2.1 File Handling and Directories

2.1.1 Including files using include and require

Including files using include and require

include()

- One of the most useful tools is to insert another php script from a file into the current php script.
- The command include("filename"); will import contents of a text file called filename and insert it at the include spot.
- The included text may be composed of XHTML, PHP or both.
- The include() function is mostly used when the file is not required and the application should continue to execute its process when the file is not found.
- The include() function will only produce a warning (E_WARNING) and the script will continue to execute.

Example:-

File 1: menu.php

```
<a href=”default.php”>HOME</a>

<a href=”contact.php”>Contact</a>

<a href=”staff.php”>Staff</a>
```

File 2 :Student.php

```
        <html>

        <body>

        <?php

        Include(“menu.php”);

        // if menu.php is not found then also remaining echo statement is script will executed

        ?>

        </body>

        </html>
```

require()

- Syntax and uses is as same as include() but the difference is that, if the file is not found the remaining script is also not executed.
- The require() function is mostly used when the file is mandatory for the application.
- The require() will produce a fatal error (E_COMPILE_ERROR) along with the warning and the script will stop its execution.

2.1.2 File operations: fopen(), fread(), fwrite(), fclose()

PHP File Handling

In PHP, File handling is the process of interacting with files on the server, such as reading files, writing to a file, creating new files, or deleting existing ones. File handling is essential for applications that require the storage and retrieval of data, such as logging systems, user-generated content, or file uploads.

Types of File Operations in PHP

Several types of file operations can be performed in PHP:

- **Reading Files:** PHP allows you to read data from files either entirely or line by line.
- **Writing to Files**: You can write data to a file, either overwriting existing content or appending to the end.
- **File Metadata**: PHP allows you to gather information about files, such as their size, type, and last modified time.
- **File Uploading**: PHP can handle file uploads via forms, enabling users to submit files to the server.

Common File Handling Functions in PHP

- **fopen()** - Opens a file
- **fclose()** - Closes a file
- **fread()** - Reads data from a file
- **fwrite()** - Writes data to a file
- **file_exists()** - Checks if a file exists
- **unlink()** - Deletes a file

Opening and Closing Files

- Before you can read or write to a file, you need to open it using the fopen() function, which returns a file pointer resource. Once you're done working with the file, you should close it using fclose() to free up resources.

Examples:
```
<?php

// Open the file in read mode
$file = fopen("gfg.txt", "r");

if ($file) {
    echo "File opened successfully!";
    fclose($file); // Close the file
} else {
    echo "Failed to open the file.";
}
```

```
?>
```

File Modes in PHP

Files can be opened in any of the following modes:

- **"w"** – Opens a file for writing only. If the file does not exist, then a new file is created, and if the file already exists, then the file will be truncated (the contents of the file are erased).
- **"r"** – File is open for reading only.
- **"a"** – File is open for writing only. The file pointer points to the end of the file. Existing data in the file is preserved.
- **"w+"** – Opens file for reading and writing both. If the file does not exist, then a new file is created, and if the file already exists, then the contents of the file are erased.
- **"r+"** – File is open for reading and writing both.
- **"a+"** – File is open for write/read. The file pointer points to the end of the file. Existing data in the file is preserved. If the file is not there, then a new file is created.
- **"x"** – New file is created for write only.

1. Reading the Entire File

You can read the entire content of a file using the fread() function or the file_get_contents() function.

```
<?php


$file = fopen("gfg.txt", "r");

$content = fread($file, filesize("gfg.txt"));


echo $content;

fclose($file);


?>
```

2. Reading a File Line by Line

You can use the fgets() function to read a file line by line.

```
<?php


$file = fopen("gfg.txt", "r");


if ($file) {
    while (($line = fgets($file)) !== false) {
        echo $line . "<br>";
    }
    fclose($file);
}


?>
```

3. Writing to Files

You can write to files using the fwrite() function. It writes data to an open file in the specified mode.

```
<?php


// Open the file in write mode
$file = fopen("gfg.txt", 'w');


if ($file) {
    $text = "Hello world\n";
    fwrite($file, $text);
            fclose($file);
}


?>
```

4. Deleting Files

Use the unlink() function to delete the file in PHP.

```
<?php


if (file_exists("gfg.txt")) {

   unlink("gfg.txt");

   echo "File deleted successfully!";

} else {

   echo "File does not exist.";

}


?>
```

2.1.3 File upload using $_FILES and move_uploaded_file()

What is $_FILES?

- $_FILES is a PHP superglobal that holds information about uploaded files via an HTML form.
- It’s an associative array of items sent via HTTP POST method with enctype="multipart/form-data".

Syntax of $_FILES:

```
$_FILES['input_name']['name']              // Original file name

$_FILES['input_name']['type']              // MIME type of the file

$_FILES['input_name']['tmp_name']          // Temporary location on the server

$_FILES['input_name']['error']             // Error code

$_FILES['input_name']['size']              // Size of uploaded file in bytes
```

move_uploaded_file()

- This function **moves the uploaded file** from the temporary location to a permanent location on the server.

Syntax:

```
move_uploaded_file(file, dest)
```

Parameter Values

Parameter	Description
file	Required. Specifies the filename of the uploaded file
dest	Required. Specifies the new location for the file

Example:

```
$_FILES['myfile']['tmp_name']
```

Example: Basic File Upload

1. HTML Form (upload.html)

```
<!DOCTYPE html>
<html>
<head>
   <title>Upload File</title>
</head>
<body>
   <h2>Upload a File</h2>
   <form action="upload.php" method="POST" enctype="multipart/form-data">
      <label>Select file:</label>
      <input type="file" name="myfile">
      <br><br>
      <input type="submit" name="submit" value="Upload">
   </form>
</body>
</html>
```

2. PHP Script (upload.php)

```
<?php
if (isset($_POST['submit'])) {
   $uploadDir = "uploads/";
```

```
    $uploadFile = $uploadDir . basename($_FILES["myfile"]["name"]);


    // Check if file was uploaded without errors
    if ($_FILES["myfile"]["error"] === 0) {
        if (move_uploaded_file($_FILES["myfile"]["tmp_name"], $uploadFile)) {
            echo " File uploaded successfully: " . htmlspecialchars($_FILES["myfile"]["name"]);
        } else {
            echo " Failed to move uploaded file.";
        }
    } else {
        echo "Error uploading file. Error code: " . $_FILES["myfile"]["error"];
    }
}
?>
```

📂 Directory Structure

```
your_project/
├── upload.html
├── upload.php
└── uploads/        ← Make sure this folder exists and is writable (chmod 755 or 777)
```

2.1.4 File download using PHP headers

Key points

- Never echo anything **before** headers.
- Validate/whitelist the requested file (avoid .. traversal).
- Send correct headers: Content-Type, Content-Length, Content-Disposition.
- Use readfile() (simple) or stream in chunks (big files).

2.1.5 Directory operations: opendir(), readdir(), mkdir(), rmdir()

- **opendir()** – Opens a directory handle to read its contents.
- **readdir()** – Reads the next file or folder name from an opened directory handle.

- **mkdir()** – Creates a new directory with given permissions.
- **rmdir()** – Removes an empty directory from the file system.

1. **opendir() + readdir()** – Read directory contents

```
<?php
$dir = "uploads"; // folder to read

if ($handle = opendir($dir)) {
    echo "Files in $dir:<br>";
    while (($file = readdir($handle)) !== false) {
        echo $file . "<br>"; // will include . and ..
    }
    closedir($handle);
}
?>
```

2. **mkdir()** – Create a new directory

```
<?php
$folder = "new_folder";

if (!is_dir($folder)) {
    mkdir($folder);
    echo "Folder created: $folder";
} else {
    echo "Folder already exists.";
}
?>
```

3. **rmdir()** – Remove an empty directory

```
<?php
$folder = "old_folder";
```

```
if (is_dir($folder)) {
    rmdir($folder); // only works if folder is empty
    echo "Folder deleted: $folder";
} else {
    echo "Folder not found.";
}
?>
```

4. **Combine** – opendir() + mkdir() + rmdir()

```
<?php
$dir = "test_dir";

// Create directory
if (!is_dir($dir)) {
    mkdir($dir);
    echo "Created $dir<br>";
}

// Read directory
if ($handle = opendir($dir)) {
    echo "Contents of $dir:<br>";
    while (($file = readdir($handle)) !== false) {
        echo $file . "<br>";
    }
    closedir($handle);
}

// Remove directory
if (is_dir($dir)) {
    rmdir($dir);
    echo "Deleted $dir";
}
?>
```

2.2 Forms, Filters, and JSON

2.2.1 Designing and handling HTML forms

HTML forms allow users to send data to the server, and PHP can handle this data using $_GET or $_POST superglobals.

Example – HTML + PHP Form Handling

```
<!-- form.html -->
<form action="process.php" method="post">
   Name: <input type="text" name="username"><br>
   Email: <input type="email" name="email"><br>
   <input type="submit" value="Submit">
</form>
<!—php code-->
<?php
// process.php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
   $name  = $_POST['username'];
   $email = $_POST['email'];

   echo "Hello, $name! Your email is $email.";
}
?>
```

2.2.2 Server-side validation techniques

Server-side validation ensures that user input is checked on the server before processing, protecting against invalid data and security threats.

Example – Simple Validation

```
<?php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $name  = trim($_POST['username']);
    $email = trim($_POST['email']);
    $errors = [];

    if (empty($name)) {
        $errors[] = "Name is required.";
    }
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        $errors[] = "Invalid email format.";
    }

    if ($errors) {
        foreach ($errors as $err) {
            echo "<p style='color:red;'>$err</p>";
        }
    } else {
        echo "Form submitted successfully!";
    }
}
?>
```

2.2.3 PHP filters: filter_var() and constants

What are Filters?

- Filters are used in PHP to validate (check) and sanitize (clean) user input.
- filter_var() is the main function used for this.

Syntax:

```
filter_var(value, filter_type);
```

Common Filter Constants:

Constant	Purpose
FILTER_VALIDATE_EMAIL	Checks if the value is a valid email.
FILTER_VALIDATE_INT	Checks if the value is a valid integer.
FILTER_SANITIZE_STRING	Removes unwanted HTML and special characters.
FILTER_VALIDATE_URL	Checks if the value is a valid URL.

Example – Validate Email

```
<?php
$email = "test@example.com";


if (filter_var($email, FILTER_VALIDATE_EMAIL)) {
   echo "Valid email!";
} else {
   echo "Invalid email!";
}
?>
```

Example – Sanitize String

```
<?php
$text = "<h1>Hello!</h1>";
$clean = filter_var($text, FILTER_SANITIZE_STRING);
echo $clean; // Output: Hello!
```

```
?>
```

2.2.4 Parsing and generating JSON with json_encode() and json_decode()

What is JSON?

- JSON (JavaScript Object Notation) is a format to store and share data.
- PHP can generate JSON from arrays/objects and parse JSON back to PHP.

1. Generating JSON – json_encode()

```
<?php
$data = [
   "name" => "Rahul",
   "age"  => 20,
   "city" => "Surat"
];

$json = json_encode($data);
echo $json;
// Output: {"name":"Rahul","age":20,"city":"Surat"}
?>
```

2. Parsing JSON – json_decode()

```
<?php
$json = '{"name":"Rahul","age":20,"city":"Surat"}';

// Decode to PHP object
$obj = json_decode($json);
echo $obj->name; // Rahul

// Decode to PHP associative array
$arr = json_decode($json, true);
```

```
echo $arr["city"]; // Surat
?>
```

2.3 Cookies, Sessions, and Emails

2.3.1 Creating and accessing cookies using setcookie() and $_COOKIE

What is Cookies?

- A cookie is a small piece of data stored in the browser by the server.
- setcookie() is used to create or update a cookie.
- $_COOKIE is used to read cookie values.
- Cookies are sent back to the server with every request until they expire or are deleted.

Syntax:

```
setcookie(name, value, expire, path, domain, secure, httponly);
```

Example 1 – Create a Cookie

```
<?php
// Create cookie "username" valid for 1 hour
setcookie("username", "Rahul", time() + 3600, "/");


echo "Cookie 'username' has been set!";
// Note: Cookie will be available on next page load
?>
```

Example 2 – Read a Cookie

```
<?php
if (isset($_COOKIE["username"])) {
   echo "Welcome back, " . $_COOKIE["username"];
} else {
   echo "Hello, new visitor!";
}
?>
```

Example 3 – Delete a Cookie

```
<?php
// Set cookie expiry time in the past to delete it
setcookie("username", "", time() - 3600, "/");
echo "Cookie deleted.";
?>
```

- **Cookies are** stored on the client (browser).
- Always set cookies before any HTML output.
- Use time() to set expiry (e.g., time() + 86400 = 1 day).

2.3.2 Session management with session_start() and $_SESSION

What is Session?

- A session stores data on the server for each user.
- Unlike cookies (stored in browser), session data is not visible to the user.
- session_start() must be called before any HTML output to start or resume a session.
- $_SESSION is a special array that holds session variables.

Example 1 – Start a Session & Store Data

```
<?php
session_start(); // Start or resume a session

$_SESSION["username"] = "Rahul";
$_SESSION["role"] = "Student";

echo "Session variables are set!";
?>
```

Example 2 – Access Session Data

```
<?php
session_start(); // Resume session

if (isset($_SESSION["username"])) {
   echo "Welcome, " . $_SESSION["username"];
   echo "<br>Your role is: " . $_SESSION["role"];
} else {
   echo "No session data found.";
}
?>
```

Example 3 – Remove Session Data

```
<?php
session_start();

// Remove one variable
unset($_SESSION["username"]);

// Remove all session variables
session_unset();

// Destroy session completely
session_destroy();

echo "Session ended.";
?>
```

- Sessions use a session ID stored in a cookie (PHPSESSID) to track the user.
- Always call session_start() before using $_SESSION.
- Use session_destroy() to end a session completely.

2.3.3 Sending Emails using mail() in PHP

- The mail() function is used to send emails directly from PHP.
- Works only if the server has mail service enabled (e.g., Sendmail, Postfix, SMTP).

syntax:

```
mail(to, subject, message, headers);
```

Example 1 – Simple Email

```
<?php
$to = "student@example.com";
$subject = "Welcome to PHP Class";
$message = "Hello Student,\n\nThis is a test email from PHP.";
$headers = "From: teacher@example.com";

if (mail($to, $subject, $message, $headers)) {
    echo " Email sent successfully!";
} else {
    echo " Email sending failed!";
}
?>
```

Example 2 – Email with HTML Content

```
<?php
$to = "student@example.com";
$subject = "PHP HTML Email";
$message = "
<html>
<head><title>PHP Email</title></head>
<body>
<h2>Welcome Student!</h2>
<p>This is an <b>HTML email</b> from your teacher.</p>
</body>
</html>
```

```
";


// To send HTML mail, set content-type header
$headers  = "MIME-Version: 1.0" . "\r\n";
$headers .= "Content-type:text/html;charset=UTF-8" . "\r\n";
$headers .= "From: teacher@example.com";


if (mail($to, $subject, $message, $headers)) {
   echo " HTML Email sent!";
} else {
   echo " Failed to send HTML email!";
}
?>
```

Example 3 – Multiple Recipients

```
<?php
$to = "student1@example.com, student2@example.com";
$subject = "Class Reminder";
$message = "This is a reminder for tomorrow's PHP session.";
$headers = "From: teacher@example.com";


mail($to, $subject, $message, $headers);
?>
```

- Use a real email server or tools **like** XAMPP with Mercury Mail, SMTP services, or PHPMailer for testing.
- Always set proper From: header to avoid spam filters.
- For sending bulk or secure emails, PHPMailer is better than mail().

2.3.4 Email formatting: headers, subject, attachments

In PHP, emails are sent using the mail() function:

```
mail(to, subject, message, headers);
```

- to → Receiver's email.
- subject → Email subject.
- message → Body of the email.
- headers → Additional information (From, CC, BCC, MIME type, etc.).

1. Sending a Simple Email

```
<?php
$to = "student@example.com";
$subject = "Welcome to PHP Class";
$message = "Hello Student,\nThis is a simple test email from PHP.";
$headers = "From: teacher@example.com";

if (mail($to, $subject, $message, $headers)) {
    echo "Email sent successfully!";
} else {
    echo "Email sending failed.";
}
?>
```

2. Adding Headers (From, CC, BCC)

```
<?php
$to = "student@example.com";
$subject = "Class Notice";
$message = "Dear Student,\nTomorrow we have a PHP practical exam.";

$headers = "From: teacher@example.com\r\n";
$headers .= "CC: hod@example.com\r\n";
$headers .= "BCC: admin@example.com\r\n";

mail($to, $subject, $message, $headers);
```

```
?>
```

3. Sending HTML Email (Formatting inside message)

```
<?php
$to = "student@example.com";
$subject = "HTML Email Example";

$message = "
<html>
<head><title>Email Test</title></head>
<body>
<h2 style='color:blue;'>Welcome to PHP Class</h2>
<p>This email is <b>HTML formatted</b>.</p>
</body>
</html>
";

// Always set content-type when sending HTML email
$headers  = "MIME-Version: 1.0\r\n";
$headers .= "Content-type:text/html;charset=UTF-8\r\n";
$headers .= "From: teacher@example.com";

mail($to, $subject, $message, $headers);
?>
```

4. Sending Email with Attachment

Sending attachments requires MIME (Multipurpose Internet Mail Extensions) format.

```
<?php
$to = "student@example.com";
$subject = "Assignment File Attached";
$message = "Hello, please find the attached file.";
```

```
$file = "assignment.pdf"; // file must exist on server
$content = file_get_contents($file);
$content = chunk_split(base64_encode($content));

$separator = md5(time());
$eol = "\r\n";

// Headers
$headers  = "From: teacher@example.com\r\n";
$headers .= "MIME-Version: 1.0\r\n";
$headers .= "Content-Type: multipart/mixed; boundary=\"".$separator."\"\r\n";

// Body
$body  = "--$separator$eol";
$body .= "Content-Type: text/plain; charset=\"UTF-8\"$eol";
$body .= "Content-Transfer-Encoding: 7bit$eol$eol";
$body .= $message . "$eol";

// Attachment
$body .= "--$separator$eol";
$body .= "Content-Type: application/octet-stream; name=\"$file\"$eol";
$body .= "Content-Transfer-Encoding: base64$eol";
$body .= "Content-Disposition: attachment; filename=\"$file\"$eol$eol";
$body .= $content . "$eol";
$body .= "--$separator--";

// Send email
mail($to, $subject, $body, $headers);
?>
```

Summary

- **Headers** → Add extra info like From, CC, BCC, MIME type.
- **Subject** → Title of the email.
- **Attachments** → Use MIME type encoding and base64.
- **HTML Email** → Format emails with colors, bold, headings.

2.4 OOP and Exception Handling in PHP

2.4.1 Creating classes and objects

- In PHP, Object-Oriented Programming (OOP) makes it easier to organize and reuse code. The two fundamental building blocks in OOP are classes and objects.
- An object is an instance of a class created using the new keyword.

PHP Classes

- A class in PHP is a blueprint for creating objects. It defines the properties (variables) and methods (functions) that the objects created from the class will have.
- By using classes, we can group related data and actions, making it easier to organize and manage our code.

In PHP a class is defined using the class keyword, followed by the class name and curly braces.

Syntax:

```
<?php
class Cars {
    // PHP code goes here...
}
?>
```

PHP Objects

- An object is an instance of a class. When you create an object from the class, memory is allocated, and the object can store data and perform actions defined in the class.
- To create an object, we use the new keyword.

Syntax:

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

Examples:

```
<?php
// Defining a class
class Car {
    // Properties (variables inside a class)
    public $brand;
    public $color;

    // Method (function inside a class)
    public function startEngine() {
        return "The engine has started!";
    }
}

// Creating an object of Car class
$myCar = new Car();

// Accessing properties
$myCar->brand = "Tesla";
$myCar->color = "Red";

// Accessing methods
echo $myCar->brand . " is " . $myCar->color . "<br>";
echo $myCar->startEngine();
?>
```

Output:

```
Tesla is Red
The engine has started!
```

2.4.2 Using constructors and property visibility

1. Constructors in PHP

A constructor is a special function in a class with the name __construct().
It is automatically called when a new object is created.

- It is used to initialize properties of the class.
- Saves us from setting values manually after object creation.

Example: Using Constructor

```
<?php
class Student {
    public $name;
    public $rollNo;

    // Constructor
    public function __construct($name, $rollNo) {
        $this->name = $name;
        $this->rollNo = $rollNo;
    }

    // Method
    public function displayInfo() {
        return "Name: " . $this->name . ", Roll No: " . $this->rollNo;
    }
}

// Objects with constructor
$student1 = new Student("Rahul", 101);
$student2 = new Student("Priya", 102);

echo $student1->displayInfo() . "<br>";
echo $student2->displayInfo();
?>
```

Output:

```
Name: Rahul, Roll No: 101
Name: Priya, Roll No: 102
```

2. Property Visibility in PHP

In PHP, visibility controls how class properties and methods can be accessed.

- public → accessible everywhere (inside/outside class).
- private → accessible only inside the class.
- protected → accessible inside the class and child classes (inheritance).

Example: Property Visibility

```
<?php
class BankAccount {
  public $accountHolder;    // Public property
  private $balance;         // Private property

  // Constructor
  public function __construct($holder, $amount) {
    $this->accountHolder = $holder;
    $this->balance = $amount;
  }

  // Public method to check balance
  public function getBalance() {
    return "Balance: $" . $this->balance;
  }

  // Public method to deposit money
  public function deposit($amount) {
    $this->balance += $amount;
    return "Deposited: $" . $amount . " | New " . $this->getBalance();
  }
}


// Creating object
$account = new BankAccount("Amit", 5000);


echo $account->accountHolder . "<br>";   //  Allowed (public)
// echo $account->balance;  ERROR (private)


echo $account->getBalance() . "<br>";
echo $account->deposit(2000);
?>
```

Output:

```
Amit
Balance: $5000
Deposited: $2000 | New Balance: $7000
```

2.4.3 Inheritance and method overriding

1. Inheritance in PHP

- Inheritance allows a class (child class) to use the properties and methods of another class (parent class).
- Use the keyword extends to inherit.
- It supports code reusability.

Example: Inheritance

```
<?php
// Parent class
class Person {
   public $name;


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

```
    }

    public function showName() {
        return "Name: " . $this->name;
    }
}

// Child class (inherits Person)
class Student extends Person {
    public $rollNo;

    public function __construct($name, $rollNo) {
        // Call parent constructor
        parent::__construct($name);
        $this->rollNo = $rollNo;
    }

    public function showDetails() {
        return $this->showName() . ", Roll No: " . $this->rollNo;
    }
}

// Create object of child class
$student1 = new Student("Rahul", 101);

echo $student1->showDetails();
?>
```

Output:

```
Name: Rahul, Roll No: 101
```

Types of Inheritance in PHP

1. Inheritance → Supported

- One child class inherits from one parent class.

Example

```
class ParentClass { }

class ChildClass extends ParentClass { }
```

2. Multilevel Inheritance → Supported

A class inherits from another child class (grandparent → parent → child).

Example

```
class A { }

class B extends A { }

class C extends B { }
```

3. Hierarchical Inheritance → Supported

Multiple child classes inherit from the same parent class.

```
class ParentClass { }

class Child1 extends ParentClass { }

class Child2 extends ParentClass { }
```

Summary

- **Supported:** Single, Multilevel, Hierarchical inheritance.
- **Not Supported:** Multiple inheritance, Hybrid inheritance.
- **Solution for Multiple →** Use Interfaces or Traits.

2. Method Overriding in PHP

- Method overriding happens when a child class redefines (overrides) a method from the parent class.
- The child's method replaces the parent's method when called.
- Use parent::methodName() if you still want to call parent method.

Example: Method Overriding

```
<?php
// Parent class
class Teacher {
    public function teach() {
        return "Teaching in a general way.";
    }
}

// Child class overrides the method
class MathTeacher extends Teacher {
    public function teach() {
        return "Teaching Mathematics with formulas and problem-solving.";
    }
}

// Another child class
class ScienceTeacher extends Teacher {
    public function teach() {
        return "Teaching Science with experiments and observations.";
    }
}

// Objects
$t1 = new MathTeacher();
$t2 = new ScienceTeacher();
```

```
echo $t1->teach() . "<br>";

echo $t2->teach();

?>
```

Output:

```
Teaching Mathematics with formulas and problem-solving.

Teaching Science with experiments and observations.
```

PHP - The final Keyword

- The **final** keyword can be used to prevent class inheritance or to prevent method overriding.
- The following example shows how to prevent class inheritance:

```
<?php
final class Fruit {
  // some code
}

// will result in error
class Strawberry extends Fruit {
  // some code
}
?>
```

The following example shows how to prevent method overriding:

```
<?php
class Fruit {
  final public function intro() {
    // some code
  }
}

class Strawberry extends Fruit {
  // will result in error
  public function intro() {
    // some code
  }
}
?>
```

2.4.4 Exception handling: try, catch, finally, throw

What is Exception Handling?

- Exceptions are errors that can be caught and handled without stopping the script.
- PHP provides try, catch, finally, and throw for exception handling.

Types of Exception Handling

- **try** - A function using an exception should be in a "try" block. If the exception does not trigger, the code will continue as normal. However if the exception triggers, an exception is "thrown".
- **throw** - This is how you trigger an exception. Each "throw" must have at least one "catch".
- **catch** - A "catch" block retrieves an exception and creates an object containing the exception information.
- **finally** - Always executes (cleanup code), no matter what happens.

1. try and catch

- Code inside try { } is executed.
- If an exception occurs, control goes to catch { }.

Example

```
<?php
try {
   // risky code
   $num = 10 / 0; // Division by zero (error)
   echo $num;
} catch (Exception $e) {
   // handling error
   echo "Error: " . $e->getMessage();
}
?>
```

OUTPUT:

But in PHP, 10/0 raises a warning (not an exception).
So, usually, we throw exceptions manually.

2. throw

- Used to **manually throw an exception**.
- Must be caught by catch.

Example

```
<?php
function divide($a, $b) {
   if ($b == 0) {
      throw new Exception("Division by zero is not allowed.");
   }
   return $a / $b;
}

try {
   echo divide(10, 0);
} catch (Exception $e) {
   echo "Caught Exception: " . $e->getMessage();
}
?>
OUTPUT:
Caught Exception: Division by zero is not allowed.
```

3. finally

- The finally { } block **always executes**, whether an exception occurs or not.
- Useful for cleanup tasks (closing DB connection, files, etc.).

Example

```
<?php
function divide($a, $b) {
   if ($b == 0) {
      throw new Exception("Division by zero not allowed.");
   }
   return $a / $b;
}

try {
   echo divide(10, 2);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    echo "<br>Execution completed!";
}
?>
OUTPUT:
5
Execution completed!
```

4. Multiple catch Blocks

- You can catch **different types of exceptions** separately.

```
<?php
try {
    throw new InvalidArgumentException("Invalid argument!");
} catch (InvalidArgumentException $e) {
    echo "Caught InvalidArgumentException: " . $e->getMessage();
} catch (Exception $e) {
    echo "Caught General Exception: " . $e->getMessage();
}
?>
```

Summary

- **try** → Code that may throw exception.
- **catch** → Handles the exception.
- **throw** → Used to raise an exception.
- **finally** → Always executes (cleanup code).

2.4.5 Input Validation using Regular Expressions in PHP

What is Input Validation?

- Input validation means checking user inputs (like email, phone number, password) to make sure they follow the correct format.
- In PHP, we use Regular Expressions (regex) with preg_match() function for validation.

preg_match() Function

Syntax:

```
preg_match(pattern, string)
```

Returns 1 if pattern matches, 0 if not.

Common Validation Examples

1. Validate Email

```
<?php
$email = "student@example.com";

if (preg_match("/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-z]{2,}$/", $email)) {
   echo "Valid Email";
} else {
   echo "Invalid Email";
}
?>
```

Output:

```
Valid Email
```

2. Validate Phone Number (10 digits)

```
<?php
$phone = "9876543210";

if (preg_match("/^[0-9]{10}$/", $phone)) {
   echo "Valid Phone Number";
} else {
   echo "Invalid Phone Number";
}
?>
```

Output:

```
Valid Phone Number
```

3. Validate Username (only letters & numbers, 5–15 chars)

```php
<?php
$username = "student123";

if (preg_match("/^[a-zA-Z0-9]{5,15}$/", $username)) {
    echo "Valid Username";
} else {
    echo "Invalid Username";
}
?>
```

Output:

```
Valid Username
```

4. Validate Strong Password

At least: 1 uppercase, 1 lowercase, 1 digit, 1 special character, min 8 chars

```php
<?php
$password = "Abcd@1234";

if (preg_match("/^(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9])(?=.*[\W]).{8,}$/", $password)) {
    echo "Strong Password";
} else {
    echo "Weak Password";
}
?>
```

Output:

Strong Password