# Multithreading in Java

**CPU :** Often referred as brain of the computer, is responsible for executong the instructions from the program. Ti perform the basic arithmetic logic , control and i/o operations specified by the instruction.

**CORE:** A core is an individual processing unit within a CPU. Modern CPUs can have multiple cores, allowing them to perform multiple tasks simultaneously.

**Program:** set of instructions written in a programming language that tells the computer how to perform the specifiec task.

**Process:** is an instance of a program that is being executed. When a program runs, the OS creates a process to manage its execution.

**Thread:** A thread is the smallest unit of execution within the process. A process can have multiple threads, which share the same resources but can run independently.
**ex:** A web browser like google chrome might uses multiple threads for the different tabs, with each tab running as a separate thread.

**MultiTasking:** Multitasking allows an OS to run multiple processes simultaneously. On single threaded CPUs, this is done through time sharing, rapidaly switching between the tasks. On multicore CPUs, true parallel execution occurs, with tasks distributed across cores. The OS scheduler balances the load, ensuring efficient and responsive system performance.
Ex: browsing on internet while litening the music and downloading the file.

Multitasking utilizes the capabilities of CPU and its cores. When an operating system performs multitasking, it can assign different tasks to diffrenet cores. This is more efficient tah assinging all the tasks to a single core.

**Multithreading:** refers to the ability to execute multiple threads within a single process concurrently. A web browser can use multitreading by having separate threads for rendering the page, running JS, and managing user input. This makes the browser more responsive and efficient.

Multithreading enhances the efficiency of multitasking by breaking down individual tasks in to smaller sub-tasks or threads. These threads can be processed simultaneously, making better use of CPUs capabilities.

In **Single core system:** Both threads and processes are managed by the OS scheduler through time slicing and context switching to create the illusion of simulyaneous execution.

In **Multi core system:** Bothe threads and process can run in true parallel on a different cores, with the OS scheduler distributing the tasks across the cores to optimize performance.

**Time Slicing:**

**Def**: Time slicing divides the CPU time into small intervals called time slices or quanta.

**Function:** The OS scheduler allocates these time slices to different processes and threads, ensuring each gets a fair share of CPU time.

**Purpose:** This prevents any single process or thread from monopolizing the CPU, improving responsiveness ans enabling concurrent execution.

**Context Switching:**

**Def:** It is the process of saving the state of a currently running process or thread and loading the state of the next one to be executed.

**Function:** when the process or thread's time slice expires, the OS scheduler performs the context switch to move the CPUs focus to the another process or thread.

**Purpose:** This allows multiple processes and thread to share CPU, giving the appearance of simultaneous execution on a single-core CPU or improving parrallelism on multi-core CPUs.

**Diff:**

Multithreading can be achieved through multithreading where each task is divided into threads that are managed concurrently.

While

Multitasking typically refers to the running of multiple applications, multithreading is more granular, dealing with multiple threads within the same application or process.

Multitasking operates at the level of processes , which are the operating system's primary units of execution.

Multithreading operates at the level of threads, which are smaller units within a process.

Multitasking involves manageing resources between completely separate programs, which may have independently memory spaces and system resources.

Multithreading involes managing resources within a single program, where threads share the same memory and resources.

MultiTasking allows us to run multiple applications simultaneously, improving the productivity and system utilization.

MultiThreading allows a single application to perform multiple tasks at a same time, improving application performance and responsivesness.

**Ex:** The office manager (Operating system) assigns different employees(process) to work on diff projects(applications) simultaneously. Each employee works on a different project independently. Within a single project (application), a team(processes) of employees(threads) works on the different parts of the project at the same time, collaborationg and sharing the resources.

**Java** provides the robust support for multithreading, allowing developers to create applications that can perforrm multiple tasks simultaneously, improving performance and responsiveness.

In Java, multithreading is the cincurrent execution of two or more threads to maximize the utilization of the CPU.

Java's multithreading capabilities are part of the **java.lang** package**,** making it easy to implement concurrent execution.

In a single-core environment, Java's multithreading is managed by the JVM and the OS, which switch between threads to give the illusion of concurrency.

The threads share the single core, and time-slicing is used to manage thread execution.

In a multi-core environment, Java's multithreading can take full advantage of the available cores.

The JVM can distribute threads across multiple cores, allowing true parallel execution of threads.

A thread is a lightweight process, the smallest unit of processing. Java supports multithreading through its **java.lang.Thread** class and the **java.lang.Runnable** interface.

When a Java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program.

**Code:**

```
public class Test {
   public static void main(String[] args) {
      System.out.println("Hello world !");

      System.out.println(Thread.currentThread().getName());
   }
}
```

**Output:**

```
Hello world

Main
```

To create a new thread in Java, you can either extend the Thread class or implement the Runnable interface.

**Method 1: Extend the Thread class**

1. A new class World is created that extends Thread.
2. The run() method is overridden to define the code that constitutes the new thread.
3. start() method is called in main() method to initiate the new thread.

**1. Creation of new thread that is to be executed by extending the Thread class:**

```
public class World extends Thread {
   @Override
   public void run() {
      for (; ; ) {
         System.out.println("World");
      }
   }
}
```

**2.Passing that thread to the main() method and an independent thread is created to execute the world() class.**

```
public class Test {
   public static void main(String[] args) {
      World world = new World();
      world.start();
      for (; ; ) {
            System.out.println("Hello");
        }
    }
}
```

**Method 2: Implement Runnable interface**

1. A new class World is created that implements Runnable.
2. The run method is overridden to define the code that constitutes the new thread.
3. A Thread object is created by passing an instance of World.
4. start method is called on the Thread object to initiate the new thread.

**Creating a new class "world" that implements Runnable interface.**

```
public class World implements Runnable {
    @Override
    public void run() {
        for (; ; ) {
            System.out.println("World");
        }
    }
}
```

**Creating an instance of "World" and an instance of "Thread" by passing the instance if "World" to the constructor of "Thread", then start() the execution of thread.**

```
public class Test {
    public static void main(String[] args) {
        World world = new World();
        Thread thread = new Thread(world);
        thread.start();
        for (; ; ) {
            System.out.println("Hello");
        }
```

```
  }
}
```

To create a new thread in java, you can either extend the “Thread” or implement the “Runnable” interface.

In both cases, the run method contains the code that will be executed in the new thread.

**Thread Lifecycle**

The lifecycle of a thread in Java consists of several states, which a thread can move through during its execution.

- **New:** A thread is in this state when it is created but not yet started.
- **Runnable:** After the start method is called, the thread becomes runnable. It’s ready to run and is waiting for CPU time.
- **Running:** The thread is in this state when it is executing.
- **Blocked/Waiting:** A thread is in this state when it is waiting for a resource or for another thread to perform an action.
- **Terminated:** A thread is in this state when it has finished executing.

```
public class MyThread extends Thread{
  @Override
  public void run() {
    System.out.println("RUNNING"); // RUNNING
    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      System.out.println(e);
    }
  }

  public static void main(String[] args) throws InterruptedException {
    MyThread t1 = new MyThread();
    System.out.println(t1.getState()); // NEW
    t1.start();
    System.out.println(t1.getState()); // RUNNABLE
    Thread.sleep(100);
```

```
        System.out.println(t1.getState()); // TIMED_WAITING
        t1.join();
        System.out.println(t1.getState()); // TERMINATED
    }
}
```

**Runnable vs Thread**

In Java, both Runnable and Thread are used for creating and executing threads, but they serve different purposes and offer distinct advantages.

**Runnable:** Using the Runnable interface allows you to separate the task (the code that needs to be executed) from the thread itself. This approach is beneficial when you want your class to extend another class since Java does not support multiple inheritance. Implementing Runnable gives you flexibility, as you can pass a Runnable instance to a Thread object and execute it. This is particularly useful for implementing a task that can be reused across multiple threads.

**Thread:** Extending the Thread class directly is an option when you need to override its methods, such as run() or start(). This method is straightforward but limits your class to extend only Thread, as Java allows single inheritance. Use this approach when the task inherently requires direct control over the thread, such as managing thread-specific operations or overriding lifecycle methods.

In summary, use Runnable for better design flexibility and to separate task logic from thread management, while Thread is suitable for direct control over thread behavior when necessary.

**Methods Of Thread**

**1. start()**

**Definition**: Starts the execution of a thread. The Java Virtual Machine calls the run() method of this thread.

Begins the execution of the thread. The Java Virtual Machine (JVM) calls the run() method of the thread.

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

public class ThreadStartExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread and calls run() method
    }
}
```

**2. run()**

**Definition**: The entry point for the thread's execution. Contains the code that will be executed by the thread.

The entry point for the thread. When the thread is started, the run() method is invoked. If the thread was created using a class that implements Runnable, the run() method will execute the run() method of that Runnable object.

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

public class ThreadRunExample {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start(); // Executes the run() method
    }
}
```

**3. sleep(long millis)**

**Definition**: Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

```
class SleepExample extends Thread {
   public void run() {
      try {
         System.out.println("Sleeping for 2 seconds...");
         Thread.sleep(2000); // Sleep for 2 seconds
         System.out.println("Awoke!");
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}

public class ThreadSleepExample {
   public static void main(String[] args) {
      SleepExample thread = new SleepExample();
      thread.start();
   }
}
```

**4. join()**

**Definition**: Waits for this thread to die. The current thread will pause until the thread it joins has completed its execution.

Waits for this thread to die. When one thread calls the join() method of another thread, it pauses the execution of the current thread until the thread being joined has completed its execution.

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


public class ThreadJoinExample {
    public static void main(String[] args) throws InterruptedException {
        JoinExample thread = new JoinExample();
        thread.start();
        thread.join(); // Waits for the thread to finish
        System.out.println("Thread has finished execution.");
    }
}
```

**5. setPriority(int newPriority)**

**Definition**: Changes the priority of the thread. Priority is a value between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY (10).

```
class PriorityExample extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " with priority " +
Thread.currentThread().getPriority());
    }
}


public class ThreadPriorityExample {
    public static void main(String[] args) {
        PriorityExample thread1 = new PriorityExample();
        PriorityExample thread2 = new PriorityExample();
        thread1.setPriority(Thread.MAX_PRIORITY); // Set to max priority
        thread2.setPriority(Thread.MIN_PRIORITY); // Set to min priority
        thread1.start();
        thread2.start();
    }
}
```

**6. interrupt()**

**Definition**: Interrupts a thread that is in a blocked state (like sleeping or waiting). It sets the thread’s interrupt status.

```
class InterruptExample extends Thread {
    public void run() {
        try {
            Thread.sleep(5000); // Sleep for 5 seconds
        } catch (InterruptedException e) {
            System.out.println("Thread was interrupted!");
        }
    }
}

public class ThreadInterruptExample {
    public static void main(String[] args) throws InterruptedException {
        InterruptExample thread = new InterruptExample();
        thread.start();
        Thread.sleep(1000); // Main thread sleeps for 1 second
        thread.interrupt(); // Interrupt the thread
    }
}
```

**7. isAlive()**

**Definition**: Tests if this thread is alive. Returns true if the thread has been started and has not yet died.

```
class AliveExample extends Thread {
    public void run() {
        try {
            Thread.sleep(1000); // Sleep for 1 second
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class ThreadAliveExample {
    public static void main(String[] args) throws InterruptedException {
        AliveExample thread = new AliveExample();
        thread.start();
        System.out.println("Thread is alive: " + thread.isAlive()); // Should print true
        thread.join(); // Wait for the thread to finish
        System.out.println("Thread is alive: " + thread.isAlive()); // Should print false
    }
}
```

**8. yield()**

**Definition**: Suggests that the currently executing thread should yield its execution time to allow other threads of the same priority to execute.

Thread.yield() is a static method that suggests the current thread temporarily pause its execution to allow other threads of the same or higher priority to execute. It’s important to note that yield() is just a hint to the thread scheduler, and the actual behavior may vary depending on the JVM and OS.

```
class YieldExample extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is yielding.");
        Thread.yield(); // Yield to other threads
        System.out.println(Thread.currentThread().getName() + " is resuming.");
    }
}

public class ThreadYieldExample {
    public static void main(String[] args) {
        YieldExample thread1 = new YieldExample();
        YieldExample thread2 = new YieldExample();
        thread1.start();
        thread2.start();
    }
}
```

**9. setName(String name) / getName()**

**Definition**: Sets or retrieves the name of the thread, which can help with debugging.

```
class NameExample extends Thread {
    public void run() {
        System.out.println("Thread name: " + getName());
    }
}

public class ThreadNameExample {
    public static void main(String[] args) {
        NameExample thread = new NameExample();
        thread.setName("MyCustomThread");
        thread.start(); // Displays the custom thread name
    }
}
```

**10. getPriority()**

**Definition**: Retrieves the priority of the thread.

```
class PriorityGetExample extends Thread {
    public void run() {
        System.out.println("Thread priority: " + getPriority()); // Displays the thread's priority
    }
}


public class ThreadGetPriorityExample {
    public static void main(String[] args) {
        PriorityGetExample thread = new PriorityGetExample();
        thread.setPriority(7); // Set custom priority
        thread.start();
    }
}
```

**11. wait(), notify(), notifyAll()**

**Definition**: Used for inter-thread communication. wait() makes the thread wait until notified, while notify() wakes one waiting thread and notifyAll() wakes all waiting threads.

```
class WaitNotifyExample {
    private final Object lock = new Object();


    public void waitMethod() {
        synchronized (lock) {
            try {
                System.out.println("Waiting...");
                lock.wait(); // Wait for notification
                System.out.println("Notified!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void notifyMethod() {
        synchronized (lock) {
            System.out.println("Notifying...");
            lock.notify(); // Notify waiting thread
        }
    }
}

public class ThreadWaitNotifyDemo {
    public static void main(String[] args) {
        WaitNotifyExample example = new WaitNotifyExample();

        Thread waitingThread = new Thread(example::waitMethod);
        waitingThread.start();

        try {
            Thread.sleep(1000); // Ensure the waiting thread is waiting
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        new Thread(example::notifyMethod).start(); // Notify the waiting thread
    }
}
```

**12. sleep(long millis, int nanos)**

**Definition**: A more precise way to pause a thread for a specified duration (milliseconds and nanoseconds).

```
class SleepPreciseExample extends Thread {
    public void run() {
        try {
            System.out.println("Sleeping for 1.5 seconds...");
            Thread.sleep(1500, 500000); // Sleep for 1.5 seconds
            System.out.println("Awoke!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class ThreadSleepPreciseExample {
    public static void main(String[] args) {
        SleepPreciseExample thread = new SleepPreciseExample();
        thread.start();
    }
}
```

These examples illustrate the functionality of each method in the context of threading in Java.

**13. Thread.setDaemon(boolean)**

**Definition**: This method marks the thread as either a daemon thread or a user thread. Daemon threads are those that do not prevent the JVM from exiting when the program finishes executing. If the only threads running are daemon threads, the JVM will exit.

**Example**

Here's a simple example demonstrating the use of setDaemon(boolean):

```
class MyDaemonThread extends Thread {
    public void run() {
        while (true) {
            System.out.println("Daemon thread is running...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Daemon thread interrupted.");
            }
        }
    }
}

public class DaemonThreadExample {
    public static void main(String[] args) {
        MyDaemonThread daemonThread = new MyDaemonThread();

        // Marking the thread as a daemon
        daemonThread.setDaemon(true);
        daemonThread.start();

        // Main thread sleeps for a short duration
        try {
            Thread.sleep(3000); // Main thread sleeps for 3 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread is finishing...");
        // When the main thread exits, the daemon thread will also terminate
    }
}
```

**Importance of setDaemon(boolean)**

1. **Resource Management**: Daemon threads are typically used for background tasks, such as garbage collection, monitoring, or handling resources. They can help manage resources without keeping the application alive unnecessarily.
2. **Application Exit**: Since daemon threads do not prevent the JVM from exiting, they are useful for tasks that should run only as long as the application runs. This helps avoid resource leaks or hanging processes.
3. **Lifecycle Control**: It allows developers to control the lifecycle of threads based on the needs of the application. Daemon threads can be useful for long-running background tasks that do not need to block the program from exiting.

**Summary**

Using Thread.setDaemon(boolean) provides a way to define thread behavior in relation to the JVM's lifecycle. Understanding how to properly use daemon threads is important for efficient application design and resource management.

## <u>Problems Before Synchronization</u>

Before synchronization was introduced in Java, multi-threaded applications faced several problems, primarily due to concurrent access to shared resources. Here are the main issues:

1. **Data Inconsistency**: Multiple threads modifying shared variables could lead to unpredictable values, as one thread might overwrite changes made by another.
2. **Race Conditions**: Threads competing to read and write shared data could result in race conditions, where the outcome depends on the unpredictable timing of thread execution.

3. **Corrupted State**: Without proper control, shared resources could end up in an invalid state, causing errors or crashes in the application.
4. **Difficulty in Debugging**: Bugs resulting from concurrent modifications were often hard to reproduce and debug, leading to unreliable software.

Synchronization mechanisms were introduced to address these issues by ensuring that only one thread could access critical sections of code at a time, maintaining data integrity and consistency.

**For Example:**

**Counter.java**

```
package MultiThreading.synchronization;

public class Counter {

    private int count = 0;

    public void increament(){
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

**MyThread.java (extending Thread class)**

```
package MultiThreading.synchronization;

public class MyThread extends Thread{

    private Counter counter;

    public MyThread(Counter counter){
        this.counter = counter;
    }

    @Override
    public void run(){
        for (int i=0; i<1000; i++){
            counter.increament();
        }
    }
}
```

**Main.java**

```
package MultiThreading.synchronization;


public class Main {
    public static void main(String[] args) {

        Counter counter = new Counter();

        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);
        t1.start();
        t2.start();

        try{
            t1.join();
            t2.join();
        }catch (Exception e){

        }

        System.out.println(counter.getCount()); // Expected: 2000, Actual will be random <=
2000
    }
}
```

Here in this program, The Output could be varied which results in <=2000 (**output<=2000)**. Because the **Main** method created 2 threads and both of them executing the **Counter** class concurrently.

Since the **increment()** method in the Counter class is not synchronized. This results in a race condition when both threads try to increment the count variable concurrently.

Without synchronization, one thread might read the value of **count** before the other thread has finished writing its incremented value. This can lead to both threads reading the same value at the same time, incrementing it at the same time, and writing it back at the same time, effectively losing one of the increments.

Therefore, the **Synchronization** comes into play.

## Synchronization

Synchronization in programming, particularly in Java, is a mechanism that ensures that multiple threads can safely access shared resources without causing data inconsistency or corruption. It allows only one thread to execute a particular section of code at a time, ensuring that shared data remains consistent and preventing issues like race conditions.

**Key Points:**

1. **Mutual Exclusion**: Synchronization provides mutual exclusion, meaning that only one thread can access the synchronized block or method at any given time.
2. **Critical Sections**: Parts of code that modify shared resources should be synchronized to protect them from concurrent access.
3. **Types of Synchronization**:
   - **Synchronized Methods**: Whole methods are synchronized, allowing only one thread to execute them on an object.

```
class Counter {
    private int count = 0;
    // Synchronized method
    public synchronized void increment() {
        count++;
    }
}
```

   - **Synchronized Blocks**: Specific blocks of code are synchronized, providing more granular control and potentially better performance.

```
class Counter {
    private int count = 0;
    public void increment() {
        // Synchronized block
        synchronized (this) {
            count++;
        }
    }
```

   -

4. **Java Constructs**: Java provides several constructs for synchronization, including the **synchronized** keyword and more advanced tools in the **java.util.concurrent** package.
5. **Visibility**: Synchronization ensures that changes made by one thread are visible to others, preventing stale data.
6. **Deadlock**: Improper use of synchronization can lead to deadlocks, where two or more threads are waiting indefinitely for each other to release resources.

By using synchronization, developers can create safe and reliable multi-threaded applications that avoid common pitfalls associated with concurrent programming.

By synchronizing the **increment()** method, you ensure that only one thread can execute this method at a time, which prevents the race condition. With this change, the output will consistently be 2000.

## Lock in Java

In Java, a **lock** is a synchronization mechanism that **allows threads to control access to shared resources**. Locks provide a way to **enforce mutual exclusion**, ensuring that only one thread **can access a particular resource or section of code at any given time**. This helps **prevent data inconsistencies** and **race conditions** in multi-threaded applications.

**The synchronized keyword in Java offers a basic level of thread safety, but it comes with several drawbacks:**

First, it locks the entire method or block of code, which can lead to performance bottlenecks when multiple threads compete for access. Additionally, it lacks a **try-lock** feature, meaning that threads can end up blocking indefinitely if they can't acquire the lock, which increases the likelihood of deadlocks. Furthermore, **synchronized** only allows for a single monitor per object, limiting its support for multiple condition variables and providing only basic **wait/notify** mechanisms.

On the other hand, **explicit locks** implemented through the **Lock** interface present in **java.util.concurrent.locks** package provide more granular control over synchronization. They offer a **try-lock** capability that enables threads to attempt to acquire a lock without getting stuck, thus helping to avoid blocking. Moreover, explicit locks support multiple condition variables, facilitating more sophisticated thread coordination. This flexibility makes them a more powerful choice for managing concurrency in complex applications.

## Problems arises before the Lock in Java

Understanding with example:

**BankAccount class.**

```
package MultiThreading.lock;

public class BankAccount {

    private int balance = 100;

```

```
  public synchronized void withdraw(int amt){
    System.out.println(Thread.currentThread().getName()+ " attempting to withdraw: " +
amt);
    if (balance >= amt){
      System.out.println(Thread.currentThread().getName()+ " proceeding with withdrawal:
" + amt);
      try{
        Thread.sleep(10000);
      } catch (InterruptedException e) {
      }
      balance -= amt;
      System.out.println(Thread.currentThread().getName()+ " completed with withdrawal
of amount: " + amt);
    }else{
      System.out.println(Thread.currentThread().getName()+ " insufficient balance " +
balance);
    }
    System.out.println("Remaining Balance: "+ balance );
  }
}
```

This class defined here is having a method **withdraw()** which is performing some actions and since this method is defined **synchronised** as well so that only one thread can access this method at a time.

**Main class**

```
package MultiThreading.lock;

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

    Runnable task = new Runnable() {
      @Override
      public void run() {
        sbi.withdraw(50);
      }
    };
    Thread t1 = new Thread(task, "Thread-1");
    Thread t2 = new Thread(task, "Thread-2");
    t1.start();
```

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

Using this **Main** class there 2 threads are created and wants to access the **withdraw()** method, but due to being **synchronized,** only one thread can access **withdraw()** method at a time.

**Problem arises when any of the thread starts executing the methods and took infinte time to execute it.**

**Then the problem of deadlock and starvation will arise and second thread will not get the chance for executing that method.**

---

---

**Extra to Understand:**

**Understanding this method:**

```
        Runnable task = new Runnable() {
            @Override
            public void run() {
                sbi.withdraw(50);
            }
        };
```

This code snippet creates an instance of an **anonymous inner class** that implements the Runnable interface in Java. Here's a breakdown of the code:

**Breakdown of the Code**

1. **Runnable Interface**:
    - The Runnable interface is a functional interface in Java that contains a single method, **run()**. It is designed to represent a task that can be executed by a thread.
2. **Anonymous Inner Class**:
    - The new Runnable() { ... } syntax defines an anonymous inner class that implements the Runnable interface without explicitly naming the class.
    - This is often used for quick, one-time tasks where creating a separate class is unnecessary.
3. **Override Method**:

    - Inside the curly braces { ... }, the run() method is overridden. This method contains the code that will be executed when the thread runs.
    - In this case, the **run() method calls sbi.withdraw(50);,** indicating that it attempts to withdraw 50 units (e.g., dollars) from an account represented by sbi.

4. **sbi**:
    - The sbi object is likely an instance of a class that has a withdraw(int amount) method, which presumably handles the logic for withdrawing money from a bank account or similar structure.

---

**The <u>Runnable interface</u> in Java is a functional interface, which means it contains only one abstract method.**

**<u>Here's what you need to know:</u>**

**Methods in Runnable**

1. **run():**
    - This is the single abstract method that must be implemented by any class that implements the Runnable interface.
    - It contains the code that will be executed when the thread is started.

**Summary**

- **Total Methods:** The Runnable interface has one method **(run()).**
- **Default Methods**: Since Java 8, interfaces can also have default methods, but Runnable itself does not define any default methods.

The simplicity of the Runnable interface makes it straightforward for creating tasks that can be executed by threads.

## <u>Functional Interface:</u>

A functional interface in Java is an interface that has exactly one abstract method. This allows the interface to be implemented using a lambda expression or method reference, which makes it easier to write concise and readable code, especially when dealing with functional programming concepts.

**Examples:**

- **Java Built-in Functional Interfaces**: Java provides several built-in functional interfaces in the java.util.function package, such as:
    - **Runnable (no arguments, returns void)**
    - **Callable<V> (returns a value)**
    - **Consumer<T> (takes an argument and returns no result)**
    - **Supplier<T> (provides a result)**
    - **Function<T, R> (takes an argument and returns a result)**
    - **Predicate<T> (takes an argument and returns a boolean)**

## Anonymouse Inner Class

An **anonymous inner class** in Java is a class defined without a name and is declared and instantiated in a single expression.

It is typically used to instantiate a class that may not need a separate, named class definition.

Anonymous inner classes are often used for implementing interfaces or extending classes in situations where a one-time use is sufficient.

**Key Features:**

1. **No Name**: As the name suggests, anonymous inner classes do not have a name. They are defined on-the-fly at the point of instantiation.
2. **Single Use**: They are often used when you need a class for a short time and don't want to formally define it elsewhere.
3. **Access to Outer Class Members**: They can access the members (including private members) of the enclosing class.
4. **Extends or Implements**: An anonymous inner class can either extend a superclass or implement an interface.

**Syntax:**

**The syntax generally looks like this:**

```
ClassName obj = new ClassName() {
   // body of the class
};
```

**Example:**

1. **Implementing an Interface**

```
Runnable task = new Runnable() {
   @Override
   public void run() {
      System.out.println("Running task");
   }
};
new Thread(task).start();
```

---

2. **Extending a Class**:

```
class Animal {
   void sound() {
      System.out.println("Animal makes a sound");
   }
}

Animal cat = new Animal() {
   @Override
   void sound() {
      System.out.println("Meow");
   }
};

cat.sound();  // Output: Meow
```

---

**Now, To solve these problems we will use explicit locking system**

We will use **Lock interface**

## Lock Interface

The **Lock interface** in Java is part of the **java.util.concurrent.locks** package and provides a more sophisticated and flexible locking mechanism than the traditional synchronized keyword. It is designed for managing concurrent access to shared resources in a multi-threaded environment.

### Key Features of the Lock Interface

1. **Explicit Locking**:
   - Unlike synchronized, where locks are implicitly managed by the Java runtime, the Lock interface requires you to explicitly acquire and release locks, giving you more control over the locking process.
2. **Multiple Lock Implementations**:
   - The Lock interface has various implementations, the most common being **ReentrantLock**, which allows the same thread to acquire the lock multiple times.
3. **Try-Lock**:
   - The Lock interface provides a tryLock() method, which allows a thread to attempt to acquire a lock without blocking. If the lock is not available, it can proceed with other tasks.

```
if (lock.tryLock()) {
        try {
        // Critical section
        } finally {
        lock.unlock();
        }
 } else {
```

```
            // Handle the case where the lock is not acquired
        }
```

4. **Lock Interruptibly**:
   - You can acquire a lock in a way that allows the thread to be interrupted while waiting for the lock, using the **lockInterruptibly()** method.
5. **Condition Variables**:
   - The Lock interface supports condition variables via the Condition interface, allowing threads to wait for certain conditions to be met before proceeding. This is more flexible than the single monitor available with synchronized.

```
Condition condition = lock.newCondition();
lock.lock();
try {
        while (!conditionMet) {
        condition.await(); // Wait for the condition
    }
            } finally {
            lock.unlock();
}
```

**Advantages of Using the Lock Interface**

- **Fine-Grained Control**: Provides more control over locking mechanisms compared to synchronized methods or blocks.
- **Reduced Blocking**: The ability to try-lock can help reduce blocking and improve responsiveness.
- **Avoid Deadlocks**: With careful design, it can help prevent deadlocks through proper management of locks.
- **Multiple Condition Variables**: Supports more complex thread coordination.

**Conclusion**

The Lock interface is a powerful tool for managing concurrency in Java. It provides greater flexibility and control over synchronization compared to traditional synchronized blocks and methods, making it suitable for complex applications that require precise thread management.

**Now writing the code of class BankAccount without using the synchronized keyword and implementing the Lock interface:**

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

public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        System.out.println(Thread.currentThread().getName() + " attempting to withdraw " +
amount);
        try {
            if (lock.tryLock(1000, TimeUnit.MILLISECONDS)) {
                if (balance >= amount) {
                    try {
                        System.out.println(Thread.currentThread().getName() + " proceeding with
withdrawal");
                        Thread.sleep(3000); // Simulate time taken to process the withdrawal
                        balance -= amount;
                        System.out.println(Thread.currentThread().getName() + " completed
withdrawal. Remaining balance: " + balance);
                    } catch (Exception e) {
                        Thread.currentThread().interrupt();
                    } finally {
                        lock.unlock();
                    }
                } else {
                    System.out.println(Thread.currentThread().getName() + " insufficient balance");
                }
            } else {
                System.out.println(Thread.currentThread().getName() + " could not acquire the
lock, will try later");
            }
        } catch (Exception e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

## Explanation of Code

This code defines a BankAccount class that uses Java's ReentrantLock to manage concurrent access to the account's balance, ensuring thread safety during withdrawals. Here's a breakdown of how it works:

**Key Components:**

1. **Balance**:

```
private int balance = 100;
```

   - This variable holds the account balance, starting at 100.

2. **Lock**:

```
private final Lock lock = new ReentrantLock();
```

   - A ReentrantLock is created to manage access to the withdraw method. This lock allows threads to acquire and release access to shared resources safely.

3. **Withdraw Method**:

```
public void withdraw(int amount) {
   // ...
}
```

   - This method attempts to withdraw a specified amount from the account.

**Method Logic:**

1. **Lock Acquisition**:

```
if (lock.tryLock(1000, TimeUnit.MILLISECONDS)) {
```

   - The tryLock method attempts to acquire the lock for up to 1000 milliseconds. If another thread has the lock, this thread will wait for up to that time. If the lock is not acquired within that time, it will proceed without blocking indefinitely.

2. **Balance Check**:

```
if (balance >= amount) {
```

   - If the lock is successfully acquired, the method checks if the balance is sufficient for the withdrawal.

3. **Withdrawal Process**:

```
Thread.sleep(3000);

balance -= amount;
```

   - If sufficient balance is available, the thread simulates the time taken to process the withdrawal (3 seconds). Then it deducts the amount from the balance.

4. **Unlocking**:

```
finally {
   lock.unlock();
}
```

   - The lock is released in the finally block, ensuring that it is released even if an exception occurs during the withdrawal process.

5. **Insufficient Balance**:

```
else {
System.out.println(Thread.currentThread().getName() + " insufficient balance");
}
```

   - If the balance is insufficient, a message is printed.

6. **Lock Not Acquired**:

```
else {
   System.out.println(Thread.currentThread().getName() + " could not acquire the lock, will try later");
}
```

   - If the thread cannot acquire the lock, a message is printed indicating that it will try again later.

**Error Handling:**

- The method includes basic error handling with try-catch blocks to manage potential exceptions and ensures that the thread's interrupted status is set if an exception occurs.

**Summary:**

This class provides a thread-safe way to manage withdrawals from a bank account using locks. The use of ReentrantLock allows for more flexible lock handling compared to synchronized blocks, such as the ability to try to acquire the lock with a timeout. This design helps prevent issues like deadlocks and ensures that concurrent threads can safely access and modify the account balance.

## Reentrant Lock

A Reentrant Lock in Java is a type of lock that allows a thread to acquire the same lock multiple times without causing a deadlock.

If a thread already holds the lock, it can re-enter the lock without being blocked. This is useful when a thread needs to repeatedly enter synchronized blocks or methods within the same execution flow.

The ReentrantLock class from the **java.util.concurrent.locks** package provides this functionality, offering more flexibility than the synchronized keyword, including **try-locking**, **timed locking**, and **multiple condition variables** for advanced thread coordination.

**Example:**

```
public class ReentrantExample {
    private final Lock lock = new ReentrantLock();

    public void outerMethod() {
        lock.lock();
        try {
            System.out.println("Outer method");
            innerMethod();
        } finally {
            lock.unlock();
        }
    }

    public void innerMethod() {
        lock.lock();
        try {
            System.out.println("Inner method");
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ReentrantExample example = new ReentrantExample();
        example.outerMethod();
```

```
  }
}
```

The **ReentrantLock()** lock will keep on counting that how many times the lock is aquired and how many times it released.

If the count is Zero then only the the second thread can enters otherwise it cannot enter.

## Methods of ReentrantLock

### lock()

- Acquires the lock, blocking the current thread until the lock is available. It would block the thread until the lock becomes available, potentially leading to situations where a thread waits indefinitely.
- If the lock is already held by another thread, the current thread will wait until it can acquire the lock.

```
public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        lock.lock(); // Acquire the lock
        try {
            if (balance >= amount) {
                System.out.println(Thread.currentThread().getName() +
" is withdrawing " + amount);
                balance -= amount;
                System.out.println(Thread.currentThread().getName() +
" completed withdrawal. Remaining balance: " + balance);
            } else {
                System.out.println(Thread.currentThread().getName() +
" insufficient balance.");
            }
        } finally {
            lock.unlock(); // Always unlock in the finally block
        }
    }
```

**tryLock()**

- Tries to acquire the lock without waiting. Returns true if the lock was acquired, false otherwise.
- This is non-blocking, meaning the thread will not wait if the lock is not available.

```
public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        // Try to acquire the lock without waiting
        if (lock.tryLock()) {
            try {
                if (balance >= amount) {

System.out.println(Thread.currentThread().getName() + " is 
withdrawing " + amount);
                    balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed 
withdrawal. Remaining balance: " + balance);
                } else {

System.out.println(Thread.currentThread().getName() + " insufficient 
balance.");
                }
            } finally {
                lock.unlock(); // Always unlock in the finally block
            }
        } else {
            System.out.println(Thread.currentThread().getName() + " 
could not acquire the lock, will try later.");
        }
    }
```

**tryLock(long timeout, TimeUnit unit)**

- Attempts to acquire the lock, but with a timeout. If the lock is not available, the thread waits for the specified time before giving up.
- It is used when you want to attempt to acquire the lock without waiting indefinitely. It allows the thread to proceed with other work if the lock isn't available within the specified time. This approach is useful to avoid deadlock scenarios and when you don't want a thread to block forever waiting for a lock.

- Returns true if the lock was acquired within the timeout, false otherwise.

```
public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        try {
            // Try to acquire the lock for up to 1 second
            if (lock.tryLock(1, TimeUnit.SECONDS)) {
                try {
                    if (balance >= amount) {

System.out.println(Thread.currentThread().getName() + " is
withdrawing " + amount);
                        balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed
withdrawal. Remaining balance: " + balance);
                    } else {

System.out.println(Thread.currentThread().getName() + " insufficient
balance.");
                    }
                } finally {
                    lock.unlock(); // Always unlock in the finally
block
                }
            } else {
                System.out.println(Thread.currentThread().getName() +
" could not acquire the lock, will try later.");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // Restore
interrupted status
        }
    }
```

-

**unlock()**

- Releases the lock held by the current thread.
- Must be called in a finally block to ensure that the lock is always released even if an exception occurs.

**lockInterruptibly()**

- Acquires the lock unless the current thread is interrupted. This is useful when you want to handle interruptions while acquiring a lock.
- The **lockInterruptibly()** method in Java's **ReentrantLock** class allows a thread to acquire a lock, but it can be interrupted if the thread is waiting to acquire the lock. This is particularly useful in scenarios where you want to allow a thread to be interrupted (for example, when the thread is waiting on a lock for a long time).

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

```java
public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) throws InterruptedException {
        // Try to acquire the lock interruptibly
        lock.lockInterruptibly();
        try {
            if (balance >= amount) {
                System.out.println(Thread.currentThread().getName() +
" is withdrawing " + amount);
                Thread.sleep(2000); // Simulate time taken for
withdrawal
                balance -= amount;
                System.out.println(Thread.currentThread().getName() +
" completed withdrawal. Remaining balance: " + balance);
            } else {
                System.out.println(Thread.currentThread().getName() +
" insufficient balance.");
            }
        } finally {
            lock.unlock(); // Always unlock in the finally block
        }
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread thread1 = new Thread(() -> {
            try {
                account.withdraw(50);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() +
" was interrupted.");
            }
        }, "Thread-1");

        Thread thread2 = new Thread(() -> {
            try {
                account.withdraw(75);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() +
" was interrupted.");
            }
        }, "Thread-2");

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

        // Interrupt thread1 after a short delay
        try {
            Thread.sleep(500); // Wait for a moment before
interrupting
            thread1.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

__Explanation:__

- **Lock Initialization**:
- A ReentrantLock is created to manage concurrent access.
- **Withdraw Method**:
- The withdraw(int amount) method attempts to acquire the lock using **lock.lockInterruptibly().** This allows the method to be interrupted while waiting for the lock.
- If the balance is sufficient, it simulates a withdrawal process using **Thread.sleep(2000).**
- If the balance is insufficient, it prints an appropriate message.
- The **unlock()** method is called in a finally block to ensure the lock is released.
- **Main Method**:
- Two threads are created, each attempting to withdraw money from the same BankAccount instance.
- The first thread (thread1) is interrupted after a short delay (500 milliseconds) using **thread1.interrupt()**. This demonstrates that the thread can be interrupted while waiting for the lock.
- **Exception Handling**:
- Each thread catches InterruptedException when calling **withdraw(),** allowing it to handle the interruption gracefully.
- **Key Points:**
- **Interruptible Lock Acquisition**: The **lockInterruptibly()** method allows threads to respond to interruptions while waiting for the lock, making it useful in situations where you need to stop a long wait.
- **Graceful Handling**: By catching **InterruptedException**, you can handle the interruption appropriately (e.g., logging, cleanup).
- **Finally Block**: Always ensure that **unlock()** is called in a finally block to avoid leaving the lock in a locked state if an exception occurs.

## Extra Knowledge:

**What is Interruption?**

**Interruption** in programming refers to the ability to stop a running thread from what it's currently doing, allowing it to respond to a signal that indicates it should stop or change its course of action.

**Key Points:**

1. **Request to Stop**: When you interrupt a thread, you're essentially sending it a request to stop its current work. This doesn't forcibly terminate it but gives it a chance to finish up or handle the interruption.
2. **Use Case**: For example, if a thread is waiting for a lock or sleeping, you can interrupt it, and it will stop waiting or sleeping and handle the interruption, often by throwing an InterruptedException.
3. **Graceful Handling**: A well-designed program will check for interruptions regularly and can safely clean up resources or finish tasks before stopping.

**Example in Real Life:**

Think of a thread as a person working on a task. If you tap them on the shoulder (interrupt), they might look up and decide to stop what they're doing, check in with you, or finish up their task before moving on.

In programming, interruptions help manage how threads operate, especially in responsive applications where you might need to cancel ongoing tasks or processes based on user actions or other events.

### Thread.currentThread().interrupt()

When an interruption occurs within the try block (for example, when a thread is waiting, sleeping, or blocked), it typically throws an InterruptedException. When this exception is caught, the thread's interrupted status is automatically cleared (set to false).

**Purpose of Thread.currentThread().interrupt();**

1. **Restores Interrupted Status**: By calling Thread.currentThread().interrupt(); in the catch block, you are effectively restoring that interrupted status back to true. This is important because:
    - It allows any higher-level code or loops that check the interrupt status to recognize that the thread was interrupted and handle it appropriately.

2. **Graceful Handling**: Keeping track of the interrupt status is essential in a multi-threaded environment. If the thread was supposed to stop or take some action due to an interruption, restoring the status enables that behavior.

**Summary**

So yes, you're right! The line serves to "keep a record" of the interruption by setting the thread's status back to interrupted, allowing the thread or other parts of the program to react properly to the interruption later on.

---

## Fairness of Locking:

**Fairness of Locking** refers to a policy that determines how threads acquire locks when competing for the same resource. In Java, the ReentrantLock class allows you to specify whether the lock should be fair or unfair.

**Fair vs. Unfair Locking**

1. **Fair Locking**:
   - When a ReentrantLock is set to fair, it grants access to the longest-waiting thread when a lock becomes available.
   - This helps prevent thread starvation, where some threads may never get a chance to acquire the lock.

**Example**:

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

import java.util.concurrent.locks.ReentrantLock;


public class FairLockExample {

    private static final Lock fairLock = new ReentrantLock(true); // Fair lock


    public void criticalSection(int threadId) {

        fairLock.lock(); // Acquire the lock

        try {
            System.out.println("Thread " + threadId + " is in the critical section");
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            fairLock.unlock(); // Release the lock
        }
    }

    public static void main(String[] args) {
        FairLockExample example = new FairLockExample();

        Runnable task = () -> {
            int threadId = Integer.parseInt(Thread.currentThread().getName());
            example.criticalSection(threadId);
        };

        Thread t1 = new Thread(task, "1");
        Thread t2 = new Thread(task, "2");
        Thread t3 = new Thread(task, "3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

2. **Unfair Locking**:

- When a ReentrantLock is created without fairness or with fairness set to false, the lock does not guarantee any particular order of access. Threads can acquire the lock in any order, potentially leading to higher throughput but also the risk of thread starvation.

**Example**:

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

public class UnfairLockExample {
  private static final Lock unfairLock = new ReentrantLock(); // Unfair lock

  public void criticalSection(int threadId) {
    unfairLock.lock(); // Acquire the lock
    try {
      System.out.println("Thread " + threadId + " is in the critical section");
      Thread.sleep(1000); // Simulate work
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    } finally {
      unfairLock.unlock(); // Release the lock
    }
  }

  public static void main(String[] args) {
    UnfairLockExample example = new UnfairLockExample();

    Runnable task = () -> {
      int threadId = Integer.parseInt(Thread.currentThread().getName());
      example.criticalSection(threadId);
    };
```

```
        Thread t1 = new Thread(task, "1");
        Thread t2 = new Thread(task, "2");
        Thread t3 = new Thread(task, "3");


        t1.start();
        t2.start();
        t3.start();
    }
}
```

**Summary of the Examples**

- In the **fair lock example**, threads will acquire the lock in the order they requested it. This means if Thread 1 starts first, it will be granted access first, followed by Thread 2, and so on.
- In the **unfair lock example**, there's no guarantee of the order in which threads will acquire the lock. A newly arriving thread could acquire the lock before a thread that has been waiting longer.

**Considerations**

- **Fair Locks**: Ensure that threads are treated fairly and reduces the risk of starvation, but might lead to reduced throughput due to context switching and waiting.
- **Unfair Locks**: Can be more efficient in certain situations but may lead to some threads being starved if the lock is frequently contended.

Choosing between fair and unfair locking depends on the specific requirements of your application and the behavior you want to enforce.

## Read-Write Locking

**Read/Write Locking** is a concurrency control mechanism that allows multiple threads to read shared data simultaneously while restricting write access to ensure data integrity. It differentiates between read and write operations, allowing for more efficient access patterns in scenarios where reads are more frequent than writes.

**Key Concepts**

1. **Read Lock**:
   - Allows multiple threads to read the shared resource concurrently.
   - When a thread acquires a read lock, other threads can also acquire read locks but cannot acquire a write lock until all read locks are released.
2. **Write Lock**:
   - Allows only one thread to write to the shared resource at a time.
   - When a thread acquires a write lock, no other thread can acquire either a read or a write lock until the write lock is released.

**Benefits**

- **Increased Concurrency**: By allowing multiple readers, read/write locks can significantly improve performance in read-heavy applications.
- **Data Integrity**: Write locks ensure that data is not modified while being read, preventing inconsistencies.

**Example in Java**

Java provides the ReentrantReadWriteLock class in the java.util.concurrent.locks package, which implements read/write locking. Here's a simple example:

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteCounter {
    private int count = 0;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    public void increment() {
        writeLock.lock();
        try {
            count++;
            Thread.sleep(50);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            writeLock.unlock();
        }
    }

    public int getCount() {
        readLock.lock();
        try {
            return count;
        } finally {
            readLock.unlock();
        }
    }
    public static void main(String[] args) throws
InterruptedException {
        ReadWriteCounter counter = new ReadWriteCounter();

        Runnable readTask = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().getName() + " read: " +
counter.getCount());
                }
            }
        };

        Runnable writeTask = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    counter.increment();

System.out.println(Thread.currentThread().getName() + "
incremented");
                }
            }
        };

        Thread writerThread = new Thread(writeTask);
        Thread readerThread1 = new Thread(readTask);
        Thread readerThread2 = new Thread(readTask);

        writerThread.start();
        readerThread1.start();
        readerThread2.start();

        writerThread.join();
        readerThread1.join();
        readerThread2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```
2.

**Explanation of the Example**

1. **Lock Initialization**: A ReentrantReadWriteLock is created to manage access to sharedData.
2. **Reading Method**: The read() method acquires a read lock before accessing sharedData. Multiple threads can call this method simultaneously.
3. **Writing Method**: The write(int value) method acquires a write lock before modifying sharedData. Only one thread can execute this method at a time.

4. **Threads**: The example starts multiple reader threads and a single writer thread. The readers can access the data concurrently, while the writer will have exclusive access when modifying the data.

**Summary**

Read/write locking is an effective strategy for optimizing access to shared resources in concurrent programming. By allowing multiple readers and exclusive writers, it can improve performance and ensure data consistency in read-heavy applications.

## Deadlock Situation

A **deadlock** is a situation in multithreading where two or more threads are blocked forever, each waiting for the other to release a resource. In other words, a deadlock occurs when two or more threads cannot proceed because they are each waiting for the other to release a resource that they need to continue executing.

A deadlock occurs in concurrent programming when two or more threads are blocked forever, each waiting for the other to release a resource. This typically happens when threads hold locks on resources and request additional locks held by other threads.

**For example**, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1. Since neither thread can proceed, they remain stuck in a deadlock state.

Deadlocks can severely impact system performance and are challenging to debug and resolve in multi-threaded applications.

**Conditions for Deadlock**

For a deadlock to occur, the following four conditions must be true simultaneously:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode. If another thread requests that resource, it must be blocked until the resource is released.
2. **Hold and Wait**: A thread holding at least one resource is waiting to acquire additional resources that are currently being held by other threads.
3. **No Preemption**: Resources cannot be forcibly taken from a thread holding them; they must be voluntarily released by the thread holding them.
4. **Circular Wait**: There exists a set of threads {T1, T2, ..., Tn} such that T1 is waiting for a resource held by T2, T2 is waiting for a resource held by T3, and so on, with Tn waiting for a resource held by T1.

## Example of Deadlock in Java

Here's a simple example demonstrating a deadlock situation:

```java
class Pen {
    public synchronized void writeWithPenAndPaper(Paper paper) {
        System.out.println(Thread.currentThread().getName() + " is using pen " + this + " and trying to use paper " + paper);
        paper.finishWriting();
    }

    public synchronized void finishWriting() {
        System.out.println(Thread.currentThread().getName() + " finished using pen " + this);
    }
}

class Paper {
    public synchronized void writeWithPaperAndPen(Pen pen) {
        System.out.println(Thread.currentThread().getName() + " is using paper " + this + " and trying to use pen " + pen);
        pen.finishWriting();
    }

    public synchronized void finishWriting() {
        System.out.println(Thread.currentThread().getName() + " finished using paper " + this);
    }
}

class Task1 implements Runnable {
    private Pen pen;
    private Paper paper;

    public Task1(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }

    @Override
    public void run() {
        pen.writeWithPenAndPaper(paper); // thread1 locks pen and tries to lock paper
    }
}

class Task2 implements Runnable {
    private Pen pen;
    private Paper paper;

    public Task2(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }

    @Override
    public void run() {
            paper.writeWithPaperAndPen(pen); // thread2 locks paper and tries to lock pen
    }
}
```

```
public class DeadlockExample {
    public static void main(String[] args) {
        Pen pen = new Pen();
        Paper paper = new Paper();

        Thread thread1 = new Thread(new Task1(pen, paper), "Thread-
1");
        Thread thread2 = new Thread(new Task2(pen, paper), "Thread-
2");

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

Class Definitions

1. **Pen Class:**
   - Contains synchronized methods that represent actions involving a pen.
   - **writeWithPenAndPaper(Paper paper)**: This method attempts to use both the pen and the paper. It locks the Pen object and tries to call **paper.finishWriting().**
   - **finishWriting():** This method signifies that the writing action is complete.
2. **Paper Class:**
   - Similar to the Pen class, it has synchronized methods for actions involving paper.
   - **writeWithPaperAndPen(Pen pen):** This method locks the Paper object and tries to call **pen.finishWriting().**
   - **finishWriting():** Indicates that the writing action with paper is complete.

**Runnable Tasks**

3. Task1:
   - This class implements **Runnable** and is designed for Thread-1.
   - In its **run()** method, it calls **pen.writeWithPenAndPaper(paper)**, which tries to lock the **Pen** object first and then the **Paper** object.
4. Task2:
   - This class implements **Runnable** for Thread-2.
   - In its **run()** method, it synchronizes on the **pen** object first and then calls **paper.writeWithPaperAndPen(pen)**, which tries to lock the **Paper** object first and then the **Pen** object.

**Main Method**

5. **Main Method:**
    - Creates instances of **Pen** and **Paper**.
    - Starts two threads (Thread-1 and Thread-2) that execute Task1 and Task2, respectively.

**Deadlock Scenario**

Here's how a deadlock can occur with this setup:

1. Thread-1 executes **pen.writeWithPenAndPaper(paper),** acquiring the lock on the **Pen** object.
2. Thread-2 executes **synchronized (pen)** (acquiring the lock on pen) and then calls **paper.writeWithPaperAndPen(pen)**, which attempts to acquire the lock on the **Paper** object.
3. Thread-1 is now blocked, waiting to acquire the lock on **Paper** when it calls **paper.finishWriting().**
4. Thread-2 is also blocked, waiting to acquire the lock on **Pen** when it calls **pen.finishWriting()** inside **paper.writeWithPaperAndPen(pen).**

**Result**

Both threads are waiting indefinitely for each other to release their locks, leading to a deadlock situation.

**Summary**

- The code illustrates a deadlock condition caused by two threads trying to acquire locks on two resources (Pen and Paper) in different orders.
- To avoid such deadlocks, you could enforce a consistent locking order, use timeout mechanisms, or utilize higher-level abstractions like **java.util.concurrent constructs**.

**Now to avoid the Deadlock:**

```
package MultiThreading.O3_deadlock;


public class Task2 implements Runnable{

    Pen pen = new Pen();
```

```
        Paper paper = new Paper();

        public Task2(Pen pen, Paper paper){
          this.pen = pen;
          this.paper = paper;
        }

    @Override
    public void run() {
          synchronized (pen) {
            paper.writeWithPaperAndPen(pen); // thread2 locks paper and tries to lock
  pen
          }
    }
  }
```

By declaring the **Pen** as **synchronized,** The **synchronized (pen)** block means that **Task2** will acquire the lock on the **pen** object before executing the code inside the block.

And if the lock is already aquired by **Task1,** in this situation the Task2 will wait untill the lock on **Pen** is released by Task1 since **Pen** is **synchronized.**

## <u>Thread Communication:</u>

Thread communication refers to the methods and mechanisms that allow threads to communicate and coordinate their actions in a multithreaded environment. This is crucial in ensuring that threads can work together efficiently, share data, and synchronize their operations without running into issues like data inconsistency or race conditions.

**Wait and Notify Mechanism**:

- In Java, threads can communicate using the **wait(), notify(), and notifyAll()** methods:
  - **wait()**: A thread can call this method on an object to release the lock and wait until another thread invokes **notify()** or **notifyAll()** on the same object.
  - **notify()**: Wakes up a single thread that is waiting on the object's monitor (if there are any).
  - **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

**Producer-Consumer Problem**:

- A classic example of thread communication is the producer-consumer problem, where one thread (the producer) generates data and another thread (the consumer) processes that data. Proper communication mechanisms are needed to ensure the consumer waits for data to be available and the producer waits if storage is full.
- **SharedResource class**

```
package MultiThreading.04_ThreadCommunication;

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


    // Method to Produce the Data.
    public synchronized void produce(int value){

        while(available){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }

        }
        data = value;
        System.out.println("Produced " + value);
        available = true;
        notify(); // Notify consumers that data is available
    }


    // Method to Consume the data.
    public synchronized int consume(){
```

```java
        while (!available){
            try{
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        available = false;
        System.out.println("Consumed " + data);
        notify();
        return data;
    }
}
```

- **Producer Class**

```java
package MultiThreading.04_ThreadCommunication;

public class Producer implements Runnable{

    private SharedResource resource;

    public Producer(SharedResource resource){
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            resource.produce(i);
        }
    }
}
```

- **Consumer Class**

```java
package MultiThreading.04_ThreadCommunication;

public class Consumer implements Runnable {
    private SharedResource resource;

    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            int value = resource.consume();
        }
    }
}
```

- **Main Class**

```java
package MultiThreading.04_ThreadCommunication;

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

        Thread producerThread = new Thread(new Producer(resource));
        Thread consumerThread = new Thread(new Consumer(resource));
        producerThread.start();
        consumerThread.start();
    }
}
```

## Thread Safety

**Thread safety** refers to the property of a piece of code or a data structure that guarantees safe execution by multiple threads at the same time. In other words, a thread-safe component can be accessed by multiple threads without leading to data corruption or inconsistency.

**Key Concepts:**

1. **Mutual Exclusion**: Ensures that only one thread can access a resource or critical section at a time. This is often implemented using synchronization mechanisms like locks, semaphores, or monitors.
2. **Atomic Operations**: Operations that are completed in a single step from the perspective of other threads. For example, incrementing a variable might not be atomic if it involves reading, modifying, and writing back to memory.
3. **Visibility**: Changes made by one thread should be visible to others. This often requires using memory barriers or volatile variables to ensure that the latest value is read.
4. **Deadlocks**: A situation where two or more threads are blocked forever, each waiting for the other to release a resource. Designing thread-safe code involves avoiding conditions that can lead to deadlocks.
5. **Race Conditions**: Occur when two or more threads access shared data and try to change it simultaneously, leading to unpredictable results. Thread safety mechanisms help prevent race conditions.

**Common Practices for Ensuring Thread Safety:**

- **Synchronization**: Use synchronized blocks or methods to restrict access to critical sections.
- **Locks**: Use explicit locks (like ReentrantLock in Java) to manage access to shared resources.
- **Thread-safe Collections**: Use built-in thread-safe data structures (like ConcurrentHashMap in Java).
- **Immutable Objects**: Design objects that cannot be changed after they are created, making them inherently thread-safe.

**Example:**

In a multithreaded application, if multiple threads try to increment a shared counter, without proper synchronization, the counter may not reflect the correct value. Using synchronization ensures that only one thread can modify the counter at a time, maintaining its integrity.

In summary, thread safety is crucial for building robust and reliable multithreaded applications, ensuring that shared resources are accessed in a safe and predictable manner.

## Lambda Expression

Lambda expressions in Java, introduced in Java 8, provide a concise way to represent anonymous functions (or functional interfaces) using an expression. They are particularly useful for passing behavior as parameters, making code more readable and reducing boilerplate code.

**Syntax**

The basic syntax of a lambda expression is:

```
(parameters) -> expression
```

Or, if the body has multiple statements:

```
(parameters) -> { statements; }
```

**Components**

1. **Parameters**: The input parameters to the function. You can specify types or omit them (type inference).
2. **Arrow Token (->)**: Separates the parameters from the body.
3. **Body**: The implementation of the function.

**Example Usage**

Here are some common use cases for lambda expressions:

1. **With Functional Interfaces**: Java has many built-in functional interfaces in the java.util.function package, such as Consumer, Function, Predicate, and Supplier.

**1. Simple Example with a Functional Interface**

```
import java.util.function.Consumer;


public class LambdaExample {
    public static void main(String[] args) {
        Consumer<String> print = (message) -> System.out.println(message);
        print.accept("Hello, Lambda!");
    }
}
```

**2. Using Lambda with Collections**

You can use lambda expressions with collections, especially with streams.

```
import java.util.Arrays;
import java.util.List;


public class LambdaWithCollections {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");


        // Using lambda to print each name
        names.forEach(name -> System.out.println(name));
    }
}
```

**3. Using Lambda with Streams**

**Lambda expressions work seamlessly with the Stream API.**

```
import java.util.Arrays;
import java.util.List;


public class LambdaWithStreams {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        // Using lambda to filter and print even numbers
        numbers.stream()
               .filter(n -> n % 2 == 0)
               .forEach(n -> System.out.println(n));
    }
}
```

**Benefits of Lambda Expressions**

1. **Conciseness**: Reduces the amount of boilerplate code.
2. **Readability**: Makes code easier to understand by clearly expressing behavior.
3. **Improved Functionality**: Facilitates functional programming in Java.

**Limitations**

1. **Single Abstract Method**: Lambda expressions can only be used with functional interfaces (interfaces with a single abstract method).
2. **No Access to Non-final Variables**: You can only use effectively final variables from the enclosing scope.

**Conclusion**

Lambda expressions are a powerful feature in Java that enhances the language's expressiveness and support for functional programming. They simplify the way you work with collections and provide a clearer and more concise way to define behavior in your programs.

**Therefore, we can use this lambda expression for Runnable Interface as well because Runnable interface is functional Interface.**

```
1.  package MultiThreading.O5_lambdaExpression;

public class O5_lambdaExpression {

    public static void main(String[] args) {

        Runnable runnable = () -> {
            System.out.println("Hello World");
        };
        Thread thread1 = new Thread(runnable);
        thread1.start();
    }
}
```

Since the **Runnable Interface** is a **functional Interface** which contains only one method that is **run()** method. Therefore, we can use **Lambda expression** here to implement the Runnable Interface.

```
package MultiThreading.O5_lambdaExpression;

public class O2_lambdaExpression {

    public static void main(String[] args) {

        Thread t1 = new Thread(()->{
            System.out.println("Hello");
        });

        t1.start();
    }
}
```

1. Here, a new thread (t1) is created using a lambda expression.

2. new Thread(...) creates an instance of the Thread class.

3. The lambda expression () -> { System.out.println("Hello"); } implements the Runnable interface's run method.

   - The () indicates that the run method takes no parameters.
   - The body of the lambda expression contains a single statement that prints "Hello" to the console.

4. This syntax is a more concise way of creating an instance of a class that implements Runnable.

## Base To understand Thread Pooling:

**A basic program to calculate the factorial.**

```
package MultiThreading.O6_ThreadPooling.ThreadExecutorFramework;

public class NormalMethod {

    public static void main(String[] args) {

        long startTime = System.currentTimeMillis();
        for (int i = 1; i < 10; i++) {
            int result = fact(i);

            System.out.println(result);
        }

        System.out.println("Total Time: " + (System.currentTimeMillis() - startTime));
    }

    private static long fact(int n) {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        long result = 1;

        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

Here In this program, In each iteration of first for loop the fact() method is called and it takes 1 sec for thread sleep and the factorial is calculated and the value is returned.

**Now I want to create multiple threads for the calculation of this factorial program.**

```
package MultiThreading.O6_ThreadPooling.ThreadExecutorFramework;

public class NormalMethod {

    public static void main(String[] args) {

        long startTime = System.currentTimeMillis();
        for (int i = 1; i < 10; i++) {
            int finall = i;

            Thread t1 = new Thread(()->{
                long result = fact(finall);
                System.out.println(result);
            });
        }

        System.out.println("Total Time: " + (System.currentTimeMillis() - startTime));
    }

    private static long fact(int n) {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        long result = 1;

        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

Here In this case, In each iteration of the loop a new thread is created for the execution of **fact()** method.

These threads are responsible for the parrallel execution of the fact() method.

**Key Points on Parallel Execution in Your Scenario**

1. **Thread Creation**:
   - In each iteration of the loop, a new thread is created and started. However, just because the threads are started one after another in quick succession doesn't mean they run sequentially.
2. **Context of Execution**:
   - When t1.start() is called, the thread is placed in the "runnable" state, and the operating system's scheduler decides when to execute it. This means that even if the threads are created one by one, the OS can switch between threads.
3. **Operating System Scheduler**:
   - The operating system handles the scheduling of threads. Once a thread is started, it becomes eligible for execution, and the OS may run it immediately, depending on its scheduling algorithm.
   - If there are available CPU cores, the OS can execute multiple threads at the same time, even if they were created in sequence.
4. **Non-blocking Behavior**:
   - The start() method is non-blocking. This means that when you call t1.start(), the main program continues executing without waiting for that thread to finish. This allows the next iteration of the loop to create and start the next thread almost immediately.

**Example of Parallel Execution in Your Loop**

- **First Iteration**:
  - A thread for calculating the factorial of 1 is created and started.
- **Second Iteration**:
  - Before the first thread has finished executing, a new thread for calculating the factorial of 2 is created and started.
- **Execution**:
  - If your CPU has multiple cores (e.g., 4 cores), both threads may run simultaneously on different cores.
  - If your CPU has only one core, the OS will rapidly switch between the threads, giving the appearance of parallelism, although true parallel execution (simultaneous execution on multiple cores) won't occur.

**Summary**

To achieve true parallel execution in your scenario:

- You start multiple threads in quick succession.
- The OS schedules these threads for execution.
- As soon as a thread is eligible to run, it can be executed independently of when it was created.

Even if the creation of threads happens one after another, their execution can be parallel, provided the system resources allow it. The key is that once a thread starts, it runs concurrently with others, leveraging the operating system's capabilities to manage and execute multiple threads simultaneously.

**Here the Problem is:**

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

    long startTime = System.currentTimeMillis();
    for (int i = 1; i < 10; i++) {
       int finall = i;

       Thread t1 = new Thread(()->{
          long result = fact(finall);
          System.out.println(result);
       });
    }

    System.out.println("Total Time: " + (System.currentTimeMillis() - startTime));

}
```

**Problem: The main thread is responsible for the execution of this method, The main thread will not wait for other threads to be completed and and will execute this line instantly. So this will not the exact time of the execution.**

**The solution to this problem is:**

We will have to wait for all the others threads to be completed then we will start the **main thread** to be carried on.

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

    long startTime = System.currentTimeMillis();
    Thread[] threads = new Thread[9];
    for (int i = 1; i < 10; i++) {
        int finall = i;
        threads[i-1] = new Thread(
                ()->{
                    long result = fact(finall);
                    System.out.println(result);
                }
        );
        threads[i-1].start();
    }
    for(Thread thread: threads){
        try{
            thread.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    System.out.println("Total Time: " + (System.currentTimeMillis() - startTime));
}
```

Here we created a **array of threads of size 9** and we are waiting for all the threads to complete the execution using **join()** method then we will carry on the main thread to print the total time taken.

**Since here in this program we are using 9 threads for the execution of fact() method therefore the time taken for execution will be short.**

## While using the multiThreads, we incountered with some issues:

1. **Thread Management Overhead**: Creating and destroying threads frequently can be resource-intensive and slow down application performance.
2. **Complexity of Concurrent Programming**: Writing multithreaded code can be complex and error-prone, leading to issues like race conditions, deadlocks, and resource contention.
3. **Lack of Scalability**: Managing a large number of threads manually can lead to inefficiencies and make it difficult to scale applications effectively.
4. **Difficulty in Task Scheduling**: Implementing scheduling for recurring or delayed tasks manually can be cumbersome and error-prone.
5. **Error Handling Challenges**: Handling exceptions and errors in multithreaded environments can be complex, leading to unhandled exceptions or inconsistent states.
6. **Inefficient Resource Utilization**: Without proper management, threads can become idle or starved for resources, leading to wasted CPU cycles and performance degradation.
7. **Blocking Operations**: Managing blocking operations (like I/O) manually can lead to poor responsiveness in applications, especially in user interfaces.

**To solve these problems The executors frameworks comes in play:**

**Here are the key advantages of using the Executors framework in Java:**

1. **Thread Pooling**: It efficiently manages a pool of threads, reducing the overhead associated with creating and destroying threads for each task.
2. **Simplified Concurrency**: The framework abstracts the complexity of thread management, making it easier to implement concurrent applications.
3. **Task Scheduling**: It provides built-in capabilities for scheduling tasks, allowing for delayed or periodic execution without additional code.
4. **Flexible Thread Configuration**: You can easily configure the number of threads and their behavior, adapting to different application needs (e.g., fixed vs. cached thread pools).
5. **Enhanced Resource Utilization**: By reusing threads, it optimizes resource usage, reducing idle time and improving performance.
6. **Error Handling**: It includes mechanisms for managing exceptions in asynchronous tasks, improving the robustness of your applications.

7. **Future and Callable Support**: The framework supports Future and Callable interfaces, enabling the retrieval of results from asynchronous tasks.
8. **Improved Readability and Maintainability**: By reducing boilerplate code and providing clear abstractions, it makes concurrent code easier to read, understand, and maintain.
9. **Scalability**: It allows applications to scale more easily by efficiently managing multiple tasks, adapting to varying loads without significant code changes.

Overall, the Executors framework streamlines the development of multithreaded applications, enhancing both performance and code quality.

## Thread Pooling

Thread pooling is a technique used in concurrent programming to manage a group of worker threads efficiently. Instead of creating a new thread for each task, a fixed number of threads are created and reused for executing multiple tasks, which helps reduce the overhead associated with thread creation and destruction.

**Key Concepts**

1. **Thread Pool**: A collection of threads that are created once and reused for executing tasks. This avoids the cost of creating a new thread for every task, which can be significant in a high-throughput application.
2. **Task Submission**: Tasks can be submitted to the thread pool, typically through an interface or method that accepts a Runnable or Callable task.
3. **Work Distribution**: The thread pool manages the distribution of tasks to the available threads, ensuring that they are executed concurrently.
4. **Thread Reuse**: Once a thread completes its task, it returns to the pool and waits for new tasks to be assigned, allowing for efficient use of system resources.

**Benefits of Thread Pooling**

- **Performance Improvement**: Reduces the overhead of thread creation and destruction, leading to better performance in applications that require high concurrency.
- **Resource Management**: Controls the number of concurrent threads, which can help prevent resource exhaustion and improve stability.
- **Simplified Task Management**: Allows easy management of tasks without dealing with the complexities of thread lifecycle management.

**Java Implementation**

In Java, the **ExecutorService** interface provides a simple way to create and manage thread pools. The Executors class provides factory methods to create different types of thread pools.

**Example**

Here's a simple example of using a thread pool in Java:

```
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class ThreadPoolExample {

    public static void main(String[] args) {

        // Create a thread pool with 5 threads

        ExecutorService executorService = Executors.newFixedThreadPool(5);


        // Submit tasks to the thread pool

        for (int i = 0; i < 10; i++) {

            final int taskId = i;

            executorService.submit(() -> {

                System.out.println("Task " + taskId + " is being executed by " +
Thread.currentThread().getName());

                try {

                    Thread.sleep(1000); // Simulate task work

                } catch (InterruptedException e) {

                    Thread.currentThread().interrupt();

                }

            });

        }


        // Shut down the executor service

        executorService.shutdown();
    }
}
```

**Explanation of the Example**

1. **Creating the Thread Pool**:

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
```

This creates a thread pool with a fixed number of 5 threads.

2. **Submitting Tasks**:

```
for (int i = 0; i < 10; i++) {
    final int taskId = i;
    executorService.submit(() -> {
        System.out.println("Task " + taskId + " is being executed by " +
Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });
}
```

Here, we submit 10 tasks to the pool. Each task prints its ID and simulates work by sleeping for 1 second.

3. **Shutting Down the Executor**:

```
executorService.shutdown();
```

The statement executorService.shutdown(); does not immediately stop all executing threads; rather, it initiates an orderly shutdown of the ExecutorService. Here's what happens:

1. **No New Tasks Accepted**: Once shutdown() is called, the ExecutorService will no longer accept new tasks for execution. Any attempt to submit new tasks will result in a RejectedExecutionException.

2. **Running Tasks Continue**: All currently executing tasks will continue running until they complete. The shutdown does not forcibly terminate these tasks.
3. **Pending Tasks**: Any tasks that were submitted but not yet started (pending in the queue) will not be executed.
4. **Graceful Shutdown**: The shutdown process is orderly. It allows currently running tasks to finish, ensuring that resources are cleaned up properly.
5. **Awaiting Completion**: To wait for the tasks to finish, you would typically follow shutdown() with awaitTermination(), which blocks the calling thread until all tasks have completed or a specified timeout occurs.

**Summary**

So, to clarify:

- **shutdown()**: Initiates an orderly shutdown, preventing new tasks from starting, but allows ongoing tasks to complete.
- **To stop execution immediately**: You would use executorService.shutdownNow(), which attempts to stop all actively executing tasks and halts the processing of waiting tasks.

**Conclusion**

Thread pooling is a powerful technique in concurrent programming that enhances performance, resource management, and task handling in multithreaded applications. By reusing a fixed number of threads, it reduces the overhead of thread management and helps maintain a stable and efficient application.

## <u>Future</u>

**Future** is an **interface** that represents **the result of an asynchronous computation**. It acts as a placeholder for a value that may not yet be available but will be at some point in the future.

Here are the key features and functionalities of the Future interface:

1. **Asynchronous Result**: A Future allows you to retrieve the result of a computation that is performed in a separate thread, enabling non-blocking behavior.
2. **Completion Status**: It provides methods to check if the computation is complete, cancelled, or still running. This helps in managing task execution more effectively.

3. **Retrieving Results**: You can call **get()** on a **Future object** to retrieve the result of the computation. If the computation is not complete, this method will block until the result is available.
4. **Cancellation**: The Future interface includes methods like **cancel()** to attempt to cancel the execution of the task. If the task has not started yet, it will be cancelled; if it's already running, you can check if it was cancelled.
5. **Exception Handling**: If the computation throws an exception, calling **get()** will throw an **ExecutionException**, allowing you to handle errors that occurred during the task execution.

**Example Usage**

Here's a simple example of using Future with ExecutorService:

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

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;


public class FutureExample {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<Long> future = executor.submit(new Callable<Long>() {

            @Override

            public Long call() throws Exception {

                // Simulate a long-running task

                Thread.sleep(2000);

                return 42L; // The result of the computation

            }

        });
        try {
            // Do other tasks while the computation is running
            System.out.println("Doing other work...");
            Long result = future.get(); // This will block until the result is available
            System.out.println("Result: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

**Summary**

In summary, a Future in Java provides a way to manage asynchronous computations, allowing you to check the status of tasks, retrieve results, and handle exceptions effectively. It plays a crucial role in concurrent programming, making it easier to work with threads and manage the results of their computations.

**Long result = future.get();**

Calling the **get()** method on a **Future** will block the calling thread (**in this case, the main thread**) until the result of the asynchronous computation is available.

Here's how it works:

1. **Blocking Behavior**: When you invoke **future.get(),** if the computation is not yet complete, the main thread will pause and wait until the result is ready. This means that no further execution in the **main** thread will occur until the **get()** call returns.
2. **Completion**: Once the computation finishes, either successfully or with an exception, the get() method will return the result or throw an ExecutionException, respectively.
3. **Usage Context**: While the main thread is blocked, other tasks could still be executed if they were running in separate threads. However, any code following the get() call in the main thread will not execute until the result is available.

**Example**

In the example I provided earlier:

```
Long result = future.get(); // This will block until the result is available
```

If the computation takes a couple of seconds, the main thread will wait during this time. If you want to perform other tasks while waiting for the result, you would need to manage those tasks in separate threads or use non-blocking techniques.

**what is the advantage of creating a new thread even that code which is going to be executed by genereted thread can also be executed by main thread?**

While the main thread can execute the same code, using a separate thread for execution still offers several key advantages:

1. **Responsiveness**: By offloading tasks to a separate thread, the main thread can continue to execute other code or respond to user input. This is crucial in applications with a user interface, where blocking the main thread can lead to a frozen or unresponsive application.
2. **Parallelism**: On multi-core systems, separate threads can run in parallel. This means that if your task is CPU-intensive, it can utilize multiple cores, speeding up execution time significantly compared to running everything sequentially on the main thread.
3. **Non-blocking Operations**: If a task involves I/O operations (like reading from a file or making a network request), running it in a separate thread allows the main thread to remain free for other tasks. This prevents the application from becoming unresponsive while waiting for the I/O operation to complete.
4. **Separation of Logic**: Using separate threads helps keep different concerns distinct. For instance, one thread can handle background processing, while another handles user interface updates or input. This separation can lead to cleaner, more maintainable code.
5. **Task Management**: Utilizing a thread pool (via ExecutorService) allows for efficient management of multiple tasks, including reusing threads and handling exceptions more gracefully without complicating the main thread's logic.

**Summary**

In summary, while the main thread could execute the same tasks, using separate threads allows for greater responsiveness, improved performance through parallelism, and better organization of code, especially in applications where maintaining a responsive user experience is crucial.

**What is the difference between the Runnable and Callable Interfaces in java?**

The main differences between the Runnable and Callable interfaces in Java are as follows:

**1. Return Type**

- **Runnable**: The run() method does not return a value. It has a void return type.
- **Callable**: The call() method returns a value and can throw checked exceptions. You define the return type when implementing Callable.

**2. Exception Handling**

- **Runnable**: The run() method cannot throw checked exceptions. Any exception that occurs must be unchecked (i.e., a subclass of RuntimeException).
- **Callable**: The call() method can throw checked exceptions, allowing for more flexible error handling.

**3. Usage with Executor Framework**

- **Runnable**: You can submit a Runnable to an ExecutorService, but it cannot provide a result.
- **Callable**: You can submit a Callable to an ExecutorService, and it returns a Future object that can be used to retrieve the result once the computation is complete.

**4. Purpose**

- **Runnable**: Typically used for tasks that do not require a return value, such as background operations or tasks that just perform actions.
- **Callable**: Used for tasks that need to return a result or may throw exceptions, such as computations or tasks requiring result collection.

**Example**

Here's a simple comparison:

**Runnable Example**:

```
Runnable runnableTask = () -> {
    System.out.println("Running a Runnable task.");
};


new Thread(runnableTask).start();
```

**Callable Example**:

```
Callable<Integer> callableTask = () -> {
    System.out.println("Running a Callable task.");
    return 42; // Returning a result
};


// Using ExecutorService to submit the Callable
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<Integer> future = executor.submit(callableTask);


try {
    Integer result = future.get(); // Blocking until the result is available
    System.out.println("Result from Callable: " + result);
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    executor.shutdown();
}
```

**Summary**

In summary, Runnable is suitable for tasks that do not require a return value and cannot throw checked exceptions, while Callable is better for tasks that return a result and may throw checked exceptions, making it more flexible for complex task handling in concurrent programming.

## Methods of Executor Service

**1. submit(Runnable task)**

- **Description**: Submits a Runnable task for execution and returns a Future<?>.
- **Example**:

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<?> future = executor.submit(() -> {

    System.out.println("Runnable task is running.");

});
```

**2. submit(Callable<T> task)**

- **Description**: Submits a Callable task and returns a Future<T>.
- **Example**:

```
Future<Integer> future = executor.submit(() -> {

    return 42; // Callable returning a value

});
```

**3. submit(Runnable task with return statement)**

- **Description**: This is similar to submit(Runnable); however, Runnable does not return a value. Use Callable for return values.
- **Example**:

```
// This won't return a value; use Callable if you need a return

Future<?> future = executor.submit(() -> {

    System.out.println("Running Runnable task.");

});
```

**4. shutdown()**

- **Description**: Initiates an orderly shutdown of the executor service, preventing new tasks from being accepted.
- **Example**:

```
executor.shutdown(); // No more tasks will be accepted
```

**5. shutdownNow()**

- **Description**: Attempts to stop all actively executing tasks and halts the processing of waiting tasks.

- **Example**:

```
List<Runnable> notExecutedTasks = executor.shutdownNow(); // Forces shutdown
```

**6. awaitTermination(long timeout, TimeUnit unit)**

- **Description**: Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs.
- **Example**:

```
try {
    if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
        executor.shutdownNow(); // Force shutdown after timeout
    }
} catch (InterruptedException e) {
    executor.shutdownNow();
}
```

**7. isShutdown()**

- **Description**: Returns true if the executor has been shut down.
- **Example**:

```
if (executor.isShutdown()) {
    System.out.println("Executor is shut down.");
}
```

**8. isTerminated()**

- **Description**: Returns true if all tasks have completed following a shutdown.
- **Example**:

```
if (executor.isTerminated()) {
    System.out.println("All tasks are completed.");
}
```

**9. invokeAll(Collection<? extends Callable<T>> tasks)**

- **Description**: Executes a collection of Callable tasks and returns a list of Future<T>.
- **Example**:

```
List<Callable<Integer>> tasks = Arrays.asList(
    () -> { return 1; },
    () -> { return 2; }
);
```

List<Future<Integer>> results = executor.invokeAll(tasks);

**10. invokeAny(Collection<? extends Callable<T>> tasks)**

- **Description**: Executes a collection of Callable tasks and returns the result of the first successfully completed task.
- **Example**:

```
Integer result = executor.invokeAny(Arrays.asList(
    () -> { Thread.sleep(1000); return 1; },
    () -> { return 2; }
));
```

**Summary**

These methods provide robust mechanisms for managing concurrent tasks in Java, allowing you to control task execution, shutdown behavior, and manage results effectively.

**Example**

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class ExecutorServiceExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Submit Runnable task
```

```
executor.submit(() -> System.out.println("Runnable task is running."));

// Submit Callable task
Future<Integer> future = executor.submit(() -> 42);
System.out.println("Callable result: " + future.get());

// Shutdown
executor.shutdown();

// Check if shutdown
if (executor.isShutdown()) {
    System.out.println("Executor is shut down.");
}

// Invoke all tasks
List<Callable<Integer>> tasks = Arrays.asList(
    () -> { return 1; },
    () -> { return 2; }
);
List<Future<Integer>> results = executor.invokeAll(tasks);
for (Future<Integer> result : results) {
    System.out.println("Result: " + result.get());
}

// Invoke any
Integer firstResult = executor.invokeAny(Arrays.asList(
    () -> { Thread.sleep(1000); return 1; },
    () -> { return 2; }
));
```

```
        System.out.println("First completed task result: " + firstResult);


        // Await termination
        executor.shutdown();
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }
    }
}
```

## Methods on Future

The Future interface in Java represents the result of an asynchronous computation. Here are the key methods provided by the Future interface, along with brief explanations and examples:

**1. get()**

- **Description**: Retrieves the result of the computation when it is complete. This method blocks until the result is available.
- **Example**:

```
Future<Integer> future = executor.submit(() -> {
    // Simulate a long-running task
    Thread.sleep(1000);
    return 42;
});
Integer result = future.get(); // Blocks until the result is ready
System.out.println("Result: " + result);
```

**2. get(long timeout, TimeUnit unit)**

- **Description**: Retrieves the result, waiting up to the specified timeout. If the result is not available within the timeout, a TimeoutException is thrown.
- **Example**:

```
try {

   Integer result = future.get(500, TimeUnit.MILLISECONDS); // Waits up to
500 ms

   System.out.println("Result with timeout: " + result);

} catch (TimeoutException e) {

   System.out.println("Task timed out!");

}
```

**3. isDone()**

- **Description**: Returns true if the task is completed, regardless of whether it completed successfully, failed, or was canceled.
- **Example**:

```
if (future.isDone()) {

   System.out.println("Task is completed.");

} else {

   System.out.println("Task is still running.");

}
```

**4. isCancelled()**

- **Description**: Returns true if the task was canceled before it completed.
- **Example**:

```
future.cancel(true); // Attempt to cancel the task

if (future.isCancelled()) {

   System.out.println("Task was canceled.");

}
```

**5. cancel(boolean mayInterruptIfRunning)**

- **Description**: Attempts to cancel the task. If the task is already running, it can be interrupted if mayInterruptIfRunning is true.
- **Example**:

```
boolean canceled = future.cancel(true); // Attempt to cancel

if (canceled) {

   System.out.println("Task was successfully canceled.");
} else {
    System.out.println("Task could not be canceled.");
}
```

**Summary of Future Methods**

- **get()**: Blocks until the result is available.
- **get(long timeout, TimeUnit unit)**: Blocks with a timeout.
- **isDone()**: Checks if the task is completed.
- **isCancelled()**: Checks if the task was canceled.
- **cancel(boolean mayInterruptIfRunning)**: Attempts to cancel the task.

**Complete Example**

**Here’s a complete example demonstrating these methods:**

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class FutureExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(1);

        Future<Integer> future = executor.submit(() -> {
            try {
                // Simulate a long-running task
                Thread.sleep(2000);
                return 42;
            } catch (InterruptedException e) {
                return 0; // Handle interruption
            }
        });

        // Check if the task is done
        if (!future.isDone()) {
            System.out.println("Task is still running...");
        }

        // Attempt to cancel the task
        boolean canceled = future.cancel(false);
        System.out.println("Task canceled: " + canceled);

        // Try getting the result
        try {
            Integer result = future.get(1, TimeUnit.SECONDS); // Wait up to 1 second
            System.out.println("Result: " + result);
        } catch (Exception e) {
            System.out.println("Could not retrieve result: " + e.getMessage());
        }

        // Final cleanup
        executor.shutdown();
    }
}
```

This example demonstrates how to use the Future interface methods to manage and interact with asynchronous tasks effectively.

## ScheduledExecutorService:

ScheduledExecutorService is an interface in Java that provides methods to schedule tasks to be executed after a certain delay, or periodically, in a background thread pool. It is part of the java.util.concurrent package, which allows you to handle concurrent tasks in a more efficient and flexible way than traditional Timer or Thread classes.

**Key Features of ScheduledExecutorService:**

1. **Scheduling with Delay**: You can schedule tasks to be executed after a fixed delay.
2. **Periodic Tasks**: You can schedule tasks to be executed periodically with fixed-rate or fixed-delay execution policies.
3. **Thread Pool Management**: It manages a pool of worker threads to execute tasks concurrently.
4. **Task Cancellation**: It provides methods to cancel scheduled tasks before they execute.

**Common Methods:**

- **schedule(Runnable command, long delay, TimeUnit unit):** Schedules a task to run once after the specified delay.
- **schedule(Callable<V> callable, long delay, TimeUnit unit):** Schedules a Callable task to run once after the specified delay.
- **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit):** Schedules a task to run repeatedly at a fixed rate.
- **scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit):** Schedules a task to run repeatedly with a fixed delay between the end of one execution and the start of the next.
- **shutdown():** Initiates an orderly shutdown of the executor, in which previously submitted tasks are executed, but no new tasks will be accepted.

**Example Usage:**

```
import java.util.concurrent.*;


public class ScheduledExecutorExample {
    public static void main(String[] args) {

        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        // Schedule a task to run after 2 seconds
        scheduler.schedule(() -> {
            System.out.println("Task executed after 2 seconds");
        }, 2, TimeUnit.SECONDS);

        // Schedule a task to run periodically every 3 seconds, after an initial delay of 1 second
        scheduler.scheduleAtFixedRate(() -> {
            System.out.println("Periodic task executed at fixed rate");
        }, 1, 3, TimeUnit.SECONDS);

        // Optionally, shut down the executor service
        // scheduler.shutdown();
    }
}
```

**Types of Tasks:**

1. **Fixed-rate Scheduling** (scheduleAtFixedRate): The time between the start of one execution and the start of the next is fixed. This may cause overlapping executions if the task takes longer to execute than the period specified.
2. **Fixed-delay Scheduling** (scheduleWithFixedDelay): The time between the completion of one execution and the start of the next is fixed. This ensures that there's a fixed delay between executions, but the actual interval may vary depending on how long the task takes to execute.

**Choosing Between Fixed-Rate vs. Fixed-Delay:**

- Use **fixed-rate** when you want the tasks to execute at consistent intervals regardless of how long the task execution takes.
- Use **fixed-delay** when you want the task to be executed after the previous task has completed, ensuring a delay between each execution.

**Important Notes:**

- Always ensure to shut down the executor using shutdown() to prevent resource leakage.

- Be cautious of long-running tasks in the executor, as they could block the thread pool.

**Alternatives to ScheduledExecutorService:**

- **Timer**: An older approach, but less flexible and less robust than ScheduledExecutorService.
- **CompletableFuture**: Can also be used for scheduling tasks with delays, especially when you want to work with future results.

detailed example showing both **fixed-rate scheduling** and **fixed-delay scheduling** using the ScheduledExecutorService:

**Example Code:**

```
import java.util.concurrent.*;


public class ScheduledExecutorServiceExample {
   public static void main(String[] args) {
      // Create a ScheduledExecutorService with a pool of 2 threads
      ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);


      // Fixed-rate scheduling (time between the start of one execution and the start
of the next is fixed)
      scheduler.scheduleAtFixedRate(() -> {
         System.out.println("Fixed-rate task started at: " + System.currentTimeMillis());
         try {
            // Simulate a task that takes time to execute
            Thread.sleep(3000); // sleep for 3 seconds
         } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
         }
```

```
            System.out.println("Fixed-rate task completed at: " +
System.currentTimeMillis());
        }, 0, 2, TimeUnit.SECONDS);  // Initial delay: 0, Period: 2 seconds

        // Fixed-delay scheduling (time between the completion of one execution and
the start of the next is fixed)
        scheduler.scheduleWithFixedDelay(() -> {
            System.out.println("Fixed-delay task started at: " +
System.currentTimeMillis());
            try {
                // Simulate a task that takes time to execute
                Thread.sleep(3000); // sleep for 3 seconds
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Fixed-delay task completed at: " +
System.currentTimeMillis());
        }, 0, 2, TimeUnit.SECONDS);  // Initial delay: 0, Delay between completions: 2
seconds

        // Run for 10 seconds and then shut down the scheduler
        try {
            Thread.sleep(10000); // Let tasks run for 10 seconds
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            scheduler.shutdown();
        }
    }
}
```

**Explanation:**

1. **Fixed-rate Scheduling** (scheduleAtFixedRate):
    - The fixed-rate task is scheduled to start every 2 seconds (Period = 2 seconds).
    - Even though each task takes 3 seconds to execute (because of Thread.sleep(3000)), the **next task** will still start exactly **2 seconds after the last one started**, which can cause overlapping executions. This is an important behavior to note.
    - **Output**:

```
Fixed-rate task started at: 1631111105000

Fixed-rate task completed at: 1631111108000

Fixed-rate task started at: 1631111107000 (overlaps the previous task)
```

2. **Fixed-delay Scheduling** (scheduleWithFixedDelay):
    - The fixed-delay task is scheduled to start every 2 seconds **after the completion** of the previous task.
    - Since each task takes 3 seconds to execute, there will be a delay between the tasks, and the next task will start exactly 2 seconds **after the previous task completes**.
    - **Output**:

```
Fixed-delay task started at: 1631111105000

Fixed-delay task completed at: 1631111108000

Fixed-delay task started at: 1631111110000 (delay after previous completion)
```

**Key Differences:**

- **Fixed-rate**: The tasks are scheduled to start at a fixed interval (2 seconds here). If a task takes longer than the scheduled period, it can overlap with the next task.
- **Fixed-delay**: The tasks are scheduled with a delay between the **end** of one execution and the **start** of the next. This avoids overlap but results in irregular intervals if tasks take longer than expected.

**Expected Output (Example Timing):**

```
Fixed-rate task started at: 1631111105000

Fixed-rate task completed at: 1631111108000

Fixed-rate task started at: 1631111107000  // Overlaps

Fixed-rate task completed at: 1631111110000
```

```
Fixed-delay task started at: 1631111105000

Fixed-delay task completed at: 1631111108000

Fixed-delay task started at: 1631111110000

Fixed-delay task completed at: 1631111113000
```

The Fixed-rate task will execute at a constant rate, potentially leading to overlap, while the Fixed-delay task respects the execution time, ensuring that tasks don't overlap and are spaced out by the specified delay.