Exception Handling

1. Explain different types of Errors in Java

Ans1. In Java, errors are generally classified into two categories: **compile-time errors** and **runtime errors**. Each type of error can further be divided into subcategories, and here's a breakdown:

1. Compile-Time Errors

These errors occur when the code is being compiled (before execution). They prevent the code from being successfully compiled into bytecode. Common types include:

a. Syntax Errors

- Occur due to incorrect use of Java syntax (missing semicolons, wrong variable names, unmatched parentheses, etc.).
- Example:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!")
    }
}
```

b. Type Checking Errors

- Happen when there is an attempt to assign a value of an incompatible type.

- Example

```
int a = "Hello";  // Error: incompatible types
```

c. Missing or Incorrect Imports

- If a class or package is used without importing it, or if the import is incorrect, it will lead to compile-time errors.
- Example:

```
import java.util.ArrayList;


public class Main {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList();
   }
}
```

2. Runtime Errors

These errors occur when the program is running. They don't appear during compilation but instead happen during the execution of the program. Common types include:

a. Exceptions

Exceptions are errors that occur during the execution of the program. Java provides a built-in exception handling mechanism using `try-catch` blocks. Types of exceptions include:

- **Checked Exceptions:** These must be either caught or declared to be thrown. Examples include `IOException`, `SQLException`.
- **Unchecked Exceptions (Runtime Exceptions):** These are not checked during compile time and can occur during the execution. Examples include `NullPointerException`,

`ArithmeticException`, `ArrayIndexOutOfBoundsException`.

Example of handling an exception:

```
try {
   int result = 10 / 0;  // Will throw ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Error: Division by zero");
}
```

b. NullPointerException

- This occurs when trying to access a method or field on a `null` reference.
- Example:

```
String str = null;
str.length();  // Will throw NullPointerException
```

c. ArrayIndexOutOfBoundsException

- Occurs when trying to access an array element at an invalid index.
- Example:

```
int[] arr = {1, 2, 3};
System.out.println(arr[5]);  // Will throw ArrayIndexOutOfBoundsException
```

d. ClassCastException

- This happens when trying to cast an object to a subclass that is not actually an instance of

that class.
- Example:

```
Object obj = "Hello";

Integer num = (Integer) obj;  // Will throw ClassCastException
```

e. ArithmeticException

- Occurs during invalid arithmetic operations, like division by zero.
- Example:

```
int result = 10 / 0;  // Will throw ArithmeticException
```

3. Logical Errors

- These are bugs in the program that don't cause crashes but produce incorrect results or behavior. They happen due to mistakes in logic.
- Example:

```
int x = 5;

int y = 10;

System.out.println(x - y);  // You might expect the sum, but it's the difference.
```

4. Errors (JVM Errors)

These are typically more severe and are related to issues with the Java Virtual Machine (JVM) itself. These are usually not recoverable by the program.

- **OutOfMemoryError:** Happens when the JVM runs out of memory.
- **StackOverflowError:** Occurs when the call stack overflows due to deep recursion.

Example:

```
public class Main {
    public static void main(String[] args) {
        main(args);  // This will cause a StackOverflowError
    }
}
```

In summary, Java errors fall into **compile-time errors**, **runtime errors** (exceptions), **logical errors**, and **JVM errors**. Proper error handling using try-catch blocks can manage most runtime errors, and understanding compile-time errors helps catch mistakes early in the development process.

2. What is an Exception in Java

Ans2. **What is an Exception in Java?**

In Java, an **exception** is an **unexpected event** that disrupts the normal flow of a program during runtime. It occurs when the program encounters an **error condition**, such as dividing by zero, accessing an invalid array index, or trying to open a non-existent file.

Exceptions in Java are handled using the **Exception Handling Mechanism**, which allows a program to detect and respond to runtime errors without crashing.

Types of Exceptions in Java

Exceptions in Java are broadly categorized into **Checked Exceptions**, **Unchecked Exceptions**, and **Errors**.

1️⃣ Checked Exceptions (Compile-Time Exceptions)

- These exceptions are checked at compile time.

- If not handled properly, the compiler throws an error.
- These exceptions occur due to **external factors**, such as file handling, database connectivity, or network failures.
- They must be handled using `try-catch` or declared using `throws`.

Examples of Checked Exceptions:

Exception	Cause
`IOException`	Occurs when file operations fail
`SQLException`	Occurs when there's a database-related issue
`ClassNotFoundException`	Thrown when a class is not found

Example of a Checked Exception:

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

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File("nonexistent.txt");
            FileReader fr = new FileReader(file); // May throw IOException
        } catch (IOException e) {
```

```
            System.out.println("File not found!");
        }
    }
}
```

2️⃣ Unchecked Exceptions (Runtime Exceptions)

- These exceptions are **not checked at compile time**.
- They occur due to **logical errors** in the program.
- Unchecked exceptions extend `RuntimeException`.

Examples of Unchecked Exceptions:

Exception	Cause
`NullPointerException`	Accessing a method on a `null` object
`ArithmeticException`	Division by zero
`ArrayIndexOutOfBoundsException`	Accessing an invalid array index

Example of an Unchecked Exception:

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int num = 10 / 0;  // ArithmeticException: Division by zero
    }
```

}

3️⃣ Errors (JVM Errors)

- Errors are **not exceptions**; they are serious issues that occur at the **JVM level**.
- Errors **cannot be handled** in most cases.
- Examples include `OutOfMemoryError` and `StackOverflowError`.

Example of a StackOverflowError (Infinite Recursion):

```
public class StackOverflowExample {
    public static void recursiveMethod() {
        recursiveMethod();  // Infinite recursion leads to StackOverflowError
    }

    public static void main(String[] args) {
        recursiveMethod();
    }
}
```

Exception Handling in Java

Java provides a mechanism to **handle exceptions gracefully** using `try-catch`, `finally`, and `throws` keywords.

✅ 1. Using try-catch

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

```
        try {
            int result = 10 / 0;  // May throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

✅ 2. Using finally Block

The `finally` block executes **whether an exception occurs or not**.

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
        } catch (Exception e) {
            System.out.println("An error occurred.");
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

✅ 3. Using throws Keyword

The `throws` keyword is used to declare exceptions.

```
public class ThrowsExample {
    public static void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero!");
        }
        System.out.println(a / b);
    }

    public static void main(String[] args) {
        divide(10, 0);  // Throws ArithmeticException
    }
}
```

Summary Table

Exception Type	When it Occurs	Examples
Checked Exception	Compile-time	IOException, SQLException
Unchecked Exception	Runtime	NullPointerException, ArithmeticException

Error	JVM-level	StackOverflowError, OutOfMemoryError

3. How can you handle exceptions in Java?Explain with an example

Ans3. **Handling Exceptions in Java**

Java provides a robust mechanism to handle exceptions, ensuring that a program doesn't crash unexpectedly. Exception handling allows us to detect errors at runtime and take appropriate actions to prevent program failure.

Ways to Handle Exceptions in Java

1️⃣ Using try-catch Block

- The `try` block contains the code that may generate an exception.
- The `catch` block catches and handles the exception.

Example: Handling Division by Zero Exception

```
public class TryCatchExample {

    public static void main(String[] args) {

        try {

            int result = 10 / 0;  // May throw ArithmeticException

        } catch (ArithmeticException e) {

            System.out.println("Cannot divide by zero!");

        }
```

```
        System.out.println("Program continues...");
    }
}
```

🔹 **Output:**

Cannot divide by zero!

Program continues...

2️⃣ Using Multiple catch Blocks

- **Java allows multiple `catch` blocks to handle different types of exceptions.**

Example: Handling Multiple Exceptions

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]);  // ArrayIndexOutOfBoundsException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error occurred.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds!");
        } catch (Exception e) {
            System.out.println("Some other error occurred.");
```

```
    }
  }
}
```

Output:

Array index is out of bounds!

3️⃣ Using finally Block

- **The `finally` block always executes whether an exception occurs or not.**
- **It is often used to release resources like file streams, database connections, etc.**

Example: Using finally Block

```
public class FinallyExample {
  public static void main(String[] args) {
    try {
      int data = 10 / 0; // Exception occurs
    } catch (ArithmeticException e) {
      System.out.println("Exception caught: " + e);
    } finally {
      System.out.println("This block always executes.");
    }
  }
}
```

🔹 **Output:**

Exception caught: java.lang.ArithmeticException: / by zero

This block always executes.

4️⃣ Using throws Keyword

- **The `throws` keyword is used to declare exceptions in the method signature.**
- **The calling method must handle the exception.**

Example: Using throws

```
public class ThrowsExample {
    public static void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero!");
        }
        System.out.println("Result: " + (a / b));
    }

    public static void main(String[] args) {
        try {
            divide(10, 0);  // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
```

```
}
```

🔹 **Output:**

```
Error: Cannot divide by zero!
```

5️⃣ Using throw Statement

- The `throw` keyword is used to explicitly throw an exception.

Example: Throwing a Custom Exception

```
public class ThrowExample {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote!");
        }
        System.out.println("Eligible to vote.");
    }

    public static void main(String[] args) {
        try {
            validateAge(16); // Throws exception
        } catch (ArithmeticException e) {
            System.out.println("Exception: " + e.getMessage());
        }
```

```
    }
}
```

Output:

Exception: Not eligible to vote!

4. Why do we need exception handling in Java

Ans4.

1️⃣ Prevents Program Crashes

Without exception handling, runtime errors like division by zero or accessing an invalid index can crash the program. By handling exceptions, we ensure that the program continues execution instead of abruptly stopping.

Example (Without Exception Handling - Program Crashes)

```
public class NoExceptionHandling {
    public static void main(String[] args) {
        int result = 10 / 0;  // ArithmeticException: Division by zero
        System.out.println("This statement will not execute.");
    }
}
```

- **Output:**

Exception in thread "main" java.lang.ArithmeticException: / by zero

Example (With Exception Handling - Program Continues)

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
}
```

- **Output:**

Cannot divide by zero!

Program continues...

2 Helps in Debugging & Error Identification

Exception handling provides clear error messages, making it easier to debug issues.

Example: Exception Message for Debugging

```
public class DebugExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());  // NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Error: Cannot invoke "String.length()" because "str" is null
```

3️⃣ Ensures Program Reliability

Applications, especially large-scale systems (banking, e-commerce, etc.), must not fail unexpectedly. Exception handling makes applications more reliable.

Example: Handling Multiple Exceptions

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

```
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);  // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid array index accessed!");
        }
        System.out.println("Application is running smoothly...");
    }
}
```

🔹 **Output:**

```
Invalid array index accessed!
Application is running smoothly...
```

4️⃣ Helps in Resource Management

Exception handling ensures that resources (files, database connections, network sockets) are properly released even if an error occurs.

Example: Using `finally` to Close a File

```
import java.io.*;

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

```
        FileReader file = null;
        try {
            file = new FileReader("nonexistent.txt");  // FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("File not found!");
        } finally {
            System.out.println("Closing file...");
        }
    }
}
```

- **Output:**

```
File not found!
Closing file...
```

5️⃣ Enables Custom Exception Handling

In Java, we can create custom exceptions to enforce business rules.

Example: Custom Exception for Age Validation

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
```

```
    }
}

public class CustomExceptionExample {
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above!");
        }
        System.out.println("Valid age.");
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (InvalidAgeException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Output:

```
Exception: Age must be 18 or above!
```

5. What is the difference between exception and error in Java

Ans5. Difference Between Exception and Error in Java

In Java, **Exceptions** and **Errors** are both subclasses of `Throwable`, but they serve different purposes.

Feature	Exception	Error
Definition	An exception is an event that occurs during program execution and can be handled.	An error represents a serious system-level issue that is beyond the control of the application.
Type of Issue	Typically caused by logical mistakes in the program (e.g., invalid input, division by zero).	Caused by system failures, resource exhaustion, or JVM-level problems.
Handling	Can be handled using `try-catch`, `finally`, or `throws`.	Generally cannot be handled, and the program may terminate.
Recoverability	Recoverable – The program can continue execution after handling the exception.	Non-recoverable – The program usually crashes.
Causes	Incorrect user input, database connection failure, file not found, etc.	Out of memory, stack overflow, hardware failure, JVM crash, etc.
Hierarchy	Subclass of `java.lang.Exception`.	Subclass of `java.lang.Error`.
Examples	`NullPointerException`, `IOException`, `ArithmeticException`, `SQLException`.	`OutOfMemoryError`, `StackOverflowError`, `VirtualMachineError`.

Code Examples

1️⃣ Exception Example (Handled)

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int num = 10 / 0;  // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception handled: " + e.getMessage());
        }
        System.out.println("Program continues...");
    }
}
```

Output:

Exception handled: / by zero

Program continues...

2️⃣ Error Example (Cannot Be Handled)

```
public class ErrorExample {
    public static void recursiveMethod() {
```

```
        recursiveMethod();  // Infinite recursion leading to StackOverflowError
    }

    public static void main(String[] args) {
        recursiveMethod();
    }
}
```

Output (varies based on system memory):

Exception in thread "main" java.lang.StackOverflowError

6. Name the different types of exceptions in Jav

Ans6. Types of Exceptions in Java

Java exceptions are categorized into Checked Exceptions, Unchecked Exceptions, and Errors.

1️⃣ Checked Exceptions (Compile-time Exceptions)

✔ These exceptions must be handled using `try-catch` or `throws`, otherwise the program won't compile.
✔ Occur due to external factors like file handling, database connection, and network issues.

Examples of Checked Exceptions

Exception	Description

`IOException`	Occurs during input-output operations (e.g., file not found).
`SQLException`	Related to database access errors.
`FileNotFoundException`	File does not exist at the specified path.
`InterruptedException`	A thread is interrupted while waiting, sleeping, or joining.
`ClassNotFoundException`	Class is not found at runtime.
`MalformedURLException`	Invalid URL format.

Example: Handling Checked Exception

```
import java.io.*;


public class CheckedExceptionExample {
   public static void main(String[] args) {
      try {
         FileReader file = new FileReader("nonexistent.txt");  // May throw FileNotFoundException
      } catch (FileNotFoundException e) {
         System.out.println("File not found!");
      }
```

```
    }
}
```

2️⃣ Unchecked Exceptions (Runtime Exceptions)

✔ These exceptions occur at runtime due to programming logic errors.
✔ They don't need to be handled explicitly but can be caught if necessary.

Examples of Unchecked Exceptions

Exception	Description
`NullPointerException`	Trying to access an object that is `null`.
`ArrayIndexOutOfBoundsException`	Accessing an array with an invalid index.
`ArithmeticException`	Division by zero.
`ClassCastException`	Invalid type casting.
`IllegalArgumentException`	Invalid argument passed to a method.
`NumberFormatException`	Converting an invalid string to a number.

Example: Handling Unchecked Exception

```
public class UncheckedExceptionExample {

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

```
        try {
            int num = 10 / 0;  // ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

3️⃣ Errors (Non-recoverable)

✔ Errors are serious system failures that cannot be handled.
✔ They are caused by JVM issues, system failures, or memory overflows.

Examples of Errors

Error	Description
`StackOverflowError`	Infinite recursion causes stack overflow.
`OutOfMemoryError`	Insufficient memory in the heap.
`VirtualMachineError`	JVM crashes due to fatal errors.
`AssertionError`	Failed assertion using `assert`.

Example: StackOverflowError

```
public class ErrorExample {
```

```
    public static void recursiveMethod() {
        recursiveMethod();  // Infinite recursion
    }

    public static void main(String[] args) {
        recursiveMethod();
    }
}
```

7. Can we just use try instead of finally and catch blocks?

Ans7. No, in Java, a `try` block **must** be followed by either a `catch` block or a `finally` block (or both).

Why?

The `try` block alone is **not valid** because Java requires a way to handle or clean up after potential exceptions.

Valid Combinations

✔ **`try` with `catch`** → Handles exceptions
✔ **`try` with `finally`** → Ensures cleanup
✔ **`try` with both `catch` and `finally`** → Handles exceptions and ensures cleanup

---

Examples

1️⃣✅ Valid: try with catch

```
public class TryWithCatch {
    public static void main(String[] args) {
        try {
            int num = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

2️⃣✅ Valid: try with finally

```
public class TryWithFinally {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block.");
        } finally {
            System.out.println("Finally block always executes.");
        }
    }
```

```
}
```

3️⃣❌ Invalid: try without catch or finally

```
public class TryWithoutCatchFinally {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block.");
        }
    }
}
```

🚨 **Compilation Error:**

'try' without 'catch', 'finally' or resource declarations