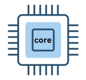
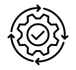
Multithreading

It is like Brain of computer (intel , AMD)



Core -Core is individual processing unit which present inside cpu
-now a days cpu consists more than 1 core etc



Process- It is like whenever we do activity like opening VS code ,surffing etc that time Our OS will start process

Thread- In simple word thread it is present inside process i.e. above and thread can run independently

ex: when i open browser the OS will starts process and, in that process multiple threads are running like 1 thread is one tab another thread 2nd tab

Single task: When we have single core CPU, this is done through a time sharing, rapidly switching between task so that you don't even notice

-earlier computers comes 1core

Multitasking: allows an OS to run multiple process in simultaneously

Multithreading: It is ability to execute multiple threads in a single process concurrently

ex: we are listening to music in YouTube and browsing in webpage and downloading file all doing in a same time

Multithreading-begin

main thread

- whenever we just run our 1st program main thread

```
public class Main {

   public static void main(String[] args) {
      System.out.println("hello world");
      System.out.println(Thread.currentThread().getName());

   }
}
```

output:

hello world
main

Process finished with exit code 0---->means 1 thread is finished

To create a new thread in java , you can either extend Thread class or implements the Runnable interface

create thread by extending Thread class

step 1 create thread class

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

step 2 creatin main

```
public class Main {

   public static void main(String[] args) {

      World world = new World();
      world.start();
      for(; ;){
         System.out.println(Thread.currentThread().getName());
      }
   }
}
```

output
main
main
Thread-0:

note : Thread will execute randmoly

create thread by implementing Runnable interface

step 1 create thread class

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

step 2 creatin main

```
public class Main {

   public static void main(String[] args) {

      World world = new World();
      Thread thread =new  Thread(world);
      thread.start();
      for(; ;){
         System.out.println(Thread.currentThread().getName());
      }
   }
}
```

output
main
main
Thread-0:

Thread Lifecycle

NEW Thread in this state when it is created but not yet started

RUNNABLE After the start method is called, thread became runnable, it is ready to run and waiting for CPU time

Running Thread is in this state when it is executing

Blocking/ waiting Thread is in this state when waiting for resource or another thread to perform action

Terminated A thread is in this state when it has finished executing

Ex:

```
public class MyThread extends Thread {

  @Override
  public void run() {
    System.out.println("Running");

    try {
      Thread.sleep(2000);
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  }

  public static void main(String[] args) throws InterruptedException {

    MyThread thread = new MyThread();
    System.out.println(thread.getState()); //NEW
    thread.start();
    System.out.println(thread.getState());//RUNNABLE , Running

    //as per above step we need next Running to perform Running we need to stop main thread
    //and then call our thread

    Thread.sleep(1000);
    System.out.println(thread.getState());//Running

    thread.join();
    System.out.println(thread.getState());//Terminated

  }
}
```

output:
NEW
RUNNABLE
Running
TIMED_WAITING
TERMINATED

Thread vs Runnable

Thread

- You use Thread only when your class not extending other class

```
public class MyThread extends Thread {

  public static void main(String[] args) {

  }
}
```

Runnable

- You use Runnable only when your class already extending other class

```
public class MyThread extends A, Thread {

  public static void main(String[] args) {

  }
}
```

```
public class MyThread extends A implements Runnable {

  @Override
  public void run() {

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

  }
}
```

Thread methods

.run()   .start()   .sleep()   .join()

```
public class MyThread extends Thread{

   @Override
   public void run() { //run method

        try {
           Thread.sleep(1000); // sleep method
        } catch (InterruptedException e) {
           throw new RuntimeException(e);
        }
   }
   public static void main(String[] args) throws InterruptedException {
      MyThread t1 = new MyThread();
      t1.start(); // start method
      t1.join(); // join method
      System.out.println("Main method");
   }
}
```

.setPriority()

You can set setPriiority as--->LOW , NORMAL , HIGH

```
 MyThread(String name){ //custom Thread name
    super(name);
 }
 @Override
 public void run() { //run method

      for (int i = 0; i <5;i++){
         System.out.println(Thread.currentThread().getName()+"  Priority  "+ Thread.currentThread().getPriority()+" count "+i);
       try {
          Thread.sleep(10);
       }
       catch (Exception e){

       }
      }
 }
 public static void main(String[] args) throws InterruptedException {
    MyThread t1 = new MyThread("High");//set custom thread name
    MyThread t2 = new MyThread("Medium");//set custom thread name
    MyThread t3 = new MyThread("Low");//set custom thread name

    t1.setPriority(Thread.MAX_PRIORITY); //set priority to max
    t2.setPriority(Thread.NORM_PRIORITY); //set priority to normal
    t3.setPriority(Thread.MIN_PRIORITY); //set priority to min
    t1.start(); // start method
    t2.start(); // start method
    t3.start(); // start method

 }
}
```

OUTPUT:

```
High  Priority  10 count 3
Medium  Priority  5 count 3
Low  Priority  1 count 3
Medium  Priority  5 count 4
High  Priority  10 count 4
Low  Priority  1 count 4
```

.interrupt()---> It will interrupt your method

```
public class MyThread extends Thread{

   MyThread(String name){ //custom Thread name
      super(name);
   }
   @Override
   public void run() { //run method

        try {
          Thread.sleep(1000);
          System.out.println("Running");
        }
        catch (Exception e){
          System.out.println("Your thread is interrupted");
        }
   }
   public static void main(String[] args) throws InterruptedException {
      MyThread t1 = new MyThread("Rahul");//set custom thread name
      t1.start(); // start method
      t1.interrupt();
   }
}
```

output:Your thread is interrupted

.yield()---> it means you can give other threads also chance to run

```
public class MyThread extends Thread{

   MyThread(String name){ //custom Thread name
      super(name);
   }
   @Override
   public void run() { //run method

       for(int i = 0; i <5;i++){
         try {
            Thread.sleep(10);
            System.out.println(Thread.currentThread().getName());
            Thread.yield();//yield method
         }
         catch (Exception e){
            System.out.println("Your thread is interrupted");
         }
       }
   }
   public static void main(String[] args) throws InterruptedException {
      MyThread t1 = new MyThread("Rahul");//set custom thread name
      MyThread t2 = new MyThread("Raj");//set custom thread name
      t1.start(); // start method
      t2.start(); // start method
   }
```

OUTPUT:
Raj
Rahul
Rahul
Raj
Rahul
Raj
Raj

- **user thread it means you working thread, it means your MyThread--->you created this myThread it is user thread**
- **Your user thread JVM will not stop user thread**

- **setDaemon() thread it is not your user thread, it is running in background thread and JVM will not wait for setDaemon() thread**
- **setDaemon() JVM will stop Daemon thread**

```
public class MyThread extends Thread{

   MyThread(String name){ //custom Thread name
      super(name);
   }
   @Override
   public void run() { //run method

       while (true) {
          System.out.println("Running");
       }
   }
   public static void main(String[] args) throws InterruptedException {
      MyThread t1 = new MyThread("Rahul");//Your user thread JVM will not stop user thread
      t1.setDaemon(true); //Now JVM will stop background thread
      t1.start(); // start method
      System.out.println("main method");

   }
}
```

NOTE: even though you are running while(true) infinite time, but when JVM sees Daemon thread it will stop your user thread

Overview

.run() .start() .sleep() .join() .setPriority() .interrupt() .yield() user thread Daemon thread

Synchronization

- Synchronization : Synchronization it means i am telling t2 thread wait until t1 thread complete its process
- we add synchronized keyword to those methods which you want to follow flow no thread interrupts

1

```
public class Balance {

   private  int amount=0;

   public int getAmount() {
      return amount;
   }

   public synchronized int increment(){

         return amount++;

   }

}
```

2

```
public class NewThread extends Thread{

   private final Balance balance;

   public NewThread(Balance balance) {
      this.balance = balance;
   }

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

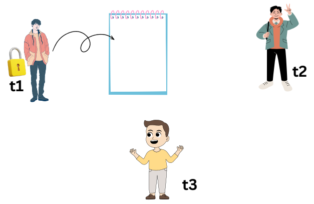
3

```
public class Testing {

   public static void main(String[] args) throws InterruptedException {
      Balance balance = new Balance();
      NewThread t1 = new NewThread(balance);
      NewThread t2 = new NewThread(balance);

      t1.start();
      t2.start();

      t1.join();
      t2.join();

      System.out.println(balance.getAmount());

   }
}
```

with synchronized
output:2000

without synchronized
output: random numbers

LOCKING



See just above diagram **t1** has lock so he can only write in page and **t2 t3** not able to writei in page because they don’t have lock

Types of locks

- **Intrinsic:** They are built in every object in java, you don’t see them, but it is present
- when your using synchronized keyword your using automatic lock

- **Explicit:** These are more advance locks you can controll yourself using class from java.util.concurrent.locks
- your explicitly when to lock when to unlock

we already know as Intrinsic which is also known as Synchronized

Explicit

1

```
public class Testing {

   public static void main(String[] args) throws InterruptedException {

      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();

   }
}
```

OUTPUT
Thread 1 attempting to withdraw
Thread 2 attempting to withdraw
Thread 1 Processing to withdraw
Thread 2 Could not aquire key try again later
Thread 1 Withdrawl successfully 50

2

```
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 ");
     try {
        if(lock.tryLock(1000, TimeUnit.MILLISECONDS)){
     if(balance>=amount){
       try {
          System.out.println(Thread.currentThread().getName()+" Processing to withdraw ");
          Thread.sleep(5000);
          balance-=amount;
          System.out.println(Thread.currentThread().getName()+" Withdrawl successfully "+ balance);
       } catch (Exception e) {
      Thread.currentThread().interrupt();
       }
       finally {
          lock.unlock();
       }
        }
     else {
        System.out.println("Insufficient balance");
     }

        }
        else{
           System.out.println(Thread.currentThread().getName() + " Could not aquire key try again later ");
        }
     } catch (InterruptedException e) {
       Thread.currentThread().interrupt();
     }
   }
```

In above example we seen different types of method uses in explicit lock i.e.

- **private final Lock lock = new ReentrantLock();**
- **lock.tryLock(1000, TimeUnit.MILLISECONDS)**
- **lock.unlock();**

What is mean by ReentrantLock ?

ReentrantLock basically which help us to-do fair and unfair lock **This topic will discuss after Outer method and inner method**

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

   public static void main(String[] args) {
      ReenterentExample reenterentExample = new ReenterentExample();

      reenterentExample.outerMethod();


   }

   public void outerMethod() {
      lock.lock();                     We did lock for example  count =1
      try {
         System.out.println("Outer method");
         innerMethod();
      } finally {
         lock.unlock();                We did unlock for example  count will reduces =1 to 0
      }
   }

   public void innerMethod() {
      lock.lock();                     We did again lock for example  count =2
      try {
         System.out.println("inner method");
      } finally {
         lock.unlock();      We did unlock for example  count will reduces =2 to 1

      }
   }
}
```

ReentrantLock basically which help us to-do fair and unfair lock

unfair lock

unfair lock: sometime it will miss /not-give opportunity to other thread

```
public class UnfairLockExample {

   private final Lock unfairLock = new ReentrantLock();
   public void  accessResource(){
      unfairLock.lock();
      try {
         System.out.println(Thread.currentThread().getName()+"acquired the lock");
      }catch (Exception e){
         Thread.currentThread().interrupt();
      }finally {
         System.out.println(Thread.currentThread().getName()+"released the lock");
         unfairLock.unlock();
      }
   }

   public static void main(String[] args) {
      UnfairLockExample unfairLockExample = new UnfairLockExample();
      Runnable task = new Runnable() {
         @Override
         public void run() {
            unfairLockExample.accessResource();
         }
      };
      Thread thread1 = new Thread(task," Thread 1 ");
      Thread thread2= new Thread(task," Thread 2 ");
      Thread thread3= new Thread(task," Thread 3 ");

      thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

fair lock

Fair Lock : It will give opportunity to all the threads it will not miss single threads

```
public class FairLockExample {

   private final Lock fairLock = new ReentrantLock(true);//just add true it become unfair to fair
   public void  accessResource(){
      fairLock.lock();
      try {
         System.out.println(Thread.currentThread().getName()+"acquired the lock");
      }catch (Exception e){
         Thread.currentThread().interrupt();
      }finally {
         System.out.println(Thread.currentThread().getName()+"released the lock");
         fairLock.unlock();
      }
   }

   public static void main(String[] args) {
      FairLockExample unfairLockExample = new FairLockExample();
      Runnable task = new Runnable() {
         @Override
         public void run() {
            unfairLockExample.accessResource();
         }
      };
      Thread thread1 = new Thread(task," Thread 1 ");
      Thread thread2= new Thread(task," Thread 2 ");
      Thread thread3= new Thread(task," Thread 3 ");

      thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

Synchronized	Vs	ReenterantLock
• No guaranteed Fairness • Blocking happening • Interruptibility not happen here • Read/write locking		• guaranteed Fairness • Blocking resolve • Interruptibility resolve • lock,unlock,tryLock,deadLock(outer,inner method) • interruptibility happen here • Read/write locking we can do here

Read and Write lock

- **See what is happening here in read and write lock**
- **when you reading that means as number of as thread can access at a same time and simultaneously**
- **Multiple threads can read at the same time (if no thread is writing).**

```
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(1000);
      } catch (Exception e) {
         throw new RuntimeException(e);
      }finally {
         writeLock.unlock();
      }
   }

   public  int getCount(){
      readLock.lock();
      try {
         return count;
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
      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()+"reading"+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()+" writing :
incremented ");
            }
         }
      };

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

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

      t1.join();
      t2.join();
      t3.join();

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

when t1 thread is writing other t2 and t3 silent

reading that means as number of as thread can access

DeadLock

What is a Deadlock?
A deadlock occurs when two or more threads are waiting for each other to release resources,
but none can proceed because they are stuck in a circular wait. This results in an indefinite blocking of all involved threads.

How Deadlock Happens?

- A deadlock usually occurs when these four conditions are met:
- Mutual Exclusion – Only one thread can access a resource at a time.
- Hold and Wait – A thread holding at least one resource is waiting for additional resources.
- No Preemption – A resource cannot be forcibly taken from a thread; it must be released voluntarily.
- Circular Wait – A circular chain of threads exists, where each thread is waiting for a resource held by the next thread in the chain.

It is similar to inner and outer method, just remember that because it is more complex

THREAD COMMUNICATION

Thread Communication in Java

Thread communication is a mechanism where multiple threads coordinate their execution using shared resources. This is typically done using methods like wait(), notify(), and notifyAll().

Why is Thread Communication Needed?

- **To prevent race conditions (where multiple threads modify shared data unpredictably).**
- **To synchronize actions between producer and consumer threads.**
- **To increase efficiency by making threads wait instead of continuously checking for conditions.**

```
package com.crud.crud.Synchronized;


class SharedResource{
    private int data;
    private boolean hasData;


    public synchronized int produce(int value){

        while (hasData){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data=value;
        hasData=true;
        System.out.println(" Produced :" +value);
        notify();

        return data;
    }

    public synchronized int consumer(int value) {

        while (!hasData){
            try {
              wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        hasData=false;
        System.out.println(" Consumer :" +value);
        notify();
        return data;
    }
}

class Producer implements Runnable{

    private  final SharedResource resource;

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


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

        }
    }
}

class Consumer implements Runnable{

    private  final SharedResource resource;

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

    @Override
    public void run() {
        for (int i=0;i<10;i++){
            int value=resource.consumer(i);
        }
    }
}
public class ThreadCommunication {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread producerThread = new Thread(new Producer(resource));
        Thread consumerrThread = new Thread(new Consumer(resource));

        producerThread.start();
        consumerrThread.start();

    }



}
```

Thread safety

Thread safety refers to the ability of a program or code segment to function correctly in a multi-threaded environment without causing race conditions, inconsistent data, or unexpected behavior.

LAMBDA EXPRESSION

we here reducing code here we already know how to use lambda expression

Runnable is a functionalinterface--> we can use lambda expression here

ex:

without lambda expression

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

  Thread t1 = new Thread(runnable);
  t1.start();
}
```

with lambda expression

```
Runnable runnable = ()->System.out.println("Thread working");
Thread t1 = new Thread(runnable);
t1.start();
```

OR

```
Thread t1 = new Thread(()-> System.out.println("Thread working"));
t1.start();
```

Thread pool

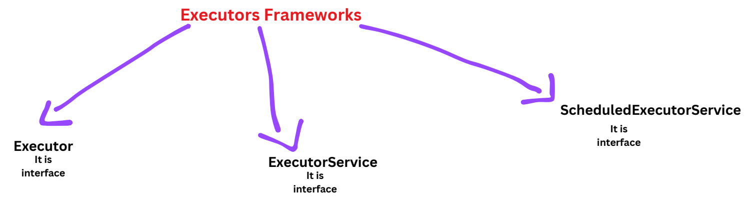
collection of pre--initialized threads

why thread pool?

- Resource management
- Response time
- control over thread count

EXECUTORS FRAMEWORK

- Executors frame work introduced in java 5
- it resolves burden of creating threads and managing threads
- it automates the task of creating thread



Executor : It is a functional interface so we can use lambda expression, it does not consists more usefull method so we do not use here Executor

ExecutorService **Extends Executor and provides methods to manage and control the lifecycle of the thread pool.**

note :
Executor and Executors is different
Executor is we know no useful method present here
but Executors consists useful methods and ExecutorService is extends Executors that means ExecutorService also contains useful methods

Executors Utility Class

- Provides factory methods for different types of thread pools:
- newFixedThreadPool(n): Fixed-size thread pool.
- newCachedThreadPool(): Dynamically growing thread pool.
- newSingleThreadExecutor(): A single-threaded executor.
- newScheduledThreadPool(n): A pool for scheduled tasks.

ex using Callable

```
ExecutorService executorService = Executors.newFixedThreadPool(50);// it means it will create 45 threads
for (int i = 0; i <10;i++){
   Callable<?> task =()-> {
      System.out.println(Thread.currentThread().getName());
      return "Data fetching...";
   };
   Future<?> future= executorService.submit(task);
   System.out.println(future.get());
}
```

using Runnable

```
ExecutorService executorService = Executors.newFixedThreadPool(50);// it means it will create 45 threads
for (int i = 0; i <10;i++){
   executorService.submit(()->{
      System.out.println(Thread.currentThread().getName());
   });
}
```

Runnable VS Callable

Feature	Runnable	Callable
Returns a value?	❌ No	✅ Yes (call() returns a result)
Exception Handling	Cannot throw checked exceptions	Can throw checked exceptions
Method Signature	void run()	T call() throws Exception
Usage	Used in Thread class	Used with ExecutorService (submit())
Result Handling	No result	Uses Future.get() to retrieve the result

ScheduledExecutorService

What is ScheduledExecutorService?
It is a Java feature that lets you run tasks after a delay or repeatedly (like a timer).
You can use it when you want something to happen later or again and again at specific times.

Example 1: Run a Task After Some Time
Imagine you want to print a message after 3 seconds. Here’s how you do it:

```
public class DelayedTaskExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        Runnable task = () -> System.out.println("Hello! This runs after 3 seconds.");

        // Schedule the task to run after 3 seconds
        scheduler.schedule(task, 3, TimeUnit.SECONDS);

        // Shutdown the scheduler after execution
        scheduler.shutdown();
    }
}
```

👉 **This will wait 3 seconds and then print:**
Hello! This runs after 3 seconds.

Example 2: Run a Task Again and Again
You can use ScheduledExecutorService to repeat a task.
1. Run every 5 seconds (Fixed Delay)

```
public class FixedDelayExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        Runnable task = () -> {
            System.out.println("Running at: " + System.currentTimeMillis());
        };

        // Run task every 5 seconds (first run after 2 sec)
        scheduler.scheduleWithFixedDelay(task, 2, 5, TimeUnit.SECONDS);
    }
}
```

🔹 **First run after 2 sec, then every 5 sec exactly, even if a task takes longer.**

When to Use This?
✅ Reminders – Send a notification every hour.
✅ Monitoring – Check system health every 10 seconds.
✅ Delays – Run a task after a few minutes.

CountDownLatch

Imagine a Real-Life Scenario
You and your 3 friends are preparing for a trip, but the car won't start until everyone is ready.

- You (main thread) are the driver.
- Your 3 friends (worker threads) must pack their bags before the trip starts.
- You will wait until all friends finish packing before starting the car.

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

public class TripPreparation {
    public static void main(String[] args) throws InterruptedException {
        int totalFriends = 3;
        CountDownLatch latch = new CountDownLatch(totalFriends); // Countdown starts from 3
        ExecutorService executor = Executors.newFixedThreadPool(totalFriends); // Thread pool

        for (int i = 1; i <= totalFriends; i++) {
            executor.execute(new Friend(latch, "Friend-" + i));
        }

        latch.await(); // 🚗 Wait until all friends are ready
        System.out.println("All friends are ready! Let's start the trip!");

        executor.shutdown(); // Shut down thread pool
    }

    static class Friend implements Runnable {
        private final CountDownLatch latch;
        private final String name;

        public Friend(CountDownLatch latch, String name) {
            this.latch = latch;
            this.name = name;
        }

        @Override
        public void run() {
            try {
                System.out.println(name + " is packing...");
                Thread.sleep((int) (Math.random() * 3000)); // Simulate packing time
                System.out.println(name + " is ready!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                latch.countDown(); // Friend finished packing, count decreases
            }
        }
    }
}
```

🔹 **What Happens in This Code?**

- We create a CountDownLatch with 3 (because we have 3 friends).
- Each friend starts packing (using a thread pool).
- Friends take random time to pack (simulated with Thread.sleep()).
- Each friend calls latch.countDown() after finishing.
- Main thread (latch.await()) waits until all friends finish.
- Once everyone is ready, the trip starts!

✅ **Example Output**
Friend-1 is packing...
Friend-2 is packing...
Friend-3 is packing...
Friend-2 is ready!
Friend-3 is ready!
Friend-1 is ready!
All friends are ready! Let's start the trip!

🔹 **Key Concepts**

ExecutorService creates and manages threads automatically.
latch.await() makes the main thread wait until all workers finish.
latch.countDown() decreases the count when a worker finishes.

What is CyclicBarrier in Java?

CyclicBarrier is used when multiple threads must wait for each other to reach a common point before continuing execution.
Unlike CountDownLatch, CyclicBarrier can be reused, meaning the same barrier can be used multiple times.

Scenario: Two Friends Arriving at a Restaurant

Two friends are arriving at a restaurant to have dinner. The restaurant doesn't allow anyone to enter until both friends arrive. Each friend waits for the other to arrive before they can enter together.

- Person 1 (Friend 1) arrives first but has to wait for Person 2 (Friend 2) to arrive.
- Person 2 (Friend 2) arrives second and waits for Person 1.

Once both have arrived, they can enter the restaurant and start their dinner.

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

public class RestaurantScenario {
    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(2, () ->
            System.out.println("Both friends have arrived! Let's enter the restaurant!")
        );

        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.execute(() -> {
            try {
                System.out.println("Friend 1 is arriving...");
                Thread.sleep(2000); // Simulating travel time
                System.out.println("Friend 1 has arrived and is waiting.");
                barrier.await(); // Wait for Friend 2
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        executor.execute(() -> {
            try {
                System.out.println("Friend 2 is arriving...");
                Thread.sleep(3000); // Simulating travel time
                System.out.println("Friend 2 has arrived and is waiting.");
                barrier.await(); // Wait for Friend 1
            } catch (Exception e) {
                e.printStackTrace();
            }
        });

        executor.shutdown();
    }
}
```

✅ **Output Example**

```
Friend 1 is arriving...
Friend 2 is arriving...
Friend 1 has arrived and is waiting.
Friend 2 has arrived and is waiting.
Both friends have arrived! Let's enter the restaurant!
```

📌 **Key Points:**

CyclicBarrier(2): Waits for 2 threads (representing 2 friends).
barrier.await(): Both friends call await() to wait for each other before entering.
When both arrive, they are allowed to proceed (i.e., enter the restaurant).

What is CompletableFuture?

CompletableFuture is a powerful tool in Java for handling asynchronous programming. I
t allows you to write code that can run in the background while still allowing you to easily handle the result when it's ready.
In simple terms, it helps you to run tasks in parallel without blocking your main program, and then combine the results when all tasks are done.

Real-Life Scenario: Ordering Pizza and Drinks
Imagine you are ordering a pizza and drinks for a party. You can do both at the same time (i.e., asynchronously), and when both are ready, you'll enjoy them.

1. Ordering the pizza.
2. Ordering the drinks.
3. After both orders are done, you enjoy the meal!

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

public class CompletableFutureExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        // Create CompletableFutures for pizza and drinks
        CompletableFuture<String> pizzaOrder = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(3000); // Simulating pizza cooking time
                System.out.println("Pizza is ready!");
                return "Pizza";
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Pizza failed";
        });

        CompletableFuture<String> drinksOrder = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000); // Simulating drink preparation time
                System.out.println("Drinks are ready!");
                return "Drinks";
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Drinks failed";
        });

        // Wait for both orders to complete, then combine results
        CompletableFuture<Void> allOf = CompletableFuture.allOf(pizzaOrder, drinksOrder);
        allOf.get(); // Wait for all futures to complete

        // Combine the results and enjoy
        System.out.println("Time to enjoy the " + pizzaOrder.get() + " and " + drinksOrder.get() + "!");
    }
}
```

🔹 **How This Works:**
1 . CompletableFuture.supplyAsync():
We start two tasks in parallel — one for ordering the pizza and one for ordering drinks. These tasks run asynchronously (without blocking the main thread).

2. **Thread.sleep()** simulates time for each task (e.g., pizza takes 3 seconds, drinks take 2 seconds).

3.**CompletableFuture.allOf(pizzaOrder, drinksOrder):**
This waits for both tasks to complete before continuing. It's like saying, "Wait until both the pizza and drinks are ready!"

4. **pizzaOrder.get() and drinksOrder.get():**
After both tasks are done, we retrieve the results (pizza and drinks), then combine them to print the final message.

✅ **Output Example:**

```
Drinks are ready!
Pizza is ready!
Time to enjoy the Pizza and Drinks!
```

📌 **Key Points about CompletableFuture:**
Run tasks asynchronously: This means tasks like ordering pizza and drinks run in parallel without blocking.
Get results when ready: You can wait for the result using .get(), or combine them with .allOf() to wait for multiple tasks to finish.
Non-blocking and easy to handle: CompletableFuture helps with parallel programming without manually managing threads.