UNIT-2

MULTI-THREADING

What is Thread in java

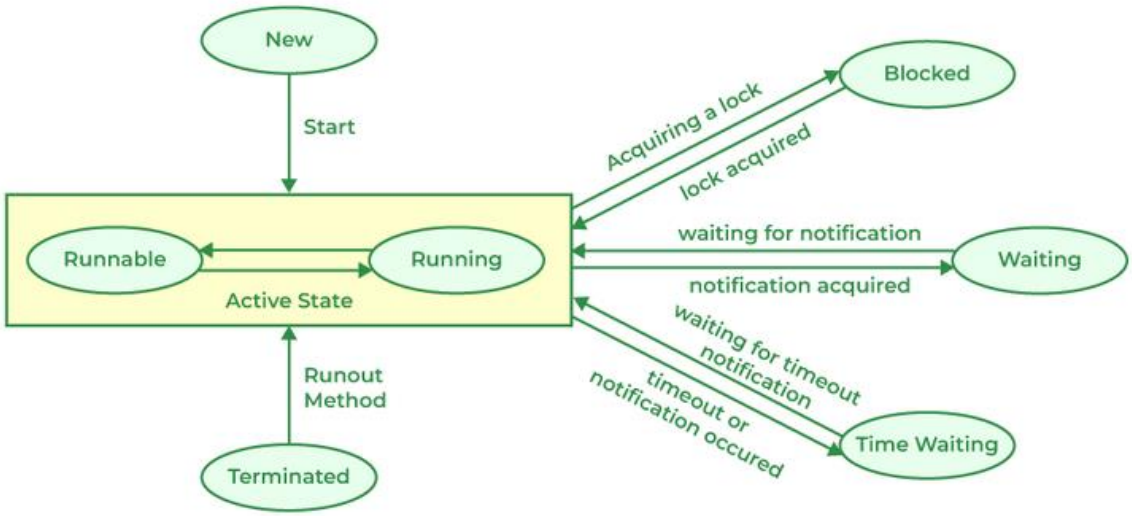
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

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

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New State
2. Runnable State
3. Blocked State
4. Waiting State
5. Timed Waiting State
6. Terminated State

The diagram shown below represents various states of a thread at any instant in time.



States of Thread in its Lifecycle

Life Cycle of a Thread

There are multiple states of the thread in a lifecycle as mentioned below:

1. ***New Thread:*** *When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.*
2. ***Runnable State:*** *A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.*
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads

can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

3. ***Blocked:*** *The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.*
4. ***Waiting state:*** *The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.*
5. ***Timed Waiting:*** *A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.*
6. ***Terminated State:*** *A thread terminates because of either of the following reasons:*
   - *Because it exits normally. This happens when the code of the thread has been entirely executed by the program.*
   - *Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.*

Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

Ex

Extend Syntax

```
public class Main extends Thread {

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

Another way to create a thread is to implement the `Runnable` interface:

Implement Syntax

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Running Threads

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

Extend Example

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

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:

Implement Example

```
public class Main implements Runnable {

  public static void main(String[] args) {

    Main obj = new Main();

    Thread thread = new Thread(obj);

    thread.start();

    System.out.println("This code is outside of the thread");

  }

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY

3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
// Importing the required classes
import java.lang.*;

public class ThreadPriorityExample extends Thread
{

// Method 1
// Whenever the start() method is called by a thread
// the run() method is invoked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}

// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

// 1st Thread
// Displaying the priority of the thread
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
```

```
35.
36. // 3rd Thread
37. // // Display the priority of the thread
38. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
39.
40. // Setting priorities of above threads by
41. // passing integer arguments
42. th1.setPriority(6);
43. th2.setPriority(3);
44. th3.setPriority(9);
45.
46. // 6
47. System.out.println("Priority of the thread th1 is : " + th1.getPriority());
48.
49. // 3
50. System.out.println("Priority of the thread th2 is : " + th2.getPriority());
51.
52. // 9
53. System.out.println("Priority of the thread th3 is : " + th3.getPriority());
54.
55. // Main thread
56.
57. // Displaying name of the currently executing thread
58. System.out.println("Currently Executing The Thread : " + Thread.currentThread().getN
    ame());
59.
60. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPrior
    ity());
61.
62. // Priority of the main thread is 10 now
63. Thread.currentThread().setPriority(10);
64.
65. System.out.println("Priority of the main thread is : " + Thread.currentThread().getPrior
    ity());
66. }
67. }
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

Java Threads | 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.

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.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.

5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

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().

1. **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

FileName: Multi.java

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8.  }
9. }
```

Output:

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

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
9. t1.start();
10.  }
11. }
```

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.

Example-1 using Thread class sleep method(create thread by extending Thread class)

```
class Multi1 extends Thread{

public void run(){

int i;

for(i=1;i<5;i++)

{

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

try

{

Thread.sleep(400);

}

catch(Exception e)

{

}

}

}


public static void main(String args[]){

Multi1 t1=new Multi1();

t1.start();

 }

}
```

Example-2->implements Runnable interface and using thread class sleep() method(create thread using Runnable interface)

```
class Multi3 implements Runnable{

public void run(){

int i;

for(i=1;i<10;i++)

{

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

try

{

Thread.sleep(4000);

}

catch(Exception e)

{

}

}

}

  public static void main(String args[]){

Multi3 m1=new Multi3();

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

t1.start();

 }

}
```

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block

3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
1. class Table{
2. void printTable(int n){//method not synchronized
3.   for(int i=1;i<=5;i++){
4.     System.out.println(n*i);
5.     try{
6.      Thread.sleep(400);
7.     }catch(Exception e){System.out.println(e);}
8.   }
9.
10.  }
11. }
12.
13. class MyThread1 extends Thread{
14. Table t;
15. MyThread1(Table t){
16. this.t=t;
17. }
18. public void run(){
19. t.printTable(5);
20. }
21.
22. }
```

```
23. class MyThread2 extends Thread{
24. Table t;
25. MyThread2(Table t){
26. this.t=t;
27. }
28. public void run(){
29. t.printTable(100);
30. }
31. }
32.
33. class TestSynchronization1{
34. public static void main(String args[]){
35. Table obj = new Table();//only one object
36. MyThread1 t1=new MyThread1(obj);
37. MyThread2 t2=new MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```
1. //example of java synchronized method
```

```
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.  }
12. }
13.
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. public class TestSynchronization2{
35. public static void main(String args[]){
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
```

41. }
42. }

Output:

```
5
 10
 15
 20
 25
100
200
300
400
500
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 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.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

Syntax

```
1. synchronized (object reference expression) {
2.   //code block
3. }
```

Example of Synchronized Block

Let's see the simple example of synchronized block.

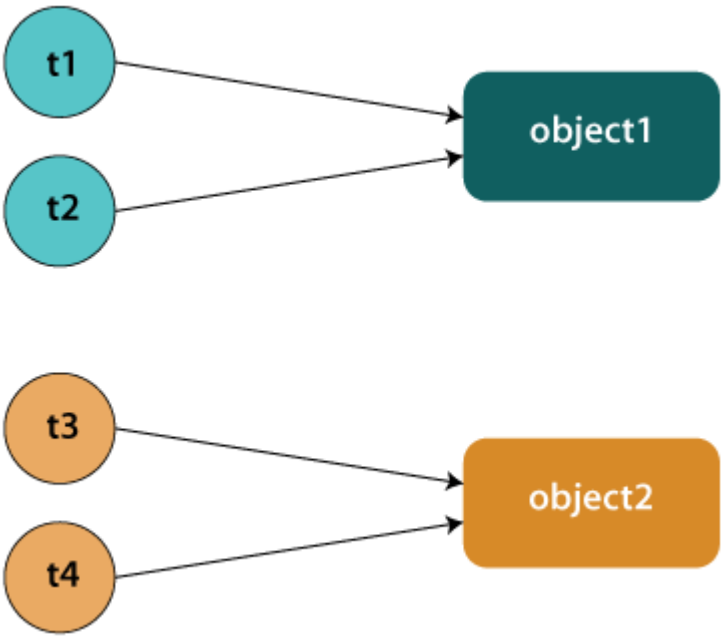
TestSynchronizedBlock1.java

```
1.  class Table
2.  {
3.   void printTable(int n){
4.     synchronized(this){//synchronized block
5.       for(int i=1;i<=5;i++){
6.        System.out.println(n*i);
7.        try{
8.         Thread.sleep(400);
9.        }catch(Exception e){System.out.println(e);}
10.      }
11.    }
12.  }//end of the method
13. }
14.
15. class MyThread1 extends Thread{
16. Table t;
17. MyThread1(Table t){
18. this.t=t;
19. }
20. public void run(){
21. t.printTable(5);
22. }
23.
24. }
25. class MyThread2 extends Thread{
26. Table t;
27. MyThread2(Table t){
28. this.t=t;
29. }
30. public void run(){
31. t.printTable(100);
32. }
33. }
34.
```

```
35. public class TestSynchronizedBlock1{
36. public static void main(String args[]){
37. Table obj = new Table();//only one object
38. MyThread1 t1=new MyThread1(obj);
39. MyThread2 t2=new MyThread2(obj);
40. t1.start();
41. t2.start();
42. }
43. }
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

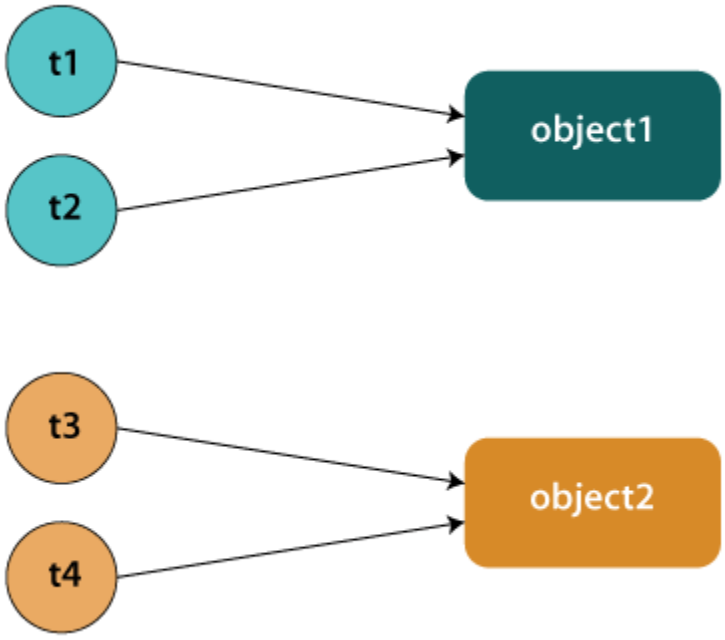
TestSynchronization4.java


```
1. class Table
2. {
3.  synchronized static void printTable(int n){
4.    for(int i=1;i<=10;i++){
5.      System.out.println(n*i);
6.      try{
7.        Thread.sleep(400);
8.      }catch(Exception e){}
9.    }
10.  }
11. }
12. class MyThread1 extends Thread{
13. public void run(){
14. Table.printTable(1);
15. }
16. }
17. class MyThread2 extends Thread{
18. public void run(){
19. Table.printTable(10);
20. }
21. }
22. class MyThread3 extends Thread{
23. public void run(){
24. Table.printTable(100);
25. }
26. }
27. class MyThread4 extends Thread{
28. public void run(){
29. Table.printTable(1000);
30. }
31. }
32. public class TestSynchronization4{
```

```
33. public static void main(String t[]){
34. MyThread1 t1=new MyThread1();
35. MyThread2 t2=new MyThread2();
36. MyThread3 t3=new MyThread3();
37. MyThread4 t4=new MyThread4();
38. t1.start();
39. t2.start();
40. t3.start();
41. t4.start();
42. }
43. }
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

TestSynchronization4.java


```
1. class Table
2. {
3.  synchronized static void printTable(int n){
4.    for(int i=1;i<=10;i++){
5.      System.out.println(n*i);
6.      try{
7.        Thread.sleep(400);
8.      }catch(Exception e){}
9.    }
10.  }
11. }
12. class MyThread1 extends Thread{
13. public void run(){
14. Table.printTable(1);
15. }
16. }
17. class MyThread2 extends Thread{
18. public void run(){
19. Table.printTable(10);
20. }
21. }
22. class MyThread3 extends Thread{
23. public void run(){
24. Table.printTable(100);
25. }
26. }
27. class MyThread4 extends Thread{
28. public void run(){
29. Table.printTable(1000);
30. }
31. }
32. public class TestSynchronization4{
```

```
33. public static void main(String t[]){
34. MyThread1 t1=new MyThread1();
35. MyThread2 t2=new MyThread2();
36. MyThread3 t3=new MyThread3();
37. MyThread4 t4=new MyThread4();
38. t1.start();
39. t2.start();
40. t3.start();
41. t4.start();
42. }
43. }
44.
```

Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description

public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
1. public final void notify()
```

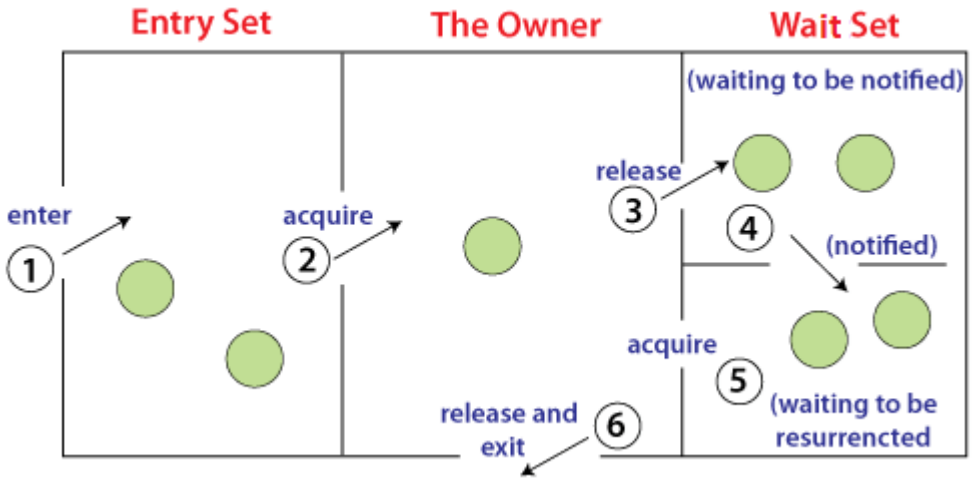
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
1. public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.

3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.