Java Multithreading Lecture: Understanding Concurrent Execution

Introduction to Multithreading:

- **Definition:** Multithreading is a concurrent execution of two or more threads. A thread is a lightweight sub-process, and multithreading allows multiple threads to exist within the same process.
- **Why Multithreading?**
  - **Parallelism:** Efficient utilization of CPU resources by executing multiple threads simultaneously.
  - **Responsiveness:** Allows a program to remain responsive to user interactions.
  - **Modularity:** Threads provide a way to break a program into smaller, manageable parts.

Basics of Threads in Java:

- **Thread Class:** Java provides the **Thread** class to create and control threads.

java code

```
class MyThread extends Thread {
 public void run() {
// Code to be executed in the new thread
 } }
```

- **Creating and Starting Threads:**
  - Create an instance of your **Thread** subclass.
  - Call the **start()** method to begin the execution of the thread.

java code

```
MyThread myThread = new MyThread();
myThread.start();
```

Example Program: Simple Multithreading in Java

Let's consider a scenario where we want to print numbers from 1 to 5 using two threads.

Java code

```
class NumberPrinter extends Thread {
private int start;
public NumberPrinter(int start) {
this.start = start; }
public void run() {
for (int i = start; i <= start + 4; i++) {
 System.out.println(Thread.currentThread().getId() + " : " + i);
 } } }
public class MultiThreadExample {
public static void main(String[] args) {
NumberPrinter thread1 = new NumberPrinter(1);
 NumberPrinter thread2 = new NumberPrinter(6); // Start the threads
thread1.start();
thread2.start();
} }
```

In this example, we have a **NumberPrinter** class that extends **Thread**. Each instance of **NumberPrinter** prints five consecutive numbers. The **main** method creates two threads and starts them. Due to multithreading, you might see interleaved output of numbers from both threads.

Key Concepts:

- **Thread Lifecycle:** Threads have a lifecycle including new, runnable, blocked, waiting, and terminated states.
- **Synchronization:** Ensuring that multiple threads do not interfere with each other while accessing shared resources.
- **Thread Safety:** Writing code that behaves correctly when executed by multiple threads.

Multithreading is a powerful concept that enhances the performance and responsiveness of Java applications. Understanding the basics and implementing it judiciously is crucial for developing efficient and responsive software.

Advanced Java Multithreading : Synchronization, Thread Pools, and java.util.concurrent

1. Synchronization in Multithreading:

- **Why Synchronization?**
    - In multithreading, when multiple threads access shared resources concurrently, it can lead to data inconsistency.
    - Synchronization ensures that only one thread can access a shared resource at a time.
- **Synchronized Methods:**
    - Use the **synchronized** keyword to make a method thread-safe.

Java code

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

- **Synchronized Blocks:**
    - Instead of synchronizing entire methods, you can use synchronized blocks for more fine-grained control.

Java code

```
class SharedResource {
private int count = 0;
private Object lock = new Object();
public void increment() {
synchronized (lock) {
count++;
} } }
```

2. Thread Pools:

- **What are Thread Pools?**
    - A thread pool is a collection of worker threads that efficiently execute tasks.
    - Reduces thread creation overhead and provides better control over the number of concurrent threads.
- **Creating a Thread Pool:**
    - Java provides the **ExecutorService** interface for managing thread pools.

java code

```
ExecutorService executor = Executors.newFixedThreadPool(5);
```

- **Submitting Tasks to a Thread Pool:**
    - Use the **submit()** method to submit tasks for execution.

java code

```
executor.submit(() -> {
// Task logic
 });
```

- **Shutting Down the Thread Pool:**
    - Always shut down the thread pool when it's no longer needed.

java code

```
executor.shutdown();
```

3. java.util.concurrent Package:

- **Introduction:**
    - The **java.util.concurrent** package provides a framework for concurrent programming.

    - It includes high-level concurrency utilities beyond basic thread management.
- **Key Classes:**
    - **ExecutorService**: Manages and controls thread execution.
    - **Future**: Represents the result of an asynchronous computation.
    - **Semaphore**: Controls the number of threads that can access a resource.
- **Example using ExecutorService and Future:**

java code

```
ExecutorService executor = Executors.newFixedThreadPool(3);
List<Future<String>> futures = new ArrayList<>();

for (int i = 0; i < 5; i++) {

Future<String> future = executor.submit(() -> {

// Task logic return "Task completed";

});

futures.add(future);

} // Retrieve results when tasks are done

for (Future<String> future : futures) {

try {

 String result = future.get(); System.out.println(result);

}

catch (InterruptedException | ExecutionException e) {

 e.printStackTrace();

}

}
```

```
executor.shutdown();
```

Advanced multithreading concepts like synchronization, thread pools, and the **java.util.concurrent** package offer powerful tools for managing concurrent execution in Java. These techniques enhance performance, improve resource utilization, and simplify the development of robust multithreaded applications.