CS20004: Object Oriented Programming using Java

Lec- 24

In this Discussion . . .

- Multithreading
  - Synchronization
  - Deadlock
  - Various Conclusion
    - To stop a Thread
    - Suspend & resume of a thread
  - Life cycle of a Thread

Synchronization

- **“Synchronized” keyword is applicable for methods and blocks but not for classes and variables**.
- If a method or block is declared with the “synchronized” keyword, then at any time only one thread is allowed to execute that method or block on the given object.
- The main **advantage** of synchronized keyword is that we can resolve inconsistency problems.

Synchronization

- **However, the main disadvantage of synchronized keyword is it increases waiting time of the Thread, thereby affecting performance of the system.**
- Hence if there is no specific requirement, then it is not recommended to use synchronized keyword.

Internally, the synchronization concept is implemented by using the lock concept.

Synchronization

- 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 then it has to:
  - first get the lock of that object.
  - Once a Thread gets the lock of that object then it is allowed to execute any synchronized method on that object.
  - Once the synchronized method execution completes then automatically the Thread releases lock.

Synchronization

- While a Thread is 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 and not on methods.

Synchronization Example

```
class Display
{
	public synchronized void wish(String name)
	{
		for(int i=0;i<5;i++)
		{
			System.out.print("good morning:");
			try
			{
				Thread.sleep(1000);
			}
		catch (InterruptedException e)
		{}
		System.out.println(name);
		}
	}
}

class MyThread extends Thread
{
	Display d;
	String name;
	MyThread(Display d,String name)
	{
		this.d=d;
		this.name=name;
	}
	public void run()
	{
		d.wish(name);
	}
}


class SynchronizedDemo
{
	public static void main(String[] args)
	{
		Display d1=new Display();
		MyThread t1=new MyThread(d1,"Sourajit");
		MyThread t2=new MyThread(d1,"Behera");
		t1.start();
		t2.start();
	}
}
```

Synchronization Example - Case I

```
class Display
{
	public void wish(String name)
	{
		for(int i=0;i<5;i++)
		{
			System.out.print("good morning:");
			try
			{
				Thread.sleep(1000);
			}
		catch (InterruptedException e)
		{}
		System.out.println(name);
		}
	}
}
```

If we are not declaring wish() method as synchronized, then both Threads will be executed simultaneously and we will get irregular output.

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java SynchronizedDemo
good morning:good morning:Behera
good morning:Sourajit
good morning:Behera
good morning:Sourajit
good morning:Behera
good morning:Sourajit
good morning:Behera
good morning:Sourajit
good morning:Behera
Sourajit
```

Synchronization Example - Case II

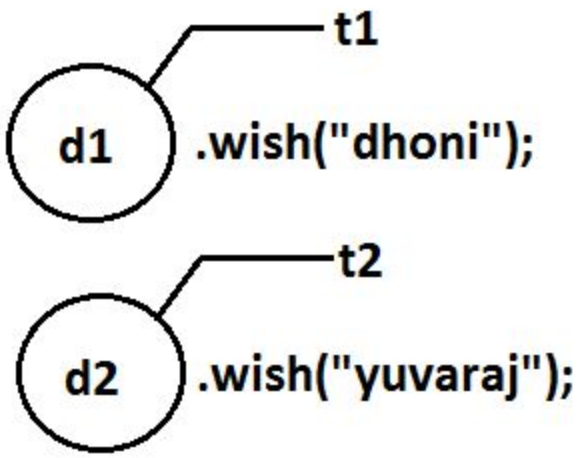
```
class Display
{
	public synchronized void wish(String name)
	{
		for(int i=0;i<5;i++)
		{
			System.out.print("good morning:");
			try
			{
				Thread.sleep(1000);
			}
		catch (InterruptedException e)
		{}
		System.out.println(name);
		}
	}
}
```

If we declare wish() method as synchronized then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait. In this case we will get regular output

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java SynchronizedDemo
good morning:Sourajit
good morning:Sourajit
good morning:Sourajit
good morning:Sourajit
good morning:Sourajit
good morning:Behera
good morning:Behera
good morning:Behera
good morning:Behera
good morning:Behera
```

Synchronization Example - Case III

```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"dhoni");
MyThread t2=new MyThread(d2,"yuvaraj");
t1.start();
t2.start();
```



- **Even though we declared wish() method as synchronized but we will get irregular output in this case, because both Threads are operating on different objects.**

If multiple threads are operating on multiple objects, then there is no impact of Synchronization.

If multiple threads are operating on same java objects then synchronized concept is required or applicable.

Class-level Lock

- Every class in java has a unique lock.
- If a Thread wants to execute a static synchronized method, then it requires a class-level lock.
- Once a Thread gets a class-level lock then it is allowed to execute any static synchronized method of that class.
- While one Thread executes any static synchronized method, the remaining Threads are not allow to execute any static synchronized method of that class simultaneously.

Class-level Lock

- **However**, remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.

Class-level lock and object lock are both different things and there exists no relationship between these two.

Synchronized Block

- It is never advisable to designate an entire method as synchronized if only a small number of lines of code need to be synchronized; **instead**, those few lines must be enclosed in a synchronized block.
- The primary benefit of a synchronized block over a synchronized method is that it increases system performance and decreases thread waiting times.

Synchronized Block

- We can declare a synchronized block in the following way to obtain the lock of the current object. Only the thread that has obtained a lock on the current object is permitted to run this block.

```
Synchronized(this){ }
```

- We must declare a synchronized block in the manner shown below in order to obtain the lock of a specific object, 'b'. Only the thread that has obtained the lock on the 'b' object is permitted to run this block.

```
Synchronized(b){ }
```

Synchronized Block

- We must define a synchronized block as follows in order to obtain a class level lock.

```
Synchronized(Display.class){ }
```

Threads are only permitted to execute this block if they have a class level lock on the display.

• We can send an object reference or a "class file" as the argument to the synchronized block. • But we cannot pass primitive values as an argument [since the lock notion only applies to objects and classes, not to primitives].	**Example:** Int x=b; Synchronized(x){} Output: Compile time error. Unexpected type. Found: int Required: reference

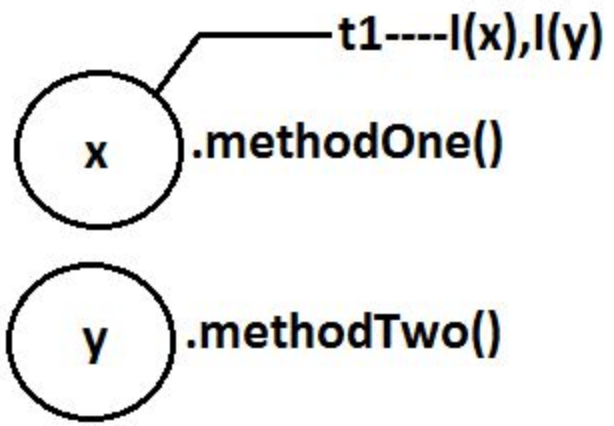
Synchronization

- Can a Thread can hold more than one lock at a time?

  Ans - Yes, but of course from different objects.

```
class X
{
      synchronized void methodOne()
      {
            Y y=new Y();
            y.methodTwo();
      }
}
```

```
class Y
{
      synchronized void methodTwo()
      {}
}
```



Synchronization Statement

- Synchronized statements are those that are contained in synchronized blocks and synchronized methods.

Deadlock

- Deadlock occurs when two or more threads are waiting on one another interminably. This kind of circumstance is known as infinite waiting.
- Deadlock cannot be resolved, however there are a number of preventative (or avoidance) strategies that can be used.

We must exercise extra caution anytime we utilize synchronized keywords because they are the source of deadlock.

Deadlock

```
class A
{
	public synchronized void foo(B b)
	{
System.out.println("Thread1 starts execution of foo() method");
		try
		{
			Thread.sleep(2000);
		}
		catch (InterruptedException e)
		{}
		System.out.println("Thread1 trying to call b.last()");
		b.last();
	}
	public synchronized void last()
	{
		System.out.println("inside A, this is last()method");
	}
}
class B
{
	public synchronized void bar(A a)
	{
System.out.println("Thread2 starts execution of bar() method");
		try
		{
			Thread.sleep(2000);
		}
		catch (InterruptedException e)
		{}
		System.out.println("Thread2 trying to call a.last()");
		a.last();
	}
	public synchronized void last()
	{
		System.out.println("inside B, this is last() method");
	}
}
class DeadLock implements Runnable
{
	A a=new A();
	B b=new B();
	DeadLock()
	{
		Thread t=new Thread(this);
		t.start();
		a.foo(b);//main thread
	}
	public void run()
	{
		b.bar(a);//child thread
	}
	public static void main(String[] args)
	{
		new DeadLock();//main thread
	}
}
```

Deadlock

Output:

```
Thread1 starts execution of foo() method
Thread2 starts execution of bar() method
Thread2 trying to call a.last()
Thread1 trying to call b.last()
//here cursor always waiting.
```

We won't get DeadLock if we delete at least one synchronized keystroke.Thus, the only cause of DeadLock is synchronized keywords, which we must use with caution while utilizing them.

- **How to kill a Thread in the middle of the line?**

  - We can call **stop()** method to stop a Thread in the middle. Then it will be entering into dead state immediately.
  - **Method: public final void stop();**

- **suspend and resume methods**

  - Through the usage of the **suspend()** method, one thread can temporarily pause another thread.
  - By employing the resume() method, a thread can bring a suspended thread back to life, allowing it to continue executing.
    - **Methods:**
      - **public final void suspend();**
      - **public final void resume();**

Lifecycle of a Thread

- In Java, the life cycle of Thread goes through **various states**. These states represent different stages of execution and include the following:
    - New or born state
    - Runnable state
    - Running state
    - Blocked state
    - Waiting state
    - Timed Waiting state
    - Terminated state

Lifecycle of a Thread (Contd.)

- New or born state
  - Description: Creating a new thread using the Thread class constructor.
  - Use case: Creating a new thread to perform a background task while the main Thread continues with other operations

- Runnable state
  - Description: After calling the start() method on a thread, it enters the runnable state.
  - Use case: Multiple threads competing for CPU time to perform their tasks concurrently.

Lifecycle of a Thread (Contd.)

- Running state
  - Description: When a thread executes its code inside the run() method.
  - Use case: A thread executing a complex computation or performing a time-consuming task.

- Blocked state
  - Description: When a thread tries to access a synchronized block or method, but another thread already holds the lock.
  - Use case: Multiple threads accessing a shared resource can only be obtained by a single Thread, such as a database or a file.

Lifecycle of a Thread (Contd.)

- Waiting state
  - Description: Using the wait method inside a synchronized block, a thread can wait until another thread calls the notify() or notifyAll() methods to wake it up.
  - Use case: Implementing the producer-consumer pattern, where a thread waits for a specific condition to be met before continuing its execution

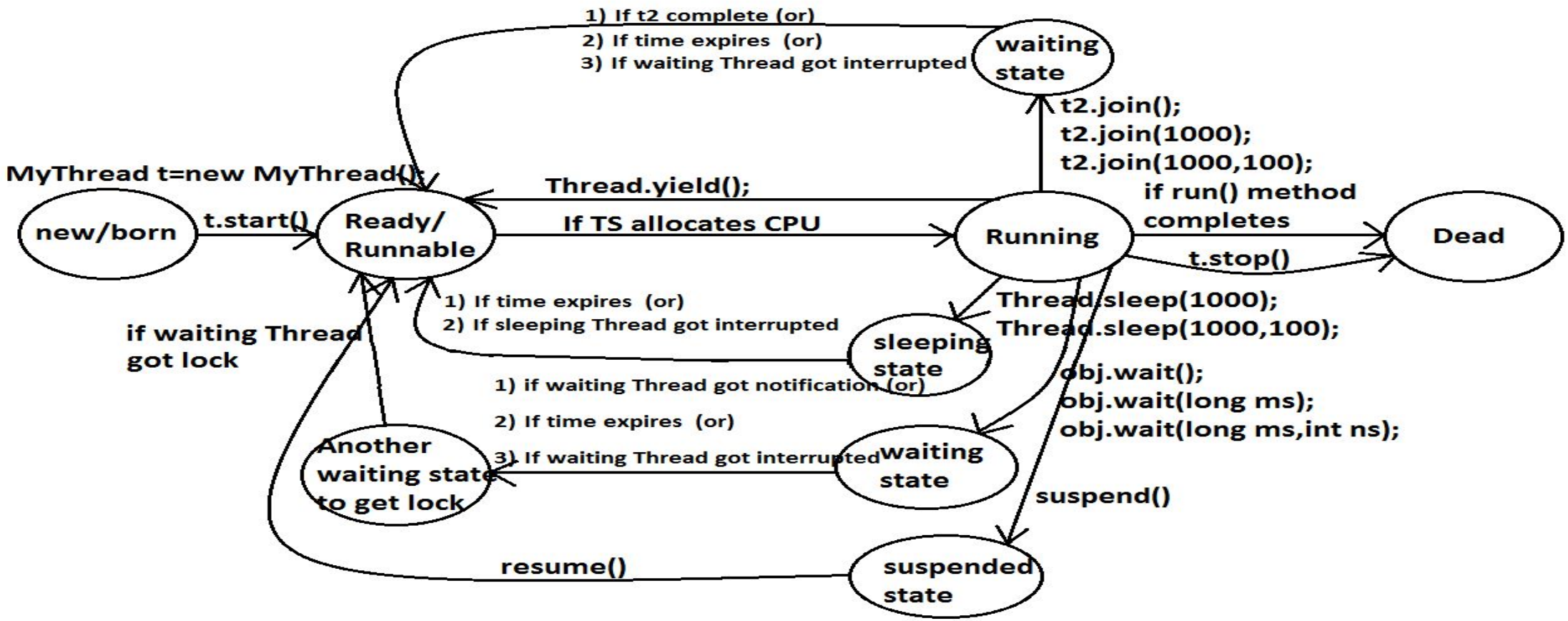
- Timed Waiting state
  - Description: Using methods like sleep(milliseconds) or join(milliseconds) causes a thread to enter the timed waiting state for the specified duration.
  - Use case: Adding delays between consecutive actions or waiting for the completion of other threads before proceeding.

Lifecycle of a Thread (Contd.)

- Terminated state
  - Description: When the run() method finishes its execution or when the stop() method is called on the Thread.
  - Use case: Completing a task or explicitly stopping a thread's execution.

- These descriptions demonstrate how threads can handle various scenarios in Java applications, allowing for concurrent and parallel execution of tasks.
- Understanding the different thread states helps in designing efficient and responsive multithreaded applications.

Lifecycle of a Thread



References

1. https://herovired.com/learning-hub/blogs/life-cycle-of-thread-in-java/
2.