## 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.

## Core

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.

## Program

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.

## Process

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.

**Thread**

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**

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**

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.*

## Multithreading in Java

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

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

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

## Method 1: extend the Thread class

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

```
public class Test {
    public static void main(String[] args) {
        World world = new World();
```

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

## Method 2: Implement Runnable interface

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

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

## Thread Lifecycle

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

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

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

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

## Runnable vs Thread

Use Runnable when you want to separate the task from the thread, allowing the class to extend another class if needed. Extend Thread if you need to override Thread methods or if the task inherently requires direct control over the thread itself, though this limits inheritance.

## Thread methods

1. **start( ):** Begins the execution of the thread. The Java Virtual Machine (JVM) calls the `run()` method of the thread.
2. **run( )**: The entry point for the thread. When the thread is started, the `run()` method is invoked. If the thread was created using a class that implements `Runnable`, the `run()` method will execute the `run()` method of that `Runnable` object.
3. **sleep(long millis):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **join( ):** Waits for this thread to die. When one thread calls the `join()` method of another thread, it pauses the execution of the current thread until the thread being joined has completed its execution.
5. **setPriority(int newPriority):** Changes the priority of the thread. The priority is a value between `Thread.MIN_PRIORITY` (1) and `Thread.MAX_PRIORITY` (10).

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

    @Override
    public void run() {
        System.out.println("Thread is Running...");
        for (int i = 1; i <= 5; i++) {
            for (int j = 0; j < 5; j++) {
                System.out.println(Thread.currentThread().getName() + " -
Priority: " + Thread.currentThread().getPriority() + " - count: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();

                }
            }
        }
    }

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

        MyThread l = new MyThread("Low Priority Thread");
        MyThread m = new MyThread("Medium Priority Thread");
        MyThread n = new MyThread("High Priority Thread");
        l.setPriority(Thread.MIN_PRIORITY);
        m.setPriority(Thread.NORM_PRIORITY);
        n.setPriority(Thread.MAX_PRIORITY);
        l.start();
        m.start();
        n.start();

    }
}
```

**6. interrupt():** Interrupts the thread. If the thread is blocked in a call to `wait()`, `sleep()`, or `join()`, it will throw an `InterruptedException`.

7. **yield():** `Thread.yield()` is a static method that suggests the current thread temporarily pause its execution to allow other threads of the same or higher priority to execute. It's important to

note that `yield()` is just a hint to the thread scheduler, and the actual behavior may vary depending on the JVM and OS.

```
public class MyThread extends Thread {
    @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();
        MyThread t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
```

**8. Thread.setDaemon(boolean)**: Marks the thread as either a daemon thread or a user thread. When the JVM exits, all daemon threads are terminated.

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

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.setDaemon(true); // myThread is daemon thread ( like
Garbage collector ) now
        MyThread t1 = new MyThread();
        t1.start(); // t1 is user thread
        myThread.start();
        System.out.println("Main Done");
    }
}
```

## Synchronisation

Let's see an example where two threads are incrementing same couter.

```
class Counter {
    private int count = 0; // shared resource

    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 static void main(String[] args) {
        Counter counter = new Counter();
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        }catch (Exception e){

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

The output of the code is not 2000 because the increment method in the Counter class is not synchronized. This results in a race condition when both threads try to increment the count variable concurrently.

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

We can fix this by using `synchronized` keyword

```
class Counter {
    private int count = 0; // shared resource

    public synchronized void increment() {
        count++;
    }

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

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

## Locks

The `synchronized` keyword in Java provides basic thread-safety but has limitations: it locks the entire method or block, leading to

potential performance issues. It lacks a try-lock mechanism, causing threads to block indefinitely, increasing the risk of deadlocks. Additionally, `synchronized` doesn't support multiple condition variables, offering only a single monitor per object with basic wait/notify mechanisms. In contrast, explicit locks (`Lock` interface) offer finer-grained control, try-lock capabilities to avoid blocking, and more sophisticated thread coordination through multiple condition variables, making them more flexible and powerful for complex concurrency scenarios.

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

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

    public void withdraw(int amount) {
        System.out.println(Thread.currentThread().getName() + " attempting
to withdraw " + amount);
        try {
            if (lock.tryLock(1000, TimeUnit.MILLISECONDS)) {
                if (balance >= amount) {
                    try {

System.out.println(Thread.currentThread().getName() + " proceeding with
withdrawal");
                        Thread.sleep(3000); // Simulate time taken to
process the withdrawal
                        balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed
withdrawal. Remaining balance: " + balance);
                    } catch (Exception e) {
                        Thread.currentThread().interrupt();
                    } finally {
                        lock.unlock();
                    }
                } else {
                    System.out.println(Thread.currentThread().getName() +
" insufficient balance");
                }
            } else {
                System.out.println(Thread.currentThread().getName() + "
could not acquire the lock, will try later");
```

```
            }
        } catch (Exception e) {
            Thread.currentThread().interrupt();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount sbi = new BankAccount();
        Runnable task = new Runnable() {
            @Override
            public void run() {
                sbi.withdraw(50);
            }
        };
        Thread t1 = new Thread(task, "Thread 1");
        Thread t2 = new Thread(task, "Thread 2");
        t1.start();
        t2.start();
    }
}
```

## Reentrant Lock

A Reentrant Lock in Java is a type of lock that allows a thread to acquire the same lock multiple times without causing a deadlock. If a thread already holds the lock, it can re-enter the lock without being blocked. This is useful when a thread needs to repeatedly enter synchronized blocks or methods within the same execution flow. The ReentrantLock class from the `java.util.concurrent.locks` package provides this functionality, offering more flexibility than the `synchronized` keyword, including try-locking, timed locking, and multiple condition variables for advanced thread coordination.

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

    public void outerMethod() {
        lock.lock();
        try {
```

```
            System.out.println("Outer method");
            innerMethod();
        } finally {
            lock.unlock();
        }
    }

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

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

## Methods of ReentrantLock

`lock()`

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

`tryLock()`

- Tries to acquire the lock without waiting. Returns `true` if the lock was acquired, `false` otherwise.

- This is non-blocking, meaning the thread will not wait if the lock is not available.

`tryLock(long timeout, TimeUnit unit)`

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

`unlock()`

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

`lockInterruptibly()`

- Acquires the lock unless the current thread is interrupted. This is useful when you want to handle interruptions while acquiring a lock.

Suresh Sencha

# Read Write Lock

A Read-Write Lock is a concurrency control mechanism that allows multiple threads to read shared data simultaneously while restricting write access to a single thread at a time. This lock type, provided by the `ReentrantReadWriteLock` class in Java, optimizes performance in scenarios with frequent read operations and infrequent writes. Multiple readers can acquire the read lock without blocking each other, but when a thread needs to write, it must acquire the write lock, ensuring exclusive access. This prevents data inconsistency while improving read efficiency compared to traditional locks, which block all access during write operations.

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

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

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

    public int getCount() {
        readLock.lock();
        try {
            return count;
        } finally {
            readLock.unlock();
        }
    }
```

```
    public static void main(String[] args) throws InterruptedException {
        ReadWriteCounter counter = new ReadWriteCounter();

        Runnable readTask = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    System.out.println(Thread.currentThread().getName() +
" read: " + counter.getCount());
                }
            }
        };

        Runnable writeTask = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    counter.increment();
                    System.out.println(Thread.currentThread().getName() +
" incremented");
                }
            }
        };

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

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

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

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

## Fairness of Locks

Fairness in the context of locks refers to the order in which threads acquire a lock. A fair lock ensures that threads acquire the lock in the order they requested it, preventing thread starvation. With a fair lock, if multiple threads are waiting, the longest-waiting thread is granted the lock next. However, fairness can lead to lower throughput due to the overhead of maintaining the order. Non-fair

locks, in contrast, allow threads to “cut in line,” potentially offering better performance but at the risk of some threads waiting indefinitely if others frequently acquire the lock.

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

public class FairnessLockExample {
    private final Lock lock = new ReentrantLock(true);

    public void accessResource() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "
acquired the lock.");
            Thread.sleep(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) {
        FairnessLockExample example = new FairnessLockExample();

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

## Deadlock

A deadlock occurs in concurrent programming when two or more threads are blocked forever, each waiting for the other to release a resource. This typically happens when threads hold locks on resources and request additional locks held by other threads. For example, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1. Since neither thread can proceed, they remain stuck in a deadlock state. Deadlocks can severely impact system performance and are challenging to debug and resolve in multi-threaded applications.

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

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

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

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

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

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

    @Override
```

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

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

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

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


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

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

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

## Thread communication

```
class SharedResource {
    private int data;
    private boolean hasData;

    public synchronized void produce(int value) {
        while (hasData) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
```

```
            }
            data = value;
            hasData = true;
            System.out.println("Produced: " + value);
            notify();
        }

        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 < 10; 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 < 10; i++) {
                int value = resource.consume();
            }
        }
    }

    public class ThreadCommunication {
        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();
    }
}
```

## Executors framework

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.

It will help in

1. Avoiding Manual Thread management
2. Resource management
3. Scalability
4. Thread reuse
5. Error handling

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

public class ExecutorFrameWork {

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        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();
        // executor.shutdown();
        try {
            executor.awaitTermination(1, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

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

    private static long factorial(int n) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        long result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

## Future

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService =
Executors.newSingleThreadExecutor();
        Future<?> future = executorService.submit(() ->
System.out.println("Hello")); // runnable parameter
        System.out.println(future.get()); // blocking call ( null )
        if(future.isDone()){
            System.out.println("Task is done !");
        }
```

```
        executorService.shutdown();
    }
}

==========
Hello
null
Task is done !

===============
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService =
Executors.newSingleThreadExecutor();
        Future<String> future = executorService.submit(() -> "Hello"); //
callable parameter
        System.out.println(future.get()); // blocking call
        if(future.isDone()){
            System.out.println("Task is done !");
        }
        executorService.shutdown();
    }
}
=================
Hello
Task is done !
```

## Atomic classes

```
public class VolatileCounter {
    private AtomicInteger counter = new AtomicInteger(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());
    }

}
```

## Volatile keyword

```
class SharedObj {
    private volatile boolean flag = false;

    public void setFlagTrue() {
        System.out.println("Writer thread made the flag true !");
        flag = true;
    }

    public void printIfFlagTrue() {
        while (!flag) {
            // do nothing
        }
        System.out.println("Flag is true !");
    }
}

public class VolatileExample {
    public static void main(String[] args) {
        SharedObj sharedObj = new SharedObj();

        Thread writerThread = new Thread(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            sharedObj.setFlagTrue();
        });

        Thread readerThread = new Thread(() ->
sharedObj.printIfFlagTrue());
```

```
        writerThread.start();
        readerThread.start();
    }
}
```

## CountDownLatch

```
import java.util.concurrent.Callable;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test {
    public static void main(String[] args) throws InterruptedException {
        int n = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(n);
        CountDownLatch latch = new CountDownLatch(n);
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        latch.await();
        System.out.println("Main");
        executorService.shutdown();
    }
}

class DependentService implements Callable<String> {
    private final 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(2000);
        } finally {
            latch.countDown();
        }
        return "ok";
    }
}
==================================
pool-1-thread-3 service started.
pool-1-thread-2 service started.
pool-1-thread-1 service started.
Main
```

# Cyclic Barrier

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Main {

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

        Thread webServerThread = new Thread(new Subsystem("Web Server",
2000, barrier));
        Thread databaseThread = new Thread(new Subsystem("Database", 4000,
barrier));
        Thread cacheThread = new Thread(new Subsystem("Cache", 3000,
barrier));
        Thread messagingServiceThread = new Thread(new
Subsystem("Messaging Service", 3500, barrier));

        webServerThread.start();
        databaseThread.start();
        cacheThread.start();
        messagingServiceThread.start();

    }


}

class Subsystem implements Runnable {
    private String name;
    private int initializationTime;
    private CyclicBarrier barrier;

    public Subsystem(String name, int 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 complete.");
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
```