WEEK 4 MODULE: EXCEPTION HANDLING AND INPUT/OUTPUT OPERATIONS

TOPICS:

- Introduction to exceptions and error handling
- Handling exceptions using try-catch blocks
- Introduction File handling
- Java I/O operations
- Reading from and writing to files (txt, csv, json files)
- Input and Output Stream
- JDBC Database Connections

LEARNING OUTCOMES:

By the end of this module, students will be able to:

- Gain a comprehensive understanding of exceptions and error handling in Java.
- Implement effective exception handling using try-catch blocks to handle and manage errors in Java programs.
- Explore file handling concepts in Java and learn techniques for interacting with files.
- Perform various I/O operations in Java, including reading from and writing to files in different formats such as text (txt), comma-separated values (csv), and JavaScript Object Notation (JSON) files.
- Understand the role of input and output streams in Java and their significance in data processing.
- Develop practical skills in reading and writing data from/to files, handling exceptions gracefully, and effectively utilizing I/O operations.
- Gain an introduction to JDBC (Java Database Connectivity) for establishing connections to databases and performing database operations.

DAY 1:
INTRODUCTION TO EXCEPTIONS AND ERROR HANDLING

EXCEPTION HANDLING: Exception handling in Java is the process of responding to the occurrence of exceptions (Exceptions are unwanted conditions that disturb the program execution), it occurred.

Exceptional handling in Java is a very powerful mechanism as it helps to identify exceptional conditions and maintain the flow of the program as expected, by handling/avoiding the errors if occurred. In some cases, it is used to make the program user-friendly.

EXCEPTION: An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- Reading a file from a disk, but the file does not exist there.
- Writing data to a disk but the disk is full or unformatted.
- When a user attempts to divide an integer value by zero, an exception occurs.
- When a data stream is in an invalid format, etc.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must be caught or
declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

Live Demo

```
public class Unchecked_Demo {

   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

If you compile and execute the above program, you will get the following exception.

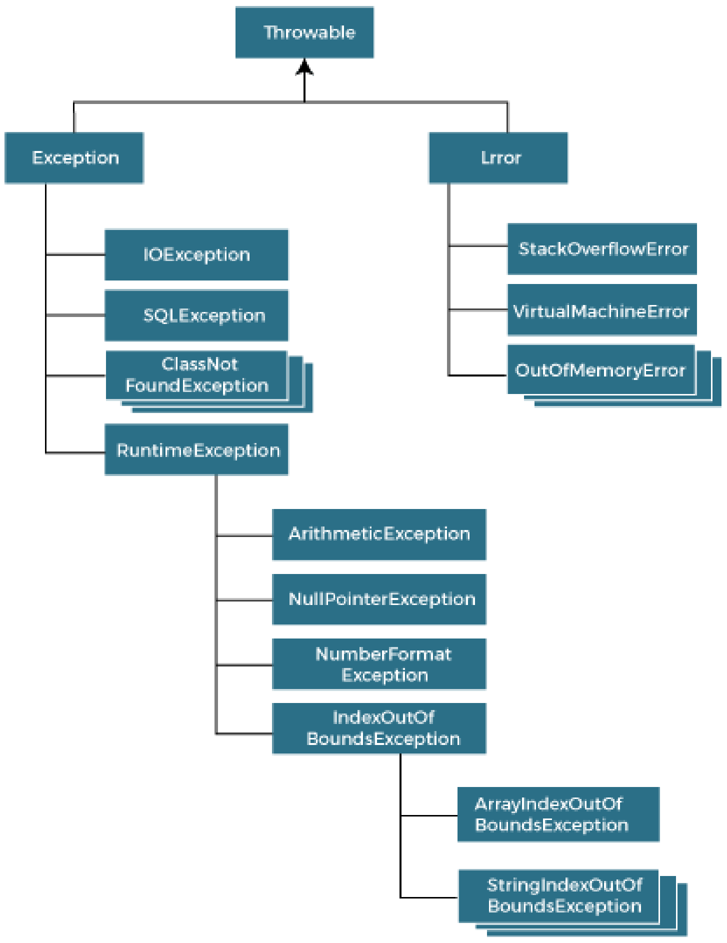
Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Java Exception Keywords

Keyword	**Description**
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Common Scenarios of Java Exceptions
There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs
If we divide any number by zero, there occurs an ArithmeticException.

1. **int** a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs
If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

2. String s=**null**;
3. System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs
If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

4. String s="abc";
5. **int** i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs
When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

6. **int** a[]=**new int**[5];
7. a[10]=50; //ArrayIndexOutOfBoundsException

How to Handle an Exception
Java provides two different options to handle an exception. You can either use the try-catch-finally approach to handle all kinds of exceptions. Or you can use the try-with-resource approach which allows an easier cleanup process for resources.

Try-Catch-Finally
That is the classical approach to handle an exception in Java. It can consist of 3 steps:

- a *try* block that encloses the code section which might throw an exception,
- one or more *catch* blocks that handle the exception and
- a *finally* block which gets executed after the *try* block was successfully executed or a thrown exception was handled.

The *try* block is required, and you can use it with or without a *catch* or *finally* block.

The Try Block
Let's talk about the *try* block first. It encloses the part of your code that might throw the exception. If your code throws more than one exception, you can choose if you want to:

- use a separate *try* block for each statement that could throw an exception or
- use one *try* block for multiple statements that might throw multiple exceptions.

The following example shows a try block which encloses three method calls.

```
public void performBusinessOperation() {
	try {
		doSomething("A message");
		doSomethingElse();
		doEvenMore();
	}
	// see following examples for catch and finally blocks
}

public void doSomething(String input) throws MyBusinessException {
	// do something useful ...
	throw new MyBusinessException("A message that describes the error.");
}

public void doSomethingElse() {
	// do something else ...
}

public void doEvenMore() throws NumberFormatException{
	// do even more ...
}
```

As you can see in the method definitions, only the first and the third method specify an exception. The first one might throw a *MyBusinessException*, and the *doEvenMore* method might throw a *NumberFormatException*.

In the next step, you can define one *catch* block for each exception class you want to handle and one *finally* block. All checked exceptions that are not handled by any of the catch blocks need to be specified.

The Catch Block
You can implement the handling for one or more exception types within a *catch* block. As you can see in the following code snippet, the catch clause gets the exception as a parameter. You can reference it within the catch block by the parameter name.

```
public void performBusinessOperation() {
	try {
		doSomething("A message");
		doSomethingElse();
		doEvenMore();
	} catch (MyBusinessException e) {
		e.printStackTrace();
	} catch (NumberFormatException e) {
		e.printStackTrace();
	}
}
```

The previous code sample shows two catch blocks. One to handle the *MyBusinessException* and one to handle the *NumberFormatException*. Both blocks handle the exceptions in the same way. Since Java 7, you can do the same with just one catch block.

```
public void performBusinessOperation() {
	try {
		doSomething("A message");
		doSomethingElse();
		doEvenMore();
	} catch (MyBusinessException|NumberFormatException e) {
		e.printStackTrace();
	}
}
```

The Finally Block

The *finally* block gets executed after the successful execution of the *try* block or after one of the *catch* blocks handled an exception. It is, therefore, a good place to implement any cleanup logic, like closing a connection or an *InputStream*.

You can see an example of such a cleanup operation in the following code snippet. The *finally* block will be executed, even if the instantiation of the *FileInputStream* throws a *FileNotFoundException* or the processing of the file content throws any other exception.

```
FileInputStream inputStream = null;
try {
	File file = new File("./tmp.txt");
	inputStream = new FileInputStream(file);

	// use the inputStream to read a file

} catch (FileNotFoundException e) {
	e.printStackTrace();
} finally {
	if (inputStream != null) {
		try {
			inputStream.close();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

As you’ve seen, the *finally* block provides a good option to prevent any leaks. And before Java 7, it was a best practice to put all cleanup code into a *finally* block.

Try-With-Resource
That changed when Java 7 introduced the try-with-resource statement. It automatically closes all resources that implement the *AutoCloseable* interface. And that is the case for most Java objects that you need to close.

The only thing you need to do to use this feature is to instantiate the object within the try clause. You also need to handle or specify all exceptions that might be thrown while closing the resource.

The following code snippet shows the previous example with a try-with-resource statement instead of a try-catch-finally statement.

```
File file = new File("./tmp.txt");
try (FileInputStream inputStream = new FileInputStream(file);) {
        // use the inputStream to read a file
} catch (FileNotFoundException e) {
        e.printStackTrace();
} catch (IOException e) {
        e.printStackTrace();
}
```

DAY 2:

Introduction to File Handling in Java:

File handling is an essential aspect of programming that involves reading from and writing to files on a computer's storage system. In Java, file handling is performed using the File, FileReader, FileWriter, BufferedReader, BufferedWriter, and other related classes provided by the Java I/O (Input/Output) API.

To begin file handling in Java, you need to understand a few key concepts:

- File Representation: In Java, a file is represented by the File class. This class provides methods to create, delete, and access file-related information, such as the file's name, path, size, and permissions. You can create an instance of the File class by specifying the file's path as a string.
- Reading from a File: To read data from a file, you can use the FileReader and BufferedReader classes. The FileReader class allows you to read characters from a file, while the BufferedReader class provides buffering capabilities and improves performance. You can read data from a file line by line or character by character using these classes.

- Writing to a File: To write data to a file, you can use the FileWriter and BufferedWriter classes. The FileWriter class allows you to write characters to a file, and the BufferedWriter class provides buffering capabilities for improved performance. You can write data to a file line by line or character by character using these classes.
- Exception Handling: File handling operations may encounter exceptions, such as file not found, permission denied, or I/O errors. It is important to handle these exceptions gracefully using try-catch blocks to prevent program crashes and provide appropriate error messages to users.
- Closing Files: After performing file handling operations, it is essential to close the file resources properly. Closing files ensures that any pending data is written to the file, and system resources are released. Failing to close files can result in memory leaks and resource exhaustion.

File handling in Java provides great flexibility and enables you to interact with different types of files, including text files, binary files, and more. It allows you to read and write data, create new files, manipulate existing files, and perform various operations related to file management.

JAVA I/O OPERATIONS:

Java I/O (Input/Output) operations involve reading and writing data to different sources, such as files, network connections, and the console. It is an essential part of Java programming that allows you to interact with external data and perform various input and output tasks.

- Input and Output Streams: In Java, I/O operations are performed using input and output streams. An input stream is used for reading data from a source, while an output stream is used for writing data to a destination. Streams provide a consistent way of handling data regardless of the data source or destination.
- InputStream and OutputStream: The InputStream and OutputStream classes are the basic building blocks for Java I/O operations. The InputStream class provides methods to read data, while the OutputStream class provides methods to write data. These classes are the foundation for more specialized I/O operations.
- Reading and Writing Data: To read data from an input source, such as a file or network connection, you can use classes like FileInputStream or SocketInputStream, along with the appropriate read() methods. To write data to an output destination, such as a file or network connection, you can use classes like FileOutputStream or SocketOutputStream, along with the appropriate write() methods.
- Buffering: Java I/O operations can be made more efficient by using buffering. Buffering involves using an intermediate buffer to store data temporarily before it is read or written. BufferedInputStream and BufferedOutputStream are examples of classes that provide buffering capabilities, improving the performance of I/O operations.
- Exception Handling: When performing I/O operations, it is important to handle exceptions that may occur. Exceptions can be raised due to file not found, I/O errors, or other issues. Exception handling using try-catch blocks allows you to handle these exceptions gracefully and provide appropriate error messages to users.
- Closing Streams: After performing I/O operations, it is crucial to close the streams properly. Closing streams ensures that any buffered data is flushed and resources are released. Failing to close streams can lead to memory leaks and resource exhaustion.

Java I/O operations are used in various scenarios, such as reading configuration files, writing log files, communicating with external systems, and processing user input. Understanding the basic concepts of Java I/O operations will enable you to handle data effectively and build applications that interact with different data sources.

READING FROM AND WRITING TO FILES (TXT, CSV, JSON FILES) AND INPUT/OUTPUT STREAM

Reading from Files:

- Select the File to Read: Start by identifying the file you want to read from. You need to provide the file's path or location in your Java code.
- Create a FileReader or FileInputStream: To read text-based files (such as txt and csv files), you can use the FileReader class. If you're working with binary files, you can use the FileInputStream class. These classes provide methods for reading data from files.
- Wrap the Reader in a BufferedReader: To improve performance, wrap the FileReader or FileInputStream in a BufferedReader. BufferedReader provides buffering capabilities, allowing you to read data more efficiently.
- Read Data from the File: Use the read methods provided by the BufferedReader to read data from the file. For example, you can use the readLine() method to read text line by line or the read() method to read individual characters.
- Process the Read Data: Once you have read the data, you can process it according to your requirements. For example, you can store the data in variables, manipulate it, or perform calculations.

Writing to Files:

- Select the File to Write: Identify the file you want to write to. You need to provide the file's path or location in your Java code. If the file doesn't exist, it will be created automatically.
- Create a FileWriter or FileOutputStream: To write text-based files, use the FileWriter class. For binary files, use the FileOutputStream class. These classes provide methods for writing data to files.
- Wrap the Writer in a BufferedWriter: Improve performance by wrapping the FileWriter or FileOutputStream in a BufferedWriter. BufferedWriter provides buffering capabilities, making the writing process more efficient.
- Write Data to the File: Use the write methods provided by the BufferedWriter to write data to the file. For example, you can use the write() method to write text or the writeBytes() method to write binary data.
- Close the Writer: After writing the data, remember to close the BufferedWriter to ensure that all the data is flushed and the resources are released properly.

Reading and Writing JSON Files:

- Use a JSON library: To read and write JSON files, you can utilize a JSON library such as Jackson, Gson, or JSON.simple. These libraries provide methods and classes specifically designed for working with JSON data.

- Read JSON from File: Use the library's methods to read JSON data from the file. This typically involves parsing the JSON file and creating objects or data structures to represent the JSON data in your Java code.
- Process JSON Data: Once you have the JSON data in your Java code, you can access and manipulate it as needed. You can retrieve specific values, modify them, or perform any required operations.
- Write JSON to File: To write JSON data to a file, use the library's methods to convert the Java objects or data structures back into JSON format. Then, follow the steps for writing to files mentioned earlier to write the JSON data to the file.

Input and Output Stream: Input and Output Streams are used to read and write data from/to different sources in Java. You can use various InputStream and OutputStream classes to handle different types of data and sources. Here are the basic steps:

- Select the InputStream or OutputStream: Depending on the data source or destination, choose the appropriate InputStream or OutputStream class. For example, FileInputStream for reading from a file, or SocketOutputStream for writing to a network socket.
- Read or Write Data: Use the read() or write() methods provided by the selected InputStream or OutputStream class to read or write data. You can read or write data in different formats, such as bytes, characters, or objects, depending on the specific requirements.
- Close the Stream: After reading or writing data, it is important to close the stream properly using the close() method. This ensures that any resources associated with the stream are released and the data is flushed.

DAY 3:
JDBC DATABASE CONNECTIONS

JDBC (Java Database Connectivity) is a Java API that allows Java programs to interact with relational databases. It provides a set of classes and methods to establish database connections, execute SQL queries, and perform database operations. Here's an explanation of JDBC database connections at a beginner level:

- JDBC Driver: Before you can establish a database connection, you need to have the appropriate JDBC driver for the specific database you want to connect to. The JDBC driver is a software component that enables communication between Java programs and the database.
- Loading the Driver: To use the JDBC driver in your Java program, you need to load it using the Class.forName() method. This step ensures that the driver class is available and can be used for database connections.
- Database URL: A database URL is a unique identifier that specifies the location and other connection details of the database. It typically includes the database type, host, port, database name, and additional parameters.

- Establishing a Connection: Use the DriverManager.getConnection() method to establish a connection to the database. Pass the appropriate database URL, username, and password as parameters to this method. If the connection is successful, a Connection object is returned.
- Creating Statements: After establishing a connection, you can create Statement objects to execute SQL queries against the database. The Statement interface provides methods to execute queries, updates, or stored procedures.
- Executing Queries: Use the Statement.execute() or Statement.executeQuery() method to execute SQL queries. The executeQuery() method is used for retrieving data from the database, while the execute() method is used for other types of queries or updates.
- Retrieving Results: If you executed a query that returns a result set (such as a SELECT statement), you can retrieve the results using the ResultSet object. The ResultSet provides methods to iterate over the rows and access the data retrieved from the database.
- Performing Updates: If you executed an update statement (such as INSERT, UPDATE, or DELETE), you can check the number of affected rows using the Statement.getUpdateCount() method.
- Handling Exceptions: When working with JDBC database connections, it is important to handle exceptions properly. Use try-catch blocks to catch any SQLExceptions that may occur during database operations. Handle exceptions gracefully by logging or displaying appropriate error messages and taking necessary actions.
- Closing the Connection and Resources: After you have finished using the database connection, it is crucial to close it properly. Use the Connection.close() method to close the connection, and also close any associated Statement and ResultSet objects. Closing the connection releases any resources held by the connection and ensures that the database connection is properly terminated.

JDBC provides a powerful and flexible way to interact with databases in Java. It allows you to execute SQL queries, retrieve data, perform updates, and manage transactions within your Java applications.

Here's a simple example that demonstrates the basic steps of establishing a JDBC database connection and executing a query:

```
import java.sql.*;
public class JDBCDemo {
  public static void main(String[] args) {
    // JDBC driver and database URL
    String jdbcDriver = "com.mysql.jdbc.Driver";
    String databaseUrl = "jdbc:mysql://localhost:3306/mydatabase";
    // Database credentials
    String username = "your_username";
    String password = "your_password";

    Connection connection = null;
    Statement statement = null;
    ResultSet resultSet = null;
```

```
        try {
            // Load the JDBC driver
            Class.forName(jdbcDriver);

            // Establish the connection
            connection = DriverManager.getConnection(databaseUrl, username, password);

            // Create a statement
            statement = connection.createStatement();

            // Execute a query
            String sqlQuery = "SELECT * FROM customers";
            resultSet = statement.executeQuery(sqlQuery);

            // Process the result set
            while (resultSet.next()) {
                String name = resultSet.getString("name");
                String email = resultSet.getString("email");
                System.out.println("Name: " + name + ", Email: " + email);
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Close the resources
            try {
                if (resultSet != null) {
                    resultSet.close();
                }
                if (statement != null) {
                    statement.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }}
```

EXTRA LEARNING MATERIALS

Understanding the basics of file handling
https://www.computerhope.com/jargon/f/file.htm

File input and output in Java (reading and writing files in Java)
https://jenkov.com/tutorials/java-io/index.html
https://www.javatpoint.com/java-io
https://zetcode.com/java/readtext/
https://www.marcobehler.com/guides/java-files

How to handle exceptions in Java

https://www.programiz.com/java-programming/exception-handling

https://zetcode.com/java/exception/

Working with packages in Java

https://zetcode.com/java/package/

https://www.edureka.co/blog/packages-in-java/