UNIT IV – ADVANCED EXCEPTION HANDLING

Advanced Exception Handling:

- Advanced exception handling in Java extends beyond the basic try-catch-finally blocks and focuses on creating robust, maintainable, and user-friendly error management strategies.
- **Specific Exception Handling:**
  - Catch the most specific exception type possible rather than generic Exception or Throwable. This allows for targeted handling of different error scenarios.
  - Example:Catch FileNotFoundException insteadof a general IOException when dealing with file operations.
- **Avoid Empty Catch Blocks:**
  - Never leave a catch block empty, as this "swallows" exceptions, making debugging and error identification extremely difficult.
  - Always log the exception or provide a meaningful error message to the user.

```
try
{
    int result = 10 / 0;
}
catch (ArithmeticException e)
{
    // empty catch block – silently ignores the exception
}
catch (ArithmeticException e)
{
    System.err.println("Arithmetic error: " + e.getMessage());
}
```

- **Custom Exceptions:**
  - ✓ Define custom exception classes to represent specific application-level errors.
  - ✓ This improves code clarity and allows for more precise error handling and reporting.
  - ✓ Example: InvalidInputException,

```
InsufficientFundsException. public class
InsufficientFundsException extends Exception
{
    public InsufficientFundsException(String message)
   {
        super(message);
    }
    public InsufficientFundsException(String message, Throwable cause)
    {
       super(message, cause);
    }
}
public class BankAccount
{
    private double balance;
    public BankAccount(double initialBalance)
    {
        this.balance = initialBalance;
    }
    public void withdraw(double amount) throws InsufficientFundsException
  {
        if (amount > balance)
         {
            throw new InsufficientFundsException("Insufficient funds. Available
balance: " + balance);
         }
        balance -= amount;
        System.out.println("Withdrawal successful. New balance: " + balance);
   }
}
public class BankingApp
{
    public static void main(String[] args)
    {
        BankAccount account = new
        BankAccount(1000.00); try
        {
                account.withdraw(500.00);
                account.withdraw(800.00);
        }
        catch (InsufficientFundsException e)
        {
                System.out.println("Caught an exception: " + e.getMessage());
        }
    }
}
```

- **Multi-catch blocks:**
    - ✓ We handle multiple exception types in a single catch block.
    - ✓ This reduces code duplication when the handling logic is the same for several exceptions.

```
try
{
    // throw ArithmeticException /ArrayIndexOutOfBoundsException
}
catch (ArithmeticException | ArrayIndexOutOfBoundsException e)
{
    System.err.println("An array or math error occurred: "+ e.getMessage());
}
catch (Exception e)
{
        System.err.println("A more general exception occurred.");
}
```

- **try-with-resources:**
    - ✓ Ensures that each resource is closed automatically at the end of the statement.

- ✓ This is a more robust alternative to using a finally block for manual cleanup, as it prevents resource leaks even if an exception occurs during the closing process.

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt")))
{
        String line;
        while ((line = reader.readLine()) != null)
        {
                System.out.println(line);
        }
}
catch (IOException e)
{
        System.err.println("Error reading file: " + e.getMessage());
}
```

- **Exception Chaining:**
  - ✓ Wrapping lower-level exceptions in higher-level, more descriptive exceptions allows for preserving the original cause of an error while providing a more meaningful context to the calling code.
  - ✓ This is achieved by passing the original exception as a cause to the new exception's constructor.

```
try
{
        // Some operation that might throw SQLException
}

catch (SQLException e)
{
        throw new DataAccessException("Error accessing data", e);
        // Chaining the SQLException
}
```

- **Logging Exceptions:**
  - ✓ Instead of simply printing stack traces to the console, using a robust logging

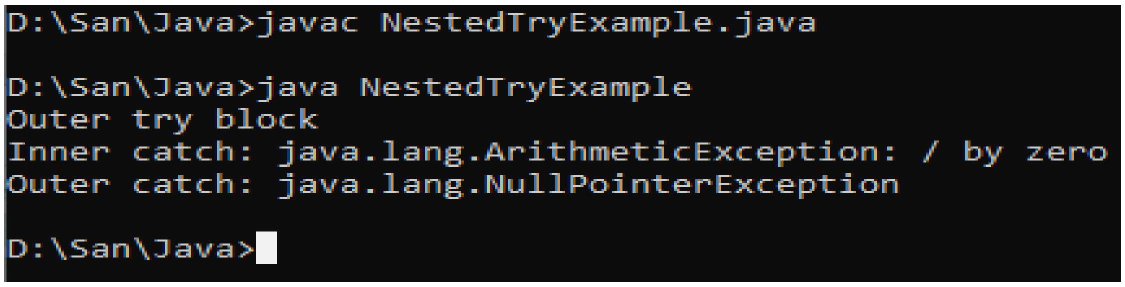
framework (e.g., Log4j, SLF4J) to log exceptions with relevant details (stack trace, context information).

- ✓ It provides structured logging, allowing for better error analysis, filtering, and integration with monitoring tools.

- ➢ **Global Exception Handlers:**
  - ✓ For web applications or large systems, implementing a global exception handler can centralize error handling logic, providing a consistent way to manage unhandled exceptions and present user-friendly error messages.

- **Documenting Exceptions:**
  - ✓ Using Javadoc to document the exceptions a method might throw (both checked and unchecked) is crucial for informing other developers about potential error scenarios and how to handle them.
- **Nested Try Block:**
  - ✓ A nested try block refers to a try block placed inside another try block.
  - ✓ This structure is used to handle exceptions that may occur at different levels of code execution, allowing for more specific and localized exception handling.
  - ✓ Nested try blocks are useful for handling exceptions at different levels of code.
  - ✓ If an exception occurs in a parent try block, the control jumps directly to the matching catch block in the parent or outer try block, and any nested try blocks are skipped.
  - ✓ If an exception occurs in an inner try block and is not caught, it propagates to the outer try block.

```
public class NestedTryExample
{
    public static void main(String[] args)
    {
        try
        {
                System.out.println("Outer try block");
                try
                {
                        int a = 10 / 0; // This causes ArithmeticException
                }
                catch (ArithmeticException e)
                {
                        System.out.println("Inner catch: " + e);
                }
                String str = null;
                System.out.println(str.length()); // NullPointerException
        }
        catch (NullPointerException e)
        {
              System.out.println("Outer catch: " + e);
        }
    }
}
```



```
D:\San\Java>javac NestedTryExample.java

D:\San\Java>java NestedTryExample
Outer try block
Inner catch: java.lang.ArithmeticException: / by zero
Outer catch: java.lang.NullPointerException

D:\San\Java>
```