Module V: Abstract Classes and Interfaces, Threading

Course: Object oriented Programming in JAVA

Dr. Vishwa Pratap Singh

NextGen Academy

February 10, 2024

NextGen Academy
Towards fulfilling a million dreams

Exception Handling in Java

- Java's built-in mechanism for handling unexpected events.
- Graceful dealing with errors during program execution.
- Key components:
    - Exception Types
    - Try-Catch Block
    - Throw and Throws Keywords
    - Finally Block
    - Exception Hierarchy

NextGen Academy
Towards fulfilling a million dreams

Exception Types in Java

- **Checked Exceptions:**
  - Mandated to handle explicitly.
  - Examples: IOException, SQLException.
- **Unchecked Exceptions (Runtime Exceptions):**
  - Do not require explicit handling.
  - Result from programming errors.
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException.

NextGen Academy
Towards fulfilling a million dreams

Try-Catch Block

- Used to surround code that might throw exceptions.
- If an exception occurs, control transferred to the catch block.
- Multiple catch blocks for different types of exceptions.

```
try {
  // code that might throw exceptions
} catch (ExceptionType1 e1) {
  // handle ExceptionType1
} catch (ExceptionType2 e2) {
  // handle ExceptionType2
}
```

NextGen Academy
Towards fulfilling a million dreams

Throw and Throws Keywords

- **Throw:** Explicitly throw an exception.
- **Throws:** Used in method declarations to indicate potential exceptions.

```
// Throw example
throw new CustomException(”Custom■message”);

// Throws example
void myMethod() throws IOException {
  // method code
}
```

Finally Block

- Specify code that must execute regardless of exceptions.
- Used for resource cleanup tasks.

```
try {
  // code that might throw exceptions
} catch (Exception e) {
  // handle exception
} finally {
  // code to execute always
}
```

Exception Hierarchy

- Java has a hierarchy of exception classes.
- Base classes: `Throwable`, `Error` (non-recoverable), `Exception` (recoverable).

Exception Types Overview

- **Checked Exceptions:**
  - Must be explicitly declared or caught.
  - Expected and can be reasonably handled.
  - Examples: IOException, SQLException.
- **Unchecked Exceptions:**
  - Do not need explicit declaration or catching.
  - Result from programming errors.
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException.

Defining Custom Exception Classes

- Create own exceptions as derived classes of `Exception`.
- Reasons for custom exceptions:
    - Specific treatment to a subset of existing exceptions.
    - Business logic exceptions for better understanding.

NextGen Academy
Towards fulfilling a million dreams

Why use Custom Exceptions

- Catch and provide specific treatment to a subset of exceptions.
- Business logic exceptions enhance understanding.
- Example: `WrongFileNameException`.

NextGen Academy
Towards fulfilling a million dreams

Example of Custom Exception

```
// Custom exception class
class WrongFileNameException extends Exception {
  public WrongFileNameException(String errorMessage) {
    super(errorMessage);
  }
}
```

NextGen Academy
Towards fulfilling a million dreams

Example 1: Custom Exception

```
// Custom exception class
class InvalidAgeException extends Exception {
  public InvalidAgeException(String str) {
    super(str);
  }
}

// Using the custom exception
public class TestCustomException1 {
  static void validate(int age) throws InvalidAgeException {
    if (age < 18) {
      throw new InvalidAgeException(”Age■is■not■valid■to■vote”)
    } else {
      System.out.println(”Welcome■to■vote”);
    }
  }

  // Main method
```

Abstract Classes in Java

- A class that cannot be instantiated directly.
- Declared using the `abstract` keyword.
- Key observations about abstract classes.

Key Observations about Abstract Classes

- Instance of an abstract class cannot be created.
- Constructors are allowed.
- Abstract class can have no abstract methods.
- Final method in abstract class is allowed.
- Static methods can be defined in an abstract class.
- Abstract keyword for top-level and inner classes.
- Abstract class must contain at least one abstract method.

Examples of Java Abstract Class

```
// Abstract class
abstract class Sunstar {
  abstract void printInfo();
}

// Abstraction performed using extends
class Employee extends Sunstar {
  void printInfo() {
    String name = "Avinash";
    int age = 21;
    float salary = 222.2F;

    System.out.println(name);
    System.out.println(age);
    System.out.println(salary);
  }
}
```

NextGen Academy
Towards fulfilling a million dreams

Abstract Class with Constructor, Data Member, and Methods

```
// Java Program to implement Abstract Class
// with constructor, data member, and methods

abstract class Subject {
  Subject() {
    System.out.println("Learning■Subject");
  }

  abstract void syllabus();

  void Learn() {
    System.out.println("Preparing■Right■Now!");
  }
}

class IT extends Subject {
  void syllabus() {
    System.out.println("C■Java■C++");
```

Properties of Abstract Class

- An abstract class cannot be instantiated directly.
- At least one pure virtual function.
- Can contain both abstract and non-abstract methods.
- Can have constructors and destructors.
- Can have member variables.
- Can be used as a base class.

NextGen Academy
Towards fulfilling a million dreams

Features of Abstract Class

- Cannot be instantiated directly.
- Contains at least one pure virtual function.
- Can have both abstract and non-abstract methods.
- Can have constructors and destructors.
- Can have member variables.
- Can be used as a base class.

NextGen Academy
Towards fulfilling a million dreams

Features of Abstract Class (Contd.)

- Abstract classes define a common interface.
- Abstract classes are inherited by other classes.
- Provide a blueprint for derived classes.
- Used to share behavior among related classes.

NextGen Academy
Towards fulfilling a million dreams

Example of Abstract Class

```
// Abstract class
abstract class Sunstar {
  abstract void printInfo();
}

// Derived class
class Employee extends Sunstar {
  void printInfo() {
    String name = ”Avinash”;
    int age = 21;
    float salary = 222.2F;

    System.out.println(name);
    System.out.println(age);
    System.out.println(salary);
  }
}
```

Conclusion

- Exception handling is crucial for graceful error management.
- Abstract classes provide a blueprint for other classes.
- Understanding these concepts is essential for Java developers.

NextGen Academy
Towards fulfilling a million dreams

The Comparable Interface

- Used to compare objects of the same class.
- Implements `java.lang.Comparable`.
- Provides ordering for user-defined class objects.
- Requires `compareTo` method implementation.
- Sorting array of pairs using Comparable.

Using Comparable Interface

- Implement Comparable in `Pair` class.
- `compareTo` decides element order.
- Use `Arrays.sort()` to sort the array.

NextGen Academy
Towards fulfilling a million dreams

Example 1 - Sorting Pairs

- Given array of Pairs: { ”abc”, 3, ”a”, 4, ”bc”, 5, ”a”, 2 }.
- Sort in ascending lexicographical order, then by integer value.

```
Input: {{"abc", 3}, {"a", 4}, {"bc", 5}, {"a", 2}}
Output: {{"a", 2}, {"a", 4}, {"abc", 3}, {"bc", 5}}
```

NextGen Academy
Towards fulfilling a million dreams

Code - Comparable Interface

```
class Pair implements Comparable<Pair> {
    String x;
    int y;

    public Pair(String x, int y) {
        // constructor
    }

    @Override
    public int compareTo(Pair a) {
        // compareTo implementation
    }
}

public class GFG {
    public static void main(String[] args) {
        // main function
    }
```

Interfaces vs. Abstract Classes

- Both define contracts in OOP.
- Abstract class cannot be instantiated.
- Interface specifies methods to implement.
- Abstract class can have implemented methods.
- Interface methods are by default abstract.

Differences - Abstract Class vs. Interface

- Method Implementation: Abstract class can have both abstract and concrete methods.
- Inheritance: Class can inherit from only one abstract class but can implement multiple interfaces.
- Access Modifiers: Abstract class can have access modifiers for methods and properties.
- Variables: Abstract class can have member variables; interface cannot.

NextGen Academy
Towards fulfilling a million dreams

Abstract Class vs. Interface (contd.)

- Type of Methods: Interface has only abstract methods, abstract class can have concrete methods.
- Final Variables: Variables in an interface are by default final.
- Type of Variables: Abstract class can have final, non-final, static, and non-static variables.
- Implementation: Abstract class can provide the implementation of the interface; the interface cannot.

Abstract Class vs. Interface (contd.)

- Inheritance vs. Abstraction: Interface is implemented using ”implements,” abstract class is extended using ”extends.”
- Multiple Implementations: Interface can extend multiple interfaces; abstract class can extend and implement.
- Multiple Inheritance: Partially achieved by interfaces; not possible with abstract classes.
- Accessibility of Data Members: Interface members are final by default; abstract class can have various access modifiers.

NextGen Academy
Towards fulfilling a million dreams

Interface Features

- Defines a set of methods and properties.
- Provides a common protocol for communication.
- Supports polymorphism.
- Enables separation of concerns in a software system.
- Improves code reusability.

NextGen Academy
Towards fulfilling a million dreams

Interface Features (contd.)

- Enforces design patterns, e.g., Adapter pattern.
- Facilitates testing by allowing independent testing of components.

NextGen Academy
Towards fulfilling a million dreams

Example - Interface Implementation

```
interface Shape {
    void draw();
    double area();
}

class Rectangle implements Shape {
    // Implementation for Rectangle
}

class Circle implements Shape {
    // Implementation for Circle
}

class GFG {
    // Main driver method
    public static void main(String[] args) {
        // Example usage
    }
```

Text I/O Handled in Java

- TextIO functions for inputting various types.
- Examples: `TextIO.getlnInt()`, `TextIO.getlnDouble()`, `TextIO.getlnBoolean()`, etc.
- Variables must be declared before using these functions.
- Functions guarantee legal values of correct type.

NextGen Academy
Towards fulfilling a million dreams

TextIO Input Functions

```
j = TextIO.getlnInt();       // Reads int value.
y = TextIO.getlnDouble();    // Reads double value.
a = TextIO.getlnBoolean();   // Reads boolean value.
c = TextIO.getlnChar();      // Reads char value.
w = TextIO.getlnWord();      // Reads String value.
s = TextIO.getln();          // Reads entire line as a String.
```

Example - Text I/O

```
public class Interest2 {
    public static void main(String[] args) {
        double principal;
        double rate;
        double interest;

        System.out.print("Enter initial investment: ");
        principal = TextIO.getlnDouble();

        System.out.print("Enter annual interest rate (as a decima
        rate = TextIO.getlnDouble();

        interest = principal * rate;
        principal = principal + interest;

        System.out.printf("Interest: $%1.2f%n", interest);
        System.out.printf("Value after one year: $%1.2f%n", princ
    }
```

Text I/O vs. Binary I/O

- Text I/O for reading/writing text data.
- Binary I/O for binary data (images, sounds).
- Text I/O deals with characters.
- Binary I/O deals with bytes.
- Text I/O is generally slower.

Text I/O Example

```
import java.io.*;

public class TextIOExample {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, world!");
            writer.close();

            FileReader reader = new FileReader("example.txt");
            int character;

            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }

            reader.close();
```

Binary I/O Example

```
import java.io.*;

public class BinaryIOExample {
    public static void main(String[] args) {
        try {
            FileOutputStream outputStream = new FileOutputStream(
            byte[] data = {0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x
                           0x6f, 0x72, 0x6c, 0x64, 0x21};

            outputStream.write(data);
            outputStream.close();

            FileInputStream inputStream = new FileInputStream("ex
            int byteRead;

            while ((byteRead = inputStream.read()) != -1) {
                System.out.print((char) byteRead);
            }
```

Binary I/O Classes Motivations

- Byte Streams for reading and writing binary data.
- Descended from `InputStream` and `OutputStream`.
- Methods like `read()` and `write()` for bytes.
- Implementation classes like `FileInputStream` and `FileOutputStream`.

NextGen Academy
Towards fulfilling a million dreams

Understanding Byte Streams

- Byte streams read and write 8-bit bytes.
- Classes like `FileInputStream` and `FileOutputStream`.
- Abstract classes `InputStream` and `OutputStream`.
- Methods like `read()` and `write()` for bytes.
- Efficient buffering with `BufferedInputStream` and `BufferedOutputStream`.

Introduction to Generics

- Programs using Generics offer code reuse and flexibility.
- Generics enhance type safety, catching errors at compile time.
- Example illustrating runtime exception without Generics.
- `ArrayList` storing names, but an integer is added causing a runtime exception.

Benefits of Generics

- Code Reuse: Write methods/classes/interfaces once for any type.
- Type Safety: Errors appear at compile time, improving code reliability.
- Examples showcasing the importance of type safety.
- Compile-time detection of issues prevents runtime exceptions.

NextGen Academy
Towards fulfilling a million dreams

Generic Methods

- Generics allow parameterized types.
- Classes, interfaces, or methods operating on parameterized types are generic entities.
- Generics offer type safety compared to using the `Object` class.
- Comparison of Generics in Java with templates in C++.

Types of Java Generics

- Generic Method: Takes a parameter, returns a value, cited by actual type.
- Generic Classes: Implemented like non-generic classes but with a type parameter section.
- Syntax for creating objects of a generic class.
- Examples illustrating the use of generic classes.

NextGen Academy
Towards fulfilling a million dreams

Processes in Java

- Process: Standalone program running independently with its own memory space.
- Each process has its own Java Virtual Machine (JVM) instance.
- Processes do not share memory space, requiring IPC mechanisms for communication.
- Processes are heavyweight and resource-intensive.

NextGen Academy
Towards fulfilling a million dreams

Threads in Java

- Threads: Lightweight sub-processes sharing the same memory space.
- Threads are suitable for concurrent execution within a program.
- Comparison between processes and threads.
- Threads are created and managed using the `Thread` class or implementing the `Runnable` interface.

NextGen Academy
Towards fulfilling a million dreams

Thread Objects

- Thread objects are fundamental for managing and controlling threads.
- Key aspects: Creating, starting, pausing, resuming, and managing threads.
- Examples demonstrating the creation and starting of threads.
- Thread priorities, joining threads, and ensuring thread safety.

NextGen Academy
Towards fulfilling a million dreams

Defining and Starting a Thread

- Two approaches: Extending `Thread` class and implementing `Runnable` interface.
- Examples illustrating both approaches.
- Proper usage of the `start()` method for thread execution.
- Importance of not calling `run()` directly.

NextGen Academy
Towards fulfilling a million dreams

Pausing Execution with Sleep

- Pause a thread's execution for a specified duration using `Thread.sleep()`.
- Useful for controlling the timing of thread execution.
- Example demonstrating the use of `Thread.sleep()`.
- Handling `InterruptedException` when using `Thread.sleep()`.

NextGen Academy
Towards fulfilling a million dreams

Interrupts in Java

- Thread interrupts provide a way to interrupt normal thread execution.
- Key concepts: Interrupting a thread, checking for interruption, handling `InterruptedException`.
- Common use cases and thread termination strategies.
- Flexibility and cooperative nature of thread interrupts.

Joins in Java

- `join()` method for waiting for the completion of another thread's execution.
- Handling `InterruptedException` when using `join()`.
- Order of joining threads and specifying timeouts.
- Coordination and synchronization using `join()`.

NextGen Academy
Towards fulfilling a million dreams

Synchronization in Java

- Controlling access to shared resources in a multi-threaded environment.
- Synchronized blocks and methods for mutual exclusion.
- Intrinsic locks (monitors) and preventing race conditions.
- Handling deadlocks and using `wait()`, `notify()`, and `notifyAll()`.

Applets - Applet Fundamentals

- Introduction to applets as small embedded programs in web browsers.
- Basics of applet development: `init()`, `start()`, `stop()`, `destroy()` methods.
- Usage of `paint()` method for rendering graphics.
- Security restrictions imposed on Java applets.

NextGen Academy
Towards fulfilling a million dreams

- Creating a Hello World applet.
- Executing applets in web browsers and using applet viewers.
- Utilizing `showStatus()` for updating the status bar.
- Features and restrictions of applet communication.

- Event-driven programming in applets.
- Basics of event handling: Event sources, listeners, and adapters.
- Registering event listeners and responding to events.
- Examples showcasing various types of events in applets.

Networking in Java

- Overview of networking concepts in Java.
- Classes in the `java.net` package for network communication.
- Basic socket programming: `Socket` and `ServerSocket`.
- Examples demonstrating client-server communication.

Remote Method Invocation (RMI)

- RMI allows invoking methods on remote objects.
- Distributed computing in Java using RMI.
- Key components: `Remote` interface, `Registry`, `Naming`.
- Example showcasing RMI usage.

Java Database Connectivity (JDBC)

- JDBC for database connectivity in Java applications.
- Establishing database connections using `DriverManager`.
- Executing SQL queries with `Statement` and `PreparedStatement`.
- Handling results using `ResultSet`.

GUI Programming in Java

- Graphical User Interfaces (GUIs) enhance user interaction.
- Swing and JavaFX frameworks for GUI development.
- Key components: Frames, Panels, Buttons, TextFields, etc.
- Examples illustrating the creation of basic GUI applications.

NextGen Academy
Towards fulfilling a million dreams

JavaFX - Advanced GUI Features

- Introduction to JavaFX for modern GUI development.
- Scene Builder for visual layout design.
- Advanced features: Animation, CSS styling, FXML, and Event Handling.
- Creating a JavaFX application with multimedia elements.

NextGen Academy
Towards fulfilling a million dreams