ISIT307 - WEB SERVER PROGRAMMING

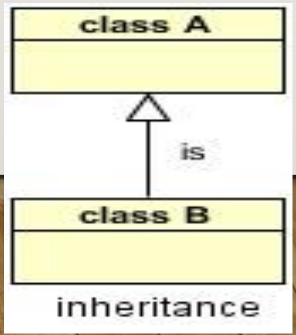
LECTURE 6.2 – PHP: OBJECT-ORIENTED PROGRAMMING – PART 2

LECTURE PLAN

- Inheritance
- Polymorphism
- Interfaces
- Abstract classes
- Traits

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();
?>
```

INHERITANCE – EXAMPLE (4)

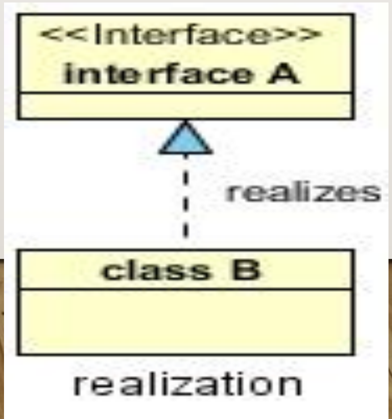
```
…
class SportsCar extends Car{

    private $style = 'fast and furious';

    public function __construct($model, $style)
    {    parent::__construct($model);
         $this->style = $style;
    }
...
}
```

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;
    ...
}
```

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, . . .)

OBJECT-ORIENTED PHP

- Example - On-line store