Java Multithreading

By

Praveen Naga

Blog: https://praveennagatech.blogspot.com

Facebook Group: https://www.facebook.com/groups/268426377837151

Github repo: https://github.com/praveennaga

Email: opraveennaga@gmail.com

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1

Multithreading concepts

- ✓ Process is an execution of a program.
- ✓ Thread is a single execution sequence in a process.

Each thread will have its own stack memory but single heap per process shared by all threads.

Java multithreading is a process of executing more than one thread simultaneously is known as multithreading in java.

The main purpose of multithreading is to achieve the concurrent execution.

Advantage of Multithreading in Java

- **Save Time**: By using multi-threading in java you can perform multiple operation at the same time so it saves time.
- **Doesn't Block Users**: Java multithreading doesn't block the user because thread are independent, if exception occurs in one thread it does not affect other thread so you can perform multiple operation at the same time

Thread

Thread in java is a light weight process and thread is a small part of a process or program.
It is used to achieve multitasking.
Threads have separate path of execution and threads are independent if exception occurs in one thread, it does not affect other threads. Thread share common memory area.

How many ways to create a thread

1. Extend the java.lang.Thread class

```
package com.praveen.multithreading.threadcreation;

public class SimpleThread extends Thread {

	public void run() {
		System.out.println("Starting " + currentThread().getName());
	}

	public static void main(String[] args) {
		Thread t1 = new SimpleThread();
		t1.setName("t1");
```

```
		Thread t2 = new SimpleThread();
		t2.setName("t2");
		t1.start();
		t2.start();

	}

}
```

Output

Starting t1

Starting t2

2. Implementing the java.lang.Runnable interface

```
package com.praveen.multithreading.threadcreation;

public class RunnableThread implements Runnable{

	public void run() {
		System.out.println("Starting " + Thread.currentThread().getName());
	}

	public static void main(String[] args) {
		RunnableThread rt1= new RunnableThread();
		RunnableThread rt2= new RunnableThread();
		Thread t1 = new Thread(rt1);
		t1.setName("t1");
		Thread t2 = new Thread(rt2);
		t2.setName("t2");
		t1.start();
		t2.start();
	}
}
```

Output

Starting t1

Starting t2

3. Implement the java.util.concurrent.callable interface

The executor framework which uses Runnable objects, unfortunately a runnable cannot return a result to the caller.
In case you expect your threads to return a computeable result you can use java.util.concurrent.callable interface.

Code

```
package com.praveen.multithreading.semaphore;

import java.util.concurrent.Semaphore;

public class SemaphoreExample {
	// max 4 people
	static Semaphore semaphore = new Semaphore(4);
	static class MyATMThread extends Thread {
	String name = "";
	MyATMThread(String name) {
	this.name = name;
	}
	public void run() {
	try {
	System.out.println(name + " : acquiring lock…");
	System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
	semaphore.acquire();
	System.out.println(name + " : got the permit!");
	try {
	for (int i = 1; i <= 5; i++) {
	System.out.println(name + " : is performing operation " + i + ", available
Semaphore permits : " + semaphore.availablePermits());
	// sleep 1 second
	Thread.sleep(1000);
	}
	} finally {
	// calling release() after a successful acquire()
	System.out.println(name + " : releasing lock…");
	semaphore.release();
	System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
	}
	} catch (InterruptedException e) {
	e.printStackTrace();
	}
	}
```

```
        }
        public static void main(String[] args) {
        System.out.println("Total available Semaphore permits : " +
semaphore.availablePermits());
        MyATMThread t1 = new MyATMThread("A"); t1.start();
        MyATMThread t2 = new MyATMThread("B"); t2.start();
        MyATMThread t3 = new MyATMThread("C"); t3.start();
        MyATMThread t4 = new MyATMThread("D"); t4.start();
        MyATMThread t5 = new MyATMThread("E"); t5.start();
        MyATMThread t6 = new MyATMThread("F"); t6.start();
        }}
```

Output

Starting 0 pool-1-thread-1

Starting 1 pool-1-thread-2

Starting 2 pool-1-thread-3

Starting 3 pool-1-thread-4

Starting 4 pool-1-thread-5

Starting 5 pool-1-thread-2

Starting 6 pool-1-thread-2

Starting 7 pool-1-thread-2

Starting 8 pool-1-thread-2

Starting 9 pool-1-thread-2

Finished all threads

Thread States

In Java, threads can have States. The Thread.State enum defines the different states that a Java thread can have.

This enum defines the following values

- ✓ **NEW**
- ✓ **RUNNABLE**
- ✓ **BLOCKED**
- ✓ **WAITING**
- ✓ **TIMED_WAITING**
- ✓ **TERMINATED**

States of a Java Thread

a) **NEW**
   This is the default state a thread gets when it is first created.
b) **RUNNABLE**
   As soon as a thread starts executing, it moves to the RUNNABLE state. Note that a thread that is waiting to acquire a CPU for execution is still in this state.
c) **BLOCKED**
   A thread moves to the BLOCKED state as soon as it gets blocked waiting for a monitor lock. This can happen in one of the following two ways
   - It's waiting to acquire a lock to enter a synchronized block/method.
   - It's waiting to reacquire the monitor lock of an object on which it invoked the Object.wait method.
d) **WAITING**
   A thread moves to this state as a result of invoking one of the following methods
   - Object.wait without a timeout
   - Thread.join without a timeout
   - LockSupport.park
e) **TIMED_WAITING**
   A thread moves to this state as a result of invoking one of the following methods
   - Thread.sleep
   - Object.wait with a timeout
   - Thread.join with a timeout
   - LockSupport.parkNanos
   - LockSupport.parkUntil
f) **TERMINATED**
   As soon as a thread terminates, it moves to this state

Let's see a sample program on Thread states

```
package com.praveen.multithreading.threadstates;

class SampleThread implements Runnable {

	public void run() {
		try {
			Thread.sleep(1500);
		} catch (InterruptedException ie) {
			System.out.println(ie.getMessage());
		}
		System.out.println("State of thread1 while it called join() on thread2 -- " +
ThreadStates.thread1.getState());
```

```
                try {
                        Thread.sleep(200);
                } catch (InterruptedException ie) {
                        System.out.println(ie.getMessage());
                }
        }

}

public class ThreadStates implements Runnable {

        public static Thread thread1;
        public static ThreadStates threadStates;

        public static void main(String[] args) {
                threadStates = new ThreadStates();
                thread1 = new Thread(threadStates);
                // thread1 created and it is in NEW state
                System.out.println("State of thread1 after creating it -- " +
thread1.getState());
                thread1.start();
                // thread1 moved to RUNNABLE state
                System.out.println("State of thread1 after calling start() -- " +
thread1.getState());
        }

        public void run() {
                SampleThread sampleThread = new SampleThread();
                Thread thread2 = new Thread(sampleThread);
                // thread2 created and it is in NEW state
                System.out.println("State of thread2 after creating it -- " +
thread2.getState());
                thread2.start();
                // thread2 moved to RUNNABLE state
                System.out.println("State of thread2 after calling start() -- " +
thread2.getState());

                // moving thread1 to WAITING state
                try {
                        // moving thread1 to TIMED_WAITING state
                        Thread.sleep(200);
                } catch (InterruptedException ie) {

                }
```

```
            // state of thread2 after calling sleep method
            System.out.println("State of thread2 after calling sleep() -- " +
thread2.getState());

            try {
                // waiting for thread2 to die
                thread2.join();
            } catch (InterruptedException ie) {

            }

            System.out.println("State of thread2 when its finished executing -- " +
thread2.getState());

        }

}
```

Output

State of thread1 after creating it -- NEW

State of thread1 after calling start() -- RUNNABLE

State of thread2 after creating it -- NEW

State of thread2 after calling start() -- RUNNABLE

State of thread2 after calling sleep() -- TIMED_WAITING

State of thread1 while it called join() on thread2 -- WAITING

State of thread2 when its finished executing – TERMINATED

Thread Lifecycle

Possible state transitions

As soon as thread is created, it moved to **NEW** state and once you call start() on the thread it moves to **RUNNABLE** state.

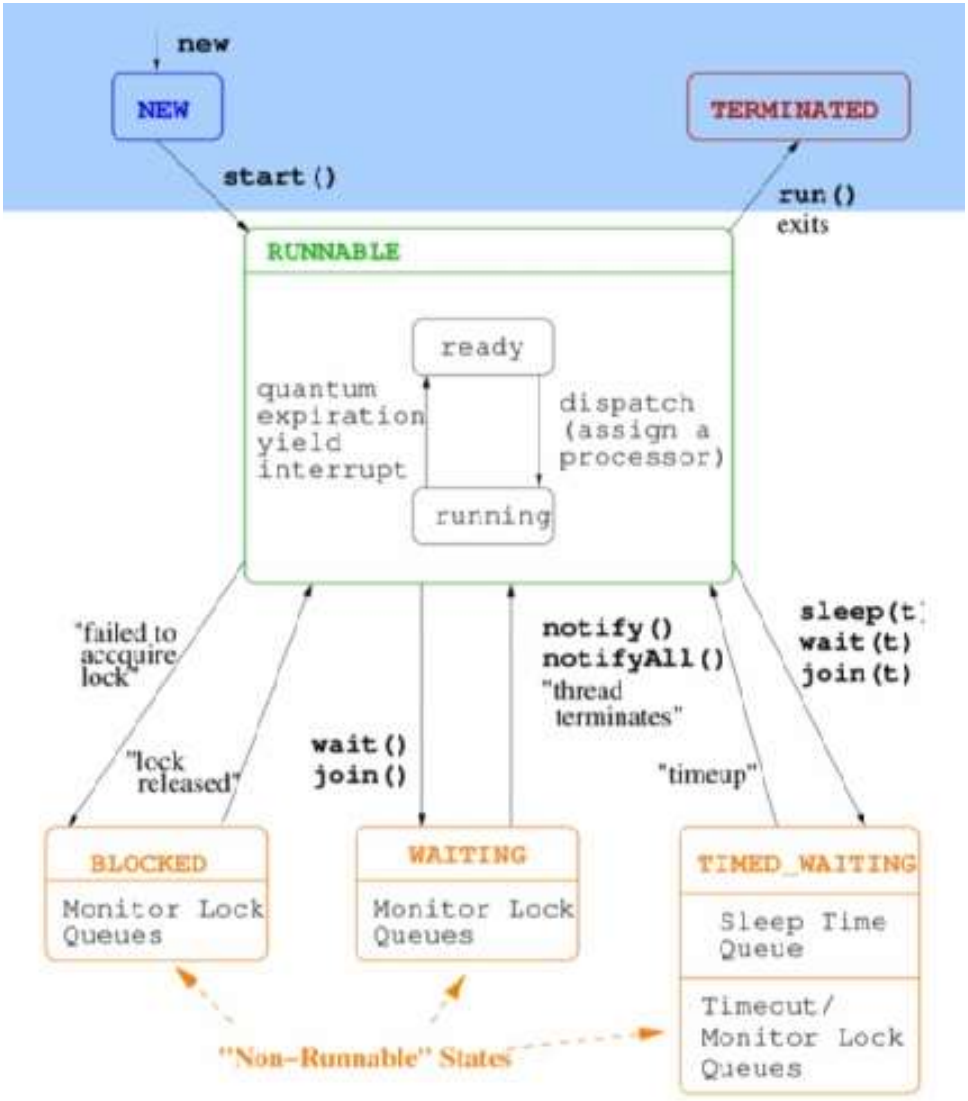
From the **RUNNABLE** state, a thread can move to any of the **BLOCKED**, **WAITING**, **TIMED_WAITING**, or **TERMINATED** state.

If Thread is in **RUNNABLE** state and it fails to acquire the lock it goes to **BLOCKED** state, once lock is released it comes to **RUNNABLE** state and it's decided by the ThreadScheduler. **BLOCKED** state is also known as Non-Runnable state.

If thread is in **RUNNABLE** state and once the wait() or join() invoked then it goes to **WAITING** state. Once notify() or notifyAll() is invoked then it will come back to **RUNNABLE**. **WAITING** state is also known as Non-Runnable state.

If thread is in **RUNNABLE** state and once sleep() or wait() is invoked with specific time duration as parameter then it goes to **TIMED_WAITING** state. Once time up, it will moves to **RUNNABLE** state.

When thread has executed the run() normally then it goes to **TERMINATED** state.



Thread Exceptions

This is generated when a method is called on a thread whose state doesn't allow for that method call. For example, IllegalThreadStateException.

State check methods

- ✓ isAlive() : **true** - if thread has been started but not terminated i.e.. either in **RUNNABLE** or one of the Non-Runnable states. **false**- if it is either in **NEW** or **TERMINATED** state.
- ✓ getState():This method used to retrieve the current state of a thread. We can use this value to monitor or debug any concurrency issues that our application might face in production.

Let's see Thread and Object methods in detail

A. sleep()

- ➢ If a thread doesn't want to perform any operation for a particular amount of time that is just pausing is required then we should go for sleep() method.
- ➢ sleep(n) says "I'm done with my timeslice, and please don't give me another one for at least n milliseconds." The OS doesn't even try to schedule the sleeping thread until requested time has passed.
- ➢ It is static and final method.
- ➢ It can be overloaded and throws InterruptedException also.

B. join()

- ➢ If a thread wants to wait until completing other threads then we should go for join() method.
- ➢ It is final method but not static method and native method.
- ➢ It can be overloaded and throws InterruptedException.

C. yield()

- ➢ It causes to pause current executing thread to give the chance for remaining waiting threads of same priority.
- ➢ yield() says "I'm done with my timeslice, but I still have work to do." The OS is free to immediately give the thread another timeslice, or to give some other thread or process the CPU the yielding thread just gave up.
- ➢ yield() method gives a notice to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint.
- ➢ It is static method and native method but not final method.
- ➢ It doesn't throw InterruptedException and it can't be overloaded.

D. wait()

- ➢ wait() method releases the lock.
- ➢ wait() is the method of Object class.
- ➢ wait() is the non-static method.
- ➢ wait() should be notified by notify() or notifyAll() methods.

E. **notify()**

When a thread calls notify() method on a particular object, only one thread will be notified which is waiting for the lock or monitor of that object. The thread chosen to notify is random i.e randomly one thread will be selected for notification. Notified thread doesn't get the lock of the object immediately. It gets once the calling thread releases the lock of that object. Until that it will be in BLOCKED state. It will move from BLOCKED state to RUNNING state once it gets the lock.

F. **notifyAll()**

When a thread calls notifyAll() method on a particular object, all threads which are waiting for the lock of that object are notified. All notified threads will move from WAITING state to BLOCKED state. All these threads will get the lock of the object on a priority basis. The thread which gets the lock of the object moves to RUNNING state. The remaining threads will remain in BLOCKED state until they get the object lock.

G. **interrupt()**
Interrupts the thread when this method is called on a thread, an InterruptedException is thrown if it was waiting due to methods like wait(), join(), sleep()). Remember that interrupting does not mean killing the thread, but just sending an "interrupt signal" to the running code. What will happen depends on the implementation of Thread.run() — so Runnable.run() — in particular ifInterruptedException is caught.

H. **interrupted()**
Tests whether the thread has been interrupted. The interrupted status of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it).

I. **isInterrupted()**
Same behaviour as that of interrupted() but the interrupted status of the thread is not affected

J. **setPriority()**
Changes the priority of the thread

Describe the purpose and working of sleep() method.

The sleep() method in java is used to block a thread for a particular time, which means it pause the execution of a thread for a specific time. There are two methods of doing so.

Syntax:

public static void sleep(long milliseconds)throws InterruptedException

public static void sleep(long milliseconds, int nanos)throws InterruptedException

Working of sleep() method

When we call the sleep() method, it pauses the execution of the current thread for the given time and gives priority to another thread(if available). Moreover, when the waiting time completed then again previous thread changes its state from waiting to runnable and comes in running state, and the whole process works so on till the execution doesn't complete.

What is the purpose of wait() method in Java?

The wait() method is provided by the Object class in Java. This method is used for inter-thread communication in Java. The java.lang.Object.wait() is used to pause the current thread, and wait until another thread does not call the notify() or notifyAll() method. Its syntax is given below.

public final void wait()

Why must wait() method be called from the synchronized block?

We must call the wait method otherwise it will throw java.lang.IllegalMonitorStateException exception. Moreover, we need wait() method for inter-thread communication with notify() and notifyAll(). Therefore It must be present in the synchronized block for the proper and correct communication.

Difference between wait and sleep method

- ✓ sleep() method is called on threads and not objects. Wait() method is called on objects.
- ✓ When wait() method is called then monitor moves the thread from running to waiting state. Once a thread is in wait() then it can move to runnable only when it has notify() or notifyall() for that object. The scheduler changes the state after this. While in sleep() method, the state is changed to wait and will return to runnable only after sleep time is over.
- ✓ Wait() method is a part of java.lang.Object class, while sleep() is a part of java.lang.Thread class.
- ✓ Wait() is always used with a synchronized block as it requires to lock an object while sleep() can be used from outside synchronized block.
- ✓ A wait can be "woken up" by another thread calling notify on the monitor which is being waited on whereas a sleep cannot it can only be interrupted.
- ✓ While sleeping a Thread does not release the locks it holds, while waiting releases the lock on the object that wait() is called on.
- ✓ sleep(n) says "I'm done with my timeslice, and please don't give me another one for at least n milliseconds." The OS doesn't even try to schedule the sleeping thread until requested time has passed. wait() says "I'm done my timeslice. Don't give me another timeslice until someone calls notify()."

- ✓ we can normally use sleep() for time-synchronization and wait() for multi-thread-synchronization.

Why methods like wait(), notify() and notify all() are present in object class and not in Thread class?

Object class has monitors which allow the thread to lock an object, while Thread does not have any monitors. When any of above methods are called it waits for another thread to release the object and notifies the monitor by calling notify() or notify all(). When notify() method is called it does the job of notifying all threads which are waiting for the object to be released. The object class's monitor checks for the object if it is available or not. Thread class having these methods would not help as multiple threads exist on an object and not vice versa.

What is the practical use of join() method?

Suppose we want to calculate the population of country and based on population number further action need to be taken.

We can break down this problem as calculating population of each state in a country.

What we can do is, if country has "n" states, we can create "n" threads(+1 main thread), each calculating population of different states.

Now main thread can't do further action until all the state thread update population result.

So we can join all state threads on main thread, So that main thread will wait for all state thread to complete and once result from all state thread is available it can make progress for further actions.

Note: there can be many other ways to solve this problem.

When join method is invoked, does thread release its resources and goes in waiting state or it keep resources and goes in waiting state?

If you look at source code of join() method, It internally invokes wait() method and wait() method release all the resources before going to WAITING state.

```
public final synchronized void join(){
...
while (isAlive()) {
wait(0);
}
...
}
```

So, YES. join() method release resources and goes to waiting state.
Let's see sample program and understand,

```
package com.praveen.multithreading.join;

public class ThreadJoinDemo extends Thread {

	static ThreadJoinDemo thread1;

	public void run() {
		try {
			synchronized (thread1) {
				System.out.println(Thread.currentThread().getName() + "
acquired a lock on thread1");
				Thread.sleep(5000);
				System.out.println(Thread.currentThread().getName() + "
completed");
			}
		} catch (InterruptedException e) {
		}
	}

	public static void main(String[] ar) throws Exception {
		thread1 = new ThreadJoinDemo();
		thread1.setName("thread1");
		thread1.start();
		synchronized (thread1) {
			System.out.println(Thread.currentThread().getName() + " acquired
a lock on thread1");
			Thread.sleep(1000);
			thread1.join();
			System.out.println(Thread.currentThread().getName() + "
completed");
		}
	}
}
```

Output

main acquired a lock on thread1
thread1 acquired a lock on thread1
thread1 completed

main completed.

If you see the code, "main" thread took a lock on Thread "thread1" and waits for thread1 to complete its task by calling thread1.join().

Thread "thread1", require a lock on "thread1" for executing its task.

If main thread doesn't release lock by calling thread1.join() then Thread "thread1" will not able to progress and goes in deadlock state.

What is the use of join method in case of threading in java?

join() method is used for waiting the thread in execution until the thread on which join is called is not completed.

Remember, the thread which will wait is the thread in execution and it will wait until the thread on which join method called is not completed.

Lets take a scenario, we have Main thread, Thread 1, Thread 2 and Thread 3 and we want our thread to execute in particular scenario like,

Main thread to start first and ends only after all 3 threads is completed.

Thread 1 to start and complete.

Thread 2 to start only after Thread 1 is completed.

Thread 3 to start only after Thread 2 is completed.

Let's see program for it.

```
package com.praveen.multithreading.join;

public class ThreadJoinDemo1 {
	public static void main(String ar[]) {
		System.out.println("Inside Main Thread");
		Thread thread1 = new Thread(new ThreadTask());
		thread1.start();
		Thread thread2 = new Thread(new ThreadTask(thread1));
		thread2.start();
		Thread thread3 = new Thread(new ThreadTask(thread2));
		thread3.start();
		try {
			thread1.join();
			thread2.join();
			thread3.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("End of Main Thread");
	}
}
```

```
class ThreadTask implements Runnable {
	public ThreadTask() {
	}

	public ThreadTask(Thread threadToJoin) {
		try {
			threadToJoin.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	@Override
	public void run() {
		System.out.println("Start Thread :" + Thread.currentThread().getName());
		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("End Thread :" + Thread.currentThread().getName());
	}
}
```

Output:

```
Inside Main Thread
Start Thread :Thread-0
End Thread :Thread-0
Start Thread :Thread-1
End Thread :Thread-1
Start Thread :Thread-2
End Thread :Thread-2
End of Main Thread
```

How to Block a Thread?

A Thread can temporarily be suspended or blocked from entering to the Runnable and subsequently running state by using,

- ✓ sleep(); —Thread gets started after a specified time interval unless it is interrupted.

- ✓ suspend(); — This method puts a thread in the suspended state and can be resumed using resume() method.
- ✓ wait(); — Causes the current thread to wait until another thread invokes the notify().
- ✓ notify(); — Wakes up a single thread that is waiting on this object's monitor.
- ✓ resume(); — This method resumes a thread, which was suspended using suspend() method.
- ✓ stop(); — This method stops a thread completely.

For communication between threads wait(), notify(),notifyAll() are used and it will be executed only in synchronized block.

How to interrupt a Thread?

```
package com.praveen.multithreading.interrupt;

import java.util.Random;

public class ThreadInterruptDemo {

	public static void main(String[] args) throws InterruptedException {
		System.out.println("Starting...");
		Runnable runnable1 = () -> {
			Random random = new Random();
			for (int i = 0; i < 1E8; i++) {
				System.out.println(Thread.currentThread().isInterrupted());
				if (Thread.currentThread().isInterrupted()) {

System.out.println(Thread.currentThread().isInterrupted());
					System.out.println("Interrupted");
					break;
				}
				Math.sin(random.nextDouble());
			}
		};

		Thread t1 = new Thread(runnable1);
		t1.start();
		Thread.sleep(500);
		t1.interrupt();
		t1.join();
		System.out.println("Finished...");
```

```
        }
}
```

Synchronization

The synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared resource while another thread is in the process of using or updating that resource.

There two synchronization syntax in Java Language. The practical differences are in controlling scope and the monitor. With a synchronized method, the lock is obtained for the duration of the entire method. With synchronized blocks you can specify exactly when the lock is needed.

Basically, synchronized blocks are more general, and synchronized methods can be rewritten to use synchronized blocks:

```
class Program {
public synchronized void f() {
.....
}
}
```

is equivalent to

```
class Program {
public void f() {
synchronized(this){
...
}
}
}
```

For example, You have a method with some parts that need synchronized and others don't. The synchronized block lets you synchronize only the partial line codes that really need it.

```
public class Program {
private static Object locker1 = new Object();
```

```
private static Object locker1 = new Object();
public void doSomething1() {
...
synchronized(locker1) {
..... //do something protected;
}
....
}
public void doSomething2() {
...
synchronized(locker2) {
..... //do something protected;
}
....
}
}
```

The synchronized block can only be executed after a thread has acquired the lock for the object or class referenced, for example the "locker1" or "locker2" in above code, in the synchronized statement.

The above code shows that synchronized block can be holding different object monitors. Maybe it's necessary to protect doSomething1() method and doSomething2() method from multiple threads, but it's fine if one thread is in the doSomething1() method and another is in the doSomething2() method. But the synchronized method can not do it.

A synchronized method synchronizes on the object instance or the class. A thread only executes a synchronized method after it has acquired the lock for the method's object or class.

static synchronized methods synchronize on the class object. If one thread is executing a static synchronized method, all other threads trying to execute any static synchronized methods, in the same class, will block.

non-static synchronized methods synchronize on "this" (the instance object). If one thread is executing a synchronized method, all other threads trying to execute any synchronized methods, in the same class, will block.

These are very public monitors, meaning some other thread could synchronize on them for the wrong reason, leading to slowdowns or deadlocks.

What is a use of synchronized keyword?

Synchronized keyword is used when the purpose Is to run only one thread at a time in an appropriate section of code. It can be used to show four types of different blocks as below:

1)Instance methods
2)Static methods
3)Code blocks inside instance methods
4)Code blocks inside static methods

It can be declared as:

Public synchronized void example () {}

Which is preferred - Synchronized method or Synchronized block?

Synchronized block is preferred as it provides more granular control. It only locks the critical section of the code and that eliminates unnecessary object locking.

Volatile keyword

Volatile keyword indicates that a variable's value will be modified by different threads.

When a java variable is declared volatile,the value of that variable will not be cached thread-locally, all read/write will be directed to the "main memory".

Access to the variable is synchronized on itself.

For example,

```
package com.praveen.multithreading.volatiledemo;

import java.util.Scanner;

// Volatile Keyword,  the volatile modifier guarantees that any thread that
// reads a field will see the most recently written value
class WorkerThread extends Thread {
	private volatile boolean isRunning = true;

	@Override
	public void run() {
		while (isRunning) {
			System.out.println("Running");

			try {
				Thread.sleep(50);
			} catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
        }

        public void stopWorker() {
            isRunning = false;
        }
}

public class VolatileDemo {

        public static void main(String[] args) {
            WorkerThread workerThread = new WorkerThread();
            workerThread.start();
            // Wait for the enter key
            System.out.println("Enter something to stop the thread,\nVolatile variable
running will be forced to true :");
            try (Scanner scanner = new Scanner(System.in);) {
                scanner.nextLine();
            }
            workerThread.stopWorker();

        }
}
```

Output

Enter something to stop the thread,

Volatile variable running will be forced to true :

Running

Running

Running

Running

Running

Running

Running

Difference between synchronized and volatile keyword in Java.

- ✓ Volatile keyword is used on the variables and not on method while synchronized keyword is applied on methods and blocks not on variables.

- ✓ Volatile does not acquire any lock on variable or object, but synchronized statement acquires lock on method or block in which it is used.
- ✓ Volatile does not cause liveness problems such as dedlock while synchronized May cause as it acquires lock.
- ✓ Volatile usually do not cause performance issues while synchronized block may cause performance issues.

Why local variables are thread safe in Java?

- ✓ Each thread will have its own stack which it uses to store local variables.
- ✓ Two threads will have two stacks and one thread never shares its stack with other thread.
- ✓ All local variables defined in method will be allocated memory in stack
- ✓ As soon as method execution is completed by this thread, stack frame will be removed.
- ✓ That means that local variables are never shared between threads.

What is deadlock?

Deadlock is a situation where two or more threads are waiting for each other to release the resource.

Databases -> deadlock happens when two processes each within its own transaction updates two rows of information but in the opposite order. For example, process A updates row 1 then row 2 in the exact timeframe that process B updates row2 then row1.

Operating system -> a deadlock is a situation which occurs when a process or thread enters a waiting state because a resource requested is being held by another waiting resource, which in turn is waiting for another resource held by another waiting process.

For example:

Thread 1 have lock over object 1 and waiting to get lock on object 2. Thread 2 have lock over object 2 and waiting to get lock on object 1. In this scenario, both threads will wait for each other indefinitely.

Code

```
package com.praveen.multithreading.deadlock;

//Deadlock is a programming situation where two or more threads are blocked
//forever, this situation arises with at least two threads and two or more resources.
public class DeadLockDemo {
	public static final Object lock1 = new Object();
	public static final Object lock2 = new Object();

	public static void main(String[] a) {
```

```
        Thread t1 = new Thread1();
        Thread t2 = new Thread2();
        t1.start();
        t2.start();
    }

    private static class Thread1 extends Thread {

        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock 1...");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException ignored) {
                }
                System.out.println("Thread 1: Waiting for lock 2...");
                synchronized (lock2) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }
            }
        }
    }

    private static class Thread2 extends Thread {

        public void run() {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock 2...");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException ignored) {
                }
                System.out.println("Thread 2: Waiting for lock 1...");
                synchronized (lock1) {
                    System.out.println("Thread 2: Holding lock 2 & 1...");
                }
            }
        }
    }
}
```

Output

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 2: Waiting for lock 1...

Thread 1: Waiting for lock 2...

What is livelock in multithreading?

A thread often acts in response to the action of another thread.

If the other thread's action is also a response to the action of another thread, then livelock may result.

Livelocked threads are unable to make further progress. However, the threads are not blocked rather they are simply too busy responding to each other to resume work.

It is a recursive situation where two or more threads would keep repeating a particular code logic. The intended logic is typically giving opportunity to the other threads to proceed in favor of current thread.

A real-world example of livelock occurs when two people attempting to pass each other in a narrow corridor: **A** moves to his left to let **B** pass, while **B** moves to his right to let **A** pass. They are still blocking each other, **A** moves to his right, while **B** moves to his left… still not good.

For example,

```
package com.praveen.multithreading.livelock;

class CommonResource {
	private Worker owner;

	public CommonResource(Worker d) {
		owner = d;
	}

	public Worker getOwner() {
		return owner;
	}

	public synchronized void setOwner(Worker d) {
		owner = d;
	}
}

class Worker {
	private String name;
```

```
	private boolean active;

	public Worker(String name, boolean active) {
		this.name = name;
		this.active = active;
	}

	public String getName() {
		return name;
	}

	public boolean isActive() {
		return active;
	}

	public synchronized void work(CommonResource commonResource, Worker
otherWorker) {
		while (active) {
			// wait for the resource to become available.
			if (commonResource.getOwner() != this) {
				try {
					wait(10);
				} catch (InterruptedException e) {
					// ignore
				}
				continue;
			}

			// If other worker is also active let it do it's work first
			if (otherWorker.isActive()) {
				System.out.println(getName() + " : handover the resource to
the worker " + otherWorker.getName());
				commonResource.setOwner(otherWorker);
				continue;
			}

			// now use the commonResource
			System.out.println(getName() + ": working on the common
resource");
			active = false;
			commonResource.setOwner(otherWorker);
		}
	}
}
```

```
public class LiveLockDemo {
	public static void main(String[] args) {
		final Worker worker1 = new Worker("Worker 1 ", true);
		final Worker worker2 = new Worker("Worker 2", true);

		final CommonResource s = new CommonResource(worker1);

		new Thread(() -> {
			worker1.work(s, worker2);
		}).start();

		new Thread(() -> {
			worker2.work(s, worker1);
		}).start();
	}
}
```

Output

```
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
Worker 1  : handover the resource to the worker Worker 2
Worker 2 : handover the resource to the worker Worker 1
```

Difference between deadlock and livelock in Java multithreading.

A deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another however none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

What causes starvation in threads?

- ✓ Threads with high priority takes up all CPU time from threads with lower priority.
- ✓ Threads are blocked indefinitely waiting to enter a synchronized block.
- ✓ Threads waiting on an object by calling wait() and remain waiting indefinitely.

Explain race condition in multithreading.

Race conditions occurs when 2 or more threads operate on same object without proper synchronization and the steps on the operation interleaves on other thread.

An example of Race condition is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. If two threads tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread. atomic operations are not subject to race conditions because those operation cannot be interleaved.

For example,

```
package com.praveen.multithreading.racecondition;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class Counter {
   int count = 0;

   public  void increment() {
      count = count + 1;
   }

   public  int getCount() {
      return count;
   }
}

public class RaceConditionDemo {

   public static void main(String[] args) throws InterruptedException {
      ExecutorService executorService = Executors.newFixedThreadPool(10);

     Counter counter = new Counter();
```

```
        for(int i = 0; i < 1000; i++) {
            executorService.submit(() -> counter.increment());
        }

        executorService.shutdown();
        executorService.awaitTermination(60, TimeUnit.SECONDS);

        System.out.println("Final count is : " + counter.getCount());
    }
}
```

Output

Final count is : 1000

Producer Consumer problem solution wait() and notify()

```
package com.praveen.multithreading.waitnotify;

import java.util.Scanner;

public class Processor {

        public void produce() throws InterruptedException {
                synchronized (this) {
                        System.out.println("Producer thread running.....");
                        wait();
                        System.out.println("Resumed...");
                }
        }

        public void consume() throws InterruptedException {
                try (Scanner scanner = new Scanner(System.in);) {
                        Thread.sleep(2000);
                        synchronized (this) {
                                System.out.println("Waiting for return key.");
                                scanner.nextLine();
                                System.out.println("Return key pressed.");
                                notify();
                                Thread.sleep(2000);
                        }
                }
```

```
	}

}
```

```
package com.praveen.multithreading.waitnotify;

public class ProducerConsumerApp {

	public static void main(String[] args) throws InterruptedException {
		final Processor processor = new Processor();
		Runnable runnable1 = () -> {
			try {
				processor.produce();
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		};
		Thread t1 = new Thread(runnable1);

		Runnable runnable2 = () -> {
			try {
				processor.consume();
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		};
		Thread t2 = new Thread(runnable2);

		t1.start();
		t2.start();

		t1.join();
		t2.join();
	}

}
```

Output

```
Producer thread running.....
Waiting for return key.
Praveen Oruganti
Return key pressed.
Resumed...
```

Locks

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of the synchronized keyword.

From Java 5 the package java.util.concurrent.locks contains several lock implementations, so you may not have to implement your own locks. But you will still need to know how to use them, and it can still be useful to know the theory behind their implementation. For more details, see my tutorial on the java.util.concurrent.locks.Lock interface.

Java Lock Implementations

The java.util.concurrent.locks package has the following implementations of the Lock interface is ReentrantLock

Lock Methods

The Lock interface has the following primary methods:

- ✓ lock()
- ✓ lockInterruptibly()
- ✓ tryLock()
- ✓ tryLock(long timeout, TimeUnit timeUnit)
- ✓ unlock()

The **lock()** method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The **lockInterruptibly()** method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The **tryLock()** method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The **tryLock(long timeout, TimeUnit timeUnit)** works like the tryLock() method, except it waits up the given timeout before giving up trying to lock the Lock.

The **unlock()** method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).

ReentrantLock

The traditional way to achieve thread synchronization in Java is by the use of synchronized keyword. While it provides a certain basic synchronization, the synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once.

Synchronized blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to starvation of resources for some other thread for a very long period of time.

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code which manipulates the shared resource is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allow threads to enter into lock on a resource more than once. When the thread first enters into lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks are provided in Java to provide synchronization with greater flexibility.
new ReentrantLock(boolean fairenessParameter) -> if fairenessParameter is true then the longest-waiting thread will get the lock but when fairenessParameter is false then there is no access order.

For example,

```
package com.praveen.multithreading.locks;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockDemo {
	private static int counter=0;
	private static Lock lock= new ReentrantLock();

	public static void increment() {
		lock.lock();
		for(int i=0;i<10000;i++) {
			counter++;
		}
		lock.unlock();
```

```
    }

    public static void main(String[] args) throws Exception {

        Runnable runnable1= () -> {
            increment();
        };
        Thread t1= new Thread(runnable1);
        Runnable runnable2= () -> {
            increment();
        };
        Thread t2= new Thread(runnable2);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        }catch(InterruptedException ie) {
            System.out.println(ie.getMessage());
        }

        System.out.println("Counter is: "+ counter);

    }
}
```

Output
Counter is: 20000

Important Points

- ✓ One can forget to call the unlock() method in the finally block leading to bugs in the program. Ensure that the lock is released before the thread exits.
- ✓ The fairness parameter used to construct the lock object decreases the throughput of the program.

ProducerConsumer problem solution using ReentrantLock

```
package com.praveen.multithreading.locks;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;

class Worker {
	private Lock lock = new ReentrantLock();
	private Condition condition = lock.newCondition();

	public void producer() throws InterruptedException {
		lock.lock();
		System.out.println("Producer method...");
		condition.await();
		System.out.println("Producer again...");
		lock.unlock();
	}

	public void consumer() throws InterruptedException {
		lock.lock();
		Thread.sleep(2000);
		System.out.println("Consumer method...");
		condition.signal();
		lock.unlock();
	}
}

public class ProducerConsumerUsingReentrantLock {

	public static void main(String[] args) {
		final Worker worker=new  Worker();
		Runnable runnable1 = () -> {
			try {
				worker.producer();
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		};
		Thread t1 = new Thread(runnable1);

		Runnable runnable2 = () -> {
			try {
				worker.consumer();
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		};
		Thread t2 = new Thread(runnable2);
```

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

                try {
                        t1.join();
                        t2.join();
                } catch (InterruptedException e) {
                        System.out.println(e.getMessage());
                }

        }

}
```

Output

Producer method...

Consumer method...

Producer again...

Let's see another example for ReentrantLock

```
package com.praveen.multithreading.locks;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockMethodsCounter {
        private final ReentrantLock lock = new ReentrantLock();

        private int count = 0;

        public int incrementAndGet() {
                // Check if the lock is currently acquired by any thread
                System.out.println("IsLocked : " + lock.isLocked());

                // Check if the lock is acquired by the current thread itself.
                System.out.println("IsHeldByCurrentThread : " +
lock.isHeldByCurrentThread());

                // Try to acquire the lock
                boolean isAcquired = lock.tryLock();
                System.out.println("Lock Acquired : " + isAcquired + "\n");
```

```
            if (isAcquired) {
                try {
                    Thread.sleep(2000);
                    count = count + 1;
                } catch (InterruptedException e) {
                    throw new IllegalStateException(e);
                } finally {
                    lock.unlock();
                }
            }
            return count;
        }

        public static void main(String[] args) {
            ExecutorService executorService = Executors.newFixedThreadPool(2);

            ReentrantLockMethodsCounter lockMethodsCounter = new
ReentrantLockMethodsCounter();

            executorService.submit(() -> {
                System.out.println("IncrementCount (First Thread) : " +
lockMethodsCounter.incrementAndGet() + "\n");
            });

            executorService.submit(() -> {
                System.out.println("IncrementCount (Second Thread) : " +
lockMethodsCounter.incrementAndGet() + "\n");
            });

            executorService.shutdown();
        }
}
```

Output

```
IsLocked : false
IsHeldByCurrentThread : false
Lock Acquired : true

IsLocked : true
IsHeldByCurrentThread : false
Lock Acquired : false
```

IncrementCount (Second Thread) : 0

IncrementCount (First Thread) : 1

ReadWriteLock

A java.util.concurrent.locks.ReadWriteLock is an advanced thread lock mechanism. It allows multiple threads to read a certain resource, but only one to write it, at a time.

The idea is, that multiple threads can read from a shared resource without causing concurrency errors. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently.

ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

Read Lock If no threads have locked the ReadWriteLock for writing,

and no thread have requested a write lock (but not yet obtained it).

Thus, multiple threads can lock the lock for reading.

Write Lock If no threads are reading or writing.

Thus, only one thread at a time can lock the lock for writing.

ReadWriteLock Implementations

ReadWriteLock is an interface. Thus, to use a ReadWriteLock

The java.util.concurrent.locks package contains the following ReadWriteLock implementation is ReentrantReadWriteLock

For example,

```
package com.praveen.multithreading.locks;

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteLockDemo {
	ReadWriteLock lock = new ReentrantReadWriteLock();

	private int count = 0;

	public int incrementAndGetCount() {
		lock.writeLock().lock();
		try {
```

```
                count = count + 1;
                return count;
            } finally {
                lock.writeLock().unlock();
            }
        }

        public int getCount() {
            lock.readLock().lock();
            try {
                return count;
            } finally {
                lock.readLock().unlock();
            }
        }
}
```

Difference between Locks and Synchronized Blocks

A Reentrant lock has the same basic behavior as we have seen for synchronization blocks but there are some extended features.

- ✓ We can make a lock fair: prevent thread starvation

  Synchronized blocks are unfair by default.
- ✓ We can check whether the given lock is held or not with reentrant locks.
- ✓ We can get the list of threads waiting for the given lock with reentrant locks
- ✓ Synchronized blocks are nicer: we don't need the try-catch-finally block.
- ✓ A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
- ✓ You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
- ✓ The synchronized block must be fully contained within a single method. A Lock can have it's calls to lock() and unlock() in separate methods.

AtomicInteger

The AtomicInteger class provides you with a int variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet(). The AtomicInteger class is located in the java.util.concurrent.atomic package, so the full class name is java.util.concurrent.atomic.AtomicInteger

For example,

```
package com.praveen.multithreading.atomic;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
        private AtomicInteger count = new AtomicInteger(0);

            public int incrementAndGet() {
               return count.incrementAndGet();
            }

            public int getCount() {
               return count.get();
            }

            public static void main(String[] args) throws InterruptedException {
               ExecutorService executorService = Executors.newFixedThreadPool(2);

               AtomicCounter atomicCounter = new AtomicCounter();

               for(int i = 0; i < 1000; i++) {
                  executorService.submit(() -> atomicCounter.incrementAndGet());
               }

               executorService.shutdown();
               executorService.awaitTermination(60, TimeUnit.SECONDS);

               System.out.println("Final Count is : " + atomicCounter.getCount());
            }
}
```

Output

Final Count is : 1000

Mutex

Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

Mutex is the Semaphore with an access count of 1. Consider a situation of using lockers in the bank. Usually the rule is that only one person is allowed to enter the locker room.

Code:

```
package com.praveen.multithreading.mutex;

import java.util.concurrent.Semaphore;

public class MutexExample {
	// max 1 people
	static Semaphore semaphore = new Semaphore(1);
	static class MyLockerThread extends Thread {
	String name = "";
	MyLockerThread(String name) {
	this.name = name;
	}
	public void run() {
	try {
	System.out.println(name + " : acquiring lock…");
	System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
	semaphore.acquire();
	System.out.println(name + " : got the permit!");
	try {
	for (int i = 1; i <= 5; i++) {
	System.out.println(name + " : is performing operation " + i + ", available
Semaphore permits : " + semaphore.availablePermits());
	// sleep 1 second Thread.sleep(1000);
	}
	} finally {
	// calling release() after a successful acquire()
	System.out.println(name + " : releasing lock…");
	semaphore.release();
	System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
	}
	} catch (InterruptedException e) {
	e.printStackTrace();
	}
	}
	}
	public static void main(String[] args) {
	System.out.println("Total available Semaphore permits : " +
semaphore.availablePermits());
```

```
MyLockerThread t1 = new MyLockerThread("A"); t1.start();
MyLockerThread t2 = new MyLockerThread("B"); t2.start();
MyLockerThread t3 = new MyLockerThread("C"); t3.start();
MyLockerThread t4 = new MyLockerThread("D"); t4.start();
MyLockerThread t5 = new MyLockerThread("E"); t5.start();
MyLockerThread t6 = new MyLockerThread("F"); t6.start();
}}
```

Semaphore

Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.

Consider an ATM cubicle with 4 ATMs, Semaphore can make sure only 4 people can access simultaneously.

Code:

```
package com.praveen.multithreading.semaphore;

import java.util.concurrent.Semaphore;

public class SemaphoreExample {
	// max 4 people
	static Semaphore semaphore = new Semaphore(4);
	static class MyATMThread extends Thread {
	String name = "";
	MyATMThread(String name) {
	this.name = name;
	}
	public void run() {
	try {
	System.out.println(name + " : acquiring lock…");
	System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
	semaphore.acquire();
	System.out.println(name + " : got the permit!");
	try {
	for (int i = 1; i <= 5; i++) {
	System.out.println(name + " : is performing operation " + i + ", available
Semaphore permits : " + semaphore.availablePermits());
	// sleep 1 second
	Thread.sleep(1000);
	}
	} finally {
	// calling release() after a successful acquire()
```

```
        System.out.println(name + " : releasing lock…");
        semaphore.release();
        System.out.println(name + " : available Semaphore permits now: " +
semaphore.availablePermits());
        }
        } catch (InterruptedException e) {
        e.printStackTrace();
        }
        }
        }
        public static void main(String[] args) {
        System.out.println("Total available Semaphore permits : " +
semaphore.availablePermits());
        MyATMThread t1 = new MyATMThread("A"); t1.start();
        MyATMThread t2 = new MyATMThread("B"); t2.start();
        MyATMThread t3 = new MyATMThread("C"); t3.start();
        MyATMThread t4 = new MyATMThread("D"); t4.start();
        MyATMThread t5 = new MyATMThread("E"); t5.start();
        MyATMThread t6 = new MyATMThread("F"); t6.start();
        }}
```

Mutex vs Semaphore

Number of threads: the mutex implements mutual exclusion, only one thread can access the resource; the semaphore allows a set number of permits. Ownership: the mutex is tied to the thread currently holding the lock.

ThreadPool Concept

Java threadpool represents a group of worker thread that are waiting for the job(performed the task) and reuse many times.

- A group of fixed size threads are created.
- Assigned the job of worker thread by the service provider
- And complete the job(finish the work) threads contained in the ThreadPool again.

Advantages

- Better performance.
- Save time.
- There is no need to create new thread.

ExecutorService

Executor service is one API which will helps you to perform asynchronous non blocking execution of thread even it support features of thread pool to resuse thread and you can group them based on your business logic.

Java ExecutorService Implementations

The Java ExecutorService is very similar to a thread pool. In fact, the implementation of the ExecutorService interface present in the java.util.concurrent package is a thread pool implementation.
Since ExecutorService is an interface, you need to its implementations in order to make any use of it.

The ExecutorService interface has three standard implementations:

- ✓ **ThreadPoolExecutor** — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.
- ✓ **ScheduledThreadPoolExecutor** allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.
- ✓ **ForkJoinPool** is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too.

Here are a few examples of creating an ExecutorService:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

ExecutorService executorService4 = Executors.newCachedThreadPool();
```

ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

```
execute(Runnable)
submit(Runnable)
submit(Callable)
invokeAny(…)
invokeAll(…)
```

Execute Runnable

The Java ExecutorService execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously.

Here is an example of executing a Runnable with an ExecutorService:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
executorService.execute(new Runnable() {
public void run() {
System.out.println(“Asynchronous task”);
}
});
executorService.shutdown();
```

There is no way of obtaining the result of the executed Runnable, if necessary you will have to use a Callable for that (explained in the following sections).

Submit Runnable

The Java ExecutorService submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable has finished executing.

Here is a Java ExecutorServicesubmit() example:

```
Future future = executorService.submit(new Runnable() {
public void run() {
System.out.println(“Asynchronous task”);
}
});
future.get(); //returns null if the task has finished correctly. The submit() method returns a Java Future object which can be used to check when the Runnable has completed.
```

Submit Callable

The Java ExecutorService submit(Callable) method is similar to the submit(Runnable) method except it takes a Java Callable instead of a Runnable.

The Callable‘s result can be obtained via the Java Future object returned by the submit(Callable)method.

Here is an ExecutorServiceCallable example:

```
Future future = executorService.submit(new Callable(){
public Object call() throws Exception {
System.out.println(“Asynchronous Callable”);
return “Callable Result”;
}
});
System.out.println(“future.get() = ” + future.get());
```

The above code example will output this:Asynchronous Callable

future.get() = Callable Result

invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable.
Invoking this method does not return a Future, but returns the result of one of the Callable objects.

You have no guarantee about which of the Callable‘s results you get. Just one of the ones that finish.If one of the tasks complete (or throws an exception), the rest of the Callable‘s are cancelled.

Here is a code example:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
public String call() throws Exception {
return “Task 1”;
}
});
callables.add(new Callable<String>() {
public String call() throws Exception {
return “Task 2”;
}
});
callables.add(new Callable<String>() {
public String call() throws Exception {
return “Task 3”;
}
});
String result = executorService.invokeAny(callables);
System.out.println(“result = ” + result);
executorService.shutdown();
```

This code example will print out the object returned by one of the Callable‘s in the given collection. I have tried running it a few times, and the result changes. Sometimes it is “Task 1”, sometimes “Task 2” etc.

invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter.
The invokeAll() returns a list of Future objects via which you can obtain the results of the

executions of each Callable.Keep in mind that a task might finish due to an exception, so it may not have “succeeded”.There is no way on a Future to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() { public String call() throws Exception { return “Task 1”; } }); callables.add(new Callable<String>() { public String call() throws Exception { return “Task 2”; } }); callables.add(new Callable<String>() { public String call() throws Exception { return “Task 3”; } }); List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){ System.out.println(“future.get = ” + future.get()); }
executorService.shutdown();

Runnable vs. Callable

The Runnable interface is very similar to the Callable interface. Both interfaces represents a task that can be executed concurrently by a thread or an ExecutorService. Both interfaces only has a single method.
There is one small difference between the Callable and Runnable interface though. The difference between the Runnable and Callable interface is more easily visible when you see the interface declarations.

Here is first the Runnable interface declaration:
public interface Runnable { public void run(); }
And here is the Callable interface declaration:
public interface Callable{ public Object call() throws Exception; }

The main difference between the Runnablerun() method and the Callablecall() method is that the call() method can return an Object from the method call.
Another difference between call() and run()is that call() can throw an exception, whereas run() cannot (except for unchecked exceptions – subclasses of RuntimeException).

If you need to submit a task to a Java ExecutorService and you need a result from the task, then you need to make your task implement the Callable interface. Otherwise your task can just implement the Runnable interface.

Cancel Task

You can cancel a task (Runnable or Callable) submitted to a Java ExecutorService by calling the cancel() method on the Future returned when the task is submitted. Cancelling the task is only possible if the task has not yet started executing. Here is an example of cancelling a task by calling the Future.cancel() method:future.cancel();

ExecutorService Shutdown

When you are done using the Java ExecutorService you should shut it down, so the threads do not keep running.
If your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in

your application.
The active threads inside this ExecutorService prevents the JVM from shutting down.

shutdown()

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down.
All tasks submitted to the ExecutorService before shutdown() is called, are executed.
Here is an example of performing a Java ExecutorService shutdown:executorService.shutdown();

shutdownNow()

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks.
Perhaps they stop, perhaps the execute until the end. It is a best effort attempt. Here is an example of calling ExecutorServiceshutdownNow:executorService.shutdownNow();

awaitTermination()

The ExecutorServiceawaitTermination() method will block the thread calling it until either the ExecutorService has shutdown completely, or until a given time out occurs.
The awaitTermination()method is typically called after calling shutdown() or shutdownNow().

What Are Executor and Executorservice? What Are the Differences Between These Interfaces?

- ✓ Executor and ExecutorService are two related interfaces of java.util.concurrent framework. Executor is a very simple interface with a single execute method accepting Runnable instances for execution. In most cases, this is the interface that your task-executing code should depend on.
- ✓ ExecutorService extends the Executor interface with multiple methods for handling and checking the lifecycle of a concurrent task execution service (termination of tasks in case of shutdown) and methods for more complex asynchronous task handling including Futures.

For example

```
package com.praveen.multithreading.executorservice;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
```

```
public class ExecutorServiceDemo {
	public static void main(String[] args) {
		System.out.println("Inside : " + Thread.currentThread().getName());

		System.out.println("Creating Executor Service with a thread pool of Size
2");
		ExecutorService executorService = Executors.newFixedThreadPool(2);

		Runnable task1 = () -> {
			System.out.println("Executing Task1 inside : " +
Thread.currentThread().getName());
			try {
				TimeUnit.SECONDS.sleep(2);
			} catch (InterruptedException ex) {
				throw new IllegalStateException(ex);
			}
		};

		Runnable task2 = () -> {
			System.out.println("Executing Task2 inside : " +
Thread.currentThread().getName());
			try {
				TimeUnit.SECONDS.sleep(4);
			} catch (InterruptedException ex) {
				throw new IllegalStateException(ex);
			}
		};

		Runnable task3 = () -> {
			System.out.println("Executing Task3 inside : " +
Thread.currentThread().getName());
			try {
				TimeUnit.SECONDS.sleep(3);
			} catch (InterruptedException ex) {
				throw new IllegalStateException(ex);
			}
		};

		System.out.println("Submitting the tasks for execution...");
		executorService.submit(task1);
		executorService.submit(task2);
		executorService.submit(task3);

		executorService.shutdown();
```

```
        }
}
```

Output

Inside : main

Creating Executor Service with a thread pool of Size 2

Submitting the tasks for execution...

Executing Task1 inside : pool-1-thread-1

Executing Task2 inside : pool-1-thread-2

Executing Task3 inside : pool-1-thread-1

Let's see an example related to Callable and Future

```
package com.praveen.multithreading.executorservice;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureAndCallableDemo {
        public static void main(String[] args) throws InterruptedException,
ExecutionException {
                ExecutorService executorService = Executors.newFixedThreadPool(5);

                Callable<String> task1 = () -> {
                        Thread.sleep(2000);
                        return "Result of Task1";
                };

                Callable<String> task2 = () -> {
                        Thread.sleep(1000);
                        return "Result of Task2";
                };

                Callable<String> task3 = () -> {
                        Thread.sleep(5000);
                        return "Result of Task3";
                };
```

```
		List<Callable<String>> taskList = Arrays.asList(task1, task2, task3);

		List<Future<String>> futures = executorService.invokeAll(taskList);

		for (Future<String> future : futures) {
			// The result is printed only after all the futures are complete. (i.e.
after 5
			// seconds)
			System.out.println(future.get());
		}

		// Returns the result of the fastest callable. (task2 in this case)
		String result = executorService.invokeAny(Arrays.asList(task1, task2,
task3));

		System.out.println(result);

		executorService.shutdown();
	}
}
```

Output

Result of Task1

Result of Task2

Result of Task3

Result of Task2

Let's see another example specific to ScheduledExecutorService

```
package com.praveen.multithreading.executorservice;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorsDemo {
	public static void main(String[] args) {
		ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
		Runnable task = () -> {
			System.out.println("Executing Task At " + System.nanoTime());
```

```
            };

            System.out.println("Submitting task at " + System.nanoTime() + " to be
executed after 5 seconds.");
            scheduledExecutorService.schedule(task, 5, TimeUnit.SECONDS);

            scheduledExecutorService.shutdown();
            ScheduledExecutorService scheduledExecutorService1 =
Executors.newScheduledThreadPool(1);
            System.out.println("scheduling task to be executed every 2 seconds with
an initial delay of 0 seconds");
            scheduledExecutorService1.scheduleAtFixedRate(task, 0, 2,
TimeUnit.SECONDS);

        }
}
```

Output:

Submitting task at 42038408029514 to be executed after 5 seconds.

scheduling task to be executed every 2 seconds with an initial delay of 0 seconds

Executing Task At 42038412087385

Executing Task At 42040411223650

Executing Task At 42042411173125

Executing Task At 42043410239136

Executing Task At 42044411221244

Executing Task At 42046411280912

Executing Task At 42048411341060

Executing Task At 42050411416607

Executing Task At 42052411453659

Executing Task At 42054411514289

Executing Task At 42056411585024

Executing Task At 42058411686073

Executing Task At 42060411680299

Executing Task At 42062411805890

Executing Task At 42064411800596

…….. and so on

Fork Join Pool Concept

Fork Join Pool introduced in Java 7 and it is a concept to manage concurrent flow of execution using thread pool concept.

In Java 5 it was introduced with different pattern using Executor Service framework but it is different from Fork Join Pool.

ExecutorService is a thread pool where each thread performs a task from the beginning till the end.In ForkJoinPool, each task is splitted into subtask which are executed concurrently by different threads.

Abbreviation:

Fork : split task

Join :merge task

Pool :Repository available active thread.

Suppose I am doing one operation with multiple functions, if I will do it as a single it will take huge time but i need an optimum approach to process in minimal time.

I can use Fork to split my task to multiple thread. Then each thread will execute with task easily with minimal time using CPU pause time.

After all thread task completion it will call join method which means only if i merge all thread outputs i can get a proper response.

Once it is done, it will return to main thread and main thread will take responsibility to finish the job.

Describe the Purpose and Use-Cases of the Fork/Join Framework.

The fork/join framework allows parallelizing recursive algorithms. The main problem with parallelizing recursion using something like ThreadPoolExecutor is that you may quickly run out of threads because each recursive step would require its own thread, while the threads up the stack would be idle and waiting.

The fork/join framework entry point is the ForkJoinPool class which is an implementation of ExecutorService. It implements the work-stealing algorithm, where idle threads try to “steal” work from busy threads. This allows to spread the calculations between different threads and make progress while using fewer threads than it would require with a usual thread pool.

For example,

```
package com.praveen.multithreading.forkjoinpool;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
```

```
import java.util.concurrent.RecursiveTask;

public class ForkJoinPoolDemo {

	/*
	 * The ForkJoinPool makes it easy for tasks to split their work up into smaller
	 * tasks which are then submitted to the ForkJoinPool too. Tasks can keep
	 * splitting their work into smaller subtasks for as long as it makes to split
	 * up the task.
	 */
	public static void main(String[] args) {
		ForkJoinPool forkJoinPool = new ForkJoinPool(4); // parallelism level of 4. The parallelism level indicates how many threads or CPUs you want to work concurrently on tasks passed to the ForkJoinPool.

		/*
		 * You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService.
		 * You can submit two types of tasks. A task that does not return any result (an "action"),
		 * and a task which does return a result (a "task").
		 * These two types of tasks are represented by the RecursiveAction and RecursiveTask classes.
		 */

		/**
		 * RecursiveAction
		 *
		 * A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.
		 * A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.
		 */

		class MyRecursiveAction extends RecursiveAction {

		   private long workLoad = 0;

		   public MyRecursiveAction(long workLoad) {
		      this.workLoad = workLoad;
		   }

		   @Override
		   protected void compute() {
```

```
                //if work is above threshold, break tasks up into smaller tasks
                if(this.workLoad > 16) {
                    System.out.println("Splitting workLoad : " + this.workLoad);

                    List<MyRecursiveAction> subtasks =
                        new ArrayList<MyRecursiveAction>();

                    subtasks.addAll(createSubtasks());

                    for(RecursiveAction subtask : subtasks){
                        subtask.fork();
                    }

                } else {
                    System.out.println("Doing workLoad myself: " + this.workLoad);
                }
            }

            private List<MyRecursiveAction> createSubtasks() {
                List<MyRecursiveAction> subtasks =
                    new ArrayList<MyRecursiveAction>();

                MyRecursiveAction subtask1 = new
MyRecursiveAction(this.workLoad / 2);
                MyRecursiveAction subtask2 = new
MyRecursiveAction(this.workLoad / 2);

                subtasks.add(subtask1);
                subtasks.add(subtask2);

                return subtasks;
            }
        }


        /**
         * RecursiveTask
         *
         * A RecursiveTask is a task that returns a result. It may split its work up
into smaller tasks,
         * and merge the result of these smaller tasks into a collective result. The
splitting and merging may take place on several levels.
         */
```

```
class MyRecursiveTask extends RecursiveTask<Long> {

   private long workLoad = 0;

   public MyRecursiveTask(long workLoad) {
      this.workLoad = workLoad;
   }

   protected Long compute() {

      //if work is above threshold, break tasks up into smaller tasks
      if(this.workLoad > 16) {
         System.out.println("Splitting workLoad : " + this.workLoad);

         List<MyRecursiveTask> subtasks =
            new ArrayList<MyRecursiveTask>();
         subtasks.addAll(createSubtasks());

         for(MyRecursiveTask subtask : subtasks){
            subtask.fork();
         }

         long result = 0;
         for(MyRecursiveTask subtask : subtasks) {
            result += subtask.join();
         }
         return result;

      } else {
         System.out.println("Doing workLoad myself: " + this.workLoad);
         return workLoad * 3;
      }
   }

   private List<MyRecursiveTask> createSubtasks() {
      List<MyRecursiveTask> subtasks =
      new ArrayList<MyRecursiveTask>();

      MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad
/ 2);
      MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad
/ 2);


      subtasks.add(subtask1);
```

```
                subtasks.add(subtask2);

                return subtasks;
            }
        }


        // Schedule a RecursiveTask
        MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

        long mergedResult = forkJoinPool.invoke(myRecursiveTask);

        System.out.println("mergedResult = " + mergedResult);
    }
}
```

Output

```
Splitting workLoad : 128
Splitting workLoad : 64
Splitting workLoad : 32
Doing workLoad myself: 16
Doing workLoad myself: 16
Splitting workLoad : 32
Doing workLoad myself: 16
Doing workLoad myself: 16
Splitting workLoad : 64
Splitting workLoad : 32
Doing workLoad myself: 16
Doing workLoad myself: 16
Splitting workLoad : 32
Doing workLoad myself: 16
Doing workLoad myself: 16
mergedResult = 384
```

What is FutureTask Class?

FutureTask implements Future interface and provides asynchronous processing. It contains the methods to start and cancel a task and also methods that can return the state of the FutureTask as whether its completed or cancelled. We need a callable

object to create a future task and then we can use Java Thread Pool Executor to process these asynchronously.

Concurrent Collections

A. CountDownLatch

"t.join()" makes the current thread waiting for "t" thread is finished and we can prepare a chain of threads when a thread is waiting for some other. But sometimes CountDownLatch/CyclicBarrier are more convenient.

Also, to use join(), each thread should have a reference to another thread to call join(). It makes your code a bit dirty especially when you have more than 2 working threads. Sharing of one instance of CountDownLatch/CyclicBarrier looks more clear.

CountDownLatch is to control the execution of threads.

CountDownLatch is a class present in the java.util.concurrent package. It was introduced in java 1.5 release.

CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing.

This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development.

CountDownLatch works in latch principle, main thread will wait until Gate is open. One thread waits for n number of threads specified while creating CountDownLatch in Java. Any thread, usually main thread of application, which calls CountDownLatch.await() will wait until count reaches zero or its interrupted by another Thread. All other thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. as soon as count reaches zero, Thread awaiting starts running.

One of the disadvantage of CountDownLatch is that it's not reusable once count reaches to zero you cannot use CountDownLatch anymore, but don't worry Java concurrency API has another concurrent utility called CyclicBarrier for such requirements.

For example,

```
class Processor implements Runnable {

   private CountDownLatch latch;

   public Processor(CountDownLatch latch) {
      this.latch = latch;
   }
```

```
    public void run() {
        System.out.println("Started.");

        try {
            Thread.sleep(3000);
        } catch (InterruptedException ignored) {}
        latch.countDown();
    }
}

public class App {

    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(3);
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 3; i++) {
            executor.submit(new Processor(latch));
        }
        executor.shutdown();

        try {
            // Application main thread waits, till other service threads which are
            // as an example responsible for starting framework services have completed
started all services.
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Completed.");
    }

     }
```

Problem Statement:

Java program requires 3 services namely CacheService, AlertService and ValidationService to be started and ready before application can handle any request.

Code:

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.CountDownLatch;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class CountDownLatchDemo {
	public static void main(String args[]) {
		final CountDownLatch latch = new CountDownLatch(3);
		Thread cacheService = new Thread(new Service("CacheService", 1000,
latch));
		Thread alertService = new Thread(new Service("AlertService", 1000,
latch));
		Thread validationService = new Thread(new Service("ValidationService",
1000, latch));
		cacheService.start(); // separate thread will initialize CacheService
		alertService.start(); // another thread for AlertService intialization
		validationService.start(); // another thread for ValidationService
initialization

// application should not start processing any thread until all service is up
// and ready to do there job.
// Countdown latch is idle choice here, main thread will start with count 3
// and wait until count reaches zero. each thread once up and read will do
// a count down. this will ensure that main thread is not started processing
// until all services is up.
// count is 3 since we have 3 Threads (Services)
		try {
			latch.await(); // main thread is waiting on CountDownLatch to finish
			System.out.println("All services are up, Application is starting
now");
		} catch (InterruptedException ie) {
			ie.printStackTrace();
		}
	}
}

/**
 * Service class which will be executed by Thread using CountDownLatch
 * synchronizer.
 */
class Service implements Runnable {
	private final String name;
	private final int timeToStart;
	private final CountDownLatch latch;

	public Service(String name, int timeToStart, CountDownLatch latch) {
```

```
                this.name = name;
                this.timeToStart = timeToStart;
                this.latch = latch;
        }

        @Override
        public void run() {
                try {
                        Thread.sleep(timeToStart);
                } catch (InterruptedException ex) {
                        Logger.getLogger(Service.class.getName()).log(Level.SEVERE,
null, ex);
                }
                System.out.println(name + " is Up");
                latch.countDown(); // reduce count of CountDownLatch by 1
        }
}
```

Output:

CacheService is Up

AlertService is Up

ValidationService is Up

All services are up, Application is starting now

Problem Statement: print 1–10,11–20,21–30.. so on up to 100 using 10 thread ?

Code:

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {
        public static void main(String[] args) {
                CountDownLatch latch = new CountDownLatch(10);
                for (int i = 1; i <= 100; i = i + 10) {
                        Printer printer = new Printer(i, i + 9, latch, i * 10);
                        printer.start();
                }
// The main task waits for 10 threads
                try {
                        latch.await();
                        // Main thread has started
```

```
                        System.out.println(Thread.currentThread().getName() + " has
finished");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}

class Printer extends Thread {
        private int start;
        private int end;
        private CountDownLatch latch;
        private int delay;

        public Printer(int start, int end, CountDownLatch latch, int delay) {
                super();
                this.start = start;
                this.end = end;
                this.latch = latch;
                this.delay = delay;
        }

        @Override
        public void run() {
                try {
                        Thread.sleep(delay);
                        for (int i = start; i <= end; i++) {
                                System.out.println(Thread.currentThread().getName() + " : "
+ i + "\t" + "*");
                        }
                } catch (Exception ex) {
                        System.out.println(ex.getLocalizedMessage());
                }
                System.out.println("*****************");
                latch.countDown();
        }
}
```

Output:

```
Thread-0 : 1  *
Thread-0 : 2  *
Thread-0 : 3  *
```

```
Thread-0 : 4  *
Thread-0 : 5  *
Thread-0 : 6  *
Thread-0 : 7  *
Thread-0 : 8  *
Thread-0 : 9  *
Thread-0 : 10         *
*****************
Thread-1 : 11         *
Thread-1 : 12         *
Thread-1 : 13         *
Thread-1 : 14         *
Thread-1 : 15         *
Thread-1 : 16         *
Thread-1 : 17         *
Thread-1 : 18         *
Thread-1 : 19         *
Thread-1 : 20         *
*****************
Thread-2 : 21         *
Thread-2 : 22         *
Thread-2 : 23         *
Thread-2 : 24         *
Thread-2 : 25         *
Thread-2 : 26         *
Thread-2 : 27         *
Thread-2 : 28         *
Thread-2 : 29         *
Thread-2 : 30         *
*****************
Thread-3 : 31         *
Thread-3 : 32         *
Thread-3 : 33         *
```

```
Thread-3 : 34       *
Thread-3 : 35       *
Thread-3 : 36       *
Thread-3 : 37       *
Thread-3 : 38       *
Thread-3 : 39       *
Thread-3 : 40       *
*****************
Thread-4 : 41       *
Thread-4 : 42       *
Thread-4 : 43       *
Thread-4 : 44       *
Thread-4 : 45       *
Thread-4 : 46       *
Thread-4 : 47       *
Thread-4 : 48       *
Thread-4 : 49       *
Thread-4 : 50       *
*****************
Thread-5 : 51       *
Thread-5 : 52       *
Thread-5 : 53       *
Thread-5 : 54       *
Thread-5 : 55       *
Thread-5 : 56       *
Thread-5 : 57       *
Thread-5 : 58       *
Thread-5 : 59       *
Thread-5 : 60       *
*****************
Thread-6 : 61       *
Thread-6 : 62       *
Thread-6 : 63       *
```

```
Thread-6 : 64       *
Thread-6 : 65       *
Thread-6 : 66       *
Thread-6 : 67       *
Thread-6 : 68       *
Thread-6 : 69       *
Thread-6 : 70       *
**********
Thread-7 : 71       *
Thread-7 : 72       *
Thread-7 : 73       *
Thread-7 : 74       *
Thread-7 : 75       *
Thread-7 : 76       *
Thread-7 : 77       *
Thread-7 : 78       *
Thread-7 : 79       *
Thread-7 : 80       *
**********
Thread-8 : 81       *
Thread-8 : 82       *
Thread-8 : 83       *
Thread-8 : 84       *
Thread-8 : 85       *
Thread-8 : 86       *
Thread-8 : 87       *
Thread-8 : 88       *
Thread-8 : 89       *
Thread-8 : 90       *
**********
Thread-9 : 91       *
Thread-9 : 92       *
Thread-9 : 93       *
```

Thread-9 : 94 *
Thread-9 : 95 *
Thread-9 : 96 *
Thread-9 : 97 *
Thread-9 : 98 *
Thread-9 : 99 *
Thread-9 : 100 *
****

main has finished

B. CyclicBarrier

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc.
CyclicBarrier is similar to CountDownLatch which allows multiple threads to wait for each other (barrier) before proceeding.

CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed.

All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await().
In general calling await() is shout out that Thread is waiting on the barrier.

await() is a blocking call but can be timed out or Interrupted by other thread.
If you look at CyclicBarrier, it also the does the same thing but there is different you cannot reuse CountDownLatch once the count reaches zero while you can reuse CyclicBarrier by calling reset () method which resets Barrier to its initial State.

What it implies that CountDownLatch is a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc.

For example,

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
class Processor implements Runnable {

	private CyclicBarrier barrier;
	private int threadId;
	public Processor(CyclicBarrier barrier, int threadId) {
		super();
		this.barrier = barrier;
		this.threadId = threadId;
	}
	public void run() {
		System.out.println("Thread"+threadId +" Started.");
		try {
			barrier.await();
		} catch (InterruptedException | BrokenBarrierException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		try {
			Thread.sleep(3000);
		} catch (InterruptedException ignored) {
		}
	}
}

public class CyclicBarrierDemo {

	public static void main(String[] args){
		CyclicBarrier barrier = new CyclicBarrier(3);
		ExecutorService executor = Executors.newFixedThreadPool(3);
		for (int i = 0; i < 3; i++) {
			executor.submit(new Processor(barrier,i));
		}
		executor.shutdown();
	}
}
```

Output

Thread0 Started.

Thread1 Started.

Thread2 Started.

Example

Here is a simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. Each thread calls await method in short duration but they don't proceed until all 3 threads reached the barrier, once all thread reach the barrier, barrier gets broker and each thread started their execution from that point.

Code:

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CyclicBarrierExample {
	// Runnable task for each thread
	private static class Task implements Runnable {
		private CyclicBarrier barrier;

		public Task(CyclicBarrier barrier) {
			this.barrier = barrier;
		}

		@Override
		public void run() {
			try {
				System.out.println(Thread.currentThread().getName() + " is
waiting on barrier");
				barrier.await();
				System.out.println(Thread.currentThread().getName() + "
has crossed the barrier");
			} catch (InterruptedException ex) {

	Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE,
null, ex);
			} catch (BrokenBarrierException ex) {

	Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE,
null, ex);
			}
		}
	}

	public static void main(String args[]) {
```

```
                // creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
                final CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {
                        @Override
                        public void run() {
                                // This task will be executed once all thread reaches barrier
                                System.out.println("All parties are arrived at barrier, lets
play");
                        }
                });
                // starting each of thread
                Thread t1 = new Thread(new Task(cb), "Thread 1");
                Thread t2 = new Thread(new Task(cb), "Thread 2");
                Thread t3 = new Thread(new Task(cb), "Thread 3");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Output:

Thread 2 is waiting on barrier

Thread 3 is waiting on barrier

Thread 1 is waiting on barrier

All parties are arrived at barrier, lets play

Thread 1 has crossed the barrier

Thread 3 has crossed the barrier

Thread 2 has crossed the barrier

When to use CyclicBarrier in Java

Given the nature of CyclicBarrier it can be very handy to implement map reduce kind of task similar to fork-join framework of Java 7, where a big task is broken down into smaller pieces and to complete the task you need output from individual small taske.g. to count population of India you can have 4 threads which count population from North, South, East, and West and once complete they can wait for each other, When last thread completed their task, Main thread or any other thread can add result from each zone and print total population.

You can use CyclicBarrier in Java :

1. To implement multi player game which cannot begin until all player has joined.
2. Perform lengthy calculation by breaking it into smaller individual tasks, In general, to implement Map reduce technique.

Important point of CyclicBarrier in Java

1. CyclicBarrier can perform a completion task once all thread reaches to the barrier, this can be provided while creating CyclicBarrier.
2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.
3. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.
4. If another thread interrupts the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:
java.util.concurrent.BrokenBarrierException
at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

5. CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

Difference between CyclicBarrier and CountDownLatch

- ✓ The main difference between CyclicBarrier and CountDownLatch is that CyclicBarrier is reusable and CountDownLatch is not. You can reuse CyclicBarrier by calling reset() method which resets the barrier to its initial state.
- ✓ CountDownLatch is good for one time event like application/module start-up time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently (re-)calculating each time when the input data changed.
- ✓ The key difference is that CountDownLatch separates threads into waiters and arrivers while all threads using a CyclicBarrier perform both roles.
- ✓ With a latch, the waiters wait for the last arriving thread to arrive, but those arriving threads don't do any waiting themselves.
  With a barrier, all threads arrive and then wait for the last to arrive.

C. Blocking Queue

An interface that represents a queue that is thread safe for put or take items from it.

For example, one thread putting items into queue and another thread taking items from it at the same time. We can do it with producer- consumer pattern.

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

class Producer implements Runnable {

        private BlockingQueue<Integer> blockingQueue;
```

```
	public Producer(BlockingQueue<Integer> blockingQueue) {
		this.blockingQueue = blockingQueue;
	}

	@Override
	public void run() {
		int counter = 0;
		while (true) {
			try {
				blockingQueue.put(counter);
				System.out.println("putting items in queue " + counter);
				counter++;
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}

		}

	}
}

class Consumer implements Runnable {

	private BlockingQueue<Integer> blockingQueue;

	public Consumer(BlockingQueue<Integer> blockingQueue) {
		this.blockingQueue = blockingQueue;
	}

	@Override
	public void run() {
		while (true) {
			try {
				int number = blockingQueue.take();
				System.out.println("taking items from the queue " + number);
				Thread.sleep(300);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}

		}

	}
```

```
}

public class BlockingQueueDemo {

	public static void main(String[] args) {
		BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);
		Producer producer = new Producer(queue);
		Consumer consumer = new Consumer(queue);
		new Thread(producer).start();
		new Thread(consumer).start();

	}

}
```

Output

```
taking items from the queue 0
putting items in queue 0
putting items in queue 1
taking items from the queue 1
putting items in queue 2
taking items from the queue 2
putting items in queue 3
taking items from the queue 3
putting items in queue 4
taking items from the queue 4
putting items in queue 5
taking items from the queue 5
putting items in queue 6
taking items from the queue 6
putting items in queue 7
taking items from the queue 7
putting items in queue 8
taking items from the queue 8
taking items from the queue 9
putting items in queue 9
putting items in queue 10
```

taking items from the queue 10

….. and so on

There is also other types of blocking queues i.e.. DelayedQueue and PriorityQueue.

D. Exchanger

With the help of exchanger, two threads can exchange the objects.

exchange() -> exchanging objects is done via one of the two exchange() methods.

For example, Generic algorithms, training neural networks.

As you know, Thread cannot access another thread variables as each thread has its own stack but in order to exchange the objects between threads we need to use Exchanger.

Code

```
package com.praveen.multithreading.concurrentcollections;

import java.util.concurrent.Exchanger;

class FirstThread implements Runnable{
	private int counter;
	private Exchanger<Integer> exchanger;

	public FirstThread(Exchanger<Integer> exchanger) {
		this.exchanger = exchanger;
	}

	@Override
	public void run() {

		while(true) {
			counter= counter+1;
			System.out.println("FirstThread incremented the counter: "+
counter);
			try {
				counter= exchanger.exchange(counter);
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
```

```
        }
}

class SecondThread implements Runnable{
        private int counter;
        private Exchanger<Integer> exchanger;

        public SecondThread(Exchanger<Integer> exchanger) {
                this.exchanger = exchanger;
        }

        @Override
        public void run() {

                while(true) {
                        counter= counter-1;
                        System.out.println("SecondThread decrements the counter: "+
counter);
                        try {
                                counter= exchanger.exchange(counter);
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
public class ExchangerDemo {

        public static void main(String[] args) {
                Exchanger<Integer> exchanger= new Exchanger<>();
                new Thread(new FirstThread(exchanger)).start();
                new Thread(new SecondThread(exchanger)).start();
        }

}
```

Output

FirstThread incremented the counter: 1

SecondThread decrements the counter: -1

SecondThread decrements the counter: 0

FirstThread incremented the counter: 0

FirstThread incremented the counter: 1
SecondThread decrements the counter: -1
FirstThread incremented the counter: 0
SecondThread decrements the counter: 0
SecondThread decrements the counter: -1
FirstThread incremented the counter: 1
SecondThread decrements the counter: 0
FirstThread incremented the counter: 0
SecondThread decrements the counter: -1
… and so on

You can refer complete code in https://github.com/praveennaga/praveennaga-multithreading

Please check out my other ebooks in https://github.com/praveennaga/PraveenNaga-Tech-Ebooks

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1