Lecture 4

CST 226-2

Web Application Development

Sanjeewanie Senanayake

sanjeewanie@uwu.ac.lk

Department of Computer Science and Technology

Working with Databases

Lesson Learning Outcomes

» After successful completion of this lesson you will be able to,
- Understand the **importance of databases in web development** and their role in storing and managing data.
- **Establish a connection between PHP applications** and a **database server**.
- **Execute SQL queries** in PHP to retrieve, insert, update, and delete data from database tables.
- **Retrieve and manipulate data from database result sets** using various fetching methods.
- **Implement prepared statements** to enhance security and optimize database operations.
- **Handle and troubleshoot database-related errors** in PHP.

Lesson Outline

» Use of Databases in Web Development

» Establishing Database Connections

» PDO - PHP Data Objects

» Steps to use a Database

» Handling ResultSet with PDO

» Fetching Styles in PDO

» Working with Prepared Statements

» Handling Database Errors

Use of Databases in Web Development

- » Databases **provide a structured** and **organized way** to **store** and **manage data**
- » Use **primary keys**, **unique constraints**, **foreign keys**, and **data validation rules** to **maintain** the **quality** and **reliability** of data
- » **Powerful querying capabilities** that allow web applications to retrieve and manipulate data easily
- » Databases can **handle large amounts of data** and **concurrent access**
- » Databases **provide a persistent storage** solution
- » **Enhance the overall security** of web applications

Establishing Database Connections

» In PHP, there are **several approaches** to **establishing database connections**

» Most **commonly used method** is:

- **MySQLi** (MySQL Improved) extension
- **PDO** (PHP Data Objects)

» The **choice** depends on factors such as the **specific database you're working with**, **your preferred programming style**, **project requirements**, and the **need for database portability**

» In addition to using these extensions, PHP **also provides an open database connectivity** (ODBC) approach

PDO - PHP Data Objects

- A database **abstraction layer in PHP that provides a consistent and unified API** for accessing databases
- Key features:
  - **Allows** you **to interact with different databases**
  - **Built on an object-oriented architecture**, utilizing **classes** and **objects** to represent database **connections**, **statements**, and **result sets**
  - **Supports prepared statements**, which provide a secure and efficient way to execute queries
  - **Supports parameter binding**, allowing you to **bind values to placeholders** in prepared statements
  - **Enables transaction management**
  - **Provides** consistent and standardized **error handling**

Steps to Use a Database

Step 1
- Set up the database connection details

Step 2
- Create a PDO

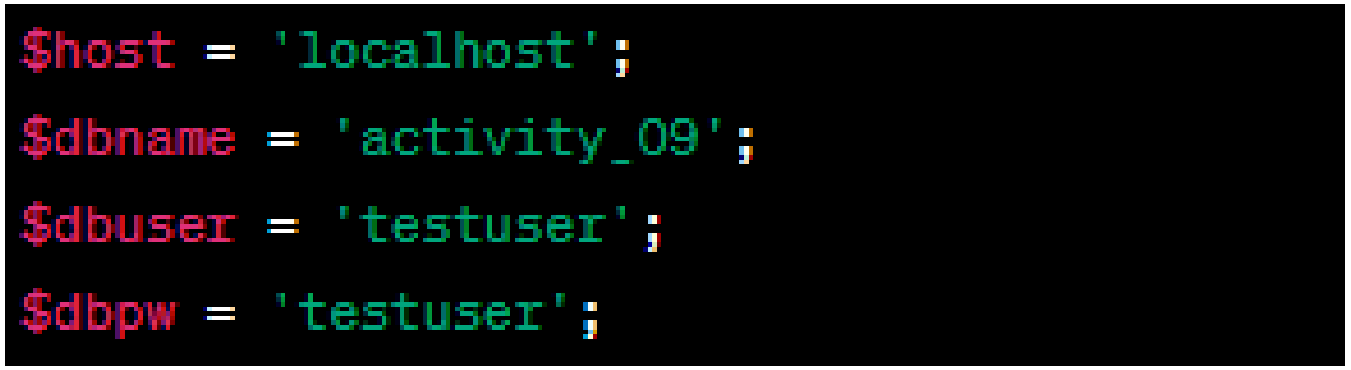
Step 3
- Perform database operations

Step 1: Set up the Database Connection

» Determine the **host address**, **database name**, **username**, and **password** for the database you want to connect to

» You can **add following** information

- **$host** = “location of your server”
- **$dbname** = “database name you are working with”
- **$dbuser** = “database user”
- **$dbpw** = “database password”



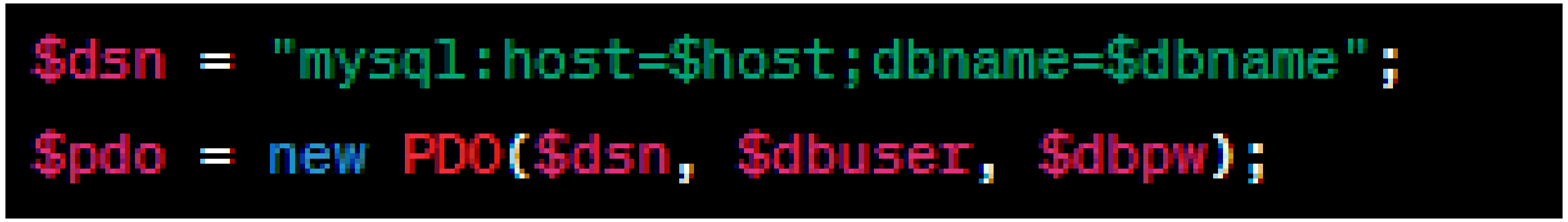
```
$host = 'localhost';
$dbname = 'activity_09';
$dbuser = 'testuser';
$dbpw = 'testuser';
```

Step 2: Create a PDO

» **Use the PDO constructor** to create a PDO and to **pass** the **necessary connection details**

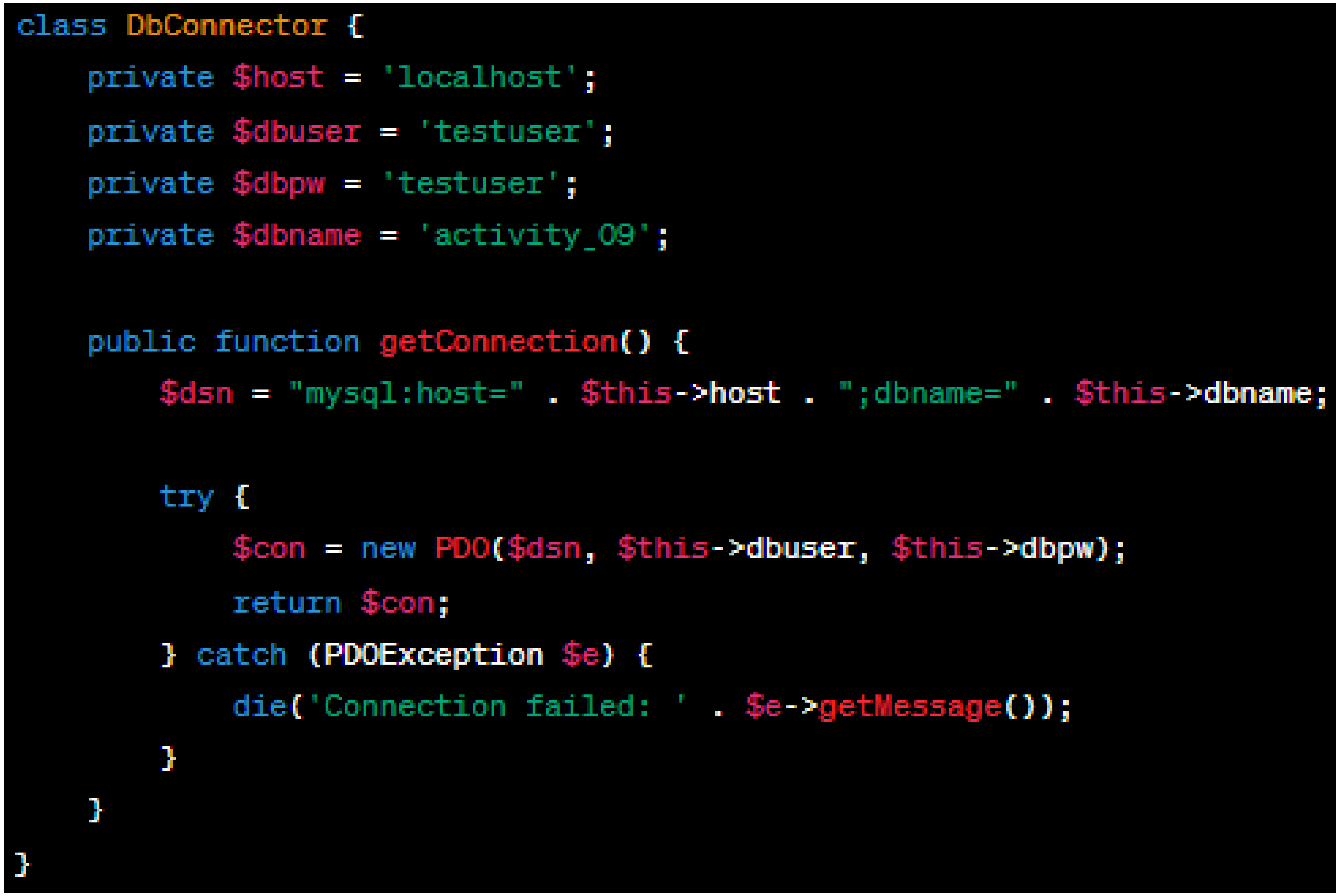
» The connection details are typically provided as a **Data Source Name (DSN)**, which includes the **database type**, **host**, and **database name**

» You also **need to specify the username and password** for authentication



```
$dsn = "mysql:host=$host;dbname=$dbname";
$pdo = new PDO($dsn, $dbuser, $dbpw);
```

Connecting to the Database



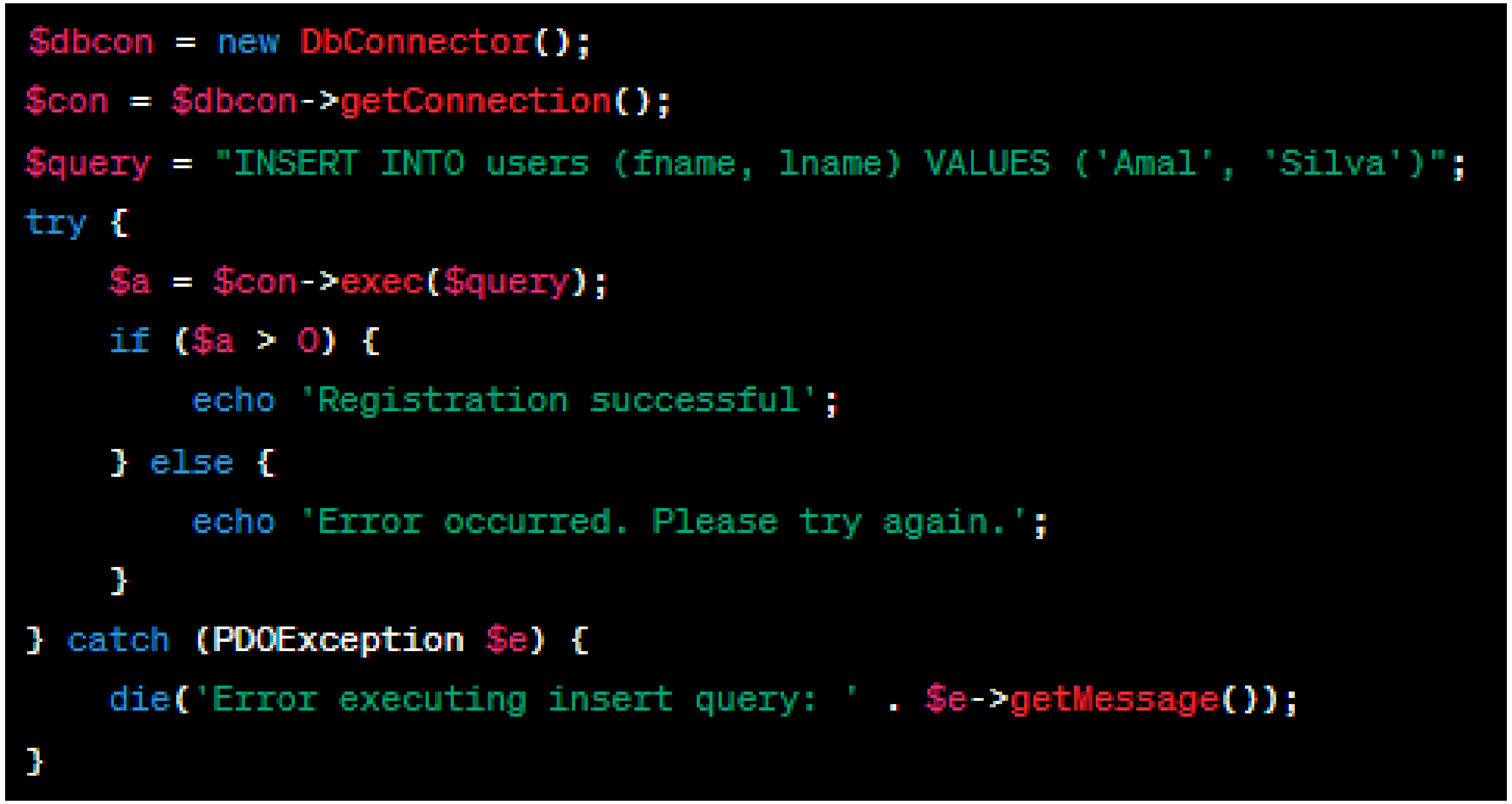
```
class DbConnector {
    private $host = 'localhost';
    private $dbuser = 'testuser';
    private $dbpw = 'testuser';
    private $dbname = 'activity_09';

    public function getConnection() {
        $dsn = "mysql:host=" . $this->host . ";dbname=" . $this->dbname;

        try {
            $con = new PDO($dsn, $this->dbuser, $this->dbpw);
            return $con;
        } catch (PDOException $e) {
            die('Connection failed: ' . $e->getMessage());
        }
    }
}
```

Step 3: Perform Database Operations

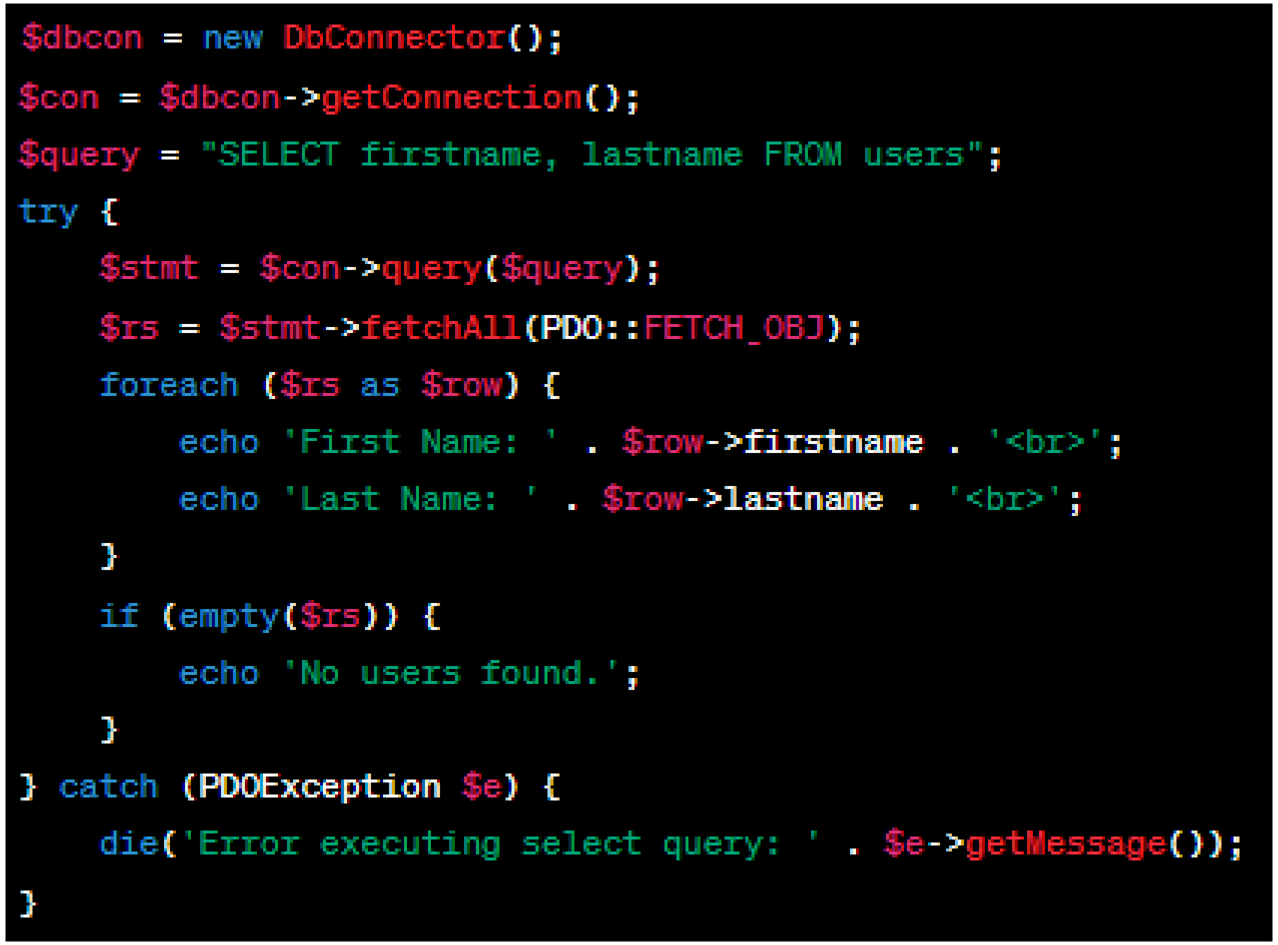
» Insert/Update/Delete Query: Use **exec()**



```
$dbcon = new DbConnector();
$con = $dbcon->getConnection();
$query = "INSERT INTO users (fname, lname) VALUES ('Amal', 'Silva')";
try {
    $a = $con->exec($query);
    if ($a > 0) {
        echo 'Registration successful';
    } else {
        echo 'Error occurred. Please try again.';
    }
} catch (PDOException $e) {
    die('Error executing insert query: ' . $e->getMessage());
}
```

Step 3: Perform Database Operations cont.

» Select Query: Use **query()**

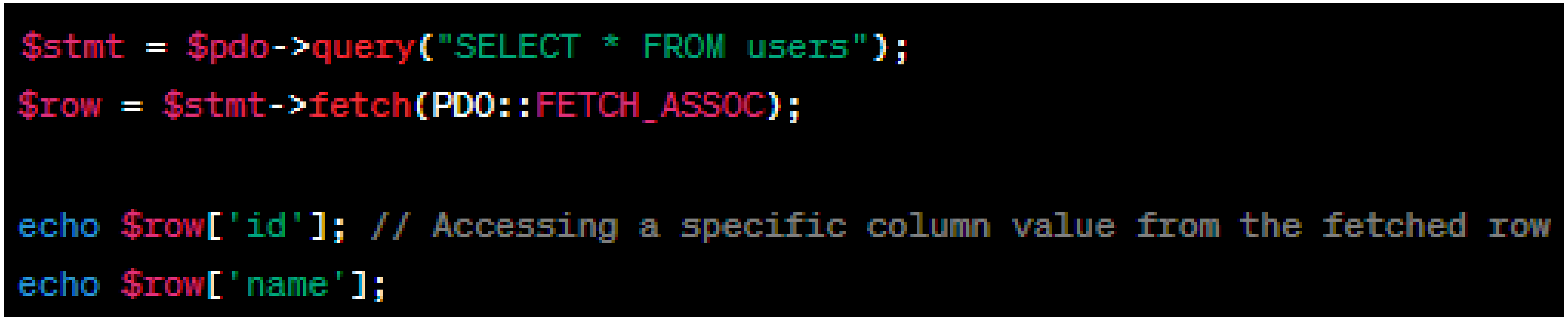


```
$dbcon = new DbConnector();
$con = $dbcon->getConnection();
$query = "SELECT firstname, lastname FROM users";
try {
    $stmt = $con->query($query);
    $rs = $stmt->fetchAll(PDO::FETCH_OBJ);
    foreach ($rs as $row) {
        echo 'First Name: ' . $row->firstname . '<br>';
        echo 'Last Name: ' . $row->lastname . '<br>';
    }
    if (empty($rs)) {
        echo 'No users found.';
    }
} catch (PDOException $e) {
    die('Error executing select query: ' . $e->getMessage());
}
```

Handling ResultSet with PDO

» There are **different methods** provided by PDO **to handle result sets** returned by database queries.

- **fetch()**
  - **Retrieves the next row from the result set** and advances the **internal pointer to the next row**
  - It allows you to **fetch a single row at a time**
  - It **supports different fetch styles** to determine how the data should be returned (i.e., FETCH_ASSOC, FETCH_OBJ, etc.)

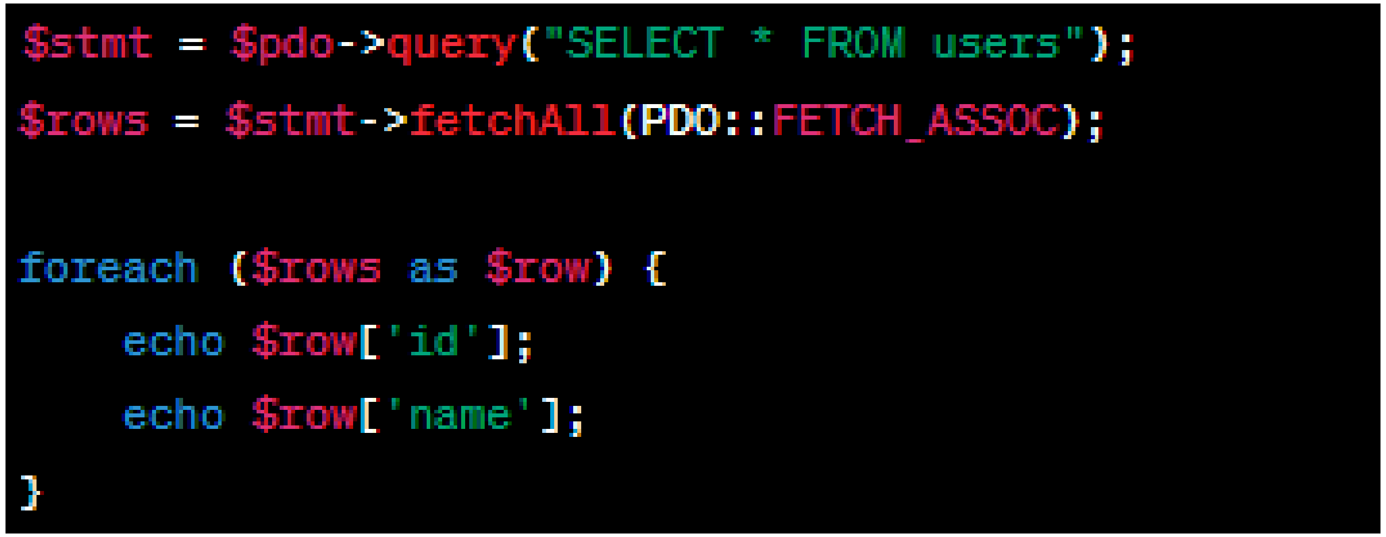


```
$stmt = $pdo->query("SELECT * FROM users");
$row = $stmt->fetch(PDO::FETCH_ASSOC);

echo $row['id']; // Accessing a specific column value from the fetched row
echo $row['name'];
```

Handling ResultSet with PDO cont.

- **fetchAll()**
  - **Retrieves all rows** from the result set
  - It **allows you to retrieve the entire result set at once**, which is useful when you need to process or display all the retrieved data
  - It **supports different fetch styles** to determine how the data should be returned (i.e., FETCH_ASSOC, FETCH_OBJ, etc.)

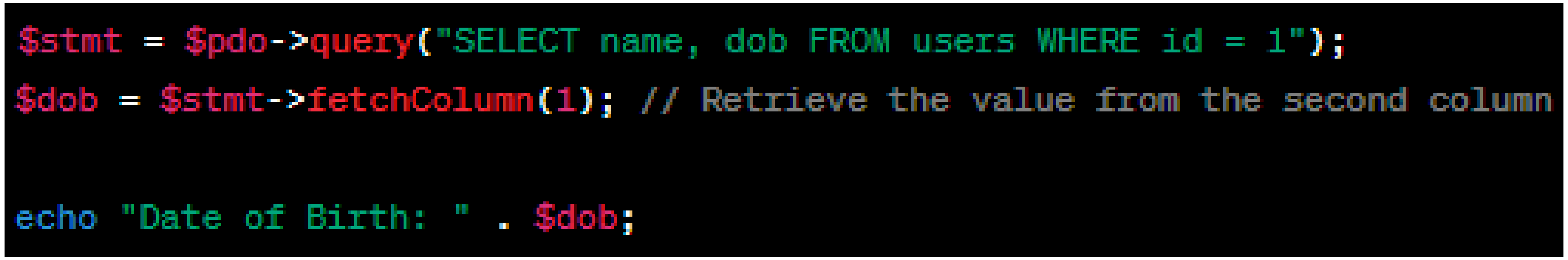


```
$stmt = $pdo->query("SELECT * FROM users");
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);

foreach ($rows as $row) {
    echo $row['id'];
    echo $row['name'];
}
```

Handling ResultSet with PDO cont.

- **fetchColumn()**
  - Retrieves **a single column value from the next row** in the result set
  - It is typically **used when you only need to fetch a specific column value** from a particular row rather than retrieving the entire row
  - The **position** of the columns **starts from 0**

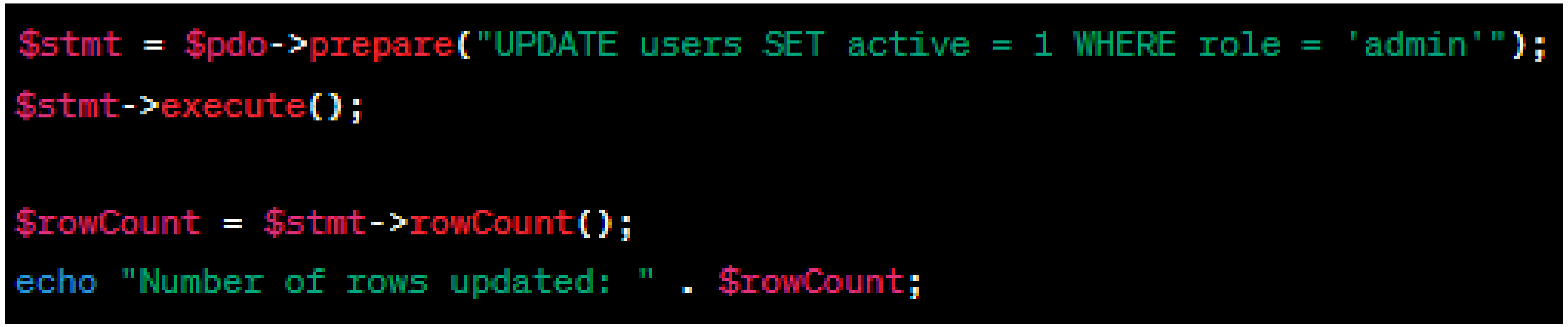


```
$stmt = $pdo->query("SELECT name, dob FROM users WHERE id = 1");
$dob = $stmt->fetchColumn(1); // Retrieve the value from the second column

echo "Date of Birth: " . $dob;
```

Handling ResultSet with PDO cont.

- **rowCount()**
  - **Returns the number of rows affected** by the previous INSERT, UPDATE, or DELETE statement
  - It can be **used to determine the number of rows that were modified or affected** by the query
  - Generally **used with prepared statements**

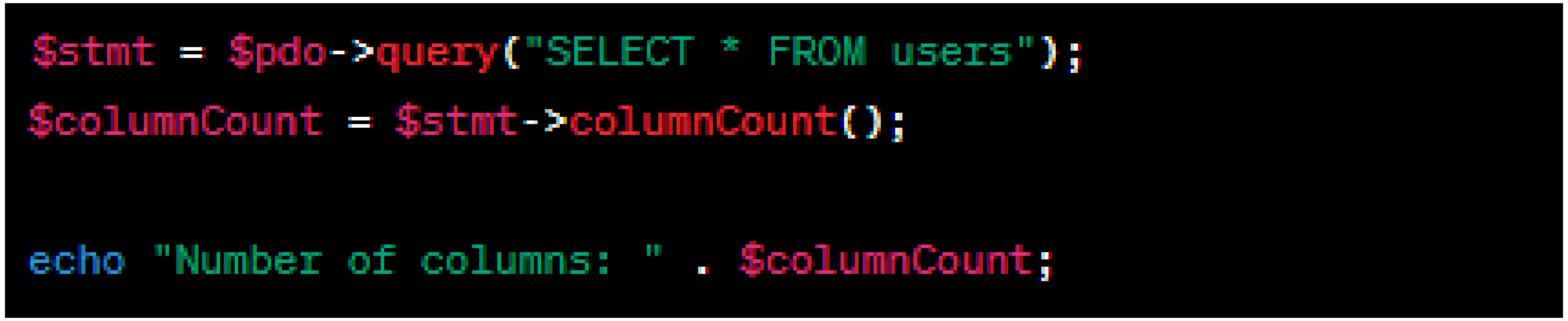


```
$stmt = $pdo->prepare("UPDATE users SET active = 1 WHERE role = 'admin'");
$stmt->execute();

$rowCount = $stmt->rowCount();
echo "Number of rows updated: " . $rowCount;
```

Handling ResultSet with PDO cont.

- **columnCount()**
  - **Returns the number of columns** in the result set
  - It **provides the ability to retrieve the count of columns returned by the query**, which can be useful for certain scenarios where you need to know the structure of the result set



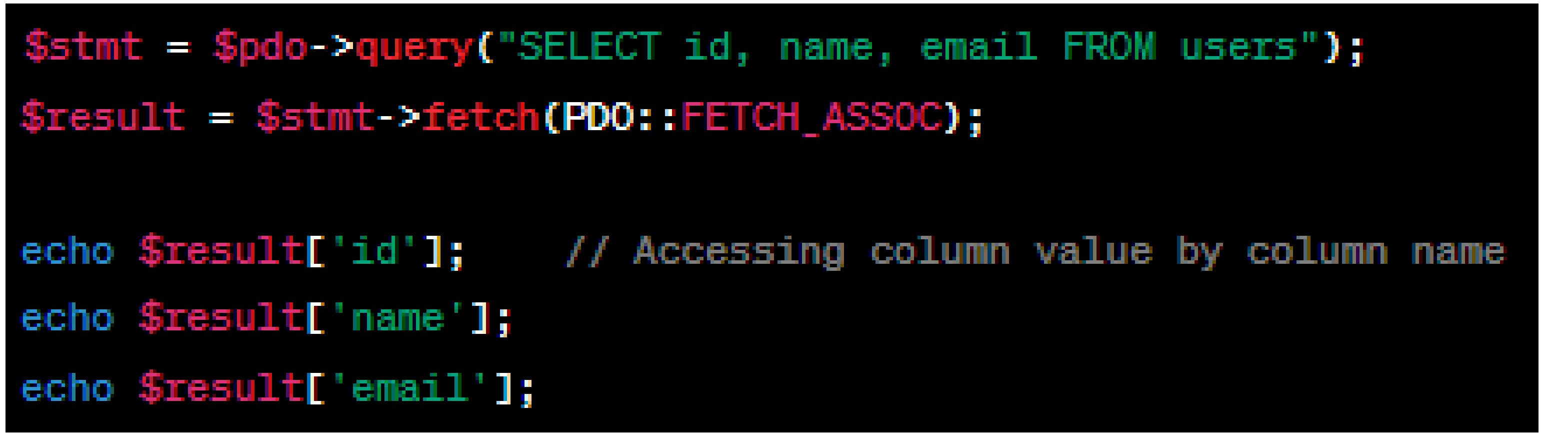
```php
$stmt = $pdo->query("SELECT * FROM users");
$columnCount = $stmt->columnCount();

echo "Number of columns: " . $columnCount;
```

Fetching Styles in PDO

» PDO **supports different fetch styles** to determine **how the data should be returned**

- **PDO::FETCH_ASSOC**
  - **Returns an associative array** where the **column names** are used as the **keys** and the **column values** are the **corresponding values**

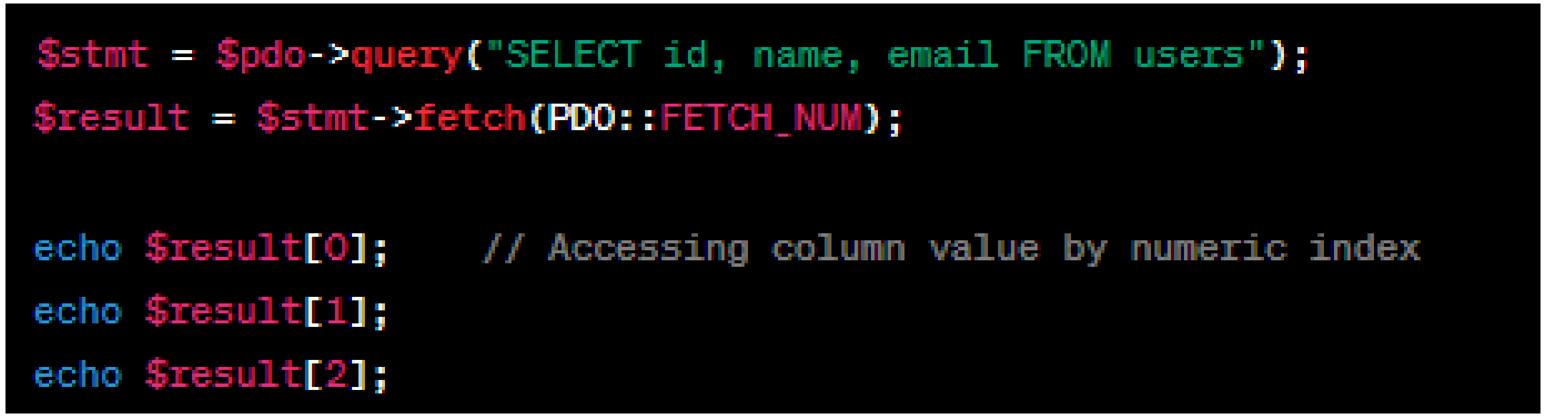


```php
$stmt = $pdo->query("SELECT id, name, email FROM users");
$result = $stmt->fetch(PDO::FETCH_ASSOC);

echo $result['id'];     // Accessing column value by column name
echo $result['name'];
echo $result['email'];
```

Fetching Styles in PDO cont.

- **PDO::FETCH_NUM**
  - **Returns a numerically indexed array** where the **column values** are **stored at numeric indices starting from 0**

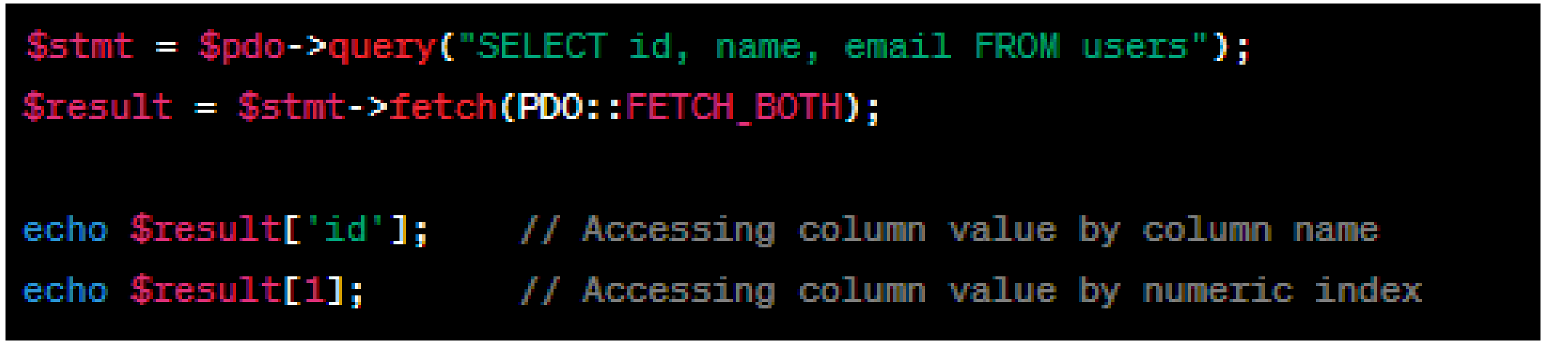


```
$stmt = $pdo->query("SELECT id, name, email FROM users");
$result = $stmt->fetch(PDO::FETCH_NUM);

echo $result[0];    // Accessing column value by numeric index
echo $result[1];
echo $result[2];
```

Fetching Styles in PDO cont.

- **PDO::FETCH_BOTH**
  - Returns an array with **both associative and numerically indexed** elements
  - Each column value is **accessible both by the column name and the numeric index**

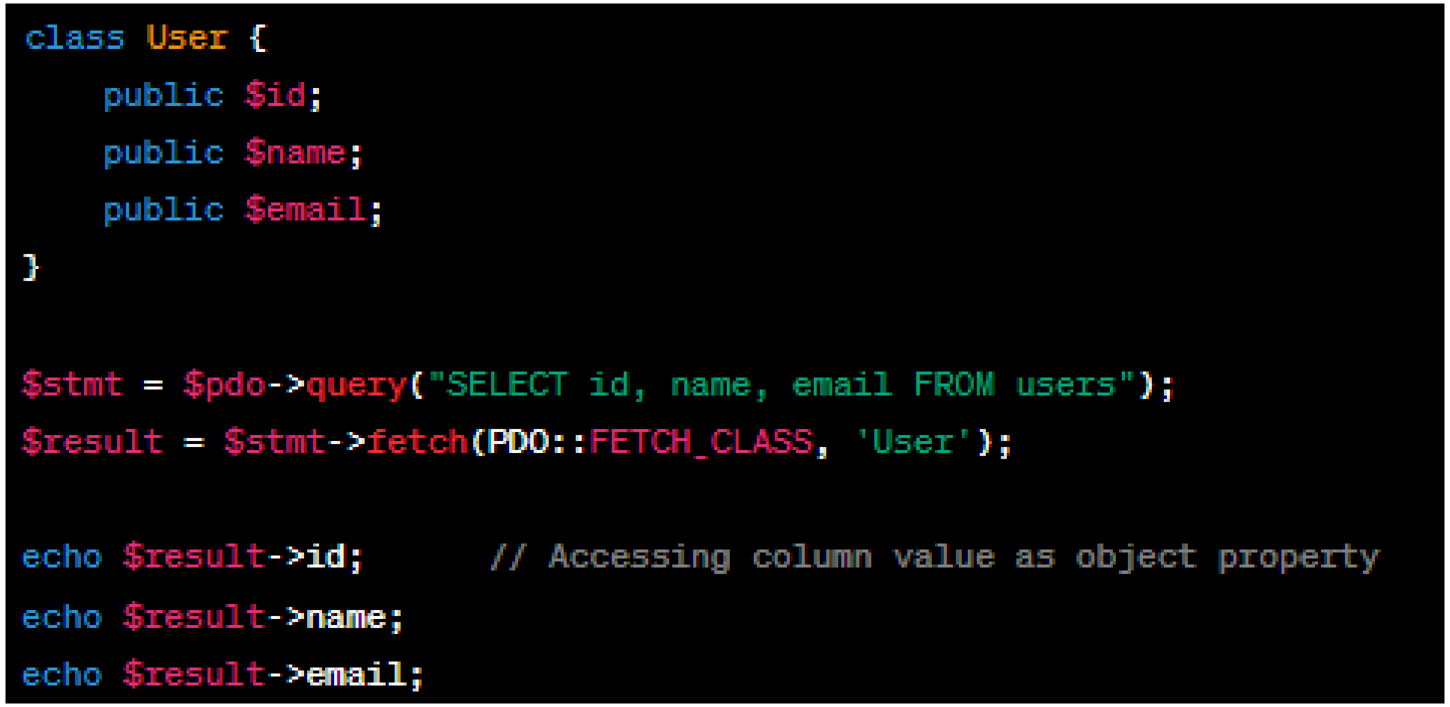


```
$stmt = $pdo->query("SELECT id, name, email FROM users");
$result = $stmt->fetch(PDO::FETCH_BOTH);

echo $result['id'];     // Accessing column value by column name
echo $result[1];        // Accessing column value by numeric index
```

Fetching Styles in PDO cont.

- **PDO::FETCH_CLASS**
  - **Returns the result into a custom class object** that you define
  - The **column values are mapped to the object properties** based on their names



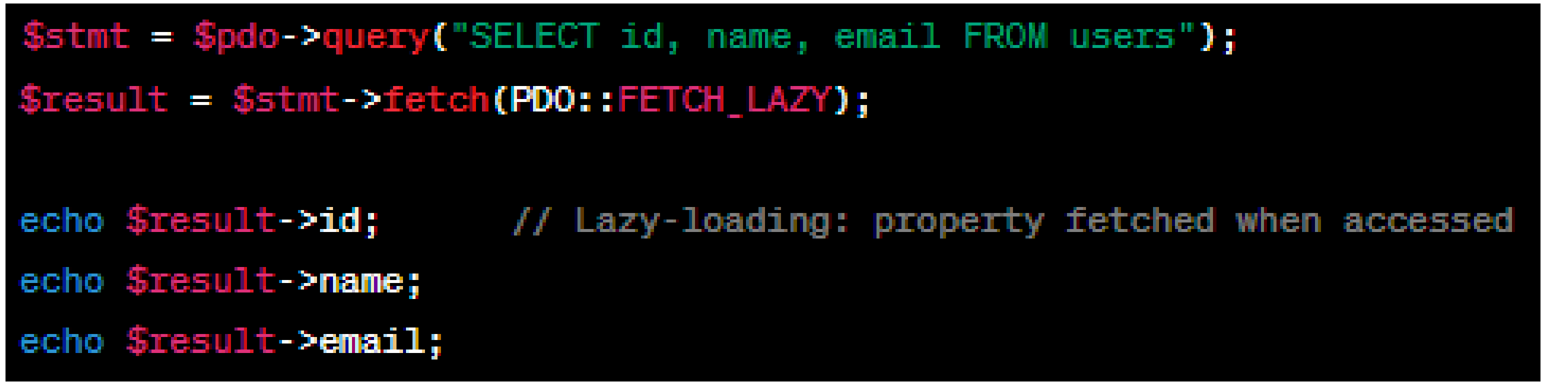
```php
class User {
    public $id;
    public $name;
    public $email;
}

$stmt = $pdo->query("SELECT id, name, email FROM users");
$result = $stmt->fetch(PDO::FETCH_CLASS, 'User');

echo $result->id;      // Accessing column value as object property
echo $result->name;
echo $result->email;
```

Fetching Styles in PDO cont.

- **PDO::FETCH_LAZY**
  - This fetch style creates **a lazy-loading object where the properties are only fetched from the database when accessed**
  - It can be **useful when dealing with large result sets** and you want to **defer the actual data retrieval until it's needed**



```
$stmt = $pdo->query("SELECT id, name, email FROM users");
$result = $stmt->fetch(PDO::FETCH_LAZY);

echo $result->id;       // Lazy-loading: property fetched when accessed
echo $result->name;
echo $result->email;
```

Activity 1

- *Create a table named ‘user’ with ‘username’ and ‘password’ fields within the database called ‘Lecture4DB’.*
- *Insert a user with the username ‘Meena’ and password ‘456’.*
- *Insert another user with the username ‘Seetha’ and password ‘123’.*
- *Display the details of the users from the ‘user’ table.*
- *Update the password of the user ‘Seetha’ to ‘001’.*
- *Delete the user ‘Seetha’ from the ‘user’ table.*

Prepared Statements in PHP

» **Provide a secure and efficient way** to execute database queries with user-supplied data

» A feature used to execute the **same SQL statements repeatedly** with high efficiency

» **Separate the SQL code from the data values**, **preventing** common security vulnerabilities such as **SQL injection attacks**

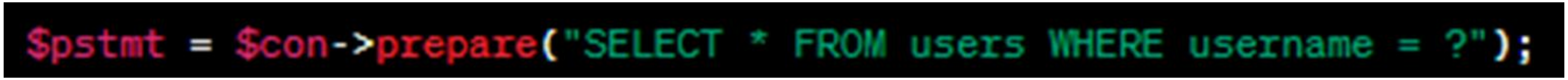
» **Four step** process:

- **Prepare** the statement
- **Bind** parameters
- **Execute** the statement
- **Fetch** results

Prepared Statements in PHP cont.

» In step 1, **use prepare()** method to **prepare the statement**

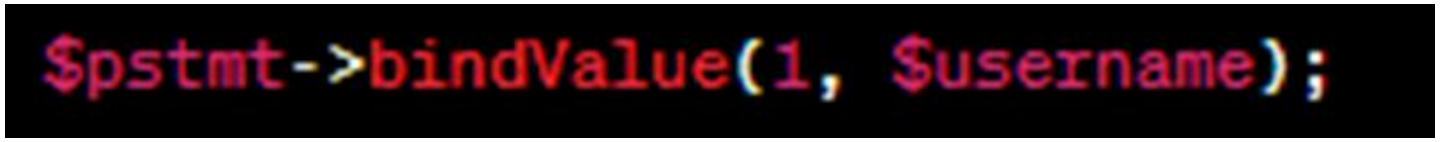
- An SQL statement template is created and sent to the database.
- Certain values are left unspecified, called parameters (labeled "?").
  - Example: “SELECT * FROM users WHERE username = ?”;



```
$pstmt = $con->prepare("SELECT * FROM users WHERE username = ?");
```

» In step 2, **bind values using bindValue()** method

- The database parses, compiles, and performs query optimization on the SQL statement template, and stores the result without executing it.

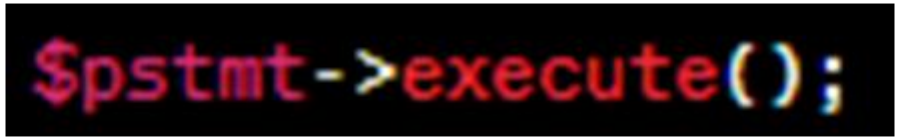


```
$pstmt->bindValue(1, $username);
```

Prepared Statements in PHP cont.

» In step 3, **execute the prepared statement** using the **execute()** method

- After the application binds the values to the parameters, the database executes the statement.
- The application may execute the statement as many times as it wants with different values.



```
$pstmt->execute();
```

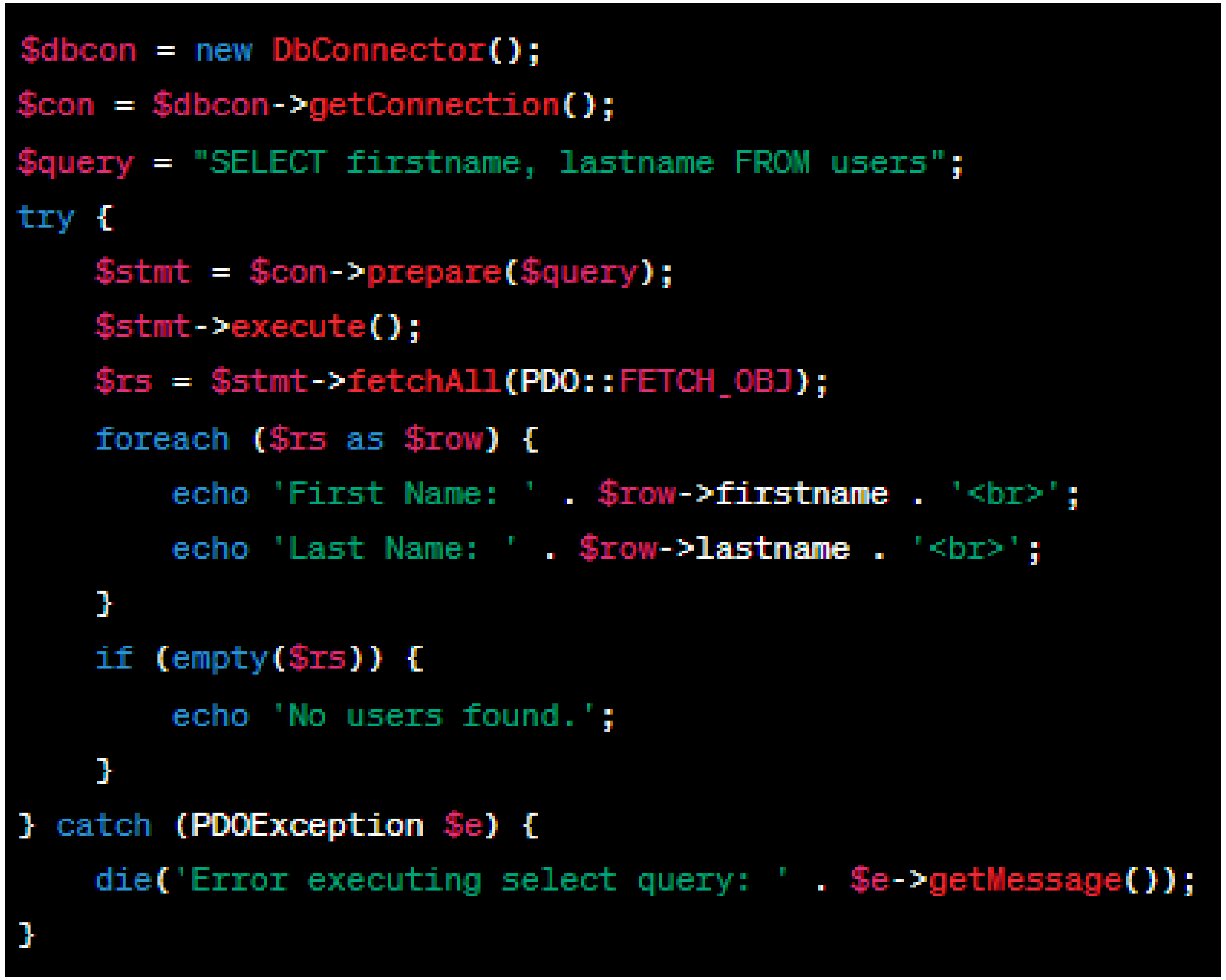
» Finally, **fetch the result set** using methods like **fetch()** or **fetchAll()**

Advantages of Prepared Statements

» Several advantages are there with prepared statements compared to executing SQL statements directly;

- Prepared statements **reduce parsing time** as the preparation on the query is done only once (although the statement is executed multiple times).
- Bound parameters **minimize bandwidth to the server** as you need send only the parameters each time, and not the whole query.
- Prepared statements are very **useful against SQL injections**.
  - (If the original statement template is derived from an external input, SQL injections can occur.)
  - The four variable types allowed for prepared statements are;
    - i - Integer; d - Double; s - String; b - Blob

Prepared Statements in PHP - Example

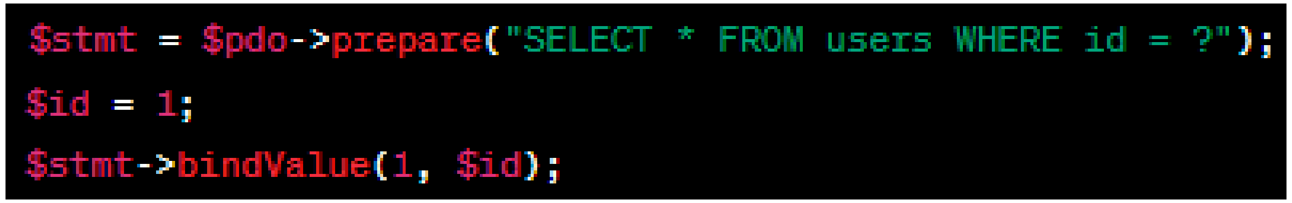


```php
$dbcon = new DbConnector();
$con = $dbcon->getConnection();
$query = "SELECT firstname, lastname FROM users";
try {
    $stmt = $con->prepare($query);
    $stmt->execute();
    $rs = $stmt->fetchAll(PDO::FETCH_OBJ);
    foreach ($rs as $row) {
        echo 'First Name: ' . $row->firstname . '<br>';
        echo 'Last Name: ' . $row->lastname . '<br>';
    }
    if (empty($rs)) {
        echo 'No users found.';
    }
} catch (PDOException $e) {
    die('Error executing select query: ' . $e->getMessage());
}
```

bindValue() vs. bindParam()

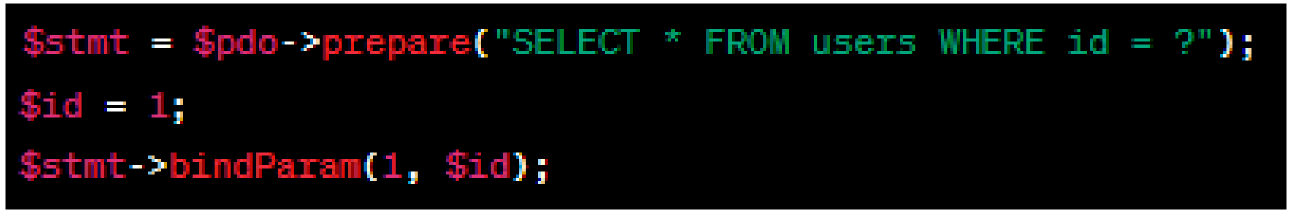
» In PHP's PDO, there are **two methods available to bind values** to prepared statements:

- **bindValue()** – **binds a specific value to a placeholder** in the prepared statement



```
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");
$id = 1;
$stmt->bindValue(1, $id);
```

- **bindParam()** – **binds a reference to a variable to a placeholder** in the prepared statement



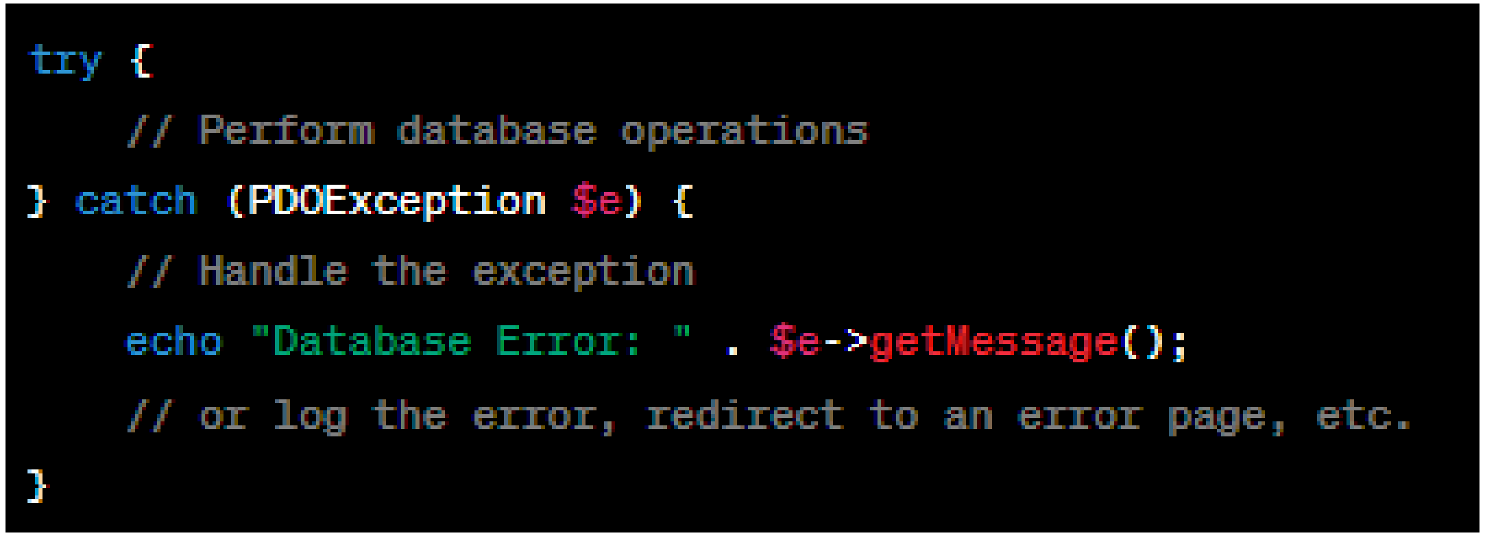
```
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = ?");
$id = 1;
$stmt->bindParam(1, $id);
```

» **Selection depends on** whether you need to **bind a static value or a variable whose value may change before executing the statement**

Handling Database Errors

» Wrap your database operations within **a try-catch block to catch any potential exceptions** that may be thrown by the database driver

» **Try-catch blocks allows you to handle errors** gracefully and **provide meaningful error messages to users**



```
try {
    // Perform database operations
} catch (PDOException $e) {
    // Handle the exception
    echo "Database Error: " . $e->getMessage();
    // or log the error, redirect to an error page, etc.
}
```

Activity 2

- *Create a table named 'course' with 'courseId', 'courseName' and 'noOfCredits' fields within the database 'Lecture4DB' (Note: courseId is the primary key field).*
- *Insert the following record to the course table without using prepared statements.*

courseId	courseName	noOfCredits
CST122-3	DBMS	3

- *Insert the following record to the course table using prepared statements (Note: All the values should be passed as a parameters in the prepared statement).*

courseId	courseName	noOfCredits
CST226-2	WAD	2

- *Display 2 credit courses from the course table using prepared statements (Note: The credit value should be passed as a parameter in the prepared statement).*

Can you remember?

- » Use of Databases in Web Development
- » Establishing Database Connections
- » PDO - PHP Data Objects
- » Steps to use a Database
- » Handling ResultSet with PDO
- » Fetching Styles in PDO
- » Working with Prepared Statements
- » Handling Database Errors