# Process, Threads, Multitasking, Multithreading, and Multiprocessing in Java

***Java*** is a versatile and widely used programming language that allows developers to create complex and efficient applications. To harness the full power of Java, it's essential to understand the concepts of processes, threads, multitasking, multithreading, and multiprocessing. These concepts are fundamental for building robust and responsive software. In this article, we'll explore each of these concepts, provide real-life examples, and discuss their significance in Java programming.

**Processes:** A ***process*** is a program in execution. It is the basic unit of resource allocation and scheduling in an operating system. A process is an independent program that runs in its own memory space. In Java, when you execute a Java application, it runs as a process. Each process has its resources, including memory, files, and system handles. Processes are isolated from each other, ensuring that one process cannot directly access the memory of another.

***For example***, a word processor, a web browser, and a music player are all separate processes.

**How do Processes work :**

- **Loading Program :** Computer loads the program into memory in a special binary code.
- **Resource Allocation :** Program needs resources like registers, program counter, and a stack, provided by the operating system.
- **Registers :** Tiny storage in the processor for instructions, memory addresses, or important data.
- **Program Counter :** Keeps track of where the program is in its sequence of instructions.
- **Stack** : Keeps information about active subroutines or tasks within the program.
- **Individual Processes :** Multiple instances of the same program running concurrently are called processes.

**Thread:** Separate flow of execution is called ***Thread***. If there is only one flow then it is called **Single Thread** programming. For every thread there would be a separate job. Processes can contain multiple threads, and these threads share the same memory space and resources within the process. Threads enable concurrent execution of tasks within a single process.

**How do Thread works :**

As we know that a thread is a sub-process or an execution unit within a process. A process can contain a single thread to multiple threads. A thread works as follows:

- When a process starts, OS assigns the memory and resources to it. Each thread within a process shares the memory and resources of that process only.
- Threads are mainly used to improve the processing of an application. In reality, only a single thread is executed at a time, but due to fast context switching between threads gives an illusion that threads are running parallelly.
- If a single thread executes in a process, it is known as a single-threaded And if multiple threads execute simultaneously, then it is known as multithreading.

**Multitasking :**

***Multitasking*** is the ability of an operating system to run multiple processes or threads concurrently. It allows users to switch between applications seamlessly and efficiently. Java applications leverage multitasking to ensure responsiveness and efficient resource utilization.

**There are two types of Multitasking:**

- **Process based multitasking :** Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking. Process based multitasking is best suited at OS level.

  *Examples :* typing a java program, listening to a song, downloading the file from internet
- **Thread based multitasking :** Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based Multitasking. Each independent part is called "Thread". This type of multitasking is best suited at Programmatic level .The main advantages of multitasking is to reduce the response time of the system and to improve the performance.

  *Examples :* to implement multimedia graphics, to develop web application servers ,and to develop video games.

**Multithreading :**

***Multithreading*** is the concurrent execution of multiple threads within a single process. In Java, multithreading is a powerful mechanism to enhance the performance and responsiveness of applications. It allows you to divide tasks into smaller threads that can execute independently and concurrently.

Threads are lightweight, independent sub-processes within a process that share the same memory space.

If multiple threads are waiting to execute, then which thread will execute first is decided by ***Thread Scheduler*** which is part of JVM. In case of ***Multithreading*** we can't predict the exact output only possible output we can expect. Since jobs of threads are important, we are not interested in the order of execution it should just execute such that performance should be improved.

*Example:* A desktop application providing functionality like editing, printing, etc. is a multithreaded application.

**Multiprocessing:**

***Multiprocessing*** is the use of multiple processes to perform tasks. Unlike multithreading, multiprocessing utilizes multiple independent processes, each with its memory space. This approach is suitable for tasks that require true parallelism and can take full advantage of multi-core processors.
*Example:* A video editing software can use multiprocessing to apply filters and effects to different frames of a video simultaneously, speeding up the rendering process.

## Implementation of Thread in Java

Java is renowned for its robust support for multithreading, allowing developers to create applications that can perform multiple tasks concurrently. In this comprehensive guide, we'll explore the fundamentals of implementing threads in Java, from the basics of thread creation to advanced threading concepts.
In Java, a thread is a lightweight, independent path of execution within a program. Threads enable applications to perform multiple tasks simultaneously, enhancing performance and responsiveness. Java provides built-in support for threads through the Thread class and the Runnable interface.

### How to create a thread in Java ?

There are two ways to create a thread:

1. **By extending Thread class.**
2. **By implementing Runnable interface.**

You can create threads by implementing the runnable interface and overriding the run() method. Then, you can create a thread object and call the start() method.

### Creating Threads Using the Thread Class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.
The simplest way to create a thread in Java is by extending the Thread class. Here's a step-by-step guide:

- **Define a Class that Extends Thread:**

```
/*package whatever //do not write package name here */
import java.io.*;

class MyThread extends Thread {
    public void run()
    {
        // Code to be executed by the thread
```

The **run()** method contains the code that the thread will execute when started.

- **Instantiate and Start the Thread:**

```
MyThread myThread = new MyThread();
myThread.start();
```

The **start()** method initiates the execution of the thread. It calls the **run()** method internally.

Example of creation of thread using Thread Class:

Java

```
/*package whatever //do not write package name here */
import java.io.*;

class MyThread extends Thread {
    @Override public void run()
    {
        for (int i = 0; i < 10; i++)
            System.out.println("child thread");
// defining a thread(writing a class and extending a Thread)
// job a thread(code written inside run())

class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread(); // Thread instantiation
        t.start(); // starting a thread
        // At this line 2 threads are there
        for (int i = 1; i <= 5; i++)
            System.out.println("Main Thread");
```

**Output**

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
child thread
child thread
```

```
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

**Behind the scenes:**

1. Main thread is created automatically by JVM.
2. Main thread creates child thread and starts the child thread.

Thread Scheduler:

If multiple threads are waiting to execute, then which thread will execute 1st is decided by Thread Scheduler which is part of JVM. In case of Multi-Threading we can't predict the exact output only possible output we can expect. Since jobs of threads are important, we are not interested in the order of execution it should just execute such that performance should be improved.

Various Constructors available in Thread class:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r, String name);
5. Thread t=new Thread(ThreadGroup g, String name);
6. Thread t=new Thread(ThreadGroup g, Runnable r);
7. Thread t=new Thread(ThreadGroup g, Runnable r, String name);
8. Thread t=new Thread(ThreadGroup g, Runnable r, String name,long stackSize);

## Case studies of start() and run() methods while using Thread class

### Case 1 : Difference between start() and run() method

In a main method if we call ***t.start()*** a separate thread will be created which is responsible for executing the run() method. If we call ***t.run()***, no separate thread will be created rather the method will be called just like the normal method by the main thread. If we replace ***t.start()*** with ***t.run()*** then the output of the program would be.

Java

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread {
    @Override public void run()
    {
        System.out.println("Normal rum method ");
    }
}

class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread(); // Thread instantiation
        //  t.start(); // starting a thread
        t.run();
        System.out.println("Main Thread");
```

### Case 2 : Importance of Thread class start() method

For every thread, required mandatory activities like registering the thread with thread scheduler will be taken care by Thread class ***start()*** method and programmer is responsible of just doing the job of the thread inside ***run()*** method ***start()*** acts like an assistance to programmer.

```
/*package whatever //do not write package name here */
// Code inside the start()
start()
{
    1 - Register thread with ThreadScheduler.
    2 - All other mandatory low level activities.
    3 - Invoke or calling run() method.
```

```
}
```
We can conclude that without executing thread class **start()** method there is no chance of starting a new Thread in java. Due to this ***start()*** is considered as heart of Multi-Threading.

### Case 3 : If we are not overriding run() method

If we are not overriding ***run()*** method then Thread class ***run()*** method will be executed which has empty implementation and hence we won't get any output.

Java

```
/*package whatever //do not write package name here */

import java.io.*;
class MyThread extends Thread
{

}
class ThreadDemo{
    public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
            System.out.println("No Output");
    }
}
```

**Output**

```
No Output
```

*It is highly recommended to override run() method, otherwise don't go for Multithreading concept.*

### Case 4 : Overloading of run() method

We can overload ***run()*** method but Thread class start() will always call ***run()*** with zero argument. If we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

Java

```
/*package whatever //do not write package name here */
import java.io.*;

class MyThread extends Thread {
    public void run()
    {
        System.out.println("No arg method");
    }
    public void run(int i)
    {
        System.out.println("Zero arg method");
    }
class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread();
        t.start();
```

**Output**

```
No arg method
```

### Case 5 : Overriding of start() method

If we override ***start()*** then our ***start()*** method will be executed just like normal method, but no new Thread will be created and no new Thread will be started.

#### *Example 1 :*

Java

```
/*package whatever //do not write package name here */

import java.io.*;
class MyThread extends Thread {
    public void run()
    {
      System.out.println("Thread are created  method");
    }
    public void start()
    {
        System.out.println("Start method executed like normal method ");
    }
}
class ThreadDemo {
    public static void main(String... args)
```

```
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

**Output**

```
Start method
```

*It is never recommended to override **start()** method.*

### *Example 2 :*

```
/*package whatever //do not write package name here */

import java.io.*;

class MyThread extends Thread{
      public void run()
    {
                System.out.println("Run method");
      }
      public void start()
    {
                System.out.println("Start  method");
      }
}
class ThreadDemo{
      public static void main(String... args)
    {
                MyThread t=new MyThread();
                t.start();
                System.out.println("Main method");
```

**Output**

```
Start  method
Main method
```

In the above example no new thread are created and start method are executed like normal method.

*Example 3 :*

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread{
     public void start(){
            super.start();
            System.out.println("Start method");
     }
     public void run(){
            System.out.println("Run method");
class ThreadDemo{
     public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
            System.out.println("Main method");
```

**Output**

```
Start method
Run method
Main method
```

The provided Java code defines a custom thread class *MyThread* that extends the *Thread* class. In this class, the *start()* method is overridden to call the parent class's *start()* method and print *start method* when the thread is started. The *run()* method is also overridden to print "run method." In the *ThreadDemo* class, an instance of *MyThread* is created and started. When executed, it will print *start method*, *run method*, and *main method*, demonstrating the order of method execution during thread creation and start.

Mark as Read

Report An Issue

## Case studies of start() and run() methods while using Thread class

### Case 1 : Difference between start() and run() method

In a main method if we call ***t.start()*** a separate thread will be created which is responsible for executing the run() method. If we call ***t.run()***, no separate thread will be created rather the method will be called just like the normal method by the main thread. If we replace ***t.start()*** with ***t.run()*** then the output of the program would be.

Java

```
/*package whatever //do not write package name here */
import java.io.*;
```

```java
class MyThread extends Thread {
    @Override public void run()
    {
        System.out.println("Normal rum method ");
    }
class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread(); // Thread instantiation
        //  t.start(); // starting a thread
        t.run();
        System.out.println("Main Thread");
```

## Case 2 : Importance of Thread class start() method

For every thread, required mandatory activities like registering the thread with thread scheduler will be taken care by Thread class ***start()*** method and programmer is responsible of just doing the job of the thread inside ***run()*** method ***start()*** acts like an assistance to programmer.

```
/*package whatever //do not write package name here */
// Code inside the start()
start()
{
    1 - Register thread with ThreadScheduler.
    2 - All other mandatory low level activities.
    3 - Invoke or calling run() method.
}
```

We can conclude that without executing thread class **start()** method there is no chance of starting a new Thread in java. Due to this ***start()*** is considered as heart of Multi-Threading.

## Case 3 : If we are not overriding run() method

If we are not overriding ***run()*** method then Thread class ***run()*** method will be executed which has empty implementation and hence we won't get any output.

Java

```java
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread
{

}
class ThreadDemo{
      public static void main(String... args){
                MyThread t=new MyThread();
                t.start();
                System.out.println("No Output");
```

**Output**

```
No Output
```

*It is highly recommended to override run() method, otherwise don't go for Multithreading concept.*

## Case 4 : Overloading of run() method

We can overload ***run()*** method but Thread class start() will always call ***run()*** with zero argument. If we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

Java

```java
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread {
    public void run()
    {
        System.out.println("No arg method");
    }
    public void run(int i)
    {
        System.out.println("Zero arg method");
    }
}
class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread();
        t.start();
```

**Output**

```
No arg method
```

## Case 5 : Overriding of start() method

If we override ***start()*** then our ***start()*** method will be executed just like normal method, but no new Thread will be created and no new Thread will be started.

***Example 1 :***

Java

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread {
    public void run()
    {
      System.out.println("Thread are created  method");
    }
    public void start()
    {
        System.out.println("Start method executed like normal method ");
    }
class ThreadDemo {
    public static void main(String... args)
    {
        MyThread t = new MyThread();
        t.start();
```

**Output**

```
Start method
```

*It is never recommended to override* ***start()*** *method.*

***Example 2 :***

```
/*package whatever //do not write package name here */

import java.io.*;

class MyThread extends Thread{
    public void run()
    {
            System.out.println("Run method");
    }
    public void start()
    {
            System.out.println("Start  method");
    }
}
class ThreadDemo{
    public static void main(String... args)
    {
            MyThread t=new MyThread();
            t.start();
            System.out.println("Main method");
    }
}
```

**Output**

```
Start  method
Main method
```

In the above example no new thread are created and start method are executed like normal method.

*Example 3 :*

Java

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread{
    public void start(){
            super.start();
            System.out.println("Start method");
    }
    public void run(){
            System.out.println("Run method");
    }
}
class ThreadDemo{
    public static void main(String... args){
```

```
                MyThread t=new MyThread();
                t.start();
                System.out.println("Main method");
```

**Output**

```
Start method
Run method
Main method
```

The provided Java code defines a custom thread class *MyThread* that extends the *Thread* class. In this class, the *start()* method is overridden to call the parent class's *start()* method and print *start method* when the thread is started. The *run()* method is also overridden to print "run method." In the *ThreadDemo* class, an instance of *MyThread* is created and started. When executed, it will print *start method*, *run method*, and *main method*, demonstrating the order of method execution during thread creation and start

## Thread Priority and Thread concurrency

### Thread Priorities

For every Thread in java has some priority. Valid range of priority is 1 to 10, it is not 0 to 10. If we try to give a different value the it would result in *IllegalArgumentException*.

- Thread.MIN_PRIORITY = 1
- Thread.MAX_PRIORITY = 10
- Thread.NORM_PRIORITY = 5

Thread class does not have priorities is *Thread.LOW_PRIORITY*, *Thread.HIGH_PRIORITY*. Thread scheduler allocates C.P.U time based on Priority. If both the threads have the same priority then which thread will get a chance as a program we can not predict because it is vendor dependent.

**We can set and get priority values of the thread using the following methods:**

1. public final void setPriority (int priorityNumber)
2. public final int getPriority ()

The allowed priorityNumber is from 1 to 10, if we try to give other values it would result in *IllegalArgumentException*.
System.out.println(Thread.currentThread().setPriority(100); // IllegalArgumentException.

### Default Priority

The default priority for only main thread is ***5***, where as for other threads priority will be inherited from parent to child. Parent Thread priority will be given as Child Thread Priority.

**Example 1 :**

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread{}
public class TestApp{
        public static void main(String... args)
    {
                System.out.println(Thread.currentThread().getPriority());//5
                Thread.currentThread().setPriority(7);
                MyThread t= new MyThread();
                System.out.println(Thread.currentThread().getPriority());//7
```

**Output**

```
5
7
```

MyThread is creating by *mainThread*, so priority of *mainThread* will be shared as a priority for MyThread.

Example 2 :

```
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread{
        @Override
        public void run()
    {
                for (int i=1;i<=5 ;i++ )
        {
                        System.out.println("child thread");
                }}}
public class TestApp{
        public static void main(String... args)
    {           MyThread t= new MyThread();
                t.setPriority(7);//line -1
                t.start();
                for (int i=1; i<=5; i++){
                        System.out.println("main thread");
        }}}
```

**Output**

```
main thread
main thread
main thread
main thread
```

```
main thread
child thread
child thread
child thread
child thread
child thread
```

Since priority of child thread is more than main thread, JVM will execute child thread first whereas for the parent thread priority is 5 so it will get last chance. If we comment line-1, then we can't predict the order of execution because both the threads have same priority.

## Understanding the Basics of Multithreading in Java

In Java, a thread is a lightweight process that executes a series of instructions independently. These threads can run concurrently, making Java applications more responsive and efficient. However, when multiple threads access shared resources or perform coordinated tasks, synchronization mechanisms become crucial to avoid data corruption and race conditions.

### 1- yield() :

It causes to pause current executing Thread for giving chance for waiting Threads of same priority. If there is no waiting Threads or all waiting Threads have low priority then same Thread can continue its execution. If all the threads have same priority and if they are waiting then which thread will get chance we can't expect, it depends on Thread Scheduler. The Thread which is yielded, when it will get the chance once again depends on the mercy on Thread Scheduler and we can't expect exactly. However, it's essential to note that yield doesn't guarantee that another thread will immediately start running, as the JVM's thread scheduler makes the final decision.

Java

```java
/*package whatever //do not write package name here */
import java.io.*;
class MyThread extends Thread{

	@Override
	public void run(){
		for (int i=1;i<=5 ;i++ ){
			System.out.println("child thread");
			Thread.yield();//line-1
		}
	}
}
public class TestApp{
	public static void main(String... args){
		MyThread t= new MyThread();
		t.start();
		for (int i=1;i<=5 ;i++ ){
			System.out.println("Parent Thread");
		}
	}
}
```

**Output**

```
Parent Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
child thread
child thread
child thread
child thread
child thread
```

**Note :** If we comment line-1, then we can't expect the output because both the threads have same priority then which thread the Thread Scheduler will schedule is not in the hands of programmer but if we don't comment line-1, then there is a possibility of main thread getting more no of times, so main thread execution is faster then child thread will get chance.

### 2- join() :

The join method in Java is a powerful tool for thread synchronization. It allows one thread to wait for another thread to complete its execution before proceeding. This is especially useful when you want to coordinate the order of execution in a multithreaded program.

If the thread has to wait until the other thread finishes its execution then we need to go for join(). If thread1 executes thread2.join() then thread1 should should wait till thread2 finishes its execution. Thread1 will be entered into waiting state until thread2 completes, once t2 completes then t1 can continue with its execution.

Java

```java
/*package whatever //do not write package name here */
import java.io.*;
public class JoinExample {
    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(() -> {
            System.out.println("Thread 1 is running");
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("Thread 2 is running");
        });
        thread1.start();
        thread2.start();
        // Wait for thread1 to finish
        thread1.join();
        System.out.println("Main thread continues after thread1");
    }}
```

**Output**

```
Thread 1 is running
Main thread continues after thread1
Thread 2 is running
```

## 3- sleep() :

The sleep method in Java is used to introduce a pause or delay in the execution of a thread. It can be helpful in scenarios where you need to wait for a certain period or introduce timeouts. If a thread do not want to perform any operation for a particular amount of time then we should go for sleep().

public static void sleep(long ms, int ns) throws InterruptedException
public static native void sleep(long ms) throws InterruptedException

Java

```java
/*package whatever //do not write package name here */
import java.io.*;
public class SleepExample {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            try {
                System.out.println("Thread is sleeping for 2 seconds");
                Thread.sleep(2000); // Sleep for 2 seconds
                System.out.println("Thread woke up after sleeping");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        thread.start(); }}
```

In this example, the thread sleeps for 2 seconds using Thread.sleep(2000) before resuming its execution. This can be useful for introducing delays or timing mechanisms in your multithreaded programs.

Every sleep method throws ***Interrupted Exception***, which is a checked Exception so we should compulsorily handle the exception using *try catch* or by *throws* keyword otherwise it would result in *compile time error*.

# Synchronization

### Why Synchronization Matters ?

In a multi-threaded environment, multiple threads run concurrently, and they may access and modify shared data simultaneously. Without proper synchronization, this can lead to a range of issues, including data corruption, race conditions, and unpredictable behavior. Synchronization mechanisms help us control the execution of threads and prevent such problems.

### Synchronization

1. Synchronized is a keyword applicable only for methods and blocks
2. If we declare a method/block as synchronized then at a time only one thread can execute that method/block on that object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using lock concept.

Java

```java
class X{
   synchronized void m1(){}
```

```
    synchronized void m2(){}
    void m3(){}
}
```

***Explanation :***

1. If t1 thread invokes m1() then on the Object X lock will applied.
2. If t2 thread invokes m2() then m2() can't be called because lock of X object is with m1.
3. If t3 thread invokes m3() then execution will happen because m3() is non-synchronized.

Lock concept is applied at the Object level not at the method level. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture. If a Thread wants to execute any synchronized method on the given object first it has to get the lock of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. Lock concept is implemented based on object not based on method.

### The Example without Synchronization:

Java

```
/*package whatever //do not write package name here */
import java.io.*;

class Display {
    public void wish(String name)
    {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Good Morning: " + name);
        } }}
class MyThread extends Thread {

    Display d;
    String name;

    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }
    @Override public void run() { d.wish(name); }
}
public class Test3 {
    public static void main(String... args)
        throws InterruptedException
    {
        Display d = new Display();
        MyThread t1 = new MyThread(d, "dhoni");
        MyThread t2 = new MyThread(d, "yuvi");
        t1.start();
        t2.start();
    }}
```

Java

```
Output:
Good Morning: yuvi
Good Morning: dhoni
Good Morning: dhoni
Good Morning: dhoni
Good Morning: yuvi
Good Morning: yuvi
```

Here we didn’t use the synchronized keyword so both the threads are executing at a time so in the output, thread-0 is interfering with thread-1, and hence, we are getting inconsistent results.

### Java Synchronized Method:

To create a synchronized method in Java, you can use the synchronized keyword in the method declaration. When a method is marked as synchronized, only one thread can execute it at a time, and other threads that attempt to access the method will be blocked until the executing thread completes the method.

If we use the Synchronized keywords in any method then that method is Synchronized Method.

- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won’t be released until the thread completes its function.

*Syntax:*

Java

```
public synchronized void synchronizedMethod() {
```

```
    // Code that needs to be synchronized
}
```

***Java Synchronized Method Example:***

```
/*package whatever //do not write package name here */

import java.io.*;

class Display {
    public  synchronized  void wish(String name)
    {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Good Morning: " + name);
        }
    }
}

class MyThread extends Thread {

    Display d;
    String name;

    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }

    @Override public void run() { d.wish(name); }
}
public class Test3 {
    public static void main(String... args)
        throws InterruptedException
    {
        Display d = new Display();
        MyThread t1 = new MyThread(d, "dhoni");
        MyThread t2 = new MyThread(d, "yuvi");
        t1.start();
        t2.start();
    }
}
```

**Output**

```
Good Morning: dhoni
Good Morning: dhoni
Good Morning: dhoni
Good Morning: yuvi
Good Morning: yuvi
Good Morning: yuvi
```

Here we used ***synchronized*** keywords. It helps to execute a single thread at a time. It is not allowing another thread to execute until the first one is completed, after completion of the first thread it allowed the second thread. Now we can see the output correctly the ***"Good Morning : dhoni"*** is printed first, and only after Thread-0 completes, ***"Good Morning : yuvi"*** is printed. This synchronization ensures thread safety and data consistency in a multi-threaded environment.

### Java Synchronized Block:

A synchronized block in Java is a way to create a critical section, where only one thread at a time can execute a specific section of code. It is defined using the synchronized keyword, and it ensures that multiple threads do not simultaneously access or modify shared resources, avoiding potential data corruption or inconsistency.

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose we have 100 lines of code in our method, but we want to synchronize only 10 lines, in such cases, we can use synchronized block. If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

*The basic syntax of a synchronized block is as follows:*

Java

```
synchronized (object) {
    // Critical section code
}
```

**In this syntax:**

- ***object*** is an object that serves as a monitor. Only one thread with the same monitor can access the critical section at a time.
- The code within the block is referred to as the critical section. This is where you put the code that needs to be synchronized to ensure thread safety.

***Java Synchronized Block Example:***

Java

```
import java.io.*;

class Display {
    public void wish(String name)
    {
      //100  lines of code
      synchronized (this){
        for (int i = 1; i <= 3; i++) {
            System.out.println("Good Morning: " + name);

class MyThread extends Thread {

    Display d;
    String name;

    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }

    @Override public void run() { d.wish(name); }
}
public class Test3 {
    public static void main(String... args)
        throws InterruptedException
    {
        Display d = new Display();
        MyThread t1 = new MyThread(d, "sachin");
        MyThread t2 = new MyThread(d, "rohit");
        t1.start();
        t2.start();
    }
}
```

**Output**

```
Good Morning: sachin
Good Morning: sachin
Good Morning: sachin
Good Morning: rohit
Good Morning: rohit
Good Morning: rohit
```

***Explanation :*** In the above example demonstrates multithreading using a *Display* class and two threads, $t1$ and $t2$. The *wish* method in the *Display* class contains a *synchronized* block that ensures only one thread can print "Good Morning" messages for a given name at a time. This prevents interleaving and guarantees orderly output. The use of synchronization is crucial in multithreading to maintain data integrity.