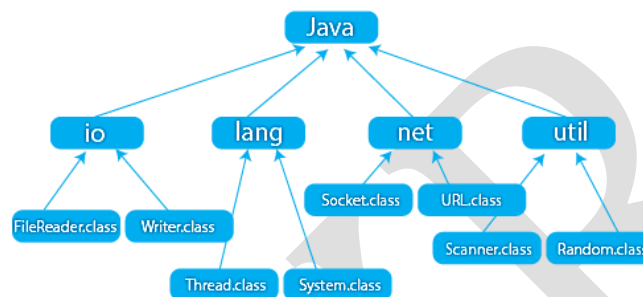


Packages: Packages, Packages and Member Access, Importing Packages.

Multithreaded Programming: The Java Thread Model, The Main Thread, Creating a Thread, Creating Multiple Threads, Using `isAlive()` and `join()`, Thread Priorities, Synchronization, Interthread Communication, Suspending, Resuming, and Stopping Threads, Obtaining a Thread's State.

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



Examples of Built-in Packages:

- java.util: Contains utility classes like ArrayList, HashMap, etc.
- java.io: Includes classes for input and output operations.
- java.lang: Automatically imported; contains essential classes like String, Math, etc.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Creating a Package

To create a package, follow the steps given below:

- 1) Define a package in your class file
- Use the package keyword at the top of your Java file.

```
package mypack;  
  
public class MyClass {  
    public void displayMessage() {  
        System.out.println("Message from MyClass in mypack!");  
    }  
}
```

- 2) Save the file
- Save the above code in a directory structure matching the package name. File path: mypackage/MyClass.java

Package Naming Convention

We follow the naming convention rules to name a package. Java has some predefined packages and allows us to create our own package. So, it is possible that a programmer can create a class with the same name as a package that already contains that type in a predefined package.

- Package names should be written in **lowercase** to avoid conflicts with class names and to improve readability.
- Use meaningful names to describe the purpose of the package. This enhances code readability and maintainability.
- Do not use Java reserved keywords (e.g., class, public, static) in package names.
- Use a hierarchical structure for subpackages to logically group related classes.
- Avoid Abbreviations and Special Characters

Importing a package

- 1) Write a new Java program in another file
 - Use the import statement to include the package.
 - Example:

```
import mypack.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.displayMessage();
    }
}
```

- 2) Save this file in the same directory as mypack or elsewhere, depending on your setup.
- 3) **Compile the program**
Compile the program with the javac command, ensuring the mypack folder is in the classpath: **javac Main.java**
- 4) **Run the program**
Run the program using the java command: **java Main**

Packages and Member Access in Java

Java provides **access modifiers** to control the visibility of members (fields, methods, and constructors) across different classes and packages. Access levels are especially important when working with packages, as they determine how members are shared between classes in the same or different packages.

Access Levels and Modifiers				
Modifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Public

- Members declared as public are accessible **from any class**, regardless of whether the class is in the same or a different package.
- Use this modifier for members you want to make globally accessible.

```
package package1;

public class ClassA {
    public void display() {
        System.out.println("Public method in ClassA");
    }
}

// In another package
package package2;

import package1.ClassA;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // Accessible
    }
}
```

Protected

- protected members are accessible:
 - Within the same package.
 - In subclasses in other packages through inheritance.

```
package package1;

public class ClassA {
    protected void display() {
        System.out.println("Protected method in ClassA");
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // Accessible
    }
}

// In a different package (through inheritance)
package package2;

import package1.ClassA;

public class ClassC extends ClassA {
    public static void main(String[] args) {
        ClassC obj = new ClassC();
        obj.display(); // Accessible through inheritance
    }
}
```

Default (No Modifier)

- Members without an explicit modifier are accessible only **within the same package**.
- This is also called **package-private** access.

```

class ClassA {
    void display() {
        System.out.println("Default method in ClassA");
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // Accessible
    }
}

// In a different package
package package2;

import package1.ClassA;

public class ClassC {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // NOT accessible
    }
}

```

Private

- Members declared as private are accessible **only within the class** where they are defined.
- Not accessible from other classes, even in the same package.

```

package package1;

public class ClassA {
    private void display() {
        System.out.println("Private method in ClassA");
    }

    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // Accessible within the same class
    }
}

// In the same package
package package1;

public class ClassB {
    public static void main(String[] args) {
        ClassA obj = new ClassA();
        obj.display(); // NOT accessible
    }
}

```

The Java Thread Model

The Java Thread Model provides a robust framework for building multithreaded applications that allow multiple tasks to run concurrently. It is based on **shared memory** and **preemptive multitasking**, where the operating system or JVM thread scheduler manages thread execution.

Key Features of the Java Thread Model

1. **Multithreading:** Allows concurrent execution of multiple threads within the same program to maximize CPU utilization and enhance application performance.
2. **Lightweight Threads:** Threads in Java are lightweight compared to processes as they share the same memory space but have independent execution paths.
3. **Thread Priorities:** Each thread has a priority that helps the scheduler determine the execution order.
4. **Synchronization:** Ensures that multiple threads can work together without interfering with shared resources, avoiding race conditions and deadlocks.
5. **Portability:** The thread model is implemented by the JVM, making it platform-independent.

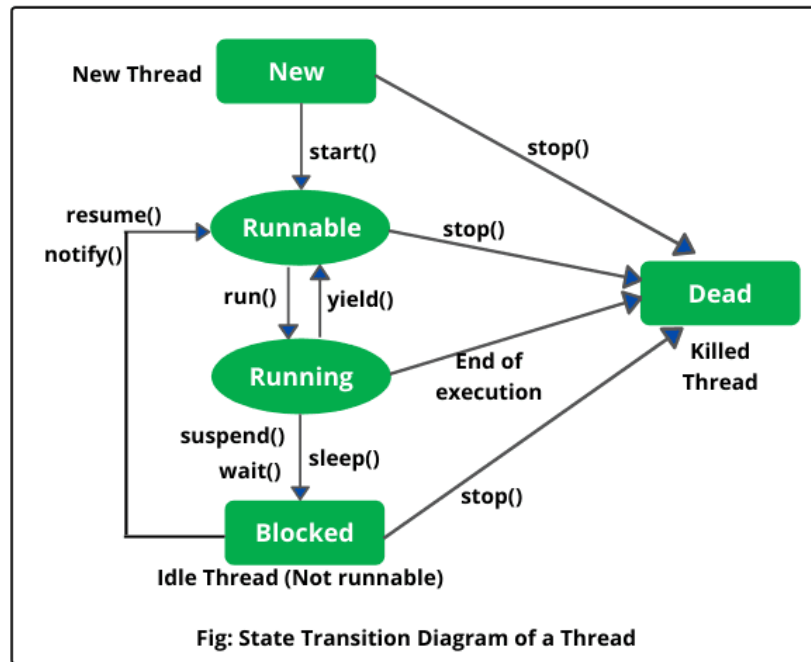
Components of the Java Thread Model

1. **Thread:** A thread represents an independent path of execution in a program.
2. **Thread Class and Runnable Interface:**
 - Java provides the Thread class and Runnable interface to create and manage threads.
3. **Thread States:**
 - Threads move through different states: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED.
4. **Thread Scheduler:**
 - The JVM's thread scheduler determines which thread gets CPU time. The scheduling algorithm may vary between JVM implementations.
5. **Thread Synchronization:**
 - Java provides synchronized blocks and methods to coordinate thread access to shared resources.

Thread Lifecycle in Java

The lifecycle of a thread in Java consists of several states:

1. **New:** A thread is created but not yet started.
2. **Runnable:** The thread is ready to run but waiting for the CPU.
3. **Running:** The thread is executing its task.
4. **Blocked/Waiting:** The thread is paused, waiting for a resource or signal.
5. **Terminated:** The thread has completed its execution or was stopped.



Advantages of the Java Thread Model

1. **Concurrency:** Multiple tasks can execute concurrently, improving application responsiveness.
2. **Efficient Resource Utilization:** Threads share the same memory space, reducing memory overhead compared to processes.
3. **Scalability:** Multithreading allows applications to scale efficiently with multi-core processors.
4. **Simplified Design:** Java's thread model simplifies the development of concurrent applications through built-in classes and synchronization features.

Creating a Thread

Java provides two primary ways to create and manage threads:

1. **Extending the Thread Class**
2. **Implementing the Runnable Interface**

Extending the Thread Class:

Creating a Thread by Extending the Thread Class

- The Thread class provides built-in functionality to create and manage threads.
- To create a thread:
 - Extend the Thread class.
 - Override its run() method.
 - Call the start() method to begin execution.

Example:

```

class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().getName());
    }
}

public class MyThread1 {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // Create a thread
        t1.start(); // Start the thread
    }
}

```

Output:

```

C:\TestJava>javac MyThread1.java

C:\TestJava>java MyThread1
Thread is running: Thread-0

```

Creating a Thread by Implementing the Runnable Interface

- Implement the Runnable interface when your class needs to perform multiple tasks and cannot extend Thread due to inheritance constraints.
- Define the run() method in your class and pass an instance of it to a Thread object.

Example:

```

class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().getName());
    }
}

public class MyThread1 {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable); // Create a thread
        t1.start(); // Start the thread
    }
}

```

Output:

```

C:\TestJava>javac MyThread1.java

C:\TestJava>java MyThread1
Thread is running: Thread-0

```

Differences Between Thread and Runnable

Feature	Extending Thread	Implementing Runnable
Inheritance	Cannot extend another class.	Can extend another class.
Code Reusability	Less reusable.	More reusable.
Memory Efficiency	Slightly less efficient.	More memory-efficient.

Creating Multiple Threads

In Java, multiple threads can run concurrently, performing different tasks simultaneously. Each thread executes independently and has its own execution path.

Steps to Create Multiple Threads

1. Define a thread class by extending Thread or implementing Runnable.
2. Create multiple instances of the thread class.
3. Start each thread using the start() method.

Example: Creating Multiple Threads Using Thread Class

```
class MyThread extends Thread {
    private String name;

    MyThread(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " is running: " + i);
            try {
                Thread.sleep(500); // Pause for 500ms
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class MultiThread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

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

Output: (Order may vary due to concurrent execution)

```
C:\TestJava>javac MultiThread.java
C:\TestJava>java MultiThread
Thread 1 is running: 1
Thread 2 is running: 1
Thread 1 is running: 2
Thread 2 is running: 2
Thread 1 is running: 3
Thread 2 is running: 3
```

Example: Creating Multiple Threads Using Runnable Interface

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

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

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " is running: " + i);
            try {
                Thread.sleep(500); // Pause for 500ms
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class MultiThread {
    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 the first thread
        t2.start(); // Start the second thread
    }
}
```

Output:

```
C:\TestJava>javac MultiThread.java

C:\TestJava>java MultiThread
Thread 2 is running: 1
Thread 1 is running: 1
Thread 1 is running: 2
Thread 2 is running: 2
Thread 2 is running: 3
Thread 1 is running: 3
```

Key Points About Multiple Threads

1. **Concurrency:** Threads execute independently and concurrently.
2. **Synchronization:** Proper synchronization must be ensured when threads share resources to avoid issues like race conditions.
3. **Thread Safety:** Using synchronized blocks or methods ensures safe access to shared resources.

Using isAlive() and join() in Java

Java provides the isAlive() and join() methods in the Thread class to manage thread execution and monitor thread states. These methods help coordinate multiple threads effectively.

1. isAlive() Method

Purpose

The isAlive() method is used to check whether a thread is still running or has finished its execution.

Key Points

- Returns **true** if the thread has started and has not yet terminated.
- Returns **false** if the thread is in the NEW state or has completed execution (TERMINATED state).

Syntax: *boolean isAlive();*

2. join() Method

Purpose

The join() method ensures that the calling thread waits for the specified thread to complete its execution before continuing.

Key Points

- Allows one thread to wait for another thread to finish.
- Helps in coordinating thread execution.
- There are three overloaded versions:
 - void join(): Waits indefinitely until the thread finishes.
 - void join(long millis): Waits for the thread to finish or until the specified time has elapsed.
 - void join(long millis, int nanos): Waits for the thread to finish or until the specified time (in milliseconds and nanoseconds) has elapsed.

Syntax: *void join() throws InterruptedException*

Key Differences Between isAlive() and join()

Aspect	isAlive()	join()
Purpose	Checks if a thread is still running.	Waits for a thread to complete.
Return Value	Returns a boolean (true or false).	Returns nothing (void).
Usage	Monitoring thread state.	Synchronizing thread execution.
Blocking	Non-blocking.	Blocking until thread finishes or timeout.
Example Use Case	Periodically check a thread's status.	Ensure thread completion before proceeding.

Example:

```
class MyThread extends Thread {
    private String threadName;

    MyThread(String threadName) {
        this.threadName = threadName;
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(threadName + " is running: Step " + i);
            try {
                Thread.sleep(1000); // Simulate work by sleeping for 1 second
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted.");
            }
        }
        System.out.println(threadName + " has finished.");
    }
}

public class AliveJoin {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

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

        // Use isAlive() to check the status of the threads
        while (t1.isAlive() || t2.isAlive()) {
            System.out.println("Checking threads...");
            System.out.println("Thread 1 alive: " + t1.isAlive());
            System.out.println("Thread 2 alive: " + t2.isAlive());
            try {
                Thread.sleep(500); // Pause to avoid flooding the output
            } catch (InterruptedException e) {
                System.out.println("Main thread interrupted.");
            }
        }
        System.out.println("Waiting for threads to complete using join()...");
        try {
            t1.join(); // Ensure Thread 1 has completed
            t2.join(); // Ensure Thread 2 has completed
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted while waiting for join.");
        }
        System.out.println("All threads have finished. Exiting main thread.");
    }
}
```

Output:

```
C:\TestJava>javac AliveJoin.java

C:\TestJava>java AliveJoin
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 1
Thread 2 is running: Step 1
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 2
Thread 2 is running: Step 2
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 3
Thread 2 is running: Step 3
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 has finished.
Thread 2 has finished.
Waiting for threads to complete using join()...
All threads have finished. Exiting main thread.
```

Thread Priorities in Java

Thread priorities in Java determine the relative importance of threads when they compete for CPU time. Every thread has a priority value, which helps the thread scheduler decide the order in which threads should be executed. However, thread priorities do not guarantee execution order; they are just hints to the thread scheduler.

Key Concepts

1. Priority Range

- Thread priorities are integers ranging from **MIN_PRIORITY** (1) to **MAX_PRIORITY** (10).
- The default priority is **NORM_PRIORITY** (5).

2. Setting and Getting Priorities

- You can set a thread's priority using the `setPriority()` method.
- Use the `getPriority()` method to retrieve a thread's priority.

3. Priority Constants

- `Thread.MIN_PRIORITY = 1`
- `Thread.NORM_PRIORITY = 5`
- `Thread.MAX_PRIORITY = 10`

4. Purpose of Priorities

- Threads with higher priorities are more likely to be selected for execution by the thread scheduler, but this depends on the JVM implementation and operating system.
- Lower-priority threads may still execute if no higher-priority threads are runnable.

Thread Priority Methods

- **Set a Thread's Priority:** *public final void setPriority(int newPriority)*
- **Get a Thread's Priority:** *public final int getPriority()*

Example:

```
class MyThread extends Thread {  
    MyThread(String name) {  
        super(name); // Set thread name  
    }  
  
    public void run() {  
        System.out.println(getName() + " with priority " + getPriority() + " is running.");  
    }  
}  
  
public class ThreadPriority {  
    public static void main(String[] args) {  
        // Create three threads with different priorities  
        MyThread t1 = new MyThread("Thread-1");  
        MyThread t2 = new MyThread("Thread-2");  
        MyThread t3 = new MyThread("Thread-3");  
  
        // Set priorities  
        t1.setPriority(Thread.MIN_PRIORITY);  
        t2.setPriority(Thread.NORM_PRIORITY);  
        t3.setPriority(Thread.MAX_PRIORITY);  
  
        // Start threads  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

Output:

```
C:\TestJava>javac ThreadPriority.java  
  
C:\TestJava>java ThreadPriority  
Thread-2 with priority 5 is running.  
Thread-1 with priority 1 is running.  
Thread-3 with priority 10 is running.
```

Important Notes

1. Platform-Dependent Behavior:

- Thread priority handling depends on the JVM and OS. On some systems, priorities might have no effect.

2. Starvation Risk:

- Threads with low priority might not get enough CPU time if high-priority threads dominate.

3. Use Priorities Carefully:

- Setting thread priorities should be done judiciously to avoid unpredictable behavior.

Synchronization in Java

Definition: Synchronization in Java is a mechanism to control access to shared resources in a multithreaded environment. It ensures that only one thread can access a critical section of code at a time, preventing data inconsistency and thread interference.

Why Synchronization is Needed

1. **Shared Resources:** In multithreaded applications, threads often share resources like variables, objects, or files.
2. **Thread Interference:** Without synchronization, multiple threads might access and modify shared data simultaneously, leading to incorrect results.
3. **Consistency:** Synchronization helps maintain consistency and prevents race conditions.

Types of Synchronization

1. Synchronized Methods

- Synchronizes the entire method.
- The thread must acquire the lock on the object (or class for static methods) before executing the method.

```
public synchronized void methodName() {  
    // critical section  
}
```

Example:

```
class Counter {  
    private int count = 0;  
  
    public synchronized void increment() {  
        count++;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}  
  
public class SynchronizedMethodExample {  
    public static void main(String[] args) throws InterruptedException {  
        Counter counter = new Counter();  
  
        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());  
    }  
}
```

Output:

```
C:\TestJava>javac SynchronizedMethodExample.java
C:\TestJava>java SynchronizedMethodExample
Final Count: 2000
```

2. Synchronized Blocks

- Synchronizes only a specific block of code, allowing finer-grained control.
- Requires a lock on a specific object.

Syntax:

```
synchronized(object)
{
    //
    critical section
}
```

Example:

```
class Counter {
    private int count = 0;

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

    public int getCount() {
        return count;
    }
}

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

        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());
    }
}
```

Output:

```
C:\TestJava>javac SynchronizedBlockExample.java  
C:\TestJava>java SynchronizedBlockExample  
Final Count: 2000
```

3. Static Synchronization

- Synchronizes static methods.
- Locks on the class object, not the instance.

Syntax:

public static synchronized void methodName() { // critical section }

Example:

```
class SharedResource {  
    public static synchronized void printMessage(String message) {  
        System.out.print("[");  
        System.out.print(message);  
        System.out.println("]");  
    }  
}  
  
public class StaticSynchronizationExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(() -> SharedResource.printMessage("Hello"));  
        Thread t2 = new Thread(() -> SharedResource.printMessage("World"));  
  
        t1.start();  
        t2.start();  
    }  
}
```

Output:

```
C:\TestJava>javac StaticSynchronizationExample.java  
C:\TestJava>java StaticSynchronizationExample  
[Hello]  
[World]
```

Inter-Thread Communication

Inter-thread communication is a mechanism to allow threads to communicate with each other efficiently in a synchronized environment. Java provides methods like `wait()`, `notify()`, and `notifyAll()` to achieve this.

Methods for Communication

- **`wait()`**: Makes the current thread wait until another thread invokes `notify()` or `notifyAll()` on the same object.
- **`notify()`**: Wakes up a single thread waiting on the object's monitor.
- **`notifyAll()`**: Wakes up all threads waiting on the object's monitor.

```
class SharedResource {
    private int data;
    private boolean available = false;

    public synchronized void produce(int value) {
        while (available) {
            try {
                wait(); // Wait until the consumer consumes the data
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        data = value;
        available = true;
        System.out.println("Produced: " + value);
        notify(); // Notify the consumer
    }

    public synchronized int consume() {
        while (!available) {
            try {
                wait(); // Wait until the producer produces data
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        available = false;
        System.out.println("Consumed: " + data);
        notify(); // Notify the producer
        return data;
    }
}

public class InterThreadCommunication {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread producer = new Thread() -> {
            for (int i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        };

        Thread consumer = new Thread() -> {
            for (int i = 1; i <= 5; i++) {
                resource.consume();
            }
        };

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

Output:

```
C:\TestJava>javac InterThreadCommunication.java
C:\TestJava>java InterThreadCommunication
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

Suspending, Resuming, and Stopping Threads

These actions allow for manual control of thread execution, but their use is deprecated in favor of modern thread management techniques (like using flags or wait/notify).

a. Suspending a Thread

- **Method:** `suspend()` (deprecated)
- Pauses a thread's execution until it is resumed.

b. Resuming a Thread

- **Method:** `resume()` (deprecated)
- Resumes a thread that has been suspended.

c. Stopping a Thread

- **Method:** `stop()` (deprecated)
- Forcefully terminates a thread's execution.

Modern Alternative: Use flags and thread-safe mechanisms.

Example:

```
class ControlledThread extends Thread {
    private volatile boolean suspended = false;

    public void suspendThread() {
        suspended = true;
    }

    public void resumeThread() {
        suspended = false;
        synchronized (this) {
            notify();
        }
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            synchronized (this) {
```

```
        while (suspended) {
            try {
                wait(); // Pause execution
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    System.out.println("Thread running: Step " + i);
    try {
        Thread.sleep(1000); // Simulate work
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

public class SuspendResumeExample {
    public static void main(String[] args) throws InterruptedException {
        ControlledThread thread = new ControlledThread();
        thread.start();

        Thread.sleep(2000);
        thread.suspendThread(); // Suspend the thread
        System.out.println("Thread suspended.");

        Thread.sleep(2000);
        thread.resumeThread(); // Resume the thread
        System.out.println("Thread resumed.");
    }
}
```

Output:

```
C:\TestJava>javac SuspendResumeExample.java

C:\TestJava>java SuspendResumeExample
Thread running: Step 1
Thread running: Step 2
Thread suspended.
Thread running: Step 3
Thread resumed.
Thread running: Step 4
Thread running: Step 5
```

Obtaining a Thread's State

Java provides methods to monitor the state of threads during their lifecycle.

Thread Lifecycle States

Department of Computer Application, BCA , ATMECE, MYSURU

1. **NEW:** The thread is created but not yet started.
2. **RUNNABLE:** The thread is ready to run and waiting for CPU time.
3. **BLOCKED:** The thread is waiting for a monitor lock.
4. **WAITING:** The thread is waiting indefinitely for another thread to perform an action.
5. **TIMED_WAITING:** The thread is waiting for a specific period.
6. **TERMINATED:** The thread has finished execution.

Methods to Get Thread State

- **getState():** Returns the current state of the thread.

Example:

```
public class ThreadStateExample {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(() -> {  
            System.out.println("Thread is running...");  
            try {  
                Thread.sleep(1000); // Simulate work  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println("Thread finished.");  
        });  
  
        System.out.println("State: " + thread.getState()); // NEW  
        thread.start();  
  
        while (thread.isAlive()) {  
            System.out.println("State: " + thread.getState()); // RUNNABLE or TIMED_WAITING  
            Thread.sleep(100); // Monitor periodically  
        }  
  
        System.out.println("State: " + thread.getState()); // TERMINATED  
    }  
}
```

Output:

```
C:\TestJava>javac ThreadStateExample.java  
  
C:\TestJava>java ThreadStateExample  
State: NEW  
State: RUNNABLE  
Thread is running...  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
Thread finished.  
State: TERMINATED
```