

Concurrency & Multithreading(Think like Real World scenario)

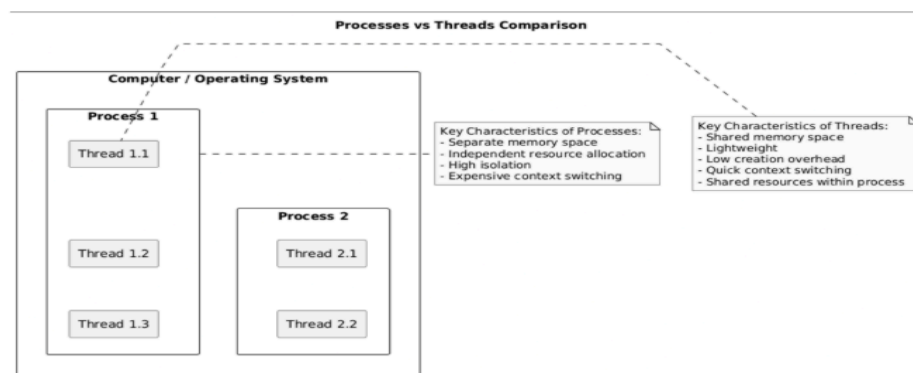
1.What Is Threads?

- Threads are fundamental units of execution that allow programs to perform multiple tasks concurrently.
- They enable developers to create responsive applications, utilize multi-core processors efficiently, and improve overall application performance.

2.Processes vs Threads?

Comparison Table: Processes vs Threads :

Characteristic	Process	Thread
Definition	An independent program with its own memory space	A lightweight unit of execution within a process
Memory	Separate memory space	Shared memory within the same process
Resource Allocation	Fully independent resource allocation	Shares resources with other threads in the same process
Overhead	High (creating a new process is resource-intensive)	Low (threads are lightweight and quick to create)
Communication	Inter-process communication requires complex mechanisms	Easy communication through shared memory
Isolation	High isolation between processes	Low isolation (threads can directly affect each other)
Switching Cost	Expensive context switching	Relatively inexpensive context switching
Failure Impact	One process crash doesn't typically affect others	Thread failure can potentially crash the entire process



3. When To Use Processes vs Threads?

Processes:

- You need strong isolation between different parts of an application
- Running completely independent tasks
- Leveraging multiple CPU cores for separate computational tasks

Threads:

- You need to perform multiple tasks within the same application
- Tasks need to share common data quickly
- You want to improve responsiveness and performance of a single application

4.Key Features of Threads?

- **Concurrent Execution:**
 - Multiple threads can run simultaneously, allowing programs to perform multiple tasks at once.
- **Resource Sharing:**
 - Threads within the same process share memory and resources, making communication between threads efficient.
- **Lightweight:**
 - Threads require fewer resources compared to creating multiple processes.

5.What Is Java Thread Class Structure?

```
package java.lang;
public class Thread implements Runnable {
    // 🗝️ Native thread handle (managed by JVM)
    private long threadId;
    // 👤 Thread name
    private String name;
    // 📊 Thread priority (1 to 10)
    private int priority;
    // 🎯 Target Runnable (if passed)
    private Runnable target;
    // 🔄 Thread group
    private ThreadGroup group;
    // 🧑 Whether thread is daemon
    private boolean daemon = false;
    // 🚦 Thread state (NEW, RUNNABLE, BLOCKED, etc.)
    private volatile Thread.State state;
    // 📁 Context Class Loader
    private ClassLoader contextClassLoader;
    // 🌱 Security and inherited access control context
    private AccessControlContext inheritedAccessControlContext;
    // =====
    // 🔨 Constructors
    // =====
    public Thread() { ... }
```

```

public Thread(Runnable target) { ... }
public Thread(Runnable target, String name) { ... }
public Thread(String name) { ... }
public Thread(ThreadGroup group, Runnable target, String name, long stackSize) { ... }
// =====
// 🎯 Core Methods
// =====
@Override
public void run() {
    if (target != null) {
        target.run(); // delegate if a Runnable is provided
    }
}
public synchronized void start() {
    // Native call to create and start new thread
}
public static void sleep(long millis) throws InterruptedException { ... }
public static void yield() { ... }
public final void join() throws InterruptedException { ... }
public void interrupt() { ... }
public boolean isAlive() { ... }
public void setPriority(int newPriority) { ... }
public int getPriority() { ... }
public void setName(String name) { ... }
public String getName() { ... }
public void setDaemon(boolean on) { ... }
public boolean isDaemon() { ... }
public Thread.State getState() { ... }
public static Thread currentThread() { ... }
public static boolean interrupted() { ... }
public boolean isInterrupted() { ... }
// =====
// 🔒 Lock & Monitor
// =====
public final void wait() throws InterruptedException { ... }
public final void notify() { ... }
public final void notifyAll() { ... }
// Many more internal and security-related methods...
}

```

6. How We Create Threads in Java?

- In Java, there are **two primary ways to create and work with threads**:

1. Extending the Thread Class :

- The Thread class provides the foundation for creating and managing threads in Java. By extending this class, you can override the run() method to define the code that need to be executed.
- **Code:**

```

1 class MyThread extends Thread {
2
3     // Override the run method to define thread behavior
4     @Override
5     public void run() {
6         for (int i = 0; i < 5; i++) {
7             System.out.println("Thread " + Thread.currentThread().getId() + " is running: " + i)
8             try {
9                 Thread.sleep(500); // Pause execution for 500 milliseconds
10            } catch (InterruptedException e) {
11                System.out.println("Thread interrupted");
12            }
13        }
14    }
15 }
16
17 public class ThreadExample {
18     public static void main(String[] args) {
19         MyThread thread1 = new MyThread(); // Create thread instance
20         MyThread thread2 = new MyThread(); // Create another thread instance
21
22         thread1.start(); // Start the first thread
23         thread2.start(); // Start the second thread
24     }
25 }

```

- **OutPut:{Note:EveryTime When We Run The Order Of Output Change}**

```

Thread id: 11 is running: 0
Thread id: 12 is running: 0
Thread id: 11 is running: 1
Thread id: 12 is running: 1
Thread id: 11 is running: 2
Thread id: 12 is running: 2
Thread id: 11 is running: 3
Thread id: 12 is running: 3
Thread id: 11 is running: 4
Thread id: 12 is running: 4

```

- **Extending Thread Class:**

- **Advantages:**
 - Simpler to implement for beginners
 - Direct access to Thread methods
- **Disadvantages:**
 - Limits inheritance (Java doesn't support multiple inheritance)
 - Each task requires a new thread instance

2. Implementing the Runnable Interface

- The Runnable interface provides a more flexible approach to creating threads. It separates the task from the thread itself, promoting better object-oriented design and allowing a class to extend another class while still being runnable in a separate thread.
- **Code:**

```
class Check{
    public void sum(){
        System.out.println(x:"Calling new class");
    }
}
//so by using runnable our thread(here MyRunnable) class can extend any other class which cannot possible in Thread extend(method 1)
class MyRunnable extends Check implements Runnable {
    @Override
    public void run(){
        for(int i=1;i<=5;i++){
            try{
                System.out.println(Thread.currentThread().getName() +" Running Task "+i);
                Thread.sleep(millis:1000);
            }
            catch (InterruptedException e){
                System.out.println(x:"Thread interrupted");
            }
        }
    }
}
public class Runnable1 {
    Run | Debug
    public void main(){
        MyRunnable myRunnable=new MyRunnable();
        Thread thread9=new Thread(myRunnable);
        Thread thread7=new Thread(myRunnable);
        myRunnable.sum();
        thread9.start();
        thread7.start();
    }
}
```

- **Output:** {Note:EveryTime When We Run The Order Of Output Change}

```
Runnable id: 11 is running: 0
Runnable id: 12 is running: 0
Runnable id: 11 is running: 1
Runnable id: 12 is running: 1
Runnable id: 11 is running: 2
Runnable id: 12 is running: 2
```

```
Runnable id: 11 is running: 3
Runnable id: 12 is running: 3
Runnable id: 11 is running: 4
Runnable id: 12 is running: 4
```

- **Implementing Runnable Interface:**
 - **Advantages:**
 - Better object-oriented design
 - Allows class to extend other classes
 - Same Runnable instance can be shared across multiple threads
 - More flexible for executor frameworks
 - **Disadvantages:**
 - Slightly more code to write
 - Indirect access to Thread methods
-

7.What is Callable Interface?

- It provides a more powerful alternative to Runnable. Unlike Runnable, Callable can return results and throw checked exceptions.
- **Key Features of Callable:**
 - **Return Values:** Callable tasks can return results, unlike Runnable tasks which return void.
 - **Exception Handling:** Callable's call() method can throw checked exceptions, while Runnable's run() method cannot.
 - **Future Objects:** Callable works with Future objects to retrieve results after task completion.
- **Code:**

```
import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.concurrent.ExecutionException;

class MyCallable implements Callable<String> {

    private final String name;

    public MyCallable(String name) {
```

```
        this.name = name }
```

```
@Override
```

```
public String call() throws Exception {
```

```
    StringBuilder result = new StringBuilder();
```

```
    for (int i = 0; i < 5; i++) {
```

```
        result.append("Callable ")
```

```
            .append(name)
```

```
            .append(" is running: ")
```

```
            .append(i)
```

```
            .append("\n");
```

```
        Thread.sleep(500); // Simulate work }
```

```
    return result.toString();}}
```

```
public class CallableExample {
```

```
    public static void main(String[] args) {
```

```
        // Create a thread pool with 2 threads
```

```
        ExecutorService executor = Executors.newFixedThreadPool(2);
```

```
        // Create Callable tasks
```

```
        Callable<String> callable1 = new MyCallable("Task 1");
```

```
        Callable<String> callable2 = new MyCallable("Task 2");
```

```
        try {
```

```
            // Submit tasks and receive Future results
```

```
            Future<String> future1 = executor.submit(callable1);
```

```
            Future<String> future2 = executor.submit(callable2);
```

```
            // Get and print results (blocking call)
```

```
            System.out.println("Result from first task:");
```

```
            System.out.println(future1.get());
```

```
            System.out.println("Result from second task:");
```

```
            System.out.println(future2.get());
```

```
        } catch (InterruptedException | ExecutionException e) {
```

```
            System.out.println("Task execution interrupted: " + e.getMessage());
```

```
        } finally {
```

```
            // Always shut down the executor
```

```
            executor.shutdown();
```

```
        }
```

```
    }
```

```
}
```

- **Output:**

Result from first task:

Callable Task 1 is running: 0

Callable Task 1 is running: 1

Callable Task 1 is running: 2

Callable Task 1 is running: 3

Callable Task 1 is running: 4

Result from second task:

Callable Task 2 is running: 0

Callable Task 2 is running: 1

Callable Task 2 is running: 2

Callable Task 2 is running: 3

Callable Task 2 is running: 4

- **Some Doubt's:**

- Both **Task 1** and **Task 2** are modifying the same variable **result** — how do we still get the correct output per task?
- If above is correct then we can get result like:

- Result from first task:

- Callable Task 1 is running: 0
- Callable Task 2 is running: 0
- Callable Task 1 is running: 1
- Callable Task 2 is running: 1
- Callable Task 2 is running: 2
- Callable Task 1 is running: 2

- But We did not get So there is something happening !
- That is Each task has **its own local result variable**. So there's **no shared data** between threads, and therefore no conflict or data corruption.

- ```

@Override
public String call() throws Exception {
 StringBuilder result = new StringBuilder(); // 🏠 Local to this thread
 for (int i = 0; i < 5; i++) {
 result.append("Callable ")
 .append(name)
 .append(" is running: ")
 .append(i)
 .append("\n");
 Thread.sleep(500);

```
- Example for Shared Resource by all Thread.
 

```

public class SharedDataCallable implements Callable<Void> {
 private static int counter = 0; // 🔥 Shared among all threads

 @Override
 public Void call() {
 for (int i = 0; i < 5; i++) {
 counter++; // 🚫 Needs synchronization!
 }
 return null;
 }
}

```

## 8. Thread vs Runnable vs Callable: Which to Choose?

| Feature             | Thread                            | Runnable                                  | Callable                                      |
|---------------------|-----------------------------------|-------------------------------------------|-----------------------------------------------|
| Return Result       | No                                | No                                        | Yes                                           |
| Throw Exceptions    | No (only unchecked)               | No (only unchecked)                       | Yes (checked & unchecked)                     |
| Creation Method     | <code>new Thread().start()</code> | <code>new Thread(runnable).start()</code> | <code>executorService.submit(callable)</code> |
| Return Type         | void                              | void                                      | Generic type <V>                              |
| Method to Implement | <code>run()</code>                | <code>run()</code>                        | <code>call()</code>                           |
| Introduced in Java  | JDK 1.0                           | JDK 1.0                                   | JDK 5                                         |
| Works with          | -                                 | Thread                                    | <code>ExecutorService</code> & Future         |

## 9. What is the difference between start() and run() methods?

- The start() method begins thread execution and calls the run() method, while the run() method simply contains the code to be executed. Directly calling run() won't create a new thread; it will execute in the current thread.
- you cannot use `notify()` (or `notifyAll()`) and `wait()` outside of a **synchronized** block or method.
- When two threads try to call two different synchronized methods on the same object at the same time, only one thread can execute at a time. This is because synchronized instance methods require the thread to acquire the object's monitor lock before entering the method. Once one thread holds the lock by entering a synchronized method, the other thread must wait until the lock is released. So, even though the methods are different, the lock on the object prevents them from running simultaneously. The second thread stays blocked until the first thread finishes and releases the lock. This ensures that only one thread executes any synchronized method of that object at a time, preventing race conditions and ensuring thread safety.

```

java
Copy Edit

class MyClass {
 synchronized void methodA() {
 System.out.println(Thread.currentThread().getName() + " inside methodA");
 try { Thread.sleep(2000); } catch (InterruptedException e) {}
 }

 synchronized void methodB() {
 System.out.println(Thread.currentThread().getName() + " inside methodB");
 try { Thread.sleep(2000); } catch (InterruptedException e) {}
 }
}

```

○

## ○ Code:

```

class MyThread extends Thread {

 public void run() {

 System.out.println("Thread running: " + Thread.currentThread().getName());

 }
}

public class Main {

 public static void main(String[] args) {

 MyThread t1 = new MyThread();

 t1.start(); // ✅ Starts a new thread

 MyThread t2 = new MyThread();

 t2.run(); // ⚠️ Runs in the main thread!

 }
}

```

## ● Output:

Thread running: Thread-0

Thread running: main

## 10.Can we call the start() method twice on the same Thread object?

- No, calling start() twice on the same Thread object will throw an `IllegalThreadStateException`. A thread that has completed execution cannot be restarted.

## ● Code:

```

public class TestThread extends Thread {

 public void run() {

 System.out.println("Thread is running...");

 }

 public static void main(String[] args) {

 TestThread t = new TestThread();
 }
}

```

```

t.start(); // ✅ Starts a new thread and runs `run()`

t.start(); // ❌ Throws java.lang.IllegalThreadStateException

}

}

```

## ● Output:

Thread is running...

Exception in thread "main" java.lang.IllegalThreadStateException

```

at java.lang.Thread.start(Thread.java:....)

at TestThread.main(TestThread.java:....)

```

- When you call `start()`, the thread is marked as **started**. After a thread has completed its task (i.e., its `run()` method finishes), it **cannot be restarted**. The `Thread` object becomes **dead** (state: `TERMINATED`) after the first `start()` call.

## 12. What is thread safety and how can it be achieved?

- Thread safety refers to code that functions correctly during simultaneous execution by multiple threads. without causing any **race conditions**, **data corruption**, or **inconsistent results**.
- If multiple threads access and modify shared data, and the final result depends on the order/timing, then you need thread safety.
- Example:

```

class Counter {

 int count = 0;

 public void increment() {

 Count++; } }

```

- If two threads run `increment()` at the same time, both might read the same value, increment it, and overwrite each other's update → incorrect result.

### → Thread Safe Using **synchronized**:

```

class Counter { private int count = 0;

 public synchronized void increment() {

 count++; // Now only one thread can execute this at a time }

 public synchronized int getCount() {

 return count; }}

```

- Now access to `count` is **serialized**, and results will be correct even with multiple threads.

### → Ways to Achieve Thread Safety:

| Method                                   | Example                                                            |
|------------------------------------------|--------------------------------------------------------------------|
| <code>synchronized</code> blocks/methods | <code>synchronized (this) { ... }</code>                           |
| Locks ( <code>ReentrantLock</code> )     | <code>lock.lock()</code> and <code>lock.unlock()</code>            |
| Atomic classes                           | <code>AtomicInteger</code> , <code>AtomicBoolean</code> , etc.     |
| Concurrent Collections                   | <code>ConcurrentHashMap</code> , <code>CopyOnWriteArrayList</code> |
| Thread-local variables                   | <code>ThreadLocal&lt;T&gt;</code>                                  |
| Immutability                             | Using final objects/data structures                                |

### 13.What happens if an exception occurs in a thread's run method?

- If an uncaught exception occurs in a thread's `run()` method, the thread terminates. The exception doesn't propagate to the parent thread and doesn't affect other threads.

- **Code:**

```
class MyThread extends Thread {
 public void run() {
 try {
 throw new RuntimeException("Exception in thread");
 } catch (Exception e) {
 System.out.println("Caught exception in thread: " + e.getMessage()); } }}

public class Main {
 public static void main(String[] args) {
 MyThread t = new MyThread();
 t.start(); // Starts a separate thread

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

- **Output:**

```
Main thread is running
Caught exception in thread: Exception in thread
```

### 14.What's the difference between `sleep()` and `wait()`?

- `sleep()` causes the current thread to pause for a specified time without releasing locks. `wait()` causes the current thread to wait until another thread invokes `notify()` or `notifyAll()` on the same object, and it releases the lock on the object.

- **Code:**

```
class SharedResource {
 synchronized void waitExample() {
```

```

 System.out.println(Thread.currentThread().getName() + " is waiting...");

 try {

 wait(); // Releases the lock and waits

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName() + " resumed after notify.");

 }

 void notifyExample() {

 System.out.println("Notifying a waiting thread...");

 notify(); // Wakes up one thread waiting on this object's monitor

 }

}

public class WaitNotifyExample {

 public static void main(String[] args) {

 SharedResource shared = new SharedResource();

 Thread t1 = new Thread(() -> shared.waitExample(), "Thread-1");

 Thread t2 = new Thread(() -> {

 try {

 Thread.sleep(2000);

 synchronized (shared) {

 shared.notifyExample();

 }

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 }, "Thread-2");

 t1.start();

 t2.start();
 }
}

```

- **Output:**

Thread-1 is waiting...

Notifying a waiting thread...

Thread-1 resumed after notify.

- What Happens to the Idle Thread Once notify() or notifyAll() is Called?

- When notify() or notifyAll() is called, the waiting thread does not immediately start running. Instead, it follows these steps:
  1. When another thread calls notify(), one waiting thread is moved to the Blocked (or Runnable) State, but it does not start execution immediately.
  2. The notified thread cannot resume execution until it successfully acquires the lock on the synchronized object.
  3. If multiple threads are waiting, only one gets notified by notify(), while notifyAll() wakes up all waiting threads (but they still compete for the lock).
  4. Once the thread reacquires the lock, it continues execution from where it called wait().
- What If We Use notifyAll()?
  - If we replace notify(); with notifyAll();, all waiting threads will be notified, but only one will acquire the lock first as they will compete for the lock, and execution depends on the thread scheduler.
- What Happens to the Resource a Thread Was Holding when the wait() method is called?
  1. When a thread calls wait(), it releases the lock on the synchronized object it was holding.
  2. Other threads can now acquire the lock and continue execution.
  3. The waiting thread remains idle until another thread calls notify() or notifyAll().
- What Happens to the Resource a Thread Was Holding when the sleep() method is called?
  - When a thread calls sleep(), it pauses execution for the specified time.
  - However, it does NOT release any locks it was holding.
  - Other threads cannot access synchronized resources held by the sleeping thread

## 15.Can you use Callable with standard Thread objects?

- No, you cannot directly use Callable with the Thread class. Callable is designed to work with the ExecutorService framework. Thread class only accepts Runnable objects. However, you can adapt a Callable to work with Thread by creating a Runnable that executes the Callable and stores its result:
  - **Code:**

```

class MyRunnable implements Runnable {
private Callable<Integer> callable;
public MyRunnable(Callable<Integer> callable) {
 this.callable = callable; }
public void run() {
 try {
 System.out.println(callable.call());
 } catch (Exception e) {
 e.printStackTrace(); } }}

```

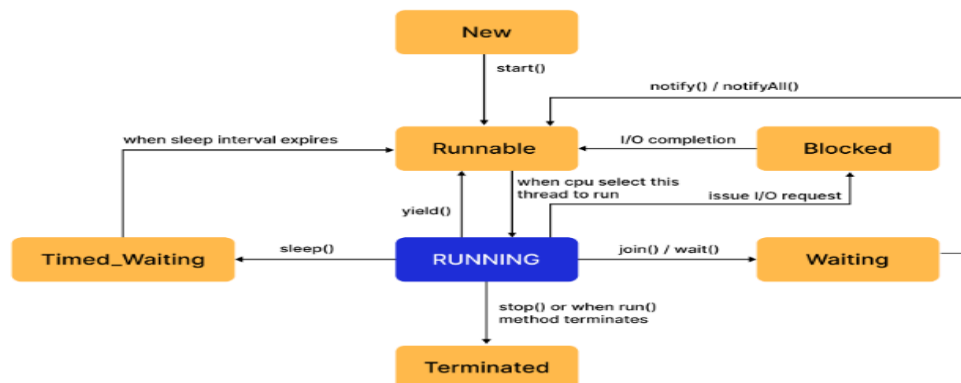
## 16.What is a Functional Interface in Java?

- A functional interface is an interface that contains exactly one abstract method.
- These interfaces can be used as the target for lambda expressions.
- Can have **default** and **static** methods (they don't count as abstract).
- Example Runnable interface

## 17.What is a Lambda Expression in Java?

- A **lambda expression** is a **short, anonymous way** to write an implementation of a **functional interface** — meaning an interface with a **single abstract method**.
- (parameters) -> { body }

## 18.What is Thread Life cycle?



### ■ New State:

```

Thread thread = new Thread(() ->
System.out.println("Hello from thread"));
Thread is in NEW state here

```

- **Runnable:** Thread is ready for execution and waiting for CPU allocation

```
Thread thread = new Thread(() ->
System.out.println("Hello from thread"));
thread.start(); // Thread moves to RUNNABLE state
```

- **RUNNING:**The thread is currently executing its task on the CPU. The CPU scheduler has allocated processing time to this thread. **Means Run() is executing**
- **BLOCKED:**Thread is temporarily inactive while waiting to acquire a lock Typically occurs when trying to enter a synchronized block/method already locked by another thread
- **WAITING:**Thread is waiting indefinitely for another thread to perform a specific action before it proceeds.Entered via methods like Object.wait(), Thread.join(), or LockSupport.park() .No timeout specified meaning the thread will remain stuck indefinitely unless another thread wakes it up using the notify() or notifyAll() method.
- **TIMED\_WAITING:**Thread is waiting for a specified period of time Entered via methods like Thread.sleep(timeout), Object.wait(timeout), etc. Will automatically return to RUNNABLE after timeout expires or notification
- **TERMINATED :**Thread has completed its execution or was stopped .The run() method has exited, either normally or due to an exception .Thread object still exists but cannot be restarted

## 19.what is Thread Pools in Java?

- Thread pools are a managed collection of reusable threads designed to execute tasks concurrently. They offer significant advantages in resource management, performance, and application stability.
- when the task is completed by a Thread it go back to pool and we can again use that thread for performing other task.

- **Code:**

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
// Task class that implements Runnable
class WorkerThread implements Runnable {
 private final int taskId;
 public WorkerThread(int taskId) {
 this.taskId = taskId;
 }
 @Override
 public void run() {
 System.out.println(Thread.currentThread().getName() + " is processing task: " + taskId);
 }
}
```



```

 try {
 // Simulate task taking 2 seconds to complete
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 System.out.println("Task interrupted: " + e.getMessage());
 }
 System.out.println(Thread.currentThread().getName() + " finished task: " + taskId);
}
}

public class ThreadPoolExample {
 public static void main(String[] args) {
 // Create a fixed thread pool with 3 threads
 ExecutorService executorService = Executors.newFixedThreadPool(3);
 // Submit 5 tasks to the pool
 for (int i = 1; i <= 5; i++) {
 executorService.submit(new WorkerThread(i));
 }
 // Shutdown the executor after submitting all tasks
 executorService.shutdown();
 // Optional: Wait for all tasks to finish (for better understanding)
 System.out.println("All tasks submitted. Thread pool is shutting down.");
 }
}

```

- **Output:{Output may Vary based on cpu}**

```

pool-1-thread-1 is processing task: 1
pool-1-thread-2 is processing task: 2
pool-1-thread-3 is processing task: 3
pool-1-thread-1 finished task: 1
pool-1-thread-1 is processing task: 4
pool-1-thread-2 finished task: 2
pool-1-thread-2 is processing task: 5
pool-1-thread-3 finished task: 3
pool-1-thread-1 finished task: 4
pool-1-thread-2 finished task: 5

```

#### **Explanation :**

##### **1. Thread Pool Creation** 🚀

- Executors.newFixedThreadPool(3) creates a pool with 3 reusable threads.

##### **2.Task Submission** 🚀

- Five tasks are submitted. Since only 3 threads exist, the first 3 tasks start immediately.
- As tasks complete, the available threads pick up the remaining tasks.

##### **2.Efficient Thread Usage** 🚀

- Threads are **reused**, avoiding the overhead of creating new threads for each task.
- The execution order may vary based on CPU scheduling.

- 

## **20.Combining Thread Lifecycle and Pools?**

- When new pool is created all Threads are in new state → When task is submitted they came to runnable state → Thread's state changes according to task operations(Blocked,waited etc) → task completion, the thread returns to the pool means new state waiting for another task → During shutdown, threads complete their current tasks and are eventually terminated.

## **21.how Thread Lifecycle Manage?**

- Handle InterruptedException properly to allow clean thread termination: A worker thread checking for updates in a loop should exit gracefully when interrupted instead of ignoring the exception.

```
class WorkerThread implements Runnable {
 @Override
 public void run() {
 try {
 while (!Thread.currentThread().isInterrupted()) {
 System.out.println("Checking for updates...");
 Thread.sleep(2000); // Simulating work
 }
 } catch (InterruptedException e) {
 System.out.println("Thread interrupted, shutting down gracefully.");
 }
 }
}

public class ThreadInterruptionExample {
 public static void main(String[] args) throws InterruptedException {
 Thread thread = new Thread(new WorkerThread());
 thread.start();
 Thread.sleep(5000); // Let it run for some time
 thread.interrupt(); // Interrupt the thread
 }
}
```

- Avoid thread leaks by ensuring threads don't get stuck in WAITING or BLOCKED states: Example : A thread waiting indefinitely for a signal can cause a leak. Use timeouts to prevent this while acquiring locks or waiting on conditions.

## 22.What Are Thread Pool Usage?

- CPU-intensive tasks → Executors.newFixedThreadPool(n)
- CPU-bound tasks (like image processing, video encoding, or complex calculations) spend most of their time using the CPU, rather than waiting for external resources.
- Too many threads can lead to excessive context switching, slowing down performance.
- A fixed number of threads (equal to the number of CPU cores, code for finding number of cpu code → `Runtime.getRuntime().availableProcessors();`) ensures that CPU resources are fully utilized without excessive overhead.
- Reduces overhead of thread creation and destruction.

## 23.What happens to a thread in a thread pool after it finishes executing a task?

- After task completion, the thread doesn't terminate but returns to the pool, ready to execute another task. This reuse eliminates the overhead of constantly creating and destroying threads.

## 24.Can a thread in TIMED\_WAITING state move directly to TERMINATED state?

- Yes, if the thread is interrupted during TIMED\_WAITING, it can throw an InterruptedException and complete its run method, transitioning to TERMINATED state.

## 25.What is thread starvation and how can thread pools help prevent it?

- Thread starvation occurs when threads are unable to gain regular access to shared resources and make progress. Thread pools help prevent this by controlling the number of active threads and implementing fair scheduling policies.

## 26.What is Thread Synchronization in Java?

- Thread synchronization is a critical concept in multithreaded programming that ensures multiple threads access shared resources in a controlled manner. Proper synchronization prevents data corruption, race conditions , and ensures thread safety in a concurrent environment.

## 27.What Are Methods of Thread Synchronization?

- The synchronized keyword is used to control access to critical sections of code so that only one thread can execute the synchronized code at a time. This ensures that shared mutable data is not corrupted by concurrent modifications
- There are two common ways to achieve this:

### 1 Synchronized Method

- When you declare an entire method as synchronized, the lock is acquired on the object instance (or on the Class object for static methods) before the method is executed and released after it finishes .
- This is useful when the whole method represents a critical section where no concurrent execution is desired. It is straightforward and reduces the chance of forgetting to protect part of the code .
- **.code:**

```
public class CounterSyncMethod {
 private int count = 0;
 // The entire method is synchronized.
 public synchronized void increment() {
 System.out.println("Synchronized Method - Start increment: " +
 Thread.currentThread().getName());
 // Critical section: updating the shared counter
 count++;
 System.out.println("Synchronized Method - Counter value after
 increment: " + count);
 System.out.println("Synchronized Method - End increment: " +
 Thread.currentThread().getName());
 }
 public int getCount() {
 return count;
 }
}
```

### 2 Synchronized Block


- A synchronized block allows you to specify a particular block of code to be synchronized, along with the object on which to acquire the lock (often called a monitor). This is more fine-grained compared to a synchronized method.
- You can perform non-critical work outside the block, while only protecting the portion of code that truly requires exclusive access .
- This can improve performance if only a subset of the method's operations need synchronization.
- The primary reason to choose a synchronized block over a synchronized method is when you have additional work in the method that doesn't need to be synchronized. This allows concurrent threads to execute the non-critical sections without waiting for the lock .
- **Code :**

```

public class CounterSyncBlock {
 private int count = 0;
 // Explicit lock object for finer control.
 private final Object lock = new Object();
 public void increment() {
 // Non-critical part: runs without locking.
 System.out.println("Non-Synchronized part (pre-processing): " +
 Thread.currentThread().getName());
 // Critical section: only this part is synchronized.
 synchronized (lock) {
 System.out.println("Synchronized Block - Start increment: " +
 Thread.currentThread().getName());
 count++;
 System.out.println("Synchronized Block - Counter value after increment: " + count);
 System.out.println("Synchronized Block - End increment: " +
 Thread.currentThread().getName());
 }
 // Non-critical part: runs after the synchronized block.
 System.out.println("Non-Synchronized part (post-processing): " +
 Thread.currentThread().getName());
 }
 public int getCount() {
 return count;
 }
}

```

## 28.What Is Volatile Keyword in Java?

- The volatile keyword in Java is used to indicate that a variable's value will be modified by multiple threads. Declaring a variable as volatile ensures two key things:
- **1 Visibility**
  - When a variable is declared volatile, its value is always read from and written to the main memory  instead of a thread's local cache.
  - This means changes made by one thread are immediately visible to others.
  - Without volatile, updates in one thread might not be seen (or might be delayed) by others due to caching .
- **2 Ordering**
  - volatile establishes a happens-before relationship .
    - Operations on a volatile variable cannot be re-ordered relative to each other.
    - This is especially helpful when using flags or controlling execution flow to ensure instructions are executed in the intended order.

## 29.When to Use volatile ?

1. Flags and Status Variables
    - a. Used to signal threads (e.g., a shutdown flag or status switch).
  2. Singleton Patterns (with double-checked locking)
    - a. In lazy initialization patterns, volatile ensures that the constructed instance is visible to all threads correctly.
  3. Lightweight Synchronization
    - a. If you only need visibility guarantees (not atomicity for compound actions like x++), volatile is lighter and faster than using synchronized.
- **Code:**

```

public class VolatileExample {
 // 'volatile' ensures visibility of changes across threads immediately
 private volatile boolean running = true;
 // Method run by the worker thread
 public void runTask() {
 System.out.println("WorkerThread: Starting execution...");
 }
}

```

```

int counter = 0;
// Loop runs until 'running' becomes false
while (running) {
 counter++;
}
System.out.println("WorkerThread: Detected stop signal. Final counter value: " + counter);
}
// Called by main thread to stop the worker thread
public void stopTask() {
 running = false;
}
public static void main(String[] args) {
 VolatileExample example = new VolatileExample();
 Thread workerThread = new Thread(() -> example.runTask(), "WorkerThread");
 workerThread.start();
 // Let worker run for 2 seconds
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println("MainThread: Stopping the worker thread.");
 example.stopTask(); // Signal worker to stop
 // Wait for worker thread to finish
 try {
 workerThread.join();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println("MainThread: Execution finished.");
}
}

```

- **Output:**

WorkerThread: Starting execution...

MainThread: Stopping the worker thread.

WorkerThread: Detected stop signal. Final counter value:  
123456789

MainThread: Execution finished.

### 30. What is Atomic Variable?

- Atomic variables in Java—found in the `java.util.concurrent.atomic` package—are designed to support lock-free, thread-safe operations on single variables.
- You should use atomic variables when you need to perform simple operations (like incrementing, decrementing, or updating) on shared variables in a multithreaded environment. They are especially useful when the overhead of locking is undesirable and when the logic remains limited to single-step atomic operations.

**Example: Code**

```

import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounterExample {
 // The AtomicInteger counter provides atomic methods for thread-safe
 operations.
 private AtomicInteger counter = new AtomicInteger(0); // 0 is initial value of
 counter
 // This method atomically increments the counter and prints the updated value.

```

```

public void increment() {
 int newValue = counter.incrementAndGet(); // Atomically increments the value.
 System.out.println(Thread.currentThread().getName() + " incremented counter to " + newValue); }
// Retrieves the current counter value.
public int getCounter() {
 return counter.get(); }
// Main method to run the AtomicCounterExample.
public static void main(String[] args) {
 final AtomicCounterExample example = new AtomicCounterExample();
 int numberOfThreads = 10;
 // Each thread will perform 100 increments.
 int incrementsPerThread = 100;
 Thread[] threads = new Thread[numberOfThreads];
 // Create and start threads that perform increments on the atomic counter.
 for (int i = 0; i < numberOfThreads; i++) {
 threads[i] = new Thread(new Runnable() {
 public void run() {
 for (int j = 0; j < incrementsPerThread; j++) {
 example.increment(); }, "Thread-" + (i + 1));
 threads[i].start();
 }
 });
 }
 // Wait for all threads to complete execution.
 for (int i = 0; i < numberOfThreads; i++) {
 try {
 threads[i].join();
 } catch (InterruptedException e) {
 e.printStackTrace(); }
 }
 // Display the final counter value.
 System.out.println("Final counter value: " + example.getCounter()); }

```

- **Output :** The exact interleaving of thread prints may vary on every run, but the final counter value will consistently reflect the total number of increments performed.

```

 Final counter value: 1000

```

## 31.What is the Difference Between Atomic and Volatile?

| Feature       | volatile Variable                                     | AtomicInteger                                               |
|---------------|-------------------------------------------------------|-------------------------------------------------------------|
| Purpose       | Guarantees visibility of updates between threads      | Provides <b>atomic</b> and <b>thread-safe</b> operations    |
| Atomicity     | ✗ Not atomic (e.g., <code>count++</code> is not safe) | ✓ Atomic operations (e.g., <code>incrementAndGet()</code> ) |
| Thread-safety | ✗ Not thread-safe for compound actions                | ✓ Thread-safe                                               |
| Locks used?   | ✗ No locks                                            | ✗ No locks (uses low-level CAS - Compare-And-Swap)          |
| Use case      | Flags, status updates (e.g., <code>isRunning</code> ) | Counters, accumulators, atomic updates                      |

## 32.What is Thread Communication in Java?

- Thread communication is a fundamental concept in concurrent programming that allows multiple threads to coordinate and share data effectively. Proper thread communication is essential for building robust, efficient, and thread-safe applications.

## 33.What are Methods of Thread Communication?

- `wait()`, `notify()`, and `notifyAll()` Methods
- These methods work with a thread's monitor (the intrinsic lock on an object) to coordinate the execution between threads. They are used when one or more threads need to wait for a specific condition to occur while another thread notifies them of the change.

- 1. **wait():**
  - When a thread calls the wait() method on an object, it releases the monitor (lock) it holds on that object and goes into a waiting state.
  - *Use wait() when a thread needs to pause execution until some condition (usually represented by a shared variable) changes. For example, a consumer thread might wait for a producer to produce an item.*
- 2. **notify():**
  - The notify() method wakes up a single thread that is waiting on the object's monitor. If more than one thread is waiting, the scheduler chooses one arbitrarily.
  - *Use notify() when only one waiting thread needs to be awakened (e.g., when one resource becomes available) to continue its execution.*
- 3. **notifyAll():**
  - The notifyAll() method wakes up all threads that are waiting on the object's monitor.
  - *Use notifyAll() when a change in the condition may be relevant to all waiting threads. For instance, when a producer adds an item to a queue that multiple consumers might be waiting for, you want to wake all waiting threads so they can re-check the condition.*
- **Important:**
  - These methods must be called from within a synchronized context (a synchronized block or method) on the same object whose monitor the thread is waiting on. They work together with a shared condition (often a flag or another shared variable) that threads check in a loop to handle spurious wakeups.

### 34.What are Locks and Types of Locks in Java?

- Lock mechanisms are essential tools for controlling access to shared resources in concurrent Java applications. While the synchronized keyword provides basic locking functionality, the java.util.concurrent.locks package offers more sophisticated and flexible lock implementations.

### 35.Why Use Explicit Locks?

- The synchronized keyword has been part of Java since its inception, so why use explicit locks? Explicit locks offer several advantages:
  1. Greater flexibility:
    - a. Fine-grained control over lock acquisition and release
  2. Non-block-structured locking:
    - a. Acquire and release locks in different scopes .by using trylock(time) a thread try to acquire a key in specific time ,if it not acquire it in that specific time it return false ,so we can move our thread forward.
  3. Timed lock attempts:
    - a. Try to acquire a lock within a specified duration .Useful to avoid deadlock.
  4. Interruptible lock acquisition:
    - a. Allow threads to be interrupted while waiting for locks ,thic can be done using `lock.lockInterruptibly()`. This is **not possible** with `synchronized`.
  5. Fairness policies:
    - a. Optional first-come-first-served lock acquisition .

## 36.Explain The Lock Interface Hierarchy?

- The `java.util.concurrent.locks` package provides a rich set of interfaces and implementations:

### 1. Locks & ReentrantLock :

Locks in Java (via the Lock interface) offer more flexible and fine-grained control over synchronization than the built-in synchronized keyword. One of the most popular implementations is ReentrantLock, which is called “reentrant” because the thread that holds the lock can re-acquire it without causing deadlock .

**"reentrant" mean?** If a thread already holds a lock and tries to acquire it again (e.g., in a recursive method or by calling another method that also needs the lock), it won't block itself. Instead, it just increases a hold count and continues execution. When the thread releases the lock, it must release it as many times as it acquired it.

→**Code:**

```
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockDemo {
 private final ReentrantLock lock = new ReentrantLock();

 public void outerMethod() {
 lock.lock(); // Acquired once
 try {
 System.out.println(Thread.currentThread().getName() + " - Inside outerMethod");
 innerMethod(); // Calls another method that tries to acquire the same lock
 } finally {
 lock.unlock(); // Released once
 }
 }

 public void innerMethod() {
 lock.lock(); // Acquired again by same thread — allowed because of reentrancy
 try {
 System.out.println(Thread.currentThread().getName() + " - Inside innerMethod");
 } finally {
 lock.unlock(); // Released again
 }
 }

 public static void main(String[] args) {
 ReentrantLockDemo demo = new ReentrantLockDemo();
 Thread thread = new Thread(demo::outerMethod, "Worker-Thread");
 thread.start();
 }
}
```

**It provides additional capabilities such as:**

- Interruptible Lock Acquisition: Using `lockInterruptibly()`
- Try-Lock Methods: With or without timeouts
- Fairness Policies: To ensure threads acquire locks in the order requested
- It is used when you need advanced control over locking (e.g., trying to acquire a lock and/or setting up fairness) or when a portion of a critical section is complex and may require more nuanced lock handling .

### 2.ReentrantReadWriteLock (Read-Write Lock) :

- ReentrantReadWriteLock(found in the `java.util.concurrent.locks` package) divides the lock into two parts—a read lock and a write lock. It is useful when:
  - Multiple Threads Need to Read: They can do so concurrently if there's no writing.



- Exclusive Writing: When a thread is updating data, no other thread (reader or writer) is allowed to access the resource.
- It is used to improve performance in scenarios with many more read operations than writes.

→ **Code:**

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLogExample {

 private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
 private int logValue = 0;

 // Simulate some processing delay
 private void simulateWork() {
 try {
 Thread.sleep(500);
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 }

 // Writer method
 public void writeValue(String taskName, int newValue) {
 rwLock.writeLock().lock();
 try {
 System.out.println(taskName + " (write): Acquired write lock.");
 simulateWork();
 logValue = newValue;
 System.out.println(taskName + " (write): Updated logValue to " + logValue);
 } finally {
 System.out.println(taskName + " (write): Released write lock.");
 rwLock.writeLock().unlock();
 }
 }

 // Reader method
 public void readValue(String taskName) {
 rwLock.readLock().lock();
 try {
 System.out.println(taskName + " (read): Acquired read lock. Reading logValue: " + logValue);
 simulateWork();
 System.out.println(taskName + " (read): Finished reading.");
 } finally {
 System.out.println(taskName + " (read): Released read lock.");
 rwLock.readLock().unlock();
 }
 }

 public static void main(String[] args) {
 ReadWriteLogExample logExample = new ReadWriteLogExample();
 ExecutorService executor = Executors.newFixedThreadPool(4);

 // Submit multiple reader and writer tasks
 executor.submit(() -> logExample.readValue("Reader-1"));
 executor.submit(() -> logExample.readValue("Reader-2"));
 executor.submit(() -> logExample.writeValue("Writer-1", 100));
 executor.submit(() -> logExample.readValue("Reader-3"));
 executor.submit(() -> logExample.writeValue("Writer-2", 200));
 }
}
```

```

 executor.submit(() -> logExample.readValue("Reader-4"));
 executor.submit(() -> logExample.readValue("Reader-5"));
 executor.shutdown();
 try {
 if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
 System.out.println("Timeout: Some tasks did not finish in time.");
 }
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }

 System.out.println("All tasks completed.");
 }
}

```

### 37.What is the difference between synchronized and Reentrant Lock?

#### 1. Acquisition and Flexibility:

##### synchronized:

- The synchronized keyword is built into the language; it automatically acquires and releases the intrinsic lock (monitor) of an object.
- It is simple to use but offers only blocking behavior—it always waits indefinitely to acquire the lock.
- You cannot try to acquire a synchronized lock with a timeout or check if the lock is available (i.e., no non-blocking acquisition).

##### ReentrantLock:

- Part of the java.util.concurrent.locks package, ReentrantLock provides explicit lock management.
- It gives you extra flexibility—for instance, with methods such as tryLock()(with or without a timeout) you can attempt to acquire the lock in a non-blocking manner.
- It also supports interruptible lock acquisition (lockInterruptibly()) and fairness policies.

#### 2. Automatic vs. Manual Release:

##### synchronized:

- The lock is automatically released when the synchronized block or method exits (even if an exception occurs).

##### ReentrantLock:

- You must explicitly call unlock()(usually in a finally block) to ensure that the lock is released. This gives you additional control but also adds responsibility.

### 38.Explain Semaphore in Java?

- Semaphores are one of the most versatile synchronization mechanisms in concurrent programming. Unlike locks which typically enforce exclusive access, semaphores can control access by providing permits to a specific number of Threads to access a Shared resources, making them perfect for implementing resource pools, throttling mechanisms, and coordinating thread execution.

## ● What is a Semaphore?

- A semaphore is a synchronization primitive that maintains a count of permits. Threads can acquire these permits (decreasing the count) or release them (increasing the count). When a thread attempts to acquire a permit and none are available, the thread blocks until a permit becomes available or until it's interrupted.
- **Conceptually, a semaphore has two primary operations:**
  - **acquire():** Obtains a permit, blocking if necessary until one becomes available
  - **release():** Returns a permit to the semaphore

## 39.Explain Types of Semaphores?

### 1. Binary Semaphore

- A binary semaphore has only two states (0 or 1 permit) and is mainly used to enforce mutual exclusion, similar to a mutex or lock.
- **Code:**

```
import java.util.concurrent.Semaphore;
public class BinarySemaphoreExample {
 private static final Semaphore mutex = new Semaphore(1); // Binary semaphore with 1
 permit
 public static void main(String[] args) {
 Thread t1 = new Thread(() -> accessCriticalSection("Thread-1"));
 Thread t2 = new Thread(() -> accessCriticalSection("Thread-2"));
 t1.start();
 t2.start();
 }
 private static void accessCriticalSection(String threadName) {
 try {
 System.out.println(threadName + " is attempting to acquire the lock.");
 mutex.acquire(); // Acquire the semaphore
 System.out.println(threadName + " acquired the lock.");
 Thread.sleep(1000); // Simulate work in the critical section
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 } finally {
 mutex.release(); // Release the semaphore
 System.out.println(threadName + " released the lock.");
 }
 }
}
```

### 2.Counting Semaphore

- A counting semaphore allows multiple permits, making it suitable for managing access to a pool of resources. It can have any non-negative number of permits.
- **Code:**

```
import java.util.concurrent.Semaphore;
public class CountingSemaphoreExample {
 private static final Semaphore resourcePool = new Semaphore(3); // Semaphore with 3
 permits
 public static void main(String[] args) {
 for (int i = 1; i <= 5; i++) {
 final int threadNum = i;
 Thread t = new Thread(() -> accessResource("Thread-" + threadNum));
 t.start();
 }
 }
 private static void accessResource(String threadName) {
 try {
 System.out.println(threadName + " is attempting to acquire a permit.");
 resourcePool.acquire(); // Acquire a permit
 System.out.println(threadName + " acquired a permit.");
 Thread.sleep(2000); // Simulate resource usage
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 } finally {
 resourcePool.release(); // Release the permit
 System.out.println(threadName + " released the permit.");
 }
 }
}
```

```
}}}
```

#### 40. what are Common Use Cases of Semaphores ?

- **Managing Access to a Pool of Resources:**

- semaphores can control access by providing permits to a specific number of Threads to access a Shared resources
- ```
Semaphore resourcePool = new Semaphore(5); // 5 permits for 5 Threads
```
- Threads acquire permits to access the shared resource and release them after use

- **Implementing Producer-Consumer Pattern:**

- a. Semaphores can synchronize producer and consumer threads by using separate semaphores to track empty and filled slots in a buffer.
- b.

```
Semaphore emptySlots = new Semaphore(bufferSize); // Track empty slots
```
- c.

```
Semaphore filledSlots = new Semaphore(0); // Track filled slots
```

- **Controlling Concurrency Levels:**

- a. When performing parallel computations, semaphores can limit the number of threads running concurrently to avoid overwhelming the system
- b.

```
Semaphore maxThreads = new Semaphore(10); // Restrict to 10 threads at a time
```

- **Enforcing Mutual Exclusion (Binary Semaphore):**

- a. Binary semaphores act like mutexes to ensure that only one thread accesses a critical section at a time.
- b.

```
Semaphore mutex = new Semaphore(1); // Single permit for mutual exclusion
```

41. What's the difference between a Semaphore and a Lock?

- A Lock allows only one thread to access a resource at a time (mutual exclusion), while a Semaphore can allow a specified number of threads to access resources concurrently. A Lock is owned by a specific thread that must release it, whereas Semaphore permits can be acquired and released by different threads. Locks support multiple condition variables, while Semaphores work on a simpler permit-based model.

42. What happens if a thread calls `release()` on a semaphore without first calling `acquire()`?

- In Java's Semaphore implementation, calling `release()` without a prior `acquire()` is perfectly legal. It simply increases the permit count beyond its initial value. This behavior can be useful in certain scenarios, such as dynamically increasing the number of available resources. However, this can lead to unexpected behavior if not managed carefully, as it might allow more concurrent access than originally intended.

- **Code:**

```
import java.util.concurrent.Semaphore;
public class SemaphoreReleaseExample {
    private final Semaphore semaphore = new Semaphore(2); // Initially allows 2 threads
    public void accessResource() {
        try {
            semaphore.acquire(); // Acquire a permit (there may be up to 3 available after the
            extra release)
            System.out.println(Thread.currentThread().getName() + " acquired semaphore");
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " released semaphore");
            semaphore.release(); // Release the permit
        }
    }
    public static void main(String[] args) {
        SemaphoreReleaseExample example = new SemaphoreReleaseExample();
        // Intentionally release a permit without acquiring one
        // This increases the available permit count from 2 to 3.
        example.semaphore.release();
        System.out.println("Permit count after extra release: " +
            example.semaphore.availablePermits());
        // Start multiple threads with descriptive names to use the semaphore.
        for (int i = 1; i <= 3; i++) {
            new Thread(example::accessResource, "SemaphoreThread-" + i).start();
        }
    }
}
```

43.What Is Java Concurrent Collections?

- Java Concurrent Collections are specialized thread-safe collection implementations designed for use in multithreaded environments. These collections provide better performance and scalability than their synchronized counterparts while ensuring thread safety.

44.What Are Major Concurrent Collections?

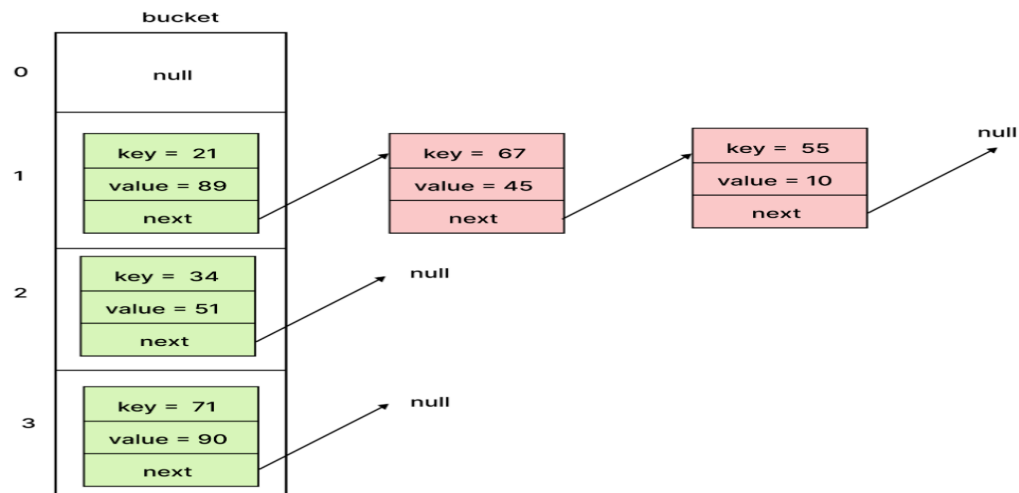
1. ConcurrentHashMap

• HashMap :

By itself, a HashMap is not thread-safe. To use it in a concurrent context, you typically have to lock (synchronize) all operations externally. This means that every operation (read or write) requires locking the entire map, which can be a performance bottleneck.

→**How HashMap works (Internals) :**

- HashMap internally uses an array of buckets to store entries. Each bucket can hold a linked list (or a balanced tree in later Java versions for better performance with many collisions) of Entry objects.
- When you put a key-value pair, the hash code of the key is calculated, and this hash code is used to determine the bucket where the entry should be placed.
- If multiple keys have the same hash code (hash collision), they are stored in the same bucket (as a linked list or tree).
- Operations like get, put, and remove involve calculating the hash code, finding the appropriate bucket, and then traversing the linked list/tree within that bucket to find or modify the entry.



→HashMap achieving Concurrency With Ensuring Thread safety :

- Locks the entire data structure. When any thread needs to modify the HashMap (e.g., add, remove, or even resize in some cases), it essentially needs exclusive access to the entire data structure. This prevents other threads from accessing this HashMap.

- Example: If one thread is putting a new entry into a HashMap, all other threads that want to read from or write to the map have to wait until the first thread is finished.

- `final Map<Integer, String> hashMap = Collections.synchronizedMap(new HashMap<>());`

- Even though `Collections.synchronizedMap` provides basic thread safety for individual operations, you still need to manually synchronize when iterating over it.

• ConcurrentHashMap :

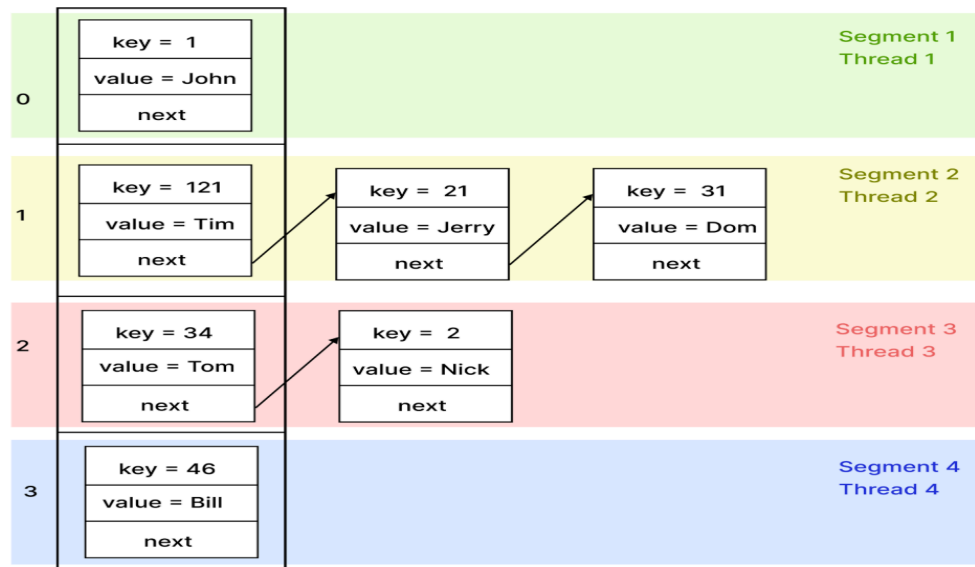
This map is designed for concurrency. It splits the locking (or uses non-blocking techniques) so that multiple threads can operate on the map without needing to lock the entire data structure. This leads to a much higher performance in multi-threaded environments, and you don't have to manage the locks manually.

Example: If one thread is putting a new entry into a HashMap, all other threads that want to read from or write to the map have to wait until the first thread is finished.

→ How ConcurrentHashMap Works (Internals):

The internal structure is similar to HashMap (using an array of nodes/buckets). However, concurrency is achieved through finer-grained locking at the bucket level (using synchronized on the first node of a bucket during modifications) and optimistic locking techniques using Compare-and-Swap (CAS) operations for read operations and some structural modifications. For read operations, most of the time, no locking is required.

Means when any write or modification operation is performing on any key(segment or bucket) then we lock only that segment and remaining all are free.



ConcurrentHashMap_achieving Concurrency With Ensuring

Thread safety:

- Locks in segments (or buckets). Instead of a single lock for everything, ConcurrentHashMap divides its internal data structure into multiple independent segments (in older versions) or individual buckets (in newer versions). Each segment or bucket has its own lock. Think of it as multiple smaller treasure chests, each with its own key. Different threads can hold the keys to different chests and access them concurrently without blocking each other.
- Example: If ConcurrentHashMap is divided into 16 segments, up to 16 different threads can potentially be performing write operations simultaneously, each on a different segment. Threads reading data generally don't even need to acquire a lock in the latest versions. This allows for much higher concurrency and better performance in multi-threaded applications compared to HashMap.
- ```
final ConcurrentHashMap<Integer, String>
concurrentMap = new ConcurrentHashMap<>();
```

### .2. CopyOnWriteArrayList:

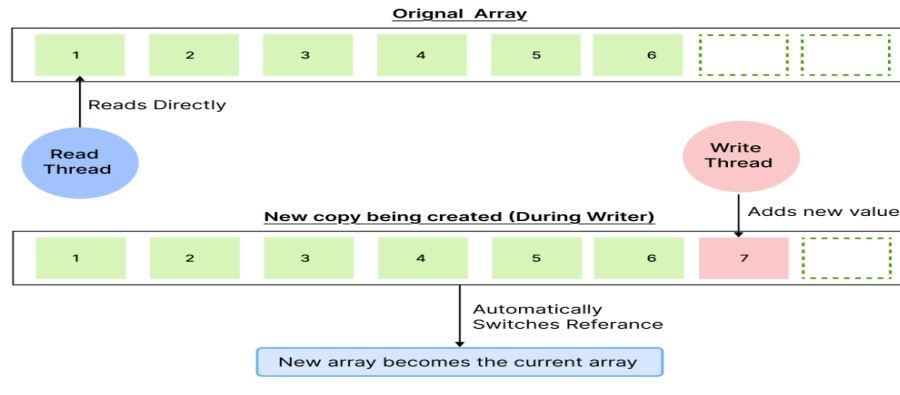
- CopyOnWriteArrayList is a thread-safe variant of ArrayList that creates a fresh copy of the underlying array for every modification operation. This unique approach ensures thread safety while providing non-blocking read operations, making it particularly well-suited for specific concurrency scenarios.
- **How ArrayList works (Internals):**
  - ArrayList internally uses a dynamic array to store elements. When the array reaches its capacity, a new larger array is created, and all elements are copied to the new array.
  - When you add or remove elements, the array is modified directly, and in multi-threaded environments, this can lead to data corruption if multiple threads modify the array concurrently.
  - Operations like get(), add(), and remove() directly access or modify the underlying array without any built-in synchronization.
  - If one thread is iterating through an ArrayList while another thread modifies it, a ConcurrentModificationException is likely to be thrown, as the iterator detects that the collection has been modified during iteration.

→ **ArrayList achieving Concurrency With Ensuring Thread safety:**

- Not thread-safe by default. When multiple threads access and modify an ArrayList concurrently, it can lead to data corruption or inconsistent states. Imagine multiple people trying to modify a document simultaneously without any coordination – the result would be chaos.
- Solutions for thread safety with ArrayList typically involve external synchronization, such as using Collections.synchronizedList() which essentially locks the entire list during any operation, severely limiting concurrency.
- ```
final List<Integer> arrayList = new ArrayList<>();
```

• **How CopyOnWriteArrayList Works (Internals):**

- CopyOnWriteArrayList maintains an immutable array that is only changed by creating and reassigning a new array instance.
- When a modification operation (add, remove, set) is performed, the entire array is copied to a new array with the modification applied, and the reference to the array is atomically updated to point to the new array.
- Read operations (like get(), size(), contains()) operate on the current array reference without any locking, providing non-blocking reads.



Benefits of Copy-on-write Strategy

- ✓ Read Operation proceed without blocking
- ✓ No ConcurrentModificationException during iteration
- ✓ Thread-safe without external synchronization

```

■ final CopyOnWriteArrayList<Integer> cowList = new
  CopyOnWriteArrayList<>();

```

3. ConcurrentLinkedQueue:

- ConcurrentLinkedQueue is a thread-safe implementation of a queue (a FIFO data structure) that uses a linked-node structure and non-blocking algorithms to achieve high concurrency. It allows multiple threads to safely add and remove elements without excessive locking, making it ideal for high-throughput, multi-producer/multi-consumer scenarios.
- Both LinkedList (when used as a queue) and ConcurrentLinkedQueue in Java implement the Queue interface, providing first-in-first-out (FIFO) operations. However, they differ significantly in their handling of concurrent access, particularly in their implementation of thread safety, making them suitable for different scenarios.
- **How LinkedList (as a Queue) works (Internals):**
 - LinkedList internally uses a doubly-linked list structure, with each node containing references to both the previous and next nodes in the sequence.
 - When elements are added (offer) to the queue, they are appended to the tail of the list. When elements are removed (poll) from the queue, they are taken from the head of the list.
 - Operations like offer(), poll(), and peek() directly modify or access the underlying linked structure without any built-in synchronization.

- In multi-threaded environments, concurrent modifications to a LinkedList can lead to data corruption, inconsistent states, or ConcurrentModificationException during iteration.

- **How ConcurrentLinkedQueue Works (Internals):**

- ConcurrentLinkedQueue uses a singly-linked list structure, with each node containing a reference to the next node in the sequence.
- The implementation is based on non-blocking algorithms using atomic compare-and-swap (CAS) operations , which allow threads to make progress without explicit locks.
- When a thread wants to add an element (offer) , it attempts to set the next reference of the current tail node to the new node. If another thread has modified the queue in the meantime, the operation is retried.
- Similarly, when removing an element (poll) , the operation attempts to set the head reference to the next node. Again, if concurrent modifications have occurred, the operation is retried.

→ This "optimistic" approach allows high throughput in multi-threaded environments, as threads don't have to wait for locks to be released.

45.What is Compare and Swap (CAS)?

→ Compare and swap is a hardware-supported atomic operation that works as follows:

- Compare: The operation reads a value from a memory location and compares it with an expected value.
- Swap: If and only if the current value in memory equals the expected value, it writes a new value into that memory location.

- ```
AtomicReference<Node> ref = new AtomicReference<>(currentNode);
```
- ```
boolean isUpdated = ref.compareAndSet(currentNode, newNode);
```

44.Future and Completable Future in Java: Asynchronous Programming Essentials .

- In modern Java applications, handling asynchronous operations efficiently is crucial for creating responsive and scalable software. Two powerful tools for managing asynchronous tasks are the Future interface and its enhanced

implementation, `CompletableFuture`. These constructs allow developers to work with results that may not be immediately available, enabling non-blocking operations and improving application performance.

- **Asynchronous** → **Runs in background/parallel.**
- **synchronous** → **Runs One after the other.**
- **Future Interface: The Foundation of Asynchronous Results**
 - The Future interface, introduced in Java 5, represents the result of an asynchronous computation. It provides a way to check if the computation is complete, wait for its completion, and retrieve the result.
 - **Key Features of Future**
 - **Result Retrieval:** Allows access to the result of an asynchronous operation once it's available.

```
import java.util.concurrent.*;
public class FutureResultExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor =
            Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(() -> 10 +
            20); //submit() starts the task Execution asynchronously.
        Integer result = future.get(); //This blocks Main Thread
        until result is ready
        System.out.println("Result: " + result); // Result: 30
        executor.shutdown();}}
```

- **Status Checking:** Provides methods to check if a task has completed or been cancelled.

```
Future<Integer> future = executor.submit(() -> 42);
// ...some time later
if (future.isDone()) { //isDone() tells if the task is finished.
    System.out.println("Task completed!"); // Prints -> Task
    completed!
} else {
    System.out.println("Still working..."); // Prints ->
    Still working..
}
```

- **Cancellation:** Supports cancellation of tasks that haven't started or are in progress.

```
Future<?> future = executor.submit(() -> {
    while (true) {
        // long-running task
        if (Thread.currentThread().isInterrupted()) break;}});
boolean cancelled = future.cancel(true); // interrupt the
thread
System.out.println("Cancelled: " + cancelled);
```

- **Explanation :**
- ○ cancel(true) tries to interrupt if the task is running.
- ○ You should check Thread.interrupted() inside your task to make cancellation responsive because :
 - When you call future.cancel(true), it sends an interrupt signal to the thread running the task.
 - But Java doesn't forcefully stop a thread — it just sets the interrupted flag.
 - So your task must periodically check if it has been interrupted using Thread.interrupted().
 - If you don't check it, the task will keep running, even if you cancel it.
- **Blocking Operations:** Primarily uses blocking methods that wait for task completion.

```
Future<String> future = executor.submit(() -> {
    Thread.sleep(3000);
    return "Finished after delay";});
System.out.println("Waiting for result...");
String value = future.get(); // blocks for ~3 seconds
System.out.println(value);
```

- **Explanation :**
- ○ future.get() blocks until the task completes.
- ○ Use get(long timeout, TimeUnit unit) if you want to avoid indefinite blocking as :
 - future.get() blocks the current thread until the task completes — this could take forever if the task hangs.
 - future.get(timeout, unit) adds a maximum wait time.
 - If the result isn't ready in that time, it throws a TimeoutException, and your thread can recover gracefully instead of hanging forever.

45.What Is Basic Usage of Future

- In Java, a Future is a placeholder for the result of an asynchronous computation. It is commonly used in multithreading to handle tasks that take time to compute, allowing the main thread to continue execution while waiting for the result. A Future is typically obtained from an ExecutorService when submitting tasks.

46.What are Limitations of the Future in Java?

1. No Composition

- Future does not support chaining multiple tasks together Like **CompletableFuture**. You cannot specify a dependent task that should execute once the Future completes, making it difficult to manage sequential asynchronous computations.

```
→import java.util.concurrent.CompletableFuture;
public class CompletableFutureExample {
    public static void main(String[] args) {
        // Start an asynchronous task using
        CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000); // Simulate a delay
            } catch (InterruptedException e) {}
            Thread.currentThread().interrupt();
        });
        return "Result from CompletableFuture";
    }
}
```

```

    })
    // Register a callback that processes the result once
    it's ready.
    .thenAccept(result -> {
        System.out.println("CompletableFuture result: " +
            result);
        // Additional operation after the result is available.
        System.out.println("Processing after CompletableFuture
            result");});
    // Optionally do other work here while the asynchronous
    task is running.
    System.out.println("Main thread is free to do other tasks
        while waiting...");
    // To prevent the main thread from exiting immediately,
    // we'll wait for the CompletableFuture to complete.
    try {
        Thread.sleep(2000); // Wait enough time for the async
        task to finish.
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

```

→ Here In **CompletableFuture** we can use chaining , means when one task needs output of another task we can use thenAccept method .When we use this method we don't need to wait for task 1 result by using get which block main thread .So now in **CompletableFuture** we need to wait only for task 2 results.where in Future we need to wait(block) for both task 1 and 2 . so **CompletableFuture** ensure **Semi Blocking**.

2. No Exception Handling

- There is no built-in mechanism to handle exceptions in Future. If an exception occurs during execution, you must manually catch it using get(), which can make error handling cumbersome.

3. Blocking Operations

- Calling get() on a Future blocks the current thread until the result is available, leading to potential performance issues if the task takes too long to complete.

4. No Completion Notification

- Future does not provide an event-driven mechanism to notify when a task completes. You must explicitly poll or call get(), which is inefficient compared to reactive approaches.

47.What is CompletableFuture: Advanced Asynchronous Programming

- `CompletableFuture`, introduced in Java 8, extends the `Future` interface and implements the `CompletionStage` interface. It addresses the limitations of `Future` and provides a rich set of methods for composing, combining, and handling asynchronous computations.

48.Key Features of `CompletableFuture`

1. Non-blocking Operations: Supports non-blocking, asynchronous programming.
2. Composition: Allows chaining of multiple asynchronous operations.
3. Combination: Provides methods to combine results from multiple futures.
4. Exception Handling: Robust exception handling mechanisms.
5. Completion Callbacks: Supports callbacks when tasks complete.
6. Explicit Completion: Can be completed explicitly, useful for complex scenarios.

49.What Are Common `CompletableFuture` Methods Explained?

1. `get()` – Wait and Retrieve the Result (Throws Checked Exception) :
2. `join()` – Wait and Retrieve the Result (Throws Unchecked Exception) - Not Recommended (Only when you know Exceptions won't come):
3. `complete(value)` – Manually Complete a Future :
4. `isDone()` – Checks if the Future is Completed :

50.Explain **final** Keyword in Java OOP?

- The **final** keyword is used to restrict modification. It can be applied to:
 - Variables
 - Methods
 - Classes
- 1. **final** Variable
 - A final variable's value cannot be changed once assigned.

```
java
final int MAX_SPEED = 120;
MAX_SPEED = 130; // ❌ Error: cannot assign a value to final variable
```

For objects, the reference cannot change, but the object's internal state *can*:

```
java
final Car car = new Car();
car.color = "Red"; // ✅ allowed
car = new Car(); // ❌ Error: cannot reassign final reference
```

- **2. final Method**

- **A final method cannot be overridden by subclasses**

```
java                                                                    Copy Edit

class Vehicle {
    final void startEngine() {
        System.out.println("Engine started");
    }
}

class Car extends Vehicle {
    // void startEngine() {} // ❌ Error: Cannot override final method
}
```

- **3. final Class**

- **A final class cannot be extended (no subclassing allowed).**

```
java                                                                    Copy Edit

final class Animal {
    void sound() {
        System.out.println("Generic sound");
    }
}

// class Dog extends Animal {} // ❌ Error: Cannot inherit from final class
```

