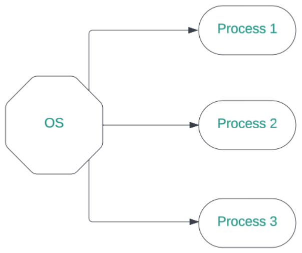
Threads

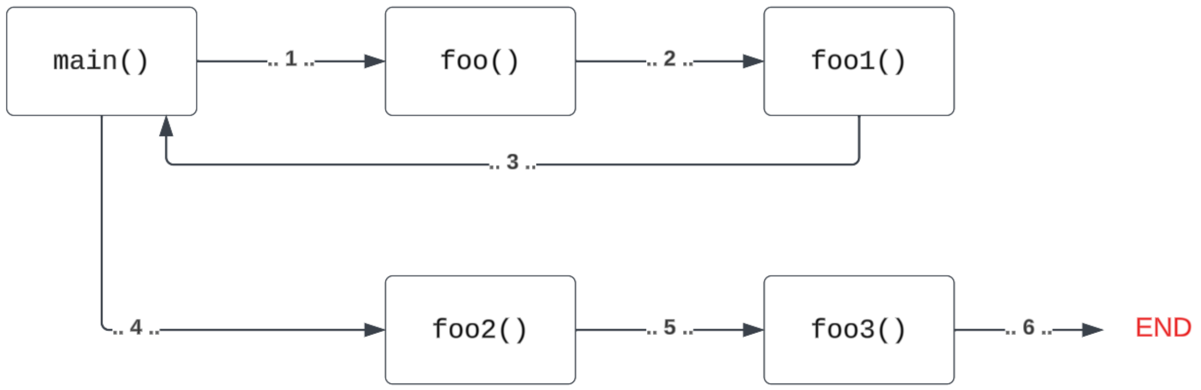
One of the techniques to achieve multi-tasking by a computer processor is threading. Another process is multiprocessing.

In the realm of computer science, **multithreading** is a widespread technique that allows a single process to manage multiple tasks concurrently. This powerful method, a subset of multitasking, enables a processor to execute multiple threads within a single process space, thereby enhancing computational efficiency and allowing for parallelism on a micro-scale. Alongside **multiprocessing**, another form of multitasking where multiple processors execute multiple tasks, multithreading plays a pivotal role in optimizing system performance and resource utilization. This article delves into the intricate world of multithreading, exploring its mechanisms, benefits, and applications in modern computing.



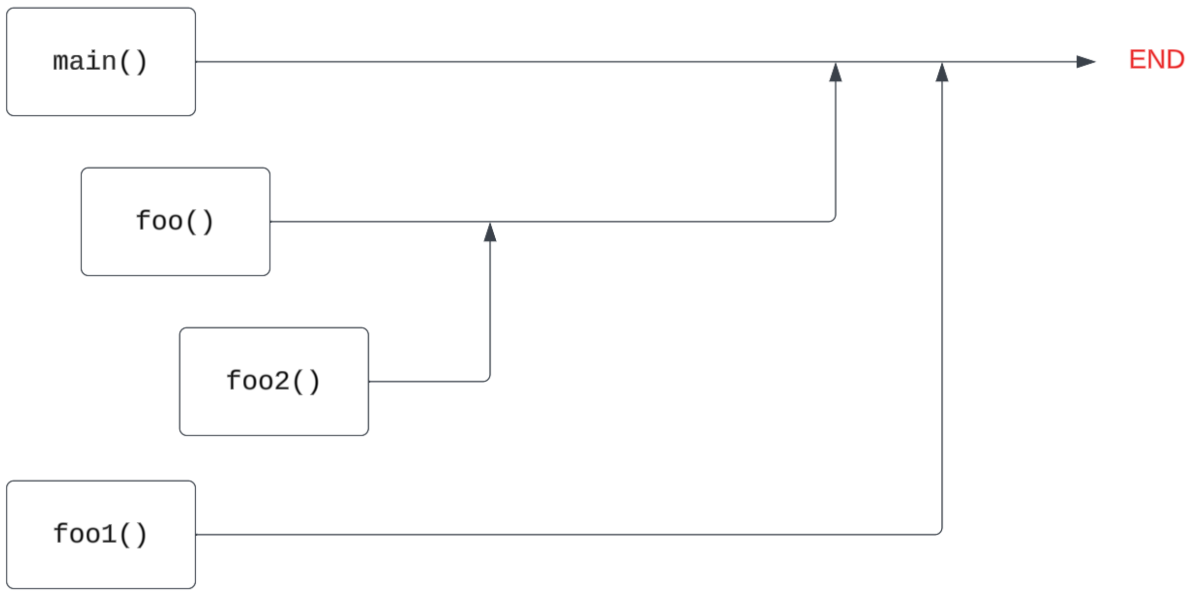
In this example, OS is handling multiple processes at once. Here processes 1, 2, and 3 are running parallel.

Program Flow Without Threading



The numbers on the line represent the timestamp when the functions are called sequentially.

Program Flow With Threading



In this diagram, we see that all the functions started simultaneously with the main function. And when their process is finished they return to the main process line. Threads are independent of anything happening in other threads.

General Overview on Thread

Shared Memory Area for Execution

Threads are lightweight processes that exist within the process, sharing the same memory space. This shared memory model allows threads to access the same data structures and variables, enabling efficient communication between them. However, care must be taken to avoid problems such as race conditions, which can occur due to concurrent access to shared resources.

Fast Data Switching

Threads have the advantage of being able to switch contexts faster than processes because they share the same memory space. This means that the time taken to switch between threads (also known as context switching) is significantly less than the time taken to switch between processes. This results in a high degree of responsiveness and efficiency in multi-threaded applications.

Independence of Threads

In a multi-threaded environment, no thread is dependent on another for execution. Each thread operates independently and executes concurrently with other threads. This means that if one thread is blocked or is performing a lengthy operation, other threads can continue executing. This is known as **asynchronous** or **non-blocking** execution and is a key feature of multi-threading that contributes to better utilization of CPU resources and improved application responsiveness. However, it's important to note that while threads can operate independently, they still need to coordinate and synchronize their actions when accessing shared resources to prevent inconsistencies and race conditions.

Using Technique of Thread

1. By Extending the `Thread` Class
2. By Implementing `Runnable` Interface

Extending `Thread` Class

Class `myThread extends Thread`

```
public class myThread extends Thread{
    private int num;
    public myThread(int num){
        this.num = num;
    }
    @Override
    public void run(){
        for(int i=1; i<5; i++){
            System.out.println(i + " --- printed from thread - " + num);

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {

            }
        }
    }
}
```

This class myThread extends the Thread class, which means it inherits all the properties and methods of the Thread class. The myThread class has a private variable num and a constructor that initializes this variable. The run() method is overridden in this class. This method contains the code that will be executed when the thread is started. In this case, it prints a message and then sleeps for 1 second. This is repeated 4 times.

Class main

```
public class main {
    public static void main(String[] args) {
        for(int i=1; i<5; i++){
            myThread t = new myThread(i);
            t.start();
        }
    }
}
```

The main class contains the main method which is the entry point for any Java application. In this method, a loop is used to create 4 instances of the myThread class, each with a unique identifier i. The start() method is called on each instance, which in turn calls the run() method of the myThread class. As a result, 4 threads are created and started, each printing a unique message and sleeping for 1 second, 4 times. This demonstrates the concurrent execution of threads.

Output From this Code

```
1 --- printed from thread - 2
1 --- printed from thread - 4
1 --- printed from thread - 3
1 --- printed from thread - 1
2 --- printed from thread - 1
2 --- printed from thread - 3
2 --- printed from thread - 2
2 --- printed from thread - 4
3 --- printed from thread - 1
3 --- printed from thread - 4
3 --- printed from thread - 2
3 --- printed from thread - 3
4 --- printed from thread - 4
```

```
4 --- printed from thread - 2
4 --- printed from thread - 3
4 --- printed from thread - 1
```

As you can see there is no sequence of executing threads. While printing 1 thread(2) finished the task first. But while printing - 2 thread(2) came last.

Implementing `Runnable` Interface

Class `myThreadR implements Runnable`

```
public class myThreadR implements Runnable{
    private int num;
    public myThreadR(int num){
        this.num = num;
    }

    @Override
    public void run(){
        for(int i=1; i<5; i++){
            System.out.println(i + " --- printed from thread - " + num);

            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {

            }
        }

    }
}
```

The class `myThreadR` implements the `Runnable` interface, which means it must provide an implementation for the `run()` method. This class has a private variable `num` and a constructor that initializes this variable. The `run()` method contains the code that will be executed when the thread is started. In this case, it prints a message and then sleeps for 1 second. This is repeated 4 times.

Class `main`

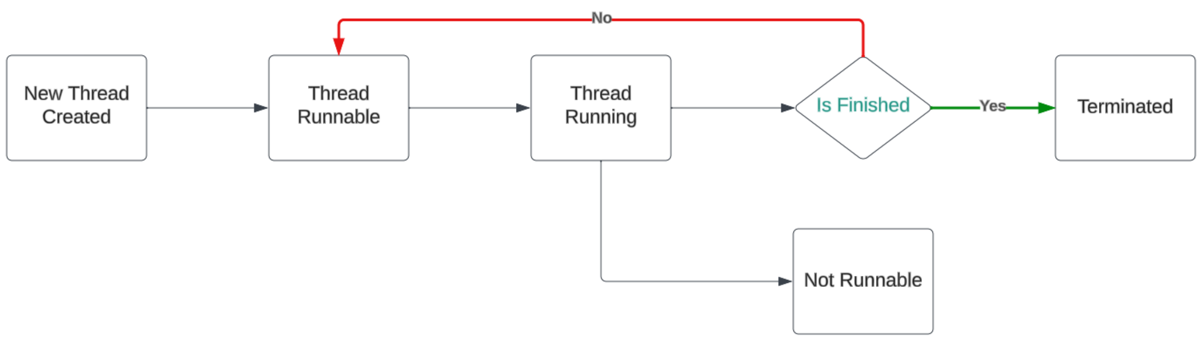
```
public class main {
    public static void main(String[] args) {
        for (int i = 1; i < 5; i++) {
            myThreadR t = new myThreadR(i);
            Thread Runnablethread = new Thread(t);
            Runnablethread.start();
        }
    }
}
```

The `main` class contains the `main` method which is the entry point for any Java application. In this method, a loop is used to create 4 instances of the `myThreadR` class, each with a unique identifier `i`. A new `Thread` object is created for each instance, passing the instance as an argument to the `Thread` constructor. The `start()` method is called on each `Thread` object, which in turn calls the `run()` method of the `myThreadR` class. As a result, 4 threads are created and started, each printing a unique message and sleeping for 1 second, 4 times. This demonstrates the concurrent execution of threads.

Output from this code

```
1 --- printed from thread - 2
1 --- printed from thread - 4
1 --- printed from thread - 1
1 --- printed from thread - 3
2 --- printed from thread - 4
2 --- printed from thread - 2
2 --- printed from thread - 1
2 --- printed from thread - 3
3 --- printed from thread - 2
3 --- printed from thread - 4
3 --- printed from thread - 1
3 --- printed from thread - 3
4 --- printed from thread - 2
4 --- printed from thread - 3
4 --- printed from thread - 4
4 --- printed from thread - 1
```

Life Cycle of a Thread



New Thread

The instance of a thread object / Thread Child Object / Runnable type object is created but `thatObject.start()` method is not invoked.

Thread Runnable

The `thatObject.start()` method is invoked but the thread is waiting to be executed,

1. Thread pool queue
2. Core Availability
3. Necessary Data from another thread

Thread Running

The thread type/subtype object has been selected for execution. A core is allocated to execute the operation. The operation on that thread is getting executed.

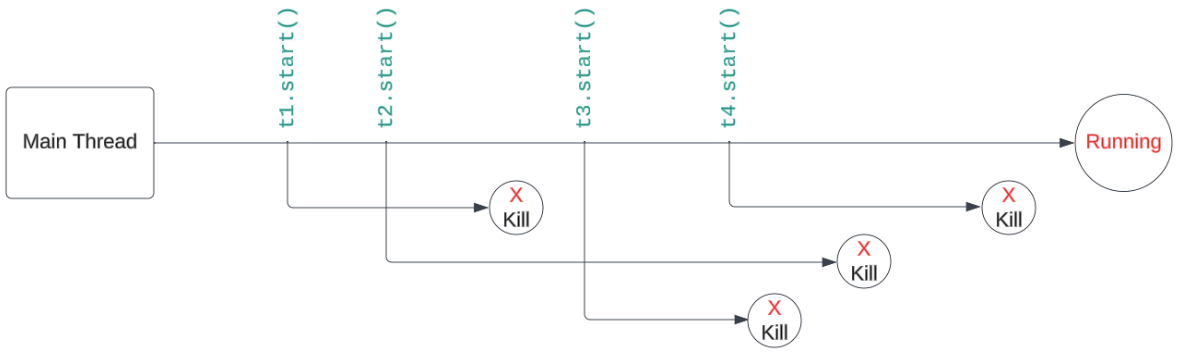
Not Runnable

The thread is created and selected but for some reason, the thread is blocked and not in execution anymore.

Terminated

`run()` method finishes its job and the thread is dismissed.

Thread Execution and Problem



Threads are created and run parallelly from and to the main thread. After a thread completes its work, Java Processor kills that thread and frees that memory.

On the other hand main thread goes on until the program ends.

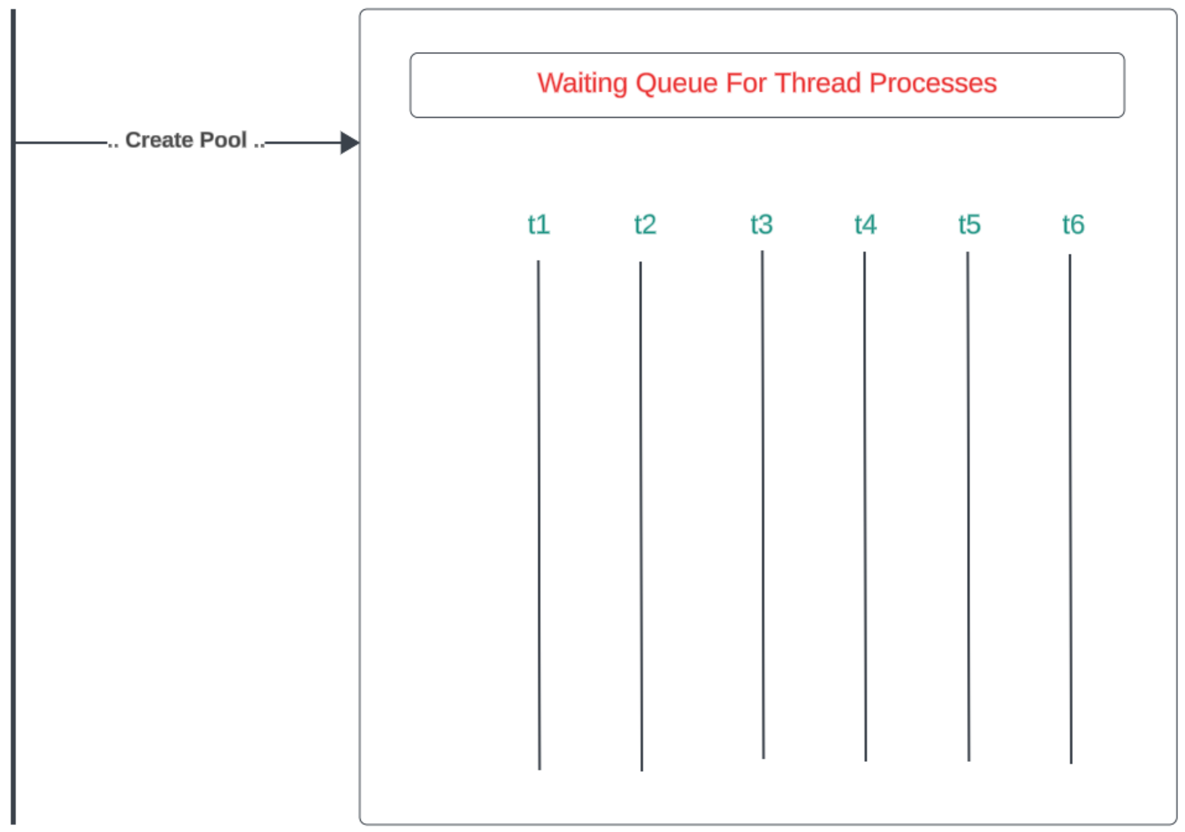
Problems of Threading

```
1 Java Thread = 1 OS Thread
```

Creating a lot of threads is expensive as taking a thread for Java is blocking OS activity on that thread. So 1000 threads can kill all the OS activities resulting in the death of the procssor.

The solution is using **Thread Pool**

Thread Pool



Here is a certain amount of threads pre-created and processes are submitted to the pool. Which are stored in a queue. When a thread gets its work done the next process takes over that thread. In this type of execution at a certain moment, there will be only `n threads` running.

The code for this execution will use the executor service framework.

Code for Executor Framework

```java
public class Task implements Runnable {
    private int id;

    public Task(int id) {
        this.id = id;
    }

    public void run() {
        System.out.println("Task with id " + id + " is in work - thread id: " +
                           Thread.currentThread().getId());
        long duration = (long) (Math.random() * 5);
        try {
            Thread.sleep(duration * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

In main,

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

public class main {
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(10);

        for (int i = 0; i < 100; i++) {
            service.execute(new Task(i));
        }
    }
}
```

The `Task` class implements the `Runnable` interface, which means it can be executed by a thread. The class has a single field, `id`, which is used to identify the task. The `run` method is overridden from the `Runnable` interface. This method is called when the thread executing this task is started. It prints out the task ID and the ID of the current thread, simulates a random duration of work using `Thread.sleep`, and handles any `InterruptedException`.

The `main` class is where the program starts. It creates a thread pool with 10 threads using `Executors.newFixedThreadPool(10)`. Then it creates 100 tasks and submits them to the `ExecutorService` to be executed. This way, the tasks are executed concurrently, but the number of active threads is limited to 10. Once a thread finishes a task, it can pick up and start another one. This is a very efficient way to manage and control threads in Java. It's especially useful in scenarios where you have a large number of tasks to be executed in parallel, but you want to limit the number of active threads for resource management.

The best number of threads to input in the pool

The number of processor threads/cores in your computer's CPU is the best for your CPU. Using Reflection to count the cores of the computer and using that for executor service.

```
public class idealPoolSizeMain {
    public static void main(String[] args) {
        int cores = Runtime.getRuntime().availableProcessors();
        ExecutorService service = Executors.newFixedThreadPool(cores);

        for (int i = 0; i < 100; i++) {
            service.execute(new Task(i));
        }
        System.out.println(cores);
    }
}
```

this `main` function is a modified version of the previous one, designed to optimize the usage of system resources.

The `idealPoolSizeMain` class contains the `main` method where the program starts. It first determines the number of available processors (or cores) on the system using `Runtime.getRuntime().availableProcessors()`. This is stored in the `cores` variable.

```
int cores = Runtime.getRuntime().availableProcessors();
```

Next, it creates a thread pool with several threads equal to the number of available processors. This is done using `Executors.newFixedThreadPool(cores)`. The idea here is to match the number of active threads to the number of available processors, which is often the ideal size for a thread pool, hence the name `idealPoolSizeMain`.

```
ExecutorService service = Executors.newFixedThreadPool(cores);
```

Then, similar to the previous `main` method, it creates 100 tasks and submits them to the `ExecutorService` to be executed. Each task is executed by one of the threads in the thread pool.

```java
for (int i = 0; i < 100; i++) {
    service.execute(new Task(i));
}
```

Finally, it prints out the number of cores to the console. This is just for informational purposes and isn’t necessary for the functioning of the thread pool.

```java
System.out.println(cores);
```

So, this version of the `main` method is more efficient than the previous one if the system has less than 10 cores. It allows the program to make full use of the system’s processing power without creating more threads than necessary. If the system has more than 10 cores, it will still only create 10 threads, just like the previous version. This is a great way to manage and control threads in Java, especially in scenarios where you have a large number of tasks to be executed in parallel but want to optimize resource usage.

Sleeping a thread

```java
public class SleepThread implements Runnable{
    String threadName;
    public SleepThread(String name) {
        this.threadName = name;
    }

    public void run() {
        System.out.println(threadName + " going to sleep for 3 seconds, current time
                           is " + LocalTime.now());
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            System.out.println(threadName + " interrupted");
        }
        System.out.println(threadName + " finished sleeping, current time is " +
                           LocalTime.now());
    }
}
```

```java
public class sleepMain {
    public static void main(String[] args) {
        Thread t1 = new Thread(new SleepThread("Thread 1"));
        Thread t2 = new Thread(new SleepThread("Thread 2"));
        t1.start();
        t2.start();
    }
}
```

Output is,

```
Thread 2 going to sleep for 3 seconds, current time is 16:38:58
Thread 1 going to sleep for 3 seconds, current time is 16:38:58
Thread 1 finished sleeping, current time is 16:39:01
Thread 2 finished sleeping, current time is 16:39:01
```

Here,

```
import java.time.LocalTime;
```

This line imports the `LocalTime` class from the `java.time` package. This class is used to get the current time.

```
public class SleepThread implements Runnable{
```

Here, we're declaring a public class named `SleepThread` that implements the `Runnable` interface. The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread.

```
    String threadName;
```

This line declares a variable `threadName` of type `String`. This will be used to store the name of the thread.

```
    public SleepThread(String name) {
         this.threadName = name;
     }
```

This is the constructor of the `SleepThread` class. It takes a `String` as an argument and assigns it to the `threadName` variable.

```
    public void run() {
```

This method is called when the thread is started. It's where the action of the thread is defined.

```
        System.out.println(threadName + " going to sleep for 3 seconds, current time is " + LocalTime.now());
```

This line prints a message to the console indicating that the thread is going to sleep and also prints the current time.

```
        try {
             Thread.sleep(3000);
         } catch (InterruptedException e) {
             System.out.println(threadName + " interrupted");
         }
```

This block of code puts the thread to sleep for 3000 milliseconds (or 3 seconds). If another thread interrupts this thread while it's sleeping (which is unlikely in this simple program), it will catch the `InterruptedException` and print a message to the console.

```
        System.out.println(threadName + " finished sleeping, current time is " + LocalTime.now());
    }
}
```

After the thread wakes up, it prints another message to the console indicating that it has finished sleeping and also prints the current time again.

```
public class sleepMain {
    public static void main(String[] args) {
        Thread t1 = new Thread(new SleepThread("Thread 1"));
        Thread t2 = new Thread(new SleepThread("Thread 2"));
        t1.start();
        t2.start();
    }
}
```

This is the `main` method where the program starts. It creates two `Thread` objects, `t1,` and `t2`, each associated with a `SleepThread` object. The threads are then started with the `start()` method, which calls the `run()` method of the `SleepThread` objects.

Output explanation

1. `Thread 2 going to sleep for 3 seconds, current time is 16:38:58`: This line is printed when Thread 2 starts and goes to sleep for 3 seconds. The current time is displayed, which is 16:38:58.
2. `Thread 1 goes to sleep for 3 seconds, current time is 16:38:58`: Almost simultaneously, Thread 1 also starts and goes to sleep for 3 seconds. The current time is still 16:38:58, indicating that Thread 1 and Thread 2 started at nearly the same time.
3. `Thread 1 finished sleeping, current time is 16:39:01`: After 3 seconds, Thread 1 wakes up and prints this message. The current time is now 16:39:01, which is 3 seconds later than when it started sleeping.
4. `Thread 2 finished sleeping, current time is 16:39:01`: Similarly, Thread 2 also wakes up after 3 seconds and prints this message. The current time is also 16:39:01, indicating that Thread 2 woke up at nearly the same time as Thread 1.

This output demonstrates that the two threads were running concurrently (at the same time), each sleeping for 3 seconds independently of each other. This is a fundamental concept in multithreading, where each thread can perform tasks independently while sharing the same memory space. In this case, the task was simply to sleep for 3 seconds, but in a real-world application, it could be any computational task like processing data or handling user input.

Thread Join

The `join()` method is used to pause the execution of the current thread until the thread it's called on finishes its execution. In other words, if a thread `A` calls `B.join()`, `A` will wait until `B` completes before it continues.

Code For Join

First Thread

```java
public class FirstThread extends Thread{
    @Override
    public void run() {
        for(int i = 0; i < 4; i++){
            System.out.println("First Thread: " + i);
            try {
                Thread.sleep(250);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Second Thread

```java
public class SecondThread extends Thread{
    @Override
    public void run() {

        for(int i = 0; i < 4; i++){
            System.out.println("Second Thread: " + i);
            try {
                Thread.sleep(250);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Main Thread

```java
public class joinMain {
    public static void main(String[] args) {
        Thread t1 = new FirstThread();
        Thread t2 = new SecondThread();
        t1.start();
        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.start();
    }
}
```

Explanation

Now we need some explanation what actually happened here,

Imagine you’re a conductor of an orchestra. In this orchestra, you have two sections: the strings (FirstThread) and the brass (SecondThread). Each section has its own sequence of notes to play (the loop from 0 to 3), and each note takes a little time to play (Thread.sleep(250)).

As the conductor (the main method), you signal the strings section to start playing their sequence (t1.start()). However, you want the brass section to wait until the strings section has finished before they start playing. To achieve this, you use a conductor’s technique (t1.join()) that makes you, and therefore the rest of the orchestra, wait until the strings section has finished their sequence.

Once the strings section is done, you then signal the brass section to start playing their sequence (t2.start()).

So, in this analogy:

- The orchestra conductor is the main thread.
- The strings section is the FirstThread (t1).
- The brass section is the SecondThread (t2).
- The conductor’s technique to wait for a section to finish before starting the next is the join() method.

This code demonstrates how one thread can wait for another thread to complete its tasks before starting its tasks, using the join() method. This is a fundamental concept in multithreading in Java.

Did you notice here actually the main thread waits not the `t2 thread`? By `join()` method we can make the caller thread wait until the called thread finishes its work. As here main waits to call t2. T2 does not wait when once called.

Thread Synchronization

Let’s take a code example and find the problems before learning about Synchronization,

Method Level Synchronization

Table Class

```
public class Table {
    void display(int n){
        for(int i=1;i<=5;i++){
            System.out.println(n + " * " + i + " = " + n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){
                System.out.println(e);
            }
        }
    }
}
```

Thread_A class

```
public class Thread_A extends Thread{
    Table t;
    public Thread_A(Table t){
        this.t=t;
    }
    @Override
    public void run(){
        t.display(5);
    }
}
```

Thread_B class

```
public class Thread_B extends Thread{
    Table t;
    public Thread_B(Table t){
        this.t=t;
    }
    @Override
    public void run(){
        t.display(6);
    }
}
```

Main Class

```
public class Main {
    public static void main(String[] args) {
        Table obj = new Table(); //only one object
        Thread_A t1=new Thread_A(obj);
        Thread_B t2=new Thread_B(obj);
        t1.start();
        t2.start();
    }
}
```

Output

```
5 * 1 = 5
6 * 1 = 6
6 * 2 = 12
5 * 2 = 10
5 * 3 = 15
6 * 3 = 18
5 * 4 = 20
6 * 4 = 24
6 * 5 = 30
5 * 5 = 25
```

The reason why the tables of 5 and 6 do not come in order is due to the nature of **multithreading** in Java.

In your code, two threads (Thread_A and Thread_B) are running concurrently. Each thread is executing the `display` method on the same `Table` object but with different inputs (5 for `Thread_A` and 6 for `Thread_B`).

When you start these threads using `t1.start();` and `t2.start();`, they begin executing their `run` methods concurrently. This means that both threads are running simultaneously, and the order in which they execute their tasks (in this case, printing the multiplication tables) is not guaranteed to be in any specific order. This is known as **thread interleaving**.

The `Thread.sleep(400);` line in the `display` method causes the current thread to pause for 400 milliseconds. However, this doesn't mean the threads will run in a specific order. While one thread is sleeping, the other thread may continue executing, leading to the interleaved output you see.

So, tables 5 and 6 are printed out of order because the threads are running concurrently, and the exact order of execution depends on how the threads are scheduled by the Java Virtual Machine (JVM), which can be influenced by many factors and is not predictable. This is a common characteristic of multithreaded programs.

In simple terms, think of it like two people (representing the two threads) given the same task (printing the multiplication table) and asked to work at the same time. Even if they take breaks (the `sleep` method), they don't necessarily start and finish at the same time or in a specific order. They work independently, and that's why the tables of 5 and 6 are not in order.

Solution

The correct way of doing this is using synchronization in table class…

```java
public class Table {
    synchronized void display(int n){
        for(int i=1;i<=5;i++){
            System.out.println(n + " * " + i + " = " + n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){
                System.out.println(e);
            }
        }
    }
}
```

Now output,

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
```

The tables are now in order because of the `synchronized` keyword that you added to the `display` method in the `Table` class.

In Java, the `synchronized` keyword is used to control the access of multiple threads to a shared resource. When a method is declared with `synchronized`, only one thread can access this method at a time, while all other threads that want to access this method are blocked until the method is released from the current thread. This is known as **mutual exclusion**.

In your code, when you declare the `display` method as `synchronized`, it means that once a thread (either `Thread_A` or `Thread_B`) enters this method, no other thread can enter the method until the first thread has finished executing the method. This prevents **thread interleaving**, which was causing tables 5 and 6 to be printed out of order in your previous code.

So, when you start the threads `t1` and `t2`, one of them will enter the `display` method first (let's say `t1` for example). While `t1` is in the `display` method, `t2` is blocked and cannot enter the method. `t1` will then print the entire table of 5 because it's not interrupted by `t2` thanks to the `synchronized` keyword. After `t1` has finished and exited the `display` method, `t2` can now enter the method and print the entire table of 6.

This is why the tables are now printed in order: the `synchronized` keyword ensures that the `display` method is fully executed by one thread before the other thread can start, resulting in tables of 5 and 6 being printed one after the other, rather than being mixed.

In simple terms, it's like having a rule that only one person (representing a thread) can use the shared resource (the `display` method) at a time. The other person has to wait their turn. This rule ensures that the tasks (printing the tables) are completed one at a time in order.

Block Level Synchronization

```java
public class Table {
    void display(int n){
        synchronized (this) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(n + " * " + i + " = " + n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }
}
```

Now the output also comes in synchronized order. Here is how it happens, In your updated code, you're using what's known as **Block Level Synchronization**. This is a more flexible form of synchronization in Java, where you specify exactly which part of the method needs to be synchronized, rather than synchronizing the entire method.

In your code, you've placed the `synchronized` keyword in front of a block of code (`synchronized (this) {...}`) inside the `display` method. This means that only the code inside this block is synchronized, and the rest of the method (if there were any other code outside this block) would not be synchronized.

The `this` keyword refers to the current object, which is the `Table` object in this case. So, similar to the previous code, only one thread can execute the synchronized block of the `display` method at a time on a given `Table` object.

The output remains in order because, just like before, the `synchronized` block ensures that the threads execute the printing of the tables one after the other, rather than concurrently.

Now, let's discuss the differences, pros, and cons of using Block Level Synchronization:

Differences:

- **Method Level Synchronization:** When a method is declared with `synchronized`, the whole method is synchronized. No two threads can access any synchronized method simultaneously on the same object.
- **Block Level Synchronization:** Only the specific block of code surrounded by the `synchronized` keyword is synchronized. Multiple threads can access the method simultaneously, but only one thread can execute the synchronized block at a time on the same object.

Pros of Block Level Synchronization:

- **More flexibility:** You can choose exactly which parts of the method need synchronization, rather than synchronizing the entire method. This can be useful when only a part of the method needs synchronization.
- **Better performance:** Since only the necessary parts of the method are synchronized, other parts of the method can be accessed by other threads concurrently, potentially leading to better performance.

Cons of Block Level Synchronization:

- **More complex:** It can be more difficult to correctly synchronize the right parts of the method, especially in complex programs. This could potentially lead to synchronization issues if not done correctly.

- **Harder to maintain:** If the code changes in the future, it might be harder to maintain the synchronization as you need to ensure the `synchronized` block still covers all necessary parts of the method.

Static Level Synchronization

Static Method Level

Let's make the table class static and change other classes accordingly,

```java
public class table {
    synchronized static void display(int n){
        for (int i = 1; i <= 5; i++) {
            System.out.println(n + " * " + i + " = " + n * i);
            try {
                Thread.sleep(400);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}
```

```java
public class Thread_A extends Thread{
    @Override
    public void run(){
        table.display(5);
    }
}
```

```java
public class Thread_B extends Thread{
    @Override
    public void run(){
        table.display(6);
    }
}
```

```java
public class statMain {
    public static void main(String[] args) {
        Thread_A t1 = new Thread_A();
        Thread_B t2 = new Thread_B();
        t1.start();
        t2.start();
    }
}
```

In your updated code, you've made the display method in the table class a static method and synchronized on it. This is a significant change and it affects how synchronization works in your program.

In Java, static methods belong to the class, not instances (objects) of the class. This means that the display method is shared across all instances of the table class.

When you synchronize on a static method, you're actually synchronizing on the class's Class object, not on individual instances of the class. In other words, even though you might have multiple instances of the table class, only one thread can access the display method at a time across all instances. This is because the lock is held on the Class object of the table class, not on individual instances.

In your specific code, you've created two threads (Thread_A and Thread_B), each of which calls the display method. Because the display method is static and synchronized, only one of these threads can execute the method at a time, even though they're operating on different instances of the table class. This is why the tables 5 and 6 are printed in order, just like in your previous code with instance method synchronization.

The main difference between this code and your previous code is the level at which synchronization occurs. In your previous code, synchronization occurred at the instance level, meaning each instance of the table class had its lock. In this code, synchronization occurs at the class level, meaning there's only one lock for the display method across all instances of the table class.

This type of synchronization can be useful when you need to ensure that only one thread can access a static method at a time, regardless of the number of instances of the class. However, it can also lead to increased contention if many threads need to access the static method, as they'll all be competing for the same lock.

Static Block Level

```java
public class table {
    static void display(int n){
        synchronized (table.class) {
            for (int i = 1; i <= 5; i++) {
                System.out.println(n + " * " + i + " = " + n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }
}
```

In your updated code, you've moved the `synchronized` keyword from the method level to a block of code inside the `display` method. This is similar to the Block Level Synchronization you used earlier, but with a key difference: you're synchronizing on `table.class`.

In Java, every type has one unique `Class` object associated with it, which represents the metadata of the class. When you write `table.class`, you're referring to the `Class` object for the `table` class.

When you synchronize on `table.class`, you're using the `Class` object of the `table` class as the lock for synchronization. This means that the synchronized block of code can only be accessed by one thread at a time across all instances of the `table` class, similar to when you synchronize on the static method.

The main difference between this code and your previous code is where the lock for synchronization is located:

- In your previous code, the lock was on the `display` method itself because you used method-level synchronization.
- In this code, the lock is on the `Class` object of the `table` class because you're using block-level synchronization with `table.class`.

The effect is the same: tables 5 and 6 are printed in order because only one thread can execute the synchronized block of code at a time, ensuring that the tables are printed one after the other, rather than concurrently.

This approach gives you the flexibility of block-level synchronization (you can choose exactly which parts of the method need to be synchronized) while still ensuring that only one thread can execute the synchronized block at a time across all instances of the `table` class, similar to method level synchronization on a static method.

Conditional Synchronization

Threads often need to communicate with each other to ensure smooth and efficient execution. Just like in a busy kitchen, where chefs must coordinate their tasks to prepare a meal, threads must synchronize their actions to share resources and avoid conflicts. This is where conditional synchronization comes into play, acting as the head chef who orchestrates the entire operation.

The `wait()` and `notify()` methods in Java are like walkie-talkies for threads. When a thread calls `wait()`, it's like a chef saying "I'm waiting for the sauce to simmer," and then stepping aside to let others work. The thread moves into a waiting state, releasing the lock it holds and allowing other threads to proceed.

On the other hand, `notify()` is like a chef announcing "The sauce is ready!" This wakes up a thread called `wait()` on the same object, signaling it to resume its work. However, just like in a kitchen where multiple chefs might be waiting for the sauce, `notify()` only wakes up one waiting thread, and it's up to the Java Virtual Machine (JVM) to decide which one.

Semaphore variables in Java act like traffic signals controlling access to a section of code. Imagine a narrow bridge on a busy road: only a certain number of cars can cross the bridge at a time to prevent congestion or accidents. Similarly, a semaphore maintains a set of permits, and a thread must acquire a permit before it can execute certain code. If all permits are taken, the thread must wait, just like a car waiting for the signal to turn green. When a thread is done with its task, it releases the permit, signaling that another thread can proceed, much like a car leaving the bridge makes room for the next car.

Through the use of `wait()`, `notify()`, and semaphore variables, Java provides powerful tools for threads to work together in harmony, much like a well-coordinated team of chefs in a bustling kitchen. In the following sections, we'll dive deeper into these concepts, exploring how they can be effectively used to manage complex multithreaded applications.

Let's See an Example

```java
public class PCMain {
    public static void main(String[] args) {
        SharedData sd = new SharedData();
        Producer p = new Producer(sd);
        Consumer c = new Consumer(sd);
        p.start();
        c.start();
    }
}
```

```java
public class Producer extends Thread{
    SharedData sd;
    public Producer(SharedData sd){
        this.sd=sd;
    }
    @Override
    public void run(){
        for(int i=1;i<=10;i++){
            sd.produce(i);
            try{
                Thread.sleep(400);
            }catch(InterruptedException e){
                System.out.println(e);
            }
        }
    }
}
```

```java
public class Consumer extends Thread{
    SharedData sd;
    public Consumer(SharedData sd){
        this.sd=sd;
    }
    @Override
    public void run(){
        for(int i=1;i<=10;i++){
            int q = sd.consume();
            System.out.println("Consumed: "+q);
            try{
                Thread.sleep(400);
            }catch(InterruptedException e){
                System.out.println(e);
            }
        }
    }
}
```

```java
public class SharedData {
    private int data;
    private boolean isProduced = true;
    public synchronized void produce(int n){
        if(isProduced == false){
            try{
                wait();
            }
            catch(InterruptedException e){
                System.out.println(e);
            }
        }
        data=n;
        System.out.println("Produced: "+n);
        isProduced = false;
        notify();
    }
    public synchronized int consume(){
        if(isProduced == true){
            try{
                wait();
            }
            catch(InterruptedException e){
                System.out.println(e);
            }
        }
        isProduced = true;
        notify();
        return data;
    }
}
```

One of the most classic problems is the Producer-Consumer problem. This problem involves two parties, a producer and a consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data, one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The code you provided is a perfect example of how to solve the Producer-Consumer problem in Java using `wait()`, `notify()`, and synchronization.

SharedData Class

The `SharedData` class represents the shared buffer between the producer and consumer. It has a `data` variable to hold the data being produced and consumed, and a `isProduced` boolean flag to track whether new data has been produced.

The `produce` method is used by the producer to produce data. If `isProduced` is `false`, meaning the previous data hasn't been consumed yet, the producer thread calls `wait()` and enters the waiting state. Once the consumer has consumed the data and calls `notify()`, the producer is awakened, produces new data, sets `isProduced` to `false`, and calls `notify()` to awaken the consumer.

The `consume` method is used by the consumer to consume data. If `isProduced` is `true`, meaning there's no new data to consume, the consumer thread calls `wait()` and enters the waiting state. Once the producer has produced new data and calls `notify()`, the consumer is awakened, consumes the data, sets `isProduced` to `true`, and calls `notify()` to awaken the producer.

Producer Class

The `Producer` class extends `Thread`, representing the producer thread in the problem. It has a `SharedData` object on which it will produce data.

In its `run` method, it produces data from 1 to 10 using the `SharedData` object's `produce` method. After producing each piece of data, it sleeps for 400 milliseconds to simulate the time it takes to produce the data.

Consumer Class

The `Consumer` class, like the `Producer` class, extends `Thread` and represents the consumer thread in the problem. It also has a `SharedData` object from which it will consume data.

In its `run` method, it consumes data using the `SharedData` object's `consume` method and prints out the consumed data. After consuming each piece of data, it also sleeps for 400 milliseconds to simulate the time it takes to consume the data.

PCMain Class

The `PCMain` class contains the `main` method where the program starts. It creates a `SharedData` object and then creates a `Producer` and `Consumer` with this `SharedData` object. It starts the producer and consumer threads, kicking off the production and consumption process.

In conclusion, this code is a demonstration of how to use `wait()`, `notify()`, and synchronization in Java to solve the Producer-Consumer problem. It shows how multiple threads can work together, communicating and synchronizing their actions to share a common resource. It's a powerful technique that's widely applicable in many areas of concurrent programming.

Output For this code

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

The output of this code is a perfect demonstration of the Producer-Consumer problem being solved using `wait()`, `notify()`, and synchronization in Java.

The reason why there is no consumption before production is due to the way synchronization and inter-thread communication are set up in your code. Here's a step-by-step explanation:

1. The `Producer` thread starts and enters the `produce` method of the `SharedData` object. Since `isProduced` is initially `true`, it doesn't enter the `wait()` state and proceeds to produce the first piece of data (1).
2. After producing the data, it sets `isProduced` to `false` and calls `notify()`. This wakes up any waiting threads (in this case, the `Consumer` thread) and tells them that new data has been produced.
3. The `Producer` thread then goes to sleep for 400 milliseconds, giving the `Consumer` thread a chance to consume the data.
4. The `Consumer` thread starts and enters the `consume` method of the `SharedData` object. Since `isProduced` is `false`, indicating that new data has been produced, it doesn't enter the `wait()` state and proceeds to consume the data.
5. After consuming the data, it sets `isProduced` to `true` and calls `notify()`. This wakes up the `Producer` thread and tells it that the data has been consumed and it can produce new data.
6. This process repeats for each piece of data from 1 to 5, resulting in the alternating "Produced" and "Consumed" output you see.

In essence, the `wait()` and `notify()` methods, along with the `isProduced` flag, ensure that the `Producer` and `Consumer` threads alternate their operations. The `Producer` doesn't produce new data until the old data has been consumed, and the `Consumer` doesn't consume data until new data has been produced. This is why you see each "Produced" output followed by a corresponding "Consumed" output.