```java
//Differnet ways to craete the thread:
class Mythread extends Thread {
    @Override
    public void run() {
        System.out.println("Running the thread 1");
    }
}
public class Threads {
    public static void main(String[] args) {
        Mythread t1 = new Mythread();
        t1.start();

        // we have public Thread(Runnable target) constructor
        Thread t2 = new Thread(() -> {
            System.out.println("Running the thread 2");
        });
        t2.start();

        // with Anonymous Inner Class
        Runnable r = new Runnable() {
          @Override
          public void run() {
              System.out.println("Running the thread 3");
          }
        };

        // so that we can implement using the abobe lambda exparession
        // as runnable is functional Interface with one abstarct method
        Runnable z = () -> {
            System.out.println("Running the thread 4");
            System.out.println("thread name " + Thread.currentThread().getName());
        };

        Thread t4 = new Thread(z, "thread name dancer");
        t4.start();

        Thread t3 = new Thread(r);
        t3.start();
    }
}
output :
Running the thread 1
Running the thread 2
Running the thread 4
Running the thread 3
thread name thread name dancer


 ******Volatile******
 Visibility: volatile ensures that changes to a variable are visible to all threads
 Without volatile, threads might cache variables value from thread cache, but using
volatile the value will be picked from RAM [Ram will have the updated value] and it
will be shared same value across all the threads.

public class VolatileExample {
    private volatile boolean flag = false;
```

```
    public void writer() {
        flag = true; // Writes to the volatile variable
    }

    public void reader() {
        while (!flag) {
            // Wait until the flag becomes true
            // Infinite loop will not run and we will save it
        }
        System.out.println("Flag is true!");
    }

    public static void main(String[] args) {
        VolatileExample example = new VolatileExample();

        Thread writerThread = new Thread(example::writer);
        Thread readerThread = new Thread(example::reader);

        readerThread.start();
        writerThread.start();
    }
}

Output:
Flag is true!

*****Atomic*******
The use of Atomic classes is crucial in scenarios where you need to perform atomic
operations on variables shared between multiple threads, ensuring data consistency
without using explicit synchronization mechanisms like synchronized blocks or Locks.

mport java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounterExample {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet(); // Atomic increment
    }

    public int getCount() {
        return count.get();
    }

    public static void main(String[] args) throws InterruptedException {
        AtomicCounterExample counter = new AtomicCounterExample();

        // Two threads incrementing the count
        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();

        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount());  // This will
always be 2000
    }
}
```

```
***************
Thread State: Thread lifecycle

public class MyThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep(2000);
            System.out.println("CHILE THREAD 1 " + Thread.currentThread().getState
());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // see here we have not created any thread just object of the class act as
thread
        MyThread t = new MyThread();
        System.out.println("CHILE THREAD 0 " + t.getState());
        t.start();
        System.out.println("CHILE THREAD 2 " + t.getState());
        Thread.sleep(100);
        System.out.println("CHILE THREAD 3 " + t.getState());
        // now first t will complete then Main thread will execute again
        t.join();
        System.out.println("CHILE THREAD 4 " + t.getState());
        System.out.println("MAIN THREAD " + Thread.currentThread().getState());
    }
}

Output
CHILE THREAD 0 NEW
CHILE THREAD 2 RUNNABLE
CHILE THREAD 3 TIMED_WAITING
CHILE THREAD 1 RUNNABLE
CHILE THREAD 4 TERMINATED
MAIN THREAD RUNNABLE
```

Purpose of Thread Priorities
Thread Scheduling: The priority of a thread is a hint to the JVM about how to prioritize the thread relative to other threads. A higher-priority thread may be given more CPU time than a lower-priority thread. However, the actual scheduling depends on the JVM and OS's thread scheduling policies.

Prioritization: Setting priorities allows developers to influence the order in which threads are executed. For example, you might set a higher priority for a thread that performs critical tasks and a lower priority for background tasks.

```
highPriorityThread.setPriority(Thread.MAX_PRIORITY);
lowPriorityThread.setPriority(Thread.MIN_PRIORITY);
```

```
Thread.Interrupted;
[when it is called, thread will stop its procees if its in sleep or running and exception will be thrown which need to be handle by catch]
```

```
public class InterruptExample {
    public static void main(String[] args) {
        Thread worker = new Thread(() -> {
            try {
                while (!Thread.currentThread().isInterrupted()) {
                    // Simulate work
                    Thread.sleep(1000);
                    System.out.println("Working...");
                }
            } catch (InterruptedException e) {
                // Handle the interruption
                System.out.println("Thread was interrupted");
                // Re-set the interrupt flag
                // Such that while loop stop the excution
                Thread.currentThread().interrupt();
                // another wat to stop just by return
                //return;
            }
        });

        worker.start();

        // Interrupt the thread after 3 seconds
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        worker.interrupt();
    }
}
```

```
********************
```

```
public class RestartThreadExample implements Runnable {

@Override
public void run() {
```

```
    for (int i = 0; i < 5; i++) {
        System.out.println("Thread running: " + i);
        try {
            Thread.sleep(1000); // Simulating long-running task
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted, exiting...");
            return; // Exit when interrupted
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    // Create and start the first thread
    Thread thread = new Thread(new RestartThreadExample());
    thread.start();

    Thread.sleep(2000); // Main thread waits for 2 seconds
    thread.interrupt(); // Interrupt the first thread

    // Wait for the first thread to finish execution
    thread.join();

    // Create and start a new thread since the previous one cannot be restarted
    System.out.println("Starting a new thread...");
    Thread newThread = new Thread(new RestartThreadExample());
    newThread.start();
}
}

output
Thread running: 0
Thread running: 1
Thread interrupted, exiting...
Starting a new thread...
Thread running: 0
Thread running: 1
Thread running: 2
Thread running: 3
Thread running: 4

Key learning
Interrupting the First Thread: The first thread is interrupted while it is sleeping,
causing it to exit early. It only runs for 2 iterations before being interrupted.
Creating a New Thread: After the first thread finishes, the main thread creates a new
 thread and starts it. The new thread runs for 5 full iterations without interruption
.

**************************
Thread.yield()
It will ask scheduler that , please now give chance to other thread, if two threads
are running that each thraed will run periodically.

Daemon Thread: A daemon thread is a thread that runs in the background, When all non-
daemon threads (user threads) have finished, the JVM will terminate, even if there
are still daemon threads running.
```

```
background services such as logging, monitoring, or periodic maintenance tasks
The Java Virtual Machine (JVM) uses daemon threads for garbage collection.
Tasks that need to be executed periodically but do not affect the main application
flow

public class DaemonThreadExample {
    public static void main(String[] args) {
        Thread daemonThread = new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    System.out.println("Daemon thread is running...");
                } catch (InterruptedException e) {
                    System.out.println("Daemon thread interrupted.");
                }
            }
        });

        // Set the thread as daemon
        daemonThread.setDaemon(true);
        daemonThread.start();

        // Main thread sleeps for 5 seconds
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread is finishing. Daemon thread will be
terminated.");
    }
}

************************************Locks
************************************************
Lock is one Interface,

public class ThreadExample {
    public void method1() {
        System.out.println("Method1 is running");
    }

    public static void main(String[] args) {
        ThreadExample example = new ThreadExample();

        // Create a Thread with method1 as the Runnable target
        Thread t1 = new Thread(example::method1);

        // Start the thread
        t1.start(); // This will internally call example.method1() in the new thread
    }
}

Thread t1 = new Thread(example::method1);  =  () -> example.method1()
            ||
```

```
public Thread(Runnable target) {
    this.target = target;
}
            ||
@FunctionalInterface
public interface Runnable {
    void run();
}
```

example::method1 is a method reference that is equivalent to a lambda expression () -> example.method1().

example::method1 matches the Runnable interface because method1 has no parameters and returns void, fitting the run method signature in the Runnable interface.