Thread

What is a Thread in Java?

In Java, a thread is a lightweight process that allows a program to perform multiple tasks concurrently. It is the smallest unit of execution within a program. A thread runs within the context of a process, and a single process can have multiple threads running simultaneously, which is called multithreading.

Threads allow you to:

1. Improve performance: By performing multiple tasks concurrently (especially on multi-core processors), threads can make better use of system resources.
2. Handle multiple tasks: For example, in a GUI application, one thread can handle the user interface, while another handles background tasks like network communication or file I/O.
3. Better responsiveness: Threads can make a program more responsive by performing time-consuming tasks in the background.

Key Concepts:

- Main Thread: Every Java program starts with a single main thread (the main() method).
- Concurrency: Multiple threads making progress independently.
- Parallelism: Multiple threads running at the same time on different processors or cores.

Types of threads

1) User Threads

- These are the primary threads created by the user or the application for executing tasks. They are non-daemon threads by default.
- **Lifecycle:** User threads keep the JVM running until all such threads have completed their execution.
- **Examples:**
    - Main thread (created automatically when a Java program starts).
    - Threads for performing tasks like file reading, network requests, or computations.

Example:

```
class UserThreadExample extends Thread {
   public void run() {
```

```
        System.out.println("User thread is running...");
    }

    public static void main(String[] args) {
        UserThreadExample t1 = new UserThreadExample();
        t1.start();
    }
}
```

---

2. Daemon Threads

- **Definition:** These are background threads designed to perform supporting tasks like garbage collection or monitoring. They are automatically terminated when all user threads are finished.
- **Characteristics:**
    - Do not prevent the JVM from exiting.
    - Used for tasks like logging, cleanup, or background monitoring.
    - Can be created by calling setDaemon(true) before starting the thread.

Example:

```
class DaemonThreadExample extends Thread {
    public void run() {
        System.out.println("Daemon thread running...");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }

    public static void main(String[] args) {
        DaemonThreadExample t1 = new DaemonThreadExample();
        t1.setDaemon(true); // Set thread as daemon
        t1.start();
        System.out.println("Main thread ends. Daemon thread will be terminated.");
    }
}
```

3) **Main Thread**

- The thread created by the JVM when a Java program starts. It is the entry point of the program.

```
public class MainThreadExample {
   public static void main(String[] args) {
      System.out.println("Main thread is running...");
   }
}
```

What's the difference between User thread and Daemon thread?

User and Daemon are basically two types of thread used in Java by using a 'Thread Class'.

User Thread : These are the primary threads created by the user or the application for executing tasks. They are non-daemon threads by default.

1) **JVM** waits for user threads to finish their tasks before termination.
2) These threads are normally created by the user for executing tasks concurrently.
3) They are used for critical tasks or core work of an application.
4) These threads are referred to as high-priority tasks, therefore are required for running in the foreground.

Daemon Thread : These are background threads designed to perform supporting tasks like garbage collection or monitoring. They are automatically terminated when all user threads are finished

1) JVM does not wait for daemon threads to finish their tasks before termination.
2) **These threads are normally created by JVM**.
3) They are not used for any critical tasks but to do some supporting tasks.
4) These threads are referred to as low priority threads, therefore are especially required for supporting background tasks like garbage collection, releasing memory of unused objects, etc.

Q: How many ways we have to create a thread

Ways to Create a Thread in Java

In Java, there are **two main ways** to create a thread:

1) **By Extending the Thread Class**

You can create a thread by subclassing the Thread class and overriding its run() method.

2) **By Implementing the Runnable Interface**

We can implement the Runnable interface and pass it to a Thread object.

```
package com.ram.thread;
// there are 2 ways to create threads in java
/*1. Extending the Thread Class
You can create a thread by defining a new class that extends the Thread class and overrides its
run() method.*/
class MyThread extends Thread {
  @Override
  public void run() {
     System.out.println("Thread is running");
  }
}
/*
* 2. Implementing the Runnable Interface Another way to create a thread is to
* implement the Runnable interface and pass an instance of this implementation to a Thread
object.
*/
class MyRunnable1 implements Runnable {
  @Override
  public void run() {
     System.out.println("Runnable thread is running");
  }
}
public class Threading {
  public static void main(String[] args) {
        // thread class
     MyThread thread = new MyThread();
     thread.start(); // Start the thread

     //for runnable interface
     Thread thread2 = new Thread(new MyRunnable1());//Create a new instance of the Thread
class, passing an instance of your Runnable implementation to its constructor.
```

```
        thread2.start(); // Start the thread
    }

}
```

Output

```
Thread is running
Runnable thread is running
```

Thread Life Cycle:

```
package com.ram.thread;
public class ThreadLifeCycleExample2 {
    public static void main(String[] args) {

            // For every program main thread will execute first
            System.out.println("Thread name:"+Thread.currentThread().getName() +""+" status:"+
Thread.currentThread().getState());

        // Create a new thread object (New state)
        Thread thread1 = new Thread(new MyRunnable2());
        // Print the state of the new thread (NEW)
        System.out.println("Thread1 state (NEW): " + thread1.getState());
        // Start the new thread (Transition from New to Runnable)
        thread1.start();
        // Print the state of the new thread (RUNNABLE)
        System.out.println("Thread1 state (RUNNABLE): " + thread1.getState());
        // Main thread sleeps for a while to observe the behavior of the new thread
        try {
            Thread.sleep(2000); // Wait for thread1 to finish
             /*
              * we can use thread1.join(); Main thread waits for thread1 to finish (join()
              * lets the main thread wait for the child thread to complete
              */

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Print the state of the thread after it has finished (TERMINATED)
```

```
        System.out.println("Thread1 state after 2 seconds: " + thread1.getState());
        // Main thread completes execution
        System.out.println("Main thread has finished execution");
    }
}
class MyRunnable2 implements Runnable {
    public void run() {
        try {
            // Thread starts running (Runnable state)
            System.out.println("Thread1 is running");
            // Simulating some work with sleep
            Thread.sleep(1000);
            // Simulate some more work or operations
            System.out.println("Thread1 has finished running");
        } catch (InterruptedException e) {
            System.out.println("Thread1 was interrupted");
        }
    }
}
```

Output

```
thread name:main  status:RUNNABLE

Thread1 state (NEW): NEW
Thread1 state (RUNNABLE): RUNNABLE
Thread1 is running
Thread1 has finished running
Thread1 state after 2 seconds: TERMINATED
Main thread has finished execution
```

Exp:

1. Program Starts (Main Thread)

public static void main(String[] args) {

- The main thread starts execution. This thread is responsible for executing the main method.
- The JVM creates this thread automatically and assigns it the name main.

2. Create a New Thread

Thread thread1 = new Thread(new MyRunnable());

System.out.println("Thread1 state (NEW): " + thread1.getState());

- A new thread (thread1) is created, but it has not started yet.
- The state of thread1 is NEW. At this point, the thread exists but has no CPU time.

Output:
Thread1 state (NEW): NEW

3. Start the New Thread

thread1.start();

System.out.println("Thread1 state (RUNNABLE): " + thread1.getState());

- The start() method is called, which transitions thread1 from the NEW state to the RUNNABLE state.
- In the RUNNABLE state, the thread is ready to run and is waiting for the CPU to schedule its execution.
- Immediately after calling start(), the thread1 state is printed as RUNNABLE.

Output:
Thread1 state (RUNNABLE): RUNNABLE

4. Execution of run() Method (thread1)

- The JVM starts executing the run() method in MyRunnable on thread1.

Inside the run() method:
System.out.println("Thread1 is running");

Thread.sleep(1000);

System.out.println("Thread1 has finished running");

-

    1. "Thread1 is running" is printed when the thread begins execution.
    2. Thread.sleep(1000) pauses thread1 for 1 second, transitioning it to the TIMED_WAITING state.

3. After sleeping, the thread finishes execution, and "Thread1 has finished running" is printed.

Output (from thread1):

Thread1 is running

Thread1 has finished running

5. Main Thread Sleeps

Thread.sleep(2000);

- The main thread pauses for 2 seconds, allowing thread1 to finish its execution.
- During this time:
    - thread1 executes its run() method and transitions to the TERMINATED state when completed.

6. Check Thread1 State After 2 Seconds

System.out.println("Thread1 state after 2 seconds: " + thread1.getState());

- After the main thread wakes up, it checks the state of thread1.
- By this time, thread1 has completed its run() method and transitioned to the TERMINATED state.

Output:
Thread1 state after 2 seconds: TERMINATED

7. Main Thread Finishes Execution

System.out.println("Main thread has finished execution");

- The main thread prints a final message and terminates.
- Once the main thread finishes execution, the program ends.

Output:
Main thread has finished execution

Final Execution Order

1. Main thread creates thread1 and prints its state as NEW.
2. Main thread starts thread1 and prints its state as RUNNABLE.
3. Thread1 begins execution:

    - Prints "Thread1 is running".
    - Sleep for 1 second.
    - Prints "Thread1 has finished running".
4. Main thread sleeps for 2 seconds.
5. After 2 seconds, the main thread checks thread1 state (TERMINATED).
6. Main thread prints the final message and terminates.

Multithreading in Java

Multithreading is the ability of a program to execute multiple threads simultaneously. A thread is the smallest unit of a process that can execute independently. Java provides built-in support for multithreading through the java.lang.Thread class and the java.util.concurrent package.

Key Concepts

- **Thread**: A lightweight process. Multiple threads can exist within a single process, sharing resources like memory.
- **Concurrency**: Allows multiple threads to execute simultaneously, improving the performance of applications, especially on multi-core processors.
- **Thread Lifecycle**: A thread can be in one of the following states:
    - **New**: Thread is created but not yet started.
    - **Runnable**: Thread is ready to run but waiting for CPU time.
    - **Running**: Thread is executing.
    - **Blocked/Waiting**: Thread is waiting for a resource.
    - **Terminated**: Thread has finished execution.

```
package com.ram.thread;
public class MultiThreadingEx {
	public static void main(String[] args) {
		Exthread thread1 = new Exthread();
		Exthread thread2 = new Exthread();
		thread1.start();
		thread2.start();
	}
}
```

```
class Exthread extends Thread {
	@Override
	public void run() {
		for (int i = 0; i < 6; i++) {
			System.out.println(Thread.currentThread().getName() + " "
+ i);
			try {
				Thread.sleep(1000); // sleep for 1000 milli sec
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
}
```

Output

```
Thread-0 0
Thread-1 0
Thread-1 1
Thread-0 1
Thread-0 2
Thread-1 2
Thread-0 3
Thread-1 3
Thread-1 4
Thread-0 4
Thread-0 5
Thread-1 5
```

Join()

Using the join() method in Java for synchronizing threads is a form of thread synchronization that ensures the execution order of threads. When you call join() on a thread, the calling thread (usually the main thread or another thread) will pause its execution until the thread on which join() was called has finished executing. It **pauses the calling thread until the thread on which join() was called finishes its execution**.

This is a form of thread coordination, but it is not about preventing access to shared resources (which is typically handled by locks, synchronized blocks, or ReentrantLock). Instead, it's about ensuring that one thread waits for another to finish before proceeding.

In short:

- join() ensures that the calling thread waits for the specified thread to complete, providing a mechanism for thread coordination.
- This is useful when you need the calling thread to wait for the completion of other threads before proceeding with further operations, but it does not manage shared resource synchronization like preventing race conditions.

```
package com.ram.thread;
public class SynchronizationEx2 {
	public static void main(String[] args) {
		Exthread1 thread1 = new Exthread1();
		Exthread1 thread2 = new Exthread1();
		thread1.start(); // Start Thread-0
		try {
			thread1.join(); // Wait for Thread-0 to finish
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		thread2.start(); // Start Thread-1 after Thread-0 completes
	}
}
class Exthread1 extends Thread {
	@Override
	public void run() {
		for (int i = 0; i < 6; i++) {
			System.out.println(Thread.currentThread().getName() + " " + i);
			try {
				Thread.sleep(1000); // Sleep for 1000 milliseconds
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Output

```
Thread-0 0
Thread-0 1
Thread-0 2
```

Thread-0 3
Thread-0 4
Thread-0 5
Thread-1 0
Thread-1 1
Thread-1 2
Thread-1 3
Thread-1 4
Thread-1 5

Synchronization in Java

Synchronization is a mechanism to control the access of multiple threads to shared resources. **It ensures that only one thread can access the critical section of the code at a time**, preventing data inconsistencies and race conditions.

Why Synchronization is Needed

When multiple threads try to modify a shared resource concurrently, it can lead to unpredictable behavior or incorrect results. Synchronization resolves this issue by making the resource access thread-safe.

Types of Synchronization

1. **Synchronized Method**: Locks the entire method for a single thread.
2. **Synchronized Block**: Locks only a specific block of code for a single thread, allowing better performance by reducing the scope of the lock.
3. **Static Synchronization**: Locks on the class-level object.

EX1:

Synchronized Block

Shared Lock Object:

- Introduced a private static final Object lock to act as a shared lock between threads. Declaring it static ensures all threads of the class share the same lock.

Synchronized Block:

- Added synchronized (lock) to ensure that the code inside the block is executed by only one thread at a time.

```
package com.ram.thread;
public class SynchronizationEx1 {
  public static void main(String[] args) {
     Exthread thread1 = new Exthread();
     Exthread thread2 = new Exthread();
     thread1.start();
     thread2.start();
  }
}
class Exthread extends Thread {
  private static final Object lock = new Object(); // Shared lock for synchronization
  @Override
  public void run() {
     synchronized (lock) { // Synchronize the block of code
        for (int i = 0; i < 6; i++) {
           System.out.println(Thread.currentThread().getName() + " " + i);
           try {
              Thread.sleep(1000); // sleep for 1000 milliseconds
           } catch (InterruptedException e) {
              e.printStackTrace();
           }
        }
     }
  }
}
```

Output
Thread-0 0
Thread-0 1
Thread-0 2
Thread-0 3
Thread-0 4

```
Thread-0 5
Thread-1 0
Thread-1 1
Thread-1 2
Thread-1 3
Thread-1 4
Thread-1 5
```

Synchronized Method:

The increment() method is synchronized so that only one thread can execute it at a time.

```
package com.ram.thread;
class Counting {
	private int count = 0;
	// Synchronized method
	public synchronized void increment() {
		count++;
	}
	public int getCount() {
		return count;
	}
}
// Thread class
class CounterThread extends Thread {
	private Counting counter;
	public CounterThread(Counting counter) {
		this.counter = counter;
	}
	@Override
	public void run() {
		for (int i = 0; i < 100; i++) {
			counter.increment();
		}
	}
}
public class MethodLevelSynchronization {
	public static void main(String[] args) throws InterruptedException {
		Counting counter = new Counting();
		// Create threads by extending Thread class
		CounterThread thread1 = new CounterThread(counter);
```

```
                CounterThread thread2 = new CounterThread(counter);
                thread1.start();
                thread2.start();
                thread1.join();
                thread2.join();
                System.out.println("Final count: " + counter.getCount());
        }
}
```

Output
Final count: 200

Synchronized static method

```
package com.ram.thread;
class Counting {
        private static int count = 0;
        // Synchronized method
        public synchronized static void increment() {
                count++;
        }
        public int getCount() {
                return count;
        }
}
// Thread class
class CounterThread extends Thread {
        private Counting counter;
        public CounterThread(Counting counter) {
                this.counter = counter;
        }
        @Override
        public void run() {
                for (int i = 0; i < 100; i++) {
                        counter.increment();
                }
        }
}
public class MethodLevelSynchronization {
```

```
    public static void main(String[] args) throws InterruptedException {
        Counting counter = new Counting();
        // Create threads by extending Thread class
        CounterThread thread1 = new CounterThread(counter);
        CounterThread thread2 = new CounterThread(counter);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("Final count: " + counter.getCount());
    }
}
```

Output
Final count: 2000

Inter-Thread Communication

Inter-thread communication in Java refers to the mechanisms used to enable communication and coordination between threads that are executing concurrently. It allows threads to communicate, wait for conditions, and notify each other when certain events or actions need to occur, ensuring smooth execution of multi-threaded programs.

Thread Synchronization Thread synchronization ensures that only one thread can access a particular block of code or resource at a time. This is critical in preventing race conditions when multiple threads interact with shared resources.
Wait/Notify Mechanism The wait(), notify(), and notifyAll() methods are used to facilitate communication between threads.

wait()

- **Purpose**: A thread can invoke wait() on an object to release the lock it holds on the object and enter the **waiting state**. The thread remains in this state until another thread calls notify() or notifyAll() on the same object.
- **Usage**: A thread typically waits for some condition to be met before continuing its execution.

Syntax:

```
synchronized (object) {

   object.wait();

}
```

It releases the lock on the object and enters the waiting state.

The thread will be re-awakened when another thread sends a signal via notify() or notifyAll().

notify()

- **Purpose**: This method is used to wake up one thread that is currently waiting on the object's monitor (lock).
- **Usage**: It is typically used to signal a waiting thread that a certain condition has been met, allowing it to resume execution.

Syntax:

```
synchronized (object) {

   object.notify();

}
```

  - It only wakes up one thread that is waiting on the object. If multiple threads are waiting, one is chosen arbitrarily.
  - The thread that is woken up will continue executing when it can acquire the lock.

notifyAll()

- **Purpose**: Similar to notify(), but it wakes up all the threads that are currently waiting on the object's monitor.
- **Usage**: It is useful when multiple threads are waiting for the same condition to be met.

Syntax:

```
synchronized (object) {

   object.notifyAll();

}
```

- **Details**:
    - All waiting threads are awakened and compete to acquire the lock to proceed with their execution.
    - This method is often used when the conditions for waiting are no longer valid, and all waiting threads need to be notified.

Ex

```
package com.ram.thread;

// Class to represent a shared message object between threads
class Message {
	private String message; // Holds the message to be shared
	// Synchronized method to read the message
	public synchronized String read() {
		return message; // Return the current message
	}
	// Synchronized method to write a message
	public synchronized void write(String message) {
		this.message = message; // Set the message to the provided value
	}
}
// Producer thread, which generates and writes a message to the shared object
class Producer extends Thread {
	private Message message; // The shared message object
	// Constructor to initialize the Producer with the shared message object
	public Producer(Message message) {
		this.message = message;
	}
	// The thread's run method which defines what the Producer does
	public void run() {
		try {
			// Simulate some work by sleeping for 1 second
			Thread.sleep(1000);
			// Synchronize access to the shared message object
			synchronized (message) {
				// Write the message to the shared object
				message.write("Hello from Producer!");
				// Notify the Consumer thread that the message is ready
				message.notify();
			}
```

```
            } catch (InterruptedException e) {
                // Handle any interruption during sleep
                e.printStackTrace();
            }
        }
}
// Consumer thread, which waits for and reads the message from the shared object
class Consumer extends Thread {
        private Message message; // The shared message object
        // Constructor to initialize the Consumer with the shared message object
        public Consumer(Message message) {
            this.message = message;
        }
        // The thread's run method which defines what the Consumer does
        public void run() {
            // Synchronize access to the shared message object
            synchronized (message) {
                try {
                    // Wait until the Producer writes a message
                    // If the message is null, wait until the Producer provides a message
                    while (message.read() == null) {
                        message.wait(); // Wait for a notification from the Producer
                    }
                    // Once the message is available, print the received message
                    System.out.println("Consumer received: " + message.read());
                } catch (InterruptedException e) {
                    // Handle any interruption while waiting or reading the message
                    e.printStackTrace();
                }
            }
        }
}
// Main class to run the program
public class ThreadInterCommunication {
        public static void main(String[] args) {
            // Create a shared Message object that will be used by both threads
            Message message = new Message();
            // Create a Consumer thread, passing the shared Message object
            Consumer consumer = new Consumer(message);
            // Create a Producer thread, passing the shared Message object
```

```
        Producer producer = new Producer(message);
        // Start the Consumer thread, it will begin executing the run() method
        consumer.start();
        // Start the Producer thread, it will begin executing the run() method
        producer.start();
    }
}
```

//**notifyAll()**: In the Producer thread, I replaced notify() with notifyAll() to notify all waiting threads. This will be useful if you have multiple consumer threads.

Output

Consumer received: Hello from Producer!

DeadLock

A **deadlock** is a situation in a multi-threaded environment where two or more threads are blocked forever because they are waiting for each other. . This typically occurs when threads acquire locks on multiple resources in different orders.

A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object

In simpler terms, deadlock happens when:

1. Thread A holds Resource 1 and waits for Resource 2.
2. Thread B holds Resource 2 and waits for Resource 1

Ex

```
package com.threadex;


public class DeadlockExample {

    // Two lock objects to simulate resource locking
```

```
    private static final Object lock1 = new Object(); // Lock 1
    private static final Object lock2 = new Object(); // Lock 2

    public static void main(String[] args) {
        // Thread 1 attempts to acquire lock1 first, then lock2
        Thread thread1 = new Thread(() -> {
            synchronized (lock1) { // Acquiring lock1
                System.out.println("Thread 1: Holding lock 1...");
                try {
                    Thread.sleep(100); // Simulate some work while holding lock1
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 1: Waiting for lock 2...");
                synchronized (lock2) { // Attempts to acquire lock2
                    System.out.println("Thread 1: Acquired lock 2!");
                }
            } // Releases lock1 when the block ends
        });

        // Thread 2 attempts to acquire lock2 first, then lock1
        Thread thread2 = new Thread(() -> {
            synchronized (lock2) { // Acquiring lock2
```

```
                    System.out.println("Thread 2: Holding lock 2...");
                    try {
                        Thread.sleep(100); // Simulate some work while
holding lock2
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("Thread 2: Waiting for lock 1...");
                    synchronized (lock1) { // Attempts to acquire lock1
                        System.out.println("Thread 2: Acquired lock 1!");
                    }
                } // Releases lock2 when the block ends
            });

            // Start both threads
            thread1.start(); // Start thread1 execution
            thread2.start(); // Start thread2 execution
        }
}
```

Output

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 1: Waiting for lock 2...
Thread 2: Waiting for lock 1...
```

We can prevent Deadlock using **Lock Ordering**

Example: Preventing Deadlock Using Lock Ordering

Both threads acquire locks in the same order (lock1 first, then lock2).

```
package com.threadex;

public class PreventDeadlockWithOrder {
   private static final Object lock1 = new Object();
   private static final Object lock2 = new Object();

   public static void main(String[] args) {
      // Thread 1 acquires lock1, then lock2
      Thread thread1 = new Thread(() -> {
         synchronized (lock1) {
            System.out.println("Thread 1: Acquired lock1");
            synchronized (lock2) {
               System.out.println("Thread 1: Acquired lock2");
            }
         }
      });

      // Thread 2 also acquires lock1 first, then lock2
      Thread thread2 = new Thread(() -> {
         synchronized (lock1) {
            System.out.println("Thread 2: Acquired lock1");
            synchronized (lock2) {
               System.out.println("Thread 2: Acquired lock2");
            }
         }
      });

      thread1.start();
      thread2.start();
```

```
        }
    }
```

Output

Thread 1: Acquired lock1

Thread 1: Acquired lock2

Thread 2: Acquired lock1

Thread 2: Acquired lock2

Livelock

A livelock happens when threads keep responding to each other and changing state but no thread makes progress.

These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

```
public class VerySimpleLivelock {
  static class Friend {
    private final String name;
    private boolean isGivingWay = true;
    public Friend(String name) {
      this.name = name;
    }
    public void pass(Friend other) {
      while (isGivingWay) {
        System.out.println(name + ": You go first, " + other.name + "!");
        try {
          Thread.sleep(100);
        } catch (InterruptedException ignored) {}
        // If other is still giving way, continue waiting
        if (!other.isGivingWay) {
          isGivingWay = false;
          System.out.println(name + ": Okay, I'm going now.");
        }
```

```
        }
    }
  }
  public static void main(String[] args) {
    Friend a = new Friend("Ram");
    Friend b = new Friend("Mah");
    new Thread(() -> a.pass(b)).start();
    new Thread(() -> b.pass(a)).start();
  }
}
```

Ram: You go first, Mah!
Mah: You go first, Ram!
Ram: You go first, Mah!
Mah: You go first, Ram!
Ram: You go first, Mah!
Mah: You go first, Ram!
……infinity

What is thread starvation?

Thread starvation is basically a situation or condition where a thread won't be able to have regular access to shared resources and therefore is unable to proceed or make progress. This is because other threads have high priority and occupy the resources for too long. This usually happens with low-priority threads that do not get CPU for its execution to carry on.

Example Scenario (Simplified):

Imagine there are 3 threads in a system:

- **Thread 1 (High Priority)**: Keeps executing and consumes all CPU time.
- **Thread 2 (Medium Priority)**: Executes occasionally, but still not enough to compete with Thread 1.
- **Thread 3 (Low Priority)**: Rarely gets CPU time and might never execute if Thread 1 and Thread 2 keep running.

In such a case, **Thread 3** is "starved" because the **higher-priority threads** consume all the available CPU time, preventing Thread 3 from executing.

Feature	Deadlock	Starvation	Livelock
Definition	Processes are blocked forever, each waiting for a resource held by another.	A process waits indefinitely because it is always bypassed by others.	Processes keep executing but fail to make progress.
Cause	Circular wait and resource holding.	Unfair resource allocation or scheduling.	Processes continuously respond to each other, preventing progress.
Process State	Blocked (not executing).	Ready but not scheduled/executed.	Actively executing but not making progress.
System Progress	No progress at all.	System progresses, but some processes do not.	System is busy, but no real work is done.
Example	A waits for B's resource; B waits for A's resource.	A low-priority task never gets CPU time.	Two processes constantly yielding to each other.
Resolution	Requires deadlock detection and recovery.	Use of fair scheduling (e.g., aging).	Needs better coordination or back-off strategies.

What about the daemon threads?

The daemon threads are the low priority threads that provide the background support and services to the user threads. Daemon thread gets automatically terminated by the JVM if the program remains with the daemon thread only, and all other user threads are ended/died. There are two methods for daemon thread available in the Thread class:

- public void **setDaemon**(boolean status): It used to mark the thread daemon thread or a user thread.
- public boolean **isDaemon()**: It checks if the thread is daemon or not.

Ex:

```
package com.threadex;
```

```
public class TestDaemonThread1 extends Thread {
	public void run() {
		if (Thread.currentThread().isDaemon()) {// checking for daemon thread
			System.out.println("daemon thread work");
		} else {
			System.out.println("user thread work");
		}
	}
	public static void main(String[] args) {
		TestDaemonThread1 t1 = new TestDaemonThread1();// creating thread
		TestDaemonThread1 t2 = new TestDaemonThread1();

		t1.setDaemon(true);// now t1 is daemon thread
		t1.start();// starting threads
		t2.start();
	}
}
```

Output

daemon thread work

user thread work

> Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

```
t1.start();
t1.setDaemon(true);//will throw exception here
```

Difference b/w Runnable Interface and Callable Interface

The **Runnable** and **Callable** interfaces in Java are both used to represent tasks that can be executed by a thread, but they have **key differences** in terms of functionality and use cases.

Runnable Interface:

1. It **does not return any result** and therefore, **cannot throw checked exceptions**.

   Runnable:

   - Does **not** return a result.
   - Method: public void run()

2. It **cannot be passed** to invokeAll() method directly (since invokeAll() requires a collection of Callable tasks).
3. Introduced in **JDK 1.0**.
4. Part of the **java.lang** package.
5. Uses the **run()** method to define a task.
6. To use this interface, you need to **override the run()** method.

✅ **Callable Interface:**

1. It **returns a result** and therefore, **can throw checked exceptions**.

Callable<V>:
Returns a result of type V.

a. Method: public V call() throws Exception

2. It **can be passed** to invokeAll() and invokeAny() methods in ExecutorService.
3. Introduced in **JDK 5.0**, so cannot be used before Java 5.
4. Part of the **java.util.concurrent** package.
5. Uses the **call()** method to define a task.
6. To use this interface, you need to **override the call()** method.

What is volatile in Java?

The volatile keyword in Java is used to indicate that a variable's value will be **modified by different threads**. It ensures **visibility** of changes to variables across threads.

🟡 But: It **does not stop** two threads from changing it at the same time (not atomic).

Syntax:

volatile int counter;

When to Use:

Use volatile when:

- Multiple threads read and write to a variable.
- You do not need **atomicity** (e.g., counter++ is **not** safe with volatile).
- You only need **visibility**, not mutual exclusion.

What volatile Does:

- Ensures visibility: All threads see the latest value.
- Prevents instruction reordering by the compiler or CPU around the volatile variable.

Simple words

It is a **keyword** used with a variable to tell Java:

> "Hey! This variable can be changed by other threads — always check the latest value from memory."

Imagine this situation:

- Two threads:
  👉 **Thread A** is checking a variable.
  👉 **Thread B** can change that variable.

- Without volatile:
  Thread A **might not see** the change made by Thread B. It could be using an **old copy** (cached version).

- With volatile:
  Thread A will **always see the latest updated value** from Thread B.

```
class MyThread extends Thread {
  // Shared variable (volatile)
  private static volatile boolean isRunning = true;
  public void run() {
     System.out.println("Thread started");
     while (isRunning) {
        // Do some work...
     }
     System.out.println("Thread stopped");
  }
  public static void main(String[] args) throws InterruptedException {
     MyThread t = new MyThread();
     t.start();
     Thread.sleep(1000); // Wait 1 second
     isRunning = false;  // Stop the thread
  }
}
```

What Happens Here?

- Thread t runs in a loop while isRunning == true.
- After 1 second, main thread changes isRunning to false.
- Because isRunning is marked **volatile**, the running thread **immediately sees** the change and **stops the loop**.
- Without volatile, the thread **might not see the change**, and the loop keeps running forever.

✅ **In Short:**

Without volatile	With volatile
Thread may use **old value**	Thread always sees **latest value**
May lead to **bugs**	Helps avoid **visibility problems**

Important:

- volatile is for **visibility** only.
- It does **NOT** make things **atomic** (e.g., count++ is not safe even with volatile).

What are atomic variables?

Java Atomic Classes
Atomic variables in Java are part of the java.util.concurrent.atomic package.

An **atomic variable** is a special variable that allows **safe updates from multiple threads at the same time**, **without locks**.

✅ **It makes sure the value is updated safely, even when many threads try at once.**

🟢 It provides **both visibility and atomicity**.

Common atomic classes:

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference<T>

Key Methods:

- get()
- set(int newValue)
- incrementAndGet()
- decrementAndGet()
- compareAndSet(expect, update)

Simple Example:

```
AtomicInteger count = new AtomicInteger(0);
count.incrementAndGet(); // Increases value safely
```

```
package programms;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicExample {
  private static AtomicInteger counter = new AtomicInteger(0);
  public static void main(String[] args) throws InterruptedException {
     Runnable task = () -> {
        for (int i = 0; i < 1000; i++) {
           counter.incrementAndGet(); // atomic increment
        }
     };
     Thread t1 = new Thread(task);
     Thread t2 = new Thread(task);
     t1.start(); t2.start();
     t1.join(); t2.join();
```

```
        System.out.println("Final count: " + counter.get()); // Should be 2000
    }
}
```

Output

Final count: 2000

Volatile = "Everyone sees the latest value."

Atomic = "Everyone sees it and updates it safely."

🔸 Difference Between `volatile` and `atomic`

Feature	`volatile`	`Atomic` (e.g., `AtomicInteger`)
Visibility	✅ Yes – all threads see latest value	✅ Yes – all threads see latest value
Atomic operations	❌ No – not safe for updates	✅ Yes – safe for updates
Thread-safe?	❌ No	✅ Yes
Example Use	Flags, status signals	Counters, sequence numbers
Package/Type	Java keyword	From `java.util.concurrent.atomic`

How Does CompletableFuture Work in Java?

✅ What is CompletableFuture?

CompletableFuture is part of java.util.concurrent and introduced in Java 8.

It allows you to write **asynchronous**, **non-blocking** code that can:

- Run tasks in the background
- Combine multiple tasks
- Handle results or exceptions when the task completes

Explain the Executor framework

Before (Manual Threads):

In simple Java applications, we do not face many challenges while working with a small number of threads.

If we have to develop a program that runs a lot of concurrent tasks, this approach will present many disadvantages such as lots of boilerplate code (create and manage threads), executing threads manually and keeping track of thread execution results

- Had to manually create and start threads.
- No thread reuse → high resource usage.
- **Couldn't return results or handle exceptions properly.**
- Hard to manage many threads and scale the app.

After (Executor Framework):

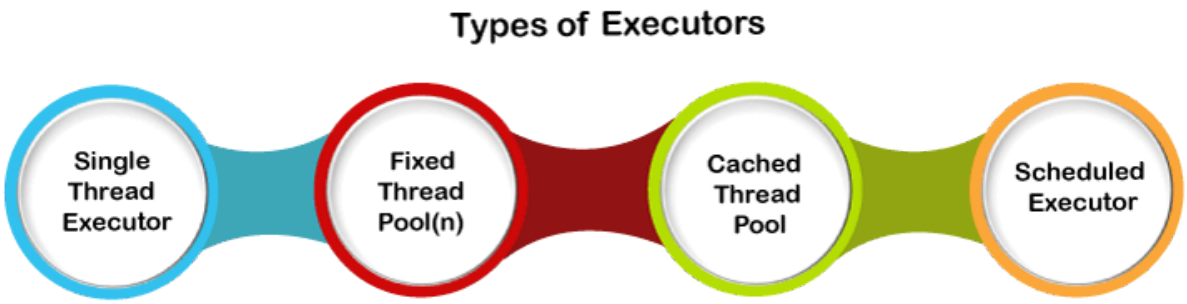
The **Executor Framework** in Java provides a high-level API to manage and control multiple threads efficiently.

- Uses **thread pools** for efficient thread reuse.
- Separates **task creation** from **execution**.
- Supports **Callable + Future** to return results and track task status.
- Simplifies error handling and improves scalability.
- Cleaner, more maintainable, and optimized for performance.

The framework consists of **three main interfaces** (and lots of child interfaces):

- *Executor*,
- *ExecutorService*
- *ThreadPoolExecutor*

Types of Executors



1) SingleThreadExecutor

The SingleThreadExecutor is a special type of executor that has only a single thread. It is used when we need to execute tasks in sequential order. In case when a thread dies due to some error or exception at the time of executing a task, a new thread is created, and all the subsequent tasks execute in that new one.

```
ExecutorService executor = Executors.newSingleThreadExecutor()
```

Code Example:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> System.out.println("Task executed by: " +
Thread.currentThread().getName()));
executor.shutdown();
```

2) FixedThreadPool(n)

A thread pool with a **fixed number (n) of threads**.

Thread Pooling :

Thread Pooling is a technique where a fixed number of threads are created **in advance** and reused to execute multiple tasks, rather than creating a new thread for each task.

```
    ExecutorService executor = Executors.newFixedThreadPool(4);
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExample {
	public static void main(String[] args) {
		ExecutorService executor = Executors.newFixedThreadPool(3); // Thread pool
with 3 threads
		// Reusable task
		Runnable task = () -> {
			System.out.println("Running in thread: " +
Thread.currentThread().getName());
		};
		// Submit the same task 5 times
		for (int i = 1; i <= 5; i++) {
			executor.submit(task); // Reuses threads from the pool
		}
		executor.shutdown(); // Shut down the executor
	}
}
```

```
Running in thread: pool-1-thread-1
Running in thread: pool-1-thread-2
Running in thread: pool-1-thread-3
Running in thread: pool-1-thread-3
Running in thread: pool-1-thread-1
```

Without Thread Pooling:

- Every time you start a new task, a **new thread is created**.
- Creating and destroying threads is **expensive** (CPU + memory

3) CachedThreadPool

The CachedThreadPool is a special type of thread pool that is used to execute short-lived parallel tasks. A thread pool that **creates new threads as needed**, but **reuses previously constructed threads** when available.

```
ExecutorService executor = Executors.newCachedThreadPool();
```

4) ScheduledExecutor

The ScheduledExecutor is another special type of executor which we use to run a certain task at regular intervals. It is also **used when we need to delay a certain task**.

```
ScheduledExecutorService scheduledExecService =
Executors.newScheduledThreadPool(1);
```

Code Example:

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

// Run after 5 seconds

scheduler.schedule(() -> {

  System.out.println("Delayed Task executed by: " + Thread.currentThread().getName());

}, 5, TimeUnit.SECONDS);
```

Benefits of Executor Framework

- The framework mainly separates task creation and execution. Task creation is mainly boilerplate code and is easily replaceable.
- With an executor, we have to create tasks that implement either **Runnable** or **Callable** interface and send them to the executor.
- **Executor** internally maintains a (configurable) thread pool to improve application performance by avoiding the continuous spawning of threads.
- **Executor** is responsible for **executing the tasks**, and **running them with the necessary threads from the pool**.

Another important advantage of the Executor framework is the use of the Callable interface. It's similar to the Runnable interface with two benefits:

1. Its **call()** method returns a result after the thread execution is complete.
2. When we send a Callable object to an executor, we get a Future object's reference. We can use this object to query the status of the thread and the result of the Callable object.

Example:

Before Java 5

```
public class ManualThreadExample {
  public static void main(String[] args) {
     Runnable task = new MyTask();  // Using a separate class
     Thread thread = new Thread(task);
     thread.start();
  }
}
class MyTask implements Runnable {
  @Override
  public void run() {
     System.out.println("Running in thread: " + Thread.currentThread().getName());
  }
}
```

Problems with manual thread approach:

- Have to **manually create/start** a new thread each time.
- No **thread reuse** — inefficient.
- Difficult to **manage many concurrent tasks**.
- Cannot easily track or return results.

After Java5 (Executor service):

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolExample {
        public static void main(String[] args) {
                ExecutorService executor = Executors.newFixedThreadPool(3); // Thread pool
with 3 threads
```

```
		// Reusable task
		Runnable task = () -> {
			System.out.println("Running in thread: " +
Thread.currentThread().getName());
		};
		// Submit the same task 5 times
		for (int i = 1; i <= 5; i++) {
			executor.submit(task); // Reuses threads from the pool
		}
		executor.shutdown(); // Shut down the executor
	}
}
```

```
Running in thread: pool-1-thread-1
Running in thread: pool-1-thread-2
Running in thread: pool-1-thread-3
Running in thread: pool-1-thread-3
Running in thread: pool-1-thread-1
```

very simple examples of using **ExecutorService with Runnable**, **Callable with Future**,

```
import java.util.concurrent.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class SimpleRunnableExample {
	public static void main(String[] args) {
		ExecutorService executor = Executors.newFixedThreadPool(1);
		MyTask task2 = new MyTask(); // Same task reused
		executor.submit(task2); // Submit task to executor
		executor.shutdown(); // Shut down executor
	}
}
// Task class implementing Runnable
class MyTask implements Runnable {
	@Override
	public void run() {
		System.out.println("Task is running in thread: " +
Thread.currentThread().getName());
	}
}
```

Output
Task is running in thread: pool-1-thread-1

Callable and Future

```
import java.util.concurrent.*;
import java.util.concurrent.*;
public class SimpleCallableExample {
	public static void main(String[] args) {
		ExecutorService executor = Executors.newSingleThreadExecutor();
		MyCallableTask task = new MyCallableTask(); // Create callable task
		Future<String> future = executor.submit(task); // Submit task to executor
		try {
			String result = future.get();
			System.out.println("Result from callable: " + result);
		} catch (InterruptedException | ExecutionException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		executor.shutdown();
	}
}
// Task class implementing Callable
class MyCallableTask implements Callable<String> {
	@Override
	public String call() throws Exception {
		return "Callable task executed by thread: " + Thread.currentThread().getName();
	}
```

Output

Result from callable: Callable task executed by thread: pool-1-thread-1

Part	Purpose
`Callable<String>`	Interface that returns a `String` result
`submit(task)`	Sends task to thread pool for execution
`Future<String>`	Placeholder for result
`future.get()`	Waits for task to complete and returns result
`executor.shutdown()`	Graceful shutdown of the executor