Exception Handling

What is an Exception in Java?

An **exception** is an **unwanted or unexpected event** that occurs during the execution of a program, which **disrupts the normal flow** of the program's instructions.

It typically occurs due to:

- Invalid user input
- File not found
- Network issues
- Arithmetic errors (e.g., division by zero)
- Accessing null or out-of-bound objects, etc.

Java provides a robust mechanism to **detect**, **handle**, and **recover** from such situations using **exception handling**.

□ Common Real-Life Examples:

Scenario	Type of Exception
Dividing a number by zero	ArithmeticException
Accessing an invalid array index	ArrayIndexOutOfBoundsException
Opening a non-existent file	FileNotFoundException
Converting a string to number with letters	NumberFormatException
Calling method on null object	NullPointerException

What is Exception Handling?

Exception Handling is a mechanism in Java to:

1. **Detect (throw)** runtime errors (exceptions)
2. **Handle (catch)** those errors gracefully
3. Ensure the program doesn't crash unexpectedly

□ Goals of Exception Handling:

- Maintain **normal program flow** after handling errors.
- Improve **readability** and **robustness**.
- Separate error-handling logic from normal business logic.

Why Use Exception Handling?

□ Avoid abrupt termination
□ Provide meaningful error messages
□ Improve maintainability
□ Handle different types of errors differently
□ Perform cleanup activities (e.g., closing files or DB connections)

Exception Hierarchy in Java

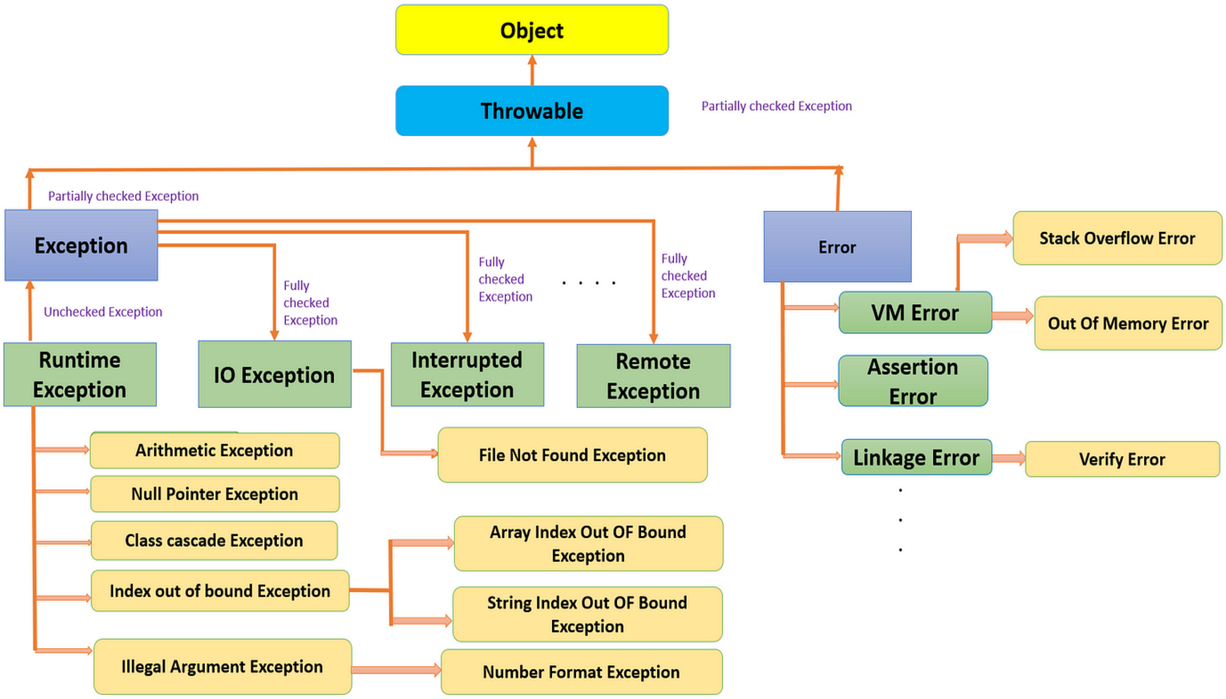

Fig. Exception Hierarchy in Java ~ by Deepti Swain

Types of Exceptions in Java

In Java, exceptions are categorized into two main types:

1. Checked Exceptions (Compile-Time Exceptions)

Checked exceptions are **checked at compile-time**. The compiler ensures that these exceptions are either handled using try-catch or declared using throws.
These are generally caused by external factors (like I/O, database, file operations, etc.) and are **recoverable**.

➤ a. Fully Checked Exceptions

These are exceptions where **all derived classes** are also checked exceptions.

Examples:

- IOException
- SQLException
- ClassNotFoundException

```
import java.io.*;
class Test {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("file.txt"); // Checked Exception
    }
}
```

➤ b. Partially Checked Exceptions

These are exceptions where **some child classes are unchecked**.

Example:

- Exception class is partially checked because:
  - IOException (child) → Checked
  - RuntimeException (child) → Unchecked

```
// 'Exception' is the parent of both checked and unchecked exceptions
```

---

2. Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions are **not checked at compile-time**. These typically occur due to **programming logic errors** and are **not recoverable** in many cases.

Examples:

- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- ClassCastException

```
class Test {
   public static void main(String[] args) {
      int a = 5 / 0; // ArithmeticException
   }
}
```

How Exceptions Work in Java to Show Messages in Terminal

When an exception occurs in Java during runtime, the Java Virtual Machine (JVM) performs the following steps:

Exception Flow: Step-by-Step

1. **Exception is Thrown**
   When an abnormal condition arises (e.g., divide by zero), Java **creates an Exception object** and **throws** it.
2. **JVM Looks for a Handler**
   The JVM checks if the exception is caught using a try-catch block in the current method.
3. **If Caught:**
   - The control is transferred to the matching catch block.
   - A message is displayed using methods like:
     - e.printStackTrace() → prints stack trace with exception
     - e.getMessage() → prints only the message
     - e.toString() → prints exception class and message

4. **If Not Caught:**
    - The exception **propagates up the call stack**.
    - If no handler is found, **JVM terminates the program** and **prints the error message** on the terminal.

How to Handle Exceptions in Java

Java provides a built-in mechanism using try, catch, and finally blocks to **gracefully handle exceptions**, preventing program crashes and ensuring clean error handling.

1.try-catch Block

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

Example:

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

□ **Flow:**

- If no exception occurs → catch is skipped.
- If exception occurs → Control jumps to the matching catch block.
- If no matching catch is found → JVM terminates the program.

2.finally Block

The finally block is **always executed** after try and catch, whether an exception is thrown or not. It's mainly used for **cleanup operations** like:

- Closing files
- Releasing DB connections
- Closing network sockets

Syntax:

```
try {
    // risky code
} catch (Exception e) {
    // handling code
} finally {
    // cleanup code - always executed
}
```

Example:

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int a = 5 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Handled: " + e);
        } finally {
            System.out.println("Finally block executed.");
        }
    }
}
```

□ **Output:**

```
Handled: java.lang.ArithmeticException: / by zero
Finally block executed.
```

Multiple Catch Blocks

Java allows multiple catch blocks to handle different exception types separately.

```
try {
    int[] arr = new int[3];
    arr[4] = 10;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic Exception");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array Index Out of Bounds");
} catch (Exception e) {
    System.out.println("General Exception");
}
```

Best Practices

- Always catch **specific exceptions first**, then general Exception.
- Use finally to close files/resources.
- Avoid empty catch blocks.
- Use try-with-resources for AutoCloseable resources (like FileReader).

Exception Mismatch in Java

What is Exception Mismatch?

Exception Mismatch occurs when:

- The catch block is written for a **different exception type** than the one actually thrown in the try block.
- As a result, the exception is **not caught**, and the program **terminates abnormally**, showing a stack trace.

□ Example of Exception Mismatch

```
public class MismatchExample {
    public static void main(String[] args) {
        try {
            int a = 10 / 0;  // Throws ArithmeticException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught an array exception");
        }
    }
}
```

□ Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
            at MismatchExample.main(MismatchExample.java:4)
```

□ The catch block doesn’t match the thrown exception (ArithmeticException vs ArrayIndexOutOfBoundsException), so the JVM **does not enter** the catch block and the program crashes.

□ Key Point

The **type of exception thrown** in the try block must **match** the catch block's exception type.
Otherwise, the catch block is **skipped**, and the program will **crash at runtime**.

throws Keyword in Java

What is throws?

The throws keyword in Java is used to **declare exceptions** that a method **might throw** but **does not handle** itself. It tells the compiler and the caller of the method that an exception **might occur**, and it is their responsibility to handle it.

□ Syntax

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

□ Purpose of throws

- To inform the **caller of the method** about possible exceptions.
- To **avoid writing try-catch** inside the method if the method itself won’t handle it.
- To delegate the **exception handling responsibility**.

Example with throws

```
import java.io.*;

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

    public static void main(String[] args) {
        try {
            readFile(); // Caller must handle it
        } catch (IOException e) {
            System.out.println("Handled IOException: " + e.getMessage());
        }
    }
}
```

□ Explanation:

- readFile() might throw IOException, so it declares it using throws.
- The main() method handles the exception using a try-catch block.

□ When to Use throws?

Situation	Use throws
You don’t want to handle checked exceptions in the current method	Yes
You want the **caller** to handle it	Yes
You are handling the exception using try-catch in the same method	No need for throws

---

Important Rules

- You can declare **multiple exceptions** using commas:

  ```
  void method() throws IOException, SQLException { }
  ```

- You **can only declare checked exceptions** using throws.
- Declaring unchecked exceptions (like NullPointerException) is **optional**.

throw Keyword in Java

What is throw?

The throw keyword in Java is used to **manually throw an exception** — either built-in or custom — during the execution of a program.

Unlike throws (used in method declaration), throw is used inside a method or block to **actually throw an exception object**.

□ Syntax

```
throw new ExceptionType("Error message");
```

- Only **one exception** can be thrown using a throw statement.

- The exception must be an instance of Throwable (or its subclass like Exception, RuntimeException).

Example: Throwing Built-in Exception

```
public class ThrowExample {
    public static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be 18 or
older.");
        } else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        checkAge(15);
    }
}
```

□ **Output:**

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You
must be 18 or older.
            at ThrowExample.checkAge(4)
            at ThrowExample.main(10)
```

Example: Throwing Checked Exception

```
import java.io.*;

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

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

□ Difference Between throw and throws

Feature	throw	throws
Purpose	Used to **actually throw** an exception	Used to **declare** possible exceptions
Used in	Method body	Method signature
Number of exceptions	Only one	Can declare multiple (comma-separated)
Requires object	Yes (throw new Exception())	No

Common Use Cases for throw

- Input validation
- Custom business rule enforcement
- Explicit error signaling

1. Custom Exceptions in Java

What are Custom Exceptions?

Custom exceptions are **user-defined exceptions** created by extending:

- Exception → for **checked exceptions**
- RuntimeException → for **unchecked exceptions**

They are useful for:

- Application-specific rules
- Making exception messages meaningful

Syntax

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

---

2. Exception Propagation

What is Exception Propagation?

When an exception is **not caught in the current method**, it is **automatically forwarded to the caller** method. This continues up the call stack until it's handled or the JVM terminates the program.

Example

```
public class Propagation {
    static void a() {
        int x = 10 / 0; // Exception here
    }

    static void b() {
        a(); // Not handled
    }

    static void c() {
        try {
            b();
        } catch (ArithmeticException e) {
            System.out.println("Exception handled in c()");
        }
    }

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

3. Try-With-Resources (Java 7+)

Purpose

Used to **automatically close resources** like files, streams, sockets after use.

Requirements

- Resource must implement AutoCloseable (e.g., FileReader, BufferedReader)

Example

```
import java.io.*;

public class ResourceExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

4. Commonly Used Exception Classes

Exception	Type	Occurs When
ArithmeticException	Unchecked	Divide by zero
NullPointerException	Unchecked	Accessing null object
ArrayIndexOutOfBoundsException	Unchecked	Invalid array index
NumberFormatException	Unchecked	Parsing string to number
ClassCastException	Unchecked	Invalid type casting
IOException	Checked	Input/Output failures
FileNotFoundException	Checked	File doesn't exist

Exception	Type	Occurs When
SQLException	Checked	DB access errors

---

5. Multiple Catch Blocks & Catch Hierarchy

Use Case

When a try block can throw multiple exceptions, **you can write multiple catch blocks** to handle them individually.

□ Rule

Always write **child exceptions first**, then the **parent (Exception) last**.

Example

```
try {
    int[] arr = new int[3];
    arr[5] = 10;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic Exception");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array Index Error");
} catch (Exception e) {
    System.out.println("General Exception");
}
```

---

6. Nested try-catch Block

Use Case

Used when one try-catch block is inside another — often required in complex programs or different error-prone sections.

Example

```
public class NestedTry {
    public static void main(String[] args) {
        try {
            try {
                int a = 10 / 0;
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: " + e);
            }

            int[] arr = new int[2];
            arr[3] = 10;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: " + e);
        }
    }
}
```

---

7. Chained Exceptions

What is Chaining?

You can associate one exception with another using constructors like:

Exception(String message, Throwable cause)

This helps **preserve the root cause** of an exception during propagation.

Example

```
public class ChainedException {
    public static void main(String[] args) {
        try {
            NullPointerException npe = new NullPointerException("Root Cause");
            throw new Exception("Top Level Exception", npe);
        } catch (Exception e) {
            System.out.println("Caught: " + e);
            System.out.println("Cause: " + e.getCause());
        }
    }
}
```