

## CPU

**The CPU, often referred to as the brain of the computer, is responsible for executing instructions from programs. It performs basic arithmetic, logic, control, and input/output operations specified by the instructions.**

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

**A quad-core processor has four cores, allowing it to perform four tasks simultaneously. For instance, one core could handle your web browser, another your music player, another a download manager, and another a background system update.**

**A program is a set of instructions written in a programming language that tells the computer how to perform a specific task**

**Microsoft Word is a program that allows users to create and edit documents.**

**A process is an instance of a program that is being executed. When a program runs, the operating system creates a process to manage its execution.**

**When we open Microsoft Word, it becomes a process in the operating system.**

**A thread is the smallest unit of execution within a process. A process can have multiple threads, which share the same resources but can run independently.**

**A web browser like Google Chrome might use multiple threads for different tabs, with each tab running as a separate thread.**

**Multitasking allows an operating system to run multiple processes simultaneously. On single-core CPUs, this is done through time-sharing, rapidly switching between tasks. On multi-core CPUs, true parallel execution occurs, with tasks distributed across cores. The OS scheduler balances the load, ensuring efficient and responsive system performance.**

## **Example**

**We are browsing the internet while listening to music and downloading a file.**

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

**Multithreading refers to the ability to execute multiple threads within a single process concurrently.**

**A web browser can use multithreading by having separate threads for rendering the page, running JavaScript, and managing user inputs. This makes the browser more responsive and efficient.**

**Multithreading enhances the efficiency of multitasking by breaking down individual tasks into smaller sub-tasks or threads. These threads can be processed simultaneously, making better use of the CPU's capabilities.**

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

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

## Time Slicing

- **Definition:** Time slicing divides 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 and enabling concurrent execution.

## Context Switching

- **Definition:** Context switching 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 a process or thread's time slice expires, the OS scheduler performs a context switch to move the CPU's focus to another process or thread.
- **Purpose:** This allows multiple processes and threads to share the CPU, giving the appearance of simultaneous execution on a single-core CPU or improving parallelism on multi-core CPUs.

**Multitasking 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**

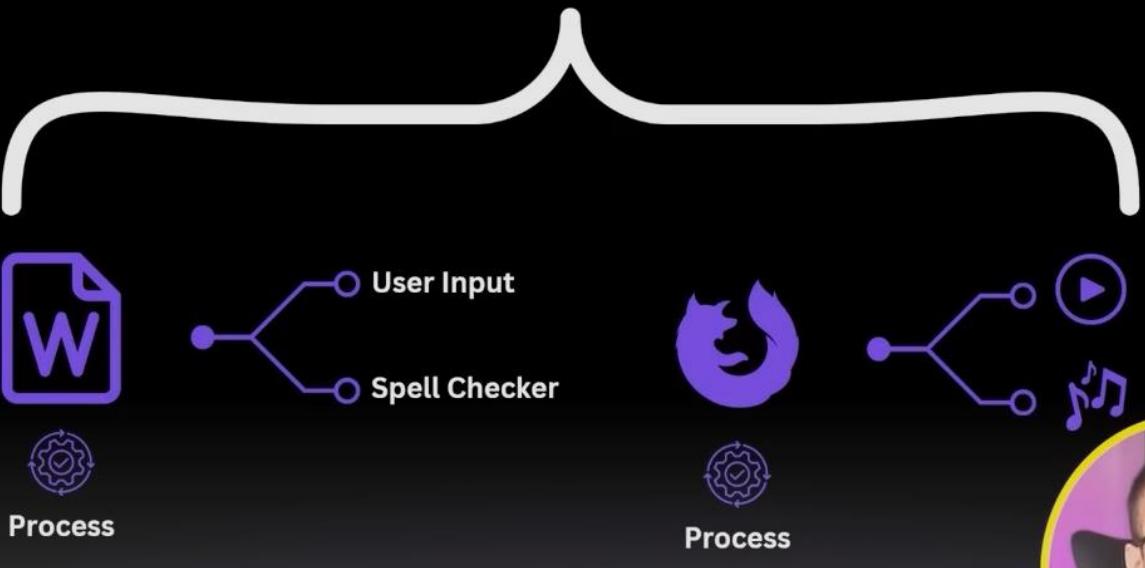
**Multitasking involves managing resources between completely separate programs, which may have independent memory spaces and system resources.**

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

**Multitasking allows us to run multiple applications simultaneously, improving productivity and system utilization.**

**Multithreading allows a single application to perform multiple tasks at the same time, improving application performance and responsiveness.**

## Multitasking ( Managed by OS)



In Java, multithreading is the concurrent 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.

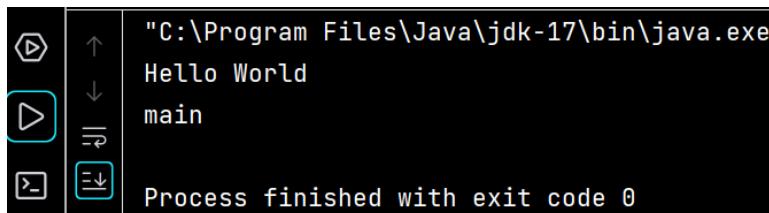
In a multi-core environment, Java's multithreading can take full advantage of the available cores.

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.**

```
public static void main(String[] args) {  
    System.out.println("Hello World");  
    System.out.println(Thread.currentThread().getName()); // main  
}
```



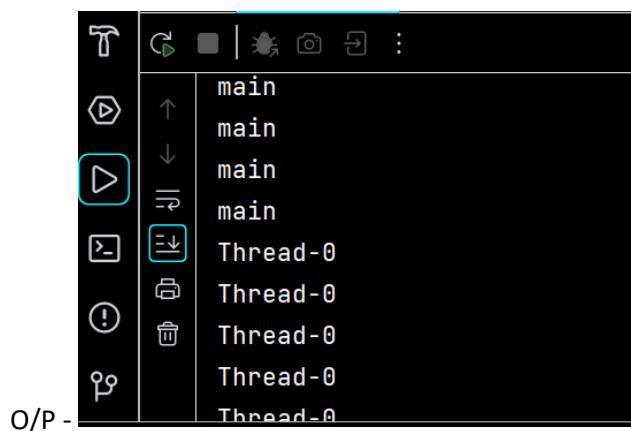
```
"C:\Program Files\Java\jdk-17\bin\java.exe  
Hello World  
main  
Process finished with exit code 0
```

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

- A new class `World` is created that extends `Thread`.
- The `run` method is overridden to define the code that constitutes the new thread.
- `start` method is called to initiate the new thread

```
public static void main(String[] args) {  
    World world = new World();  
    world.start();  
  
    while(true){  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

```
public class World extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
```



```
main
main
main
main
Thread-0
Thread-0
Thread-0
Thread-0
Thread-0
```

### Method 2 –

```
public static void main(String[] args) {
    World world = new World();
    Thread t1 = new Thread(world);
    t1.start();

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

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

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



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

### Thread Lifecycle

**New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run.

**Runnable State:** In this state, a thread might be ready to run at any instant of time or actually running.

**Blocked:** The thread will be in blocked state **when it is trying to acquire a lock** but currently the lock is acquired by the other thread.

**Waiting state:** The thread will be in waiting state when it calls `wait()` method or `join()` method. It will move to the runnable state when other thread will notify or that thread will be terminated.

**Timed Waiting:** A thread lies in a timed waiting state when it **calls a method with a time-out parameter**. A thread lies in this state until the timeout is completed. Ex- `Thread.sleep()`

**Terminated :** The thread has completed execution.

```

public class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("t1 is running");
        try {
            Thread.sleep(millis: 2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(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(millis: 100);
        System.out.println(t1.getState()); // TIMED_WAITING

        t1.join(); // main method will wait for t1 to get finished
        System.out.println(t1.getState()); // TERMINATED
    }
}

```

### Thread v/s Runnable

Suppose class A already extends class B in that case we cannot extend it by Thread class also because multiple inheritance is not allowed, in this case class A should implement Runnable interface and override Run() method.

### Threads

```

public class MyThread extends Thread{
    @Override
    public void run() {
        try {
            Thread.sleep(millis: 5000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();
        t1.join();
        System.out.println("Hello"); // will get printed after 5 sec
    }
}

```

Join() will make main thread to wait for t1 thread to complete its execution

### Methods –

```
public class MyThread extends Thread{
    public MyThread(String name){
        super(name);
    }
    @Override
    public void run() {
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName() +
                " - Priority: " + Thread.currentThread().getPriority() +
                " - count: " + i);
            try {
                Thread.sleep(millis: 100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    MyThread l = new MyThread(name: "low priority thread");
    MyThread m = new MyThread(name: "medium priority thread");
    MyThread h = new MyThread(name: "high priority thread");
    l.setPriority(Thread.MIN_PRIORITY);
    m.setPriority(Thread.NORM_PRIORITY);
    h.setPriority(Thread.MAX_PRIORITY);

    l.start();
    m.start();
    h.start();
}
```

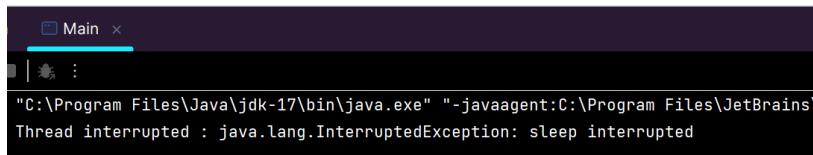
In constructor we can pass name of thread.

Using **setPriority()** we give hints to JVM about relative importance of threads, it is not strictly followed. Maybe thread with low priority execute earlier than thread with medium priority.

```
class MyThread extends Thread{
    @Override
    public void run(){
        try {
            Thread.sleep(millis: 1000);
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted : " + e);
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

```
t1.interrupt();
```



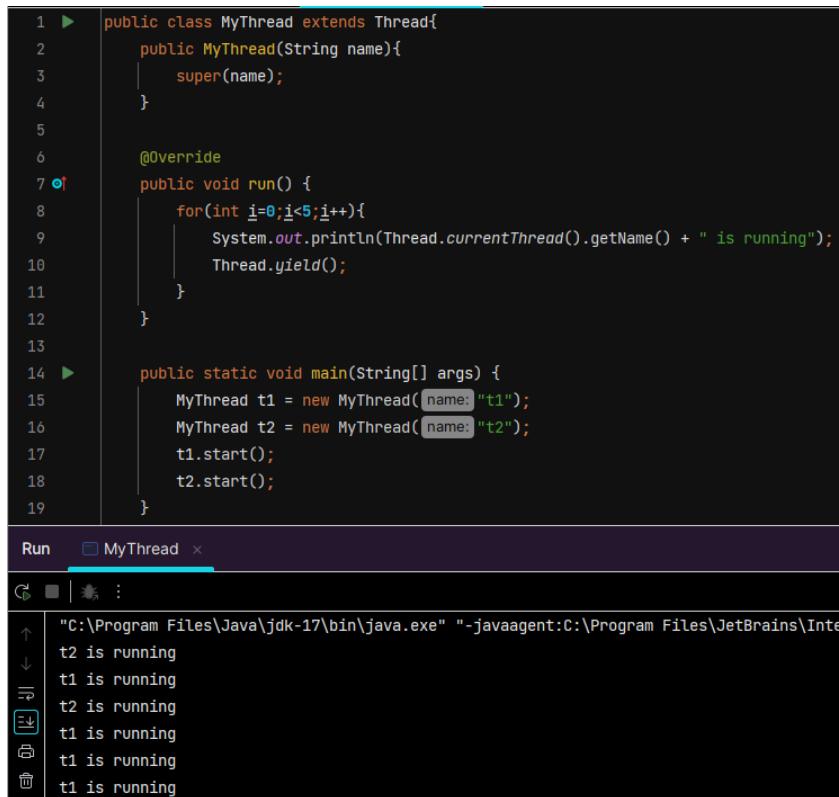
```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
Thread interrupted : java.lang.InterruptedException: sleep interrupted
```

### 🧠 What happens when you call `thread.interrupt()`?

1. If the thread is sleeping, waiting, or joining, it is interrupted immediately, and it throws an `InterruptedException`.
2. If the thread is running normally, it just sets the `interrupted` flag, which can be checked using:

```
java
Thread.currentThread().isInterrupted()
Copy Edit
```

**Yield()** - Used to hint to the thread scheduler that the current thread is **willing to pause** its execution and **current use of processor** to let other threads run. The scheduler is free to ignore this hint.



```
public class MyThread extends Thread{
    public MyThread(String name){
        super(name);
    }

    @Override
    public void run() {
        for(int i=0;i<5;i++){
            System.out.println(Thread.currentThread().getName() + " is running");
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread("t1");
        MyThread t2 = new MyThread("t2");
        t1.start();
        t2.start();
    }
}

Run MyThread
↑ ↓ → ← ⌂ ⌂ ⌂ ⌂ ⌂
" C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
t2 is running
t1 is running
```

**User Thread** – Any **normal thread** that performs the **main work of the application**. Whatever work we are performing inside `run()` method are done with help of user threads.

The **JVM will continue running** as long as there is at least **one user thread** still active — it **waits for all user threads to finish** before shutting down.

**Daemon Thread** – they run in background. Ex- garbage collector in Java.

**JVM does not wait** for them to finish before shutting down. As soon as main thread, all other user threads are done, daemon threads are also terminated and JVM terminates.

## Why?

Because daemon threads are meant to support user threads — they should not prevent the application from exiting. They're like assistants: once the main work (user threads) is done, the assistants can also leave.

```

public class MyThread extends Thread{
    @Override
    public void run() {
        while(true){
            System.out.println("Hello world !!");
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setDaemon(true);
        t1.start();
        MyThread userThread = new MyThread();
        userThread.start();
        System.out.println("main done");
    }
}

```

In this case both threads keep running and JVM will wait for userThread to finish.

## Synchronization

```

public class Counter{
    private int count = 0;
    public void increment(){
        count++;
    }
    public int getCount(){
        return count;
    }
}

```

```

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.increment();
        }
    }
}

```

```

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(); // finish both
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(counter.getCount()); // 1845, but we expect 2000
        // becoz both threads run simultaneously and both share common object
        // suppose count = 101 now both threads read at same time and increment
        // then count becomes 102

        // To make sure that at a particular instance only 1 thread can access the method
        // we make that method as synchronized, this will make other threads to wait
    }
}

```

## Why use `synchronized`?

When multiple threads access and modify the **same data**, race conditions or inconsistent results can occur.  
`synchronized` ensures that **only one thread at a time** can execute a block or method that is marked as synchronized.

**Synchronized instance methods** ➔ Only one thread per object instance can execute this method at a time. Monitor lock of that particular instance is acquired.

```
public class Counter{  
    private int count = 0;  
    public synchronized void increment(){  
        count++;  
    }  
    public int getCount(){  
        return count;  
    }  
}
```

Now O/P – 2000

Instead of whole method we can make a particular **block synchronized** –

```
public class Counter{  
    private int count = 0;  
    public void increment(){  
        synchronized (this){ // "this" signifies that if multiple threads try to access  
                           // same instance of class, then only 1 thread should be  
                           // allowed to access this block  
                           // if we have multiple objects both will run independently  
            count++;  
        }  
    }  
    public int getCount(){  
        return count;  
    }  
}
```

Both **synchronized(ClassName.class)** and a **static synchronized method** use the class-level lock – locks class object.

## What is a **Class Object** in Java?

Every class in Java is represented internally by an object of type `java.lang.Class`, called the **class object**.

You can access it via:

java

 Copy  Edit

`ClassName.class`

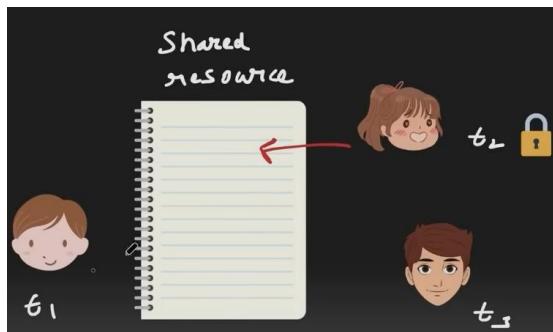
**synchronized(object)** ➔ acquires intrinsic lock of that object

**Critical section** – Part of our code where shared resources are accessed and modified. Ex-  
increment() method in above example.

**Race Condition** – when multiple threads are working on shared resources and if their timing conflicts it can lead to unexpected results.

**Mutual Exclusion** – condition when only 1 thread can access the critical section

**Locks** – after using synchronized keyword on method we achieve mutual exclusion it is done by locking. When 1 thread is working on shared resource it acquires the lock and when it is done, it releases the lock.



The diagram illustrates two types of locks. On the left, under the heading "1. Intrinsic", there is a large "Intrinsic" lock symbol (a circle with a diagonal line). To its right, text explains: "These are built into every object in Java. You don't see them, but they're there. When you use a synchronized keyword, you're using these automatic locks." Below this, under the heading "2. Explicit", there is a large "Explicit" lock symbol (a circle with a vertical line). To its right, text explains: "These are more advanced locks you can control yourself using the Lock class from java.util.concurrent.locks. You explicitly say when to lock and unlock, giving you more control over how and when people can write in the notebook."

An **intrinsic lock** (or monitor lock) is a **built-in synchronization mechanism** that every Java object has.

When you declare a method as `synchronized`:

```
java
public synchronized void someMethod() { ... }
```

Java uses the **intrinsic lock (monitor)** of the **object instance** (i.e., `this`) to control access.

**Problem with Intrinsic Locks (`synchronized`):**

When a thread (e.g., `t1`) enters a `synchronized` method or block:

- It **acquires the intrinsic lock** on the object or class.
- If `t1` enters an **infinite loop** or **never exits** the synchronized block (e.g., hangs, deadlocks, crashes), it **never releases the lock**.
- As a result, **any other threads** (e.g., `t2`, `t3`, ...) trying to acquire that same lock are **blocked forever** — they have **no way to time out or recover**.

You can't interrupt a thread that's waiting on a synchronized block/method.

Explicit locking can solve it.

**Lock** is interface and **ReentrantLock** is its impl class.

**Lock.lock()** → Acquires the lock **if available**, otherwise thread will wait for acquiring lock in BLOCKED state, just like synchronized.

**.tryLock()** → Acquires the lock if it is available and returns immediately with the value true. If the lock is not available then this method will return immediately with the value false. Prevents deadlocks by not waiting forever.

**.tryLock(1000, TimeUnit.MILLISECONDS)** → Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted. If the lock is available this method returns immediately with the value true.

**.unlock** → Releases the lock. Should **always be in a finally block** to ensure it's released.

**Note :-** The reason why Sonar suggests to re-interrupt the thread is because by the time the exception occurs, the thread's interrupted state may have been cleared, so if the exception is not handled properly the fact that the thread was interrupted will be lost.

```
Thread t = new Thread(()->{
    try {
        Thread.sleep(millis: 2000);
    } catch (InterruptedException e) {
        if(Thread.currentThread().isInterrupted()){
            System.out.println("thread is interrupted");
        }else{
            System.out.println("interrupted flag is reset");
        }
    }
});
t.start();
t.interrupt();
```

O/P – interrupted flag is reset

Example :-

```
class BankAccount{
    private int balance = 100;
    private final Lock lock = new ReentrantLock();
    public void withdraw(int amount){
        System.out.println(Thread.currentThread().getName() + " attempt to withdraw");
        try {
            if(lock.tryLock(time: 1000, TimeUnit.MILLISECONDS)){
                try {
                    if (balance >= amount) {
                        System.out.println(Thread.currentThread().getName() + " proceeding with withdrawal");
                        Thread.sleep(millis: 2000);
                        balance -= amount;
                        System.out.println(Thread.currentThread().getName() + " finished withdrawal");
                    }
                } finally {
                    lock.unlock();
                }
            } else{
                System.out.println(Thread.currentThread().getName() + " could not acquire lock");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public static void main(String[] args) {
    BankAccount sbi = new BankAccount();
    Runnable task = new Runnable(){ // anonymous class
        @Override
        public void run() {
            sbi.withdraw(amount: 50);
        }
    };
    Thread t1 = new Thread(task, name: "t1");
    Thread t2 = new Thread(task, name: "t2");
    t1.start();
    t2.start();
}
```

```
t1 attempt to withdraw
t2 attempt to withdraw
t2 proceeding with withdrawal
t1 could not acquire lock
t2 finished withdrawal
```

O/P -

## Deadlock

**Deadlock is a situation in multithreading where two or more threads are blocked forever, waiting for each other to release a resource. This typically occurs when two or more threads have circular dependencies on a set of locks.**

**Deadlocks typically occur when four conditions are met simultaneously:**

- 1. Mutual Exclusion:** Only one thread can access a resource at a time.
- 2. Hold and Wait:** A thread holding at least one resource is waiting to acquire additional resources held by other threads.
- 3. No Preemption:** Resources cannot be forcibly taken from threads holding them.
- 4. Circular Wait:** A set of threads is waiting for each other in a circular chain.



### **How Deadlock Happens with `synchronized`**

#### **💡 Scenario:**

- Thread 1 locks **Object A**, then tries to lock **Object B**
- Thread 2 locks **Object B**, then tries to lock **Object A**

Because `synchronized` is **blocking and non-interruptible**, neither thread can continue once it's stuck — they **wait forever**.

```
Object obj1 = new Object();
Object obj2 = new Object();

Thread t1 = new Thread(()->{
    synchronized (obj1){
        System.out.println("Thread t1 holding obj1 lock");

        System.out.println("Thread t1 wait for obj2 lock");
        synchronized (obj2){
            System.out.println("Thread t1 holding obj1 & obj2 lock");
        }
    }
});

Thread t2 = new Thread(()->{
    synchronized (obj2){
        System.out.println("Thread t2 holding obj2 lock");

        System.out.println("Thread t2 wait for obj1 lock");
        synchronized (obj1){
            System.out.println("Thread t2 holding obj1 & obj2 lock");
        }
    }
});

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

### Example :-

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

class Paper{
    public synchronized void writeWithPenAndPaper(Pen pen){
        System.out.println(Thread.currentThread().getName() + " is using paper " + this + " and trying to write using 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.writeWithPenAndPaper(pen); // thread2 locks paper and tries to lock pen
    }
}

public class Deadlock {
    public static void main(String[] args) {
        Pen pen = new Pen();
        Paper paper = new Paper();
        Thread t1 = new Thread(new Task1(pen, paper), name: "t1");
        Thread t2 = new Thread(new Task2(pen, paper), name: "t2");
        t1.start(); // it will acquire the lock of pen
        t2.start(); // it will acquire the lock of paper
        // when t1 try to call paper.finishWriting(); then it should first get lock of paper which is acquired by t2
        // when t2 try to call pen.finishWriting(); then it should first get lock of pen which is acquired by t1
        // deadlock situation is created
    }
}
```

To avoid deadlock ensure that all the threads acquire locks in consistent order.

```

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() {
        synchronized (pen){ // acquire lock of pen first
            paper.writeWithPenAndPaper(pen); // thread2 locks paper
        }
    }
}

```

This will ensure that t2 will run only when it

locks both pen and paper

### Deadlock Prevention in ReentrantLock

If a thread has acquired a lock, it can acquire the same lock again, without blocking.

Java maintains a **hold count** for reentrant locks:

- When a thread acquires a lock, the **count = 1**
- If it re-enters the lock, **count++**
- It must release the lock the same number of times (count--) for the lock to be fully free

Java's intrinsic locks (used with synchronized) are also reentrant.

```

public class ReentrantExample {
    private final Lock lock = new ReentrantLock();
    public void outerMethod(){
        lock.lock();
        try{
            System.out.println("Outer method");
            innerMethod();
        }finally {
            // lock will be released when every lock() is matched with its unlock()
            lock.unlock();
        }
    }

    private void innerMethod() {
        lock.lock(); // no deadlock as lock is Reentrant
        try {
            System.out.println("Inner method");
        }finally {
            lock.unlock();
            // after above statement, lock is still acquired becoz count is still 1 for lock call
        }
    }

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

```

## Scenario Setup

- A lock is being used to ensure that only one thread accesses a critical section at a time.
- Thread `t1` acquires the lock first and holds it for some time.
- Thread `t2` tries to acquire the lock while `t1` is still holding it.

### 1. Using `lock.lock()`

- `t2` will wait indefinitely until `t1` releases the lock.
- If `t2` is interrupted while waiting, it will ignore the interrupt and keep waiting for the lock.

### 2. Using `lock.lockInterruptibly()`

- `t2` will wait for the lock like before, but it will respond to an interrupt while waiting.
- If `t2` is interrupted, it will stop waiting for the lock and throw an `InterruptedException`.

```
private final static Lock lock = new ReentrantLock();
public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(()->{
        lock.lock();
        try{
            System.out.println("Thread 1: Acquired lock.");
            Thread.sleep(millis: 2000); // stimulate longer task
        } catch (InterruptedException e) { throw new RuntimeException(e); }
        finally {
            lock.unlock();
        }
    });

    Thread t2 = new Thread(()->{
        try {
            lock.lockInterruptibly();
            try {
                System.out.println("Thread 2: Acquired lock.");
            } finally {
                lock.unlock();
            }
        } catch (InterruptedException e) {
            System.out.println("Thread 2: Interrupted while waiting for the lock.");
        }
    });
}

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

// Interrupt t2 after 2 seconds to simulate cancellation
Thread.sleep(millis: 2000);
System.out.println("Main: Interrupting Thread 2...");
t2.interrupt();
}
```

```
Thread 1: Acquired lock.
Main: Interrupting Thread 2...
Thread 2: Interrupted while waiting for the lock.
```

O/P -

## Lock's Fairness

- A fair lock ensures that threads acquire the lock in the order they requested it (i.e., a first-come, first-served basis)
- A thread will not starve (i.e., will not have to wait indefinitely) because threads that arrived earlier in the queue are guaranteed to get the lock before any new threads.
- A non-fair lock (when fairness is set to false) does not guarantee order
- By default value is false.

```
public class FairLockExample {  
    private final Lock lock = new ReentrantLock(fair: true);  
    public void accessResource(){  
        lock.lock();  
        try{  
            System.out.println(Thread.currentThread().getName() + " acquired the lock");  
            Thread.sleep(millis: 1000);  
        }catch (InterruptedException e){  
            Thread.currentThread().interrupt();  
        }finally {  
            System.out.println(Thread.currentThread().getName() + " released the lock");  
            lock.unlock();  
        }  
    }  
  
    public static void main(String[] args) {  
        FairLockExample example = new FairLockExample();  
        Runnable task = new Runnable(){  
            @Override  
            public void run() {  
                example.accessResource();  
            }  
        };  
  
        Thread t1 = new Thread(task, name: "t1");  
        Thread t2 = new Thread(task, name: "t2");  
        Thread t3 = new Thread(task, name: "t3");  
  
        try {  
            t1.start();  
            Thread.sleep(millis: 50);  
            t2.start();  
            Thread.sleep(millis: 50);  
            t3.start();  
        }catch (InterruptedException e){  
        }  
    }  
}
```

Problems with Synchronized –

- No fairness
- indefinitely blocking to acquire lock
- No Interruptibility
- No distinguish between read /write operations – synchronized will block every thread.

**ReadWriteLock** – it allows multiple threads to read resources concurrently as long as no thread is writing to it. It ensures exclusive access for write operations.

```
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++;  
        }finally {  
            writeLock.unlock();  
        }  
    }  
  
    public int getCount(){  
        readLock.lock(); // multiple threads can acquire this lock when  
                      // writeLock has not been acquired by some another thread  
        try{  
            return count;  
        }finally {  
            readLock.unlock();  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        ReadWriteCounter counter = new ReadWriteCounter();  
        Runnable writeTask = new Runnable() {  
            @Override  
            public void run() {  
                counter.increment();  
            }  
        };  
        Runnable readTask = new Runnable() {  
            @Override  
            public void run() {  
                counter.getCount();  
            }  
        };  
  
        Thread writerThread = new Thread(writeTask);  
        Thread readThread1 = new Thread(readTask);  
        Thread readThread2 = new Thread(readTask);  
  
        writerThread.start();  
        readThread1.start();  
        readThread2.start();  
  
        writerThread.join();  
        readThread1.join();  
        readThread2.join();  
  
        System.out.println(counter.getCount()); // 1  
    }  
}
```

## Thread Communication

In a multithreaded environment, threads often need to communicate and coordinate with each other to accomplish a task.

Without proper communication mechanisms, threads might end up in inefficient busy-waiting states, leading to wastage of CPU resources and potential deadlocks.

For thread communication, we have methods like `wait()`, `notify()` and `notifyAll()`. These methods belong to the **Object class**, not Thread.

`Wait()` → tells current thread to release the lock and wait until some other thread run `notify()` or `notifyAll()`

`Notify()` → wakes up a single thread that is waiting

`NotifyAll()` → wakes up all threads that are waiting

These methods can only be called within a synchronized context. When you use methods like `wait()`, `notify()`, or `notifyAll()` on an object, the **calling thread must own that object's intrinsic lock**.

```
object obj = new Object();

synchronized (obj) {    // 🔒 Acquire lock on obj
    obj.wait();          // ✅ Allowed – thread owns the lock
}
```

Ex-

We want Producer to produce data but wait if `hasData` is true. And Consumer should consume data when `hasData` is true.

```
class SharedResource{
    private int data;
    private boolean hasData;
    public synchronized void produce(int value){ // we have acquired lock on object of SharedResource
        while(hasData){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data = value;
        hasData = true;
        System.out.println("Produced : " + value);
        notify(); // it will notify the different thread who is waiting & trying to lock the same object of SharedResource
        // notifyAll(); -----> if for than 1 thread are waiting
    }
    public synchronized int consume(){
        while(!hasData){
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        hasData = false;
        System.out.println("Consumed: " + data);
        notify();
        return data;
    }
}
```

```

class Producer implements Runnable{
    private SharedResource resource;
    public Producer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for(int i=0;i<3;i++) {
            resource.produce(i);
        }
    }
}

class Consumer implements Runnable{
    private SharedResource resource;
    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for(int i=0;i<3;i++) {
            resource.consume();
        }
    }
}

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();
    }
}

```

```

Produced : 0
Consumed: 0
Produced : 1
Consumed: 1
Produced : 2
Consumed: 2

```

O/P - Process finished with exit code 0

**Thread Safety** – a code is thread safe if it guarantee no unexpected results, no race condition when multiple threads try to access that piece of code with same object.

## Lambda Expression – anonymous function

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

Runnable task1 = ()-> System.out.println("Hello");
Thread t1 = new Thread(task1);
Thread t2 = new Thread(()-> System.out.println("Hello"));
```

```
@FunctionalInterface
public interface Runnable {
    When an object implementing interface Runnable is used to create a thread, starting the thread
    causes the object's run method to be called in that separately executing thread.
    The general contract of the method run is that it may take any action whatsoever.
    See Also: Thread.run()

    public abstract void run();
}
```

It does not return.

## Thread Pool – collection of reusable worker threads that are ready to perform a task concurrently.

Why ?

1. Resource management – every time creating and destroying threads is expensive.
2. Response time improves
3. Control over thread count – puts a limit on no of threads that can be created.

```
ExecutorService executor = Executors.newFixedThreadPool(3);
```

This means:

- A maximum of 3 threads will be created and reused.
- Threads are not pre-created at pool creation.
- Each thread is created only when needed, i.e., when a task is submitted and no idle threads are available.

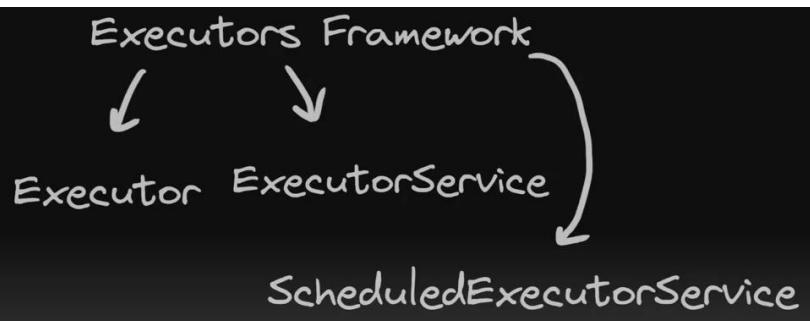
## Executors Framework

### Executors Framework in Java

The Executors framework was introduced in Java 5 as part of the `java.util.concurrent` package to simplify the development of concurrent applications by abstracting away many of the complexities involved in creating and managing threads.

Manual Thread Management  
 Resource Management  
 Scalability  
 Thread Reuse  
 Error Handling

Problems prior to executors framework -



3 interfaces available -

```

long startTime = System.currentTimeMillis(); // 1 Jan 1970
for(int i=1;i<10;i++){
    System.out.println(factorial(i));
}
System.out.println("Total Time : " + (System.currentTimeMillis() - startTime));
  
```

O/P- Total Time : 4575 ms

Using multithreading ➔ O/P – 515 ms

```

long startTime = System.currentTimeMillis(); // 1 Jan 1970
Thread[] threads = new Thread[9];
for(int i=1;i<10;i++){
    int finalI = i;
    threads[i-1] = new Thread(()->{
        long result = factorial(finalI);
        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));
  
```

## Using Executors Framework

```
long startTime = System.currentTimeMillis(); // 1 Jan 1970
ExecutorService executor = Executors.newFixedThreadPool(3);
for(int i=1;i<10;i++){
    int finalI = i;
    executor.submit(()->{
        long result = factorial(finalI);
        System.out.println(result);
    });
}
executor.shutdown(); // Initiates an orderly shutdown in which previously submitted tasks are executed,
// but no new tasks will be accepted.

try {
    executor.awaitTermination(10, TimeUnit.SECONDS);
    // Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs,
    // or the current thread is interrupted, whichever happens first.
    // timeout - we pass maximum time to wait
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
System.out.println("Total Time : " + (System.currentTimeMillis() - startTime));
```

```
try {
    // waits infinitely
    // Returns: true if this executor terminated and false if the timeout elapsed before termination
    while(!executor.awaitTermination(1, TimeUnit.MILLISECONDS)){
        System.out.println("Waiting...");
    }
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
```

**submit()** ➔ it accepts any number of tasks. The task is queued and will execute automatically if a thread from the pool is available.

**shutdown()** ➔ Gracefully shut down the executor service. It does not block the current thread.

**shutdownNow()** ➔ Forcefully stops all running/waiting tasks.

**awaitTermination()** ➔ Returns true if this executor terminated and false if the timeout elapsed before termination

```
public interface ExecutorService extends Executor {
```

```

public interface Executor {

    /**
     * Executes the given command at some time in the future. The command may execute in a new thread,
     * in a pooled thread, or in the calling thread, at the discretion of the Executor implementation.
     *
     * @param command - the runnable task
     *
     * @throws RejectedExecutionException - if this task cannot be accepted for execution
     * @throws NullPointerException - if command is null
     */
    void execute(@NotNull Runnable command);
}

```

Executor does not contain methods like shutdown() so you need to manually do it. That's why its useless. We'll use ExecutorService because it provides more features.

**Submit()** method returns **Future<T>** which represents the result of an asynchronous computation.

**future.get()** blocks the current thread until task completed, or thread interrupt or exception thrown.

```

ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> 42);
System.out.println(future.get()); // get() waits if necessary for the computation to complete,
                                // and then retrieves its result.

if(future.isDone()){ // Returns true if this task completed. Completion may be due to
    // normal termination, exception, or cancellation -- in all of these cases, this method will return true.
    System.out.println("Task is done");
}
executorService.shutdown();

```

O/P -

42  
 Task is done

Submit() can take Runnable as well as Callable –

<b>@NotNull Callable&lt;Integer&gt; task</b>
<b>@NotNull Runnable task, Integer result</b>
<b>@NotNull Runnable task</b>

```

@FunctionalInterface
public interface Callable<V> {

    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     *
     * @throws Exception - if unable to compute a result
     */
    V call() throws Exception;
}

```

Callable returns value.

```

ExecutorService executorService = Executors.newSingleThreadExecutor();
Callable<String> callable = ()->"Hello";
Future<String> future = executorService.submit(callable);
System.out.println(future.get()); // Hello
executorService.shutdown();

```

submit() method of Runnable argument is called -

```

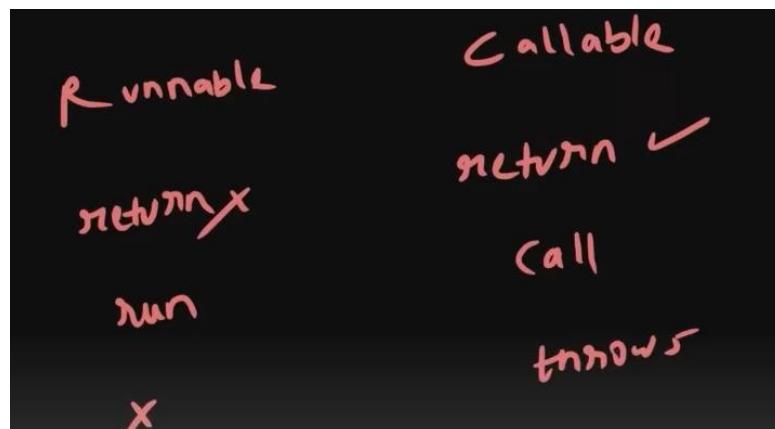
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<?> future = executorService.submit(()-> System.out.println("Hello"));
System.out.println(future.get()); // null
executorService.shutdown();

```

```

ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String> future = executorService.submit(()-> System.out.println("Hello"), result: "success");
//Submits a Runnable task for execution and returns a Future representing that task.
// The Future's get method will return the given result upon successful completion.

```



```

ExecutorService executor = Executors.newFixedThreadPool(nThreads: 2);
Future<Integer> future = executor.submit(() -> 2);
Integer i = future.get(); // main thread will wait tp get its result
System.out.println(i);

executor.shutdown();
System.out.println(executor.isShutdown()); // true
Thread.sleep(millis: 10); // give some time for termination
System.out.println(executor.isTerminated()); // true

```

If tasks are **still running**, but shutdown() has already been called → isShutdown() returns true.

**isTerminated()** → return true if all tasks have completed after shut down

```

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2 );
List<Callable<Integer>> list = Arrays.asList(()->{
    System.out.println("Task 1");
    return 1;
}, ()->{
    System.out.println("Task 2");
    return 2;
}, ()->{
    System.out.println("Task 3");
    return 3;
});
List<Future<Integer>> futures = executorService.invokeAll(list);
// Executes the given tasks, returning a list of Futures holding their status and results when all complete
// it blocks the main thread and wait for completion of all tasks
for(Future<Integer> f:futures){
    System.out.println(f.get());
}
executorService.shutdown();
System.out.println("Hello");

```

invokeAll() → It submits all tasks and then waits (blocks) until all of them finish.

O/P →

Task 2
Task 1
Task 3
1
2
3
Hello

```

ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2 );
List<Callable<Integer>> list = Arrays.asList(()->{
    Thread.sleep( millis: 1000 );
    System.out.println("Task 1");
    return 1;
}, ()->{
    Thread.sleep( millis: 1000 );
    System.out.println("Task 2");
    return 2;
}, ()->{
    Thread.sleep( millis: 2000 );
    System.out.println("Task 3");
    return 3;
});
List<Future<Integer>> futures = executorService.invokeAll(list, timeout: 1, TimeUnit.SECONDS);

```

O/P →

Task 1
Task 2

When we pass timeout in constructor, then it executes all the task which can be executed within time frame and other tasks are cancelled.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
List<Callable<Integer>> list = Arrays.asList(()->1, ()->2, ()->3);
Integer i = executorService.invokeAny(list); //Executes the given tasks, returning the result of one that has
                                              // completed successfully (i.e., without throwing an exception), if any do
System.out.println(i); // 1
executorService.shutdown();
```

invokeAny() also blocks the current thread.

future

- f. get()
- f. isDone()
- f. isCancelled()
- f. get( time )

The screenshot shows an IDE interface with a code editor and a terminal window. The code in the editor is:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000 );
    } catch (InterruptedException e) { }
    return 42;
});
Integer i = null;
try{
    i = future.get( timeout: 1, TimeUnit.SECONDS );
    System.out.println(future.isDone());
    System.out.println(i);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    System.out.println("Exception occurred: " + e);
}
```

The terminal window below shows the output of running the code:

```
C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED
Exception occurred: java.util.concurrent.TimeoutException
```

Explanation :- It waits for 1 sec, if task get completed then fine otherwise it gives TimeoutException.

**isDone()** ➔ Returns true if this task completed. Completion may be due to normal termination, an exception, or cancellation.

**cancel()** ➔ If the task has already completed it cannot be cancelled and **.cancel()** returns false.

- ⇒ If task is still running it can be cancelled and **.cancel()** returns true.
- ⇒ **mayInterruptIfRunning** parameter determines whether to run the complete task or not.
- ⇒ Future will not return any value and we get **CancellationException** if we use **.get()**

**isCancelled()** ➔ Returns true if this task was cancelled before it completed normally.

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000 );
    } catch (InterruptedException e) { }
    return 42;
});
future.cancel( mayInterruptIfRunning: true );
System.out.println(future.isCancelled());
System.out.println(future.isDone());
executorService.shutdown();
```

O/P →

```
true  
true
```

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<Integer> future = executorService.submit(() -> {
    try {
        Thread.sleep( millis: 2000 );
    } catch (InterruptedException e) {

    }
    System.out.println("Hello");
    return 23;
});
Thread.sleep( millis: 100 );
System.out.println(future.cancel( mayInterruptIfRunning: false )); // task is not interrupted, it will still run
System.out.println(future.isCancelled());
System.out.println(future.isDone());
executorService.shutdown();
```

O/P →

```
true  
true  
true  
Hello
```

**Scheduled Executor Service** – used when we want something to run periodically.

```
public interface ScheduledExecutorService extends ExecutorService {  
  
    ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(corePoolSize: 1);  
    scheduler.schedule(  
        () -> System.out.println("Task executed after 5 second delay"),  
        delay: 5,  
        TimeUnit.SECONDS);  
    scheduler.shutdown();
```

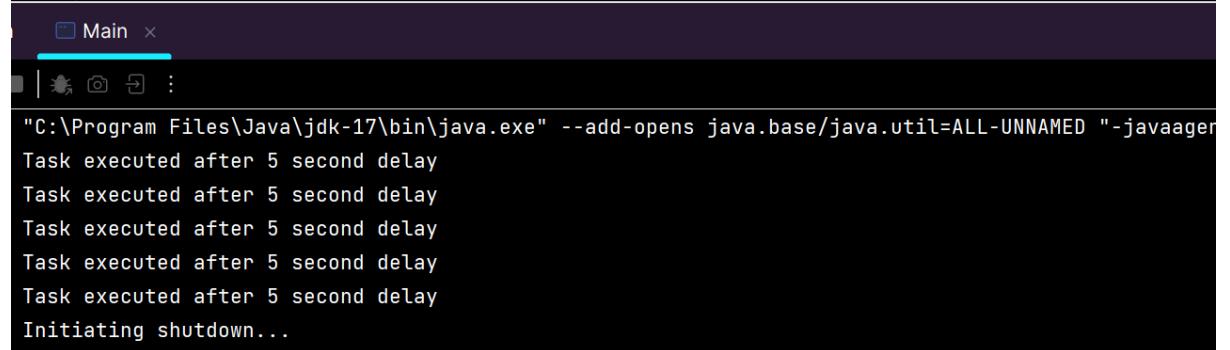
O/P → Task executed after 5 second delay

**.schedule()** → Submits a one-shot task that becomes enabled after the given delay

**.scheduleAtFixedRate()** → Submits a periodic task that becomes enabled first after the given initial delay, and subsequently with the given period; that is, executions will commence after initialDelay, then initialDelay + period, then initialDelay + 2 \* period, and so on.

If `scheduler.shutdown()` is called **before the initial delay expires**, the scheduler will terminate immediately, and the task will never run. So we need to handle it properly –

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(corePoolSize: 1);  
scheduler.scheduleAtFixedRate(  
    () -> System.out.println("Task executed after 5 second delay"),  
    initialDelay: 0,  
    period: 5,  
    TimeUnit.SECONDS);  
scheduler.schedule(() ->{  
    System.out.println("Initiating shutdown...");  
    scheduler.shutdown();  
}, delay: 20, TimeUnit.SECONDS);  
}
```



Main

```
C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-javaagent  
Task executed after 5 second delay  
Initiating shutdown...
```

**scheduleWithFixedDelay()** → Submits a periodic action that becomes enabled first after the given initial delay, and subsequently with the given delay between the termination of one execution and the commencement of the next.

```

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(corePoolSize: 1);
ScheduledFuture<?> scheduledFuture = scheduler.scheduleWithFixedDelay(
    () -> System.out.println("Task executed after 5 second delay"),
    initialDelay: 0,
    delay: 5, // how much time to wait after completing 1 task to start the next task
    TimeUnit.SECONDS);

scheduler.schedule(()->{
    System.out.println("Initiating shutdown...");
    scheduler.shutdown();
}, delay: 20, TimeUnit.SECONDS);

```

O/P →

```

Task executed after 5 second delay
Initiating shutdown....

```

**Cached Thread Pool –** It creates a thread pool where it creates new threads as per need and terminates them after 60 seconds of inactivity. It dynamically adjusts the pool size. It is useful when load is variable and short lived. Do not use it when task are very time consuming.

```

ExecutorService executorService = Executors.newCachedThreadPool();

```

**CountDownLatch –** It allows one or more threads to wait for a set of operations being performed by other threads to complete before proceeding further.

**count** – the number of times countDown must be invoked before threads can pass through await

Ex- we want to achieve that our main thread wait for some other dependent services.

```

class DependentService implements Callable<String>{
    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " service started");
        Thread.sleep(millis: 2000);
        return "ok";
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(nThreads: 3);
        Future<String> future1 = executorService.submit(new DependentService());
        Future<String> future2 = executorService.submit(new DependentService());
        Future<String> future3 = executorService.submit(new DependentService());
        future1.get();
        future2.get();
        future3.get();
        System.out.println("All dependent services finished. Starting main service.....");
        executorService.shutdown();
    }
}

```

Without –

```

class DependentService implements Callable<String>{
    CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public String call() throws Exception {
        try {
            System.out.println(Thread.currentThread().getName() + " service started");
            Thread.sleep(millis: 2000);
        }finally {
            latch.countDown();
        }
        return "ok";
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
        CountDownLatch latch = new CountDownLatch(numberOfServices);
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        latch.await();

        System.out.println("All dependent services finished. Starting main service.....");
        executorService.shutdown();
    }
}

```

**With CountDownLatch -**

```

pool-1-thread-1 service started
pool-1-thread-2 service started
pool-1-thread-3 service started
All dependent services finished. Starting main service.....
```

**Using threads –**

```

class DependentService implements Runnable{
    CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " service started");
            Thread.sleep(millis: 2000);
        }catch (InterruptedException e) { }
        finally {
            latch.countDown();
        }
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        CountDownLatch latch = new CountDownLatch(numberOfServices);
        for(int i=0;i<numberOfServices;i++){
            new Thread(new DependentService(latch)).start();
        }
        latch.await();
        System.out.println("All dependent services finished. Starting main service.....");
    }
}

```

```
O/P → Thread-2 service started  
Thread-1 service started  
Thread-0 service started  
All dependent services finished. Starting main service.....
```

**await()** ➔ Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.

We can also pass timeout in await()

```
class DependentService implements Runnable{  
    CountDownLatch latch;  
    public DependentService(CountDownLatch latch) {  
        this.latch = latch;  
    }  
    @Override  
    public void run(){  
        try {  
            Thread.sleep(millis: 6000);  
            System.out.println(Thread.currentThread().getName() + " service started");  
        }catch (InterruptedException e) {}  
        finally {  
            latch.countDown();  
        }  
    }  
}  
public class Main {  
    public static void main(String[] args) throws InterruptedException, ExecutionException {  
        int numberofServices = 3;  
        CountDownLatch latch = new CountDownLatch(numberofServices);  
        for(int i=0;i<3;i++){  
            new Thread(new DependentService(latch)).start();  
        }  
        latch.await(timeout: 5, TimeUnit.SECONDS); // wait for 5 seconds and start executing main thread  
  
        System.out.println("Main thread executing....");  
    }  
}
```

```
O/P → Main thread executing....  
Thread-1 service started  
Thread-2 service started  
Thread-0 service started
```

```

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

    int numberOfServices = 3;
    ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
    CountDownLatch latch = new CountDownLatch(numberOfServices);
    executorService.submit(new DependentService(latch));
    executorService.submit(new DependentService(latch));
    executorService.submit(new DependentService(latch));
    latch.await(timeout: 5, TimeUnit.SECONDS);
    System.out.println("Main");
    executorService.shutdownNow();

```

.shutdownNow() will terminate the worker threads immediately.

CyclicBarrier – It is reusable

CountDownLatch is not reusable, once the count reaches 0 it cannot be reset.

In cyclic barrier, multiple threads waits at barrier until all threads reached at barrier. When the last thread reaches the barrier then all threads are released. It does not block main thread. It is used to make sure that certain no of threads reach at a particular point before any of them proceed.

It automatically resets itself after all participating threads have reached the barrier.

```

class DependentService implements Callable<String>{
    CyclicBarrier barrier;
    public DependentService(CyclicBarrier barrier) {
        this.barrier = barrier;
    }
    @Override
    public String call() throws Exception {
        System.out.println(Thread.currentThread().getName() + " service started");
        Thread.sleep(millis: 1000);
        System.out.println(Thread.currentThread().getName() + " is waiting at the barrier");

        barrier.await(); // all worker threads wait here.
        return "ok";
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        int numberOfServices = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfServices);
        CyclicBarrier barrier = new CyclicBarrier(numberOfServices);
        executorService.submit(new DependentService(barrier));
        executorService.submit(new DependentService(barrier));
        executorService.submit(new DependentService(barrier));
        System.out.println("Main executing.....");

        System.out.println(barrier.getParties()); // 3
        barrier.reset(); // reset it
        executorService.shutdown();
    }
}

```

```
Main executing.....  
3  
pool-1-thread-2 service started  
pool-1-thread-3 service started  
pool-1-thread-1 service started  
pool-1-thread-2 is waiting at the barrier  
pool-1-thread-1 is waiting at the barrier  
pool-1-thread-3 is waiting at the barrier
```

**await()** ➔ Waits until all parties have invoked await on this barrier.

**getParties()** ➔ Returns the number of parties required to trip this barrier.

**reset()** ➔ Resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.

Ex- image a big application having multiple subsystem

```
class Subsystem implements Runnable{  
    private String name;  
    private long initializationTime;  
    private CyclicBarrier barrier;  
    public Subsystem(String name, long initializationTime, CyclicBarrier barrier) {  
        this.name = name;  
        this.initializationTime = initializationTime;  
        this.barrier = barrier;  
    }  
    @Override  
    public void run() {  
        try{  
            System.out.println(name + " initialization started");  
            Thread.sleep(initializationTime); // simulate time taken to initialize  
            System.out.println(name + " initialization completed");  
            barrier.await();  
        }catch (Exception e){ }  
    }  
}  
public class Main {  
    public static void main(String[] args) throws InterruptedException, ExecutionException {  
        int numberOfSubsystems = 4;  
        CyclicBarrier barrier = new CyclicBarrier(numberOfSubsystems, new Runnable() {  
            @Override  
            public void run() {  
                System.out.println("All subsystems are up and running. System startup complete.");  
            }  
        }); // when all the waiting threads reach the barrier, then this barrier action is executed  
        // performed by the last thread entering the barrier  
  
        Thread webServerThread = new Thread(new Subsystem( name: "Web Server", initializationTime: 2000, barrier));  
        Thread dbThread = new Thread(new Subsystem( name: "DB", initializationTime: 4000, barrier));  
        Thread cacheThread = new Thread(new Subsystem( name: "Cache", initializationTime: 3000, barrier));  
        Thread messagingServiceThread = new Thread(new Subsystem( name: "Messaging Service", initializationTime: 3500, barrier));  
        webServerThread.start();  
        dbThread.start();  
        cacheThread.start();  
        messagingServiceThread.start();  
    }  
}
```

```

Messaging Service initialization started
Cache initialization started
DB initialization started
Web Server initialization started
Web Server initialization completed
Cache initialization completed
Messaging Service initialization completed
DB initialization completed
All subsystems are up and running. System startup complete.

O/P →

```

### CompletableFuture

Introduced in JAVA 8 to handle asynchronous programming

```

CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});
System.out.println("Main");

```

```

O/P → Main

```

we don't see sout of "Worker" because by default, CompletableFuture.supplyAsync() uses the **ForkJoinPool.commonPool()**, and the threads in this pool are **daemon threads**.

If we want to run the task –

```

CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});
String s = completableFuture.get(); // Waits if necessary for this future to complete,
                                    // and then returns its result.
System.out.println(s);
System.out.println("Main");

```

```

Worker
ok
Main
O/P →

```

```

CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
});

String s = completableFuture.getNow( valueIfAbsent: "noo" ); // Returns the result value if completed,
                                                               // else returns the given valueIfAbsent.

System.out.println(s);
System.out.println("Main");

```

O/P →   
 noo  
 Main

```

CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "OK";
});

CompletableFuture<String> f2 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "OK";
});

CompletableFuture<Void> f = CompletableFuture.allOf(f1, f2); // Returns a new CompletableFuture that is completed
                                                               // when all the given CompletableFutures complete
f.join(); // blocking method that makes the calling thread wait until the CompletableFuture (f) completes.
System.out.println("Main");

```

Main x

Worker  
Worker  
Main

```

String f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}).get();
System.out.println("Main");
}

```

Main x

"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-javaagent:C:\Program Files\JetBrains\

Worker  
Main

```
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 2000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}).thenApply(x->x.toUpperCase());
System.out.println(f1.get());
System.out.println("Main");
}

}

Main x
```

```
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 5000 );
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}).orTimeout( timeout: 2, TimeUnit.SECONDS ).exceptionally(s -> "Timeout occurred");
System.out.println(f1.get());
System.out.println("Main");
}
```

**By default, CompletableFuture tasks often run on daemon threads due to the use of ForkJoinPool.commonPool**

**You can control the thread type by providing a custom executor service.**

The `CompletableFuture` task itself doesn't dictate whether it's a daemon or user thread.

```
ExecutorService executorService = Executors.newFixedThreadPool(nThreads: 3);
CompletableFuture<String> f1 = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(millis: 5000);
        System.out.println("Worker");
    } catch (InterruptedException e) { }
    return "ok";
}, executorService); // now it will use threads from our pool
System.out.println(f1.get());
executorService.shutdown();
System.out.println("Main");
}

Main
```

## Volatile vs Atomic

Volatile → Always **read and write directly from/to main memory.**

```
class SharedResource{
    // if we want to tell Threads that don't cache it and read/manipulate directly the main memory
    // make the variable volatile
    private volatile boolean flag = false;
    public void setFlagTrue() {
        System.out.println("Writer thread made the flag true");
        this.flag = true;
    }
    public void printIfFlagTrue() {
        while(!flag){
            // flag became true after 1000 ms but readerThread is still stuck in infinite loop
            // because each thread keeps local copy of variables in its cache for performance reasons
            // cached value of flag is false
            // problematic because flag is sharedResource
        }
        System.out.println("Flag is true");
    }
}
public class Main {
    public static void main(String[] args){
        SharedResource sharedObj = new SharedResource();
        Thread writerThread = new Thread(()->{
            try {
                Thread.sleep( millis: 1000 );
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            sharedObj.setFlagTrue();
        });
        Thread readerThread = new Thread(()->sharedObj.printIfFlagTrue());
        writerThread.start();
        readerThread.start();
    }
}
```

**Note:** - All fields being read in a synchronized block will be read from main memory.

If we don't make variable volatile then readerThread is stuck in infinite loop. But in complex scenarios it also has disadvantages as there is no thread safety.

AtomicInteger, AtomicLong, AtomicBoolean - does not use locks. It achieves thread safety using **non-blocking, lock-free algorithms** that rely on **low-level hardware support** for atomic operations.

```
public class VolatileCounter {  
    private int counter = 0;  
    public void increment(){  
        counter++;  
    }  
    public int getCounter(){  
        return counter;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        VolatileCounter vc = new VolatileCounter();  
        Thread t1 = new Thread(()->{  
            for(int i=0;i<1000;i++) vc.increment();  
        });  
        Thread t2 = new Thread(()->{  
            for(int i=0;i<1000;i++) vc.increment();  
        });  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println(vc.getCounter()); // value < 2000  
    }  
}
```

```
public class VolatileCounter {  
    // ensure atomicity without using locks  
    private AtomicInteger counter = new AtomicInteger(initialValue: 0);  
    public void increment(){  
        counter.incrementAndGet();  
    }  
    public int getCounter(){  
        return counter.get();  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        VolatileCounter vc = new VolatileCounter();  
        Thread t1 = new Thread(()->{  
            for(int i=0;i<1000;i++) vc.increment();  
        });  
        Thread t2 = new Thread(()->{  
            for(int i=0;i<1000;i++) vc.increment();  
        });  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println(vc.getCounter()); // 2000  
    }  
}
```