Introduction

Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel.

For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially.

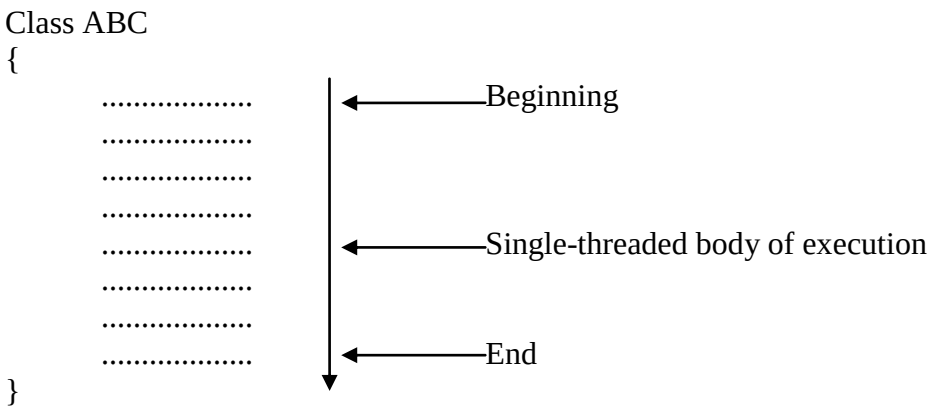

Figure: Single-threaded program

A unique property of java is its support for multithreading. That is, java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as *thread* that runs in parallel to others.

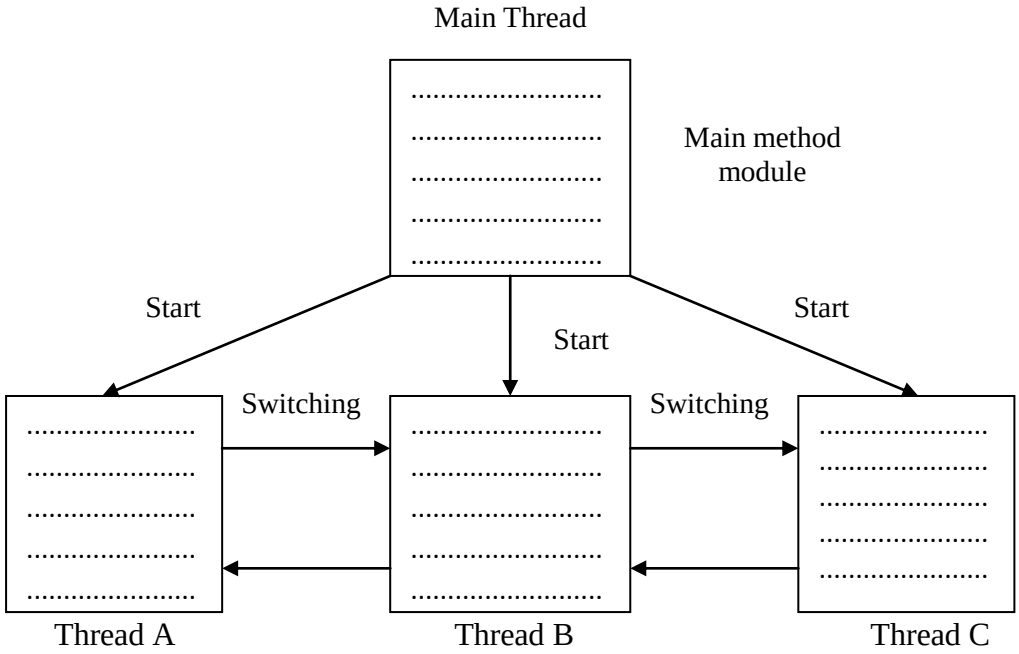

Figure: A Multithreaded program

- A program that contains multiple flows of control is known as multithreaded program.
- The ability of a language to support multithreads is referred to as concurrency.
- Threads in java are subprograms of a main application program and share the same memory space, they are known as lightweight threads or lightweight processes.
- If all the threads are running on a single processor, the flow of execution is shared between the threads.

Difference between Process-based and Thread-based Multitasking.

Process-based Multitasking	Thread-based Multitasking
In process based multitasking a process or a program is the smallest unit.	In thread base multitasking a thread is the smallest unit.
In process based multitasking two or more processes and programs can run concurrently.	In thread based multitasking a two or more threads can run concurrently.
Processes are heavy weight.	Threads are light weight.
Process requires its own address space.	Threads share same address space.
Process is a bigger unit.	Thread is a smaller unit.
Process to Process communication is expensive.	Thread to Thread communication is not expensive.

Life Cycle of a Thread

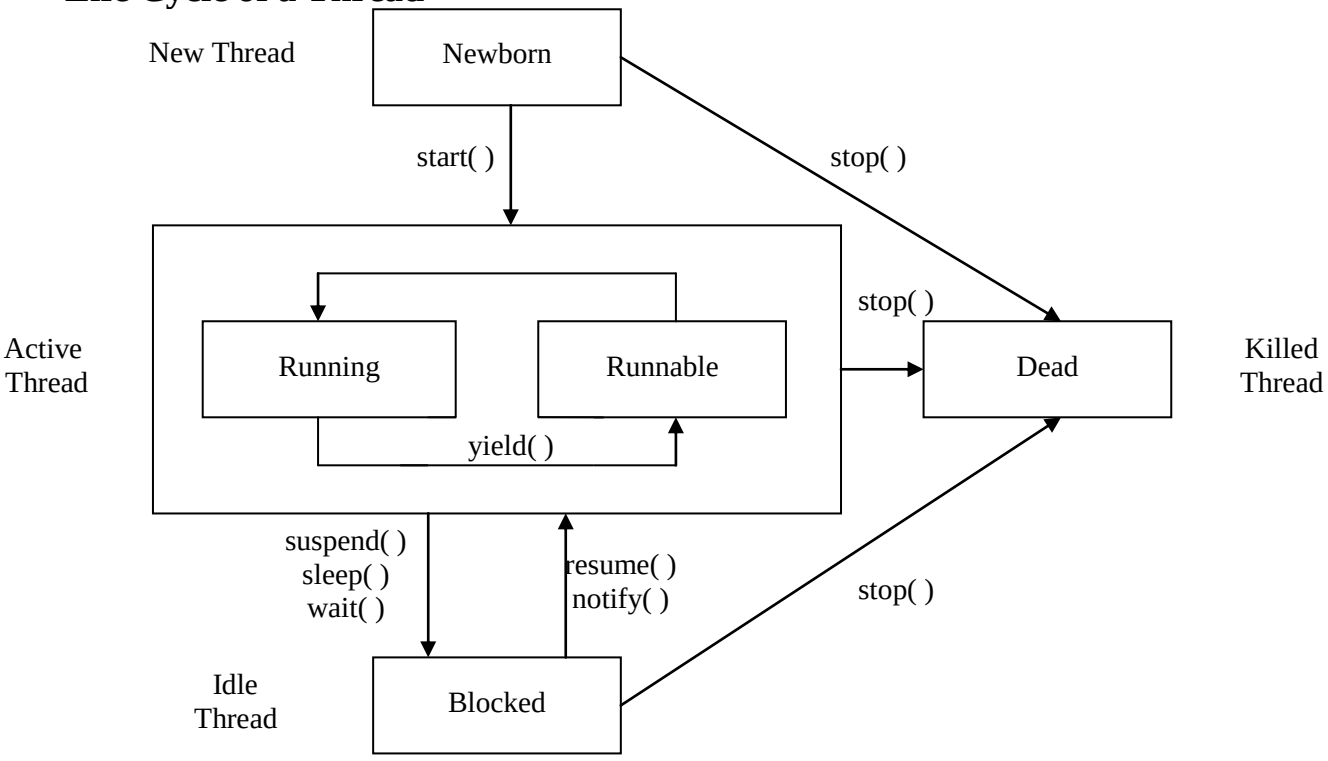

Figure: State transition diagram of a thread

A thread is always in one of these five states.

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

It can move from one state to another via a variety of ways as shown in fig.

Newborn State

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- Schedule it for running using ***start()*** method.
- Kill it using ***stop()*** method.

If scheduled, it moves to the runnable state.

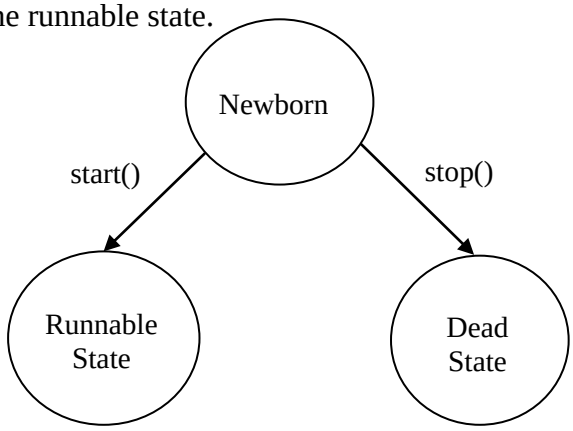

Figure: Scheduling a newborn thread

Runnable State

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the *yield()* method.

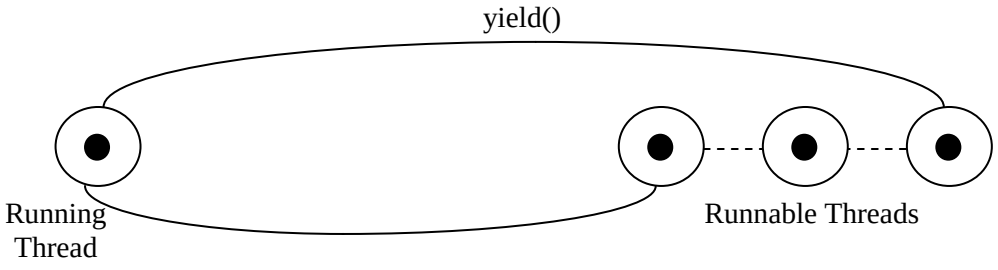

Figure: Relinquishing control using yield() method

Running State

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is pre-empted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

- It has been suspended using *suspend()* method. A suspended thread can be revived by using the *resume()* method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

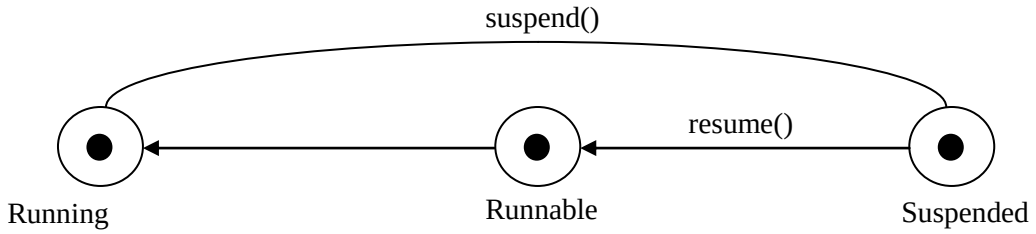

Figure: Relinquishing control using suspend() method

- It has been made to sleep. We can put a thread to sleep for a specified time-period using the method *sleep(time)* where time is milliseconds. This means that the thread is out of the queue during its time period. The thread re-enters the runnable state as soon as this time period is elapsed.

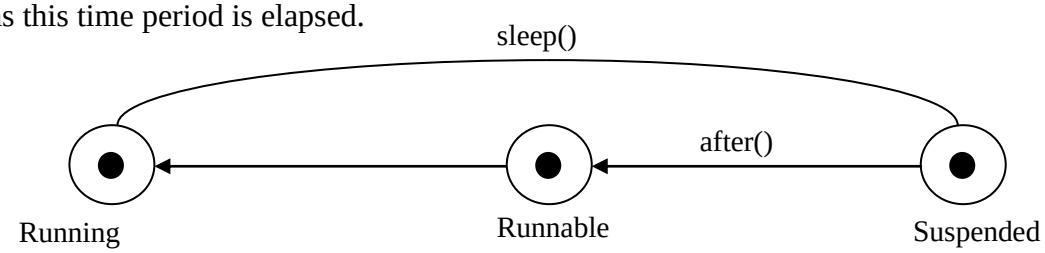

Figure: Relinquishing control using sleep() method

- It has been told to wait until some event occurs. This is done using the *wait()* method. The thread can be scheduled to run ag *notify()* method.

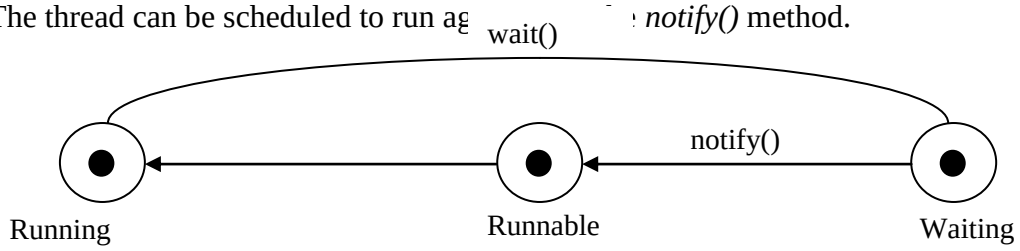

Figure: Relinquishing control using wait() method

Blocked State

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or

waiting in order to satisfy certain requirements. A blocked thread is considered 'not runnable' but not dead and therefore fully qualified to run again.

Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its *run()* method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in blocked condition.

Creating Threads

A new thread can be created in two ways:

1. By extending a 'Thread' class
2. Implementing the 'Runnable' interface

By extending a 'Thread' class

It includes the following steps:
Step-1: Declare the class as extending the 'Thread' class.
Step-2: Implement the *run()* method that is responsible for executing the sequence of code that the thread will execute.
Step-3: Create a thread object and call the *start()* method to initiate the thread execution.

Example:

Example1.java

```
class A extends Thread
{
	public void run()
	{
		for(int i=1;i<=20;i++)
		{
			if(i%2==0)
			{
				System.out.println("The even thread : "+i);
			}
		}
		System.out.println("Exit from the even thread");
	}
}
class B extends Thread
{
	public void run()
	{
		for(int j=1;j<=20;j++)
		{
			if(j%2!=0)
			{
				System.out.println("The odd thread : "+j);
			}
```

```
                }
                System.out.println("Exit from the odd thread");
        }
}
class Example1
{
        public static void main(String args[])
        {
                A even=new A();
                even.start();
                B odd=new B();
                odd.start();
                System.out.println("Exit from the main thread");
        }
}
```

Output

```
The even thread : 2
The even thread : 4
Exit from the main thread
The odd thread : 1
The even thread : 6
The odd thread : 3
The even thread : 8
The odd thread : 5
The even thread : 10
The odd thread : 7
The even thread : 12
The odd thread : 9
The even thread : 14
The odd thread : 11
The even thread : 16
The odd thread : 13
The even thread : 18
The odd thread : 15
The even thread : 20
The odd thread : 17
Exit from the even thread
The odd thread : 19
Exit from the odd thread
```

Implementing the 'Runnable' Interface

It includes the following steps:
Step-1: Declare the class as implementing the 'Runnable' interface.
Step-2: Implement the *run()* method.
Step-3: Create a thread by defining an object that is instantiated from this 'runnable' class.
Step-4: Call the thread's *start()* method to run the thread.

Example:

Example2.java

```
class A implements Runnable
{
	public void run()
	{
		for(int i=1;i<=20;i++)
		{
			if(i%2==0)
			{
				System.out.println("Even thread A = "+i);
			}
		}
		System.out.println("Exit from even body");
	}
}
class B implements Runnable
{
	public void run()
	{
		for(int i=1;i<=20;i++)

		{
			if(i%2!=0)
			{
				System.out.println("Odd thread B = "+i);
			}
		}
		System.out.println("Exit from odd body");
	}
}
class Example2
{
	public static void main(String args[])
	{
		A obj1=new A();
		Thread even=new Thread (obj1);
		B obj2=new B();
		Thread odd=new Thread (obj2);
		even.start();
		odd.start();
		System.out.println("Exit from the main thread");
	}
}
```

Output

```
Exit from the main thread
Even thread A = 2
Odd thread B = 1
```

```
Odd thread B = 3
Even thread A = 4
Odd thread B = 5
Even thread A = 6
Odd thread B = 7
Even thread A = 8
Odd thread B = 9
Even thread A = 10
Odd thread B = 11
Even thread A = 12
Odd thread B = 13
Even thread A = 14
Odd thread B = 15
Even thread A = 16
Even thread A = 18
Odd thread B = 17
Even thread A = 20
Odd thread B = 19
Exit from even body
Exit from odd body
```

Difference between 'implements Runnable' and 'extends Thread'

Implementing the 'Runnable' interface is preferred over extending a 'Thread' class because of the following reasons:

- *Inheritance:* The limitation with "extends Thread" approach is that if we extend Thread, we cannot extend anything else because Java does not support multiple inheritance.

  On the other hand, implementing the Runnable interface gives us the choice to extend any class.

- *Reusability:* In "implements Runnable", we are creating a different Runnable class for a specific behaviour. It gives us the freedom to reuse the specific behaviour whenever required.

  On the other hand, "extends Thread" contains both thread and job specific behaviour code. Hence once thread completes execution, it cannot be restart again.

- *Loosely-coupled*: "implements Runnable" makes the code loosely-coupled and easier to read because the code is split into two classes. Thread class for the thread specific code and your Runnable implementation class for your job that should be run by a thread code.

  On the other hand "extends Thread" makes the code tightly coupled. Single class contains the thread code as well as the job that needs to be done by the thread.

- *Functions overhead:* "extends Thread" means inheriting all the functions of the Thread class which we may do not need. *Job* can be done easily by Runnable without the Thread class functions overhead.

Questions asked in semester paper

Question-What is thread? How to create a thread in java?

[2015-2016]

Question-What is a thread? How a thread is created? Differentiate between process based multitasking and thread based multitasking.

[2006-2007]

Question-Java has two ways to create child threads, what are these two methods? Explain them. Which of the two, do you think is better and why?

[2005-2006]

Question-What is the difference between Process-based and Thread-based multitasking? Give the two ways by which a thread can be created in Java. Write a short program for the creation of a new thread by extending the thread class.

[2003-2004]

Question-What do you mean by thread States? Name the possible states of thread and explain them in short.

[2002-2003]