Practical No: 11

Roll.No: TTA-43 **Date:10 /09/2025**

Teacher's Signature

Title: Error Handling and Logging

Sub Title: Add proper error handling and meaningful logging to your code. Write a program that processes user input. Use clear error messages and log essential information to help identify issues. Avoid generic error handling.

Bad Code : Python (Division)

```
x = int(input("Enter numerator: "))
y = int(input("Enter denominator: "))
print("Result:", x / y)   # No error handling
```

- Crashes if denominator = 0.
- Crashes if input is not an integer.
- No logging, debugging is hard.

Refactored Code:

```
import logging

logging.basicConfig(filename="app.log", level=logging.INFO)

try:
    x = int(input("Enter numerator: "))
    y = int(input("Enter denominator: "))
    result = x / y
    logging.info(f"Division successful: {x}/{y}={result}")
    print("Result:", result)
except ZeroDivisionError:
    logging.error("Attempted division by zero")
    print("Error: Cannot divide by zero")
except ValueError as e:
    logging.warning(f"Invalid input: {e}")
    print("Error: Please enter integers only"}
```

- Handles **division by zero** gracefully.
- Handles **invalid input**.
- Adds **logging (INFO, WARNING, ERROR)** for debugging.

Q.2] Java (File Reading)

Bad Code:

```
import java.io.*;

public class FileReadExample {
   public static void main(String[] args) throws Exception {
      BufferedReader br = new BufferedReader(new FileReader("data.txt"));
      String line = br.readLine();
      System.out.println("First line: " + line); // No error handling
      br.close();
   }
}
```

- Throws raw exceptions, no proper handling.
- No logs → difficult to debug.
- User sees ugly stack trace.

Refactored code:

```
import java.io.*;
import java.util.logging.*;

public class FileReadExample {
   private static final Logger logger =
Logger.getLogger(FileReadExample.class.getName());

   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("data.txt")))
{
         String line = br.readLine();
         System.out.println("First line: " + line);
         logger.info("File read successfully");
      } catch (FileNotFoundException e) {
         logger.severe("File not found: " + e.getMessage());
         System.out.println("Error: File not found");
      } catch (IOException e) {
         logger.severe("IO error: " + e.getMessage());
         System.out.println("Error reading file");
      }
   }
}
```

- Uses **try-with-resources** for safe closing.
- Handles **FileNotFoundException** and **IOException** separately.
- Logs errors clearly with Logger.

Q.3] C (Integer Input Validation)

Bad Code:

```
#include <stdio.h>

int main() {
   int num;
   scanf("%d", &num);   // No validation
   printf("You entered: %d\n", num);
   return 0;
}
```

- If user enters non-integer → undefined behavior.
- No error message shown.
- No separation of error output.

Refactored Code:

```
#include <stdio.h>

int main() {
   int num;
   printf("Enter an integer: ");
   if (scanf("%d", &num) != 1) {
      fprintf(stderr, "Error: Invalid input, not an integer\n");
      return 1;
   }
   printf("You entered: %d\n", num);
   return 0;
}
```

- Validates **scanf return value**.
- Prints errors to stderr.
- Prevents crashes and gives user-friendly message.

4] C++ (Division)

Bad Code:

```
#include <iostream>
using namespace std;

int main() {
   double a, b;
   cin >> a >> b;
   cout << "Result: " << a / b << endl; // No handling for zero
   return 0;
}
```

- Division by zero → crash or inf.
- No error info shown.
- User confusion if input invalid.

Refactored Code:

```
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {
   double a, b;
   cout << "Enter numerator and denominator: ";
   cin >> a >> b;
   try {
      if (b == 0) throw runtime_error("Division by zero");
      cout << "Result: " << a / b << endl;
   } catch (runtime_error &e) {
      cerr << "Error: " << e.what() << endl;
   }
   return 0;
}
```

- Uses **C++ exceptions**.
- Detects **division by zero**.
- Prints error to cerr (standard error).

5] Login Logs

login log function:

```
def log_user_action(user_id, username, action="login", status="success"):
   """
   Log user login activities
   """
   try:
      conn = get_db_connection()
      if not conn:
         return False

      cursor = conn.cursor(dictionary=True)

      # Get IP address from request
      ip_address = request.remote_addr if hasattr(request, 'remote_addr') else 'unknown'

      cursor.execute(
         """INSERT INTO user_login_logs (user_id, username, action, status, ip_address)
            VALUES (%s, %s, %s, %s, %s)""",
         (user_id, username, action, status, ip_address)
      )
      conn.commit()
      cursor.close()
      conn.close()
```

```
        return True
    except Exception as e:
        print(f"❌ Error logging user action: {e}")
        return False
```

logs.html:

```
{% extends 'base.html' %}

{% block content %}
<div class="container mt-4">
    <h2><i class="fas fa-clipboard-list"></i> User Login Logs</h2>

    <div class="card">
        <div class="card-body">
            <div class="table-responsive">
                <table class="table table-striped">
                    <thead>
                        <tr>
                            <th>Date/Time</th>
                            <th>User</th>
                            <th>Action</th>
                            <th>Status</th>
                            <th>IP Address</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for log in logs %}
                        <tr>
                            <td>{{ log.login_time.strftime('%Y-%m-%d %H:%M:%S') }}</td>
                            <td>
                                <strong>{{ log.full_name or log.username }}</strong><br>
                                <small class="text-muted">{{ log.email }}</small>
                            </td>
                            <td>
                                <span class="badge bg-info">{{ log.action }}</span>
                            </td>
                            <td>
                                {% if log.status == 'success' %}
                                <span class="badge bg-success">Success</span>
                                {% elif log.status == 'failed' %}
                                <span class="badge bg-danger">Failed</span>
                                {% else %}
                                <span class="badge bg-warning">Error</span>
```

```
                    {% endif %}
                </td>
                <td><code>{{ log.ip_address }}</code></td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
</div>
</div>
</div>
{% endblock %}
```

Output: