Absolutely! Here's a complete, beginner-to-advanced explanation of Multithreading in Java, based on the official Oracle Java Documentation. This includes:

🚀 Multithreading in Java – Full Guide

🧠 What is Multithreading?

Multithreading is the ability of a program to execute multiple threads concurrently. It helps in performing many tasks at the same time and makes efficient use of CPU resources.

Java supports multithreading by:

- Providing the Thread class
- Providing the Runnable interface
- Providing the java.util.concurrent package

🔧 Ways to Create Threads

✅ 1. By Extending the Thread class

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

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();  // start() calls run()
    }
}
```

✅ 2. By Implementing the Runnable Interface

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

```
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

✅ 3. Using Lambda Expressions (Java 8)

```
public class LambdaThread {
    public static void main(String[] args) {
        Runnable task = () -> System.out.println("Thread using Lambda");
        new Thread(task).start();
    }
}
```

📝 Thread Lifecycle

1. New – Thread is created.
2. Runnable – Ready to run (after start() is called).
3. Running – CPU assigns time to run.
4. Blocked/Waiting – Waiting for resource or signal.
5. Terminated – Completed execution or stopped.

🔁 Thread Methods

Method	Description
start()	Starts the thread and calls run()
run()	Entry point of the thread logic
sleep(ms)	Pauses execution for given milliseconds
join()	Waits for a thread to die
yield()	Pauses current thread and gives chance to others

Method	Description
isAlive()	Checks if thread is still running

🧰 Thread Priorities

Each thread has a priority (default = 5)

Constant	Value
Thread.MIN_PRIORITY	1
Thread.NORM_PRIORITY	5
Thread.MAX_PRIORITY	10

Threads with higher priority may get more CPU time (not guaranteed).

⚔️ Thread Synchronization

Multiple threads accessing shared resources can lead to race conditions.

✅ Synchronized Methods

```
synchronized void update() {
    // thread-safe code
}
```

✅ Synchronized Blocks

```
void print() {
    synchronized(this) {
        // only one thread can enter here
    }
}
```

💣 Inter-thread Communication (wait/notify)

Used when threads want to communicate and coordinate.

Method	Description
wait()	Pauses until notify() is called

Method	Description
notify()	Wakes up one waiting thread
notifyAll()	Wakes all waiting threads

Must be used inside synchronized block.

🔩 Thread-safe Classes (from java.util.concurrent)

- ReentrantLock – advanced locking
- ExecutorService – manages thread pools
- AtomicInteger – thread-safe integers
- ConcurrentHashMap – safe map in multithreaded context
- Semaphore, CountDownLatch, CyclicBarrier – advanced coordination

---

🔐 Daemon Threads

Daemon threads are background services (e.g., Garbage Collector). JVM stops when only daemon threads are left.

```
Thread t = new Thread(() -> {...});
t.setDaemon(true);
```

🎯 Real-Life Example: Download Manager

```
class DownloadTask extends Thread {
    private String fileName;

    DownloadTask(String fileName) {
        this.fileName = fileName;
    }

    public void run() {
        System.out.println("Downloading " + fileName + " by " +
Thread.currentThread().getName());
```

```
    }

    public static void main(String[] args) {
        new DownloadTask("file1.zip").start();
        new DownloadTask("file2.zip").start();
        new DownloadTask("file3.zip").start();
    }
}
```

📝 Best Practices

- Avoid Thread.sleep() in production
- Always use ExecutorService for large-scale apps
- Use synchronization only where necessary
- Prefer immutability for shared objects

Awesome! Let's break down these advanced multithreading topics in a clear, practical, beginner-friendly way — all based on the official Java documentation.

📝 1. Thread Pool with ExecutorService

🔹 Why use it?

Instead of creating a new thread for every task, which is costly, a Thread Pool reuses a fixed number of threads to execute many tasks — it's faster and more scalable.

✅ Code Example:

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

public class ThreadPoolExample {
    public static void main(String[] args) {
        // Create a thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);
```

```
        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " is running by " +
Thread.currentThread().getName());
            });
        }

        executor.shutdown(); // Prevents new tasks, allows existing to finish
    }
}
```

🔸 Key Methods:

- submit(Runnable task): Submits task
- shutdown(): Graceful shutdown (waits for tasks to finish)
- shutdownNow(): Force shutdown (interrupts tasks)
- awaitTermination(): Waits for pool to finish

🛠️ 2. ReentrantLock & Deadlocks

🔹 What is ReentrantLock?

A more flexible alternative to synchronized, with advanced features:

- Try lock with timeout
- Interruptible locking
- Manual lock/unlock

✅ Basic Example:

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

public class LockExample {
    private final ReentrantLock lock = new ReentrantLock();

    public void print() {
```

```
        lock.lock();
        try {
            System.out.println("Thread " + Thread.currentThread().getName() + " is printing...");
        } finally {
            lock.unlock(); // Always unlock in finally
        }
    }

    public static void main(String[] args) {
        LockExample obj = new LockExample();
        Runnable task = obj::print;

        new Thread(task).start();
        new Thread(task).start();
    }
}
```

⚠️ Deadlock Example:

Two threads trying to lock the same two objects in reverse order.

class A { synchronized void methodA(B b) { synchronized(b) {} } }

class B { synchronized void methodB(A a) { synchronized(a) {} } }

To avoid deadlocks:

- Use lock ordering
- Use tryLock() with timeout

🧰 3. Thread-safe Collections

Java provides concurrent collections to safely share data between threads without explicit synchronization.

✅ Common Classes:

Class	Description
ConcurrentHashMap	Thread-safe map
CopyOnWriteArrayList	Safe list for read-heavy operations
BlockingQueue	Safe queue with blocking operations
ConcurrentSkipListMap	Thread-safe sorted map

- Example: ConcurrentHashMap

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

public class SafeMap {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

        Runnable writer = () -> {
            map.put(Thread.currentThread().getName(), 1);
            System.out.println("Written by " + Thread.currentThread().getName());
        };

        new Thread(writer).start();
        new Thread(writer).start();
    }
}
```

- Example: BlockingQueue with Producer-Consumer

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

public class ProducerConsumer {
    public static void main(String[] args) {
        BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);
```

```
        // Producer
        new Thread(() -> {
            try {
                queue.put(1); // waits if full
                System.out.println("Produced");
            } catch (InterruptedException e) {}
        }).start();

        // Consumer
        new Thread(() -> {
            try {
                Thread.sleep(1000);
                int value = queue.take(); // waits if empty
                System.out.println("Consumed: " + value);
            } catch (InterruptedException e) {}
        }).start();
    }
}
```

📝 Summary Table

Concept	Purpose	Interface/Class
Thread Pool	Manage & reuse threads	ExecutorService
Flexible Locking	Lock with more control	ReentrantLock
Avoid thread issues	Share data safely	ConcurrentHashMap, etc.

🧠 **Preemptive Scheduling**

In **Preemptive Scheduling**, the operating system has the **authority to forcibly stop (preempt)** a running thread to give CPU time to another thread. In this case, the **scheduler** decides which thread

will run next, and it may interrupt a running thread at any time, regardless of whether it has finished its task.

- **Characteristics of Preemptive Scheduling:**
  - The operating system **actively interrupts** threads and can suspend them.
  - Ensures **fair distribution** of CPU time across all threads.
  - Thread priorities play a major role — higher priority threads are executed first.
  - Thread can be preempted if:
    - A higher priority thread becomes available.
    - The time slice (quantum) of a thread expires.
- **Example:**

In **Java**, the Java **Thread Scheduler** generally follows **preemptive** scheduling for threads that are part of the operating system's kernel. For instance:

- A **high priority thread** can preempt a lower priority thread from execution.

- **Advantages:**
  - **Fairness** in multi-tasking environments.
  - **Responsive** systems since threads can be interrupted as needed.
- **Disadvantages:**
  - **Overhead** from frequent context switching (interrupting and resuming threads).
  - **Race conditions** might occur when shared resources are accessed simultaneously.

🧠 Non-Preemptive Scheduling

In **Non-Preemptive Scheduling**, once a thread starts executing, it **runs to completion** unless it voluntarily gives up control (e.g., it completes its task, sleeps, or waits for input). The **operating system cannot forcibly interrupt a running thread**; it can only wait for the thread to yield control back to the scheduler.

- **Characteristics of Non-Preemptive Scheduling:**
  - The **thread itself must yield control** (via yield() or sleep()).
  - **Priority** is considered, but only **after** the current thread voluntarily gives up control.
  - The scheduler does not forcibly preempt threads; hence, threads can run indefinitely if they don't yield.
- **Example:**

In Java, this can happen in scenarios where thread control is entirely dependent on the thread's **voluntary actions** like waiting for input or performing a sleep operation. If threads do not yield or give up control, they keep running.

- **Advantages:**
  - **Lower overhead** as context switching is reduced.
  - **More predictable** behavior, as threads run until completion.
- **Disadvantages:**
  - **Starvation**: Low-priority threads may never get CPU time if high-priority threads always finish first.
  - **Less fairness** in allocating CPU time between threads.

Comparison:

Feature	Preemptive Scheduling	Non-Preemptive Scheduling
Thread Control	OS can interrupt and stop threads	Threads control their own execution
CPU Allocation	OS decides which thread gets the CPU	Threads run until they voluntarily release CPU
Fairness	High (fair distribution of CPU)	Low (higher-priority threads can monopolize CPU)
Overhead	Higher (due to frequent context switching)	Lower (less context switching)
Risk of Starvation	Low (due to fairness in scheduling)	High (if lower-priority threads aren't yielding)
Used In	Modern OS with multithreading	Older OS or systems with simpler thread management

---

Example in Java:

Java uses a **preemptive scheduling model** with the default **Thread Scheduler**. However, it also depends on the underlying OS (e.g., Windows, Linux) for exact scheduling policies. You can control **thread priorities** (but not preemption directly) by using methods like Thread.setPriority().

```
public class PreemptiveExample {
    public static void main(String[] args) {
        Thread lowPriority = new Thread(() -> {
            System.out.println("Low Priority Thread Running");
```

```
        });
        lowPriority.setPriority(Thread.MIN_PRIORITY);

        Thread highPriority = new Thread(() -> {
            System.out.println("High Priority Thread Running");
        });
        highPriority.setPriority(Thread.MAX_PRIORITY);

        lowPriority.start();
        highPriority.start();
    }
}
```

In this example, even though the **low-priority thread** starts first, the **high-priority thread** might be **preempted** by the operating system to run earlier (depending on OS scheduling).

Would you like to explore this more with specific examples or use cases in real-world applications?