# Java Exception Handling

**Presenter**:Rama Shanker

**Date**: 24/01/2025

## What is Exception Handling?

**Definition**: Mechanism to handle runtime errors to maintain the flow of a program.

**Common Errors:**

- Division by zero (ArithmeticException)
- Null references (NullPointerException)
- Array out of bounds (ArrayIndexOutOfBoundsException)

**Objective**: Prevent program crashes and ensure robust application performance.

## Try-Catch Blocks

Purpose: To catch and handle exceptions.

**Flow**:

try: Contains code that might throw an exception.

catch: Handles specific exceptions.

Optional finally: Executes regardless of exceptions.

**Syntax**

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

## Try-Catch Example

**Code Example:**

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

Output: Cannot divide by zero!

## Best Practices for Try-Catch Blocks

Catch **specific exceptions** rather than generic ones.

Avoid empty catch blocks.

Keep the try block concise and focused.

Use **logging** for debugging and tracking exceptions.

## Custom Exceptions

**What are Custom Exceptions?**

- User-defined exceptions to handle application-specific errors.
- Extends the Exception class (checked) or RuntimeException class (unchecked).

**Why Use Custom Exceptions?**

- Improves code clarity and maintainability.
- Allows meaningful error messages for business logic.

## How to Create a Custom Exception

**Steps**:

Extend Exception or RuntimeException.

Define constructors to pass messages or causes.

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

# Custom Exception Example

**Code Example**

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

public class TestCustomException {
    public static void validateAge(int age) throws AgeValidationException {
        if (age < 18) {
            throw new AgeValidationException("Age must be 18 or older.");
        }
    }

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

## Advantages of Custom Exceptions

Clear communication of specific error scenarios.

Enables detailed error logging.

Simplifies debugging and error handling.

## Exception Handling Best Practices

Don’t use exceptions for control flow.

Always clean up resources using finally or try-with-resources.

Avoid catching Exception or Throwable directly.

Document custom exceptions properly.

## Summary

**Try-Catch Blocks**: Essential for handling exceptions gracefully.

**Custom Exceptions**: Provide meaningful, domain-specific error messages.

Exception handling ensures robust and error-resilient applications.