Advance Educational Activities Pvt. Ltd.

Unit 4: Exceptional Handling & File Handling

4.1.1 What is Exception Handling?

Exception Handling in Java is a powerful mechanism to **handle runtime errors** so the normal flow of the application can be maintained.

What is an Exception?

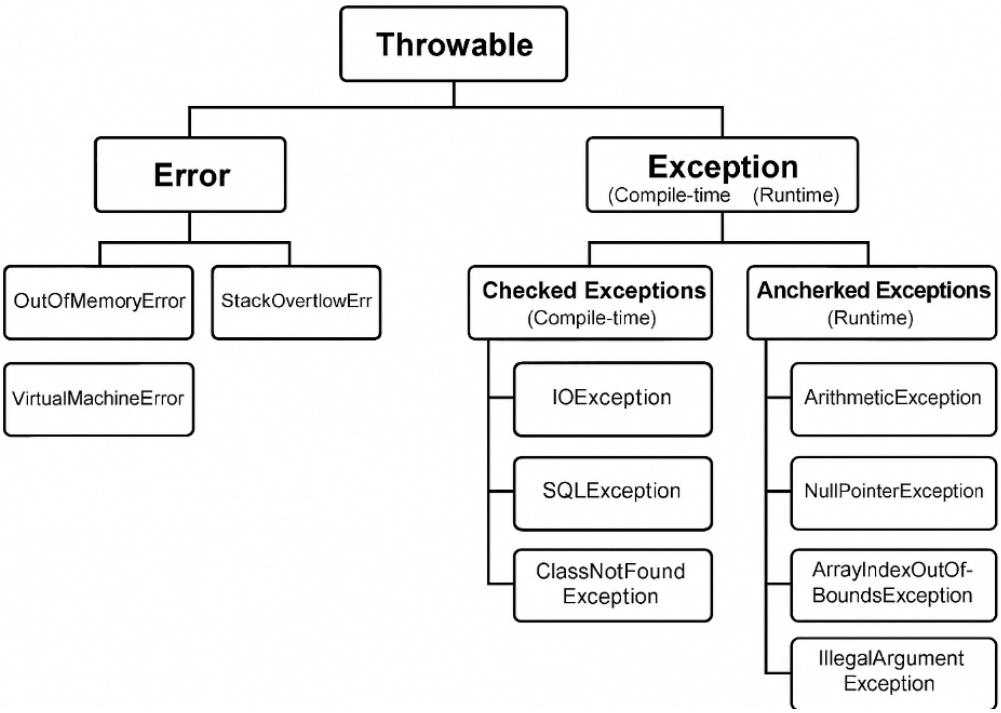
An **exception** is an **event** that occurs during the execution of a program that **disrupts the normal flow** of instructions.

Example: Dividing by zero, accessing an array out of bounds, file not found, etc.

Why Use Exception Handling?

- To **catch and handle errors** gracefully
- Prevent application **crashes**
- Log and fix **unpredictable conditions**
- Keep the **code clean and safe**

Exception Hierarchy



4.1.2 Keywords in Java Exception Handling

KEYWORD	DESCRIPTION
try	Code that might throw an exception
catch	Handles the exception
finally	Executes regardless of exception (cleanup code)
throw	Manually throw an exception
throws	Declare exceptions a method might throw

Syntax:

```
try {
    // risky code
} catch (ExceptionType e) {
    // handling code
} finally {
    // cleanup code
}
```

Example:

```
public class Example {
    public static void main(String[] args) {
        try {
            int a = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("Always executed");
        }
    }
}
```

Output:

```
Cannot divide by zero!
Always executed
```

throw vs throws

throw: Used to manually throw an exception.

throw new ArithmeticException("Divide by zero");

throws: Used to declare exceptions a method might throw.

```
void readFile() throws IOException {
    // code that may throw IOException
}
```

Real-World Analogy

Imagine:

- **try block** = Entering a risky zone (e.g., ATM transaction)
- **catch block** = Security camera catches and reports issue
- **finally block** = Clean-up like removing card from ATM

Best Practices

- Catch **specific exceptions** first, then general
- Avoid empty catch blocks
- Always clean resources in finally or use **try-with-resources**
- Don't overuse exceptions for control flow

4.1.3 Types of Exceptions in Java

Java broadly divides exceptions into **two main types**:

CATEGORY	DESCRIPTION	HANDLED AT?
CHECKED EXCEPTION	Known at **compile-time**	Must be handled
UNCHECKED EXCEPTION	Known at **runtime**	Optional to handle

There's also a third category:

CATEGORY	DESCRIPTION	HANDLED AT?
ERROR	Serious issues, not meant to be caught	System-level issue

1. Checked Exceptions (Compile-Time Exceptions)

These exceptions must be either:

- **caught using try-catch**, or
- **declared using throws**

Examples:

- IOException
- SQLException
- FileNotFoundException
- ClassNotFoundException

Example:

```
import java.io.*;

public class CheckedEx {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("file.txt");  // Checked exception
    }
}
```

2. Unchecked Exceptions (Runtime Exceptions)

These are exceptions that:

- **Do not need to be declared**
- Usually caused by **programming errors** (e.g., logic flaws)

All **subclasses of RuntimeException** are unchecked.

Examples:

- ArithmeticException
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException

Example:

```
public class UncheckedEx {
    public static void main(String[] args) {
        int a = 10 / 0;  // ArithmeticException
    }
}
```

Errors (Not Exceptions)

Errors are serious problems that **cannot be handled by the application**. They indicate **system failure** or **resource issues**.

Examples:

- OutOfMemoryError
- StackOverflowError
- VirtualMachineError

Note:

You should **never try to catch Errors** — they're meant to crash the program or JVM.

Summary Table

TYPE	PARENT CLASS	MUST HANDLE?	EXAMPLES
CHECKED EXCEPTION	Exception	Yes	IOException, SQLException
UNCHECKED EXCEPTION	RuntimeException	❌ No	ArithmeticException, NPE
ERROR	Error	❌ No	OutOfMemoryError, StackOverflow

Best Practices

- Always handle **checked exceptions** properly using try-catch or throws.
- Avoid catching **generic Exception** or **Error** unless absolutely necessary.
- Handle **unchecked exceptions** through proper validation and safe coding.

4.1.4 Multiple catch Block

A **multiple catch block** allows you to handle **different types of exceptions separately** using different handlers for different exception classes — all associated with a single try block.

Syntax:

```
try {
    // Code that might throw multiple exceptions
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
} catch (Exception e) {
    // General exception handler (optional, must be last)
}
```

Real-World Analogy

Imagine you're trying to withdraw money from an ATM:

- If the card is invalid → CardException
- If there's no network → NetworkException
- If ATM is out of cash → CashUnavailableException

Each error needs its own solution.

Example:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 100 / 0;  // Causes ArithmeticException first
        } catch (ArithmeticException ae) {
            System.out.println("Cannot divide by zero.");
        } catch (ArrayIndexOutOfBoundsException ai) {
            System.out.println("Array index is out of bounds.");
        } catch (Exception e) {
            System.out.println("General exception occurred.");
        }
    }
}
```

Output:

```
Cannot divide by zero.
```

Only the **first matching catch block** is executed.

Catch Order Rule

- Catch blocks must go from **most specific to most general**.
- If a **superclass exception** (like Exception) is caught before a **subclass** (like ArithmeticException), you'll get a **compile-time error**.

```
// ❌ Compile-time error
catch (Exception e) { ... }
catch (ArithmeticException ae) { ... }
Correct way:
catch (ArithmeticException ae) { ... }
catch (Exception e) { ... }
```

Java 7+ Feature: Multi-Catch (Single Catch for Multiple Exceptions)

```
try {
    // code
} catch (IOException | SQLException e) {
    System.out.println("IO or SQL error: " + e.getMessage());
}
```

- Use **| (pipe)** to combine exceptions.
- All exceptions **must not be in a parent-child relationship**.

Best Practices

- Catch **specific exceptions** first.
- Use **multi-catch** to reduce redundancy when exceptions need same handling.
- Log or handle each exception meaningfully.
- Don't swallow exceptions silently (catch (Exception e) {} with empty body is bad).

Summary

FEATURE	PURPOSE
MULTIPLE CATCH	Handle different exceptions differently
ORDER MATTERS	Specific → General
MULTI-CATCH (JAVA 7+)	Handle multiple exceptions together

4.1.5 throw and throws in Java

Both throw and throws are used in **exception handling**, but they serve **different purposes**:

KEYWORD	PURPOSE
THROW	To **explicitly throw an exception**
THROWS	To **declare an exception** in method signature

1. throw Keyword

Definition:

The throw keyword is used to **manually throw an exception** (either checked or unchecked).

Syntax:

throw new ExceptionType("Error message");

Example:

```
public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        }
        System.out.println("You can vote!");
    }
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: Not eligible to vote

2. throws Keyword

Definition:

The throws keyword is used in the method signature to **declare** one or more exceptions that the method might throw. This shifts the responsibility to the method caller.

EXCEPTION TYPE	THROWS REQUIRED?	EXAMPLE
checked exception	Yes	IOException, SQLException
unchecked exception	❌ No	NullPointerException, ArithmeticException

Syntax:

```
returnType methodName() throws ExceptionType1, ExceptionType2 {
    // method code
}
```

Example:

```
import java.io.*;

public class ThrowsExample {
    static void readFile() throws IOException {
        FileReader fr = new FileReader("file.txt"); // May throw IOException
    }

    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found!");
        }
    }
}
```

throw vs throws – Comparison Table

FEATURE	THROW	THROWS
PURPOSE	Actually throws an exception	Declares exceptions a method may throw
PLACEMENT	Inside method body	In method signature
NUMBER OF EXCEPTIONS	One at a time	Can declare multiple, comma-separated
USED FOR	Instantiating and throwing exception	Forwarding responsibility to calling method
FOLLOWS BY	Instance of Throwable subclass	List of exception classes

Real-World Analogy

- throw = You manually **raise a red flag** (you throw the error).
- throws = You **warn others** that this method **might throw a red flag**.

Summary

- Use throw to **actually throw** the exception.
- Use throws to **declare** that a method might throw an exception.
- Always **handle checked exceptions** either using try-catch or throws.

4.1.7 Exception Chaining in Java — Complete Explanation

Exception chaining is a powerful concept in Java that allows you to associate one exception with another — making it easier to **track the root cause** of a problem across multiple layers of code.

What Is Exception Chaining?

Exception chaining means **wrapping one exception inside another** so that you can propagate the **original cause** while throwing a **higher-level exception**.

This helps preserve the actual root problem even when re-throwing a new exception.

Why Use Exception Chaining?

- To preserve the **original exception context**
- To **abstract internal details** while providing user-friendly messages
- For better **debugging and logging**
- To maintain **clean exception architecture** in multi-layered applications

Syntax

```
Throwable getCause();
Throwable initCause(Throwable cause);
```

Or use constructors directly:

```
public NewException(String message, Throwable cause);
```

Example: Without Chaining

```
public class NoChaining {
    public static void main(String[] args) {
        try {
            parseNumber("abc");
        } catch (NumberFormatException e) {
            throw new RuntimeException("Failed to parse input.");
        }
    }

    static void parseNumber(String s) {
        Integer.parseInt(s); // Throws NumberFormatException
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Failed to parse input
```

The actual **cause** (NumberFormatException) is lost.

Example: With Exception Chaining

```
public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            parseNumber("abc");
        } catch (NumberFormatException e) {
            throw new RuntimeException("Failed to parse input", e);  // Chaining
        }
    }

    static void parseNumber(String s) {
        Integer.parseInt(s); // Throws NumberFormatException
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Failed to parse input
```

Caused by: java.lang.NumberFormatException: For input string: "abc"

You can now **trace the root cause**!

Real-World Analogy

Imagine a customer order fails because:

1. Payment service failed.
2. Payment failed because database access failed.

Each layer throws a new exception with a user-friendly message but **chains the original cause**, so developers can debug the actual root — **DB failure**.

How to Create a Custom Exception with Chaining

```
class MyCustomException extends Exception {
    public MyCustomException(String message, Throwable cause) {
        super(message, cause);  // Proper chaining
    }
}
```

Usage:

```
try {
    throw new IOException("Disk failure");
} catch (IOException e) {
    throw new MyCustomException("System error occurred", e);
}
```

Best Practices

- Always **chain exceptions** if you're re-throwing at a higher level.
- Avoid hiding the root cause.
- Use getCause() when logging or debugging.
- Custom exceptions should include a constructor that accepts a cause.

Summary Table

FEATURE	DESCRIPTION
PURPOSE	Preserve root cause when rethrowing exceptions
METHOD	Use constructors with Throwable cause
BENEFITS	Easier debugging, cleaner error propagation
COMMON USAGE	Service layers, API abstraction, frameworks

4.2.1 File Handling in Java

File handling allows you to **create**, **read**, **write**, **update**, and **delete** files using the Java I/O (java.io) and NIO (java.nio) packages.

Common Classes in File Handling

CLASS	PURPOSE
File	Represent file/directory
FileReader	Read character files
FileWriter	Write character files
BufferedReader	Efficient reading of text
BufferedWriter	Efficient writing of text
PrintWriter	Convenient file writing with print methods
Scanner	Read text using regex/token patterns

1. Creating and Checking Files with File Class

```
import java.io.File;
import java.io.IOException;

public class CreateFileDemo {
    public static void main(String[] args) {
        try {
            File myFile = new File("example.txt");

            if (myFile.createNewFile()) {
                System.out.println("File created: " + myFile.getName());
            } else {
                System.out.println("File already exists.");
            }

        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

2. Writing to a File

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteToFile {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, this is a file write example.");
            writer.close();
            System.out.println("Successfully written to the file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Reading from a File

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFile {
    public static void main(String[] args) {
```

```
try {
            File file = new File("example.txt");
            Scanner reader = new Scanner(file);
            while (reader.hasNextLine()) {
                String data = reader.nextLine();
                System.out.println(data);
            }
            reader.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

4. Deleting a File

```
import java.io.File;

public class DeleteFile {
    public static void main(String[] args) {
        File file = new File("example.txt");
        if (file.delete()) {
            System.out.println("Deleted the file: " + file.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

4.2.2 BufferedReader (Efficient Reading)

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try {
            BufferedReader br = new BufferedReader(new FileReader("example.txt"));
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4.2.3 PrintWriter (Efficient Writing)

```
import java.io.*;

public class PrintWriterExample {
    public static void main(String[] args) throws IOException {
        PrintWriter pw = new PrintWriter("example.txt");
        pw.println("Line 1");
        pw.println("Line 2");
        pw.close();
        System.out.println("Data written using PrintWriter");
    }
}
```

Best Practices

- Always close streams (close() or try-with-resources).
- Prefer BufferedReader/Writer for large files.
- Use try-with-resources for auto-closing streams (Java 7+).
- Always handle IOException.

Summary Table

OPERATION	CLASS USED
CREATE	File
READ	FileReader, Scanner, BufferedReader
WRITE	FileWriter, PrintWriter, BufferedWriter
DELETE	File

4.2.4 File Handling with .csv file

1. Create and Write a .csv File (Manual – Without Libraries)

```
import java.io.FileWriter;
import java.io.IOException;

public class CsvWrite {
    public static void main(String[] args) {
        String filePath = "data.csv";
        try (FileWriter writer = new FileWriter(filePath)) {
            writer.append("ID,Name,Email\n");
            writer.append("1,John Doe,john@example.com\n");
            writer.append("2,Jane Smith,jane@example.com\n");
            System.out.println("CSV file created and written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Read a .csv File (Manual Read)

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CsvRead {
    public static void main(String[] args) {
        String filePath = "data.csv";
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] values = line.split(",");
                for (String v : values) {
                    System.out.print(v + "\t");
                }
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Delete a .csv File

```
import java.io.File;

public class CsvDelete {
    public static void main(String[] args) {
        File file = new File("data.csv");
        if (file.delete()) {
            System.out.println("CSV file deleted successfully.");
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

Optional: Use OpenCSV (Simplified & Cleaner)

Add Dependency

If you're using Maven:

```
<dependency>
    <groupId>com.opencsv</groupId>
    <artifactId>opencsv</artifactId>
    <version>5.7.1</version>
</dependency>
```

Write with OpenCSV

```
import com.opencsv.CSVWriter;
import java.io.FileWriter;
import java.io.IOException;

public class OpenCsvWrite {
    public static void main(String[] args) {
        try (CSVWriter writer = new CSVWriter(new FileWriter("data.csv"))) {
            String[] header = { "ID", "Name", "Email" };
            String[] record1 = { "1", "John", "john@example.com" };
            String[] record2 = { "2", "Jane", "jane@example.com" };

            writer.writeNext(header);
            writer.writeNext(record1);
            writer.writeNext(record2);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Read with OpenCSV

```
import com.opencsv.CSVReader;
import java.io.FileReader;
import java.io.IOException;

public class OpenCsvRead {
    public static void main(String[] args) {
        try (CSVReader reader = new CSVReader(new FileReader("data.csv"))) {
            String[] line;
            while ((line = reader.readNext()) != null) {
                for (String cell : line) {
                    System.out.print(cell + "\t");
                }
                System.out.println();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Summary Table

TASK	JAVA I/O CLASSES	LIBRARY (OPTIONAL)
CREATE	FileWriter	OpenCSV CSVWriter
READ	BufferedReader	OpenCSV CSVReader
DELETE	File	—

4.2.5 Exception handling in file handling

Common Exceptions in File Handling

EXCEPTION	CAUSE
`FILENOTFOUNDEXCEPTION`	File doesn't exist (when reading)
`IOEXCEPTION`	General I/O error (read/write/close fails)
`SECURITYEXCEPTION`	Access denied due to JVM SecurityManager policy
`NULLPOINTEREXCEPTION`	Stream used without being properly initialized
`EOFEXCEPTION`	Reached end of file unexpectedly during reading

Note: Best Practice: Always Use try-catch-finally or try-with-resources

Example: Handling Exception While Reading a File

```
import java.io.*;

public class ReadFileWithExceptionHandling {
    public static void main(String[] args) {
        BufferedReader br = null;

        try {
            br = new BufferedReader(new FileReader("input.txt"));
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }

        } catch (FileNotFoundException e) {
            System.out.println("⚠️ File not found. Please check the file path.");
        } catch (IOException e) {
            System.out.println("⚠️ An error occurred while reading the file.");
        } finally {
            try {
                if (br != null)
                    br.close();
            } catch (IOException e) {
                System.out.println("⚠️ Failed to close the file properly.");
            }
```

```
        }
    }
}
```

Better Way: Try-With-Resources (Java 7+)

```
import java.io.*;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("⚠ File not found!");
        } catch (IOException e) {
            System.out.println("⚠ Error reading the file.");
        }
    }
}
```

Automatically closes file
Cleaner code
No need for finally

Example: Writing File with Exception Handling

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteFileWithExceptionHandling {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt")) {
            fw.write("File written successfully.");
        } catch (IOException e) {
            System.out.println("⚠ Cannot write to file: " + e.getMessage());
        }
    }
}
```

When to Use throws?

In **method definitions** for file operations in modular programs:

```
public void readFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    // ...
} //The calling method must handle it.
```