Name- Arya Vats

Supersetid- 6358118

WEEK-2 (MANDATORY HANDS-ON)

Q.1. Advanced SQL Exercises for Online Retail Store

Exercise 1: Ranking and Window Functions

Goal: Use ROW_NUMBER(), RANK(), DENSE_RANK(), OVER(), and PARTITION BY.

Scenario:

Find the top 3 most expensive products in each category using different ranking functions.

Steps:

1. Use ROW_NUMBER() to assign a unique rank within each category.

2. Use RANK() and DENSE_RANK() to compare how ties are handled.

3. Use PARTITION BY Category and ORDER BY Price DESC

CODE:

1) Create your database

```
CREATE DATABASE IF NOT EXISTS OnlineRetail;
-- 2) Tell MySQL to use it
USE OnlineRetail;

-- 3) Create the Products table
CREATE TABLE IF NOT EXISTS Products (
  ProductID    INT         AUTO_INCREMENT PRIMARY KEY,
  ProductName  VARCHAR(100) NOT NULL,
  Category     VARCHAR(50)  NOT NULL,
  Price        DECIMAL(10,2) NOT NULL
);

-- 4) Insert sample data
```

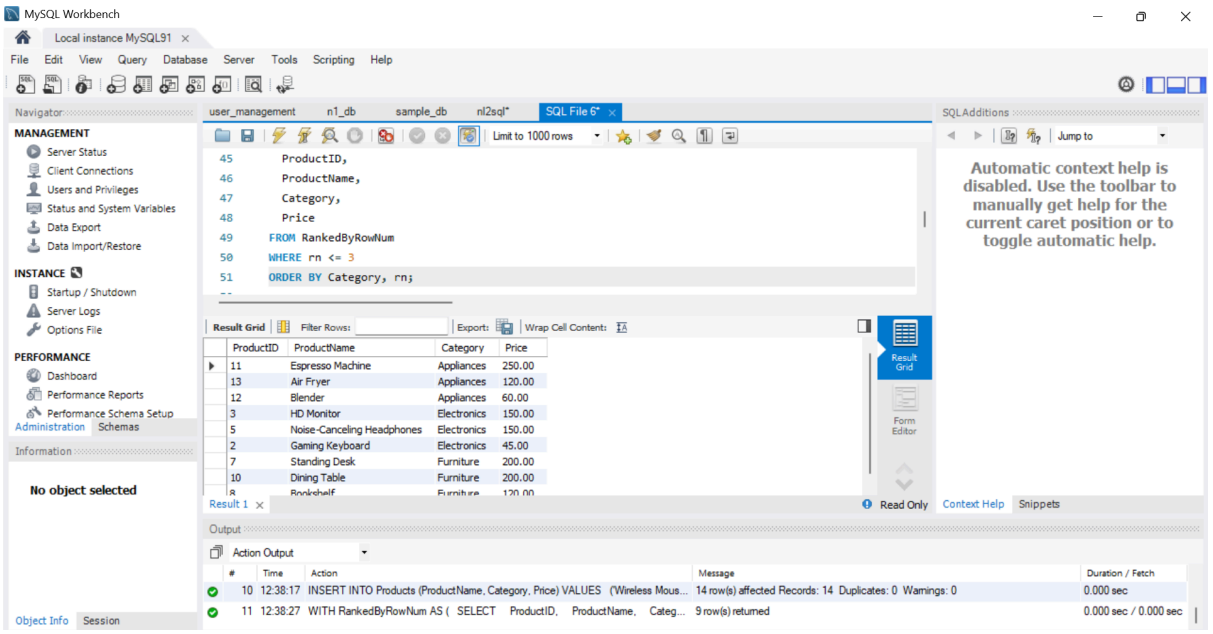
```
INSERT INTO Products (ProductName, Category, Price)
VALUES
  ('Wireless Mouse',    'Electronics', 25.99),
  ('Gaming Keyboard',   'Electronics', 45.00),
  ('HD Monitor',        'Electronics', 150.00),
  ('USB-C Cable',       'Electronics', 10.00),
  ('Noise-Canceling Headphones','Electronics', 150.00),  -- tied
price
  ('Office Chair',      'Furniture',   85.50),
  ('Standing Desk',     'Furniture',   200.00),
  ('Bookshelf',         'Furniture',   120.00),
  ('Lamp',              'Furniture',   45.00),
  ('Dining Table',      'Furniture',   200.00),          -- tied price
  ('Espresso Machine',  'Appliances',  250.00),
  ('Blender',           'Appliances',   60.00),
  ('Air Fryer',         'Appliances',  120.00),
  ('Toaster',           'Appliances',   30.00);

-- 5a) Top 3 per category using ROW_NUMBER()
WITH RankedByRowNum AS (
  SELECT
    ProductID,
    ProductName,
    Category,
    Price,
    ROW_NUMBER() OVER (
      PARTITION BY Category
      ORDER BY Price DESC
    ) AS rn
  FROM Products
)
SELECT
  ProductID,
```

```
  ProductName,
  Category,
  Price
FROM RankedByRowNum
WHERE rn <= 3
ORDER BY Category, rn;


-- 5b) Top “3” per category using RANK()  (includes ties, may
return >3 rows)
WITH RankedByRank AS (
  SELECT
    ProductID,
    ProductName,
    Category,
    Price,
    RANK() OVER (
      PARTITION BY Category
      ORDER BY Price DESC
    ) AS rnk
  FROM Products
)
SELECT
  ProductID,
  ProductName,
  Category,
  Price
FROM RankedByRank
WHERE rnk <= 3
ORDER BY Category, rnk;
```

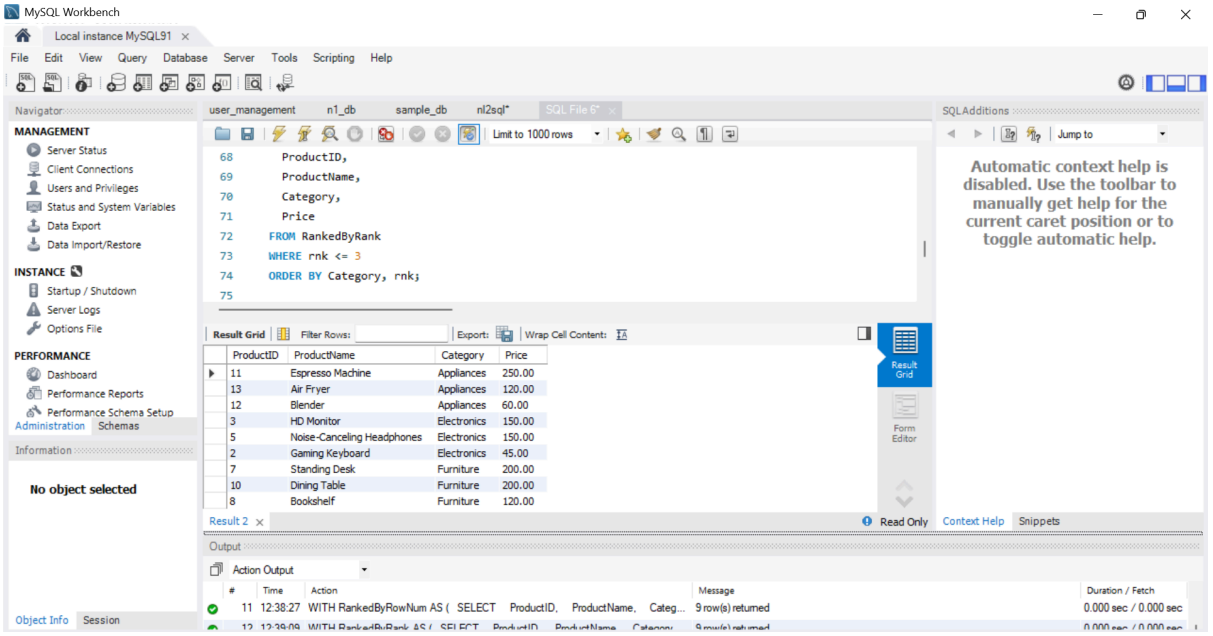
```
-- 5c) Top “3” per category using DENSE_RANK() (includes ties, no gaps in ranking)
WITH RankedByDenseRank AS (
  SELECT
    ProductID,
    ProductName,
    Category,
    Price,
    DENSE_RANK() OVER (
      PARTITION BY Category
      ORDER BY Price DESC
    ) AS drnk
  FROM Products
)
SELECT
  ProductID,
  ProductName,
  Category,
  Price
FROM RankedByDenseRank
WHERE drnk <= 3
ORDER BY Category, drnk;
```

OUTPUT:-

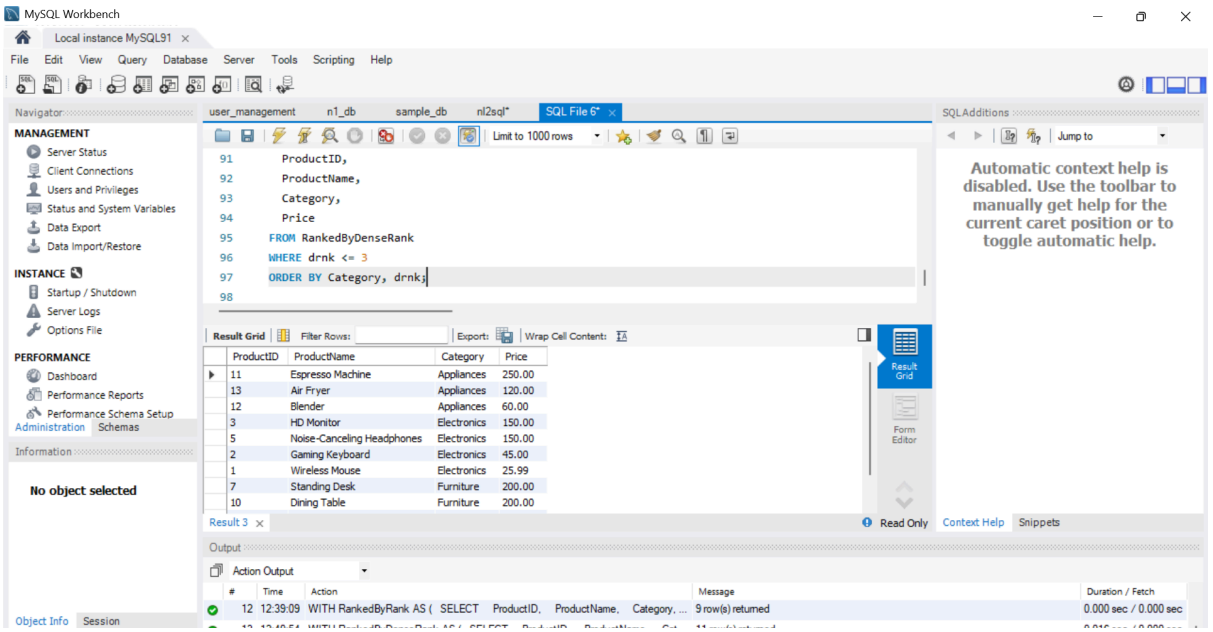
1. Top 3 per category using ROW_NUMBER()



2.Top “3” per category using RANK() (includes ties, may return >3 rows)



3.Top “3” per category using DENSE_RANK() (includes ties, no gaps in ranking)



Q.2. Exercise 1: Create a Stored Procedure

Goal: Create a stored procedure to retrieve employee details by department.

Steps:

1. Define the stored procedure with a parameter for DepartmentID.
2. Write the SQL query to select employee details based on the DepartmentID.
3. Create a stored procedure named `sp_InsertEmployee` with the following code:

```
CREATE PROCEDURE sp_InsertEmployee
@FirstName VARCHAR(50),
@LastName VARCHAR(50),
@DepartmentID INT,
@Salary DECIMAL(10,2),
@JoinDate DATE
AS
BEGIN
INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary, JoinDate)
VALUES (@FirstName, @LastName, @DepartmentID, @Salary, @JoinDate);
END;
```

CODE:-

```
-- 1) Create and switch to the database
CREATE DATABASE IF NOT EXISTS EmpManagementSystem;
USE EmpManagementSystem;

-- 2) Create Departments table
CREATE TABLE IF NOT EXISTS Departments (
  DepartmentID   INT PRIMARY KEY,
  DepartmentName VARCHAR(100) NOT NULL
```

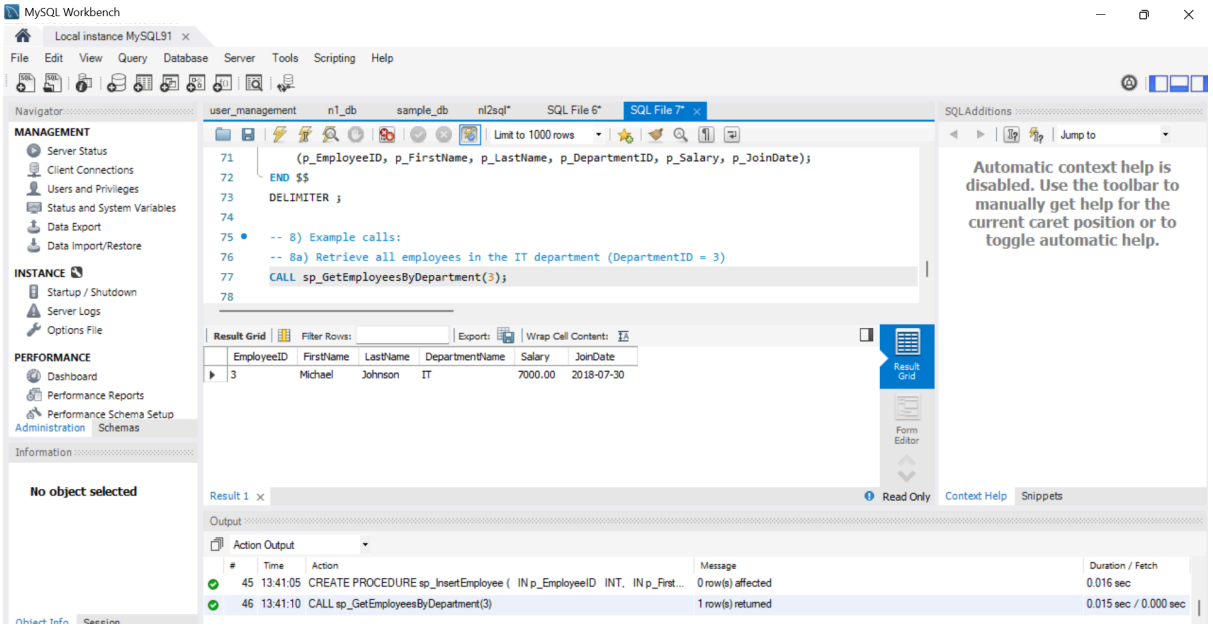
```
);

-- 3) Create Employees table
CREATE TABLE IF NOT EXISTS Employees (
  EmployeeID   INT PRIMARY KEY,
  FirstName    VARCHAR(50)  NOT NULL,
  LastName     VARCHAR(50)  NOT NULL,
  DepartmentID INT,
  Salary       DECIMAL(10,2) NOT NULL,
  JoinDate     DATE         NOT NULL,
  FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
);

-- 4) Seed Departments via REPLACE (no warnings)
REPLACE INTO Departments (DepartmentID, DepartmentName)
VALUES
  (1, 'HR'),
  (2, 'Finance'),
  (3, 'IT'),
  (4, 'Marketing');

-- 5) Seed Employees via REPLACE (no warnings)
REPLACE INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary, JoinDate) VALUES
  (1, 'John',    'Doe',      1, 5000.00, '2020-01-15'),
  (2, 'Jane',    'Smith',    2, 6000.00, '2019-03-22'),
  (3, 'Michael', 'Johnson', 3, 7000.00, '2018-07-30'),
  (4, 'Emily',   'Davis',   4, 5500.00, '2021-11-05');

-- 6) Stored procedure: get employees by department
DELIMITER $$
CREATE PROCEDURE sp_GetEmployeesByDepartment (
  IN p_DepartmentID INT
)
BEGIN
  SELECT
```

```
    e.EmployeeID,
    e.FirstName,
    e.LastName,
    d.DepartmentName,
    e.Salary,
    e.JoinDate
  FROM Employees AS e
  JOIN Departments AS d
    ON e.DepartmentID = d.DepartmentID
  WHERE e.DepartmentID = p_DepartmentID
  ORDER BY e.JoinDate;
END $$
DELIMITER ;

-- 7) Stored procedure: insert a new employee
DELIMITER $$
CREATE PROCEDURE sp_InsertEmployee (
  IN p_EmployeeID   INT,
  IN p_FirstName    VARCHAR(50),
  IN p_LastName     VARCHAR(50),
  IN p_DepartmentID INT,
  IN p_Salary       DECIMAL(10,2),
  IN p_JoinDate     DATE
)
BEGIN
  INSERT INTO Employees
    (EmployeeID, FirstName, LastName, DepartmentID, Salary, JoinDate)
  VALUES
    (p_EmployeeID, p_FirstName, p_LastName, p_DepartmentID,
p_Salary, p_JoinDate);
END $$
DELIMITER ;

-- 8) Example calls:
-- 8a) Retrieve all employees in the IT department (DepartmentID = 3)
CALL sp_GetEmployeesByDepartment(3);
```

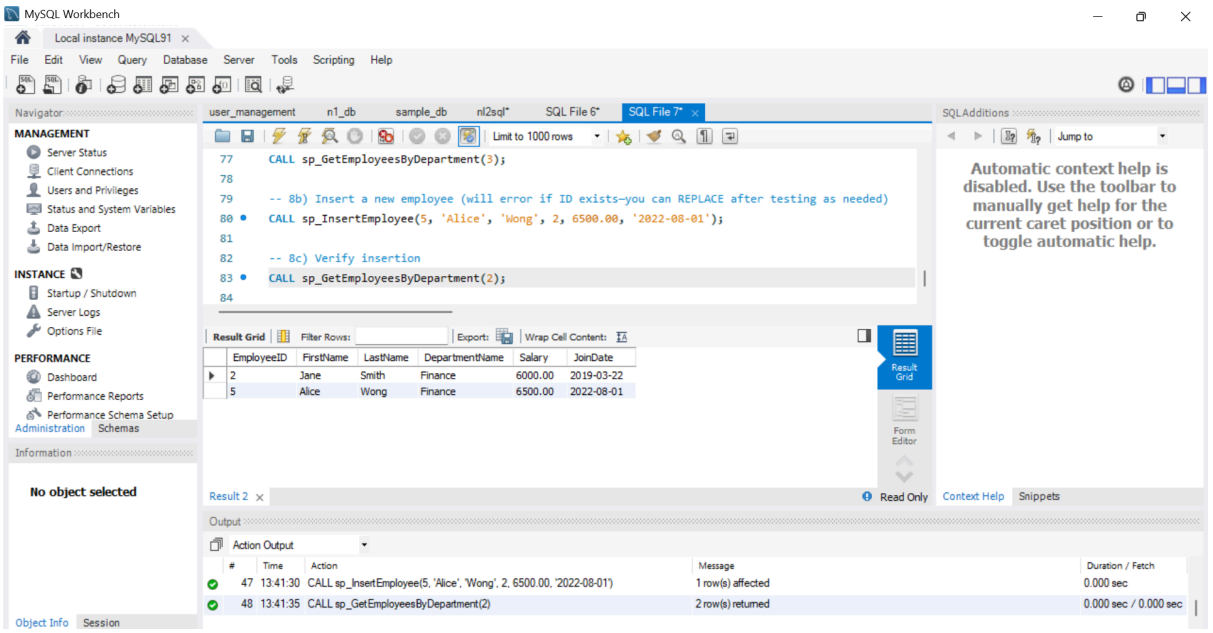
-- 8b) Insert a new employee (will error if ID exists—you can REPLACE after testing as needed)
CALL sp_InsertEmployee(5, 'Alice', 'Wong', 2, 6500.00, '2022-08-01');

-- 8c) Verify insertion
CALL sp_GetEmployeesByDepartment(2);

OUTPUT:-1. Retrieve all employees in the IT department (DepartmentID= 3) CALL sp_GetEmployeesByDepartment(3);



2.Verify insertion
CALL sp_GetEmployeesByDepartment(2);



Q.3. Exercise 5: Return Data from a Stored Procedure
Goal: Create a stored procedure that returns the total number of employees in a
department.
Steps:
1. Define the stored procedure with a parameter for DepartmentID.
2. Write the SQL query to count the number of employees in the specified department.
3. Save the stored procedure by executing the Stored procedure conten

CODE:-

```
-- 1) Create and switch to the database
CREATE DATABASE IF NOT EXISTS EmployeeManagementSystem;
USE EmployeeManagementSystem;

-- 2) Create Departments table
CREATE TABLE IF NOT EXISTS Departments (
  DepartmentID   INT PRIMARY KEY,
  DepartmentName VARCHAR(100) NOT NULL
);

-- 3) Create Employees table
CREATE TABLE IF NOT EXISTS Employees (
  EmployeeID   INT PRIMARY KEY,
  FirstName    VARCHAR(50)  NOT NULL,
  LastName     VARCHAR(50)  NOT NULL,
  DepartmentID INT,
  Salary       DECIMAL(10,2) NOT NULL,
  JoinDate     DATE         NOT NULL,
  FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
);

-- 4) Seed Departments
```

```
REPLACE INTO Departments (DepartmentID, DepartmentName)
VALUES
  (1, 'HR'),
  (2, 'Finance'),
  (3, 'IT'),
  (4, 'Marketing');

-- 5) Seed Employees
REPLACE INTO Employees (EmployeeID, FirstName, LastName,
DepartmentID, Salary, JoinDate) VALUES
  (1, 'John',    'Doe',     1, 5000.00, '2020-01-15'),
  (2, 'Jane',    'Smith',   2, 6000.00, '2019-03-22'),
  (3, 'Michael', 'Johnson', 3, 7000.00, '2018-07-30'),
  (4, 'Emily',   'Davis',   4, 5500.00, '2021-11-05');

-- 6) Stored procedure: get employees by department
DELIMITER $$
CREATE PROCEDURE sp_GetEmployeesByDepartment (
  IN p_DepartmentID INT
)
BEGIN
  SELECT
    e.EmployeeID,
    e.FirstName,
    e.LastName,
    d.DepartmentName,
    e.Salary,
    e.JoinDate
  FROM Employees AS e
  JOIN Departments AS d
    ON e.DepartmentID = d.DepartmentID
  WHERE e.DepartmentID = p_DepartmentID
  ORDER BY e.JoinDate;
END $$
DELIMITER ;

-- 7) Stored procedure: insert a new employee
```

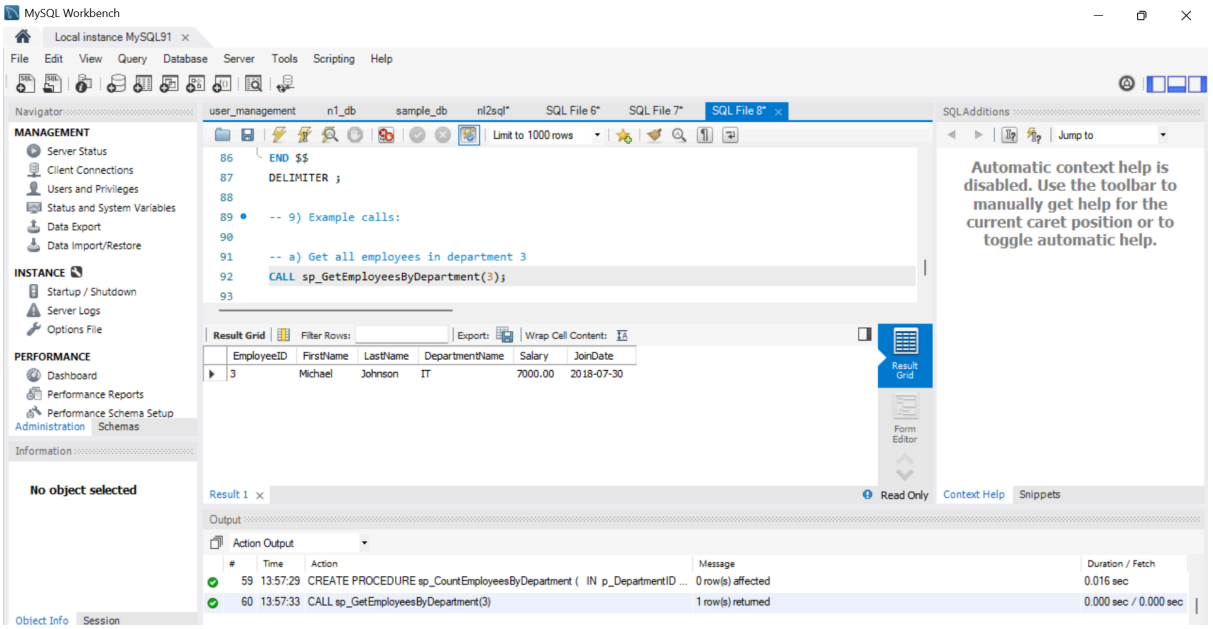
```
DELIMITER $$
CREATE PROCEDURE sp_InsertEmployee (
  IN p_EmployeeID   INT,
  IN p_FirstName    VARCHAR(50),
  IN p_LastName     VARCHAR(50),
  IN p_DepartmentID INT,
  IN p_Salary       DECIMAL(10,2),
  IN p_JoinDate     DATE
)
BEGIN
  INSERT INTO Employees
    (EmployeeID, FirstName, LastName, DepartmentID, Salary,
JoinDate)
  VALUES
    (p_EmployeeID, p_FirstName, p_LastName, p_DepartmentID,
p_Salary, p_JoinDate);
END $$
DELIMITER ;

-- 8) Stored procedure: count employees in a department (Exercise
5)
DELIMITER $$
CREATE PROCEDURE sp_CountEmployeesByDepartment (
  IN  p_DepartmentID INT,
  OUT p_TotalCount   INT
)
BEGIN
  SELECT COUNT(*)
    INTO p_TotalCount
  FROM Employees
  WHERE DepartmentID = p_DepartmentID;
END $$
DELIMITER ;

-- 9) Example calls:

-- a) Get all employees in department 3
```

CALL sp_GetEmployeesByDepartment(3);

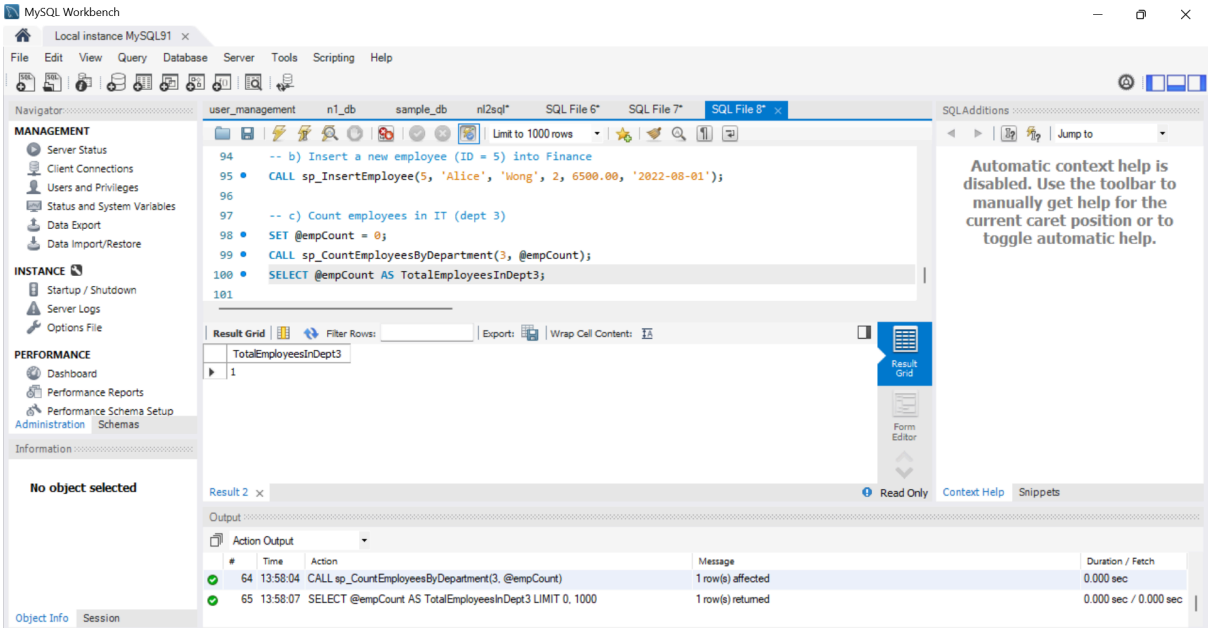
-- b) Insert a new employee (ID = 5) into Finance
CALL sp_InsertEmployee(5, 'Alice', 'Wong', 2, 6500.00, '2022-08-01');

-- c) Count employees in IT (dept 3)
SET @empCount = 0;
CALL sp_CountEmployeesByDepartment(3, @empCount);
SELECT @empCount AS TotalEmployeesInDept3;

OUTPUT:- Get all employees in department 3
CALL sp_GetEmployeesByDepartment(3);



Count employees in IT (dept 3)



NUNIT:-

Objectives

- Explain the meaning of Unit testing and its difference on comparison with Functional testing
    - Smallest unit to test mocking dependencies
- List various types of testing
    - Unit testing, Functional testing, Automated testing, Performance testing
- Understand the benefit of automated testing
- Explain what is loosly coupled & testable design
    - Write code that is NOT dependent on the class for data.
- Write your first testing program to validate a calculator addition operation
    - TestFixture, Test
- Understand the need of [SetUp], [TearDown] & [Ignore] attributes.
- Explain the benefit of writing parameterised test cases.
    - TestCase

TestFixture & Test

Please download the application available here. This will be used to write Unit test cases

Follow the steps listed below to write the NUnit test cases for the application.

· Create a Unit test project(.Net Framework) in the solution provided.

· Add the CalcLibrary project as reference

· Create a class “CalculatorTests” to write all the test cases for the methods in the solution

· Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup activities respectively

· Create a Test method to check the addition functionality

· Use the ‘TestCase’ attribute to send the inputs and the expected result

· Use Assert.That to check the actual and expected result match

1. Project Overview

Solution Name: CalculatorSolution
Projects:

- `CalcLibrary`: A class library that contains the `Calculator` class.

- `CalcLibrary.Tests`: An NUnit test project to validate the `Add()` method of the `Calculator`.

2. Calculator Class Code (`CalcLibrary`)

```
namespace CalcLibrary
{
    public class Calculator
    {
        // This method adds two integers
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

3. Unit Test Code (`CalcLibrary.Tests`)

```
using NUnit.Framework;
using CalcLibrary;  // Accessing the Calculator class

namespace CalcLibrary.Tests
{
    [TestFixture]  // This class contains unit tests
    public class CalculatorTests
    {
        private Calculator _calculator;

        [SetUp]  // Runs before each test
        public void SetUp()
        {
            _calculator = new Calculator();
        }

        [TearDown]  // Runs after each test
        public void TearDown()
        {
```

```
        _calculator = null;
    }

    [TestCase(1, 2, 3)]
    [TestCase(-1, 5, 4)]
    [TestCase(0, 0, 0)]
    [TestCase(100, 200, 300)]
    public void Add_ShouldReturnExpected(int a, int b, int expected)
    {
        var result = _calculator.Add(a, b);
        Assert.That(result, Is.EqualTo(expected), $"Expected {expected},
but got {result}");
    }
  }
}
```

OUTPUT:-

```
Test Run Successful.
Total tests: 4
Passed: 4
Failed: 0
Skipped: 0
```

Objectives:

· Understand how Mocking can enhance Test-Driven Development (TDD)

- Mocking, Isolation, Test doubles, Mock Vs Fake Vs Stub, Key advantages of TDD

· Explain the meaning of Mocking in Unit Testing and why use mocks in Unit Testing

- Mocking and Isolation in Unit Testing, Isolating dependencies in Tests using Mocks and Stubs

· Understand the basics of DI (Dependency Injection) and how dependency injection helps unit testing in applications

- Dependency Injection, Constructor Injection, Method Injection

· Demonstrate on how to create a testable code with Moq.

- Testable code

· Demonstrate on how to create a mock object that access database for unit tests

- Mock database for Unit Tests

· Demonstrate on mock object that access the file system for unit tests

- Mock files for Unit Tests

1. Write Testable Code with Moq

Scenario

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with a mail server communication in which your application tries to send mail to its users upon every transaction. Your role is to write unit testing the module that contains send mail functionality. You wanted to perform testing the module without sending any email.

After investigating the problem scenario, you found a solution and that is creating **mock** objects of these external dependencies in the unit testing project so that you can achieve speedier test execution and loose coupling of code.

Note: Duration to complete this exercise is **30 min**.

Task1

In this task, you will create a class library that will be used for unit testing.

· Create a **Class Library (Language C#)** project using Visual Studio IDE, and name it as **CustomerCommLib.**

· Rename the default **Class1** class name as **MailSender.**

· Include the following namespaces with 'using' directive.

- **System.Net**
- **System.Net.Mail**

· Define an interface as follow.

```
public interface IMailSender
{
    bool SendMail(string toAddress, string message);
}
```

· And provide implementation of **IMailSender** in the **MailSender** class as seen below.

```
namespace CustomerCommLib
{
    public class MailSender:IMailSender
    {
```

```
        public bool SendMail(string toAddress, string message)
        {
MailMessage mail = new MailMessage();
            SmtpClient SmtpServer = new SmtpClient("smtp.gmail.com");

            mail.From = new MailAddress("your_email_address@gmail.com");
            mail.To.Add(toAddress);
            mail.Subject = "Test Mail";
            mail.Body = message;

            SmtpServer.Port = 587;
                                SmtpServer.Credentials = new NetworkCredential("username",
                "password");
            SmtpServer.EnableSsl = true;

            SmtpServer.Send(mail);

        }
        }
}
```

The above class can't be unit testing since the code access the STMP mail server.

- Create another class called **CustomeComm** which is the **class under test** in the given scenario.

```
namespace CustomerCommLib
{
        public class CustomerComm
        {
```

```
IMailSender _mailSender;


public CustomerComm(IMailSender mailSender)

{

_mailSender=mailSender;

}


public bool SendMailToCustomer()

{

//Actual logic goes here

//define message and mail address


                                    _mailSender.SendMail(cust123@abc.com,”Some Message”);


                                    return true;

}

        }

}
```

In the above code we **injected the dependency** (IMailSender) through **constructor** of **CustomerComm** class so that we can **pass the mock object** of the dependency wherever it is necessary.

We have successfully created a class that's written in such a way that we can run a unit test against it and an exception won't be thrown. We achieve this by mocking the call to IMailSender.SendMail() and adding a mocked return value of true to it.

- Finally **build** your project and be ready for the unit testing with NUnit and Moq.

Task2

In this task, you will create unit test project which make use of NUnit framework and Moq.

· Create a new class library project called **CustomerComm.Tests** and add the following external dependencies to it using **NuGet Package Manager.**

  - NUnit
  - NUnit Test Adapter
  - Moq

· Add the references of assemblies as appropriate including **CustomerCommLib.**

· Write unit test code and **mock** the **MailSender (IMailSender)** class.

· Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.

· **Configure** the mock object in such away that **SendMail()** method will accept any two string arguments and always return true when **SendMailToCustomer()** gets invoked.

· Finally **assert** the return value to “true”.

2. Mock file object for Unit Tests

Scenario

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with the file system and it searches for files and retrieves files under the specified path. In the existing system, **Directory.GetFiles()** method has been used. You found that it's not good idea to use Directory.GetFiles from the System.IO being its **static** and **unable to unit tes**t such methods.

After investigating the problem scenario, you found a solution and that is refactoring the code. Instead of using directly the static method Directory.GetFiles, you decided to create your own implementation to the method so that be able to **mock** files in the Unit Tests.

Note: Duration to complete this exercise is **30 min**.

Task1

· Create a **Class Library (Language C#)** project using Visual Studio IDE, and name it as **MagicFilesLib.**

· Rename the default **Class1** class name as **DirectoryExplorer** and include the following code snippet into it.

· Include the following namespaces with 'using' directive.

- **System.Collections.Generic**
- **System.IO**

· Define an interface as follow.

```
public interface IDirectoryExplorer
{
    ICollection<string> GetFiles(string path);
```

```
    }
```

- And provide implementation of **IDirectoryExplorer** in the **DirectoryExplorer** class as seen below.

```
namespace MagicFilesLib
{
    public class DirectoryExplorer: IDirectoryExplorer
    {
        public ICollection<string> GetFiles(string path)
        {
            string[] files = Directory.GetFiles(path);
            return files;
        }
    }
}
```

Finally **build** your project and be ready for the unit testing with NUnit and Moq.

Task2

- Create a new class library project called **DirectoryExplorer.Tests** and add the following external dependencies to it using **NuGet Package Manager.**

    - NUnit
    - NUnit Test Adapter
    - Moq

· Add the references of assemblies as appropriate including **MagicFilesLib.**

· Write unit test code and **mock the DirectoryExplorer (IDirectoryExplorer),** which is the class under test, with some hard coded file names.

· Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.

· Add the following declarations in the test class.

```
private readonly string _file1 = "file.txt";

private readonly string _file2 = "file2.txt";
```

· In the test method, **assert** the following so that,

the collection is not null

the collection count is equal to 2

the collection contains _file1

3. Mock database for Unit Tests

Scenario

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with a network database in which your application stores the record or certain players. It involves storing and retrieval of player details. Your role is to write unit testing the player module which involves an external dependency. You can't proceed with unit testing.

After investigating the problem scenario, you found a solution and that is creating **mock** objects of these external dependencies in the unit testing project so that you can achieve speedier test execution and loose coupling of code.

Note: Duration to complete this exercise is **60 min**.

Task1

In this task, you will create a class library that will be used for unit testing.

· Create a **Class Library (Language C#)** project using Visual Studio IDE, and name it as **PlayersManagerLib.**

· Rename the default **Class1** class name as **PlayerManager.**

· Include the following namespaces with 'using' directive.

- **System.Data**
- **System.Data.SqlClient**

· Define an interface as follow.

```
public interface IPlayerMapper
{
    bool IsPlayerNameExistsInDb(string name);
```

```
        Void AddNewPlayerIntoDb(string name);
}
```

· And provide implementation of **IPlayerMapper** in the **PlayerMapper** class as seen below.

```
namespace PlayersManagerLib
{
    public class PlayerMapper: IPlayerMapper
    {
        private readonly string _connectionString =
            "Data Source=(local);Initial Catalog=GameDB;Integrated Security=True";

        public bool IsPlayerNameExistsInDb(string name)
        {
            using(SqlConnection connection = new SqlConnection(_connectionString))
            {
                connection.Open();

                using(SqlCommand command = connection.CreateCommand())
                {
                        command.CommandText = "SELECT count(*) FROM Player WHERE
'Name' = @name";

                    command.Parameters.AddWithValue("@name", name);
```

```
                // Get the number of player with this name
                var existingPlayersCount = (int) command.ExecuteScalar();

                // Result is 0, if no player exists, or 1, if a player already exists
                return existingPlayersCount > 0;
            }
        }
    }

    public void AddNewPlayerIntoDb(string name)
    {
        using(SqlConnection connection = new SqlConnection(_connectionString))
        {
            connection.Open();

            using(SqlCommand command = connection.CreateCommand())
            {
                command.CommandText = "INSERT INTO Player ([Name]) VALUES
(@name)";

                command.Parameters.AddWithValue("@name", name);

                command.ExecuteNonQuery();
            }
        }
    }
```

```
        }
```

The above class can’t be unit testing since the code access the database.

· Create another class called **Player** and add the following codes.

```
public class Player
{
    public string Name { get; private set; }
    public int Age { get; private set; }
    public string Country { get; private set; }
    public int NoOfMatches {get; private set;}

    public Player(string name, int age, string country, int noOfMatches)
    {
        Name = name;
        Age=age;
        Country= country;
        NoOfMatches = noOfMatches;
    }

        public static Player RegisterNewPlayer(string name, IPlayerMapper playerMapper =
null)
    {
        // If a PlayerMapper was not passed in, use a real one.
        // This allows us to pass in a "mock" PlayerMapper (for testing),
        // but use a real PlayerMapper, when running the program.
        if(playerMapper == null)
        {
```

```
            playerMapper = new PlayerMapper();
        }

        if(string.IsNullOrWhiteSpace(name))
        {
            throw new ArgumentException("Player name can’t be empty.");
        }

        // Throw an exception if there is already a player with this name in the database.
        if(playerMapper.IsPlayerNameExistsInDb (name))
        {
            throw new ArgumentException("Player name already exists.");
        }

        // Add the player to the database.
        playerMapper. AddNewPlayerIntoDb (name);

        return new Player(name, 23, “India”,30);
    }
}
```

Finally **build** your project and be ready for the unit testing with NUnit and Moq.

Task2

In this task, you will create unit test project which make use of NUnit framework and Moq.

· Create a new class library project called **PlayerManager.Tests** and add the following external dependencies to it using **NuGet Package Manager.**

  - NUnit
  - NUnit Test Adapter
  - Moq

· Add the references of assemblies as appropriate including **PlayersManagerLib.**

· Write unit test code and **mock** the **PlayerMapper (IPlayerMapper)** class.

· Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.

· Use **ExpectedException** attribute to specify that the execution of a test will throw an exception.

· When the **RegisterNewPlayer** function calls **IsPlayerNameExistsInDb**, you need to make sure that the mock object to return **"false".**

· In the test method, **assert** various player attributes.

1. CustomerCommLib → CustomerComm.Tests

```
using Moq;

using NUnit.Framework;

using CustomerCommLib;


namespace CustomerComm.Tests

{
```

```
[TestFixture]
public class CustomerCommTests
{
    private Mock<IMailSender> _mailSenderMock;
    private CustomerComm _customerComm;

    [OneTimeSetUp]
    public void Init()
    {
        // Arrange: mock IMailSender to always return true
        _mailSenderMock = new Mock<IMailSender>();
        _mailSenderMock
            .Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()))
            .Returns(true);

        // Inject mock into CustomerComm
        _customerComm = new CustomerComm(_mailSenderMock.Object);
    }

    [TestCase]
    public void SendMailToCustomer_ReturnsTrue_WhenMailSenderSucceeds()
    {
        // Act
        var result = _customerComm.SendMailToCustomer();

        // Assert
```

```
        Assert.IsTrue(result);
        _mailSenderMock.Verify(
            m => m.SendMail(It.IsAny<string>(), It.IsAny<string>()),
            Times.Once);
    }
  }
}
```

OUTPUT:-

```
NUnit Adapter 3.17.0.0: Test execution starting
Running all tests in CustomerComm.Tests.dll

Test Run Summary
  Overall result: Passed
  Test Count: 1, Passed: 1, Failed: 0, Skipped: 0
  Start time: 2025-06-29 17:12:05Z
  End time:   2025-06-29 17:12:06Z
  Duration:   0.45 seconds
```

2. MagicFilesLib → DirectoryExplorer.Tests

```
using System.Collections.Generic;
using Moq;
using NUnit.Framework;
using MagicFilesLib;

namespace DirectoryExplorer.Tests
{
  [TestFixture]
```

```
public class DirectoryExplorerTests
{
    private Mock<IDirectoryExplorer> _dirExplorerMock;
    private const string File1 = "file.txt";
    private const string File2 = "file2.txt";

    [OneTimeSetUp]
    public void Init()
    {
        // Arrange: mock GetFiles to return two hard-coded filenames
        _dirExplorerMock = new Mock<IDirectoryExplorer>();
        _dirExplorerMock
            .Setup(d => d.GetFiles(It.IsAny<string>()))
            .Returns(new List<string> { File1, File2 });
    }

    [TestCase]
    public void GetFiles_ReturnsNonNullCollection_WithExpectedFiles()
    {
        // Act
        var files = _dirExplorerMock.Object.GetFiles("anyPath");

        // Assert
        Assert.IsNotNull(files, "Files collection should not be null.");
        Assert.AreEqual(2, files.Count, "Should return exactly two files.");
        CollectionAssert.Contains(files, File1, $"Should contain {File1}");
```

```
            CollectionAssert.Contains(files, File2, $"Should contain {File2}");
        }
    }
}
```

OUTPUT:-

```
NUnit Adapter 3.17.0.0: Test execution starting
Running all tests in DirectoryExplorer.Tests.dll

Test Run Summary
  Overall result: Passed
  Test Count: 1, Passed: 1, Failed: 0, Skipped: 0
  Start time: 2025-06-29 17:12:20Z
  End time:   2025-06-29 17:12:21Z
  Duration:   0.38 seconds
```

3. PlayersManagerLib → PlayerManager.Tests

CODE:-

```
using System;
using Moq;
using NUnit.Framework;
using PlayersManagerLib;

namespace PlayerManager.Tests
{
    [TestFixture]
    public class PlayerTests
```

```
{
    private Mock<IPlayerMapper> _playerMapperMock;

    [OneTimeSetUp]
    public void Init()
    {
        _playerMapperMock = new Mock<IPlayerMapper>();
    }

    [TestCase]
    public void RegisterNewPlayer_ReturnsPlayer_WithCorrectDefaults()
    {
        // Arrange
        _playerMapperMock
            .Setup(m => m.IsPlayerNameExistsInDb(It.IsAny<string>()))
            .Returns(false);
        _playerMapperMock
            .Setup(m => m.AddNewPlayerIntoDb(It.IsAny<string>()));

        // Act
        var player = Player.RegisterNewPlayer("Alice", _playerMapperMock.Object);

        // Assert
        Assert.AreEqual("Alice", player.Name);
        Assert.AreEqual(23, player.Age);
        Assert.AreEqual("India", player.Country);
```

```
        Assert.AreEqual(30, player.NoOfMatches);


                    _playerMapperMock.Verify(m  =>  m.IsPlayerNameExistsInDb("Alice"),
Times.Once);
                        _playerMapperMock.Verify(m  =>  m.AddNewPlayerIntoDb("Alice"),
Times.Once);
    }


    [TestCase]
    public void RegisterNewPlayer_Throws_WhenNameIsEmpty()
    {
        Assert.Throws<ArgumentException>(
            () => Player.RegisterNewPlayer("", _playerMapperMock.Object),
            "Player name can't be empty.");
    }


    [TestCase]
    public void RegisterNewPlayer_Throws_WhenNameAlreadyExists()
    {
        // Arrange
        _playerMapperMock
            .Setup(m => m.IsPlayerNameExistsInDb(It.IsAny<string>()))
            .Returns(true);


        // Act & Assert
        Assert.Throws<ArgumentException>(
            () => Player.RegisterNewPlayer("Bob", _playerMapperMock.Object),
```

```
            "Player name already exists.");
        }
    }
}
```

OUTPUT:-

```
NUnit Adapter 3.17.0.0: Test execution starting
Running all tests in PlayerManager.Tests.dll

Test Run Summary
  Overall result: Passed
  Test Count: 3, Passed: 3, Failed: 0, Skipped: 0
  Start time: 2025-06-29 17:12:35Z
  End time:   2025-06-29 17:12:36Z
  Duration:   0.52 seconds
```