MODULE 4

Exception Handling and I/O Exceptions

Java Object-Oriented Programming

Semester 3 | DCSA250059

PART A: EXCEPTION HANDLING

1. Introduction to Exceptions

What is an Exception?

An exception is an abnormal event that disrupts the normal flow of program execution during runtime.

Common examples:

```
int result = 10 / 0;   // ArithmeticException
int[] arr = new int[5];
arr[10] = 50;          // ArrayIndexOutOfBoundsException
String s = null;
s.length();            // NullPointerException
```

Why Handle Exceptions?

- Prevents program from crashing abnormally
- Provides meaningful error messages
- Allows graceful recovery from errors
- Separates error-handling from normal code

2. Exception Hierarchy

Java exceptions follow a hierarchical structure rooted at java.lang.Throwable:

```
java.lang.Object
    └── java.lang.Throwable
            ├── java.lang.Error  (Unrecoverable)
            │       ├── OutOfMemoryError
            │       ├── StackOverflowError
            │       └── VirtualMachineError
            └── java.lang.Exception  (Recoverable)
                    ├── Checked Exceptions (Compile-time)
```

```
│       ├── IOException
│       ├── SQLException
│       └── FileNotFoundException
└── RuntimeException (Unchecked)
        ├── ArithmeticException
        ├── NullPointerException
        ├── ArrayIndexOutOfBoundsException
        └── NumberFormatException
```

Types of Exceptions

Type	Checked by Compiler	When Occurs	Examples
Checked Exception	Yes	At compile time	IOException, SQLException
Unchecked Exception	No	At runtime	ArithmeticException, NullPointerException
Error	No	At runtime (JVM level)	OutOfMemoryError

3. Exception Handling Keywords

Keyword	Purpose	Syntax
try	Block where exception may occur	try { // risky code }
catch	Handles specific exception	catch(ExceptionType e) { // handler }
finally	Always executes (cleanup)	finally { // cleanup }
throw	Throws exception explicitly	throw new ExceptionType()
throws	Declares exceptions method may throw	void method() throws IOException

4. try-catch Block

Basic Syntax

```
try {
    // Code that may throw exception
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero!");
    System.out.println("Error: " + e.getMessage());
}
```

Multiple catch Blocks

```
try {
    int[] numbers = {10, 20, 30};
```

```
    int result = numbers[5] / 0;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic error: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index error: " + e.getMessage());
} catch (Exception e) {
    System.out.println("General error: " + e.getMessage());
}
```

Multi-catch (Java 7+)

```
try {
    String str = "abc";
    int num = Integer.parseInt(str);
} catch (ArithmeticException | NumberFormatException | NullPointerException e) {
    System.out.println("Exception: " + e.getClass().getSimpleName());
}
```

5. Nested try Statements

```
try {
    System.out.println("Outer try block");
    int[] arr = {1, 2, 3};

    try {
        System.out.println("Inner try block");
        int result = 10 / 0;  // ArithmeticException
    } catch (ArithmeticException e) {
        System.out.println("Inner catch: " + e);
    }

    arr[5] = 100;  // ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Outer catch: " + e);
}
```

6. throw Keyword

Used to explicitly throw an exception from within a method.

Syntax

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

throw vs throws

throw	throws
Used to actually throw exception	Used to declare exceptions
Inside method body	In method signature

Can throw only one exception	Can declare multiple exceptions
Followed by exception object	Followed by exception class(es)

7. Custom Exceptions (User-defined)

Creating a Custom Exception

```
// Custom checked exception (extends Exception)
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
    public InvalidAgeException(String message, Throwable cause) {
        super(message, cause);
    }
}

// Custom unchecked exception (extends RuntimeException)
class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}
```

Complete Example – Voter Validation

```
class AgeNotValidException extends Exception {
    private int age;
    public AgeNotValidException(String message, int age) {
        super(message);
        this.age = age;
    }
    public int getAge() { return age; }
}

class Voter {
    private String name;
    private int age;
    public Voter(String name, int age) throws AgeNotValidException {
        if (age < 18)
            throw new AgeNotValidException("Age not valid for voting", age);
        this.name = name;
        this.age = age;
    }
    void display() {
        System.out.println("Voter: " + name + ", Age: " + age);
    }
}

// Output:
// Voter: Alice, Age: 25
// Exception: Age not valid for voting
// Invalid age provided: 16
```

8. throws Keyword

Declares that a method may throw one or more exceptions.

Syntax

```
returnType methodName(params) throws ExceptionType1, ExceptionType2 { ... }
```

Exception Propagation

```
void method3() throws ArithmeticException {
    int result = 10 / 0;  // Exception occurs here
}
void method2() throws ArithmeticException {
    method3();  // Propagates up
}
void method1() {
    try {
        method2();  // Handled here
    } catch (ArithmeticException e) {
        System.out.println("Handled in method1: " + e);
    }
}
```

9. finally Block

The finally block always executes regardless of whether an exception occurs or not.

Key Characteristics

- Executes after try/catch blocks
- Used for cleanup code (closing files, database connections, etc.)
- Executes even if there is a return statement in try/catch
- Does NOT execute only if System.exit() is called

Example

```
try {
    int a = 10, b = 0;
    int result = a / b;
} catch (ArithmeticException e) {
    System.out.println("Exception caught: " + e);
} finally {
    System.out.println("Finally block always executes");
}
```

Try-with-Resources (Java 7+)

> *Best practice: Use try-with-resources for automatic resource management — no finally block needed!*

```
try (FileReader fr = new FileReader("sample.txt");
```

```
     BufferedReader br = new BufferedReader(fr)) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
}
// Resources automatically closed — no finally needed
```

10. Built-in Exceptions Reference

Common Unchecked Exceptions (RuntimeException)

Exception	Description	Example
ArithmeticException	Division by zero	10 / 0
NullPointerException	Accessing null object	str.length() when str is null
ArrayIndexOutOfBoundsException	Invalid array index	arr[10] when size is 5
StringIndexOutOfBoundsException	Invalid string index	"hello".charAt(10)
NumberFormatException	Invalid number format	Integer.parseInt("abc")
ClassCastException	Invalid casting	(String) new Integer(5)
IllegalArgumentException	Invalid argument passed	Passing negative to sqrt
NegativeArraySizeException	Negative array size	new int[-5]

Common Checked Exceptions

Exception	Description	When Occurs
IOException	I/O operation failed	Reading/writing files
FileNotFoundException	File not found	Opening non-existent file
EOFException	End of file reached	Reading beyond file end
ClassNotFoundException	Class not found	Class.forName() on missing class
InterruptedException	Thread interrupted	Thread.sleep() interrupted
SQLException	Database error	Database operations

Common Errors (Non-recoverable)

Error	Description
OutOfMemoryError	JVM has run out of memory
StackOverflowError	Stack overflow (infinite recursion)

VirtualMachineError	JVM internal error
NoClassDefFoundError	Class definition not found at runtime

PART B: INPUT / OUTPUT (I/O)

11. Java I/O Streams Overview

What is a Stream?

A stream is a sequence of data flowing between a program and a source/destination (e.g., file, console, network).

Stream Classification

```
                    Streams
                       |
          +--------------+--------------+
          |                             |
  Byte Streams                 Character Streams
(8-bit binary data)           (16-bit text data)
          |                             |
  InputStream              Reader    Writer
  OutputStream
```

Stream Hierarchy

```
InputStream (Byte input)
    ├── FileInputStream
    ├── BufferedInputStream
    ├── DataInputStream
    └── ObjectInputStream

OutputStream (Byte output)
    ├── FileOutputStream
    ├── BufferedOutputStream
    ├── DataOutputStream
    └── ObjectOutputStream

Reader (Character input)
    ├── FileReader
    ├── BufferedReader
    └── InputStreamReader

Writer (Character output)
    ├── FileWriter
    ├── BufferedWriter
    └── PrintWriter
```

12. Byte Streams

FileInputStream and FileOutputStream

FileInputStream reads bytes from a file. FileOutputStream writes bytes to a file.

```
// Writing bytes to file
try (FileOutputStream fos = new FileOutputStream("data.txt")) {
    String text = "Hello, Byte Stream!";
    fos.write(text.getBytes());
}

// Reading bytes from file
try (FileInputStream fis = new FileInputStream("data.txt")) {
    int ch;
    while ((ch = fis.read()) != -1) {
        System.out.print((char) ch);
    }
}
```

Buffered Byte Streams

BufferedInputStream and BufferedOutputStream improve performance by reducing disk access.

```
// Buffered write
try (BufferedOutputStream bos = new BufferedOutputStream(
        new FileOutputStream("file.txt"))) {
    bos.write(data.getBytes());
    bos.flush();  // Ensure all data is written
}

// Buffered read
try (BufferedInputStream bis = new BufferedInputStream(
        new FileInputStream("file.txt"))) {
    byte[] buffer = new byte[1024];
    int bytesRead;
    while ((bytesRead = bis.read(buffer)) != -1) {
        System.out.print(new String(buffer, 0, bytesRead));
    }
}
```

DataInputStream and DataOutputStream

Used to read and write Java primitive data types directly.

```
// Write primitives
try (DataOutputStream dos = new DataOutputStream(
        new FileOutputStream("data.dat"))) {
    dos.writeInt(100);
    dos.writeDouble(99.99);
    dos.writeBoolean(true);
    dos.writeUTF("Hello World");
}

// Read primitives (must read in same order written)
try (DataInputStream dis = new DataInputStream(
        new FileInputStream("data.dat"))) {
    int i = dis.readInt();
    double d = dis.readDouble();
    boolean b = dis.readBoolean();
```

```
    String s = dis.readUTF();
}
```

13. Character Streams

FileReader and FileWriter

Used for reading and writing character (text) data. Support for Unicode characters.

```
// Write characters
try (FileWriter fw = new FileWriter("text.txt")) {
    fw.write("Hello, Character Stream!");
}

// Append to file
try (FileWriter fw = new FileWriter("text.txt", true)) {  // true = append
    fw.write("\nAppended line");
}

// Read characters
try (FileReader fr = new FileReader("text.txt")) {
    int ch;
    while ((ch = fr.read()) != -1) {
        System.out.print((char) ch);
    }
}
```

BufferedReader and BufferedWriter

Provides efficient character I/O with buffering and line-by-line operations.

```
// Write lines
try (BufferedWriter bw = new BufferedWriter(new FileWriter("lines.txt"))) {
    bw.write("Line 1");
    bw.newLine();  // Platform-independent line separator
    bw.write("Line 2");
    bw.newLine();
}

// Read line by line
try (BufferedReader br = new BufferedReader(new FileReader("lines.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

PrintWriter

Provides formatted text output, similar to System.out.

```
try (PrintWriter pw = new PrintWriter(new FileWriter("output.txt"))) {
    pw.println("Hello World");
    pw.printf("Formatted: %.2f%n", 99.99);
    pw.print("No newline");
}
```

14. Standard Input/Output Streams

System.in, System.out, System.err

```
// System.out - Standard output
System.out.println("Standard output");
System.out.printf("Formatted: %d %s %f%n", 10, "Hi", 3.14);

// System.err - Standard error output
System.err.println("This is an error message");

// System.in - Standard input via Scanner
Scanner sc = new Scanner(System.in);
System.out.print("Enter name: ");
String name = sc.nextLine();
System.out.print("Enter age: ");
int age = sc.nextInt();
```

15. Quick Reference — I/O Methods

InputStream / OutputStream Methods

Method	Description
int read()	Reads next byte of data
int read(byte[] b)	Reads bytes into array
void write(int b)	Writes one byte
void write(byte[] b)	Writes byte array
void close()	Closes the stream

Reader / Writer Methods

Method	Description
int read()	Reads single character
int read(char[] cbuf)	Reads into char array
String readLine()	Reads line of text (BufferedReader only)
void write(int c)	Writes single character
void write(String str)	Writes a string
void newLine()	Writes line separator (BufferedWriter only)

File Class Methods

Method	Description

exists()	Checks if file/directory exists
isFile()	Returns true if it is a file
isDirectory()	Returns true if it is a directory
length()	Returns file size in bytes
delete()	Deletes the file
mkdir()	Creates a directory

Key Points to Remember

> *These points are critical for exams and practical programming!*

1. Checked exceptions MUST be handled — use try-catch or declare with throws
2. Unchecked (RuntimeException) exceptions are optional to handle
3. The finally block always executes — except when System.exit() is called
4. Try-with-resources (Java 7+) auto-closes resources — preferred approach
5. Use PrintWriter for formatted text output
6. Use BufferedReader for reading text files line by line
7. Always close streams to prevent resource leaks
8. Use byte streams for binary files, character streams for text files
9. throw is used to throw an exception; throws declares method exceptions
10. Custom exceptions extend Exception (checked) or RuntimeException (unchecked)

End of Module 4: Exception Handling and I/O Exceptions