evaluation 3:report:

Abstract classes and methods are when the parent class has a named method, but need its child class(es) to fill out the tasks.

An abstract class is a class that contains at least one abstract method. An abstract method is a method that is declared, but not implemented in the code.

<?php  
abstract class ParentClass {  
  abstract public function someMethod1();  
  abstract public function someMethod2($name, $color);  
  abstract public function someMethod3() : string;  
}  
?>

So, when a child class is inherited from an abstract class, we have the following rules:

* The child class method must be defined with the same name and it redeclares the parent abstract method
* The child class method must be defined with the same or a less restricted access modifier
* The number of required arguments must be the same

<?php  
// Parent class  
abstract class Car {  
  public $name;  
  public function \_\_construct($name) {  
    $this->name = $name;  
  }  
  abstract public function intro() : string;  
}  
  
// Child classes  
class Audi extends Car {  
  public function intro() : string {  
    return "Choose German quality! I'm an $this->name!";  
  }  
}  
  
class Volvo extends Car {  
  public function intro() : string {  
    return "Proud to be Swedish! I'm a $this->name!";  
  }  
}  
  
class Citroen extends Car {  
  public function intro() : string {  
    return "French extravagance! I'm a $this->name!";  
  }  
}  
  
// Create objects from the child classes  
$audi = new audi("Audi");  
echo $audi->intro();  
echo "<br>";  
  
$volvo = new volvo("Volvo");  
echo $volvo->intro();  
echo "<br>";  
  
$citroen = new citroen("Citroen");  
echo $citroen->intro();  
?>

CONCRETE CLASS:

The class which implements an interface is called the Concrete Class. It must implement all the methods defined in an interface.

<?php

interface MyInterfaceName{

    public function method1();

    public function method2();

}

class MyClassName implements MyInterfaceName{

    public function method1(){

        echo "Method1 Called" . "\n";

    }

    public function method2(){

        echo "Method2 Called". "\n";

    }

}

$obj = new MyClassName;

$obj->method1();

$obj->method2();

?>

output: Method1 Called

Method2 Called

3)

HOW multiple inheritance can achieved in php:

PHP OOP does not allow multiple inheritance, it allow only multilevel inheritance.

In simple word, subclass can not extend more than one super class.  
But PHP allow hierarchical inheritance, Hierarchical inheritance means child can get property of their parent and parent can get property of grand parent, so in this way child can get also some property of their grand parent.

example:

1. **class** demo
2. **{**
3. //Your class body
4. **}**
5. **class** demo1
6. **{**
7. //Your class body
8. **}**
9. **class** demo3 **extends** demo1 demo2
10. **{**
11. //your class body
12. **}**
13. multilevel inheritance in php: **class** grandParent
14. **{**
15. //Body of grand parent class
16. **}**
17. **class** parent **extends** grandParent
18. **{**
19. //Body Of parent class
20. **}**
21. **class** child **extends** parent
22. **{**
23. //Body of child class
24. **}**
25. PHP does not support Multiple Inheritance but using interfaces in PHP, we can implement it.
26. **Using class with Traits:**
27. The trait is a type of class that enables multiple inheritance. Classes, objects, and traits that do not extend more than one class, but it can extend multiple traits at the same time.class chi
28. clsyntax:

syntax: class child\_class\_name extends parent\_class\_name

{

use trait\_name;

---------

----------

child\_class functions

}

program in folder

4)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.

According to precedence:

An inherited method from a base class is overridden by the method inserted into MyHelloWorld from the SayWorld Trait. The behavior is the same for methods defined in the MyHelloWorld class. The precedence order is that methods from the current class override Trait methods, which in turn override methods from the base class.

<?php  
class Base {  
    public function sayHello() {  
        echo 'Hello ';  
    }  
}  
  
trait SayWorld {  
    public function sayHello() {  
        parent::sayHello();  
        echo 'World!';  
    }  
}  
  
class MyHelloWorld extends Base {  
    use SayWorld;  
}  
  
$o = new MyHelloWorld();  
$o->sayHello();  
?>

output: Hello World!

Multiple traits usage:

<?php  
trait Hello {  
    public function sayHello() {  
        echo 'Hello ';  
    }  
}  
  
trait World {  
    public function sayWorld() {  
        echo 'World';  
    }  
}  
  
class MyHelloWorld {  
    use Hello, World;  
    public function sayExclamationMark() {  
        echo '!';  
    }  
}  
  
$o = new MyHelloWorld();  
$o->sayHello();  
$o->sayWorld();  
$o->sayExclamationMark();  
?>

The above example will output:

Hello World!

There are two trait members in php:

static trait members: Traits can define both static members and static methods.

<?php  
trait Counter {  
    public function inc() {  
        static $c = 0;  
        $c = $c + 1;  
        echo "$c\n";  
    }  
}  
  
class C1 {  
    use Counter;  
}  
  
class C2 {  
    use Counter;  
}  
  
$o = new C1(); $o->inc(); // echo 1  
$p = new C2(); $p->inc(); // echo 1  
?>

static methods:

<?php  
trait StaticExample {  
    public static function doSomething() {  
        return 'Doing something';  
    }  
}  
  
class Example {  
    use StaticExample;  
}  
  
Example::doSomething();  
?>

properties:

<?php  
trait PropertiesTrait {  
    public $x = 1;  
}  
  
class PropertiesExample {  
    use PropertiesTrait;  
}  
  
$example = new PropertiesExample;  
$example->x;  
?>

Abstract trait members:

Traits support the use of abstract methods in order to impose requirements upon the exhibiting class.

<?php  
trait Hello {  
    public function sayHelloWorld() {  
        echo 'Hello'.$this->getWorld();  
    }  
    abstract public function getWorld();  
}  
  
class MyHelloWorld {  
    private $world;  
    use Hello;  
    public function getWorld() {  
        return $this->world;  
    }  
    public function setWorld($val) {  
        $this->world = $val;  
    }  
}  
?>

Namespaces in php:

PHP Namespaces are the way of encapsulating items so that same names can be reused without name conflicts.

* It can be seen as an abstract concept in many places. It allows **redeclaring** the same functions/classes/interfaces/constant functions in the separate namespace without getting the fatal error.
* A namespace is a hierarchically labeled code block holding a regular PHP code.
* A namespace can contain valid PHP code.
* Namespace affects following types of code: classes (including abstracts and traits), interfaces, functions, and constants.
* Namespaces are declared using the namespace keyword.

A namespace must be declared the namespace at the top of the file before any other code – with one exception: the declare keyword.

<?php

namespace MyNamspaceName {

    // Regular PHP code

    function hello()

    {

        echo 'Hello namespace!';

    }

}

?>

if namespace declared globally then within single php code:

<?php

namespace {

    // Global space!

}

?>

Multiple namespaces can be declared within a single PHP code.

<?php

namespace MyNamespace1 {

}

namespace MyNamespace2 {

}

namespace {

}

?>

Just like directories, namespace can contain a hierarchy know as subnamespaces. PHP uses the backslash as its namespace separator.

<?php

namespace MyNamespaceName;

function hello()

    {

        echo 'Hello I am Running from a namespace!';

    }

// Resolves to MyNamespaceName\hello

hello();

// Explicitly resolves to MyNamespaceName\hello

namespace\hello();

?>

**Aliasing in Namespaces:**

Importing is achieved by using the ‘use’ keyword.

<?php

namespace MyNamespaceName;

require 'project/database/connection.php';

use Project\Database\Connection as Connection;

$connection = new Connection();

use Project\Database as Database;

$connection = new Database\Connection();

?>

5)Dependency injection in php:

Dependency Injection is a design pattern that helps *avoid hard-coded dependencies* for some piece of code or software.

The dependencies can be changed at *run time as well as compile time*. We can use Dependency Injection to write modular, testable and maintainable code:

* **Modular**: The Dependency Injection helps create completely self-sufficient classes or modules
* **Testable**: It helps write testable code easily eg unit tests for example
* **Maintainable**: Since each class becomes modular, it becomes easier to manage it.
* injection is nothing but injecting a dependency explicitly.

**class User { private $database = null;**

**public function \_\_construct(Database $database)**

**{ $this->database = $database;**

**}**

**public function getUsers()**

**{**

**return $this->database->getAll('users'); }**

**}**

**$database = new Database('host', 'user', 'pass', 'dbname');**

**$user = new User($database);**

**$user->getUsers();**

we are injecting it to a constructor:

**public function \_\_construct(Database $database)**

**we are passing database instance:**

**$database = new Database('host', 'user', 'pass', 'dbname'); $user = new User($database); $user->getUsers();**

We have already made database dependency explicit by requiring it into the constructor of the User class:

**public function \_\_construct(Database $database)**

There are three ways you can inject dependencies:

* Constructor Injection
* Setter Injection
* Interface Injection

**Constructor Injection:**

* A dependency is **required** and class can't work without it. By using constructor injection. we make sure all its required dependencies are passed.
* Since constructor is called only at the time of instantiating a class, we can make sure that its dependencies cant be changed during the life time of the object.

Constructor injection suffer from one problem though:

* Since constructor has dependencies, it becomes rather difficult to extend/override it in child classes.

**Setter Injection:**

Unlike Constructor injection which makes it **required** to have its dependencies passed, setter injection can be used to have **optional dependencies**. Let's pretend that our User class doesn't require Database instance but uses optionally for certain tasks.

**class User { private $database = null;**

**public function setDatabase(Database $database)**

**{ $this->database = $database;**

**} public function getUsers()**

**{ return $this->database->getAll('users'); }**

**} $database = new Database('host', 'user', 'pass', 'dbname');**

**$user = new User();**

**$user->setDatabase($database);**

**$user->getUsers();**

**Interface Injection**

In this type of injection, an interface enforces the dependencies for any classes that implement it, for example:

**interface someInterface {**

**function getUsers(Database $database);**

**}**