Xebia

Multi Threading

Understanding Threads

- Multithreading allows concurrent execution of multiple threads in a Java program.
- Threads can be created by extending the Thread class or implementing the Runnable interface.
- Java supports both concurrency and parallelism.
- Threads can exist in various states, including NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED.
- Synchronization mechanisms like synchronized blocks, locks, and semaphores are used to ensure thread safety.
- Thread priorities can be assigned, but they may not have a significant impact on all platforms.
- Thread groups organize and manage threads hierarchically but are rarely used in modern Java development.
- Proper synchronization and coordination are crucial for thread safety and avoiding race conditions and deadlocks.
- Java provides a rich set of concurrency utilities in the java.util.concurrent package.
- Threads can communicate and coordinate using mechanisms like wait() and notify(), condition variables, and thread barriers.
- Thread-local variables allow each thread to have its own independent copy of a variable.
- Thread pools manage a pool of worker threads, improving performance and resource utilization.
- Techniques like locks, semaphores, and atomic operations ensure thread safety and prevent deadlocks in shared resource access.

Need for Multi-Threading Programming

- Improved Performance
- Responsiveness
- Resource Utilization
- Concurrent Access to Resources
- Parallel Algorithms
- Task Parallelism
- Efficient I/O Operations
- Real-time Systems
- Scalability
- Frequent Updates
- Modularity and Maintainability
- Resource Sharing and Synchronization
- Asynchronous Programming

- **Program :**

  Set of Instruction to Perform a specific task

- **Process :**

  Time taken to Execute the Code

- **Processor :**

  it executes the Code

- **Thread :**

  Small execution of Code within process

- Thread is light weight Process
- Java is Thread based language
- C and C++ are process based language

- **Multi-tasking :**

  executing several task simultaneously

- **Multi-processing :**

  executing several task simultaneously where each task is independent of each other

  ex: typing program listening to music downloading file simultaneously

- **Multi-threading :**

executing several task simultaneously where each task is independent of each, but they are part of the same program

Ex: executing 2 separate block of the same program simultaneously

- **Thread scheduler :** is component of jvm whose work is to schedule the thread execution

Some common method of Thread class

currentThread() :

gives current thread details

sleep() **:**

used to sleep the thread

suspend() **:**

used to suspend the thread

resume() **:**

used to resume the thread

wait() **:**

used to wait the thread

notify() :

notify the thread which is waiting

notifyAll() :

notify all thread which are waiting

start() :

used to start new thread

stop() :

stop the thread

join() :

thread waits until another thread done its execution

setName() :

used to set the name to thread

getName() :

used to get the name of thread

setPriority() :

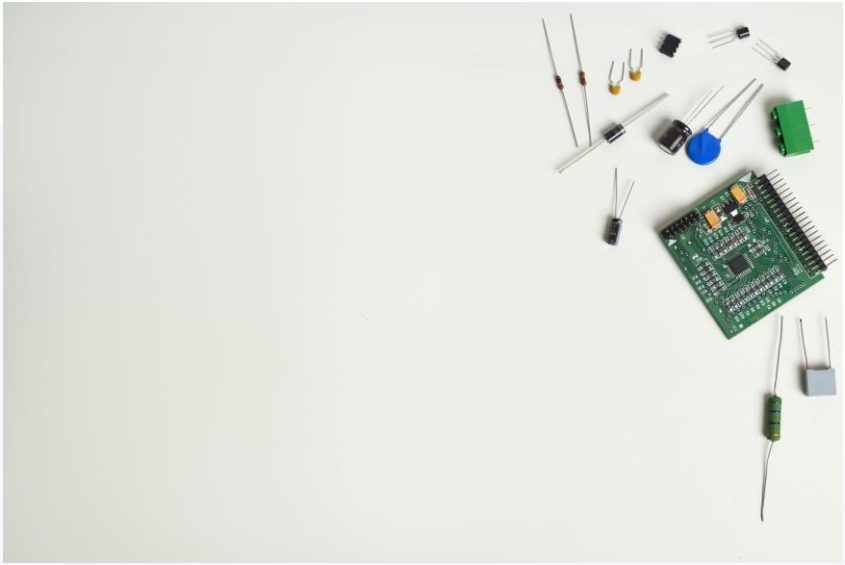
used to set the Priority to thread

getPriority() :

used to get the priority

activeCount() :

give the number of active thread



isDaemon() :

returns boolean value

setDaemon() :

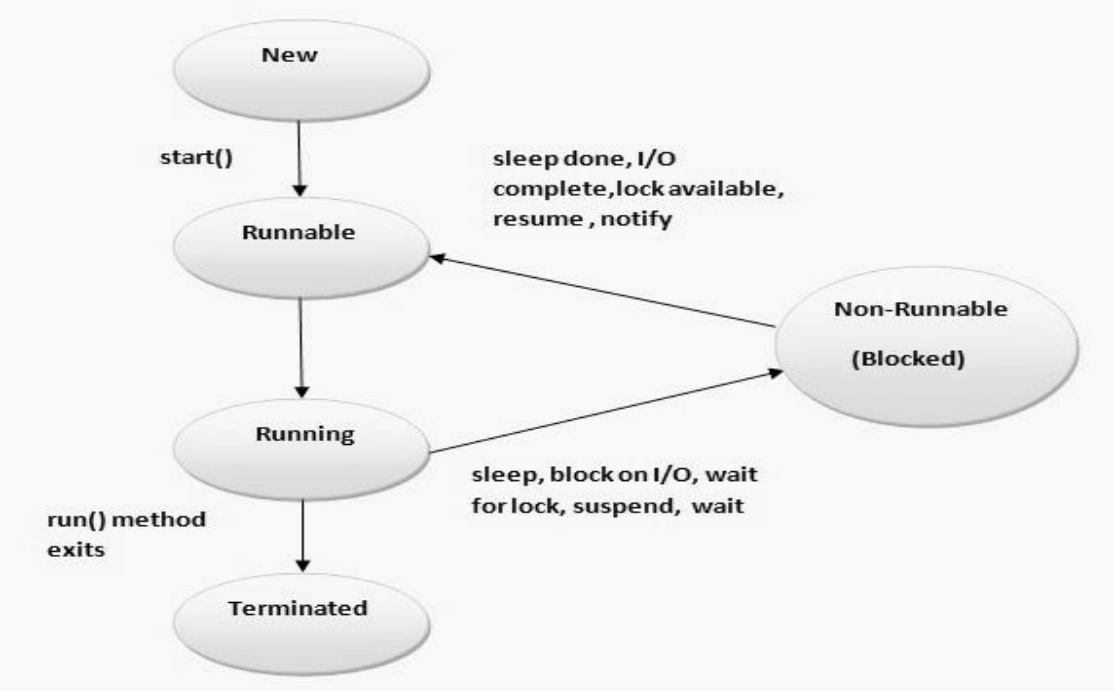
used to set thread Daemon

yield() :

checks any thread waiting which are having priority value greater than this thread if available it will give chance to execute the thread and goes into waiting state

Thread life cycle

1.New
2.Runnable
3.Running
4.Non-Runnable (Blocked)
5.Terminated(dead)



Thread Lifecycle

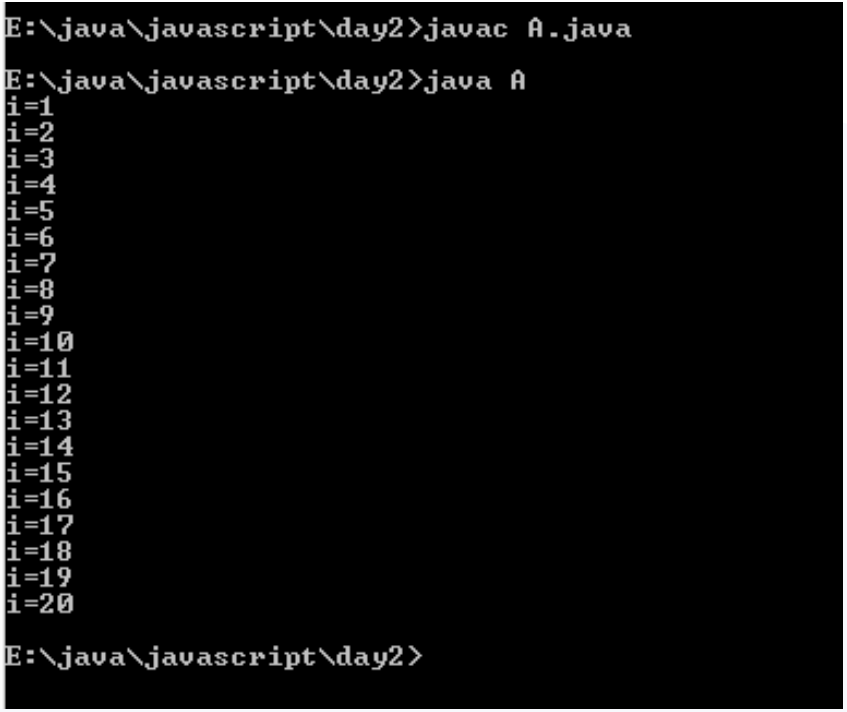
- New (NEW) state: The thread is created but not yet started.
- Runnable (RUNNABLE) state: The thread is eligible to run but may not be executing.
- Running (RUNNING) state: The thread is actively executing its code.
- Blocked (BLOCKED) state: The thread is waiting for a resource or condition.
- Waiting (WAITING) state: The thread is waiting for a specific condition to be met.
- Timed Waiting (TIMED_WAITING) state: The thread is waiting for a specified time period or until a specific time is reached.
- Terminated (TERMINATED) state: The thread has completed its execution or has been terminated.
- Threads transition between these states as they execute and wait for resources.
- Proper synchronization and state management are crucial for multi-threaded programming to avoid race conditions and ensure thread safety.

- Thread is pre-defined class belong to the package java.lang.Thread

Demo :normal program

```
class A
{
public static void main(String[] args)
{
        for(int i=1;i<=20;i++)
        {
        System.out.println(“I =”+i);
        }
}
}
```

Output : 1st and 2nd run



➢ **In 2 ways we can achieve multi-threading**

1.extending Thread class

2.implementing runnable interface

1.extending Thread class

- we need to extends Thread class and override run()
- Create Object of our class
- call start() on object of our class
- Calling start() will create new thread and automatically invoke run()

Demo :

```
class  A extends Thread
{
        public void run()
        {
                For(int  i=1;i<=20;i++)
                {
                        System.out.println(“i =“+i)          ;
                }
        }
}
class ThreadDemo
{
        Public static void main(String[] args)
        {
                A  a1=new A();
                A a2=new  A();
                a1.start();
                a2.start();
        }
}
```

Output :

1st run

```
C:\Users\sharu_000\Desktop>java ThreadDemo
i =1
i =1
i =2
i =2
i =3
i =3
i =4
i =4
i =5
i =5
i =6
i =6
i =7
i =7
i =8
i =8
i =9
i =9
i =10
i =10
i =11
i =12
i =13
i =11
i =12
i =13
i =14
i =15
i =14
i =16
i =15
i =17
i =18
i =16
i =19
i =20
i =17
i =18
i =19
i =20
```

2nd run

```
C:\Users\sharu_000\Desktop>javac ThreadDemo.java

C:\Users\sharu_000\Desktop>java ThreadDemo
i =1
i =2
i =1
i =3
i =2
i =4
i =3
i =5
i =4
i =6
i =5
i =7
i =6
i =8
i =7
i =9
i =8
i =10
i =9
i =11
i =10
i =12
i =11
i =13
i =12
i =14
i =13
i =15
i =14
i =16
i =15
i =17
i =16
i =18
i =17
i =19
i =20
i =18
i =19
i =20

C:\Users\sharu_000\Desktop>
```

Thread Priorities

- Thread priorities are represented as integers from Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10).
- The default priority is Thread.NORM_PRIORITY (5).
- Thread priorities influence the order in which threads are chosen to run by the thread scheduler.
- Higher-priority threads have a higher chance of being scheduled, but behavior is platform-dependent.
- Thread priorities can be used to give preference to critical or time-sensitive tasks.
- Avoid excessive reliance on thread priorities, as they may not behave consistently across platforms.
- Priority inversion can occur when lower-priority threads hold resources needed by higher-priority threads.
- Priority inheritance and priority ceiling protocols can be used to mitigate priority inversion.
- You can set a thread's priority using setPriority(int priority).
- You can get a thread's priority using getPriority().
- Modern Java applications often use high-level concurrency utilities for better control over thread execution and resource management.

Daemon Thread

- Daemon threads are those thread which provides support for other thread or non-daemon Threads
- Daemon threads will run in background

Ex : Garbage collector, main Thread

- Daemon Thread priority will less than non-daemon thread but based on situation jvm will increase them thread priority

- if there is no thread then jvm will terminate the daemon thread
- Methods for Daemon threads

    1. public void setDaemon(boolean status)

    used to set thread as Daemon Thread

    2. public boolean isDaemon()

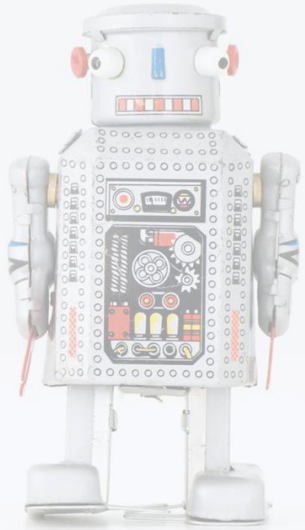
        return boolean value checks the thread is Daemon or not
- To make thread as daemon it should set Daemon before it starting otherwise illegalthreadstateException will throw

Thread Pool

- Thread pool is group of thread or worker threads which are waiting for job
- Advantage of thread pool is we can reuse the thread as many time hence performance will be increases
- In thread pool a group of fixed number of threads create and a thread from thread pool will be pulled and assign the job
- After completion of the job the thread will go back into the thread pool

- thread pools are used in jsp and servlet where container creates a thread pool to process the request
- The following code will create thread pool of 10 threads

Executor Service executor = Executors.newFixedThreadPool(10)



Thread group

Java provides a convenient way to group multiple threads in a single object

Thread group can be used to suspend resume interrupt group of thread by single method

Thread group implemented by *java.lang.ThreadGroup*

Constructors of ThreadGroup class

1. ThreadGroup(String name)

   create thread group of given name

2. ThreadGroup(ThreadGroup parent, String name)

   create thread group of given parent group and name

Methods in ThreadGroup

1. int activeCount()

   returns no. of threads running in current group

2. Int activeGroupCount()

returns a no. of active group in this thread group

3. Void destroy()

Destroy the thread group and all its subgroup

3. String getName()

returns the name of the group

4. ThreadGroup getParent()

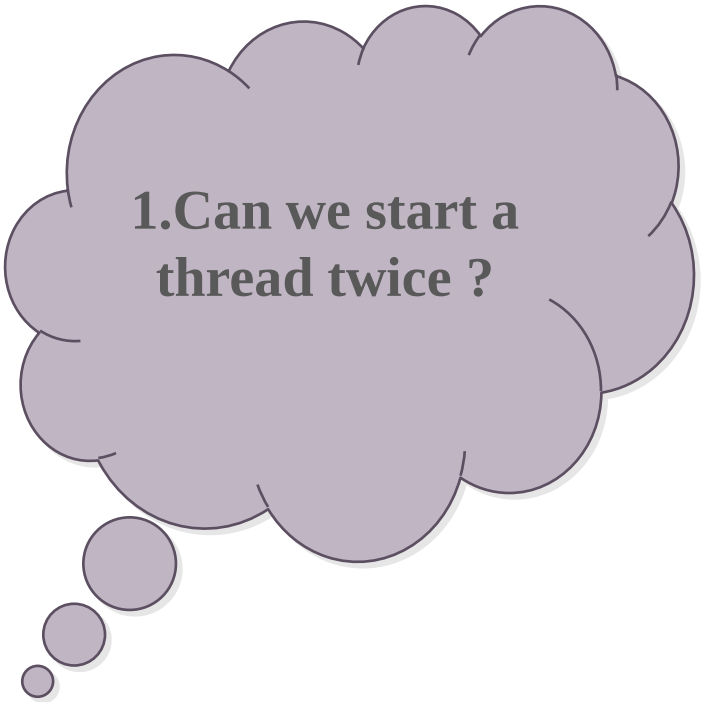
returns the parent of this group

5. Void interrupt()

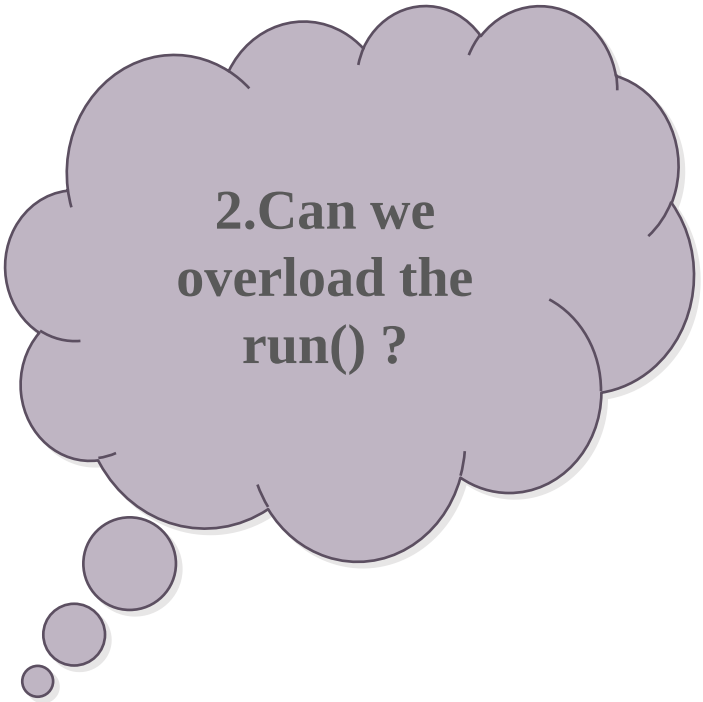
interrupt all the thread of the thread group

Synchronizing Threads

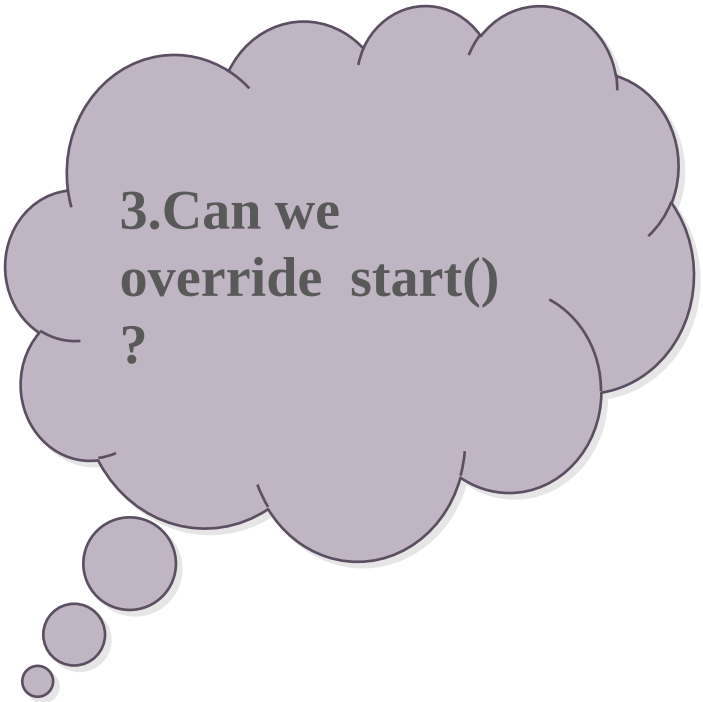
- Thread safety ensures that multiple threads can run concurrently without data corruption or inconsistencies.
- Synchronization mechanisms are used to coordinate access to shared resources and ensure thread safety.
- Java provides several synchronization mechanisms, including synchronized blocks and methods, locks, semaphores, and atomic classes.
- Synchronized blocks and methods use an intrinsic lock to ensure exclusive access to shared resources.
- Locks provide fine-grained control over synchronization and allow multiple locks to be acquired by a single thread.
- Semaphores are used to control access to a limited number of resources.
- Atomic classes provide atomic operations without explicit synchronization.
- The volatile keyword ensures that a variable's value is always read from and written to the main memory.



No (java.lang.IllegalThreadStateException)



Ans: : yes, but start method always look for default run() if not available then no output)



Ans :yes, but it will invoked as normal method

Ans : it will execute as normal method

Inter Communication of Threads

- Thread communication allows threads to coordinate their activities and share data in a multi-threaded environment.
- Common thread communication scenarios include producer-consumer, worker pool, reader-writer, barrier synchronization, and thread signaling.
- Java provides methods like wait(), notify(), and notifyAll() for thread signaling and coordination.
- wait(): Causes a thread to release the lock and wait until another thread invokes notify() or notifyAll() on the same object.
- Notify(): Wakes up one waiting thread that was previously waiting on the same object.
- NotifyAll(): Wakes up all waiting threads that were previously waiting on the same object.
- Condition variables, provided by java.util.concurrent.locks.Condition, allow for more complex thread coordination based on conditions.
- Blocking queues, such as java.util.concurrent.BlockingQueue, offer built-in thread communication for producer-consumer scenarios.

Critical Factor in Thread Deadlock

- **Deadlock** occurs in a multi-threaded environment when two or more threads are unable to proceed because each is waiting for a resource that the others hold.
- Critical factors contributing to deadlock include mutual exclusion, hold and wait, no preemption, and circular wait.

Deadlock Prevention Techniques:

- **Resource Hierarchy**:
  - Assign a hierarchy to resources and ensure that threads acquire resources in a predefined hierarchical order.
  - This ensures that all threads follow the same order when requesting resources, preventing circular waits.
- **Avoidance of Hold and Wait**:
  - Ensure that threads request all the resources they need at the beginning of their execution rather than acquiring them incrementally.
- **Timeouts**:
  - Set timeouts for resource acquisition attempts to limit the time a thread can wait for a resource.
  - If a timeout is reached, the thread can release any acquired resources and retry.
- **Resource Preemption**:
  - Allow higher-priority threads to preempt resources from lower-priority threads.
  - Preemption ensures that resources are released when needed by higher-priority threads.

Critical Factor in Thread Deadlock

Use of Locks with Timeouts:	• Use locks that support timeouts for resource acquisition to prevent indefinite waiting.
Graph-Based Algorithms:	• Implement graph-based algorithms to detect potential circular waits and take preventive actions.
Proper Resource Management:	• Ensure proper release of resources when they are no longer needed to avoid resource-holding situations.
Thread Design and Orderly Shutdown:	• Design threads and applications to avoid situations where threads may be indefinitely waiting for resources. • Implement orderly shutdown procedures to ensure that all threads can gracefully release resources.
Testing and Profiling:	• Thoroughly test and profile multi-threaded code to identify and diagnose potential deadlock scenarios. • Use profiling tools to analyze thread interactions.

Thread Executor Framework

- The Thread Executor Framework simplifies the management of threads and tasks in a multi-threaded environment.
- It provides a pool of worker threads and abstracts away low-level thread creation and management details.
- The core interfaces in the framework include Executor and ExecutorService.
- Executor defines a single method, execute(Runnable command), for executing tasks asynchronously.
- ExecutorService extends Executor and provides additional methods for task scheduling, thread management, and shutdown.
- Common implementations of ExecutorService include ThreadPoolExecutor, Executors.newFixedThreadPool(), Executors.newCachedThreadPool(), and Executors.newScheduledThreadPool().
- ThreadPoolExecutor allows for fine-grained control over thread pool parameters, such as core pool size, maximum pool size, and work queue type.
- Tasks are submitted to an executor for execution using methods like execute(Runnable command) or submit(Callable<T> task).
- Executor services manage the execution of tasks by assigning them to available worker threads.
- Properly shutting down an executor service is essential to release resources and terminate worker threads gracefully, typically done using shutdown().

Intro to Fork and Join Model

- **Parallel Computing Paradigm**: The Fork-Join Model is a parallel computing paradigm that focuses on breaking down tasks into smaller subtasks for concurrent execution.
- **Divide and Conquer**: It follows the "divide and conquer" approach, where complex tasks are decomposed into smaller, independent subtasks.
- **Fork**: "Fork" represents the process of splitting a task into smaller subtasks, creating a tree-like structure of subtasks.
- **Join**: "Join" represents the process of combining the results of subtasks to obtain the final result of the original task.
- **Work Stealing**: The Fork-Join Model employs a work-stealing algorithm to efficiently distribute and balance the workload among available threads.
- **Java's ForkJoinPool**: Java provides the ForkJoinPool class to manage and execute tasks using the Fork-Join Model. It handles thread management and task distribution.
- **RecursiveTask and RecursiveAction**: In Java, tasks in the Fork-Join Model are typically implemented using RecursiveTask (for tasks with results) and RecursiveAction (for tasks without results).
- **Parallelism**: The model leverages parallelism, allowing multiple CPU cores to execute subtasks concurrently, improving performance on multi-core processors.
- **Task Decomposition**: Tasks are recursively decomposed into smaller tasks until base cases are reached, making it suitable for divide-and-conquer algorithms.
- **Efficient Load Balancing**: The work-stealing algorithm ensures efficient load balancing by allowing idle threads to "steal" tasks from other busy threads.
- **Synchronization**: Proper synchronization and coordination are essential to ensure tasks are executed correctly and efficiently.

xebia.com