MINOR CS-25: Practical Based on CS-22, CS – 23, CS-24

Prepared By : Lathiya_Harshal

- Q.1 (Java) – 20 Marks
- Q.2 (C#) – 20 Marks
- Q.3 (Shell Scripting) – 10 Marks

🔷 Multilevel Inheritance with Method Overriding and `super` Keyword

✅ Definition:

Multilevel Inheritance means a class is derived from a class which is already derived from another class.
Method Overriding means redefining a superclass method in a subclass.
The `super` keyword is used to access superclass methods or constructors.

Program Name: `CS_StudentHierarchy.java`

```
// Parent class
class Person {
    void displayRole() {
        System.out.println("Role: General Person");
    }
}
// Intermediate class
class Student extends Person {
    void displayRole() {
        super.displayRole();  // Calls parent method
        System.out.println("Role: Student");
    }
}
// Final subclass
class ComputerScienceStudent extends Student {
    void displayRole() {
        super.displayRole();  // Calls Student's method
        System.out.println("Role: Computer Science Student (B.C.A.)");
    }
}
public class CSStudentHierarchy {
    public static void main(String[] args) {
        ComputerScienceStudent cs = new ComputerScienceStudent();
        cs.displayRole();
    }
}
```

✅ Output:

Role: General Person

Role: Student

Role: Computer Science Student (B.C.A.)

🔷– JavaFX GUI: Add Two Numbers with Exception Handling

✅ Definition:

JavaFX is a GUI toolkit to create rich client applications.
Event Handling is used to perform actions when user interacts.
Exception Handling ensures the program handles invalid inputs gracefully.

🧠 Program Name: `AdditionApp.java`

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

public class AdditionApp extends Application {
    @Override
    public void start(Stage stage) {
        // UI Elements
        TextField num1 = new TextField();
        num1.setPromptText("Enter first number");

        TextField num2 = new TextField();
        num2.setPromptText("Enter second number");

Button addBtn = new Button("Add");
Label result = new Label();

// Event Handling
addBtn.setOnAction(e -> {
    try {
        int a = Integer.parseInt(num1.getText());
        int b = Integer.parseInt(num2.getText());
        int sum = a + b;
        result.setText("Sum = " + sum);
    } catch (NumberFormatException ex) {
```

```
        result.setText("Please enter valid integers.");
    }
});

// Layout
VBox root = new VBox(10, num1, num2, addBtn, result);
root.setStyle("-fx-padding: 20; -fx-alignment: center;");

Scene scene = new Scene(root, 300, 200);
stage.setTitle("CS Addition App");
stage.setScene(scene);
stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

✅ Expected Output:

- Input `10` and `20` → Output label: `Sum = 30`
- Invalid input (e.g., text) → Output label: `Please enter valid integers.`

C#

•Write a C# program for perform Jagged Array.

A **jagged array** in C# is an **array of arrays**, where each inner array can have a different size or length. Unlike a multidimensional array (which has a fixed number of rows and columns), a jagged array allows for **non-uniform rows**, making it flexible for scenarios where each row might hold different amounts of data.

```
using System;

class Program

{

    public static void Main()

    {

        // Declare a jagged array

        int[][] jaggedArray = new int[3][];



        // Initialize the rows of the jagged array

        jaggedArray[0] = new int[] { 1, 2, 3 };

        jaggedArray[1] = new int[] { 4, 5 };

        jaggedArray[2] = new int[] { 6, 7, 8, 9 };



        // Display the elements of the jagged array

        for (int i = 0; i < jaggedArray.Length; i++)

        {

            Console.Write("Row " + (i + 1) + ": ");

            for (int j = 0; j < jaggedArray[i].Length; j++)

            {

                Console.Write(jaggedArray[i][j] + " ");

            }

            Console.WriteLine();

        }

    }

}
```

✅ Delegate

🔷 Definition:

A **delegate** in C# is a **type-safe function pointer** that holds a reference to a method with a specific signature and return type. It allows methods to be passed as parameters, enabling **callback mechanisms**, **event handling**, and **dynamic method invocation**.

- **Single-cast delegate**

```
//write a C# program to perform single_delegate
using System;

public delegate void del(int n1, int n2);

class del_cls
{
    public void sum(int num1, int num2)
    {
        Console.WriteLine("Sum is :" + (num1 + num2));
    }
}

class single_delegate
{
    static void Main(string[] args)
    {
        del_cls obj = new del_cls();
        del del_obj = new del(obj.sum);
        del_obj(10, 10);  // Calling the delegate
        Console.ReadLine();
    }
}
```

- **Multi-cast Delegate**

```
//Write a C# program to perform multicast delegate
using System;

public delegate void del(int n1, int n2);

class del_cls
{
    public void sum(int num1, int num2)
    {
        Console.WriteLine("Addition is: " + (num1 + num2));
    }

    public void sub(int num1, int num2)
    {
        Console.WriteLine("Subtraction is: " + (num1 - num2));
    }

    public void mul(int num1, int num2)
    {
        Console.WriteLine("Multiplication is: " + (num1 * num2));
    }
}

class multicast_delegate
{
    static void Main(string[] args)
    {
        // Create an object of the del_cls class
        del_cls obj = new del_cls();
```

```
        // Create multicast delegate and assign methods to it
        del del_obj = null;

        // Adding methods to the delegate
        del_obj += obj.sum;
        del_obj += obj.sub;
        del_obj += obj.mul;

        del_obj(10,10);

        Console.ReadLine();
    }
}
```

✅ `ref` and `out` Keywords in C#

Both `ref` and `out` are used to **pass arguments by reference** to a method.
This means any changes made inside the method will reflect outside the method too.

But there's a key difference:

Keyword	Value Before Passing	Must Be Initialized Before Use?
`ref`	Already contains value	✅ Yes
`out`	May or may not contain value	❌ No (must assign in method)

```
using System;

class Demo
{
    // Method demonstrating ref and out parameters
    public void Fun(ref int n1, out int n2)
    {
        n1 = 15;  // Modifying ref parameter
        n2 = 20;  // Assigning value to out parameter
    }
}

class RefOutExample
{
    static void Main(string[] args)
    {
        int num1 = 10, num2; // num2 is uninitialized
        Demo obj = new Demo();

        Console.WriteLine("Value of num1 before function call: " + num1);

        // Calling function with ref and out parameters
        obj.Fun(ref num1, out num2);

        Console.WriteLine("\nValue of num1 after function call: " + num1);
        Console.WriteLine("Value of num2 after function call: " + num2);
    }
}
```

✅ Indexers in C#

An **indexer** in C# allows an object to be **indexed like an array**.

- 🔹 It lets you **access class objects using square brackets `[ ]`**, just like arrays, but with your own custom logic.

👉 You can think of indexers as **customizable array access** for classes or objects.

```
using System;

class StudentNames {
    private string[] names = new string[5];

    // Define an indexer
    public string this[int index]
    {
        get
        {
            return names[index];
        }
        set
        {
            names[index] = value;
        }
    }
}

class Program {
    static void Main() {
        StudentNames students = new StudentNames();

        // Using indexer to set values
        students[0] = "Mihir";
        students[1] = "Harshal";
        students[2] = "Varshil";

        // Using indexer to get values
        Console.WriteLine("Student at index 0: " + students[0]);
        Console.WriteLine("Student at index 1: " + students[1]);
    }
}
```

✅ Collections in C#

A **Collection** in C# is a **group of related objects** that can be **stored, managed, and accessed** together.

🔹 Collections are like **containers** (lists, queues, stacks, dictionaries) that let you:

- Store multiple items
- Add/remove/search items
- Loop through items

📦 Instead of using arrays (which are fixed size), collections are **dynamic** in size.

✅ Types of Collections in C#

C# provides 2 main types:

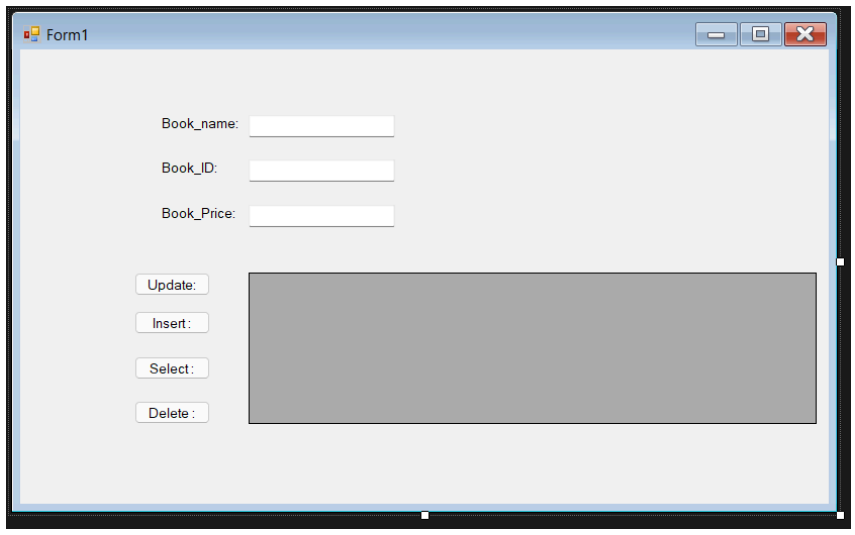
Category	Example Classes	Description
Non-Generic	`ArrayList`, `Hashtable`, `Stack`, `Queue`	Can store any type of data (not type-safe)
Generic (✔ Recommended)	`List<T>`, `Dictionary<K,V>`, `Queue<T>`, `Stack<T>`	Type-safe, faster, and flexible

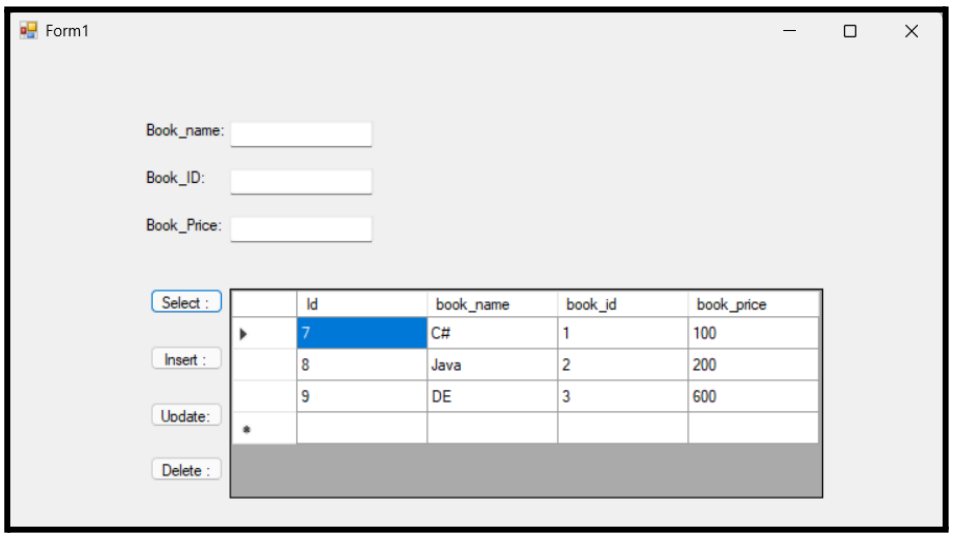
```
using System;
using System.Collections.Generic;

class Program {
    static void Main() {
        List<string> sub = new List<string>();  // Create a string list

        // Add items
        sub.Add("C");
        sub.Add("CPP");
        sub.Add("CSharp");

        // Access using index
        Console.WriteLine("First items : " + sub[0]);

        // Loop through all items
        Console.WriteLine("All Subject:");
        foreach (string newvar in sub) {
            Console.WriteLine(newvar);
        }
    }
}
```

✅ ADO.NET Connected Architecture to Insert, Update, Delete, Select

🔷 Definition:

Connected architecture uses `SqlConnection`, `SqlCommand`, and `SqlDataReader` to directly interact with the database.






```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using static System.Windows.Forms.VisualStyles.VisualStyleElement;

namespace exam_demo.cs
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

SqlConnection Con = new SqlConnection(@"Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=G:\BCA_SEM-4\SEM_4_PRACTICAL\B_416_C#\Win_Application\exam_demo.cs\exa
m_demo.cs\stationery.mdf;Integrated Security=True");

 private void button1_Click(object sender, EventArgs e) //UPDATE BUTTON
        {
            Con.Open();
            SqlCommand cmd = new SqlCommand("UPDATE STATIONERYINFO SET book_name='"
                + textBox1.Text+"',book_price='"+textBox3.Text+"' WHERE book_id ='"+textBox2.Text+"' ", Con);
            cmd.ExecuteNonQuery();
            Con.Close();

            MessageBox.Show("Data Updated Successfully");
        }
        private void button2_Click(object sender, EventArgs e) //INSERT BUTTON
        {
            Con.Open();
            SqlCommand cmd = new SqlCommand("INSERT INTO STATIONERYINFO VALUES('" + textBox1.Text + "', '" + textBox2.Text + "', '" +
textBox3.Text + "')", Con);
            cmd.ExecuteNonQuery();
            Con.Close();

            MessageBox.Show("Data Inserted Successfully");

        }
```

```
 private void button3_Click(object sender, EventArgs e)//SELECT BUTTON
      {
         Con.Open();
         SqlCommand cmd = new SqlCommand("SELECT *FROM STATIONERYINFO", Con);
         SqlDataAdapter ad = new SqlDataAdapter(cmd);
         DataTable dt = new DataTable();
         ad.Fill(dt);
         dataGridView1.DataSource = dt;
         cmd.ExecuteNonQuery();
         Con.Close();
      }

      private void button4_Click(object sender, EventArgs e)//DELETE BUTTON
      {
         Con.Open();
         SqlCommand cmd = new SqlCommand("DELETE FROM STATIONERYINFO WHERE book_id='" + textBox2.Text + "'", Con);
         cmd.ExecuteNonQuery();
         Con.Close();
         MessageBox.Show("Data Deleted Successfully");
      }
   }
}
```

✅Disconnected Architecture

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace exam_demo.cs
{
   public partial class Form1 : Form
   {
          public Form1()
      {
         InitializeComponent();
      }
 SqlConnection Con = new
SqlConnection(@"DataSource=(LocalDB)\MSSQLLocalDB;AttachDbFilename=G:\BCA_SEM-4\SEM_4_PRACTICAL\B_416_C#\Win_Applica
tion\exam_demo.cs\exam_demo.cs\stationery.mdf;Integrated Security=True");
```

```
 private void button1_Click(object sender, EventArgs e) // UPDATE BUTTON (DISCONNECTED)
 {
    SqlDataAdapter ad = new SqlDataAdapter("SELECT * FROM STATIONERYINFO WHERE book_id='" + textBox2.Text + "'", Con);
    DataSet ds = new DataSet();
    ad.Fill(ds, "STATIONERYINFO");

    if (ds.Tables["STATIONERYINFO"].Rows.Count > 0)
    {
       DataRow row = ds.Tables["STATIONERYINFO"].Rows[0];
       row["book_name"] = textBox1.Text;
       row["book_price"] = textBox3.Text;

       SqlCommandBuilder cb = new SqlCommandBuilder(ad);
       ad.Update(ds, "STATIONERYINFO");

       MessageBox.Show("Data Updated Successfully");
    }
    else
    {
       MessageBox.Show("Record not found!");
     }
}
```

```
private void button2_Click(object sender, EventArgs e) // INSERT BUTTON (DISCONNECTED)
    {
        SqlDataAdapter ad = new SqlDataAdapter("SELECT * FROM STATIONERYINFO", Con);
        DataSet ds = new DataSet();
        ad.Fill(ds, "STATIONERYINFO");

        DataRow newRow = ds.Tables["STATIONERYINFO"].NewRow();
        newRow["book_name"] = textBox1.Text;
        newRow["book_id"] = textBox2.Text;
        newRow["book_price"] = textBox3.Text;
        ds.Tables["STATIONERYINFO"].Rows.Add(newRow);

        SqlCommandBuilder cb = new SqlCommandBuilder(ad);
        ad.Update(ds, "STATIONERYINFO");

        MessageBox.Show("Data Inserted Successfully");
    }


    private void button3_Click(object sender, EventArgs e) // SELECT BUTTON (AIready DISCONNECTED)
    {
        Con.Open();
        SqlCommand cmd = new SqlCommand("SELECT *FROM STATIONERYINFO", Con);
        SqlDataAdapter ad = new SqlDataAdapter(cmd);
        DataTable dt = new DataTable();
        ad.Fill(dt);
        dataGridView1.DataSource = dt;
        cmd.ExecuteNonQuery();
        Con.Close();
    }



private void button4_Click(object sender, EventArgs e) // DELETE BUTTON (DISCONNECTED)
    {
        SqlDataAdapter ad = new SqlDataAdapter("SELECT * FROM STATIONERYINFO WHERE book_id='" + textBox2.Text + "'", Con);
        DataSet ds = new DataSet();
        ad.Fill(ds, "STATIONERYINFO");

        if (ds.Tables["STATIONERYINFO"].Rows.Count > 0)
        {
            ds.Tables["STATIONERYINFO"].Rows[0].Delete();

            SqlCommandBuilder cb = new SqlCommandBuilder(ad);
            ad.Update(ds, "STATIONERYINFO");

            MessageBox.Show("Data Deleted Successfully");
        }
        else
        {
            MessageBox.Show("Record not found!");
        }
    }
  }
}
```

Operating System

✅ 1. Redirection and Piping

🔹 Redirection:

Redirection is used to **send the output** of a command to a **file**, or **take input** from a file instead of the keyboard.

✅ Types of Redirection:

Symbol	Purpose	Example
`>`	Redirect output to a new file (overwrite)	`ls > list.txt`
`>>`	Redirect output and **append** to file	`echo "Hi" >> file.txt`
`<`	Take input from file instead of keyboard	`sort < data.txt`

🔹 Piping (`|`):

Piping connects **two or more commands**, sending **output of one command as input to another**.

✅ Example:

ls | sort

➡️ This lists all files and **sorts** them.

✅ 2. File and Directory Related Commands

These commands help manage **files and folders** in Unix/Linux.

Command	Description	Example
`ls`	List files/folders	`ls -l`
`cd`	Change directory	`cd Documents/`
`pwd`	Print current directory path	`pwd`
`mkdir`	Create new directory	`mkdir mydir`
`rmdir`	Remove empty directory	`rmdir mydir`
`rm`	Remove file	`rm file.txt`
`cp`	Copy file	`cp a.txt b.txt`
`mv`	Move or rename file	`mv old new`
`touch`	Create empty file	`touch file.txt`
`cat`	View file content	`cat file.txt`

✅ 3. Finding Pattern in Files

Used to **search specific text or pattern** inside files using tools like `grep`.

✅ Example using `grep`:

grep "Mihir" names.txt

➡️ This finds and shows all lines containing **"Mihir"** from `names.txt`.

You can also use:

grep -i "mihir" file.txt   # case-insensitive
grep -n "error" log.txt   # show line numbers with result

✅ 4. Positional Parameters

Positional parameters are **special variables** in shell scripting (`$1`, `$2`, ..., `$9`) that **store values passed to a script**.

✅ Example:

Suppose you have a script named `greet.sh`:

#!/bin/bash
echo "Hello $1, Welcome to $2!"

Run it like:

bash greet.sh Mihir Linux

✅ Output:

Hello Mihir, Welcome to Linux!

Variable	Holds
`$0`	Script name
`$1`	First argument
`$2`	Second argument
`$@`	All arguments as list
`$#`	Number of arguments

✅File Permission Checker Script

🔷 Definition:

Shell has **file test operators** like `-e`, `-r`, `-w`, `-x` to check if a file **exists**, is **readable**, **writable**, and **executable**.

✅ Script: `check_file.sh`

```
#!/bin/bash

echo "Enter the filename:"
read file

if [ -e "$file" ]; then
    echo "File exists."

    if [ -r "$file" ]; then
        echo " File is readable."
    else
        echo " File is not readable."
    fi

    if [ -w "$file" ]; then
        echo "File is writable."
    else
        echo " File is not writable."
    fi

    if [ -x "$file" ]; then
        echo "File is executable."
    else
        echo "File is not executable."
    fi

else
    echo "File does not exist."
fi
```

📌 How to Run:

```
chmod +x check_file.sh
./check_file.sh
```

✅ Sample Output:

```
Enter the filename:
Myfile.txt

File exists.
File is readable.
File is not writable.
File is not executable.
```

✅ Factorial using While Loop

🔷 Definition:

A **factorial** is the product of all numbers from 1 to N.
Shell uses `while` loop and `expr` or `$(( ))` for arithmetic.

✅ Script: `factorial.sh`

```
#!/bin/bash

echo "Enter a number:"
read num

fact=1
i=1

while [ $i -le $num ]
do
    fact=$((fact * i))
    i=$((i + 1))
done

echo "Factorial of $num is: $fact"
```

📌 How to Run:

```
chmod +x factorial.sh
./factorial.sh
```

✅ Sample Output:

```
Enter a number:
5
Factorial of 5 is: 120
```
