MEDICAPS UNIVERSITY

Exception Handling

Laxmikant Soni

Learning Objectives

- By the end of this lecture, students will:
  - Understand the need for exception handling.
  - Use `try`, `throw`, and `catch` blocks effectively.
  - Handle multiple exceptions with multiple `catch` blocks.

What is an Error?

- **Definition**: An error is a serious issue that occurs due to problems in the code or environment, often unrecoverable.
- **Analogy**: Like a *"car engine failure"*—the system cannot continue without fixing it.
- **Examples**:
  - Parse error (syntax error)
  - Fatal error (calling an undefined function)
  - Memory exhaustion

What is an Exception?

- **Definition**: An exception is an unexpected event that disrupts normal program flow, but it can be handled using `try...catch`.

- **Analogy**: Like a *"fire alarm"*—it interrupts, but you can take action and continue.

- **Examples**:
  - Division by zero
  - File not found
  - Database connection failure

Key Differences

- **Errors**:
  - Critical issues, usually cannot be recovered at runtime.
  - Indicate problems in code/environment.
- **Exceptions**:
  - Abnormal conditions that can be *caught* and handled.
  - Allow program to continue after corrective action.

Why use Exception Handling

- **Prevents abrupt termination**
  Ensures that scripts do not crash unexpectedly and allows graceful recovery from errors.
- **Separates error-handling from business logic**
  Keeps the core functionality clean and readable by isolating error management code.
- **Improves maintainability and debugging**
  Makes it easier to identify, fix, and manage errors throughout the codebase.

Basic Syntax of Exception Handling in PHP

- **try** block:
  Wraps the code that may cause an exception.
- **throw** statement:
  Used to trigger an exception manually.
- **catch** block:
  Catches and handles the exception thrown.

```php
try {
    // Code that may throw an exception
    throw new Exception("Something went wrong!");
} catch (Exception $e) {
    echo 'Caught exception: ', $e->getMessage();
}
```


The `try` Block

- Contains code that may throw exceptions
- Keeps risky code isolated
- **Best practice**: Keep `try` blocks small and focused

The `throw` Statement

- Used to trigger exceptions manually
- Can be used with built-in or custom exceptions

```
// PHP example (as in your original code)
if ($age < 18) {
  throw new Exception("Underage signup not
allowed.");
}
```

The `catch` Block

- Catches exceptions thrown in `try`
- Provides code to handle the problem
- Can have multiple `catch` blocks for different exceptions

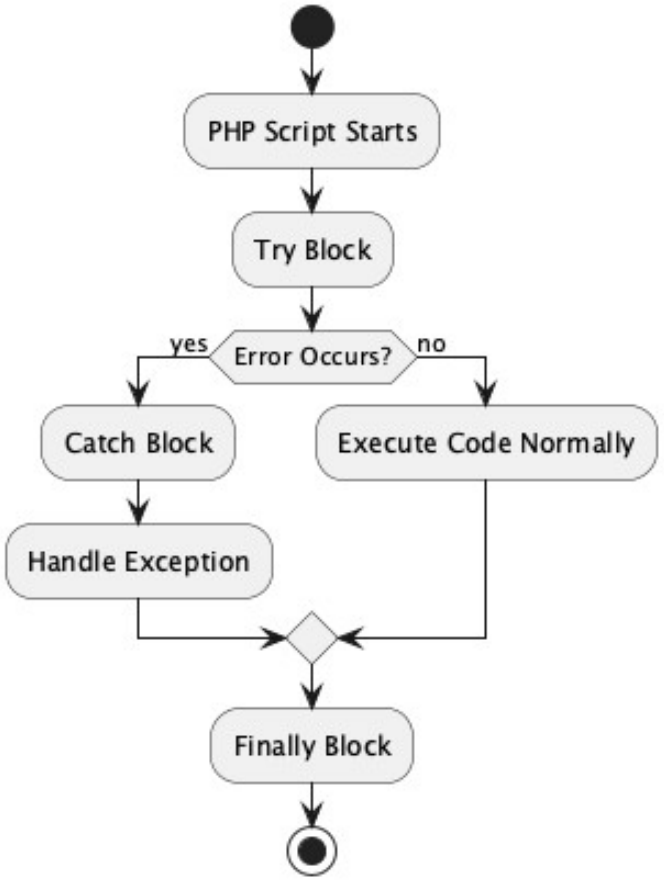
```
try {
    $result = 10 / 0;
} catch (DivisionByZeroError $e) {
    echo "Error: " . $e->getMessage();
}
```

The `finally` Block

- Runs regardless of whether exception occurred.
- Often used for cleanup operations (closing files, DB connections).

```php
try {
    // Code
} catch (Exception $e) {
    echo $e->getMessage();
} finally {
    echo "Cleanup done!";
}
```

Exception Handling Flow in PHP



Multiple `catch` Blocks in PHP

- PHP supports **multiple `catch` blocks** to handle different types of exceptions.
- Each `catch` block can target a specific exception class.

```
try {
    // Some code that might throw different types
of exceptions
    throw new TypeError("Invalid type!");
} catch (TypeError $e) {
    echo "Caught a TypeError: " . $e-
>getMessage();
} catch (Exception $e) {
    echo "Caught a general Exception: " . $e-
>getMessage();
}
```

Before PHP 7.1

- Only one type of exception could be caught in each `catch` block.

- Developers had to **duplicate code** if multiple exceptions required the same handling.

```
try {
    // Code that may throw an Exception or
ArithmeticError.
} catch (ArithmeticError $e) {
    // handle error
} catch (Exception $e) {
    // handle error
}
```

Why Use Multiple `catch` Blocks?

- Improves **readability** and **code structure**
- Allows **different actions** for different error types
- Makes debugging easier by **isolating error causes**

Nested try catch Blocks

- One try block inside another.
- Useful for handling different levels of exceptions.

```php
try {
    try {
        throw new Exception("Inner error");
    } catch (Exception $e) {
        echo "Caught inner: " . $e->getMessage();
        throw $e;
    }
} catch (Exception $e) {
    echo "Caught outer: " . $e->getMessage();
}
```

PHP 7.1 Multiple Exception Handling

- **Definition**: PHP 7.1 introduced the ability to catch multiple exceptions in a single `catch` block using the pipe (`|`) operator.

- **Analogy**: Like having a single *"safety net"* that can catch different kinds of thrown objects.

- **Example**: Handling both `ArithmeticError` and `Exception` without duplicating catch blocks.

```php
try {
    // Code that may throw an Exception or ArithmeticError.
} catch (ArithmeticError | Exception $e) {
    // pass
}
```

Exception Chaining in PHP

Exception chaining allows you to:

- Catch an exception
- Wrap it inside a new exception
- Throw the new exception while preserving the original one

Key Benefits

- **Preserves the full error history** (stack trace)
- **Adds contextual information** at each level
- **Makes debugging easier** by showing the complete flow

Basic Syntax of Exception Chaining in PHP

```php
try {
    // Some code that may fail
} catch (SpecificException $e) {
    throw new CustomException(
        "Higher-level error message",
        0, // Error code
        $e  // Previous exception (chaining)
    );
}
```

Practical Example of Exception Chaining

```
class DatabaseConnectionException extends
RuntimeException {}

function connectToDatabase() {
    try {
        // Connection attempt that fails
        throw new PDOException("Could not connect to DB
server");
    } catch (PDOException $e) {
        throw new DatabaseConnectionException(
            "Database connection failed",
            500,
            $e
        );
    }
}
```


Exception Class Hierarchy in PHP

- All exceptions derive from **Throwable**, Two main branches:
- Exception
- Error

Commonly Used Built-in Error Classes in PHP

Error Class	Description
`Error`	Base class for all internal PHP errors.
`TypeError`	Thrown when a function argument or return value is of the wrong type.
`ParseError`	Thrown when there is a syntax error during parsing.
`ArithmeticError`	Thrown when an illegal arithmetic operation occurs.
`DivisionByZeroError`	Subclass of `ArithmeticError`, occurs when dividing a number by zero.
`AssertionError`	Thrown when an assertion made via `assert()` fails.

Commonly Used Built-in Exception Classes in PHP

Exception Class	Description
`Exception`	Base class for all user-defined and built-in exceptions.
`RuntimeException`	Represents errors that occur during the program's execution.
`InvalidArgumentException`	Thrown when an argument is not of the expected type or value.
`OutOfRangeException`	Thrown when a value is not within the expected range.
`LengthException`	Thrown when the length of an argument is invalid.

Commonly Used Built-in Exception Classes in PHP

Exception Class	Description
`DomainException`	Thrown when a value does not adhere to a defined valid data domain.
`LogicException`	Base class for exceptions that represent errors in program logic.
`OverflowException`	Thrown when adding an element exceeds the allowed capacity.
`UnderflowException`	Thrown when performing an invalid operation on an empty container/structure.
`BadFunctionCallException`	Thrown when a callback refers to an undefined function.

Practical Example of File handling Exception

- Scenario: Reading a file that may not exist.
- Goal: Handle potential file-related exceptions gracefully.

```php
try {
    if (!file_exists("data.txt")) {
        throw new Exception("File not found.");
    }
    $file = fopen("data.txt", "r");
    // Process the file
    fclose($file);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    echo "Cleaning up...";
}
```

Best Practices: File Handling Exceptions

- Always check if file exists.
- Handle permissions errors.
- Use finally to close resources.

Example: Database Exception

```
try {
   $conn = new
PDO("mysql:host=localhost;dbname=test", "root",
"");
} catch (PDOException $e) {
   echo "Connection failed: " . $e-
>getMessage();
}
```

Exception Handling in Database

- PDO throws exceptions on failure.
- Prevents SQL errors from crashing the program.
- Encourages secure handling of DB operations.

Exception vs Error in Database Context

- Error: Syntax mistake in SQL query.

Exception: Connection failure, constraint violation.

Exception Handling in Forms

- Validate user input.
- Throw exceptions for invalid cases.

```
if (!filter_var($email, FILTER_VALIDATE_EMAIL))
{
  throw new Exception("Invalid email address");
}
```

Importance in Web Applications

- Prevents crashes from bad user input.
- Improves user experience with meaningful messages.
- Protects against security vulnerabilities.

What is a Custom Exception?

- **Definition**: A custom exception in PHP is a user-defined class that extends the built-in `Exception` class. It allows developers to handle specific error conditions in a more meaningful way.
- **Analogy**: Like a *"special warning light"* in a car that only turns on for a specific issue (e.g., low tire pressure) instead of a generic engine failure.
- **Examples**:
  - Negative value input
  - Division by zero
  - Invalid user action

Creating a Custom Exception in PHP

```php
<?php
// Define a custom exception class
class MyCustomException extends Exception {}

// Example usage
try {
    $value = -5;
    if ($value < 0) {
        throw new MyCustomException("Value must be
positive.");
    }
} catch (MyCustomException $e) {
    echo "Custom Exception Caught: " . $e-
>getMessage();
}
?>
```

Multiple Custom Exceptions Example

```php
<?php
class NegativeValueException extends Exception {}
class ZeroValueException extends Exception {}

function checkValue($val) {
    if ($val < 0) throw new NegativeValueException("Value is
negative");
    if ($val == 0) throw new ZeroValueException("Value is
zero");
    return true;
}

try {
    checkValue(0);
} catch (NegativeValueException $e) {
    echo $e->getMessage();
} catch (ZeroValueException $e) {
    echo $e->getMessage();
}
?>
```


Benefits of Custom Exceptions

- Provides meaningful error messages for specific situations.
- Allows fine-grained exception handling for different error types.
- Improves readability and maintainability of code

Default Exception Handler in PHP

- **Definition**:
  PHP has a built-in mechanism to handle uncaught exceptions. If an exception is thrown but not caught using a `try-catch` block, the **default exception handler** is invoked automatically. It stops script execution and displays an error message.
- **Analogy**:
  Like a *"backup safety net"*—if no one catches the falling object, the net catches it to prevent a complete disaster.
- **Behavior**:
  - Stops script execution when an uncaught exception occurs.
  - Outputs the exception message, code, and stack trace.
  - Can be overridden by a **custom exception handler** using `set_exception_handler()`.

Example: Default Exception Handler

```php
<?php
class MyException extends Exception {}

function testFunction() {
    throw new MyException("Something went wrong!");
}

// No try-catch block here
testFunction();
?>
```

- The exception MyException is thrown but not caught.
- PHP invokes the default handler, prints the error message, and terminates the script.

Overriding the Default (Global) Exception Handler

- You can register a custom exception handler to handle all uncaught exceptions:

```
<?php
function myExceptionHandler($exception) {
    echo "Custom Handler: " . $exception-
>getMessage();
}

set_exception_handler('myExceptionHandler');

throw new Exception("Test exception");
?>
```

Benefits of a Custom Handler:

- Provides cleaner error messages for users.
- Allows logging of exceptions.
- Can prevent script termination in some cases.

Practical Problems on Custom Exception Handling in PHP

Problem 1: Negative Input for Age

Scenario:

You are building a registration form where users must enter their age. Age cannot be negative.

```php
<?php
class NegativeAgeException extends Exception {}

function registerUser($age) {
    if ($age < 0) {
        throw new NegativeAgeException("Age cannot be negative.");
    }
    echo "Registration successful!";
}

try {
    registerUser(-5);
} catch (NegativeAgeException $e) {
    echo $e->getMessage();
}
?>
```


Practical Problems on Custom Exception Handling in PHP

Problem 2: DivisionByZeroException

```
<?php
class DivisionByZeroException extends Exception {}

function divide($a, $b) {
    if ($b == 0) {
        throw new DivisionByZeroException("Cannot divide by
zero.");
    }
    return $a / $b;
}

try {
    echo divide(10, 0);
} catch (DivisionByZeroException $e) {
    echo $e->getMessage();
}
?>
```


Common Pitfalls & Best Practices

- ❌ **Pitfall:** Swallowing exceptions without logging
  ✅ *Best Practice:* Always log exceptions for future debugging.
- ❌ **Pitfall:** Using generic `Exception` type for all errors
  ✅ *Best Practice:* Catch specific exception classes when possible.
- ❌ **Pitfall:** Missing `finally` block for cleanup
  ✅ *Best Practice:* Use `finally` to ensure resource release (e.g., close files, DB connections).
- ❌ **Pitfall:** Throwing exceptions inappropriately
  ✅ *Best Practice:* Use exceptions for **exceptional** cases, not regular control flow.
- ✅ **Best Practice:** Create custom exception classes for better clarity and control.

Summary & Key Takeaways

- **Exception handling** allows PHP programs to deal with errors gracefully.
- Use `try`, `throw`, `catch`, and `finally` blocks effectively.
- Handle different exceptions using **multiple `catch` blocks**.
- Filters in PHP are used for **validating** and **sanitizing** user input.
- Follow **best practices**: log errors, use specific exceptions, and clean up resources.
- Aim for **robust, maintainable, and secure** code.

Q&A / Discussion

- ❓ **What happens if an exception isn't caught?**
  ➤ The script terminates and displays a fatal error message unless a global handler is defined.
- ❓ **When would you use multiple catch blocks in real projects?**
  ➤ When different types of errors need different handling logic.
  Example: A `FileNotFoundException` may prompt the user to upload a file, while a `DatabaseException` might log the error and show a maintenance message.

Hands-on Exercise- Division by zero

🔧 **Task:** Implement basic exception handling in PHP.

Scenario:

Write a PHP script that:

- Accepts a number from the user.
- Attempts to divide 100 by the input number.
- Catches and handles division by zero using exception handling.
- Displays appropriate success or error messages.

Bonus:

- Add a `finally` block to display “Execution completed.”
- Try adding a custom exception for negative inputs.

Hands-on Exercise- Division by zero

```php
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    try {
        // Get user input
        $number = $_POST['number'] ?? null;

        // Validate input
        if (!is_numeric($number)) {
            throw new InvalidArgumentException("Please enter a valid number.");
        }
        // Check for division by zero
        if ($number == 0) {
            throw new DivisionByZeroError("Cannot divide by zero.");
        }
        // Perform division
        $result = 100 / $number;
        // Success message
        echo "Success! 100 divided by $number is $result.";
    } catch (DivisionByZeroError $e) {
        echo "Error: " . $e->getMessage();
    } catch (InvalidArgumentException $e) {
        echo "Error: " . $e->getMessage();
    }
}
?>
<!-- HTML form for user input -->
<form method="post" action="">
    <label for="number">Enter a number:</label>
    <input type="text" name="number" id="number" required>
    <button type="submit">Divide</button>
</form>
```

1. Handling Invalid Argument Exception

🔧 **Task:** Validate user input type.

Scenario:

Write a PHP script that:

- Accepts a user age input.
- Throws an `InvalidArgumentException` if the input is not numeric or less than 0.
- Catches and displays an error message if the exception occurs.
- Displays a success message if the input is valid.

1. Handling Invalid Argument Exception

```php
<?php
// Check if form is submitted
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    try {
        // Get age input from user
        $age = $_POST['age'] ?? null;

        // Validate age
        if (!is_numeric($age) || $age < 0) {
            throw new InvalidArgumentException("Age must be a positive number.");
        }

        // Success message
        echo "Success! Your age is $age.";

    } catch (InvalidArgumentException $e) {
        // Display error message
        echo "Error: " . $e->getMessage();
    }
}
?>

<!-- Simple HTML form for input -->
<form method="post" action="">
    <label for="age">Enter your age:</label>
    <input type="text" name="age" id="age" required>
    <button type="submit">Submit</button>
</form>
```

2. Handling Out of Range Exception

🔧 Task: Validate a number within a range.

Scenario: Write a PHP script that:

- Accepts a rating from 1 to 5.
- Throws an OutOfRangeException if the input is outside the 1–5 range.
- Catches the exception and displays a warning message.
- Displays the rating if it is valid.

2. Handling Out of Range Exception

```php
<?php
try {
    $rating = $_POST['rating'] ?? null;
    if ($rating < 1 || $rating > 5) {
        throw new OutOfRangeException("Rating must be between 1 and 5.");
    }
    echo "Your rating: $rating";
} catch (OutOfRangeException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```


3. Handling Length Exception

🔧 Task: Validate string length.

Scenario: Write a PHP script that:

- Accepts a username input from the user.
- Throws a LengthException if the username is less than 5 characters.
- Catches the exception and displays a message to the user.
- Confirms success if the username meets the length requirement.

3. Handling Length Exception

```php
<?php
try {
    $username = $_POST['username'] ?? '';
    if (strlen($username) < 5) {
        throw new LengthException("Username must be at least 5 characters long.");
    }
    echo "Username accepted: $username";
} catch (LengthException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```


4. Handling Type Error

🔧 Task: Validate function argument types.

Scenario: Write a PHP script that:

- Defines a function that multiplies two numbers.
- Passes user input to the function.
- Throws a TypeError automatically if non-numeric values are passed.
- Catches the TypeError and display a message to the user.

4. Handling Type Error

```php
<?php
function multiply(int $a, int $b): int {
    return $a * $b;
}

try {
    $x = $_POST['x'] ?? null;
    $y = $_POST['y'] ?? null;
    $result = multiply($x, $y);
    echo "Result: $result";
} catch (TypeError $e) {
    echo "Error: " . $e->getMessage();
}
?>
```


5. Handling Runtime Exception

🔧 Task: Handle unexpected runtime errors.

Scenario: Write a PHP script that:

- Reads a file uploaded by the user.
- Throws a RuntimeException if the file does not exist or cannot be read.
- Catches the exception and shows an error message.
- Confirms success if the file is read properly

5. Handling Runtime Exception

```php
<?php
try {
    $file = $_POST['filename'] ?? '';
    if (!file_exists($file) || !
is_readable($file)) {
        throw new RuntimeException("File does
not exist or cannot be read.");
    }
    $content = file_get_contents($file);
    echo "File content: <pre>$content</pre>";
} catch (RuntimeException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```


Question 1

Which PHP keyword is used to trigger an exception?

- A. `try`
- B. `catch`
- C. `throw`
- D. `raise`

Answer 1

Answer: C. `throw` ✅

Question 2

What is the purpose of the `catch` block in PHP?

- A. To test risky code
- B. To define custom exceptions
- C. To execute code after exception
- D. To handle the thrown exception

Answer 2

Answer: D. To handle the thrown exception ✅

Question 3

Which block will execute regardless of an exception being thrown or not?

- A. `try`
- B. `finally`
- C. `throw`
- D. `catch`

Answer 3

Answer: B. `finally` ✅

Question 4

What happens when an exception is thrown but not caught?

- A. PHP ignores it
- B. The script terminates with a fatal error
- C. The catch block is skipped
- D. The program runs normally

Answer 4

Answer: B. The script terminates with a fatal error ✅

Question 5

How do you create a custom exception class in PHP?

- A. `class MyException extends Throwable {}`
- B. `class MyException extends Error {}`
- C. `class MyException extends Exception {}`
- D. `class Exception extends MyException {}`

Answer 5

Answer: C. `class MyException extends Exception {}` ✅

Question 6

Which of the following is a built-in PHP error class for numeric calculation issues?

- A. `ArithmeticError`
- B. `RuntimeException`
- C. `LogicException`
- D. `LengthException`

Answer 6

Answer: A. `ArithmeticError` ✅

Question 7

Which PHP exception is thrown when you try to access an invalid array index?

- A. `OutOfBoundsException`
- B. `InvalidArgumentException`
- C. `RuntimeException`
- D. `DivisionByZeroError`

Answer 7

Answer: A. `OutOfBoundsException` ✅

Question 8

Which error is thrown when dividing a number by zero in PHP 7+?

- A. `ArithmeticError`
- B. `DivisionByZeroError`
- C. `TypeError`
- D. `RuntimeException`

Answer 8

Answer: B. `DivisionByZeroError` ✅

Question 9

Which exception should be used when a function argument does not meet required conditions?

- A. `RuntimeException`
- B. `InvalidArgumentException`
- C. `LengthException`
- D. `LogicException`

Answer 9

Answer: B. `InvalidArgumentException` ✅

Question 10

What type of exception is thrown if a method or function receives a value of the wrong type?

- A. `TypeError`
- B. `OutOfRangeException`
- C. `RuntimeException`
- D. `DivisionByZeroError`

Answer 10

Answer: A. `TypeError` ✅

What is a PHP Filter?

- **Definition**: A PHP filter is a built-in feature used to validate and sanitize external input such as data from forms, URLs, or cookies. It ensures the data is safe and in the expected format before processing.

- **Analogy**: Like a *"water purifier"*, filters clean or check the input before you consume it. You only get safe and valid data.

- **Examples**:
  - Validate if an email is correct (`FILTER_VALIDATE_EMAIL`)
  - Check if an age is a valid integer (`FILTER_VALIDATE_INT`)
  - Remove unwanted HTML tags from input (`FILTER_SANITIZE_STRING`)

Why Use PHP Filters?

- Prevent invalid or malicious data from entering the system.
- Protect against security threats like SQL injection and XSS.
- Reduce the need for writing custom validation code.

Key Functions in PHP Filters

Function	Description
`filter_var()`	Filters a single variable
`filter_input()`	Gets and filters input
`filter_var_array()`	Filters multiple variables

Commonly Used PHP Filters

Validation Filters

Filter Constant	Description
`FILTER_VALIDATE_INT`	Checks if the value is a valid integer.
`FILTER_VALIDATE_FLOAT`	Checks if the value is a valid floating-point number.
`FILTER_VALIDATE_EMAIL`	Checks if the value is a valid email address.
`FILTER_VALIDATE_URL`	Checks if the value is a valid URL.
`FILTER_VALIDATE_IP`	Checks if the value is a valid IP address (IPv4 or IPv6).

Commonly Used PHP Filters

Sanitizing Filters

Filter Constant	Description
`FILTER_SANITIZE_STRING`	Removes tags and encodes/strips special characters from a string. *(Deprecated in PHP 8)*
`FILTER_SANITIZE_EMAIL`	Removes all illegal characters from an email.
`FILTER_SANITIZE_URL`	Removes all illegal characters from a URL.
`FILTER_SANITIZE_NUMBER_INT`	Removes all characters except digits, +, and −.
`FILTER_SANITIZE_NUMBER_FLOAT`	Removes all characters except digits, +, −, and . (decimal).

Input Validation vs Input Sanitization

Aspect	Input Validation	Input Sanitization
Definition	Ensures that the input data is in the correct format and meets required rules.	Cleans the input data by removing or modifying unwanted/illegal characters.
Purpose	To **check correctness** of data.	To **clean and make safe** the data for use.
Example Use Case	Check if an email is valid (`FILTER_VALIDATE_EMAIL`).	Remove illegal characters from an email (`FILTER_SANITIZE_EMAIL`).
Result	Returns **true/false** or the validated value.	Returns a **modified version** of the input.
When to Use	When you need to enforce rules (e.g., valid age, proper URL).	When you need to clean input before storing or displaying it.
PHP Functions	`FILTER_VALIDATE_*` filters.	`FILTER_SANITIZE_*` filters.

Example: Validate Email

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

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

Example: Sanitizing

```php
<?php
$str = "<h1>Hello</h1>
<script>alert('hack');</script>";
$cleanStr = filter_var($str,
FILTER_SANITIZE_STRING);

echo $cleanStr;
?>
```

Filter Options in PHP

- **Definition**:
  Filter options in PHP allow developers to provide **extra rules** when validating or sanitizing data. These options make filters more flexible and precise.
- **Analogy**:
  Think of filter options as *"special instructions"* to a water purifier — not just filtering, but also controlling how much water flows, or at what temperature.

Example 1: Validate Integer with Range

```php
<?php
$age = 25;

// Validate age between 18 and 30
$options = array(
    "options" => array(
        "min_range" => 18,
        "max_range" => 30
    )
);

if (filter_var($age, FILTER_VALIDATE_INT, $options)) {
    echo "Valid age";
} else {
    echo "Invalid age";
}
?>
```

Common Filter Options

Filter Type	Option/Flag	Description
Validation (INT)	`min_range`	Minimum acceptable value.
	`max_range`	Maximum acceptable value.
Validation (IP)	`FILTER_FLAG_IPV4`	Accept only IPv4 addresses.
	`FILTER_FLAG_IPV6`	Accept only IPv6 addresses.
Validation (URL)	`FILTER_FLAG_PATH_REQUIRED`	URL must contain a path.
	`FILTER_FLAG_QUERY_REQUIRED`	URL must contain a query string.
Sanitization	`FILTER_FLAG_STRIP_LOW`	Removes characters with ASCII < 32.
	`FILTER_FLAG_STRIP_HIGH`	Removes characters with ASCII > 127.
	`FILTER_FLAG_NO_ENCODE_QUOTES`	Prevents encoding of quotes.

Example 2: Sanitize String with Flags

```php
<?php
$str = "<h1>Hello</h1> <b>World</b>";

// Strip tags but allow <b>
$clean = filter_var($str,
FILTER_SANITIZE_STRING,
            array("flags" =>
FILTER_FLAG_STRIP_HIGH));

echo $clean;
?>
```

Practical Example: Form Validation

👉 In this example:

- **email** is validated as a proper email format.
- **age** must be an integer with a minimum value of **18**.
- **website** must be a valid URL.

Handling Validation Failures

If a field does not pass validation:

- The corresponding value in `$filtered` will be set to **false**.

- If a field is missing in the input, it will be set to **null**.

👉 **For example**:

- If the user enters an invalid email → `$filtered['email'] === false`

- If the user leaves out the website field → `$filtered['website'] === null`

- If the user enters `age = 15` → `$filtered['age'] === false`

Handling Validation Failures

If a field does not pass validation:

- The corresponding value in `$filtered` will be set to **false**.

- If a field is missing in the input, it will be set to **null**.

👉 **For example**:

- If the user enters an invalid email → `$filtered['email'] === false`

- If the user leaves out the website field → `$filtered['website'] === null`

- If the user enters `age = 15` → `$filtered['age'] === false`

Handling Validation Failures

- INPUT_POST is a constant that tells PHP to fetch values from the $_POST superglobal (form data sent using the POST method).

```php
<?php
$inputs = [
    'email' => FILTER_VALIDATE_EMAIL,
    'age'   => [
        'filter'  => FILTER_VALIDATE_INT,
        'options' => ['min_range' => 18]
    ],
    'website' => FILTER_VALIDATE_URL
];

$filtered = filter_input_array(INPUT_POST, $inputs);

if ($filtered['email'] === false) {
    echo "Invalid email address.<br>";
}
if ($filtered['age'] === false) {
    echo "Age must be 18 or older.<br>";
}
if ($filtered['website'] === null) {
    echo "Website field was not provided.<br>";
}
?>
```

Question 1

How do you create a custom exception class in PHP?

- A. `class MyException extends Throwable {}`
- B. `class MyException extends Error {}`
- C. `class MyException extends Exception {}`
- D. `class MyException implements Exception {}`

Answer 1

Answer: C. `class MyException extends Exception {}` ✅

Question 2

What is the main advantage of creating a custom exception in PHP?

- A. To reduce code execution time
- B. To provide more meaningful error handling
- C. To avoid using try-catch blocks
- D. To replace built-in exceptions

Answer 2

Answer: B. To provide more meaningful error handling ✅

Question 3

Which of the following PHP filters is used to validate an email address?

- A. `FILTER_SANITIZE_EMAIL`
- B. `FILTER_VALIDATE_EMAIL`
- C. `FILTER_VALIDATE_INT`
- D. `FILTER_SANITIZE_STRING`

Answer 3

Answer: B. `FILTER_VALIDATE_EMAIL` ✅

Question 4

Which PHP filter removes all characters except digits, plus and minus sign?

- A. `FILTER_VALIDATE_INT`
- B. `FILTER_SANITIZE_STRING`
- C. `FILTER_SANITIZE_NUMBER_INT`
- D. `FILTER_VALIDATE_FLOAT`

Answer 4

Answer: C. `FILTER_SANITIZE_NUMBER_INT` ✅

Question 5

Which filter would you use to check if a variable contains a valid integer?

- A. `FILTER_VALIDATE_INT`
- B. `FILTER_SANITIZE_NUMBER_INT`
- C. `FILTER_VALIDATE_FLOAT`
- D. `FILTER_SANITIZE_STRING`

Answer 5

Answer: A. `FILTER_VALIDATE_INT` ✅

Question 6

What happens if a variable does not pass `FILTER_VALIDATE_EMAIL`?

- A. It returns `false`
- B. It throws an exception
- C. It returns the original variable
- D. It returns `null`

Answer 6

Answer: A. It returns `false` ✅

Question 7

Which of the following is TRUE about `FILTER_SANITIZE_STRING`?

- A. It validates if the string is safe
- B. It removes tags and encodes special characters
- C. It checks if the string is not empty
- D. It only works on numeric values

Answer 7

Answer: B. It removes tags and encodes special characters ✅

Question 8

When defining a custom exception in PHP, which keyword is used to throw it?

- A. `catch`
- B. `throw`
- C. `finally`
- D. `error`

Answer 8

Answer: B. `throw` ✅

Question 9

Which filter would you use to check if an input value is a valid IP address?

- A. `FILTER_VALIDATE_URL`
- B. `FILTER_VALIDATE_IP`
- C. `FILTER_SANITIZE_STRING`
- D. `FILTER_VALIDATE_DOMAIN`

Answer 9

Answer: B. `FILTER_VALIDATE_IP` ✅

Question 10

Which statement is correct about using filters in PHP?

- A. Filters are only available in PHP 8 and above
- B. Filters can only be applied to integers
- C. Filters are used for both validating and sanitizing input
- D. Filters replace the need for exceptions

Answer 10

Answer: C. Filters are used for both validating and sanitizing input ✅

Practical Problem: Registration Form Validation

Design a **registration form** in PHP that collects the following inputs:

- **Name** (string, required, cannot be empty)
- **Email** (must be a valid email address)
- **Age** (must be an integer and ≥ 18)
- **Website** (optional, but if provided must be a valid URL)

Instructions:

1. Use the `filter_input_array()` function with appropriate filters (`FILTER_VALIDATE_EMAIL`, `FILTER_VALIDATE_INT`, `FILTER_VALIDATE_URL`).
2. Handle validation failures:
   - If invalid, the value should return `false`.
   - If missing, the value should return `null`.
3. Display a **success message** if all inputs are valid, otherwise display an **error message** showing which fields failed.

Example Code

```php
<?php
$inputs = [
    'name' => [
        'filter' => FILTER_CALLBACK,
        'options' => 'trim'
    ],
    'email' => FILTER_VALIDATE_EMAIL,
    'age'   => [
        'filter'  => FILTER_VALIDATE_INT,
        'options' => ['min_range' => 18]
    ],
    'website' => FILTER_VALIDATE_URL
];

$filtered = filter_input_array(INPUT_POST, $inputs);

if ($filtered) {
    if (in_array(false, $filtered, true)) {
        echo "❌ Validation failed. Please check your inputs.<br>";
        print_r($filtered);
    } else {
        echo "✅ Registration successful!";
    }
}
?>
```

Task

👉 Implement the above registration form with fields for Name, Email, Age, and Website. 👉 Submit the form using POST and test different inputs:

- Invalid email
- Age below 18
- Empty website field 👉 Observe how PHP filters handle each case.