Advanced Programming (OOP)

Module 4: Access Modifiers, Exceptions, and Multithreading

SDB

Spring 2025

Module 4: Topics

- Access Modifiers: Public, Private, Protected, and Default.
- Java APIs and Packages
- Exception Handling: Try, Catch, Finally, Throw, and Throws.
- User-Defined Exceptions.
- Basics of Multithreading and Synchronization.

Outline

Access Specifiers in Java

Definition: Access specifiers define the visibility and accessibility of classes, methods, and variables in Java.

Types of Access Specifiers:

- **Public:** Accessible from anywhere.
- **Private:** Accessible only within the same class.
- **Protected:** Accessible within the same package and subclasses.
- **Default (Package-Private):** Accessible only within the same package.

Example:

```
class Example {
    public int publicVar = 10;
    private int privateVar = 20;
    protected int protectedVar = 30;
    int defaultVar = 40; // Default access
}
```

Listing 1: Access Specifiers

The ‘this‘ and ‘super‘ Keywords

‘this‘ Keyword:

- Refers to the current instance of the class.
- Used to access current class methods, fields, and constructors.
- Resolves naming conflicts between instance variables and parameters.

‘super‘ Keyword:

- Refers to the immediate parent class instance.
- Used to call parent class methods and constructors.
- Accesses hidden fields or overridden methods in the parent class.

Examples: ‘this‘ and ‘super‘ I

Using ‘this‘ to Resolve Naming Conflicts:

```
class Example {
    int value;

    Example(int value) {
        this.value = value; // Resolves conflict with
            parameter
    }

    void display() {
        System.out.println("Value: " + this.value); //
            Refers to instance variable
    }
}
```

Listing 2: Using this Keyword

Examples: ‘this‘ and ‘super‘ II

Using ‘super‘ to Access Parent Class Members:

```
class Parent {
    void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void show() {
        super.show(); // Calls parent class method
        System.out.println("Child method");
    }
}
```

Listing 3: Using super Keyword

Outline

Introduction to Java API

Definition: Java API (Application Programming Interface) is a collection of prewritten classes, interfaces, and packages that provide standard functionality for building Java applications.

Key Features:

- Vast collection of libraries for various functionalities.
- Simplifies application development by providing reusable components.
- Organized into packages (e.g., ‘java.util‘, ‘java.io‘).

Example:

- Collections Framework: Handling data structures.
- Streams API: Data processing.
- Concurrency Utilities: Multithreading and parallelism.

Java API: Commonly Used Packages

1. **‘java.lang‘: Core classes.**
   - ‘Object‘: The root class of the Java hierarchy.
   - ‘String‘: Immutable strings.
   - ‘Math‘: Mathematical operations.
2. **‘java.util‘: Utility classes.**
   - ‘ArrayList‘: Resizable arrays.
   - ‘HashMap‘: Key-value pairs.
   - ‘Date‘: Date and time handling.
3. **‘java.io‘: Input and output.**
   - ‘File‘: File operations.
   - ‘BufferedReader‘: Reading text.
   - ‘PrintWriter‘: Writing text.

Example: Using ‘java.util‘ Package

Working with Collections:

```
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Charlie");

        list.forEach(System.out::println);
    }
}
```

Listing 4: ArrayList Example

Advanced API: Java Streams

Definition: A Java Stream is a sequence of elements supporting sequential and parallel aggregate operations.

Example:

```java
import java.util.*;
import java.util.stream.*;

public class StreamsExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.stream()
                         .filter(n -> n % 2 == 0)
                         .mapToInt(n -> n * n)
                         .sum();

        System.out.println("Sum of squares of even numbers: " + sum);
    }
}
```

Listing 5: Streams Example

Concurrency Utilities in Java API

Key Classes:

- 'ExecutorService': Managing thread pools.
- 'ConcurrentHashMap': Thread-safe key-value pairs.
- 'CountDownLatch': Synchronizing threads.

Example: Using 'ExecutorService'

```java
import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable task1 = () -> System.out.println("Task 1");
        Runnable task2 = () -> System.out.println("Task 2");

        executor.submit(task1);
        executor.submit(task2);

        executor.shutdown();
    }
}
```

Listing 6: ExecutorService Example

Advantages of Java API

- Provides prebuilt, well-tested components.
- Saves development time by avoiding reinventing the wheel.
- Encourages best practices and standardization.
- Enhances code readability and maintainability.

Disadvantages of Java API

- Steep learning curve for beginners.
- Potential for misuse or overuse of APIs.
- Some classes and methods may have performance overhead.
- Deprecated APIs can lead to maintenance issues.

Common Mistakes Using Java API

1. Misusing Collections:

```
Map<String, String> map = new HashMap<>();
map.put(null, "Value"); // Null keys allowed in HashMap
```

Listing 7: HashMap Key Issue

2. Ignoring Exceptions:

```
try (BufferedReader br = new BufferedReader(new
    FileReader("file.txt"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    // No handling of exception
}
```

Listing 8: Ignoring IOException

Best Practices for Using Java API

- Read and understand the API documentation.
- Use appropriate APIs for the task (e.g., ‘ArrayList‘ for lists, ‘HashMap‘ for maps).
- Avoid deprecated methods or classes.
- Write unit tests for code using API methods.
- Leverage modern APIs (e.g., Streams, Concurrency Utilities) for better performance and clarity.

Outline

Introduction to Java Packages

Definition: A package in Java is a namespace that organizes classes and interfaces, preventing naming conflicts and improving modularity.

Key Features:

- Facilitates code reusability and organization.
- Controls access using access modifiers.
- Provides built-in libraries and user-defined groupings.

Types of Packages:

- Built-in packages (e.g., ‘java.util‘, ‘java.io‘).
- User-defined packages.

Creating and Using Packages I

Steps to Create a Package:

- Use the ‘package‘ keyword to define a package.
- Compile the file with the package name.
- Use ‘import‘ to access the package in other files.

Example:

```java
// File: MyPackage/MyClass.java
package MyPackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyPackage");
    }
}
```

Listing 9: Creating a Package

Creating and Using Packages II

```java
// File: Main.java
import MyPackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.display();
    }
}
```

Listing 10: Using a Package

Built-in Java Packages

Common Built-in Packages:

- **‘java.lang‘:** Core classes (e.g., ‘String‘, ‘Math‘).
- **‘java.util‘:** Data structures and utilities (e.g., ‘ArrayList‘, ‘HashMap‘).
- **‘java.io‘:** Input/output operations (e.g., ‘File‘, ‘BufferedReader‘).
- **‘java.net‘:** Networking (e.g., ‘Socket‘, ‘URL‘).
- **‘java.sql‘:** Database access (e.g., ‘Connection‘, ‘ResultSet‘).

Advanced Concept: Sub-packages I

Definition: A sub-package is a package within a package, allowing further modularization.

Example:

```java
// File: MyPackage/SubPackage/MySubClass.java
package MyPackage.SubPackage;

public class MySubClass {
    public void display() {
        System.out.println("Hello from SubPackage");
    }
}
```

Listing 11: Creating Sub-packages

Advanced Concept: Sub-packages II

```
import MyPackage.SubPackage.MySubClass;

public class Main {
    public static void main(String[] args) {
        MySubClass obj = new MySubClass();
        obj.display();
    }
}
```

Listing 12: Using Sub-packages

Access Modifiers and Packages

Access Levels:

- **Public:** Accessible from any package.
- **Protected:** Accessible within the same package and subclasses.
- **Default:** Accessible only within the same package.
- **Private:** Not accessible outside the class.

Example:

```
package MyPackage;

public class MyClass {
    public void publicMethod() {}
    protected void protectedMethod() {}
    void defaultMethod() {}
    private void privateMethod() {}
}
```

Listing 13: Access Modifiers

Advantages and Disadvantages of Packages

Advantages:

- Avoids naming conflicts by providing namespaces.
- Enhances modularity and maintainability.
- Facilitates reusability of code.
- Provides access control through access modifiers.
- Simplifies project structure in large applications.

Disadvantages:

- Increases complexity in smaller projects.
- Requires understanding of package structure and imports.
- Mismanagement of package structure can lead to confusion.
- Longer compilation and execution commands.

Common Mistakes with Packages I

1. Misplacing Classes:

```
// File is not in the correct directory matching the
    package name
package MyPackage;
public class MyClass {}
```

2. Ignoring Imports:

```
// Error: MyPackage.MyClass not found
MyClass obj = new MyClass();
```

3. Duplicate Class Names:

```
import java.util.Date;
import java.sql.Date; // Conflict
```

Common Mistakes with Packages II

4. Using Deprecated APIs:

- Relying on outdated methods or classes leads to compatibility issues.
- Example: Using ‘java.util.Date‘ instead of ‘java.time.LocalDate‘.

5. Inadequate Exception Handling: Ignoring exceptions from APIs can cause hidden bugs.

```
File file = new File("example.txt");
file.createNewFile(); // Throws IOException, unhandled
```

6. Overuse of Wildcard Imports: Importing entire packages unnecessarily can lead to name conflicts and increased memory usage.

```
import java.util.*; // Avoid if importing only one class
```

Common Mistakes with Packages III

7. Ignoring Access Modifiers: Misusing ‘protected‘ or ‘default‘ access can expose sensitive functionality.

```
package mypackage;

class SensitiveData {
    String secret = "Hidden"; // Exposed to entire
        package
}
```

Best Practices for Using Packages

- Follow a consistent naming convention (e.g., ‘com.company.project‘).
- Avoid creating excessively deep package hierarchies.
- Group related classes logically.
- Use access modifiers to encapsulate and protect data.
- Leverage built-in packages for standard functionalities before creating custom ones.

Outline

Introduction to Exception Handling in Java

Definition: Exception handling is a mechanism to handle runtime errors, ensuring the normal flow of the application.

Key Concepts:

- An exception is an event that disrupts the normal flow of the program.
- Exception handling uses ‘try‘, ‘catch‘, ‘finally‘, ‘throw‘, and ‘throws‘ keywords.
- Java has a robust hierarchy of exceptions.

Types of Exceptions in Java

1. **Checked Exceptions:**
   - Compile-time exceptions.
   - Must be handled using try-catch or declared using ‘throws‘.
   - Examples: ‘IOException‘, ‘SQLException‘.
2. **Unchecked Exceptions:**
   - Runtime exceptions.
   - Not mandatory to handle.
   - Examples: ‘NullPointerException‘, ‘ArrayIndexOutOfBoundsException‘.
3. **Errors:**
   - Serious issues that the application cannot handle.
   - Examples: ‘StackOverflowError‘, ‘OutOfMemoryError‘.

Basic Exception Handling Syntax

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle exception
} finally {
    // Code that always executes
}
```

Listing 14: Basic Exception Handling

Example:

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // May throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero.");
        } finally {
            System.out.println("Execution complete.");
        }
    }
}
```

Listing 15: Try-Catch Example

Using Multiple Catch Blocks

Example:

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            String text = null;
            System.out.println(text.length()); //
                NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Null value encountered.")
                ;
        } catch (Exception e) {
            System.out.println("General exception: " + e.
                getMessage());
        }
    }
}
```

Listing 16: Multiple Catch Blocks

Nested Try-Catch Blocks

Definition: Nesting allows try-catch blocks within another try block to handle different exceptions at different levels.

Example:

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            try {
                int data = 50 / 0; // ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Division by zero");
            }
            String s = null;
            System.out.println(s.length()); // NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Outer catch: Null value");
        }
    }
}
```

Listing 17: Nested Try-Catch Blocks

Exception Propagation

Definition: Exceptions propagate from the method where they occur to its caller, unless handled.

Example:

```
public class PropagationExample {
    public void method1() {
        int data = 50 / 0; // Throws ArithmeticException
    }

    public void method2() {
        method1(); // Exception propagates here
    }

    public static void main(String[] args) {
        PropagationExample obj = new PropagationExample();
        try {
            obj.method2();
        } catch (ArithmeticException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

Listing 18: Exception Propagation

Advantages of Exception Handling

- Separates error-handling code from regular code.
- Enhances program reliability and robustness.
- Prevents abrupt termination of the program.
- Allows handling of different types of exceptions.
- Enables centralized error logging.

Advanced Exception Handling: Re-Throwing Exceptions

Definition: Re-throwing allows exceptions to be caught and re-thrown to higher levels for additional handling.

Example:

```
public class ReThrowExample {
    public static void processFile() throws IOException {
        throw new IOException("File error");
    }

    public static void main(String[] args) {
        try {
            processFile();
        } catch (IOException e) {
            System.out.println("Handling and re-throwing");
            throw new RuntimeException(e);
        }
    }
}
```

Listing 19: Re-Throwing Exceptions

Best Practices for Exception Handling

- Always catch specific exceptions.
- Use ‘finally‘ or try-with-resources to release resources.
- Avoid using exceptions for normal control flow.
- Provide meaningful error messages.
- Log exceptions for debugging and auditing.
- Use custom exceptions for domain-specific error representation.

Custom Exceptions

Definition: Custom exceptions allow you to create user-defined exceptions specific to your application's needs.

Example:

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

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

    public static void main(String[] args) {
        try {
            validate(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

Listing 20: Custom Exception

Exception Chaining

Definition: Exception chaining is a technique where one exception is linked to another as its cause.

Example:

```
public class ChainingExample {
    public static void main(String[] args) {
        try {
            throw new Exception("Root Cause", new
                NullPointerException("Null Pointer"));
        } catch (Exception e) {
            System.out.println("Exception: " + e.
                getMessage());
            System.out.println("Cause: " + e.getCause());
        }
    }
}
```

Listing 21: Exception Chaining

Outline

Introduction to Multithreading in Java I

Definition: Multithreading is a feature in Java that allows concurrent execution of two or more threads, enabling maximum utilization of CPU.

Key Concepts:

- A thread is the smallest unit of a process.
- Java provides built-in support for multithreading through the ‘Thread‘ class and the ‘Runnable‘ interface.
- Threads can run independently, sharing resources of the same process.

Why Multithreading?

- **Improved Performance:** Utilize CPU cores effectively by executing tasks in parallel.
- **Responsiveness:** Keep the application responsive (e.g., GUIs) by running background tasks.
- **Concurrency:** Allow multiple operations to proceed simultaneously.

Introduction to Multithreading in Java II

Challenges and Caveats:

- **Race Conditions:** When threads access shared resources simultaneously, leading to inconsistent states.
- **Deadlocks:** Threads waiting indefinitely for each other to release resources.
- **Thread Interference:** One thread's modifications overwrite another's changes.
- **Debugging Difficulty:** Multithreading issues are harder to reproduce and debug.

Use Case Example:

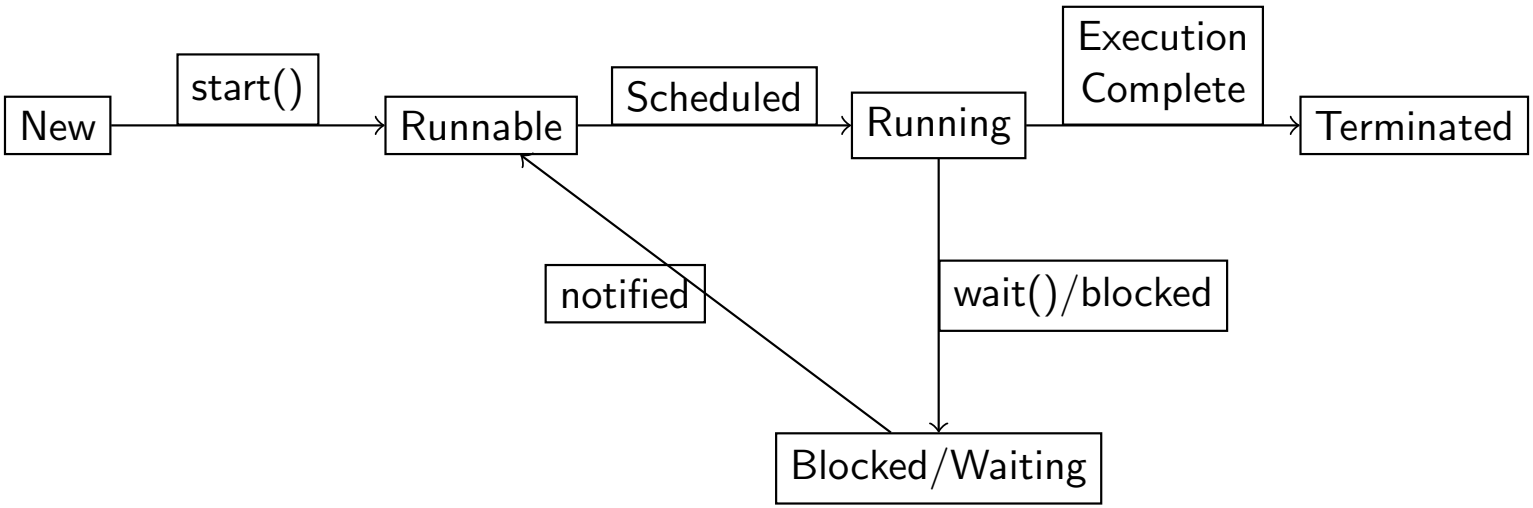
- A web server handling multiple client requests simultaneously using threads.
- Video processing while playing audio in multimedia applications.

Thread Life Cycle

States of a Thread:

- **New:** A thread object is created but not started.
- **Runnable:** A thread is ready to run but waiting for CPU.
- **Running:** A thread is executing.
- **Blocked/Waiting:** A thread is waiting for resources or other threads.
- **Terminated:** A thread completes its execution.

Thread Life Cycle Diagram:



Creating Threads in Java – Two ways

```
class MyThread extends Thread {
    public void run() { System.out.println("Thread is running.");}
}
public class Test {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

Listing 22: Extending Thread Class

```
class MyRunnable implements Runnable {
    public void run() { System.out.println("Thread is running.");}
}
public class Test {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }
}
```

Listing 23: Implementing Runnable Interface

Thread Priorities

Thread Priority Levels:

- **MIN_PRIORITY (1):** Lowest priority.
- **NORM_PRIORITY (5):** Default priority.
- **MAX_PRIORITY (10):** Highest priority.

Setting Priority:

```
Thread thread = new Thread(new MyRunnable());
thread.setPriority(Thread.MAX_PRIORITY);
```

Listing 24: Setting Thread Priority

Note: *Thread priorities are not guaranteed to be respected by the JVM.*

Comparison: ‘Thread‘ vs ‘Runnable‘

Thread Class:

- Inherits the ‘Thread‘ class directly.
- Cannot inherit from another class simultaneously.
- Suitable for simple tasks with no need for multiple inheritance.

Runnable Interface:

- Implements the ‘Runnable‘ interface.
- Allows the class to inherit from other classes.
- Better for complex designs requiring multiple inheritance.

Advantages of Multithreading

- Efficient CPU utilization through concurrent execution.
- Simplifies modeling real-world systems (e.g., producer-consumer problems).
- Enables background processing and asynchronous tasks.

Challenges of Multithreading

- **Deadlocks:** Occur when two or more threads are waiting for each other indefinitely.
- **Race Conditions:** Data inconsistency due to unsynchronized access to shared resources.
- **Thread Starvation:** Lower priority threads are unable to execute.
- **Complexity:** Debugging and testing multithreaded applications is challenging.

Solutions:

- Use proper synchronization.
- Avoid nested locks to prevent deadlocks.
- Use ‘ReentrantLock‘ for advanced locking mechanisms.
- Prefer ‘ExecutorService‘ over manually managing threads.

Synchronization in Multithreading

Definition: Synchronization is the process of controlling access to shared resources by multiple threads to prevent data inconsistency.

Types of Synchronization:

- **Synchronized Method:** Locks the entire method.
- **Synchronized Block:** Locks only a specific block of code.
- **Static Synchronization:** Synchronizes static methods or blocks.

Example: Synchronized Method Using 'Thread' Class

```
class Counter {
    private int count = 0;
    public synchronized void increment() { count++;}
    public synchronized int getCount() { return count;}
}
class CounterThread extends Thread {
    private Counter counter;
    CounterThread(Counter counter) { this.counter = counter;}
    public void run() {
        for (int i = 0; i < 1000; i++) { counter.increment();}
    }
}

public class Test {
    public static void main(String[] args) {
        Counter counter = new Counter();
        CounterThread t1 = new CounterThread(counter);
        CounterThread t2 = new CounterThread(counter);

        t1.start(); t2.start();
        try { t1.join(); t2.join();}
        catch (InterruptedException e) { e.printStackTrace();}

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Listing 25: Synchronized Method

Example: Synchronized Block Using ‘Runnable‘ Interface

```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) { count++;}
    }
    public int getCount() { return count;}
}
class CounterRunnable implements Runnable {
    private Counter counter;
    CounterRunnable(Counter counter) { this.counter = counter;}

    public void run() {
        for (int i = 0; i < 1000; i++) { counter.increment();}
    }
}
public class Test {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new Thread(new CounterRunnable(counter));
        Thread t2 = new Thread(new CounterRunnable(counter));

        t1.start(); t2.start();
        try { t1.join(); t2.join();}
        catch (InterruptedException e) { e.printStackTrace();}

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Listing 26: Synchronized Block

Handling Deadlocks

Definition: Deadlock occurs when two threads are waiting for each other to release resources.

```
class Resource {
    void method1(Resource r) {
        synchronized(this) {
            System.out.println("Inside method1");
            r.method2(this);
        }
    }
    void method2(Resource r) {
        synchronized(this) {
            System.out.println("Inside method2");
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Resource r1 = new Resource();
        Resource r2 = new Resource();

        Thread t1 = new Thread(() -> r1.method1(r2));
        Thread t2 = new Thread(() -> r2.method1(r1));

        t1.start(); t2.start();
    }
}
```

Listing 27: Deadlock Example

Inter-Thread Communication

Mechanism: Threads communicate using ‘wait()‘, ‘notify()‘, and ‘notifyAll()‘.

```
class Message {
    private String content;
    public synchronized void write(String message) {
        this.content = message;
        notify();
    }
    public synchronized String read() {
        try { wait();}
        catch (InterruptedException e) { e.printStackTrace();}
        return content;
    }
}
public class Test {
    public static void main(String[] args) {
        Message message = new Message();

        Thread writer = new Thread(() -> message.write("Hello, World!"));
        Thread reader = new Thread(() -> System.out.println(message.read()));

        reader.start();  writer.start();
    }
}
```

Listing 28: Inter-Thread Communication

Common Mistakes in Thread Programming I

```
class Counter {
    private int count = 0;
    public void increment() { count++;}
    public int getCount() { return count;}
}
public class Test {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) { counter.increment();}
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) { counter.increment();}
        });

        t1.start(); t2.start();

        try { t1.join(); t2.join();}
        catch (InterruptedException e) { e.printStackTrace();}

        System.out.println("Final count (unsynchronized): " + counter.getCount());
    }
}
```

Listing 29: Accessing Shared Resources Without Synchronization

Common Mistakes in Thread Programming II

```
class Deadlock {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void task1() {
        synchronized (lock1) { System.out.println("Task1 acquired lock1");
            synchronized (lock2) { System.out.println("Task1 acquired lock2");}
        }
    }
    public void task2() {
        synchronized (lock2) { System.out.println("Task2 acquired lock2");
            synchronized (lock1) { System.out.println("Task2 acquired lock1");}
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Deadlock deadlock = new Deadlock();
        Thread t1 = new Thread(deadlock::task1);
        Thread t2 = new Thread(deadlock::task2);

        t1.start(); t2.start();
    }
}
```

Listing 30: Deadlock Example

Best Practices for Multithreading

- Minimize shared resources to reduce synchronization overhead.
- Use thread-safe collections (e.g., ‘ConcurrentHashMap‘).
- Prefer higher-level concurrency utilities like ‘ExecutorService‘ and ‘ForkJoinPool‘.
- Avoid busy-waiting; use proper synchronization techniques.
- Test extensively under different scenarios to detect issues early.

Best Practices for Synchronization

- Minimize the scope of synchronization to reduce contention.
- Avoid nested synchronization to prevent deadlocks.
- Use higher-level concurrency utilities like ‘ReentrantLock‘ and ‘ExecutorService‘.
- Prefer immutable objects to reduce synchronization needs.
- Always test for race conditions and deadlocks in multithreaded applications.

Outline

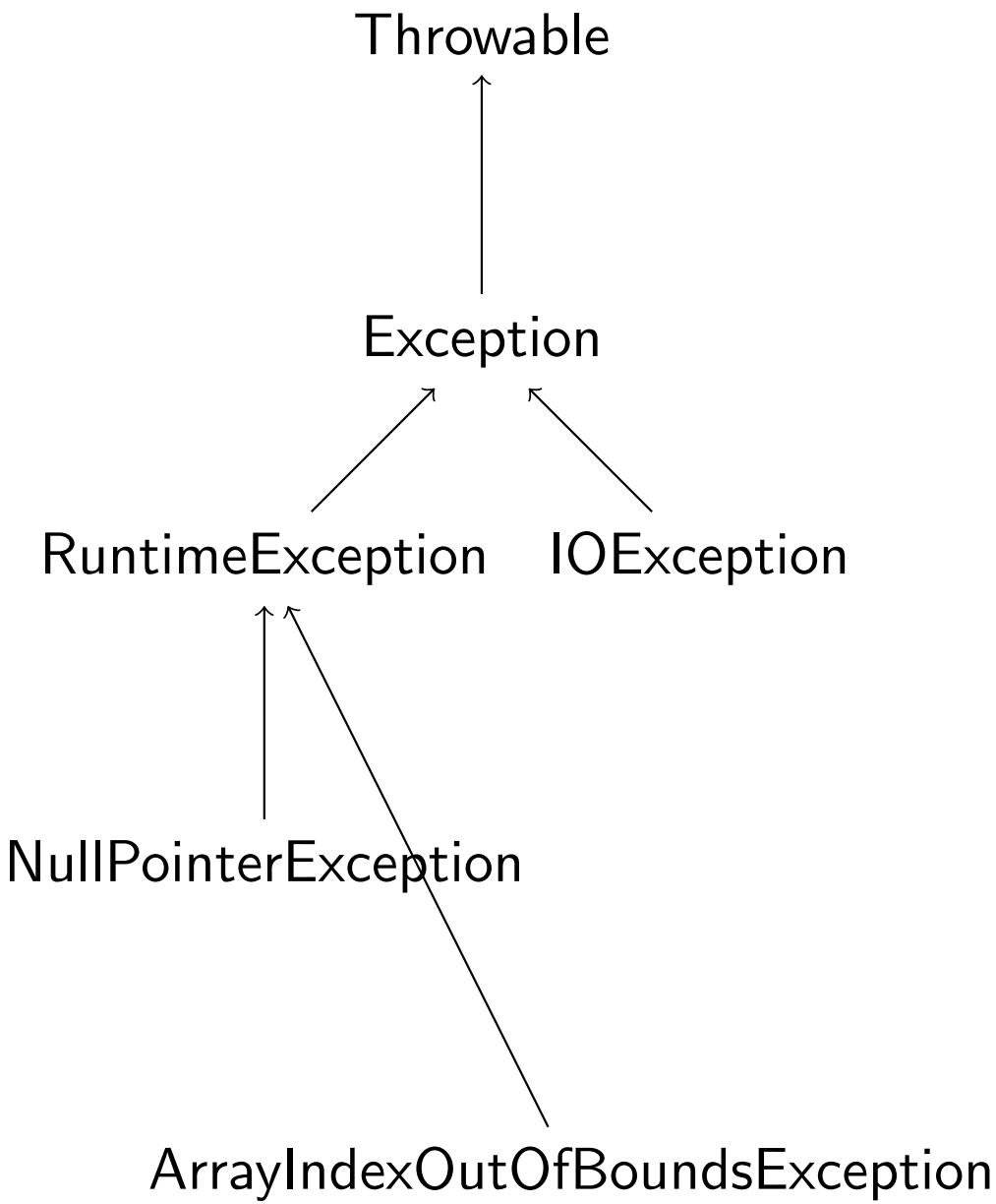
Exercises for Students

1. **Access Modifiers:**
   - Create a class with variables and methods using all access modifiers. Test access from different packages.
2. **Exception Handling:**
   - Write a program to handle a ‘FileNotFoundException‘.
   - Create a user-defined exception for validating passwords.
3. **Multithreading:**
   - Implement a program with two threads: one to print even numbers and another to print odd numbers.
   - Demonstrate thread synchronization with a shared counter.

Discussion Questions

- What are the benefits of using synchronized methods in multithreading?
- How do custom exceptions improve program readability?
- Compare the ‘Thread‘ class and ‘Runnable‘ interface for creating threads.

Exception Hierarchy



Thread Lifecycle Methods

- `start()` - Begins execution of a thread.
- `run()` - Contains thread logic (overridden).
- `sleep(ms)` - Pauses thread execution.
- `join()` - Waits for another thread to finish.
- `interrupt()` - Stops a sleeping/waiting thread.

Thread Control Methods

- `isAlive()` - Checks if a thread is running.
- `setDaemon(true/false)` - Marks a thread as a daemon.
- `setPriority(int)` - Sets thread priority.
- `getPriority()` - Retrieves thread priority.

Static Methods in Thread Class

- `Thread.yield()` - Hints that the current thread is willing to yield execution.
- `Thread.currentThread()` - Returns reference to currently executing thread.
- `Thread.sleep(ms)` - Puts the current thread to sleep for specified milliseconds.
- `Thread.activeCount()` - Returns number of active threads in current thread's group.
- `Thread.holdsLock(obj)` - Checks if the current thread holds the lock on the specified object.

Example: Creating a Thread

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Example: Using join()

```
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Running: " + i);
            try { Thread.sleep(1000); } catch (
                InterruptedException e) {}
        }
    }
}

public class Main {
    public static void main(String[] args) throws
        InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();
        t1.join();
        System.out.println("Main thread finished");
    }
}
```

Example: Interrupting a Thread

```
class MyThread extends Thread {
    public void run() {
        try {
            Thread.sleep(5000);
            System.out.println("Thread completed");
        } catch (InterruptedException e) {
            System.out.println("Thread was interrupted!")
                ;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
        t1.interrupt();
    }
}
```

Example: Using Static Methods in Thread Class

```
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("Current Thread: " +
                Thread.currentThread().getName());
            System.out.println("Active Threads: " +
                Thread.activeCount());
        });
        t1.start();
    }
}
```

Example: Using setName() and getName()

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread Name: " + getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("WorkerThread");
        t1.start();
    }
}
```

Example: Using setPriority() and getPriority()

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread Priority: " +
            getPriority());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setPriority(Thread.MAX_PRIORITY);
        t1.start();
    }
}
```

Example: Using Different Thread Constructors

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(); // Default constructor
        Thread t2 = new Thread(new MyRunnable()); //
            Runnable constructor
        Thread t3 = new Thread(new MyRunnable(), "
            NamedThread");

        t2.start();
        System.out.println("Thread 3 name: " + t3.getName
            ());
    }
}
```

Introduction to Java Thread Synchronization

Why Synchronization?

- Prevents data inconsistency due to concurrent access.
- Ensures thread-safe operations on shared resources.

Synchronization Mechanisms:

- **wait(), notify(), notifyAll()** – Object-level thread coordination.
- **synchronized methods and blocks** – Locks at object level.
- **static synchronized methods and blocks** – Locks at class level.
- **Semaphore** – Allows limited concurrent access.
- **ReadWriteLock** – Optimized for read-heavy scenarios.

Exercise: Why do race conditions occur in multi-threaded applications?

Using `wait()`, `notify()`, and `notifyAll()`

Concept:

- `wait()` – Releases the lock and waits for notification.
- `notify()` – Wakes up one waiting thread.
- `notifyAll()` – Wakes up all waiting threads.

Example: Producer-Consumer Problem

```
class SharedResource {
    private boolean available = false;
    public synchronized void produce() throws InterruptedException {
        while (available) wait();
        System.out.println("Producing...");
        available = true;  notify();
    }
    public synchronized void consume() throws InterruptedException {
        while (!available) wait();
        System.out.println("Consuming...");
        available = false;  notify();
    }
}
```

Exercise: Modify the above code to use `notifyAll()` instead of `notify()`.

Synchronized Methods and Blocks

Synchronization Levels:

- **Synchronized Method:** Locks the entire method.
- **Synchronized Block:** Locks only a critical section.

Example: Synchronized Method

```
class Counter {
    private int count = 0;
    public synchronized void increment() {
        count++;
    }
}
```

Example: Synchronized Block

```
class Counter {
    private int count = 0;
    public void increment() {
        synchronized (this) { count++;}
    }
}
```

Exercise: When would you use a synchronized block instead of a synchronized method?

Producer-Consumer Problem using Synchronization

- Uses `wait()` and `notify()` for inter-thread communication.
- Producer generates data, Consumer consumes it.
- Ensures proper synchronization.

Example: Producer-Consumer Problem

```
class SharedResource {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) throws InterruptedException {
        while (available) wait();
        data = value;
        available = true;
        notify();
    }

    public synchronized int consume() throws InterruptedException {
        while (!available) wait();
        available = false;
        notify();
        return data;
    }
}
```

Using Semaphore for Synchronization

Concept:

- Limits the number of concurrent threads accessing a resource.

Example: Semaphore with Limited Access

```
import java.util.concurrent.Semaphore;

class SharedResource {
    private Semaphore semaphore = new Semaphore(2);

    public void accessResource() throws InterruptedException {
        semaphore.acquire();
        System.out.println(Thread.currentThread().getName() + " accessing resource
            ");
        Thread.sleep(1000);
        semaphore.release();
    }
}
```

Exercise: Modify the code to allow only one thread at a time.

ReadWriteLock for Multi-Threaded Reads/Writes

Concept:

- **Read Lock:** Multiple threads can read simultaneously.
- **Write Lock:** Only one thread can write at a time.

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

class SharedData {
    private int data = 0;
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    public void readData() {
        lock.readLock().lock();
        try {
            System.out.println("Reading Data: " + data);
        } finally { lock.readLock().unlock(); }
    }
    public void writeData(int value) {
        lock.writeLock().lock();
        try {
            data = value;
            System.out.println("Writing Data: " + value);
        } finally { lock.writeLock().unlock(); }
    }
}
```

Exercise: Explain how ReadWriteLock improves performance in read-heavy scenarios.

Executor Framework

- Provides thread pool management.
- Uses `ExecutorService` to manage tasks.
- Efficient for handling multiple concurrent tasks.

```
import java.util.concurrent.*;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(() -> System.out.println("Task 1 running"));
        executor.execute(() -> System.out.println("Task 2 running"));
        executor.shutdown();
    }
}
```

Reentrant Locks

- Alternative to synchronized blocks.
- Allows reentrant behavior (a thread can re-acquire lock it already holds).
- Uses `ReentrantLock` from `java.util.concurrent.locks`.

```
import java.util.concurrent.locks.*;

class Shared {
    private final ReentrantLock lock = new ReentrantLock();

    public void safeMethod() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + " is executing");
        } finally {
            lock.unlock();
        }
    }
}
```

Nested Locks (Deadlock Scenario)

- Deadlock occurs when two threads hold locks that the other needs.
- Careful lock acquisition order prevents deadlocks.

```
class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            synchronized (lock2) {
                System.out.println("Thread 1 running");
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            synchronized (lock1) {
                System.out.println("Thread 2 running");
            }
        }
    }
}
```

ThreadGroup in Java

- Manages multiple threads as a single unit.
- Can set priority and handle uncaught exceptions.
- Helps in security and resource management.

```
public class ThreadGroupExample {
    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("MyGroup");
        Thread t1 = new Thread(group, () -> System.out.println("Thread 1 running"));
        Thread t2 = new Thread(group, () -> System.out.println("Thread 2 running"));

        t1.start();
        t2.start();
        System.out.println("Active Threads in group: " + group.activeCount());
    }
}
```