Table of Contents

Week 5: Exception Handling and File I/O

Teaching Guide & Hands-on Project

📋 Session Overview

Duration: 2 - 3 Hours
Teaching: 90 minutes | **Hands-on Practice:** 50 minutes

🎯 Learning Objectives

By the end of this session, students will be able to:
1. Understand exceptions and error handling concepts
2. Use try-catch-finally blocks effectively
3. Handle multiple exception types
4. Throw and create custom exceptions
5. Read and write text files in Java
6. Work with CSV (Comma-Separated Values) files
7. Use try-with-resources for automatic resource management
8. Implement data persistence in applications using files

📚 Quick Review (5 minutes)

Quick Questions:
- What's the difference between ArrayList and HashSet?
- When would you use HashMap instead of ArrayList?
- What does an interface define?
- Can a class implement multiple interfaces?

Key Review Points:
- Collections provide dynamic data structures
- Choose collections based on needs: order, uniqueness, lookup
- Interfaces define contracts for behavior

🕐 Detailed Lesson Plan (120 Minutes)

Part 1: Understanding Exceptions (15 minutes)

What are Exceptions? (5 minutes)

Definition:
- Exceptions are events that disrupt normal program flow
- Represent errors or unexpected situations
- Allow graceful error handling instead of program crashes

Real-world analogy:

```
Ordering food at a restaurant:

Normal Flow:
1. You order food
2. Kitchen prepares it
3. You receive your meal
4. You eat

Exception Scenarios:
- Item not available → MenuItemException
- Kitchen is closed → KitchenClosedException
- Payment declined → PaymentException
- You're allergic → AllergyException

Each exception is handled differently!
```

Exception Hierarchy (5 minutes)

Draw the hierarchy:

```
                    Throwable
                        |
        ┌───────────────┴────────────────┐
        |                                |
    Exception                          Error
        |                                |
        ├── IOException              StackOverflowError
        ├── SQLException             OutOfMemoryError
        ├── FileNotFoundException    (Don't catch these!)
        └── RuntimeException
                |
                ├── NullPointerException
```

```
├── ArithmeticException
                └── IllegalArgumentException
```

Two Main Categories:

1. **Checked Exceptions** (must handle)
    - IOException, FileNotFoundException, SQLException
    - Compiler forces you to handle them
2. **Unchecked Exceptions** (Runtime Exceptions)
    - NullPointerException, ArrayIndexOutOfBoundsException
    - Can occur anywhere, not required to handle

Common Exceptions Demonstration (5 minutes)

Example 1: Common Runtime Exceptions

```java
public class CommonExceptionsDemo {
    public static void main(String[] args) {
        // 1. ArithmeticException
        try {
            int result = 10 / 0;  // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
        }

        // 2. NullPointerException
        try {
            String text = null;
            System.out.println(text.length());  // Calling method on null
        } catch (NullPointerException e) {
            System.out.println("Error: String is null!");
        }

        // 3. ArrayIndexOutOfBoundsException
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);  // Index doesn't exist
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Array index out of bounds!");
        }

        // 4. NumberFormatException
        try {
            int num = Integer.parseInt("abc");  // Not a valid number
        } catch (NumberFormatException e) {
            System.out.println("Error: Cannot convert to number!");
        }
```

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

Output:

```
Error: Cannot divide by zero!
Error: String is null!
Error: Array index out of bounds!
Error: Cannot convert to number!
Program continues running!
```

Part 2: Try-Catch-Finally Blocks (20 minutes)

Basic Try-Catch (8 minutes)

Example 2: Simple Try-Catch

```java
import java.util.Scanner;

public class TryCatchDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter first number: ");
        int num1 = scanner.nextInt();

        System.out.print("Enter second number: ");
        int num2 = scanner.nextInt();

        try {
            int result = num1 / num2;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
            System.out.println("Please enter a non-zero divisor.");
        }

        System.out.println("Thank you for using the calculator!");
        scanner.close();
    }
}
```

Key Points:
- Code that might fail goes in `try` block
- Exception handling goes in `catch` block
- Program continues after catch block

Multiple Catch Blocks (7 minutes)

Example 3: Handling Multiple Exceptions

```java
import java.util.Scanner;

public class MultipleCatchDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter array size: ");
            int size = scanner.nextInt();

            int[] numbers = new int[size];

            System.out.print("Enter index to access: ");
            int index = scanner.nextInt();

            System.out.print("Enter value: ");
            int value = scanner.nextInt();

            numbers[index] = value;
            System.out.println("Value stored successfully!");

            // Division operation
            System.out.print("Divide by: ");
            int divisor = scanner.nextInt();
            int result = value / divisor;
            System.out.println("Result: " + result);

        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Index out of array bounds!");
            System.out.println("Index must be between 0 and " + (e.getMessage()));
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero!");
        } catch (NegativeArraySizeException e) {
            System.out.println("Error: Array size cannot be negative!");
        } catch (Exception e) {
            // Catch-all for any other exceptions
            System.out.println("Error: Something went wrong!");
            System.out.println("Details: " + e.getMessage());
        }

        scanner.close();
    }
}
```

Important: Order matters! More specific exceptions before general ones.

Finally Block (5 minutes)

Example 4: Try-Catch-Finally

```java
import java.util.Scanner;

public class FinallyDemo {
    public static void main(String[] args) {
        Scanner scanner = null;

        try {
            scanner = new Scanner(System.in);
            System.out.print("Enter a number: ");
            int number = scanner.nextInt();

            int result = 100 / number;
            System.out.println("Result: " + result);

        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero!");
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            // This ALWAYS runs, even if exception occurs
            System.out.println("Cleaning up resources...");
            if (scanner ≠ null) {
                scanner.close();
                System.out.println("Scanner closed.");
            }
        }

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

Finally Block Rules:

- ALWAYS executes (even if exception occurs)
- Used for cleanup (closing files, connections, etc.)
- Runs even if there's a return statement in try/catch

Part 3: Throwing Exceptions and Custom Exceptions (15 minutes)

Throwing Exceptions (7 minutes)

Example 5: Throwing Exceptions

```java
public class ThrowExceptionDemo {

    public static void validateAge(int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative!");
        }
        if (age < 18) {
            throw new IllegalArgumentException("Must be 18 or older!");
        }
        System.out.println("Age is valid: " + age);
    }

    public static double calculateDiscount(double price, double discount) {
        if (price < 0) {
            throw new IllegalArgumentException("Price cannot be negative!");
        }
        if (discount < 0 || discount > 100) {
            throw new IllegalArgumentException("Discount must be between 0 and 100!");
        }
        return price * (1 - discount / 100);
    }

    public static void main(String[] args) {
        try {
            validateAge(25);
            validateAge(-5);  // This will throw exception
        } catch (IllegalArgumentException e) {
            System.out.println("Validation Error: " + e.getMessage());
        }

        try {
            double finalPrice = calculateDiscount(100, 20);
            System.out.println("Final price: $" + finalPrice);

            calculateDiscount(100, 150);  // Invalid discount
        } catch (IllegalArgumentException e) {
            System.out.println("Calculation Error: " + e.getMessage());
        }
    }
}
```

Custom Exceptions (8 minutes)

Example 6: Creating Custom Exceptions

```java
// Custom exception class
public class InvalidGradeException extends Exception {
    public InvalidGradeException(String message) {
        super(message);
```

```
    }    }
```

```
public class InsufficientBalanceException extends Exception {
    private double balance;
    private double amount;

    public InsufficientBalanceException(double balance, double amount) {
        super("Insufficient balance. Balance: $" + balance + ", Requested: $" + amount);
        this.balance = balance;
        this.amount = amount;
    }

    public double getBalance() {
        return balance;
    }

    public double getShortfall() {
        return amount - balance;
    }
}
```

```
// Using custom exceptions
public class BankAccount {
    private String accountNumber;
    private double balance;

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

    public void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException(balance, amount);
        }
        balance -= amount;
        System.out.println("Withdrawal successful. New balance: $" + balance);
    }

    public void deposit(double amount) {
        if (amount ≤ 0) {
            throw new IllegalArgumentException("Deposit amount must be positive!");
        }
        balance += amount;
        System.out.println("Deposit successful. New balance: $" + balance);
    }

    public double getBalance() {
        return balance;
```

```
    }    }
```

```
public class BankDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("ACC001", 1000);

        try {
            account.deposit(500);
            account.withdraw(300);
            account.withdraw(2000);  // This will throw exception
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction Failed: " + e.getMessage());
            System.out.println("You need $" + e.getShortfall() + " more.");
        } catch (IllegalArgumentException e) {
            System.out.println("Invalid Operation: " + e.getMessage());
        }

        System.out.println("Final balance: $" + account.getBalance());
    }
}
```

Part 4: File I/O Basics (20 minutes)

Reading Files (10 minutes)

Example 7: Reading Text Files

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileReadingDemo {
   public static void main(String[] args) {
      // Method 1: Using Scanner
      try {
         File file = new File("data.txt");
         Scanner fileScanner = new Scanner(file);

         System.out.println("=== Reading File ===");
         while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            System.out.println(line);
         }

         fileScanner.close();
```

```
            System.out.println("Error: File not found!");
            System.out.println("Please create 'data.txt' in the project directory.");      }
    }     }
```

Example 8: Reading with BufferedReader

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

public class BufferedReaderDemo {
    public static void main(String[] args) {
        BufferedReader reader = null;

        try {
            reader = new BufferedReader(new FileReader("students.txt"));
            String line;
            int lineNumber = 1;

            System.out.println("=== File Contents ===");
            while ((line = reader.readLine()) != null) {
                System.out.println(lineNumber + ": " + line);
                lineNumber++;
            }

        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found!");
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing file: " + e.getMessage());
            }
        }
    }
}
```

Writing Files (10 minutes)

Example 9: Writing to Files

```
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;
```

```
public class FileWritingDemo {    public static void main(String[] args) {
        // Method 1: Simple FileWriter        try {
            FileWriter writer = new FileWriter("output.txt");
            writer.write("Hello, File I/O!\n");            writer.write("This is line 2.\n");
            writer.write("This is line 3.\n");            writer.close();
            System.out.println("File written successfully!");
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());        }
        // Method 2: BufferedWriter (more efficient)        try {
            BufferedWriter bw = new BufferedWriter(new FileWriter("students.txt"));
                        bw.write("Alice,20,Computer Science,3.8");            bw.newLine();
            bw.write("Bob,22,Mathematics,3.5");            bw.newLine();
            bw.write("Charlie,21,Physics,3.9");            bw.newLine();
            bw.close();            System.out.println("Student data written successfully!");
                    } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());        }    }    }
```

Example 10: Appending to Files

```
import java.io.FileWriter;
import java.io.IOException;

public class AppendFileDemo {
    public static void main(String[] args) {
        try {
            // true parameter means append mode
            FileWriter writer = new FileWriter("log.txt", true);

            writer.write("New log entry at " + new java.util.Date() + "\n");
            writer.write("User logged in\n");
            writer.write("---\n");

            writer.close();
            System.out.println("Log appended successfully!");

        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Part 5: Try-With-Resources (10 minutes)

Modern Resource Management (10 minutes)

Example 11: Try-With-Resources

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

public class TryWithResourcesDemo {

    // Old way (verbose)
    public static void readFileOldWay(String filename) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(filename));
            String line;
            while ((line = reader.readLine()) ≠ null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            try {
                if (reader ≠ null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing: " + e.getMessage());
            }
        }
    }

    // New way (cleaner with try-with-resources)
    public static void readFileNewWay(String filename) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) ≠ null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
        // Resource is automatically closed!
    }

    public static void main(String[] args) {
        System.out.println("═══ Reading with try-with-resources ═══");
        readFileNewWay("data.txt");
    }
}
```

Benefits of Try-With-Resources:

- Automatic resource closing

- Cleaner, more readable code
- Reduces resource leaks
- Handles multiple resources easily

📚 Additional Resources

- Java Exception Handling
- Java I/O Tutorial
- Try-With-Resources
- Working with Files

📌 Key Takeaways

> ***"Exceptions = Graceful Error Handling"***
> *Don't let your program crash - handle errors elegantly*

> ***"Try-With-Resources = Modern Best Practice"***
> *Automatic resource management prevents leaks*

> ***"Files = Data Persistence"***
> *Your data lives beyond program execution*

> ***"Always Validate Input"***
> *Never trust user input - validate and handle errors*

> ***"Backups Save Lives"***
> *Always create backups before modifying data*