```java
/*

Multitasking: Executing several tasks simultaneously is the concept of multitasking.
There are two types of multitasking's.
        1. Process based multitasking
        2. Thread based multitasking
 */


/*
CASE 1: Define, Instiantiate and Start a new thread
--> Thread can be created by Two way:
        1. By extending Thread Class (java.lang.Thread)
        2. By implementing Runnable interface (java.lang.Runnable)

--> Thread Scheduler:
    >> If multiple threads are waiting to execute then which Thread will execute 1st is
decided by "Thread Scheduler" which is
        part of JVM
    >> Which algoritham or behaviour followed by Thread Scheduler we can't expect exactly.
    >> It is the JVM vendor dependent hence in multithreading examples we can't expect
exact execution order and exact output.
 */

class CThread1 extends Thread {
    public void run() {
        for (int i = 0; i <= 3; i++) { // This is the job of Thread
            System.out.println("Thread class Run method");
        }
    }
}

public class DMT1 {
    public static void main(String[] args) {
        CThread1 ct1 = new CThread1();
        ct1.start();
        for (int i = 0; i <= 3; i++) {
            System.out.println("Main method");
        }
    }
}
```

```
/*

CASE 2: Difference between start() and run() methods

>> In the case of start(), a new thread will be created which is responsible for the
execution
   of run() method.
>> But in case of run() method no new thread will be created and run() method will be
executed
   just like a normal method by the main Thread.
>> In the Case 1 program, if we replace start() method with run() method the output is
changed.

CASE 3: Importance of Thread class start() method

>> For every thread the required mandarotry activity like registering the Thread with
   Thread Scheduler will be taken care by Thread class start() method.
>> Programerr is responsible just to define the job of Thread inside run() method.
>> That is start() method acts as best assistant to the programmer.

EX: start(){
    1. Register thread with Thread Scheduler.
    2. All other mandatory low level activities.
    3. Invoke or Calling run() method.
}

--> Conclusion: Without executing Thread class start() method, there is no chance of
starting
    new thread in java.
 */
class CThread2 extends Thread {
    public void run() {
        for (int i = 0; i <= 3; i++) { // This is the job of Thread
            System.out.println("Thread class Run method");
        }
    }
}

public class DMT2 {
    public static void main(String[] args) {
        CThread2 ct2 = new CThread2();
        ct2.run();
        for (int i = 0; i <= 3; i++) {
            System.out.println("Main method");
        }
    }
}
```

```java
/*

CASE 4: If we are not overriding run() method

--> If we are not overriding run() method then Thread class run() method will be executed
which has
    empty implementation and hence we won't get any output.
 */

 class CThread3 extends Thread {

    }

    public class DMT3 {
        public static void main(String[] args) {
            CThread3 ct3 = new CThread3();
            ct3.run();
        }
    }

/*

It is highly recommended to override run() method. Otherwise don't go for multithreading
concept.

 */
```

```java
/*
 CASE 5: Overloading of run() method

 --> We can overload run() method but Thread class start() method always invokes no
argument
     run() method the other overload run()method we have to call explicitly then only it
will be
     executed just like normal method.

 */

class CThread4 extends Thread {
    public void run() {
        System.out.println("No-Arg run method");
    }

    public void run(int i) {
        System.out.println("int-Arg run method");
    }
}

public class DMT4 {
    public static void main(String[] args) {
        CThread4 ct4 = new CThread4();
        ct4.start(); // Calls only no-arg run() method
    }
}
```

```java
/*

CASE 6: Overriding of start() method

--> If we override start() method then our start() method will be executed just like a
normal
    method call and no new thread will be started.

 */

 class CThread5 extends Thread {
    public void run() {
        System.out.println("Run method");
    }

    public void start() {
        System.out.println("Start method");
    }
}

public class DMT5 {
    public static void main(String[] args) {
        CThread5 ct5 = new CThread5();
        ct5.start();
        System.out.println("Main method");
    }
}
```

```
/*

CASE 7: Use of super.start() method:


CASE 8: Life Cycle of Thread:

1. New / Born :
    >> MyThread t = new MyThread();
    >> Once we create a Thread object then the Thread is said to be in new state or born
state

2. Ready / Runnable:
    >> t.start() - Once we call start() method then the Thread will be entered into Ready
or Runnable state.

3. Running:
    >> If Thread Scheduler allocates CPU then the Thread will be entered into running state
.
    >> It means using run() method will allow Thread to enter into running state.

4. Blocked:
    >> Once we call suspend(), sleep() or wait() method, Thread will enter into Blocked
state.
    >> After blocking, Resuming is compulsory.
    >> resume() method will again lead Thread to Runnable state.

5. Dead:
    >> Once run() mehtod completes then the Thread will enter into dead state.
    >> Again, if we call stop() method then Thread will enter into dead state.

 */

class CThread6 extends Thread {
    public void run() {
        System.out.println("Run method");
    }

    public void start() {
        super.start();
        System.out.println("Start method");
    }
}

public class DMT6 {
    public static void main(String[] args) {
        CThread6 ct6 = new CThread6();
        ct6.start();
        System.out.println("Main method");
    }
}
```

```java
/*

CASE 9:

--> After starting a Thread, we are not allowed to restart the same Thread once again otherwise
    we will get runtime exception saying "IllegalThreadStateException".

 */

class CThread7 extends Thread {
    public void run() {
        System.out.println("Run method");
    }
}

public class DMT7 {
    public static void main(String[] args) {
        CThread7 ct7 = new CThread7();
        ct7.start();
        // ct7.start(); // R.E.: Exception in thread "main" java.lang.IllegalThreadStateException
        System.out.println("Main method");
    }
}
```

```java
/*

Defining a Thread by implementing Runnable interface:  (2nd Approach)

1st approach : MyThread --> Thread -> Runnable
2nd approach : MyRunnable --> Runnable

--> Runnable interface contains only one method run().

 */

class CRunnable1 implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Child Thread");
        }
    }
}

public class DMT8 {
 public static void main(String[] args) {
        CRunnable1 cr1 = new CRunnable1();
        Thread tr1 = new Thread(cr1); // Here, cr1 is target runnable
        tr1.start();
        // Thread t = new Thread(new CRunnable1()); (It is also possible)
        for (int i = 0; i < 5; i++) {
            System.out.println("Main Thread");
        }
    }
 }

 // We can't expect exact output but there are several possible outputs.
```

```
/*

Case Study:
    MyRunnable r = new MyRunnable();
    Thread t1 = new Thread();
    Thread t2 = new Thread(r);


CASE 1: t1.start():
    >> A new thread will be created which is responsible for the execution of Thread class
run() method.

CASE 2: t1.run():
    >> No new thread will be created but Thread class run() method will be executed just
like a normal method call.

CASE 3: t2.start():
    >> New Thread will be created which is responsible for the execution of MyRunnable run
() method.

CASE 4: t2.run():
    >> No new Thread will be created and MyRunnable run() method will be executed just like
 a normal method call.

CASE 5: r.start():
    >> We will get compile time error saying start() method is not available in MyRunnable
class.

CASE 6: r.run():
    >> No new thread will be created an MyRunnable class run() method will be executed just
 like normal method call.


 */

 class CRunnable2 implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Child Thread");
        }
    }
}

public class DMT9 {
    public static void main(String[] args) {
        CRunnable2 cr2 = new CRunnable2();
        Thread tr2 = new Thread();
        Thread tr3 = new Thread(cr2); // Here, cr2 is target runnable

        tr2.start(); // provide only output of Main method and calls and no run() method
called for empty constructor.

        tr2.run(); // provide only output of Main method and calls and no run() method
called for empty constructor.

        tr3.start();

        tr3.run(); // No thread created

        // cr2.start(); // C.E:  Exception in thread "main" java.lang.Error: Unresolved
```

```
compilation problem:  The method start() is undefined for the type CRunnable2

        cr2.run();

        // Thread t = new Thread(new CRunnable1()); (It is also possible)

        for (int i = 0; i < 5; i++) {
            System.out.println("Main Thread");
        }
    }
}

/*

Best Approach to Define a Thread:

--> Among the 2 ways of defining a Thread, implements Runnable approach is always
recommended.

--> In the 1st approach our class should always extends Thread class, there is no chance of
 extending
    any other class, hence we are missing the benefits of inheritance.

--> But in the 2nd approach while implementing Runnable interface we can extend some other
class also.
    Hence, implements Runnable mehchanism is recommmended to define a Thread.

 */
```

```
/*

Thread class constructors:

        1. Thread t = new Thread();
        2. Thread t = new Thread (Runnable r);
        3. Thread t = new Thread (String name);
        4. Thread t = new Thread (Runnable r, String name);

 */

/*

Getting and Setting name of Thread:

1. We can get current executing Thread object reference by using "Thread.currentThread()"
method.

2. Every Thread in java has some name, it may be provided explicitly by the programmer or
automatically generated by JVM.

3. Thread class defines the following method to get and set name of a Thread:

    --> public final String getName()
    --> public final void setName(String name)

 */

class CThread8 extends Thread {

}

public class DMT10 {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()); // main
        CThread8 ct8 = new CThread8();
        System.out.println(ct8.getName()); // Thread-0
        Thread.currentThread().setName("CVM");
        System.out.println(Thread.currentThread().getName()); // CVM
    }
}
```

```
/*

Thread Priority:

--> Every thread in java has some priority it may be default priority generated by JVM (OR
) explicitly provided by the programmer.
--> The valid range of Thread priorities is 1 to 10, Where 1 = Least Priority, 10 = Highest
 Priority.
--> Thread class defines the following constants to represent some standard priorities.

        (1) Thread.MIN_PRIORITY --- 1
        (2) Thread.MAX_PRIORITY --- 10
        (3) Thread. NORM_PRIORITY --- 5

--> The allowed values are 1 to 10 otherwise we will get runtime exception saying "
IllegalArgumentException".

--> Default priority only for the main Thread is 5. But for all the remaining Thread the
default priority will be inheriting from Parent to Child.
    i.e., Whatever the priority parent has, by default the same priority will be for the
child also.



 */
class CThread9 extends Thread {
    CThread9(String name) {
        super(name);
    }
    public void run() {
        System.out.println(Thread.currentThread()); // [CThread9,7,main]
    }
}

public class DMT11 {
    public static void main(String[] args) {
        Thread t = new Thread();
        CThread9 ct9 = new CThread9("CThread9");


        System.out.println("Max Priority is: "+Thread.MAX_PRIORITY); // 10
        System.out.println("Min Priority is: "+Thread.MIN_PRIORITY); // 1
        System.out.println("Norm Priority is: "+Thread.NORM_PRIORITY); // 5

        System.out.println(t.currentThread()); // [main,5,main]

        t.setPriority(5);

        ct9.setPriority(7);
        ct9.start();

        Thread.currentThread().setPriority(9);

        System.out.println(t.currentThread()); // [main,9,main]
        System.out.println(Thread.currentThread()); // [main,9,main]
    }
}
```

```java
/*

sleep() method:

--> If a Thread don't want to perform any operation for a particular amount of time then we
 should go for sleep() method.

Syntax:

        (1) public static native void sleep (long ms) throws InterruptedException
        (2)public static void sleep (long ms, int ms) throws InterruptedException

 */

class CThread10 extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(3000);
                System.out.println("Thread 1");
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class DMT12 {
    public static void main(String[] args) throws InterruptedException {
        CThread10 t = new CThread10();
        t.start();
        for (int i = 0; i < 5; i++) {
            Thread.sleep(1000);
            System.out.println("Thread 2");
        }
    }
}
```

```java
/*

join() method:

--> If a Thread wants to wait until completing some other Thread then we should go for join
() method.

--> Syntax:

            (1) public final void join() throws InterruptedException
            (2) public final void join(long ms) throws InterruptedException
            (4) public final void join (long ms, int ns) throws InterruptedException

--> Every join() method throws InterruptedException, which is checked exception hence
compulsory we should hand either by try catch OR by throws keyword.


 */

class CThread11 extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(3000);
                System.out.println("Thread 1");
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class DMT13 {
    public static void main(String[] args) throws InterruptedException {
        CThread11 t = new CThread11();
        t.start();
        t.join();
        //t.join(4000);
        //t.join(5000, 100);
        for (int i = 0; i < 5; i++) {
            Thread.sleep(1000);
            System.out.println("Thread 2");
        }
    }
}
```

```java
/*

Synchronization:

--> Synchronized is the keyword applicable for methods and blocks but not for classes and
variables.
--> If a method or block declared as the synchronized then at a time only one thread is
allowed to execute that method or block on the given object.
--> Internally Synchronization concept is implemented by using lock concept.



 */

class Receiver {
    public synchronized void receiver(String name) {
        for (int i = 0; i < 5; i++) {
            System.out.print("Ringing");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println("------" + name);
        }

    }
}

class Caller extends Thread {
    Receiver rec;
    String name;

    Caller(Receiver rec, String name) {
        this.rec = rec;
        this.name = name;
    }

    public void run() {
        rec.receiver(name);
    }
}

class DMT14 {
    public static void main(String[] args) {
        Receiver r1 = new Receiver();
        //Receiver r2 = new Receiver();
        Caller c1 = new Caller(r1, "CVM");
        Caller c2 = new Caller(r1, "HDS");
        //Caller c3 = new Caller(r2, "PAT");
        //Caller c4 = new Caller(r2, "BBK");
        c1.start();
        c2.start();
        //c3.start();
        //c4.start();
    }
}
```

```
/*

Synchronized Block:

--> In Java, a synchronized block is used to ensure that only one thread can access a
particular block of code at a time.
--> This is crucial for maintaining data consistency and avoiding race conditions,
especially when multiple threads are accessing shared resources concurrently.
--> When a synchronized block is encountered, Java ensures that only one thread can execute
 the code within the block at any given time.
--> Other threads attempting to execute the same block will be blocked until the currently
executing thread exits the synchronized block.

Syntax:
        synchronized (object) {
            // Code block that needs to be synchronized
        }

In the above syntax:

--> object: It's an object reference used for synchronization.
--> Java locks this object while executing the synchronized block.
--> Multiple threads can access synchronized blocks if they are synchronized on different
objects.
--> However, if they are synchronized on the same object, only one thread can execute the
block at a time.

Example 1:

 */


class Sender {

    public void sendermsg(String msg) {
        System.out.println("Sending message");
        try {
            Thread.sleep(2000);
        } catch (Exception e) {
            System.out.println(e);
        }
        System.out.println(msg + "------- Sent");
    }
}

class SThread extends Thread {
    Sender s;
    String msg;

    SThread(Sender s, String msg) {
        this.s = s;
        this.msg = msg;
    }

    public void run() {
        synchronized (s) {
            s.sendermsg(msg);
        }
    }
}
```

```java
class DMT15 {
    public static void main(String[] args) {
        Sender s = new Sender();
        SThread st1 = new SThread(s, "Hey");
        SThread st2 = new SThread(s, "Hello How are you?");
        st1.start();
        st2.start();
    }
}
```

```java
/*
Synchronized Block: Example 2
 */
class BankAccount {

    private int balance = 1000;

    public void withdraw(int amount) {
        System.out.println("Processing......");
        try {
            Thread.sleep(3000);
        } catch (Exception e) {
            System.out.println(e);
        }
        synchronized (this) {
            if (amount < balance) {
                balance = balance - amount;
                System.out.println("Withdrawal successful. Remaining balance: " + balance);
            } else {
                System.out.println("Insufficient Balance");
            }
        }
        System.out.println("Thank You for Visit");
    }
}

class CThread extends Thread {
    BankAccount b;
    int amount;

    public CThread(BankAccount b, int amount) {
        this.b = b;
        this.amount = amount;
    }

    public void run() {
        b.withdraw(amount);
    }
}

public class DMT16 {
    public static void main(String[] args) {
        BankAccount b = new BankAccount();
        CThread c1 = new CThread(b, 800);
        CThread c2 = new CThread(b, 800);
        c1.start();
        c2.start();
    }
}
```

```
/*
 Inter Thread Communication:

--> Inter-thread communication in Java refers to the mechanism where threads cooperate with
 each other by signaling or notifying each other about the state changes or data
availability.
--> This is typically done using the wait() and notify() (or notifyAll()) methods provided
by the Object class.
--> Inter-thread communication allows threads to synchronize their activities and
communicate effectively.

Basic Concepts:

--> wait(): The wait() method causes the current thread to wait until another thread
invokes the notify() method or the notifyAll() method for this object.
--> notify(): The notify() method wakes up a single thread that is waiting on this object's
 monitor. If more than one thread is waiting, one of them is chosen arbitrarily to be
awakened.
--> notifyAll(): The notifyAll() method wakes up all threads that are waiting on this
object's monitor.

How InterThreadCommunication Works:

--> Synchronization: To ensure proper inter-thread communication, synchronization is
essential.
--> Threads usually synchronize on a common object lock.
--> This can be any object, but it's commonly an object associated with the shared resource
 or the task being synchronized.
--> Synchronization is achieved using the synchronized keyword or synchronized blocks.
--> wait() and notify(): Threads that need to wait for a condition to be satisfied invoke
the wait() method on the common lock object.
--> This temporarily releases the lock and puts the thread into a waiting state until
another thread notifies it.
--> Threads that are in a position to notify waiting threads about a condition change
invoke the notify() or notifyAll() method on the same lock object.
--> This wakes up one or all of the waiting threads, allowing them to continue their
execution.

Race Conditions and Deadlocks:

--> Care should be taken to avoid race conditions and deadlocks.
--> Race conditions occur when the correctness of a program depends on the timing or
interleaving of threads.
--> Deadlocks occur when two or more threads are blocked indefinitely, each waiting for the
 other to release a resource.

 */

class ThreadA extends Thread {
    int total = 0;

    public void run() {
        synchronized (this) {
            for (int i = 0; i <= 100; i++) {
                total += i;
            }
            this.notify(); // Notify the waiting thread when calculation is done
        }
    }
}
```

```java
class DMT17 {
    public static void main(String[] args) throws InterruptedException {
        ThreadA t = new ThreadA();
        t.start();
        synchronized (t) {
            t.wait(); // Main thread waits for ThreadA to finish calculation
        }
        System.out.println("Total is: " + t.total);
    }
}
```

```java
// Producer - Consumer Problem

/*

If we use produce() and consume() method without synchronized then we will get error
    -->  Exception in thread "Thread-0" java.lang.IllegalMonitorStateException: current
thread is not owner
*/

class ProCon {
    boolean isAvailable = false;
    int data;

    synchronized void produce(int n) {
        if (isAvailable) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        data = n;
        System.out.println("Produce = " + data);
        isAvailable = true;
        notify();
    }

    synchronized void consume() {
        if (!isAvailable) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
        System.out.println("Consumed = " + data);
        isAvailable = false;
        notify();
    }
}

// Prodcer Thread

class Producer1 extends Thread {
    ProCon pc;

    Producer1(ProCon pc) {
        this.pc = pc;
        start();
    }

    public void run() {

        for (int i = 1; i <= 5; i++) {
            pc.produce(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
```

```
        }
    }
}

// Consumer Thread

class Consumer1 extends Thread {
    ProCon pc;

    Consumer1(ProCon pc) {
        this.pc = pc;
        start();
    }

    public void run() {

        for (int i = 1; i <= 5; i++) {
            pc.consume();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class DMT18 {
    public static void main(String[] args) {
        ProCon pc = new ProCon();
        Producer1 p = new Producer1(pc);
        Consumer1 c = new Consumer1(pc);
    }
}
```

```java
/* Producer - Consumer Problem

--> The Producer-Consumer problem is a classic synchronization problem in computer science where there are two types of threads: producers and consumers, which share a common, fixed-size buffer or queue.
--> Producers produce data items and put them into the buffer, while consumers take data items from the buffer and consume them.
--> The problem arises in ensuring that the producers do not produce data if the buffer is full and that consumers do not consume data if the buffer is empty.
--> This synchronization is crucial to prevent issues such as race conditions, deadlocks, and resource wastage.

 */

class ProducerConsumerProblem {
    static int capacity = 3;
    static int[] buffer = new int[capacity];
    static int count = 0;
    static int in = 0;
    static int out = 0;

    public static void main(String[] args) {
        ProducerConsumerProblem pc = new ProducerConsumerProblem();
        Producer p = new Producer(pc);
        Consumer c = new Consumer(pc);
        p.start();
        c.start();
    }

    public synchronized void produce() throws InterruptedException {
        int value = 1;
        while (true) {
            while (count == capacity) {
                // Buffer is full, wait for the consumer to consume items
                wait();
            }
            buffer[in] = value;
            System.out.println("Produced: " + value);
            value++;
            in = (in + 1) % capacity;
            count++;
            notify();
            Thread.sleep(1000);
        }
    }

    public synchronized void consume() throws InterruptedException {
        while (true) {
            while (count == 0) {
                // Buffer is empty, wait for the producer to produce items
                wait();
            }
            int value = buffer[out];
            System.out.println("Consumed: " + value);
            out = (out + 1) % capacity;
            count--;
            notify();
            Thread.sleep(1000);
        }
    }
```

```
}

class Producer extends Thread {
    ProducerConsumerProblem pc;

    Producer(ProducerConsumerProblem pc) {
        this.pc = pc;
    }

    public void run() {
        try {
            pc.produce();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

class Consumer extends Thread {
    ProducerConsumerProblem pc;

    Consumer(ProducerConsumerProblem pc) {
        this.pc = pc;
    }

    public void run() {
        try {
            pc.consume();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

```java
// isAlive() method returns boolean - if thread is alive or not it returns - true or false.

public class DMT20 {
    public static void main(String[] args) {
        Thread t = new Thread();
        System.out.println(t.isAlive());
        t.start();
        System.out.println(t.isAlive());
        System.out.println(Thread.currentThread().isAlive());

    }
}
```

```java
/*

Producer Consumer Problem - Another Approach

 */

class Buffer {
    private int[] buffer;
    private int size;
    private int count;

    public Buffer(int size) {
        this.size = size;
        buffer = new int[size];
        count = 0;
    }

    public synchronized void produce(int item) throws InterruptedException {
        while (count == size) {
            wait(); // Wait if buffer is full
        }
        buffer[count] = item;
        count++;
        System.out.println("Produced: " + item);
        notify(); // Notify consumers that a new item is available
    }

    public synchronized int consume() throws InterruptedException {
        while (count == 0) {
            wait(); // Wait if buffer is empty
        }
        int item = buffer[count - 1];
        count--;
        System.out.println("Consumed: " + item);
        notify(); // Notify producers that space is available
        return item;
    }
}

class Producer extends Thread {
    private Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                buffer.produce(i);
                Thread.sleep(1000); // Simulate production time
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
class Consumer extends Thread {
    private Buffer buffer;
```

```java
    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                buffer.consume();
                Thread.sleep(1500); // Simulate consumption time
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
class DMT21 {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(5); // Buffer size 5
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();
        consumer.start();
    }
}
```