Topics to cover:

1. Threading in Java

    1. Threading

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If exception occurs in one thread, it doesn't affect other threads. It uses a shared memory area.

Threads allow a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Multithreading in Java

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

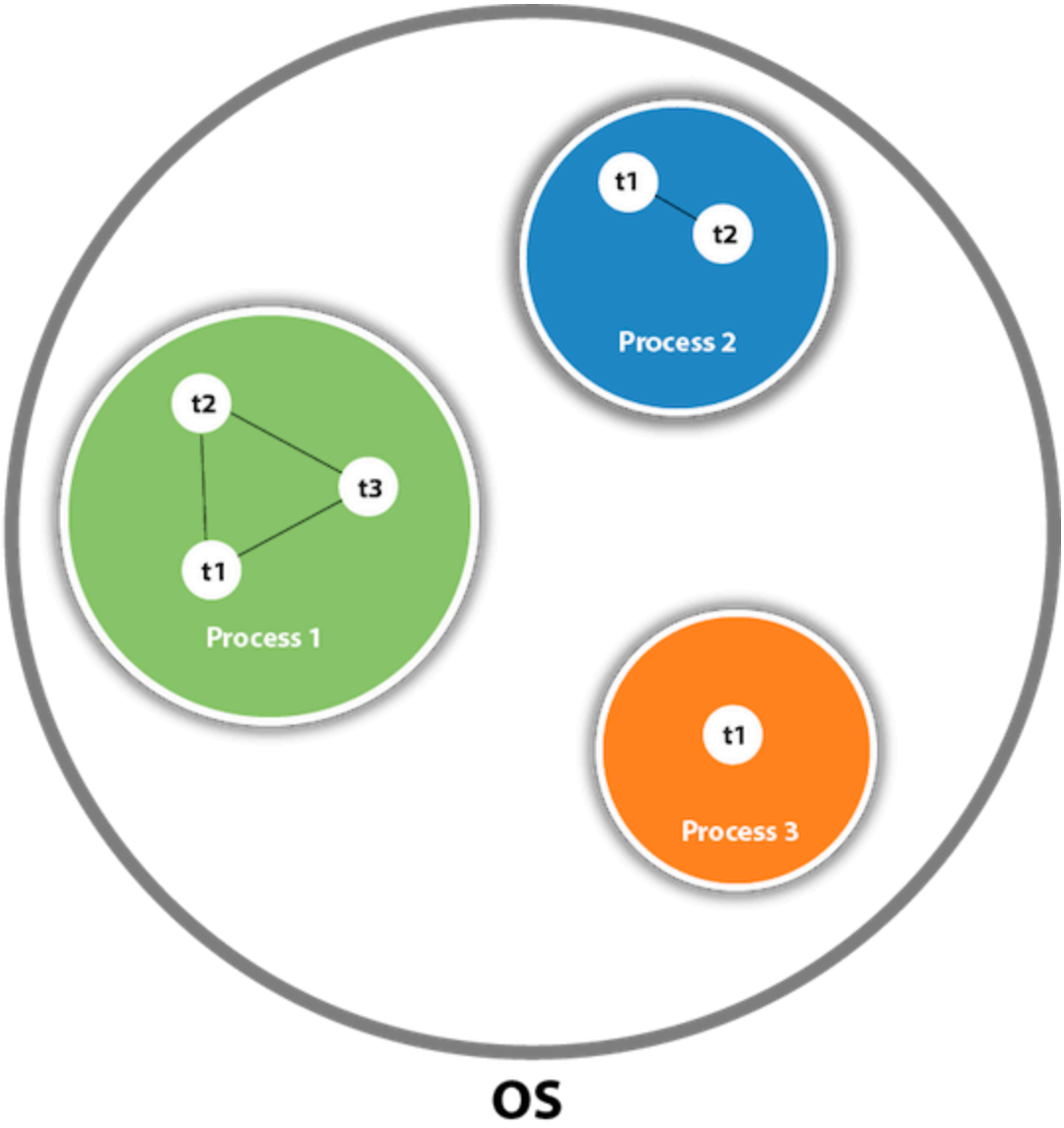
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.

- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

> Note: At least one process is required for each thread.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

> Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state

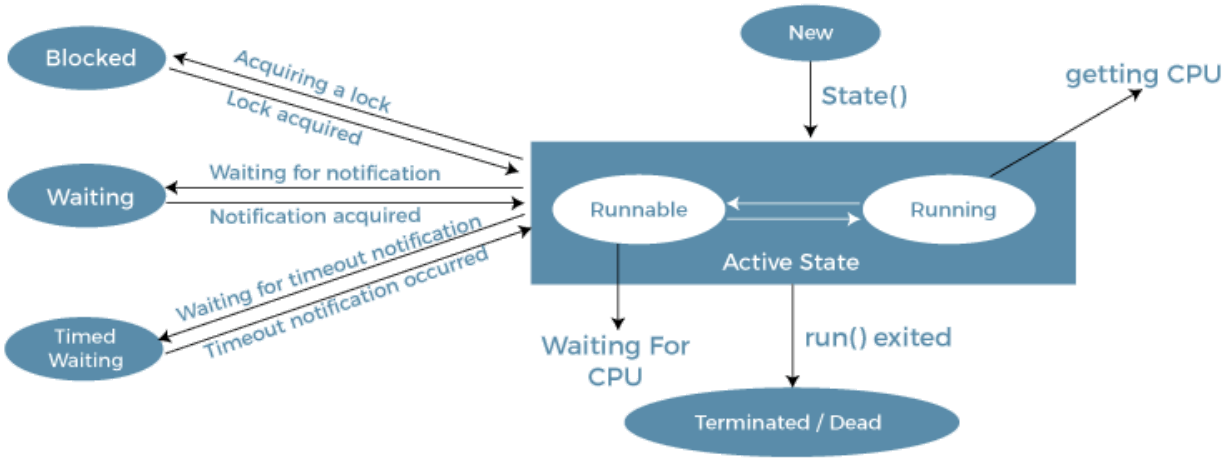
for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Java Threads | How to create a thread in Java

There are two ways to create a thread:

1. By extending the Thread class and overriding its run() method
2. By implementing Runnable interface.

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.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

If the class extends the Thread class, the thread can be run by creating an instance of the class and call its start() method:

FileName: Multi.java

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

If the class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:

FileName: Multi3.java

```
class Multi3 implements Runnable{

public void run(){

System.out.println("thread is running...");

}



public static void main(String args[]){

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)

t1.start();

 }

}
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Now we will see some more programs.
Let us consider the following program without thread.

```
class A{
        public void show() {
```

```
                for(int i=1;i<10;i++) {
                        System.out.println("hi");
                }
        }
}
class B{
        public void show() {
                for(int i=1;i<10;i++) {
                        System.out.println("hello");
                }
        }
}
public class ThreadProgram1 {
        public static void main(String[] args) {
                A objA = new A();
                B objB = new B();

                objA.show();
                objB.show();
        }
}
```

Output:

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
hello
hello
hello
```

```
hello
hello
hello
hello
hello
hello
```

Let us now extend the Thread class in declaration of the both class A and B as follows:

```
class A extends Thread{
	public void show() {
		for(int i=1;i<10;i++) {
			System.out.println("hi");
		}
	}
}
class B extends Thread{
	public void show() {
		for(int i=1;i<10;i++) {
			System.out.println("hello");
		}
	}
}
public class ThreadProgram1 {
	public static void main(String[] args) {
		A objA = new A();
		B objB = new B();

		objA.show();
		objB.show();
	}
}
```

Output:

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
hello
hello
hello

hello
hello
hello
hello
hello
hello
```

It has still the same output, so the concurrency has still not been achieved.

The run() method should be overridden in the class that extends Thread in order to achieve concurrency.

Let us now again change the program to the following:

```
class A extends Thread{
        public void run() {
                for(int i=1;i<10;i++) {
                        System.out.println("hi");
                }
```

```
	}
}
class B extends Thread{
	public void run() {
		for(int i=1;i<10;i++) {
			System.out.println("hello");
		}
	}
}
public class ThreadProgram1 {
	public static void main(String[] args) {
		A objA = new A();
		B objB = new B();

		objA.run();
		objB.run();
	}
}
```

Output:

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
hello
hello
hello

hello
```

```
hello
hello
hello
hello
Hello
```

We still have the same output, run here still runs in the same thread not in any separate thread. We need to use the start method inside the main method as follows.

```
….
…

public static void main(String[] args) {
                A objA = new A();
                B objB = new B();

                objA.start();
                objB.start();
        }

…..
…..
```

Output:

```
hi
hi
hi
hi
hi
hi
hi
hi
hi
```

hello
hello
hello

hello
hello
hello
hello
hello
Hello

This still produces the same output, the reason may be the small number of iterations (10) in the for loop. Let's increase it and see what happens below.

```
class A extends Thread{
	public void run() {
		for(int i=1;i<100;i++) {
			System.out.println("hi");
		}
	}
}
class B extends Thread{
	public void run() {
		for(int i=1;i<100;i++) {
			System.out.println("hello");
		}
	}

….
….
```

Output:

…..
…..

hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hi
hi
hi
hello
Hello
…..
…..

Now we see the concurrency as hello occurs several times and hi occurs only thrice in a part of the whole sequence. The number of occurrences of hi and hello are different in different part of the output.

If want to see the priority of the thread we can use,

System.***out***.println(objA.getPriority());

In the above program, the thread priority is likely to be 5 by default which is medium priority. We can change the priority and set it to some other value between 1 and 10 where 1 is least priority and 10 is highest priority. However, even if we assign a priority for a thread, this is not guaranteed as this is decided by the scheduler. Setting priority is just the way of suggesting to the scheduler that a certain thread needs to have certain priority. The ultimate decision is upon the scheduler that decides.

A priority the highest value can be assigned as follows:

```
objA.setPriority(Thread.MAX_PRIORITY);
```

Or,

```
objA.setPriority(10);
```

But the output is still similar to the previous one. This means that even if we set priority to 10 the scheduler did not grant it.

Now our aim is to get the output where one “hi” is followed by one “hello” to show perfect concurrency. We can try to achieve this by introducing the sleep method using the Thread class after each iteration inside the loop of both of the run methods.

```
….
…
                    try {
                            Thread.sleep(10);
                    } catch (InterruptedException e) {
                            e.printStackTrace();
                    }
….
…..
```

Doing this gives much better output as follows:

```
…
….
hi
hello
hi
```

```
hello
hi
hello
hello
hi
hello
….
…..
```

However, this is still not perfect output but still much better than previous ones.

Now, instead of extending a thread we will do the same program by implementing the runnable interface.

```
class A1 implements Runnable{
	public void run() {
		for(int i=1;i<10;i++) {
			System.out.println("hi");
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
class B1 implements Runnable{
	public void run() {
		for(int i=1;i<10;i++) {
			System.out.println("hello");
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
```

```
}
public class ThreadProgram2 {
	public static void main(String[] args) {
		Runnable objA = new A1();
		Runnable objB = new B1();

		Thread t1 = new Thread(objA);
		Thread t2 = new Thread(objB);

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

We now replace the above code with anonymous inner class as follows:

```
public class ThreadProgram3 {
	public static void main(String[] args) {
		Runnable objA = new Runnable() {
			public void run() {
				for(int i=1;i<10;i++) {
					System.out.println("hi");
					try {
						Thread.sleep(10);
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
				}
			}
		};
		Runnable objB = new Runnable() {
			public void run() {
				for(int i=1;i<10;i++) {
					System.out.println("hello");
					try {
						Thread.sleep(10);
```

```
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
};

Thread t1 = new Thread(objA);
Thread t2 = new Thread(objB);

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

We can also convert the above program to a different version using lambda expression, that performs the same operation.

```
public class ThreadProgram4 {
    public static void main(String[] args) {
        Runnable objA = ()-> {
            {
                for(int i=1;i<10;i++) {
                    System.out.println("hi");
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        Runnable objB = ()->{ {
                for(int i=1;i<10;i++) {
                    System.out.println("hello");
                    try {
                        Thread.sleep(10);
```

```
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
};

Thread t1 = new Thread(objA);
Thread t2 = new Thread(objB);

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

We can also add some statements in the run methods to see how the output look like.

```
public class ThreadProgram4 {
    public static void main(String[] args) {
        Runnable objA = ()-> {
            {
                Integer x = 10;
                system.out.println("Is x integer: "+(x instanceof Integer));
                for(int i=1;i<10;i++) {
                    System.out.println("hi");
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };
        Runnable objB = ()->{ {
                Integer  y= 10;
            System.out.println("Y is member of: "+y.getClass());
                for(int i=1;i<10;i++) {
```

```
                                        System.out.println("hello");
                                        try {
                                                Thread.sleep(10);
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }
                };

                Thread t1 = new Thread(objA);
                Thread t2 = new Thread(objB);

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