Exception Handling in Java - A Developer's Lifesaver!

What is Exception Handling?

Exception handling is a mechanism to handle runtime errors gracefully, ensuring the **normal flow of the program** isn't interrupted.

It allows us to manage **unexpected situations** (like division by zero, null references, file not found, etc.) without crashing the application.

Types of Exceptions in Java

Java exceptions are categorized into two main types:

- **Checked Exceptions**
  These are exceptions that are checked at compile time. It means if a method is likely to result in one of these exceptions and does not handle it, the program will not compile. Checked exceptions must be either caught in a try-catch block or declared in the method signature using throws. Examples of checked exceptions include IOException and SQLException.
  - Checked at **compile-time**
  - Must be handled using try-catch or declared with throws
  - Example: IOException, SQLException

- **Unchecked Exceptions**
  These are not checked at compile time, which means the compiler does not require methods to catch or declare these exceptions. Unchecked exceptions usually indicate programming errors such as logic mistakes (e.g., NullPointerException, IndexOutOfBoundsException). They are subclasses of RuntimeException.
  - Checked at **runtime**
  - Usually caused by logical errors or improper use of API
  - Example: NullPointerException, ArrayIndexOutOfBoundsException

try-catch-finally in Java

What is it?

try-catch-finally is used to handle exceptions gracefully by wrapping the **risky code** in a try block, **handling the exception** in catch, and **executing cleanup code** in finally.

Syntax:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Handle the exception
} finally {
    // Code that always executes (cleanup logic)
}
```

Key Concepts:

- try: Block of code where exceptions **might occur**.
- catch: Handles the **specific exception** thrown in the try block.
- finally: Always executed whether an exception is thrown or not used to **close resources** like files, DB connections, etc.

Example:

```java
public class TryCatchFinallyDemo {
    public static void main(String[] args) {
        try {
            int a = 10, b = 0;
            int result = a / b;  // This will throw ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("Cleanup done: This block always executes.");
        }
    }
}
//Exception caught: / by zero
//Clenup done: This block always executes.
```

When to use finally?

- Closing file streams
- Releasing database connections
- Unlocking resources
- Logging end of operation messages

Sure! Here's the revised version of the throw vs throws explanation with proper examples and their **expected outputs** inside the code blocks:

throw vs throws

throw

- Used to **explicitly throw an exception** from a method or any block of code.
- It only throws **one exception** at a time.

```java
public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;

        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 18 years old.");
        }

        System.out.println("Access granted - You are old enough!");
    }
}

//Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
//  at ThrowExample.main(ThrowExample.java:6)
```

throws

- Used in **method signature** to declare exceptions a method might throw.
- It passes the responsibility of handling the exception to the **caller method**.

```java
import java.io.IOException;

public class ThrowsExample {

    public static void checkFile() throws IOException {
        throw new IOException("File not found");
    }

    public static void main(String[] args) {
        try {
            checkFile();
        } catch (IOException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
//caught exception: File not found
```

Summary Table

Keyword	Purpose	Used in	Type
throw	To explicitly throw an exception	Inside method block	Statement
throws	To declare possible exceptions	In method signature	Declaration

Creating Custom Exceptions in Java

Sometimes, predefined exceptions are not enough. In such cases, you can create your own exception by extending the Exception or RuntimeException class.

Example: Custom Checked Exception

```java
class AgeTooLowException extends Exception {
    public AgeTooLowException(String message) {
        super(message);
    }
}

public class CustomCheckedExceptionExample {

    public static void validateAge(int age) throws AgeTooLowException {
        if (age < 18) {
            throw new AgeTooLowException("Age must be at least 18.");
        }
        System.out.println("Valid age!");
    }

    public static void main(String[] args) {
        try {
            validateAge(16);
        } catch (AgeTooLowException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
//Caught exce\ption: Age must be at least 18.
```

Example: Custom Unchecked Exception

```java
class InvalidInputException extends RuntimeException {
    public InvalidInputException(String message) {
        super(message);
    }
}

public class CustomUncheckedExceptionExample {

    public static void processInput(int number) {
        if (number <= 0) {
            throw new InvalidInputException("Number must be greater than zero.");
        }
        System.out.println("Processing number: " + number);
    }

    public static void main(String[] args) {
        processInput(-5);
    }
}
//Exception in thread "main" InvalidInputException: Number must be greater than zero.
//  at CustomUncheckedExceptionExample.processInput(CustomUncheckedExceptionExample.java:7)
//  at CustomUncheckedExceptionExample.main(CustomUncheckedExceptionExample.java:12)
```

Checked vs Unchecked Exceptions

Checked Exceptions

Definition:

Checked exceptions are the exceptions that are checked at **compile-time**. These exceptions must be either **caught using try-catch** or **declared using throws** in the method signature.

Examples: IOException, SQLException, FileNotFoundException

Use case: External resources like files, databases, or network connections.

Example: Checked Exception

```java
import java.io.*;

public class CheckedExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); // File might not exist
            BufferedReader fileInput = new BufferedReader(file);
            System.out.println(fileInput.readLine());
            fileInput.close();
        } catch (IOException e) {
            System.out.println("Caught Checked Exception: " + e.getMessage());
        }
    }
}
//Caught Checked Exception: data.txt (No such file or directory)
```

Unchecked Exceptions

Definition:

Unchecked exceptions are **not checked at compile-time**, but they occur during **runtime**. These are usually programming bugs like logic errors or improper use of APIs.

Examples: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

Use case: Typically for programming errors or improper object state.

Example: Unchecked Exception

```java
public class UncheckedExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        System.out.println(numbers[5]);  // Accessing invalid index
    }
}
//Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for
length 3
```

Summary Table

Feature	Checked Exception	Unchecked Exception
Checked at	Compile-time	Runtime
Must handle with	try-catch or throws	Optional handling
Common examples	IOException, SQLException	NullPointerException, ArithmeticException
Parent class	Exception	RuntimeException

Multiple Catch Blocks

In Java, you can use **multiple catch blocks** to handle **different types of exceptions** that may occur in a single try block. The catch blocks are evaluated **top to bottom**, and the **first matching catch block** is executed.

Syntax:

```java
try {
    // Code that may throw more than one type of exception
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
} catch (ExceptionType3 e3) {
    // Handle ExceptionType3
}
```

Note: Order matters always catch **more specific exceptions first**, followed by more generic ones like Exception.

Example:

```java
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 10 / 0;  // This will cause ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception: " +
e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out of Bounds: " +
e.getMessage());
        } catch (Exception e) {
            System.out.println("General Exception: " +
e.getMessage());
        }
    }
}
//Arithmetic Exception: / by zero
```

Key Points:

- You **cannot** catch a parent exception before its child (e.g., Exception before ArithmeticException).
- Java executes **only the first matching catch block**.
- Great for cleanly handling multiple possible failure points in one block of code.

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

Try-with-resources is a Java language feature introduced in **Java 7** (so not new to Java 8, but still very important) that makes managing resources (like files, streams, database connections) easier and safer.

It ensures that **resources are automatically closed** after use, without needing a separate finally block.

Why use try-with-resources?

In traditional try-catch-finally, you have to close resources manually in the finally block:

```java
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("file.txt"));
    String line = br.readLine();
    System.out.println(line);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (br != null) {
        try {
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This is verbose and error prone (you might forget to close, or cause resource leaks).

How try-with-resources works

Instead, with **try-with-resources**, you declare the resource inside the parentheses of the try statement:

```java
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line = br.readLine();
    System.out.println(line);
} catch (IOException e) {
    e.printStackTrace();
}
// No finally block needed - br is closed automatically!
```

- The resource (br) **must implement AutoCloseable interface** (which BufferedReader and many other resources do).
- The JVM automatically calls br.close() at the end of the try block, even if an exception is thrown.
- This eliminates boilerplate and reduces bugs.

Multiple resources

You can also declare **multiple resources** separated by semicolons:

```java
try (
    BufferedReader br = new BufferedReader(new FileReader("file1.txt"));
    BufferedWriter bw = new BufferedWriter(new FileWriter("file2.txt"))
) {
    String line;
    while ((line = br.readLine()) != null) {
        bw.write(line);
        bw.newLine();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Key points:

- Resources declared in try-with-resources are **closed automatically and in reverse order of declaration**.
- You don't need to write finally to close resources.
- The resource type must implement AutoCloseable (which extends Closeable).

Summary

Feature	Traditional try-finally	Try-with-resources
Closing resources	Manual close in finally	Automatic close at block end
Code verbosity	Verbose	Concise
Exception safety	Easy to miss closing, leaks	Safer, no leaks

Best Practices in Java Exception Handling

Avoid Empty catch Blocks:

Don't swallow exceptions silently. It hides bugs and makes debugging hard.

```
try {
    // risky code
} catch (IOException e) {
    // BAD practice
}

```

Instead you can write:

```
catch (IOException e) {
    e.printStackTrace(); // Or log the error
}
```

Always Log Exceptions:

Use a proper logging framework like Log4j, SLF4J, or java.util.logging.

```
Logger logger = Logger.getLogger(MyClass.class.getName());
catch (Exception e) {
    logger.severe("Something went wrong: " + e.getMessage());
}
```

Catch Specific Exceptions First:

Always catch the most specific exceptions before general ones.

```
try {
    // code
} catch (FileNotFoundException e) {
    // handle specific case
} catch (IOException e) {
    // handle general I/O
}
```

Use finally for Cleanup:

Always close resources like streams or DB connections in the finally block or use try-with-resources.

```java
try {
    FileReader reader = new FileReader("file.txt");
    // read file
} catch (IOException e) {
    e.printStackTrace();
} finally {
    reader.close(); // ensure cleanup
}
```

Use Custom Exceptions for Business Logic:

Helps to separate technical vs business errors.

```java
public class InvalidOrderException extends Exception {
    public InvalidOrderException(String msg) {
        super(msg);
    }
}
```

Don't Use Exceptions for Flow Control:

Avoid writing logic like this

```java
try {
    int x = Integer.parseInt("abc"); // BAD: avoid relying on exception to control flow
} catch (NumberFormatException e) {
    x = 0;
}
```

Rethrow Exceptions When Needed:

Don't suppress important errors pass them on when appropriate.

```java
catch (IOException e) {
    throw new RuntimeException("Failed to read file", e);
}
```

Some handy Interview Q&A with Examples

1. What is the difference between throw and throws in Java?

Answer:

- throw is used to explicitly throw an exception.
- throws is used in the method signature to declare exceptions that might be thrown.

Example:

```java
public void checkAge(int age) throws IllegalArgumentException {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18 or older");
    }
}
```

2. What is the difference between Checked and Unchecked exceptions?

Answer:

Checked Exception	Unchecked Exception
Checked at compile time	Checked at runtime
Must be handled explicitly	Optional handling
e.g., IOException, SQLException	e.g., NullPointerException, ArithmeticException

Example of Checked Exception:

```
public void readFile() throws IOException {
   FileReader reader = new FileReader("file.txt");
}
```

Example of Unchecked Exception:

```
public void divide(int a, int b) {
   int result = a / b; // May throw ArithmeticException if b == 0
}
```

3. What is a finally block? When does it not execute?

Answer:

finally block always executes after try-catch, even if an exception is not caught.

Only time it may not execute:

- When JVM exits via System.exit(0)
- When a fatal error (like segmentation fault or fatal JVM crash) occurs

Example:

```java
try {
    int x = 5 / 0;
} catch (ArithmeticException e) {
    System.out.println("Caught Exception");
} finally {
    System.out.println("This will always execute");
}
```

4. Can you catch multiple exceptions in one catch block?

Answer:

Yes, from Java 7 onwards using multi-catch (| operator):

```java
try {
    // code that may throw multiple exceptions
} catch (IOException | SQLException ex) {
    ex.printStackTrace();
}
```

5. What is exception propagation?

Answer:

If an exception is not caught in the current method, it propagates to the calling method.

Example:

```java
public void method1() {
    method2();
}
public void method2() {
    int a = 10 / 0; // Exception not handled here
}
//method1() will receive the ArithmeticException thrown from
method2().
```

6. What are custom exceptions?

Answer:

Custom exceptions are user-defined exceptions that extend Exception or RuntimeException.

Example:

```java
public class InvalidInputException extends Exception {
    public InvalidInputException(String msg) {
        super(msg);
    }
}
```

7. Can a finally block override a return value from the try or catch block?

Answer:

Yes, if you return in the finally, it **overrides** other returns, which is a bad practice.

Example:

```java
public int test() {
    try {
        return 1;
    } finally {
        return 2; // Overrides the return from try
    }
}
//2
```

8. What's the difference between Error and Exception?

Answer:

- Error is a serious problem and is not meant to be handled (e.g., OutOfMemoryError)
- Exception is meant to be caught and handled in code.

9. What is try-with-resources?

Answer:

Introduced in Java 7, it automatically closes resources like files, sockets, etc.

Example:

```java
try (BufferedReader br = new BufferedReader(new
FileReader("file.txt"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

10. What is exception chaining in Java?

Answer:

When one exception causes another, we use exception chaining to keep the root cause.

Example:

```java
try {
    throw new IOException("File not found");
} catch (IOException e) {
    throw new RuntimeException("Unable to process file", e);
}
```

11. Can a try block exist without a catch?

Answer:

Yes, only if there is a finally block.

```java
try {
} finally {
}
```

12. Is it good to catch Exception or Throwable?

Answer:

No, it's considered bad practice to catch Throwable or generic Exception, as it may catch unexpected and unrecoverable errors.

13. Can we create a catch block for multiple exceptions with different handling logic?

Answer:

No, for different logic use separate catch blocks.

14. Can you throw an exception from static block or constructor?

Answer:

- Static block: Only unchecked exceptions.
- Constructor: Both checked and unchecked, but must be declared using throws.

15. What happens if an exception is not handled?

Answer:

It causes **abnormal termination** of the program and prints the **stack trace**.

Final Tip for Interviews, always highlight that:

- You follow best practices (never leave empty catch blocks)
- You log exceptions with a proper logging framework
- You use custom exceptions where appropriate