Car {
      private $model="";     // for the ex.3 it needs to be protected
      public function setModel($model)
      {     $this->model = $model; }
      public function hello()
      {     return "I am a <i>" . $this -> model . "</i><br />"; }
}
class SportsCar extends Car {
  //No code in the child class }

$sportsCar1 = new SportsCar(); //Create an instance from the child class

$sportsCar1->setModel('Jaguar');

echo $sportsCar1->hello();
?>
```

INHERITANCE – EXAMPLE (2)

```
…
class SportsCar extends Car{
  private $style = 'fast and furious';
  public function driveItWithStyle()
  {
    return $this->hello() . 'Drive me ' . '<i>' .
           $this->style . '</i>';
  }
}

$sportsCar1 = new SportsCar();
$sportsCar1->setModel('Ferrari');
echo $sportsCar1->driveItWithStyle();
?>
```

INHERITANCE – EXAMPLE (3)

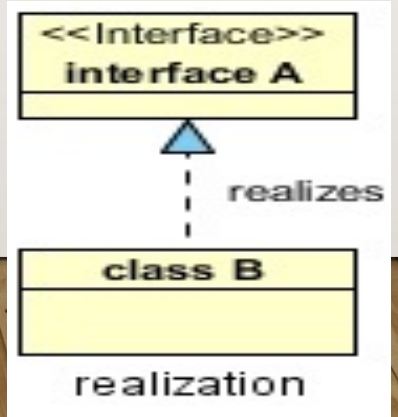
```
…
class SportsCar extends Car{
  private $style = 'fast and furious';
  public function driveItWithStyle()
  {
    return 'I am ' . $this->model . '! Drive me ' . '<i>' .
           $this->style . '</i>';
  }
  public function hello()
  {   return "I am a <i>overriden</i> method <br />"; }
}
$sportsCar1 = new SportsCar();
$sportsCar1->setModel('Ferrari');
echo $sportsCar1->driveItWithStyle();
echo $soprtsCar1->hello();
?>
```

POLYMORPHISM

- **Polymorphism** is the ability of a class instance to behave as if it were an instance of another class in its inheritance tree, most often one of its ancestor classes.
- Polymorphism simply means using the same function name to invoke one response in objects of the base class and another response in objects of a derived class.
- For instance, if there is one `area()` function for the `Figure` class, and another one for the `Circle` (derived) class, you have used the concept of polymorphism.

INTERFACES

- An Interface allows the users to create programs, specifying only the public methods that a class must implement, without involving the complexities and details of how the particular methods are implemented
- It is generally referred to as the next level of abstraction – *the class implements the interface*
- An Interface is defined using the `interface` keyword and declaring only the function prototypes



INTERFACES

```
interface MyInterfaceName
{
   public  function methodA();
   public  function methodB();
}
---
class MyClassName implements MyInterfaceName
{
   public  function methodA() {
     // method A implementation
   }
   public  function methodB(){
     // method B implementation
   }
}
```

ABSTRACT CLASSES

- Methods defined as abstract simply declare the method's signature - they cannot define the implementation
- Any class that contains at least one abstract method must be declared as abstract
- Classes defined as abstract can not be instantiated
- When inheriting from an abstract class
  - all abstract methods must be defined by the child class
  - additionally, these methods must be defined with the same (or a less restricted) visibility (for example, abstract method is defined as protected, the function implementation must be defined as either protected or public, but not private)
  - the signatures of the methods must match

ABSTRACT CLASSES

```
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";
    }
}
```

TRAITS

- A Trait is intended to group together functionality that may be reused in multiple classes
- A Trait includes the implementation of the functions
- It is not possible to instantiate a Trait on its own
- A Trait is intended to reduce some limitations of single inheritance and enables horizontal composition of behaviour - the application of class members without requiring inheritance

TRAITS

```
trait myTrait {
    function getTemp() { ... //implementation }
    function setTemp() { ... //implementation}
}


class MyClassA extends SomeClass {
    use myTrait;
    ...
}

class MyClassB extends OtherClass {
    use myTrait;
    ...
}
```

COLLECTING GARBAGE

- **Garbage collection** refers to cleaning up or reclaiming memory that is reserved by a program
- PHP knows when your program no longer needs a variable or object and automatically cleans up the memory for you
- The one exception is with open database connections

SERIALIZING OBJECTS

- **Serialization** refers to the process of converting an object into a string that you can store for reuse
- Serialization stores both data members and member functions into strings
- To serialize an object we use the `serialize()` function

```
$SavedAccount = serialize($checking);
```

SERIALIZING OBJECTS

- To convert serialized data back into an object, we use the `unserialize()` function

```
$checking = unserialize($savedAccount);
```

- To use serialized objects between scripts, a serialized object can be assigned to a session variable

```
session_start();
$_SESSION['SavedAccount'] = serialize($checking);
```

SERIALIZATION FUNCTIONS

- When PHP serialize an object with the `serialize()` function, it looks in the object's class for a special function named `__sleep()`
- The primary reason for including a `__sleep()` function in a class is to specify which data members of the class to serialize

```
function __sleep() {
    $serialVars = array('balance');
    return $serialVars; }
```

- If a `__sleep()` function is not included in the class, the `serialize()` function serializes all of its data members

SERIALIZATION FUNCTIONS

- When the `unserialize()` function executes, PHP looks in the object's class for a special function named `__wakeup()`
- The `__wakeup()` function can be used to perform many of the same tasks as a constructor function
  - it is called to perform any initialization the class requires when the object is restored (initialize data members, restore database or file connections, . . .)

WORKING WITH DATABASE CONNECTIONS AS OBJECTS

- To connect to the MySQL database server using object-oriented style, needs to instantiate an object from the `mysqli` class :

```
$conn = new mysqli("localhost", "root","", "test");
```

- This statement uses the `mysqli()` constructor function to instantiate a `mysqli` class object named `$conn`
- To explicitly close the database connection, the `close()` method of the `mysqli` class can be used

```
$conn->close();
```

HANDLING MYSQL ERRORS

- With object-oriented style there are few data members
  - `$conn->connect_errno, $conn->connect_error`
  - `$conn->error, $conn->errno`

```
try{
    $conn = new mysqli($servername, $username, $password, $db);
    echo "successful connection";
}
catch (mysqli_sql_exception $e)
{
    die ($e->getCode(). ": " . $e->getMessage());
}
$conn->close();
```

EXECUTING SQL STATEMENTS

- With object-oriented style, the `query()` method of the `mysqli` class should be used
- The resultset can be accessed by:
  - The `fetch_row()` method of the `mysqli` class
  - The `fetch_assoc()` method of the `msqli` class

EXECUTING SQL STATEMENTS

```
$servername = "localhost"; $username = "root"; $password = "";
$db = "myDB"; $TableName = "MyGuests";
try{
     $conn = new mysqli($servername, $username, $password, $db);
     $sql = "SELECT * FROM $TableName";
     $qRes = @$conn->query($sql);
     echo "<table width='100%' border='1'>\n";
     echo "<tr><th>ID</th><th>name</th><th>surname</th>" .
          "<th>email</th></tr>\n";
     while (($Row = $qRes->fetch_row()) !== null) {
          echo "<tr><td>{$Row[0]}</td>";
          echo "<td>{$Row[1]}</td>";
          echo "<td>{$Row[2]}</td>";
          echo "<td align='right'>{$Row[3]}</td>";
     }
     echo "</table>\n"; }
catch (mysqli_sql_exception $e) {
    die ($e->getCode(). ": " . $e->getMessage());}
$conn->close();
```

OBJECT-ORIENTED PHP

- Example - On-line store