Java Unit 3

Topics :
Multithreaded Programming : Thread classes - Runnable Interface - synchronization - using synchronized methods - using synchronized statement - interthreaded communication - deadlock
I/O streams : concepts of streams - stream classes - byte and character stream - Reading console input and writing console output - file handling

Write on Your Own Risk ⚠️☠️🚨

Multithreaded Programming

Multithreading is a technique where multiple tasks execute concurrently, sharing resources. It enhances performance and responsiveness, making it useful for background computations, animations, and handling user interactions.

Key Benefits of Multithreading:

1. Efficient CPU utilization – Multiple threads can run on multi-core processors.
2. Improved responsiveness – UI applications remain responsive.
3. Parallel execution – Tasks run simultaneously, reducing execution time.
4. Resource sharing – Threads can share resources like memory.
5. Better performance – No need to create separate processes.

Thread Classes

1. Threads in Java allow multiple operations to run concurrently, improving performance and responsiveness. A thread is a lightweight process that runs independently within a program. Java provides two ways to create a thread:
2. Extending the Thread class
3. Implementing the Runnable interface
4. When a class extends Thread, it must override the run() method, which contains the code that will be executed in the thread. The thread is started using the start() method, which internally calls run() and begins execution in a separate thread.

Syntax :

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

Example:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread running: " + i);
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
        }
    }
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Explanation:

1. The MyThread class extends Thread and overrides run(), which defines the task.
2. The start() method initiates execution, creating a new thread.
3. Thread.sleep(1000) pauses execution for one second.

Runnable Interface

The Runnable interface is another way to create threads in Java. This approach is preferred when a class needs to extend another class, as Java does not support multiple class inheritance. It involves implementing Runnable and defining the run() method.

Syntax:

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

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Runnable running: " + i);
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
        }
    }
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

Explanation:

- Runnable is implemented instead of extending Thread.
- The run() method is defined with the required functionality.
- A Thread object is created using Runnable and started with start().

Important Points:

- Better for flexibility since Java allows multiple interface inheritance.
- Provides a cleaner approach when working with multiple threads.

Synchronization

Synchronization ensures that only one thread can access a shared resource at a time. It prevents data inconsistency caused by race conditions. In Java, synchronization can be implemented using:

- Synchronized methods
- Synchronized blocks

Synchronized Method

Syntax:

```
synchronized void method() { /* Code */ }
```

Example:

```
class Shared {
   synchronized void printNumbers(int n) {
      for (int i = 1; i <= 5; i++) {
         System.out.println(n * i);
         try { Thread.sleep(500); } catch (Exception e) {}
      }
   }
}
class MyThread extends Thread {
   Shared s;
   MyThread(Shared s) { this.s = s; }
   public void run() { s.printNumbers(5); }
}
public class SyncDemo {
   public static void main(String[] args) {
      Shared obj = new Shared();
      MyThread t1 = new MyThread(obj);
      MyThread t2 = new MyThread(obj);
      t1.start();
      t2.start();
   }
}
```

Explanation:

- The printNumbers() method is synchronized, allowing only one thread to execute it at a time.
- If synchronized is removed, both threads can access it simultaneously, leading to inconsistent results.

Important Points:

- Synchronization is essential when multiple threads modify a shared resource.
- Can be applied to methods or specific blocks of code.

Synchronized Statement
A synchronized statement allows synchronization at a more granular level than synchronized methods. Instead of locking an entire method, only a specific block of code is synchronized, reducing contention and improving performance.

Syntax:
```
synchronized(object) {
   // Critical section
}
```

Example of Synchronized Statement

```
class Shared {
   void printNumbers(int n) {
      synchronized (this) { // Synchronized block
         for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
            try { Thread.sleep(500); } catch (InterruptedException e) {}
         }
      }
   }
}
class MyThread extends Thread {
   Shared s;
   MyThread(Shared s) { this.s = s; }
   public void run() { s.printNumbers(5); }
}
public class SyncBlockExample {
   public static void main(String[] args) {
      Shared obj = new Shared();
      MyThread t1 = new MyThread(obj);
      MyThread t2 = new MyThread(obj);
      t1.start();
      t2.start();
   }
}
```

Explanation:

- The synchronized block inside printNumbers() ensures that only one thread can execute that section at a time.
- Instead of synchronizing the entire method, only the critical section (loop printing numbers) is locked.
- Other parts of the method can still execute without blocking.

Important Points:

- Use synchronized statements when only part of a method needs synchronization.
- Reduces unnecessary blocking compared to synchronized methods.
- Always synchronize on a common object (this in this case).

Inter-thread Communication

Inter-thread communication allows threads to interact and coordinate execution. Java provides the wait(), notify(), and notifyAll() methods to facilitate this.

Syntax:

```
synchronized(obj) {
   obj.wait();  // Releases lock and waits
   obj.notify();  // Wakes up one waiting thread
}
```

Example:

```
class Chat {
   boolean flag = false;
   synchronized void Question(String msg) {
      if (flag) try { wait(); } catch (InterruptedException e) {}
      System.out.println(msg);
      flag = true;
      notify();
   }
   synchronized void Answer(String msg) {
      if (!flag) try { wait(); } catch (InterruptedException e) {}
      System.out.println(msg);
      flag = false;
      notify();
   }
}
```

```
class Thread1 extends Thread {
   Chat c;
   Thread1(Chat c) { this.c = c; }
   public void run() { c.Question("What is Java?"); }
}
class Thread2 extends Thread {
   Chat c;
   Thread2(Chat c) { this.c = c; }
   public void run() { c.Answer("Java is a programming language."); }
}
public class InterThread {
   public static void main(String[] args) {
      Chat c = new Chat();
      new Thread1(c).start();
      new Thread2(c).start();
   }
}
```

Explanation:

- One thread asks a question, and the other answers using wait() and notify().
- wait() releases the lock and makes the thread wait, while notify() wakes up a waiting thread.

Important Points:

- wait() and notify() must be inside a synchronized block.
- Used in producer-consumer problems.

Deadlock

A deadlock occurs when two or more threads are waiting indefinitely for each other to release resources.

Example:

```
class A {
    synchronized void methodA(B b) {
        System.out.println("Thread 1: Holding A, waiting for B");
        try { Thread.sleep(1000); } catch (Exception e) {}
        synchronized (b) { System.out.println("Thread 1: Got B"); }
    }
}
class B {
    synchronized void methodB(A a) {
        System.out.println("Thread 2: Holding B, waiting for A");
        try { Thread.sleep(1000); } catch (Exception e) {}
        synchronized (a) { System.out.println("Thread 2: Got A"); }
    }
}
public class DeadlockExample {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        new Thread(() -> a.methodA(b)).start();
        new Thread(() -> b.methodB(a)).start();
    }
}
```

Explanation:

- One thread locks A and waits for B, while the other locks B and waits for A.
- Neither thread can proceed, leading to an infinite wait.

Important Points:

- Avoid nested locks to prevent deadlocks.
- Use timeouts or try-lock mechanisms to detect and recover from deadlocks.

I/O Streams in Java

I/O (Input and Output) streams in Java handle data flow between input sources (like files, keyboards, and network connections) and output destinations (like consoles and files). Java provides various stream classes to facilitate reading and writing data in different formats.

Concepts of Streams

A stream is a sequence of data that flows from a source (input) to a destination (output). Streams are categorized into:

1. Byte Streams: Used for handling raw binary data (e.g., images, audio, video).
2. Character Streams: Used for handling textual data (e.g., reading and writing text files).

Streams are classified into:

- Input Streams: Read data (e.g., FileInputStream, BufferedReader).
- Output Streams: Write data (e.g., FileOutputStream, PrintWriter).

Stream Classes in Java

Java provides several classes for I/O operations:

Byte Stream Classes

- InputStream (abstract class for reading bytes)
  - FileInputStream (reads bytes from a file)
  - BufferedInputStream (buffers input for efficiency)
- OutputStream (abstract class for writing bytes)
  - FileOutputStream (writes bytes to a file)
  - BufferedOutputStream (buffers output for efficiency)

Character Stream Classes

- Reader (abstract class for reading characters)
  - FileReader (reads characters from a file)
  - BufferedReader (buffers character input for efficiency)
- Writer (abstract class for writing characters)
  - FileWriter (writes characters to a file)
  - BufferedWriter (buffers character output for efficiency)

Byte and Character Streams

Byte Stream Example (Reading a file in bytes)

```
import java.io.FileInputStream;
import java.io.IOException;

public class ByteStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("example.txt")) {
            int i;
            while ((i = fis.read()) != -1) {
                System.out.print((char) i); // Converting byte to character
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Explanation:

- FileInputStream reads data byte by byte.
- read() returns an integer representing a byte; -1 indicates the end of the file.
- Each byte is converted to a character before printing.

Character Stream Example (Reading a file in characters)

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

public class CharacterStreamExample {
   public static void main(String[] args) {
      try (FileReader fr = new FileReader("example.txt")) {
         int i;
         while ((i = fr.read()) != -1) {
            System.out.print((char) i);
         }
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Explanation:

- FileReader reads character by character instead of bytes.
- Works better for text files than FileInputStream.

Important Points:

- Byte streams (FileInputStream) handle all file types.
- Character streams (FileReader) are optimized for text files.

Reading Console Input and Writing Console Output
Reading Console Input
Java provides different ways to read user input from the console.
Using Scanner (Recommended for simple input)

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
        scanner.close();
    }
}
```

Explanation:

- Scanner reads input from the console.
- nextLine() reads a full line of input.
- scanner.close() releases the resource.

Using BufferedReader (For efficiency with large inputs)

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

public class BufferedReaderExample {
   public static void main(String[] args) throws IOException {
      BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
      System.out.print("Enter your age: ");
      int age = Integer.parseInt(br.readLine());
      System.out.println("Your age is: " + age);
   }
}
```

Explanation:

- BufferedReader reads input efficiently using buffering.
- InputStreamReader converts System.in (byte stream) to a character stream.
- readLine() reads a full line as a string.

File Handling in Java

File handling allows reading and writing data to files using FileReader, FileWriter, BufferedReader, and BufferedWriter.

Writing to a File (Character Stream)

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

public class FileWriteExample {
   public static void main(String[] args) {
      try (FileWriter fw = new FileWriter("output.txt")) {
         fw.write("Hello, Java File Handling!");
         System.out.println("Data written to file.");
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Explanation:

- FileWriter writes character data to a file.
- write() writes text to the file.
- try-with-resources ensures the file is closed automatically.

Reading from a File (Character Stream)

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

public class FileReadExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("output.txt")) {
            int i;
            while ((i = fr.read()) != -1) {
                System.out.print((char) i);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Explanation:

- FileReader reads characters from the file.
- read() method returns characters one by one.

Appending to a File

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

public class FileAppendExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("output.txt", true)) {
            fw.write("\nAppending new data.");
            System.out.println("Data appended.");
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Explanation:

- FileWriter("output.txt", true) enables append mode.
- New content is added without overwriting the existing file data.

Important Points

- Streams are categorized as Byte Streams and Character Streams.
- Use FileReader and FileWriter for text files.
- Use FileInputStream and FileOutputStream for binary files (images, audio, etc.).
- Use BufferedReader for efficient reading of large text files.
- Always close streams (close()) or use try-with-resources for automatic closure