Unit-4 Object oriented concept and database connectivity

4.1 Classes and Objects

Classes:

A class is a blueprint or a template for creating objects. It defines the properties (attributes) and methods (functions) that objects of the class will have.

```
class Car {

    // Properties

    public $brand;

    public $model;

    public $color;


    // Methods

    public function startEngine() {

        echo "Engine started!";

    }


    public function brake() {

        echo "Braking...";

    }

}
```

In this example, the **Car** class has three properties (**$brand**, **$model**, **$color**) and two methods (**startEngine()** and **brake()**).

Objects:

An object is an instance of a class. You can create multiple objects based on the same class, and each object will have its own set of property values.

```
// Creating an object of the Car class

$myCar = new Car();
```

```
// Setting property values
$myCar->brand = "Toyota";
$myCar->model = "Camry";
$myCar->color = "Blue";

// Calling methods
$myCar->startEngine(); // Output: Engine started!
$myCar->brake();      // Output: Braking...
```

In this example, **$myCar** is an object of the **Car** class. You can access its properties using the arrow (**->**) notation.

```
<!DOCTYPE html>
<html>
<body>
<?php
class car {
  // Properties
  public $brand;
  public $model;
  public $color;
      // Methods
   public function startEngine($a) {
       echo "Engine started!";
       echo"<br>";
       echo $a;
   }
}
```

```
$myCar = new Car();
// Setting property values
$myCar->brand = "Toyota";
$myCar->model = "Camry";
$myCar->color = "Blue";

$myCar->startEngine($myCar->brand);
echo"<br>";
echo $myCar->brand;
echo "<br>";
echo $myCar->model;
?>

</body>
</html>
```

4.2 Access Modifiers

PHP supports access modifiers (**public**, **private**, **protected**) to control the visibility of properties and methods.

- **public**: The property or method is accessible from anywhere, both inside and outside the class.

```
class MyClass {
   public $publicProperty;

   public function publicMethod() {
      // Code here
   }
}
```

- **private**: The property or method is only accessible within the class where it is defined.

```
class MyClass {
   private $privateProperty;
```

```
    private function privateMethod() {
        // Code here
    }
}
```

- **protected**: The property or method is accessible within the class and its subclasses (child classes).

  ```
  class MyClass {
      protected $protectedProperty;


      protected function protectedMethod() {
          // Code here
      }
  }
  ```

4.3 Constructors and Destructors

A constructor is a special method that is automatically called when an object is created. It is used to initialize the object's properties.

Purpose:

Initialization of object properties.

Setup tasks that need to be performed when an object is created.

Key Points:

The constructor method is named __construct().

It can accept parameters for initialization.

If a class has no constructor, PHP will provide a default constructor.

```
class Car {
    public $brand;
    public $model;
    public $color;
```

```
    // Constructor

    public function __construct($brand, $model, $color) {

        $this->brand = $brand;

        $this->model = $model;

        $this->color = $color;

    }


    // Methods...

}
```

When you create an object of the **Car** class, you can pass values to the constructor:

```
$myCar = new Car("Toyota", "Camry", "Blue");
```

Destructors:

A destructor is a special method that is automatically called when an object is no longer referenced or when the script finishes execution. It is used to perform cleanup tasks, such as releasing resources (closing files, database connections, etc.).

Purpose:

Cleanup tasks or releasing resources before an object is destroyed.

Key Points:

The destructor method is named __destruct().

It does not take any parameters.

PHP automatically calls the destructor when the object is destroyed or goes out of scope.

```
class MyClass {

    // Destructor

    public function __destruct() {

        echo "Object destroyed!";

    }

}

// Creating an object
```

```
$myObject = new MyClass();

// Explicitly unset or let it go out of scope to trigger the destructor

unset($myObject); // Output: Object destroyed!
```

The destructor is called automatically when an object is destroyed, either explicitly by using **unset()** or when it goes out of scope.

Example with Constructor and Destructor:

```
class Resource {

   public function __construct() {

      echo "Resource initialized!";

   }

   public function performAction() {

      echo "Action performed!";

   }

   public function __destruct() {

      echo "Resource destroyed!";

   }

}

// Creating an object

$myResource = new Resource(); // Output: Resource initialized!

// Performing an action

$myResource->performAction(); // Output: Action performed!

// The destructor will be automatically called when the script finishes execution.
```

In this example, the constructor initializes the resource, the **performAction()** method does some work, and the destructor is automatically called when the script completes, indicating that the resource is destroyed.

4.4 Inheritance and Scope

PHP supports inheritance, allowing a class to inherit properties and methods from another class.

```
class SportsCar extends Car {

   // Additional properties and methods...

}
```

Inheritance allows you to create a new class (**SportsCar**) based on an existing class (**Car**). Inheritance allows a class (child class) to inherit properties and methods from another class (parent class). The child class can then extend or override the functionality of the parent class. This promotes code reusability and helps in creating a hierarchy of classes.

```
class Animal {
   public function makeSound() {
      echo "Generic animal sound";
   }
}
class Dog extends Animal {
   public function makeSound() {
      echo "Bark!";
   }
}
$dog = new Dog();
$dog->makeSound();  // Output: Bark!
```

In this example, the **Dog** class inherits the **makeSound()** method from the **Animal** class. The **makeSound()** method in the **Dog** class overrides the method in the **Animal** class.

Scope Resolution Operator (::):

The scope resolution operator (**::**) is used to access static properties and methods, as well as to call a parent class's methods when dealing with inheritance. It allows you to reference these elements without creating an instance of the class.

```
class Animal {
   public static $counter = 0;
   public function __construct() {
      self::$counter++;
   }
}
class Dog extends Animal {
   public function __construct() {
      parent::__construct(); // Call the parent class's constructor
   }
}
```

```
$dog = new Dog();

echo Animal::$counter; // Output: 1
```

In this example, the **Dog** class calls the constructor of the **Animal** class using **parent::__construct()**. The scope resolution operator (**::**) is used to access the static property **$counter** of the **Animal** class.

Here are the main uses of the scope resolution operator:

1. Accessing Static Properties:

```
class MyClass {
   public static $myStaticProperty = "Hello, World!";
}
echo MyClass::$myStaticProperty;  // Outputs: Hello, World!
```

2. Accessing Constants:

```
class MathOperations {
   const PI = 3.14159;
}
echo MathOperations::PI;  // Outputs: 3.14159
```

3. Calling Static Methods:

```
class MathOperations {
   public static function add($a, $b) {
      return $a + $b;
   }
}
echo MathOperations::add(5, 3);  // Outputs: 8
```

4. Accessing Parent Class's Elements in Inheritance:

```
class ParentClass {
   public static function myStaticMethod() {
      return "Hello from ParentClass!";
   }
}
```

```
class ChildClass extends ParentClass {

    public static function myStaticMethod() {

        // Calling the parent class's static method using the parent keyword

        return parent::myStaticMethod() . " Hello from ChildClass!";

    }

}

echo ChildClass::myStaticMethod();

// Outputs: Hello from ParentClass! Hello from ChildClass!
```

In the last example, the parent::myStaticMethod() call is used to invoke the static method from the parent class within the context of the child class.

It's important to note that the scope resolution operator is not used to access non-static (instance) properties or methods. For those, you would need to create an instance of the class and use the object operator (->).

4.5 Overwriting Methods

In PHP, when a child class inherits from a parent class, it has the option to override (or overwrite) methods defined in the parent class. This allows the child class to provide its own implementation of a method while still maintaining the same method signature as the parent class. This concept is known as method overriding.

Here's a simple example to demonstrate method overriding in PHP:

```
class ParentClass {

    public function myMethod() {

        return "Hello from ParentClass!";

    }

}

class ChildClass extends ParentClass {

    public function myMethod() {

        // Call the parent class's method using the parent keyword

        return parent::myMethod() . " Hello from ChildClass!";

    }
}

// Create an instance of the child class

$childObject = new ChildClass();

// Call the overridden method

echo $childObject->myMethod();

// Outputs: Hello from ParentClass! Hello from ChildClass!
```

In this example, the ChildClass extends ParentClass and overrides the myMethod method. Inside the overridden method, the parent::myMethod() call is used to invoke the method from the parent class, and the child class then appends its own message.

Key points about method overriding in PHP:

Method Signature: The overriding method in the child class must have the same method signature (name and parameters) as the method in the parent class.

Parent Keyword: If you want to call the overridden method from the parent class within the child class, you can use the parent::methodName() syntax.

Visibility: The visibility (public, protected, private) of the overriding method in the child class must be the same or less restrictive than the visibility of the overridden method in the parent class.

4.6 Database Connectivity

4.6.1. Creating database with server-side script

In PHP, we can use the SQL CREATE DATABASE statement to create a database. Here's an example of how you can create a database using mysqli:

```
<?php

// Database credentials

$hostname = 'your_database_host';

$username = 'your_username';

$password = 'your_password';


// Connect to MySQL server

$mysqli = new mysqli($hostname, $username, $password);


// Check connection

if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

// Create a new database named "your_database_name"
$databaseName = 'your_database_name';
$sql = "CREATE DATABASE $databaseName";

if ($mysqli->query($sql) === TRUE) {
    echo "Database created successfully";
} else {
    echo "Error creating database: " . $mysqli->error;
}
// Close the connection
$mysqli->close();
?>
```

4.6.2. Connecting server-side script to database

```
<?php
// Database credentials
$hostname = 'your_database_host';
$database = 'your_database_name';
$username = 'your_username';
$password = 'your_password';

// Connect to MySQL using mysqli
$mysqli = new mysqli($hostname, $username, $password, $database);

// Check connection
if ($mysqli->connect_error) {
    die("Connection failed: " . $mysqli->connect_error);
}

// Now you can use $mysqli to perform database operations

// For example, you can execute queries, fetch data, etc.

echo "Connected successfully";

?>
```

4.6.3 Multiple Connections

If you need to establish multiple database connections in your PHP script, you can create separate instances of mysqli for each connection. Below, is example for mysqli with two different database connections.

```
<?php

// Database credentials for the first connection

$host1 = 'host1';

$db1 = 'database1';

$user1 = 'user1';

$pass1 = 'password1';

// Database credentials for the second connection

$host2 = 'host2';

$db2 = 'database2';

$user2 = 'user2';

$pass2 = 'password2';

// First connection

$mysqli1 = new mysqli($host1, $user1, $pass1, $db1);

// Check connection for the first connection

if ($mysqli1->connect_error) {

   die("Connection failed: " . $mysqli1->connect_error);

}

// Second connection

$mysqli2 = new mysqli($host2, $user2, $pass2, $db2);

// Check connection for the second connection

if ($mysqli2->connect_error) {
    die("Connection failed: " . $mysqli2->connect_error);
}
// Now you can use $mysqli1 and $mysqli2 for different database operations
// ...
echo "Connected successfully to both databases";
?>
```

4.6.4 Making queries

```
<?php
// Assume $mysqli is the mysqli connection object

// SELECT query
$result = $mysqli->query('SELECT * FROM your_table');
while ($row = $result->fetch_assoc()) {
    print_r($row);
}

// INSERT query
$sql = "INSERT INTO your_table (column1, column2) VALUES (?, ?)";
$stmt = $mysqli->prepare($sql);
$stmt->bind_param('ss', $value1, $value2);
// Set values for parameters
$value1 = 'some_value';
$value2 = 'another_value';
// Execute the prepared statement
$stmt->execute();

// UPDATE query
$sql = "UPDATE your_table SET column1 = ? WHERE column2 = ?";
$stmt = $mysqli->prepare($sql);
```

```
$stmt->bind_param('ss', $new_value, $condition);

// Set values for parameters

$new_value = 'updated_value';

$condition = 'some_condition';

// Execute the prepared statement

$stmt->execute();


// DELETE query

$sql = "DELETE FROM your_table WHERE column = ?";

$stmt = $mysqli->prepare($sql);

$stmt->bind_param('s', $value);

// Set value for parameter

$value = 'value_to_delete';

// Execute the prepared statement

$stmt->execute();

?>
```

4.6.5 Building in Error Checking

Error checking and handling are crucial aspects of database operations to ensure that your application behaves appropriately when issues arise. Mysqli provide mechanisms for handling errors during database queries. Below are examples of how you can incorporate error checking into your PHP code when interacting with a database using mysqli.

```
<?php

// Assume $mysqli is the mysqli connection object

// SELECT query

$result = $mysqli->query('SELECT * FROM your_table');

if (!$result) {

   die("Error: " . $mysqli->error);

}

while ($row = $result->fetch_assoc()) {

   print_r($row);
```

```
}

// INSERT query
$sql = "INSERT INTO your_table (column1, column2) VALUES (?, ?)";
$stmt = $mysqli->prepare($sql);
if (!$stmt) {
    die("Error: " . $mysqli->error);
}
$stmt->bind_param('ss', $value1, $value2);
// Set values for parameters
$value1 = 'some_value';
$value2 = 'another_value';
// Execute the prepared statement
if (!$stmt->execute()) {
    die("Error: " . $stmt->error);
}

// UPDATE query
$sql = "UPDATE your_table SET column1 = ? WHERE column2 = ?";
$stmt = $mysqli->prepare($sql);
if (!$stmt) {
    die("Error: " . $mysqli->error);
}
$stmt->bind_param('ss', $new_value, $condition);
// Set values for parameters
$new_value = 'updated_value';
$condition = 'some_condition';
// Execute the prepared statement
if (!$stmt->execute()) {
    die("Error: " . $stmt->error);
}

// DELETE query
$sql = "DELETE FROM your_table WHERE column = ?";
$stmt = $mysqli->prepare($sql);
if (!$stmt) {
    die("Error: " . $mysqli->error);
}
$stmt->bind_param('s', $value);
// Set value for parameter
$value = 'value_to_delete';
// Execute the prepared statement
if (!$stmt->execute()) {
    die("Error: " . $stmt->error);
}
?>
```

These examples demonstrate how to catch and handle errors using conditional statements for mysqli. It's important to adapt error handling to your application's specific needs and log or display error messages appropriately for debugging and user feedback.

4.6.7 Displaying Queries in tables

To display query results in tables in PHP, you can use HTML along with the PHP code to generate the table structure and populate it with data fetched from the database. Here's an example using mysqli to fetch data from a database and display it in an HTML table:

```
<?php
// Assume $mysqli is the mysqli connection object
// SELECT query
$result = $mysqli->query('SELECT * FROM your_table');
if (!$result) {
    die("Error: " . $mysqli->error);
}
// Check if there are rows in the result
if ($result->num_rows > 0) {
    echo '<table border="1">';
    echo '<tr><th>ID</th><th>Column1</th><th>Column2</th></tr>';
    // Output data into table rows
    while ($row = $result->fetch_assoc()) {
       echo '<tr>';
       echo '<td>' . $row['id'] . '</td>';
       echo '<td>' . $row['column1'] . '</td>';
       echo '<td>' . $row['column2'] . '</td>';
       echo '</tr>';
    }
    echo '</table>';
} else {
    echo 'No rows found.';
}
?>
```

Notes: To fetch data as an associative array, the fetch_assoc() method is used with mysqli.

4.6.8 Building forms and control form data using queries

Building forms and controlling form data using queries typically involves creating an HTML form to collect user input and then using PHP to process the form data, perform database queries, and handle any necessary actions. Below is an example that demonstrates how to create a simple form, validate user input, and insert data into a database using mysqli:

HTML form:

```
<!DOCTYPE html>

<html>

<head>

   <title>Sample Form</title>

</head>

<body>

   <h2>Contact Form</h2>

   <form action="process_form.php" method="post">

      <label for="name">Name:</label>

      <input type="text" name="name" required><br>

      <label for="email">Email:</label>

      <input type="email" name="email" required><br>

      <label for="message">Message:</label>

      <textarea name="message" rows="4" required></textarea><br>

      <input type="submit" value="Submit">

   </form>

</body>

</html>
```

PHP Script (process_form.php):

```
<?php
// Assume $mysqli is the mysqli connection object
if ($_SERVER["REQUEST_METHOD"] == "POST") {
  // Validate and sanitize user input
  $name = htmlspecialchars($_POST["name"]);
  $email = filter_var($_POST["email"], FILTER_SANITIZE_EMAIL);
  $message = htmlspecialchars($_POST["message"]);

  // Insert data into the database
  $sql = "INSERT INTO messages (name, email, message) VALUES (?, ?, ?)";
  $stmt = $mysqli->prepare($sql);

  // Check if the statement was prepared successfully
  if ($stmt) {
    $stmt->bind_param('sss', $name, $email, $message);

    // Execute the prepared statement
    if ($stmt->execute()) {
      echo "Data inserted successfully!";
    } else {
      echo "Error executing statement: " . $stmt->error;
    }

    // Close the statement
    $stmt->close();
  } else {
    echo "Error preparing statement: " . $mysqli->error;
  }
} else {
    echo "Invalid request method.";
}
?>
```

This example demonstrates a simple contact form with fields for name, email, and a message. The PHP script (process_form.php) processes the form data when the form is submitted. It validates and sanitizes the user input and then inserts the data into a database table named messages using a prepared statement with mysqli.

Make sure to create the messages table in your database with appropriate columns (name, email, message) before using this script.