Exception Handling in Java

Object-Oriented Programming in Java

PT821: Object-Oriented Programming

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

What is an Exception?

Definition

An **exception** is an event that disrupts the normal flow of a program during execution. It is an object that represents an error or unexpected condition.

Common Causes:

- Dividing a number by zero
- Accessing a null object
- Array index out of bounds
- File not found
- Invalid user input
- Network connection failure

Without Exception Handling

If an exception is not handled, the program **crashes** and prints an ugly error message to the user.

With Exception Handling

The program **gracefully recovers** and continues or exits cleanly.

A Program That Crashes

Code without exception handling:

```
public class NoCatch {
  public static void main(String[] args) {
    int a = 10;
    int b = 0;
    // This will CRASH the program!
    int result = a / b;
    System.out.println(result);
  }
}
```

Output (Program Crash):

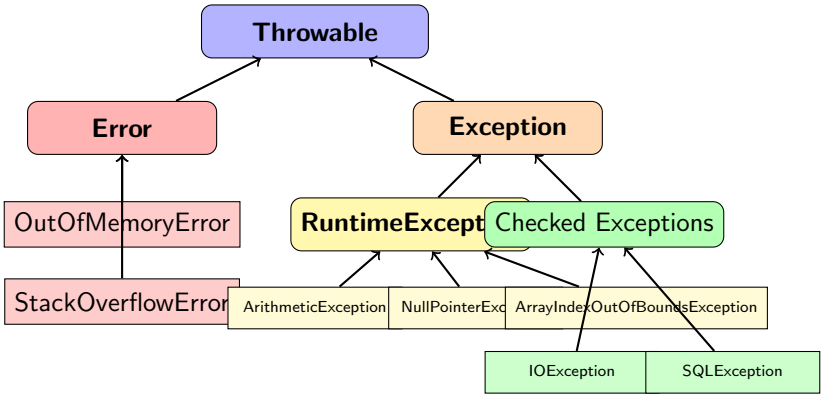
Error Message

```
Exception in thread "main"
java.lang.ArithmeticException:
  / by zero
  at NoCatch.main(NoCatch.java:6)
```

Problem

The program terminates abruptly. The user sees a confusing error.

Java Exception Hierarchy



Checked vs. Unchecked Exceptions

Checked Exceptions

- Checked at **compile time**
- Must be handled or declared
- Extend `Exception` (but not `RuntimeException`)
- Examples:
  - `IOException`
  - `FileNotFoundException`
  - `SQLException`

Unchecked Exceptions

- Checked at **runtime**
- Handling is optional
- Extend `RuntimeException`
- Examples:
  - `ArithmeticException`
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `NumberFormatException`

Errors

Errors (e.g., `OutOfMemoryError`) represent serious problems that a program should *not* try to catch.

The try-catch Block

Syntax

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

Example:

```
public class SafeDivide {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        try {
            int result = a / b;  // May throw ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
        }
        System.out.println("Program continues normally.");
```

The finally Block

Definition

The **finally** block *always* executes, whether or not an exception occurred. Used for cleanup code (closing files, database connections, etc.).

```
public class FinallyDemo {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught: " + e.getMessage());
        } finally {
            // This ALWAYS runs
            System.out.println("Finally block executed.");
            System.out.println("Cleanup done.");
        }
    }
}
```

Output

Multiple catch Blocks

Key Rule

You can have multiple `catch` blocks. Java matches the **first** compatible catch block. Always catch *more specific* exceptions before *more general* ones.

```
public class MultiCatch {
    public static void main(String[] args) {
        try {
            String text = null;
            int[] arr = new int[3];
            arr[10] = 5;                    // ArrayIndexOutOfBoundsException
            System.out.println(text.length()); // NullPointerException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array error: " + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("Null pointer error: " + e.getMessage());
        } catch (Exception e) {
            // Catches any other exception
            System.out.println("General error: " + e.getMessage());
        } finally {
            System.out.println("Done.");
        }
```

Multi-catch (Java 7+)

Multi-catch Syntax

Since Java 7, you can catch multiple exception types in a single `catch` block using `|`.

```
public class MultiCatchModern {
    public static void main(String[] args) {
        try {
            String input = "abc";
            int number = Integer.parseInt(input); // NumberFormatException
            int[] arr = new int[number];
            System.out.println(arr[100]);          // ArrayIndexOutOfBoundsException
        } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
            // Handle both exception types the same way
            System.out.println("Input error: " + e.getMessage());
        }
    }
}
```

Benefit

Avoids duplicate code when handling multiple exceptions the same way.

The `throw` Keyword

Definition

The **throw** keyword is used to *explicitly* throw an exception from your code.

```
public class ThrowDemo {
    static void validateAge(int age) {
        if (age < 18) {
            // Manually throw an exception
            throw new IllegalArgumentException(
                "Age must be 18 or older. Given: " + age
            );
        }
        System.out.println("Age is valid: " + age);
    }

    public static void main(String[] args) {
        try {
            validateAge(15); // This will throw
        } catch (IllegalArgumentException e) {
            System.out.println("Caught: " + e.getMessage());
        }
        validateAge(20); // This will pass
    }
```

The `throws` Keyword

Definition

The **throws** keyword declares that a method *might* throw an exception. The *caller* is responsible for handling it.

```
import java.io.*;

public class ThrowsDemo {
    // Declares that this method might throw IOException
    static void readFile(String filename) throws IOException {
        FileReader fr = new FileReader(filename);
        BufferedReader br = new BufferedReader(fr);
        System.out.println(br.readLine());
        br.close();
    }

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

throw vs. throws

Feature	`throw`	`throws`
Purpose	Actually throws an exception	Declares possible exceptions
Used in	Method body	Method signature
Object	Throws one exception object at a time	Can declare multiple exceptions
Example	`throw new Exception()`	`void f() throws E`

throw

```
throw new IOException("File missing");
```

throws

```
void read() throws IOException { }
```

Creating Custom Exceptions

Why Custom Exceptions?

Custom exceptions make your code more **readable** and **meaningful** by using domain-specific error types.

```
// Step 1: Create the custom exception class
public class InsufficientFundsException extends Exception {
    private double amount;

    public InsufficientFundsException(double amount) {
        super("Insufficient funds. Short by: " + amount + " TZS");
        this.amount = amount;
    }

    public double getAmount() {
        return amount;
    }
}
```

Rule

- Extend `Exception` for **checked** custom exceptions

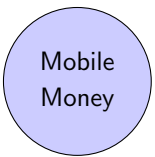
Using Custom Exceptions

```
public class BankAccount {
    private String owner;
    private double balance;

    public BankAccount(String owner, double initialBalance) {
        this.owner = owner;
        this.balance = initialBalance;
    }

    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException(amount - balance);
        }
        balance -= amount;
        System.out.println("Withdrew " + amount + " TZS. Balance: " + balance);
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount("Ali Hassan", 5000);
        try {
            account.withdraw(3000); // OK
            account.withdraw(4000); // Will throw exception
        } catch (InsufficientFundsException e) {
```

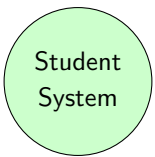
Case Studies Overview

Practical Application

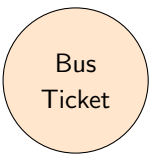
We will examine **three real-world case studies** that demonstrate exception handling in context:



Case Study 1
Mobile Money Transfer System



Case Study 2
Student Grade Management



Case Study 3
Daladala Booking System

Case Study 1: Mobile Money Transfer (Part 1)

Scenario: A mobile money system (like M-Pesa/Airtel Money) in Zanzibar.

```
// Custom Exceptions
class InvalidPhoneNumberException extends Exception {
    public InvalidPhoneNumberException(String phone) {
        super("Invalid phone number: " + phone +
              ". Must be 10 digits starting with 07X.");
    }
}

class DailyLimitExceededException extends RuntimeException {
    public DailyLimitExceededException(double limit) {
        super("Daily transfer limit of " + limit +
              " TZS exceeded.");
    }
}

class RecipientNotFoundException extends Exception {
    public RecipientNotFoundException(String phone) {
        super("Recipient with number " + phone + " not found.");
    }
}
```

Case Study 1: Mobile Money Transfer (Part 2)

```
public class MobileMoney {
    private String phoneNumber;
    private double balance;
    private double dailyTransferred = 0;
    private static final double DAILY_LIMIT = 1_000_000;

    public MobileMoney(String phone, double balance) {
        this.phoneNumber = phone;
        this.balance = balance;
    }

    public void transfer(String recipientPhone, double amount)
            throws InvalidPhoneNumberException,
                   RecipientNotFoundException,
                   InsufficientFundsException {
        // Validate phone number format
        if (!recipientPhone.matches("07[0-9]{8}")) {
            throw new InvalidPhoneNumberException(recipientPhone);
        }
        // Check daily limit (unchecked)
        if (dailyTransferred + amount > DAILY_LIMIT) {
            throw new DailyLimitExceededException(DAILY_LIMIT);
        }
```

Case Study 1: Mobile Money Transfer (Part 3)

```
public class MobileMoneyDemo {
    public static void main(String[] args) {
        MobileMoney account = new MobileMoney("0712345678", 50000);

        // Test 1: Valid transfer
        try {
            account.transfer("0756789012", 10000);
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }

        // Test 2: Invalid phone number
        try {
            account.transfer("123", 5000);
        } catch (InvalidPhoneNumberException e) {
            System.out.println("Phone Error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }

        // Test 3: Insufficient funds
        try {
            account.transfer("0712000000", 100000);
```

Case Study 2: Student Grade Management

Scenario: A SUZA student registration system that validates grades.

```
class InvalidGradeException extends Exception {
    public InvalidGradeException(double grade) {
        super("Grade " + grade + " is invalid. Must be between 0 and 100.");
    }
}

class StudentNotFoundException extends Exception {
    public StudentNotFoundException(String regNo) {
        super("Student with registration number " + regNo + " not found.");
    }
}

public class GradeManager {
    public void addGrade(String studentId, double grade)
            throws StudentNotFoundException, InvalidGradeException {
        if (studentId == null || studentId.isEmpty()) {
            throw new StudentNotFoundException(studentId);
        }
        if (grade < 0 || grade > 100) {
            throw new InvalidGradeException(grade);
        }
        System.out.println("Grade " + grade + " added for " + studentId);
```

Case Study 2: Student Grade Management (Demo)

```
public class GradeDemo {
    public static void main(String[] args) {
        GradeManager manager = new GradeManager();

        String[][] tests = {
            {"BITA2024001", "85"},
            {"",            "90"},   // Invalid student ID
            {"BITA2024002", "110"},  // Invalid grade
            {"BITA2024003", "-5"},   // Negative grade
        };

        for (String[] test : tests) {
            try {
                double grade = Double.parseDouble(test[1]);
                manager.addGrade(test[0], grade);
            } catch (StudentNotFoundException e) {
                System.out.println("Student Error: " + e.getMessage());
            } catch (InvalidGradeException e) {
                System.out.println("Grade Error:   " + e.getMessage());
            } catch (NumberFormatException e) {
                System.out.println("Format Error:  " + e.getMessage());
            } finally {
                System.out.println("  --> Attempt processed.");
```

Case Study 3: Daladala Ticket Booking

Scenario: A booking system for Zanzibar's Daladala (minibus) service.

```
class NoSeatsAvailableException extends Exception {
    public NoSeatsAvailableException(String route) {
        super("No seats available on route: " + route);
    }
}

public class DaladalaBus {
    private String route;
    private int totalSeats;
    private int bookedSeats = 0;

    public DaladalaBus(String route, int seats) {
        this.route = route;
        this.totalSeats = seats;
    }

    public String bookSeat(String passengerName)
            throws NoSeatsAvailableException {
        if (bookedSeats >= totalSeats) {
            throw new NoSeatsAvailableException(route);
        }
        bookedSeats++;
```

Case Study 3: Daladala Booking (Demo)

```
public class DaladalDemo {
    public static void main(String[] args) {
        DaladalaBus bus = new DaladalaBus("Stone Town - Chwaka", 3);

        String[] passengers = {
            "Fatuma Said", "Ali Hassan", "Zuwena Omar",
            "Mohammed Juma", "Khadija Ally" // Seats 4 & 5: no space!
        };

        for (String passenger : passengers) {
            try {
                bus.bookSeat(passenger);
            } catch (NoSeatsAvailableException e) {
                System.out.println("Booking failed for " + passenger +
                                   ": " + e.getMessage());
                System.out.println("Please wait for next bus.");
            } finally {
                System.out.println("Remaining seats: " +
                                   bus.getAvailableSeats());
            }
        }
    }
}
```

Exception Handling Best Practices

Do:

1. **Catch specific exceptions** before general ones
2. **Always close resources** in `finally` or use try-with-resources
3. **Log exceptions** to help with debugging
4. **Provide meaningful messages** in custom exceptions
5. **Let exceptions propagate** when you cannot handle them

Don't:

1. **Don't swallow exceptions** silently (empty catch block)
2. **Don't use exceptions** for normal control flow
3. **Don't catch** `Error` classes
4. **Don't catch** `Exception` when you mean a specific type
5. **Don't ignore** the exception object `e`

Anti-Pattern to Avoid

```
try { ... } catch (Exception e) { } // NEVER do this!
```

This hides bugs and makes debugging impossible.

Try-with-Resources (Java 7+)

Definition

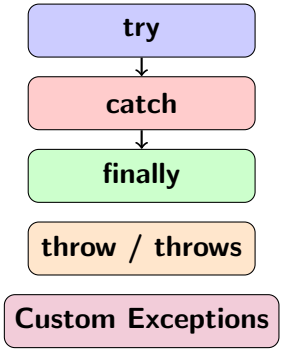
Try-with-resources automatically closes resources (files, connections) when done. No need for a `finally` block to close them.

```
import java.io.*;

public class TryWithResources {
    public static void readFile(String path) {
        // Resource automatically closed after try block
        try (BufferedReader br = new BufferedReader(new FileReader(path))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + path);
        } catch (IOException e) {
            System.out.println("Read error: " + e.getMessage());
        }
        // br is automatically closed here - even if exception occurred
    }
```

Summary

Key Concepts Covered:

- What exceptions are and why they occur
- Java exception hierarchy
- Checked vs. unchecked exceptions
- `try`, `catch`, `finally` blocks
- `throw` vs. `throws`
- Creating custom exceptions
- Try-with-resources



Key Takeaway

Exception handling makes programs **robust**, **user-friendly**, and **maintainable**. Always plan for what can go wrong!

Practice Questions

1. What is the difference between a `checked` and an `unchecked` exception? Give one example of each.
2. Write a Java method `divide(int a, int b)` that throws an `ArithmeticException` if `b` is zero, and returns the result otherwise. Write a `main` method to test it.
3. Create a custom exception `InvalidStudentIDException`. Write a method that validates a student ID (must start with *BITA* and be 12 characters long). Throw the custom exception if invalid.
4. Why should the `finally` block be used? Give a real-world example where it is important.
5. What is the difference between `throw` and `throws`?