**1.Difference between FileReader and BufferedReader?**

BufferReader :- It is used to read characters from any type of character input stream (String ,files ,etc)

Uses Buffer internally to read characters from file .

Buffer speed is faster than file reader.

Much more efficient for reading files .

BufferedReader can be used asingle character at a time as well as a line at a time .

FileReader :- It can be used only for reading files .

Doesn’t use Buffer .Directly reads from the file by accessing the hard drive .

FileReader slower than BufferReader.

It can be read only one character at a time can not read line .

**2.What is try-with-resources?**

In java is a feature introduced in java 7 that allows to automatically close resources (like ,files,sockets ,or database connections) after youre done using them without needing a seprate finally block .

It works with any object that implements the AutoCloseable (or older closeable ) interface .

try (ResourceType resource = new ResourceType()) {

// use the resource

} catch (Exception e) {

// handle exception

}

// resource is automatically closed here

**3.How to handle IOException?**

Handalling IOException in java means writing code to deal with input /output errors (like reading from a missing file .netwoork issues ,or permission problems so your program doesn’t crash unexpectedly.

IoException is a checked exception ,so the compiler forces you to either :

1)Catch it with try-catch ,or

2)Declare it with throws in your method signature.

1)Using try-catch

your wrap the code that may throw IoException inside a try block and handel it in a catch block .

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileReader fr = new FileReader("data.txt");

int c;

while ((c = fr.read()) != -1) {

System.out.print((char) c);

}

fr.close();

} catch (IOException e) {

System.out.println("Error reading the file: " + e.getMessage());

e.printStackTrace(); // optional: print full error details

}

}

}If the file doesn’t exist or can’t be read, the program won’t crash — the catch block runs instead.

Declaring with throws :-

**You can declare that your method might throw an IoException and let the** caller handel it .

3. Best Practices

Use try-with-resources to automatically close files/sockets.

Log the exception details (e.printStackTrace() or logging framework).

Provide a user-friendly message instead of just crashing.

Avoid swallowing exceptions silently (catch (IOException e) {} is bad).

4.What are checked and unchecked exceptions?

1)Checked Exception is a Exception that the compiler checks at compile time .

You must either handel them using try –catch or declare them using throws

They usually represent recoverable problems (things you can handel ).

Superclass:-java.lang.Exception (but not RuntimeException ).

Example :-

IoException (file not found ,read /write errors )

SQLException (database errors )

ClassNotFoundException

FileNotFoundException

import java.io.\*;

public class CheckedExample {

public static void main(String[] args) {

try {

FileReader fr = new FileReader("data.txt"); // may throw FileNotFoundException

} catch (IOException e) {

System.out.println("File not found: " + e.getMessage());

}

}

}

2. Unchecked Exceptions

This Exceptions that the compiler does not checked at compile time .

You are not forced to handel them

They usually represent programming errors (bugs) that could be avoided with better code

Superclass :-java.lang .RuntimeException

Arithmatic Exception (divide by zero)

NullPointerException

ArrayIndexOutOfBoundException

NumberFormatException

public class UncheckedExample {

public static void main(String[] args) {

int result = 10 / 0; // ArithmeticException at runtime

System.out.println(result);

}

}

**5.How does file writing work in Java?**

Java provides several classes for this in the **java.io** package (and also in java.nio for advanced I/O).

**1. Basic Process of Writing a File**

**Open/Create the file** → Java will create a new file if it doesn’t exist.

**Write data** → Convert the data into bytes or characters.

**Flush & Close** → Ensure all data is actually written and resources are released.

**2. Common Ways to Write Files**

**a) Using** FileWriter **(for text)**

import java.io.FileWriter;

import java.io.IOException;

public class FileWriteExample {

public static void main(String[] args) {

try (FileWriter writer = new FileWriter("output.txt")) {

writer.write("Hello, Java File Writing!\n");

writer.write("This will be saved in the file.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

 Writes **characters**.

 Can be wrapped in BufferedWriter for efficiency.

b) Using BufferedWriter (faster for large text)

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class BufferedWriteExample {

public static void main(String[] args) {

try (BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"))) {

bw.write("Line 1");

bw.newLine();

bw.write("Line 2");

} catch (IOException e) {

e.printStackTrace();

}

}

}

c) Using FileOutputStream (for binary data)

import java.io.FileOutputStream;

import java.io.IOException;

public class BinaryWriteExample {

public static void main(String[] args) {

try (FileOutputStream fos = new FileOutputStream("data.bin")) {

byte[] data = {65, 66, 67}; // ASCII for A, B, C

fos.write(data);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Always close the file (or use **try-with-resources** to close automatically).

Writing replaces existing content by default — use **append mode** if you want to add data:

new FileWriter("file.txt", true) // true = append mode

6.What is the difference between append and overwrite mode?

**. Overwrite Mode (default)**

**Behavior:**

Deletes existing file content.

Starts writing from the **beginning** of the file.

**How to use:**

Use file-writing constructors **without** the append flag.

**Example:**

**java**

**Copy code**

**import java.io.FileWriter;**

**import java.io.IOException;**

**public class OverwriteExample {**

**public static void main(String[] args) {**

**try (FileWriter writer = new FileWriter("data.txt")) { // overwrite mode**

**writer.write("Hello, overwrite mode!");**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

2. Append Mode

**Append Mode**

**Behavior:**

Keeps existing file content.

Starts writing **at the end** of the file.

**How to use:**

Pass true as the second argument in the file-writing constructor.

 **Example:**

import java.io.FileWriter;

import java.io.IOException;

public class AppendExample {

public static void main(String[] args) {

try (FileWriter writer = new FileWriter("data.txt", true)) { // append mode

writer.write("\nThis line is appended.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

**7.What is exception propagation?**

**Exception propagation** in Java is the process of passing an exception **up the call stack** from one method to another until it is either **caught** or causes the program to terminate.

Think of it as: If a method doesn’t know how to handle an exception, it “passes the problem” to the method that called it.

**How It Works**

A method throws an exception.

If the method has no try-catch to handle it, the exception moves (**propagates**) to its caller.

This continues up the stack until:

A method **catches** the exception, OR

The exception reaches the **main()** method and the program ends with an error.

public class PropagationExample {

public static void main(String[] args) {

try {

method1();

} catch (ArithmeticException e) {

System.out.println("Exception handled in main: " + e.getMessage());

}

}

static void method1() {

method2();

}

static void method2() {

method3();

}

static void method3() {

int result = 10 / 0; // ArithmeticException occurs here

}

}

8.How to log exceptions?

**Logging exceptions** means recording error details (like message, stack trace, and time) so you can debug issues later without relying only on console output.  
Instead of just using System.out.println() or e.printStackTrace(), we use proper **logging frameworks** or Java’s built-in logging tools.

**. Why Log Exceptions?**

Keeps a **permanent record** (in a file or database).

Helps debug production issues where you can’t see the console.

Allows different **log levels** (INFO, WARN, ERROR).

More professional than printStackTrace().

2. Common Ways to Log Exceptions

a) Using printStackTrace() (basic way)

try {

int result = 10 / 0;

} catch (ArithmeticException e) {

e.printStackTrace(); // prints full error details to console

}

b) Using Java’s Built-in java.util.logging

**import java.util.logging.\*;**

**public class LogExample {**

**private static final Logger logger = Logger.getLogger(LogExample.class.getName());**

**public static void main(String[] args) {**

**try {**

**int result = 10 / 0;**

**} catch (ArithmeticException e) {**

**logger.log(Level.SEVERE, "An exception occurred", e);**

**}**

**}**

**}**

c) Using Apache Log4j (Popular in industry)

**import org.apache.logging.log4j.LogManager;**

**import org.apache.logging.log4j.Logger;**

**public class Log4jExample {**

**private static final Logger logger = LogManager.getLogger(Log4jExample.class);**

**public static void main(String[] args) {**

**try {**

**int result = 10 / 0;**

**} catch (ArithmeticException e) {**

**logger.error("An exception occurred", e);**

**}**

**}**

**}** d) Using SLF4J with Logback (Modern & Recommended)

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**public class Slf4jExample {**

**private static final Logger logger = LoggerFactory.getLogger(Slf4jExample.class);**

**public static void main(String[] args) {**

**try {**

**int result = 10 / 0;**

**} catch (ArithmeticException e) {**

**logger.error("An exception occurred", e);**

**}**

**}**

**}**

**9.What is a stack trace?**

A **stack trace** in Java is a detailed report that shows the sequence of method calls in your program at the exact moment an **exception** occurred.

It helps you **trace back** where the error started and how it traveled through the code.

**Printed automatically** by the JVM when an uncaught exception occurs.

Shows:

**Type of exception** (e.g., NullPointerException, IOException)

**Error message** (optional)

**Method call hierarchy** (stack frames) from the point of failure back to the start.

Printed using:

Automatically (when exception not caught)

Manually using e.printStackTrace().

**Example**

**java**

**Copy code**

**public class StackTraceExample {**

**public static void main(String[] args) {**

**method1();**

**}**

**static void method1() {**

**method2();**

**}**

**static void method2() {**

**int result = 10 / 0; // ArithmeticException**

**}**

**}**

How to Read It

java.lang.ArithmeticException: / by zero → Exception type and reason.

at StackTraceExample.method2(StackTraceExample.java:11) → The exact method and line number where it happened.

The next lines → The methods that called it, going backwards up the stack.

10.When to use finally block?

The **finally block** in Java is used when you want a piece of code to **always run**, no matter what happens — whether an exception is thrown, caught, or not thrown at all.

It’s typically used for **cleanup tasks** like closing files, database connections, or network sockets.

try {

// Code that might throw an exception

} catch (Exception e) {

// Handle exception

} finally {

// Always runs

}

2. When to Use the finally Block

**Files, database connections, sockets** need to be closed no matter what.

FileReader fr = null;

try {

fr = new FileReader("data.txt");

// process file

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (fr != null) fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

**3. Important Notes**

The finally block runs **even if**:

An exception is thrown and caught.

No exception is thrown.

A return statement is in try or catch.

The finally block **will NOT run** if:

The JVM exits (System.exit()).

The thread is killed abruptly.

**Example Showing** finally **Always Runs**

java

Copy code

public class FinallyExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // Exception here

} catch (ArithmeticException e) {

**System.out.println**("Exception caught: " + e.getMessage());

} finally {

**System.out.println**("Finally block executed.");

}

}

}