Form Data Handling

- PHP uses superglobals (Always Accessible Variables) to access form data. These variables are available throughout your script regardless of scope.

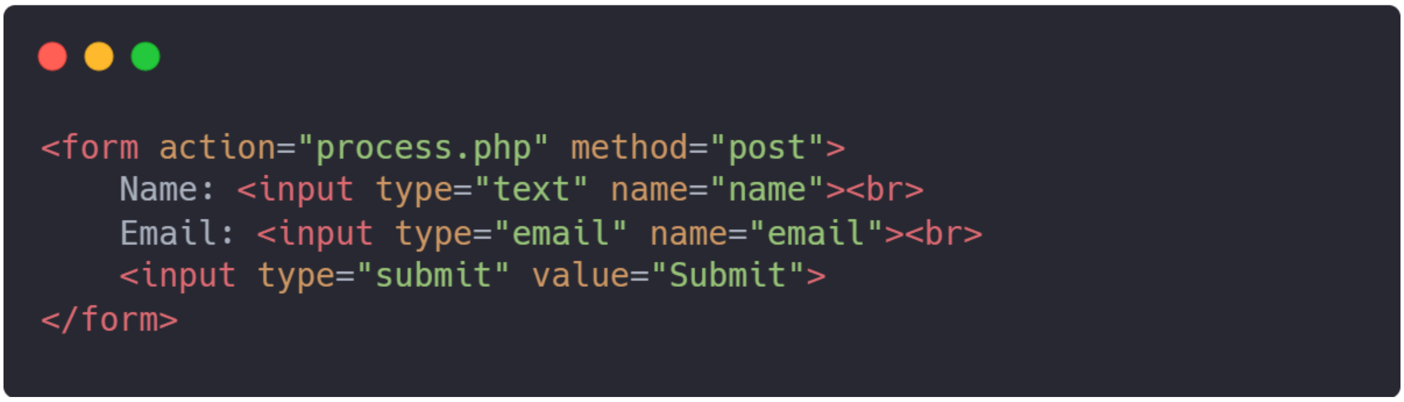
The two main ones for forms are:

- `$_GET`: Used for data sent through the URL with the GET method (less secure, limited data size).
- `$_POST`: Used for data sent hidden within the request body with the POST method (more secure, larger data size).

Submitting Forms:

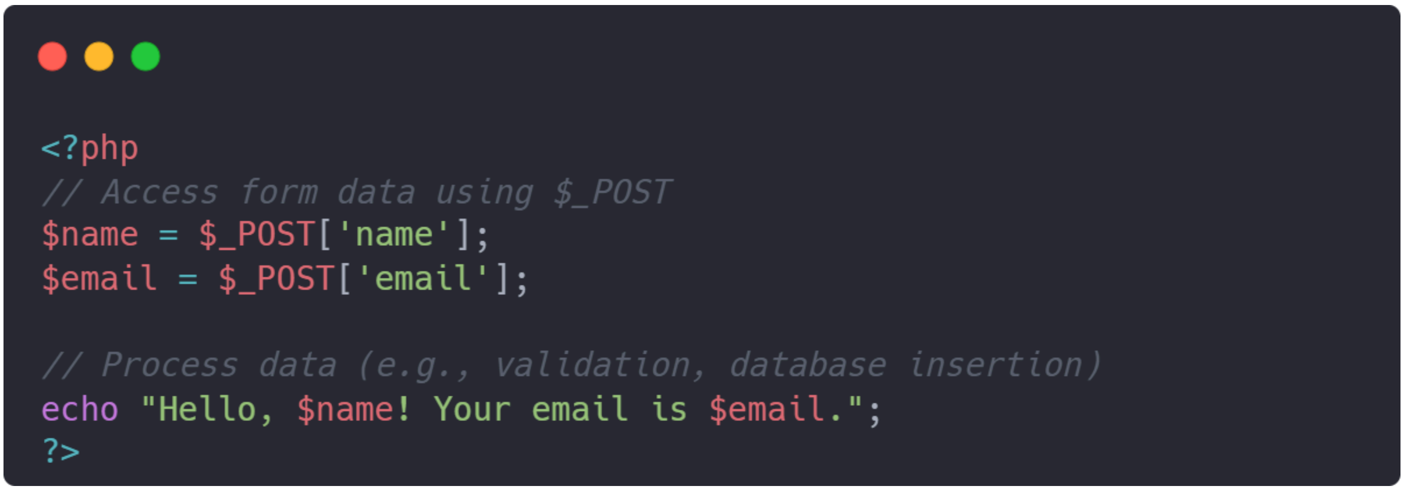
- HTML forms specify the action using the `action` attribute in the `<form>` tag. This points to the PHP script that will handle the data.
- The `method` attribute defines how the data is sent (GET or POST).

Example (POST method):



```
<form action="process.php" method="post">
    Name: <input type="text" name="name"><br>
    Email: <input type="email" name="email"><br>
    <input type="submit" value="Submit">
</form>
```

- **PHP script (process.**



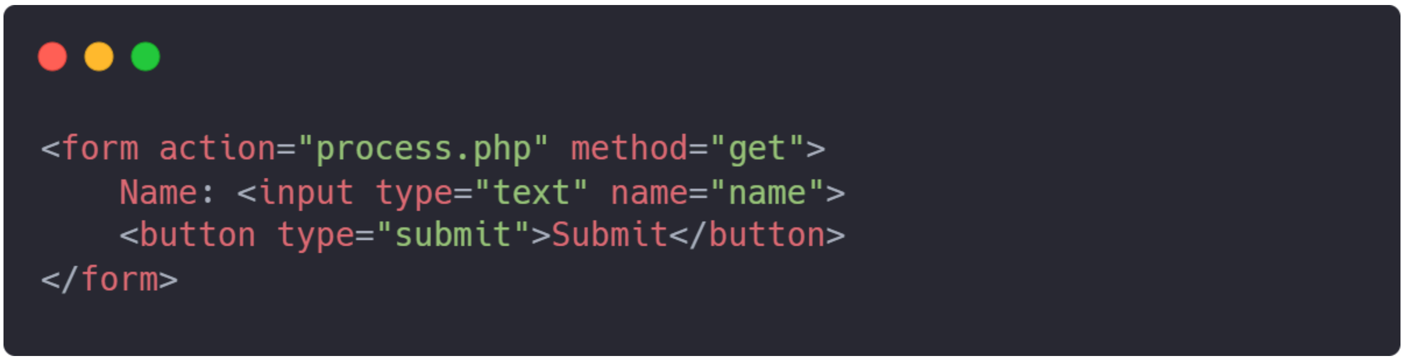
```
<?php
// Access form data using $_POST
$name = $_POST['name'];
$email = $_POST['email'];

// Process data (e.g., validation, database insertion)
echo "Hello, $name! Your email is $email.";
?>
```

Superglobal Variables

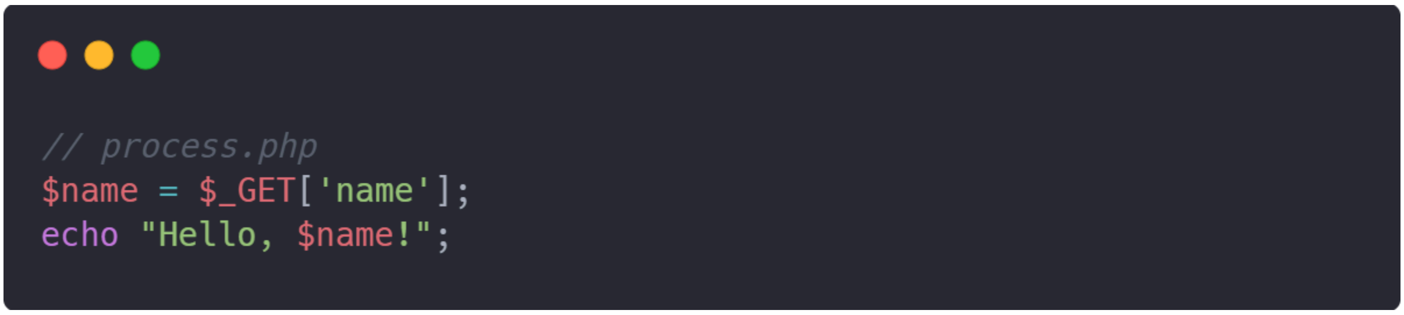
- In PHP, these variables are accessible throughout your script without the need for global declarations. Here are some of the Superglobal variables:

$_GET

- **Purpose:** Captures data passed through a URL query string.
- **Accessibility:** Values are visible in the URL after a `?` symbol, making them less secure for sensitive information.
- **Example:**



```
<form action="process.php" method="get">
    Name: <input type="text" name="name">
    <button type="submit">Submit</button>
</form>
```



```
// process.php
$name = $_GET['name'];
echo "Hello, $name!";
```

When the form is submitted, the URL might look like `process.php?name=John`. The `process.php` script can access the `name` value using `$_GET['name']`.

$_POST

- **Purpose:** Captures data submitted through an HTML form using the `POST` method.
- **Accessibility:** Values are not visible in the URL, making them more secure for sensitive data.
- **Example:**

```
<form action="process.php" method="post">
    Password: <input type="password" name="password">
    <button type="submit">Submit</button>
</form>
```

```
// process.php
$password = $_POST['password'];
// Validate and process password securely
```

$_REQUEST

- **Purpose:** Combines data from both $_GET and $_POST, along with cookies (if enabled).
- **Caution:** Use with care due to potential security risks. Data from external sources like cookies might be manipulated.
- **Example:**

```
$data = $_REQUEST['some_data']; // Could be from GET, POST, or
cookies
```

Cookies

- A cookie is a small piece of data (usually text) that a web server stores on the user's computer.
- It's used to remember information about the user, such as login preferences, browsing history, or items added to a shopping cart.
- Cookies are essential for maintaining user state across different web pages on the same website.

Need for Cookies

- **State Management:** Cookies help websites remember user actions or preferences across different pages, providing a seamless and personalized experience.
- **Authentication:** Cookies can store login information (e.g., username, session ID) to keep users logged in without re-entering credentials on every page.
- **Personalization:** Websites can use cookies to tailor content or recommendations based on user preferences or browsing history.
- **Tracking:** Cookies are commonly used in web analytics to track user behavior and website usage statistics.

Setting Up a Cookie

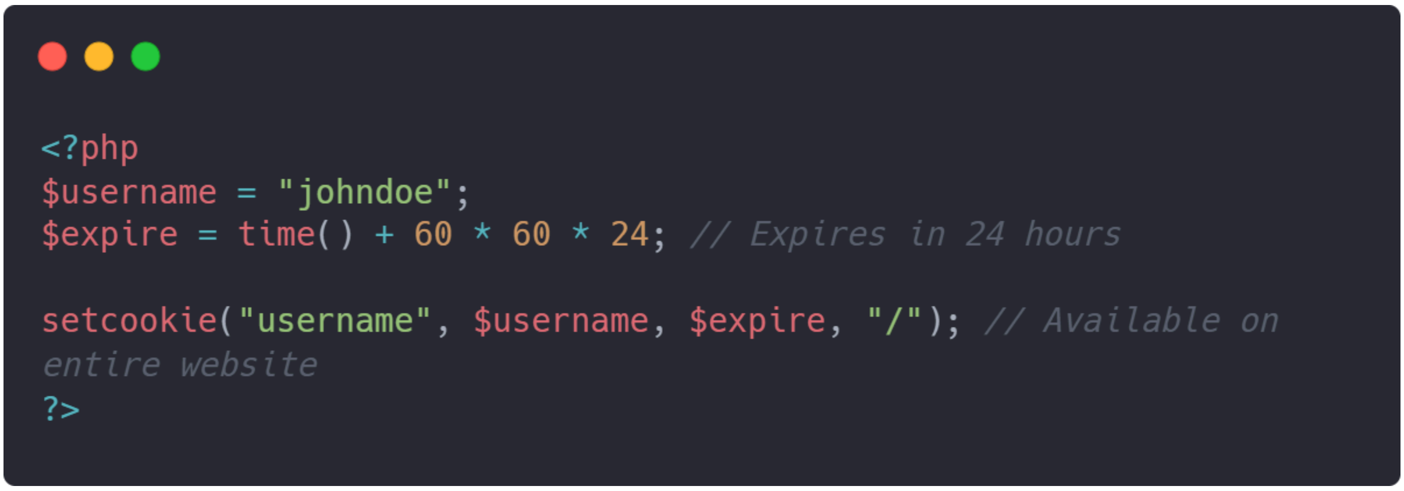
- Requires arguments like name, value, expiry time, path, domain, and security flags (optional).

1. **`setcookie()` Function:** Use the `setcookie()` function to create and send a cookie to the user's browser.

Arguments: The function takes several arguments:

- `name`: The name for the cookie (string). ,
- `value`: The data to store in the cookie (string).
- `expire`,`path`,`domain`,`secure`,`httponly` : All these are optional

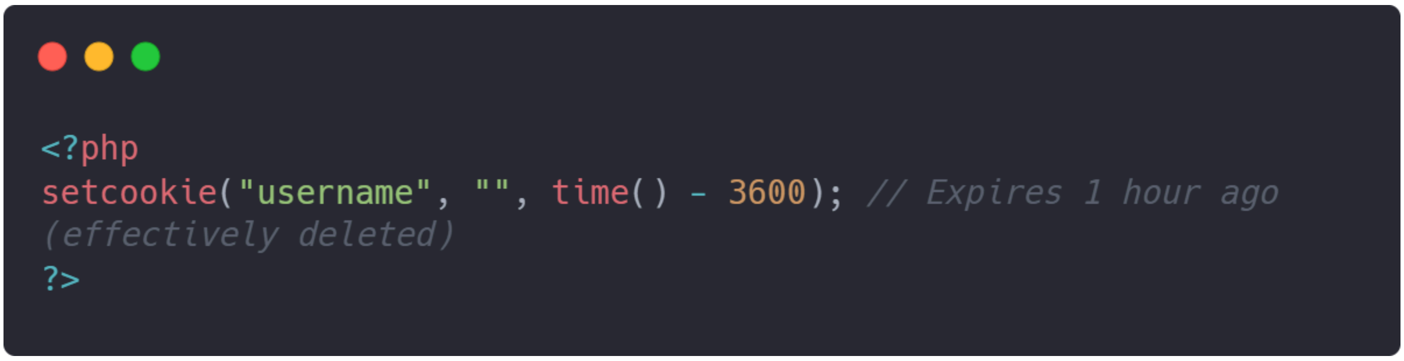
Example:



```php
<?php
$username = "johndoe";
$expire = time() + 60 * 60 * 24; // Expires in 24 hours

setcookie("username", $username, $expire, "/"); // Available on
entire website
?>
```

Deleting a Cookie

- **Set Expired Cookie:** To delete a cookie, set the `expire` argument in `setcookie()` to a time in the past.
- **Example:**



```php
<?php
setcookie("username", "", time() - 3600); // Expires 1 hour ago
(effectively deleted)
?>
```

Session Management

Concept:

- Sessions enable you to store user-specific information (like preferences, shopping cart items) across multiple pages on a website.
- Unlike cookies (which reside on the client-side), session data is stored on the server-side, enhancing security.
- Session data is temporary and typically expires when the user closes the browser.

Creating Session Variables

Initiating a Session:

- You must start a session using the `session_start()` function before creating session variables. This function checks for existing sessions and creates a new one if none exists.

Creating Session Variables:

- Use the superglobal `$_SESSION` array to store session variables. It acts like an associative array where you define a key and a value.
- Example:

```php
<?php
session_start();

$_SESSION["username"] = "johnDoe";
$_SESSION["userId"] = 123;
?>
```

Retrieving Session Variables:

- Session variables are stored in the associative array `$_SESSION`.
- You can access individual variables using their names within square brackets.

Example:

```php
<?php
session_start();

echo "Welcome " . $_SESSION["username"] . "!";
?>
```

Additional Notes:

- **Session Expiration:** Sessions typically expire when the user closes their browser or after a specific time limit (controlled by `session.cookie_lifetime` in php.ini).
- **Session IDs:** Each session is identified by a unique session ID (SID), typically stored as a cookie on the client-side.
- `session_destroy()`: Ends the current session.
- `session_unset()`: Removes a specific session variable.
- Call `session_start()` at the beginning of every page that needs to access session data.

Exception Handling

Exception handling is a robust mechanism in PHP for managing runtime errors gracefully, preventing your application from crashing and enhancing code readability. It involves three primary keywords: try, catch, and throw.

Understanding Exceptions and Errors

1. Errors:

- Unexpected conditions that halt script execution.
- Examples:
  - Trying to access a non-existent variable.
- Not recoverable within the script itself.
- Handled with functions like `die()` or displayed as a fatal error message.

2. Exceptions:

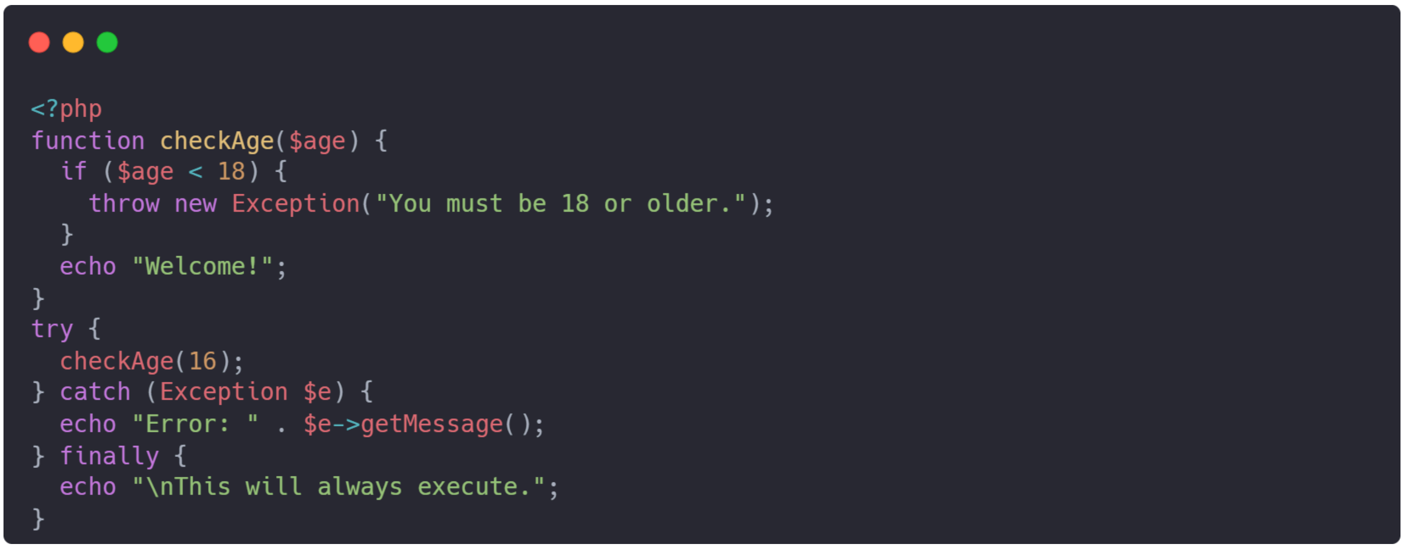
- Objects representing unexpected events.
- Thrown by PHP functions or your own code.
- Designed to be caught and handled gracefully.
- Provide information about the error (message, type, etc.).

3. Exception Handling:

- Uses `try...catch...finally` blocks.
- `try`: Code that might throw an exception.
- `catch`: Captures a specific exception type and defines how to handle it.
- `finally` (optional): Code that always executes, regardless of exceptions.

Key Points:

- Exceptions offer more control over error handling compared to basic errors.
- You can create custom exception classes for specific error scenarios.
- Proper exception handling improves code readability and maintainability.

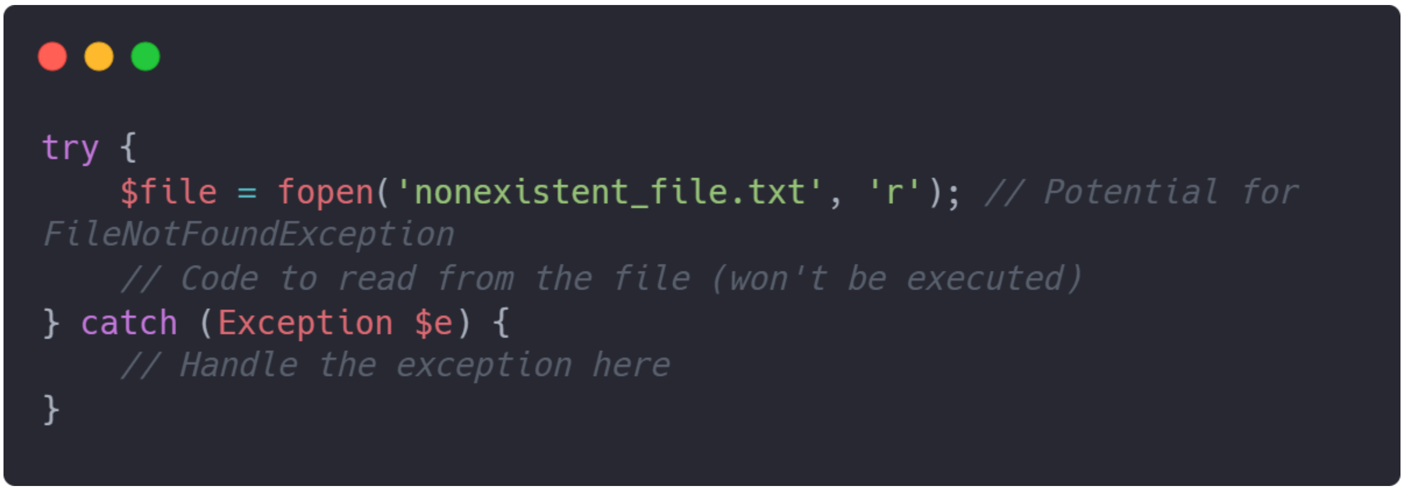


```php
<?php
function checkAge($age) {
  if ($age < 18) {
    throw new Exception("You must be 18 or older.");
  }
  echo "Welcome!";
}
try {
  checkAge(16);
} catch (Exception $e) {
  echo "Error: " . $e->getMessage();
} finally {
  echo "\nThis will always execute.";
}
```

Handling Exceptions in PHP

1. `try` Block:

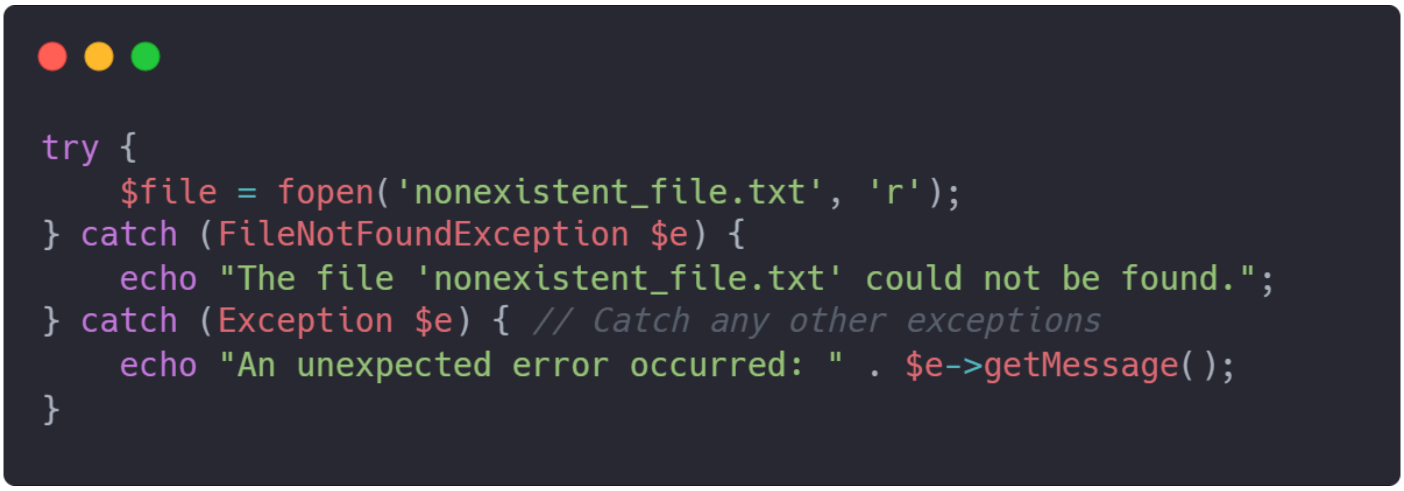
- Encapsulates code that might potentially throw exceptions.
- Execution continues within this block until an exception is thrown.
- Example:



```php
try {
    $file = fopen('nonexistent_file.txt', 'r'); // Potential for FileNotFoundException
    // Code to read from the file (won't be executed)
} catch (Exception $e) {
    // Handle the exception here
}
```

2. `catch` Block:

- Follows a `try` block and is used to capture specific exceptions or general exceptions.
- The first matching `catch` block is executed if an exception is thrown within the `try` block.
- You can access information about the exception using the caught exception object.
- Example (catching a specific exception type):

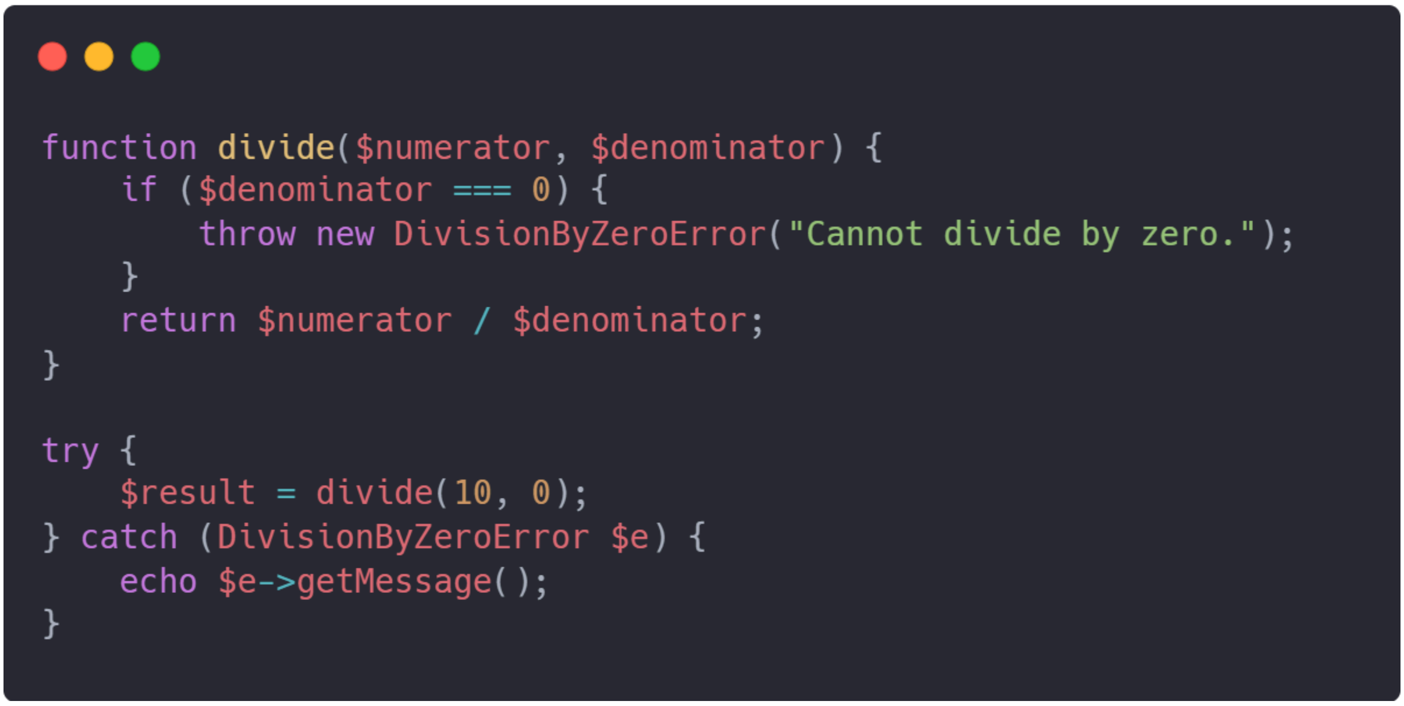


```php
try {
    $file = fopen('nonexistent_file.txt', 'r');
} catch (FileNotFoundException $e) {
    echo "The file 'nonexistent_file.txt' could not be found.";
} catch (Exception $e) { // Catch any other exceptions
    echo "An unexpected error occurred: " . $e->getMessage();
}
```

3. `throw` Keyword:

- Used to explicitly throw an exception object when an error condition is encountered.
- The exception object provides details about the error.

- Example (throwing a custom exception):



```php
function divide($numerator, $denominator) {
    if ($denominator === 0) {
        throw new DivisionByZeroError("Cannot divide by zero.");
    }
    return $numerator / $denominator;
}

try {
    $result = divide(10, 0);
} catch (DivisionByZeroError $e) {
    echo $e->getMessage();
}
```

User Defined Exceptions

What are they?

- Custom exceptions created by extending the built-in `Exception` class.
- Allow you to handle specific errors in your application.

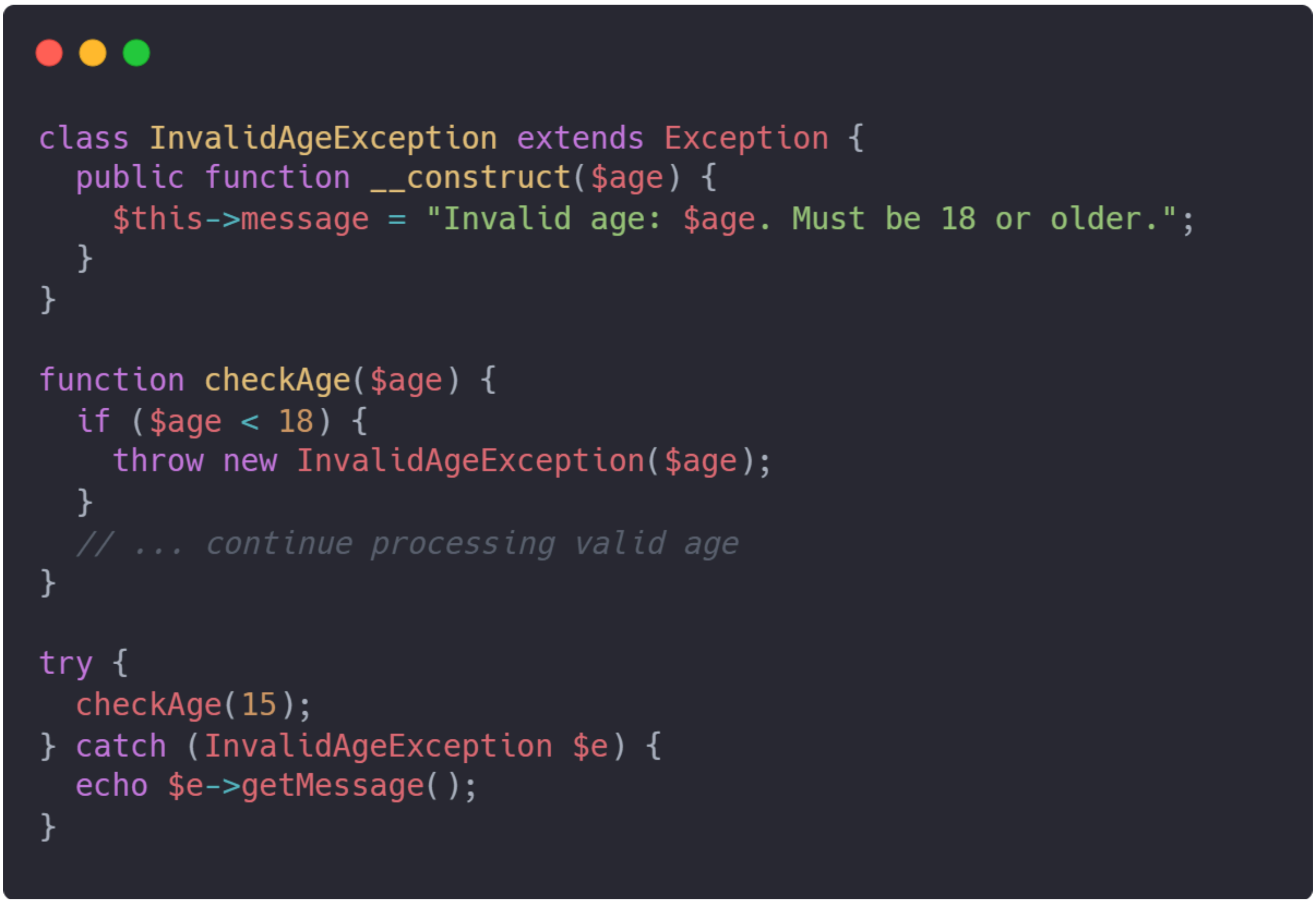
Benefits:

- Improved code readability and maintainability.
- Clearer error messages for developers.
- Centralized handling of specific error conditions.

Creating a Custom Exception Class:

- Extend the `Exception` class.
- Optionally define a constructor to set the error message.
- (Optional) Add custom methods specific to your exception type.

Example:



```php
class InvalidAgeException extends Exception {
  public function __construct($age) {
    $this->message = "Invalid age: $age. Must be 18 or older.";
  }
}

function checkAge($age) {
  if ($age < 18) {
    throw new InvalidAgeException($age);
  }
  // ... continue processing valid age
}

try {
  checkAge(15);
} catch (InvalidAgeException $e) {
  echo $e->getMessage();
}
```

Throwing Exceptions:

- Use the `throw` statement with your custom exception object.

Catching Exceptions:

- Use a `try...catch` block to handle thrown exceptions.
- Catch specific exception types or use a generic `Exception` catch.