CHAPTER NO:05
Multithreading {10 MARKS}

5.1 The Java Thread Model,The Thread Life Cycle, Thread class methods

5.2 The Thread Class and the Runnable Interface

5.3 The Main Thread, creating a Thread, Extending Thread, Implementing Runnable

5.4 Creating Multiple Threads, Using isAlive( ) and join( )

5.5 Thread Priorities, Synchronization, Using Synchronized Methods, The synchronized Statement

5.6 Interthread Communication

The Java Thread Model:

- Java supports **multithreading**, allowing concurrent execution of two or more parts of a program.
- Threads are managed by the JVM and mapped to native OS threads.
- Java threads are instances of the **Thread** class or implemented via the **Runnable** interface.

LIFE CYCLE OF THREAD IN JAVA:

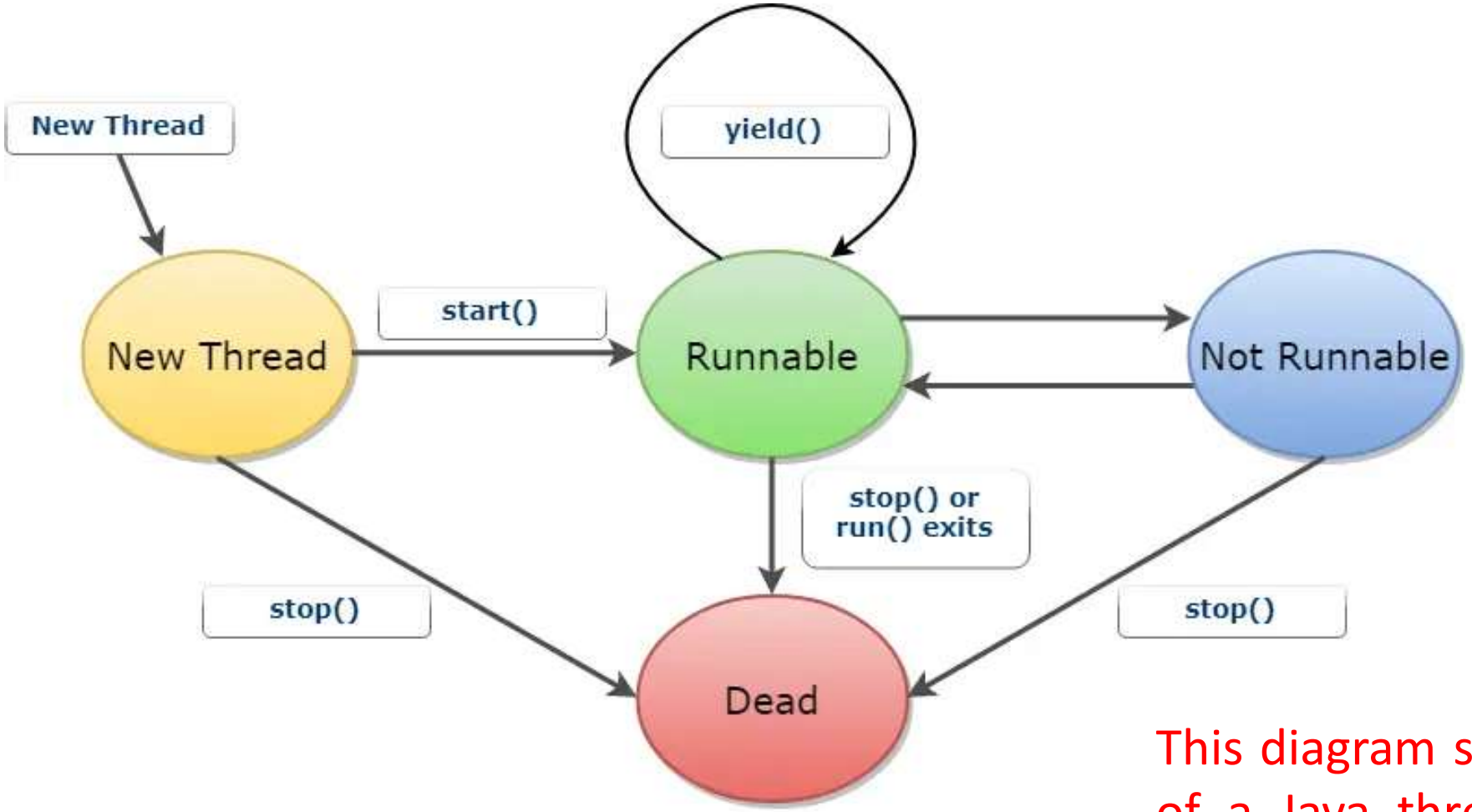


Fig: Life Cycle of a Thread in Java

This diagram shows the various states of a Java thread and how a thread transitions between these states during its life cycle.

🟡 1. New Thread

- A thread is in this state after an object of the `Thread` class (or a class that implements `Runnable`) is created but `start()` **has not been called** yet.
- **Transition to Runnable:**
  - When `start()` is called, the thread enters the **Runnable** state.
- **Transition to Dead:**
  - If `stop()` is called before `start()`, the thread goes to the **Dead** state directly (not common or recommended).

🟢 2. Runnable

- The thread is **ready to run** but not necessarily running. It's waiting for the CPU scheduler to allocate time.
- The thread may enter the running state depending on the thread scheduler.
- **Transition to Itself:**
  - If `yield()` is called, the thread gives up its current slot and goes back to **Runnable** (effectively letting other threads of equal priority run).
- **Transition to Not Runnable:**
  - Occurs when the thread is **blocked**, **sleeping**, or **waiting** (e.g., after calling `sleep()`, `wait()`, or trying to acquire a locked resource).
- **Transition to Dead:**
  - Happens when the `run()` method completes or `stop()` is explicitly called.

🔵 3. Not Runnable (Blocked / Waiting / Sleeping)

- The thread is **alive** but not eligible to run until a specific condition is met.
  - Examples: `sleep()`, `wait()`, `join()`, waiting for I/O, etc.
- **Transition to Runnable:**
  - When the condition is satisfied (e.g., timer ends, lock is acquired), the thread moves back to the **Runnable** state.

🔴 4. Dead

- A thread enters this state when it **finishes execution** (normal completion or an exception) or is **explicitly stopped**.
- Once a thread is dead, it **cannot be restarted**.

1. New Thread (New)

- This is the initial state when a thread object is created but **not yet started**.
- At this point, the thread is just an instance of the Thread class or Runnable, but it hasn't begun execution.

2. start() → Runnable

- When the start() method is called on the thread, it moves from the **New** state to the **Runnable** state.
- The thread is now ready to run and waiting for the CPU scheduler to allocate CPU time.

3. Runnable

- In this state, the thread is ready to run or currently running.
- It can transition back to **Runnable** from itself using the **yield()** method.
- **yield()** tells the thread scheduler that the current thread is willing to yield its current use of CPU, allowing other threads to run.

4. Runnable → Not Runnable

- A thread can move from **Runnable** to **Not Runnable** (often called waiting or blocked state) when it is waiting for some condition or resource.
- **For example,** when a thread is waiting to acquire a lock or waiting for I/O.

5. Not Runnable → Runnable

- When the condition that caused the thread to wait is satisfied (e.g., lock is released), the thread moves back to **Runnable.**

6. stop() or run() exits → Dead

- When the thread finishes its execution normally by completing the run() method, it moves to the **Dead** state.
- Alternatively, if the stop() method is called (deprecated and unsafe), the thread also moves to **Dead**.
- Threads in the **Dead** state cannot be restarted.

From New Thread → Dead

If the stop() method is called before the thread is started, it directly moves to the **Dead** state.

From Not Runnable → Dead

If stop() is called while the thread is waiting or blocked (Not Runnable), it transitions to **Dead.**

Important Thread Class Methods:

- `start()` : Starts the thread; calls the `run()` method internally.
- `run()` : Contains the code executed by the thread.
- `sleep(long millis)` : Pauses thread execution for given milliseconds.
- `join()` : Waits for the thread to finish execution.
- `interrupt()` : Interrupts a sleeping or waiting thread.
- `isAlive()` : Checks if the thread is still running.
- `setPriority(int)` : Sets thread priority (1 to 10).
- `yield()` : Suggests to scheduler to give other threads a chance to run.

5.2 The Thread Class and the Runnable Interface:

The Thread Class:

- Thread is a class that implements Runnable.
- You can create a thread by extending Thread and overriding run().
- **Example:**

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

The Runnable Interface:

- Runnable is a functional interface with a single method run().
- Using Runnable allows the class to extend other classes (Java doesn't support multiple inheritance).
- You implement Runnable and pass an instance to a Thread object.
- **Example:**

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

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

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

5.3 The Main Thread, Creating a Thread, Extending Thread, Implementing Runnable:

- **The Main Thread:**
- When a Java program starts, the JVM creates a main thread to execute the main() method.
- This main thread is the first thread running in any Java program.
- You can create additional threads from this main thread to perform concurrent tasks.

Creating a Thread

There are two primary ways to create a thread in Java:

1. **Extending the `Thread` class**
2. **Implementing the `Runnable` interface**

1. Extending the `Thread` Class

- Create a new class that extends `Thread`.
- Override the `run()` method with the code you want the thread to execute.
- Create an instance of your class and call `start()` to begin the thread.

```java
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("MyThread running: " + i);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();  // Starts the new thread and calls run()
        System.out.println("Main thread running");
    }
}
```

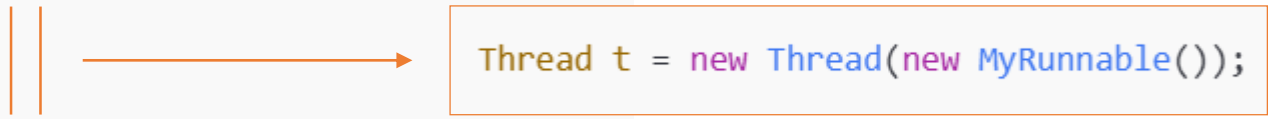
```
Main thread running
MyThread running: 0
MyThread running: 1
MyThread running: 2
MyThread running: 3
MyThread running: 4
```

2. Implementing the `Runnable` Interface

- Create a class that implements `Runnable`.
- Implement the `run()` method.
- Create a `Thread` object by passing an instance of your class to the `Thread` constructor.
- Call `start()` on the `Thread` object.

```java
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("MyRunnable running: " + i);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();  // Starts the new thread and calls run()
        System.out.println("Main thread running");
    }
}
```



Extending Thread	Implementing Runnable
You cannot extend any other class (Java supports single inheritance).	You can extend another class as well (more flexible).
Simpler when you don't need to extend any other class.	Preferred approach in most cases.
Less separation between task and thread	Clear separation between task and thread.

- The **main thread** is the starting point of any Java program.
- You create new threads by either **extending** `Thread` or **implementing** `Runnable`.
- The new thread runs the code inside the `run()` method.
- To start the thread, call `start()`, **not** `run()` directly.

5.4 Creating Multiple Threads, Using isAlive() and join()

Creating Multiple Threads:

- You can create and run multiple threads concurrently in Java by creating multiple instances of the Thread class (either by extending Thread or implementing Runnable).

```java
class MyRunnable implements Runnable {
    private String name;

    public MyRunnable(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(name + " is running: " + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println(name + " interrupted");
            }
        }
    }
}
```

```java
public class MultipleThreadsExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable("Thread 1"));
        Thread t2 = new Thread(new MyRunnable("Thread 2"));

        t1.start();  // Start first thread
        t2.start();  // Start second thread

        System.out.println("Main thread finished launching threads");
    }
}
```

Using `isAlive()`

- The `isAlive()` method checks whether a thread is still running.
- It returns `true` if the thread has been started and **not yet terminated**, otherwise `false`.

```
if (t1.isAlive()) {
    System.out.println("Thread 1 is still running");
} else {
    System.out.println("Thread 1 has finished");
}
```

This is useful to check the state of a thread before performing certain actions.

Using `join()`

- The `join()` method **waits for a thread to complete** before allowing the next line of code to execute.
- When you call `t.join()`, the current thread (usually main thread) **pauses** and waits until thread `t` finishes.
- This is useful when you want to wait for threads to finish before proceeding.

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

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();

        t.start(); // Start the thread

        System.out.println("Is thread alive? " + t.isAlive()); // true (likely)

        try {
            t.join(); // Wait for thread to finish
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Is thread alive after join? " + t.isAlive()); // false
        System.out.println("Main thread finished.");
    }
}
```

OUTPUT:

```
Thread is running...
Is thread alive? true
Is thread alive after join? false
Main thread finished.
```

JOIN EXAMPLE:

```java
public class JoinExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable("Thread 1"));
        Thread t2 = new Thread(new MyRunnable("Thread 2"));

        t1.start();
        t2.start();

        try {
            t1.join();  // Waits for Thread 1 to finish
            t2.join();  // Waits for Thread 2 to finish
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Both threads have finished execution");
    }
}
```

OUTPUT:

```
Thread 1 is running...
Thread 2 is running...
Both threads have finished execution
```

1. **Thread Start**: The two threads are started (`t1.start()` and `t2.start()`).
2. `join()` **Method**: The main thread calls `t1.join()` and `t2.join()`, which causes the main thread to wait until both `Thread 1` and `Thread 2` finish their execution.
3. **After Threads Finish**: Once both threads complete, the main thread prints "Both threads have finishec execution."

`start()`	Begins thread execution by calling `run()` asynchronously
`isAlive()`	Checks if thread is still running
`join()`	Waits for thread to finish before continuing execution

✅ How to Create a Thread

There are **2 main ways** to create a thread:

1. Using `extends Thread`:

java

Copy code

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

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread(); // Create a thread object
        t.start(); // Start the thread
    }
}
```

- 🔹 `run()` is what the thread will do.
- 🔹 `start()` tells the thread to begin (it runs `run()` in a new thread).

2. Using `implements Runnable`:

java

Copy code

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

public class Main {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r); // Wrap in a Thread
        t.start(); // Start the thread
    }
}
```

```java
class AdderThread extends Thread {
    int a, b;

    AdderThread(int x, int y) {
        a = x;
        b = y;
    }

    public void run() {
        int sum = a + b;
        System.out.println("Sum is: " + sum);
    }
}

public class Main {
    public static void main(String[] args) {
        AdderThread t1 = new AdderThread(10, 20);
        t1.start(); // Thread chalu hota hai
    }
}
```

```
Sum is: 30
```

```java
// Thread for even numbers
class EvenThread extends Thread {
    public void run() {
        for (int i = 0; i <= 10; i++) {
            if (i % 2 == 0) {
                System.out.println("Even: " + i);
            }
        }
    }
}

// Thread for odd numbers
class OddThread extends Thread {
    public void run() {
        for (int i = 0; i <= 10; i++) {
            if (i % 2 != 0) {
                System.out.println("Odd: " + i);
            }
        }
    }
}
```

```java
// Main class
public class Main {
    public static void main(String[] args) {
        EvenThread even = new EvenThread();
        OddThread odd = new OddThread();

        even.start(); // Start even thread
        odd.start();  // Start odd thread
    }
}
```

Sample Output:

makefile

```
Even: 0
Odd: 1
Even: 2
Odd: 3
Even: 4
Odd: 5
Even: 6
Odd: 7
Even: 8
Odd: 9
Even: 10
```

```java
// Thread to print even numbers
class EvenThread extends Thread {
    public void run() {
        System.out.println("Even numbers:");
        for (int i = 0; i <= 10; i += 2) {
            System.out.println(i + " from " + Thread.currentThread().getName());
        }
    }
}

// Thread to print odd numbers
class OddThread extends Thread {
    public void run() {
        System.out.println("Odd numbers:");
        for (int i = 1; i <= 10; i += 2) {
            System.out.println(i + " from " + Thread.currentThread().getName());
        }
    }
}
```

```java
// Main class
public class Main {
    public static void main(String[] args) {
        EvenThread even = new EvenThread();
        OddThread odd = new OddThread();

        even.start(); // Start even number thread
        odd.start();  // Start odd number thread
    }
}
```

OUTPUT:

```
Even numbers:
0 from Thread-0
2 from Thread-0
4 from Thread-0
6 from Thread-0
8 from Thread-0
10 from Thread-0
Odd numbers:
1 from Thread-1
3 from Thread-1
5 from Thread-1
7 from Thread-1
9 from Thread-1
```

5.5 Thread Priorities, Synchronization, Using Synchronized Methods, The synchronized Statement

1. Thread Priorities

- Every thread in Java has a priority.
- Priorities range from `MIN_PRIORITY (1)` to `MAX_PRIORITY (10)`.
- Default priority is `NORM_PRIORITY (5)`.
- Higher priority threads get more CPU time (but this depends on the JVM and OS scheduler).
- Set priority using:

```
thread.setPriority(Thread.MAX_PRIORITY);
```

- Get priority using:

```
thread.getPriority();
```

2. Synchronization

- Synchronization controls access to shared resources (like variables or methods) to prevent **race conditions** and inconsistent data.
- When multiple threads access the same data, synchronization ensures only one thread accesses it at a time.

3. Using Synchronized Methods

- Mark a method with `synchronized` keyword to allow only one thread to execute it on the same object at a time.
- Example:

```
public synchronized void increment() {
    count++;
}
```

- This locks the object for that method until it finishes.

4. The synchronized Statement (Block)

- Instead of synchronizing the entire method, you can synchronize a block of code.
- Useful when only part of the method needs synchronization.
- Syntax:

```
public void increment() {
    synchronized(this) {
        count++;
    }
}
```

- `this` means the current object. You can also synchronize on other objects.

Concept	What it Does	Usage Example
Thread Priority	Influences thread scheduling	`thread.setPriority(8);`
Synchronization	Prevents concurrent access conflicts	N/A
Synchronized Method	Locks entire method for one thread at a time	`public synchronized void m() {}`
Synchronized Block	Locks only specific code block for synchronization	`synchronized(obj) { ... }`

Simple Java Program with Thread Priorities and Synchronization

```
class Counter {
    int count = 0;

    // Synchronized method example
    public synchronized void increment() {
        count++;
    }

    // Get current count
    public int getCount() {
        return count;
    }
}
```

```
class MyThread extends Thread {
    Counter counter;

    MyThread(Counter counter, String name) {
        super(name);
        this.counter = counter;
    }
```

```
    public void run() {
        for (int i = 0; i < 5; i++) {
            // Synchronized block example
            synchronized (counter) {
                counter.increment();
                System.out.println(getName() + " incremented count to " + counter.getCount());
            }

            try {
                Thread.sleep(100); // Pause for a bit
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class SimpleThreadExample {
    public static void main(String[] args) {
        Counter counter = new Counter();

        MyThread t1 = new MyThread(counter, "Thread 1");
        MyThread t2 = new MyThread(counter, "Thread 2");

        // Set different priorities
        t1.setPriority(Thread.MIN_PRIORITY); // Priority 1
        t2.setPriority(Thread.MAX_PRIORITY); // Priority 10

        t1.start();
        t2.start();
    }
}
```

- **Thread priorities**: `t1` has low priority (1), `t2` has high priority (10).
- **Synchronized method**: `increment()` is synchronized, so only one thread can increment `count` at a time.
- **Synchronized block**: Inside the `run` method, we also use a synchronized block on the `counter` object for demonstration.

OUTPUT:

```
Thread 1 incremented count to 1
Thread 2 incremented count to 2
Thread 2 incremented count to 3
Thread 1 incremented count to 4
Thread 2 incremented count to 5
Thread 1 incremented count to 6
Thread 2 incremented count to 7
Thread 1 incremented count to 8
Thread 2 incremented count to 9
Thread 1 incremented count to 10
```

5.6 Inter thread Communication:

In Java, **inter-thread communication** allows threads to communicate with each other. This is typically done using the `wait()`, `notify()`, and `notifyAll()` methods in the `Object` class.

- `wait()`: Causes the current thread to wait until another thread sends a signal (using `notify()` or `notifyAll()`).
- `notify()`: Wakes up one thread that is waiting on the object.
- `notifyAll()`: Wakes up all threads that are waiting on the object.

These methods can only be used inside a **synchronized block** or **synchronized method**, because they involve shared resources.

```java
class Data {
    int value = 0;

    // Producer thread will call this method
    public synchronized void produce() throws InterruptedException {
        if (value == 1) {
            wait();  // Wait until the consumer consumes the data
        }
        value = 1;  // Produce data
        System.out.println("Produced: " + value);
        notify();  // Notify the consumer to consume the data
    }

    // Consumer thread will call this method
    public synchronized void consume() throws InterruptedException {
        if (value == 0) {
            wait();  // Wait until the producer produces data
        }
        System.out.println("Consumed: " + value);
        value = 0;  // Consume the data
        notify();  // Notify the producer to produce more data
    }
}
```

```
class Producer extends Thread {
    Data data;

    Producer(Data data) {
        this.data = data;
    }

    public void run() {
        try {
            while (true) {
                data.produce();
                Thread.sleep(1000);  // Simulate time taken to produce data
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class Consumer extends Thread {
    Data data;

    Consumer(Data data) {
        this.data = data;
    }

    public void run() {
        try {
            while (true) {
                data.consume();
                Thread.sleep(1500);  // Simulate time taken to consume data
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```java
public class InterThreadCommunicationExample {
    public static void main(String[] args) {
        Data data = new Data();

        Producer producer = new Producer(data);
        Consumer consumer = new Consumer(data);

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

EXPECTED OUTPUT:

```
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
...
```

1. **Shared Resource**: The `Data` class represents a shared resource, with a `value` variable that is produced and consumed.
2. **Producer Thread**: The `Producer` thread calls the `produce()` method, which sets `value` to 1 and notifies the consumer.
3. **Consumer Thread**: The `Consumer` thread calls the `consume()` method, which waits for the producer to produce the data. Once data is available, it consumes it by setting `value` to 0 and notifies the producer.
4. **Synchronization**: Both `produce()` and `consume()` methods are synchronized to ensure only one thread can modify the shared `value` at a time.

- The `Producer` produces data, and then the `Consumer` consumes it. The two threads alternate based on the synchronization and communication (`wait()` / `notify()`) mechanism.

This demonstrates **inter-thread communication** where the producer waits for the consumer, and the consumer waits for the producer.