Multithreading (ref MultiThreading_Learning)

Before moving directly into multithreading, let's understand some points.

Multitasking

Multitasking allows several activities to occur concurrently on the computer.

- Process-based multitasking
- Thread-based multitasking

Process Based Multitasking :

Allows processes (i.e programs) to run concurrently on the computer.
Eg: Running the Ms Paint while also working with the word processor.

Thread Based Multitasking

Allows parts of the same program to run concurrently on the computer.
Eg: Ms Word that is printing and formatting text at the same time.

Within one process, multiple threads run concurrently to complete the task; one process contains many threads.

- Thread vs Process
    - Multiple threads run in the same program or process, which is why threads share the same address space.
    - Address space is the same in a thread, so context switching between threads is faster as compared to context switching between process.
- Why do we need multithreading?
    - In a single-threaded environment, only one task at a time can be performed.
    - CPU cycles are wasted, for example, when waiting for user input.
    - Multitasking allows idle CPU time to be put to good use.

Threads :

- A thread is an independent sequential path of execution within a program.
- Many threads can run concurrently within a program.
- At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).
- They also share the process running the program.

- 3 Important Concepts related to Multithreading in Java
  1. Creating threads and providing the code that gets executed by a thread.
  2. Accessing common data and code through synchronization.
  3. Transitioning between thread states.

- The Main Thread
  - When the standalone application is run, the user thread is automatically created to execute the main() method of the application. This thread is called the main thread.
  - If no other thread is running, the program gets terminated when the main() method finishes its execution.
  - All other threads are called child threads, and they are all created from the main thread.
  - The main() method execution is over, but the program still waits until all user-level threads do not complete their execution, and only the main method is terminated.
  - At runtime, there are two types of thread
    - User Threads
      - When all user threads complete their task, irrespective of whether daemon threads are complete or not, the program will terminate.
    - Daemon threads
      - Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.
      - As long as a user thread is alive, the JVM does not terminate.
      - A daemon thread is at the mercy of the runtime system : it is stopped if there are no more user threads running, thus terminating the program.

- Thread Creation
  - A thread in Java is represented by an object of the Thread class. Creating threads is achieved in one of two ways :

- Implementing the java.lang.Runnable interface
- Extending the java.lang.Thread class

Main.java

```
1
2  public class Main extends Thread
3  {
4
5      public static void main(String[] args)
6      {
7          Main th = new Main();
8          th.start();
9      }
10     public void run()
11     {
12         System.out.println("Thread is running..!");
13     }
14
15 }
```

Overriding run method:
Here, I create a Main class and Thread1 class; the Thread1 class extends Thread and overrides the run method.

Main.java | Thread1.java

```
1
2  public class Main
3  {
4
5      public static void main(String[] args)
6      {
7          System.out.println("Thread Start..!");
8          Thread1 th = new Thread1();
9          th.start();
10         System.out.println("Thread End..!");
11     }
12
13 }
14
```

From the main thread, we start other threads; it is called parent-child relationship.

Main.java | Thread1.java

```
1
2  public class Thread1 extends Thread
3  {
4      @Override
5      public void run()
6      {
7          for(int i=0;i<5;i++)
8          {
9              System.out.println("Inside Thread 1: "+i);
10         }
11     }
12 }
13
```

The main class is itself one user thread, and the Thread1 class is another user thread with an execution sequence we don't know; JVM will select which thread needs to execute.

Problems | Javadoc | Declaration | Console

```
<terminated> Main [Java Application] /snap/eclipse/73/plugins/
Thread Start..!
Thread End..!
Inside Thread 1: 0
Inside Thread 1: 1
Inside Thread 1: 2
Inside Thread 1: 3
Inside Thread 1: 4
```

The output shows that the main thread is ended, but the user thread is running. JVM will only terminate the program once all user threads complete their execution.

Now you want to start the Daemon thread, you have setDaemon true; in the Daemon thread, all user threads stop running, and then the Daemon thread stops at that time only even they left of execution.

The main method thread is the user thread (main thread); it is only the main thread now, and for other threads, we created the daemon thread. See output: it does not execute the daemon thread after the user thread completes its task.

Main.java | Thread1.java

```java

public class Main
{

    public static void main(String[] args)
    {
        System.out.println("Thread Start..!");
        Thread1 th = new Thread1();
        th.setDaemon(true);
        th.start();
        System.out.println("Thread End..!");
    }

}
```

Main.java | Thread1.java

```java

public class Thread1 extends Thread
{
    @Override
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Inside Thread 1: "+i);
        }
    }
}
```

Problems | Javadoc | Declaration | Cons

```
<terminated> Main [Java Application] /snap/eclipse
Thread Start..!
Thread End..!
```

If you want to give the name to the thread, do the following,

- You need to pass name to the constructor

- Main.java

```java

public class Main
{

    public static void main(String[] args)
    {
        System.out.println("Thread Start..!");
        Thread1 th = new Thread1("Thread1");  // Thread1 is name
        th.start();
        System.out.println("Thread End..!");
    }

}
```

- Thread1.java

```java

public class Thread1 extends Thread
{
    public Thread1(String ThreadName)
    {
        super(ThreadName);
    }
    @Override
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Inside: "+Thread.currentThread()+" "+i);
        }
    }
}

```

- Output

```
Problems  Javadoc  Declaration  Console
<terminated> Main [Java Application] /snap/eclipse/73/plu
Thread Start..!
Thread End..!
Inside: Thread[Thread1,5,main] 0
Inside: Thread[Thread1,5,main] 1
Inside: Thread[Thread1,5,main] 2
Inside: Thread[Thread1,5,main] 3
Inside: Thread[Thread1,5,main] 4
```

- Inside: Thread[Thread1,5,main] 0 → “Thread1” is name, “5” is a priority, “main” means it was calling from main method

Runnable Interface

- When implementing the Runnable Interface, we must override the run method by providing its implementation details. Its Interface only declares a run() method, not its definition.

Main.java × Thread1.java Thread2.java

```
1
2  public class Main
3  {
4
5      public static void main(String[] args)
6      {
7          System.out.println("Thread Start..!");
8          Thread2 th2 = new Thread2();
9          Thread th = new Thread(th2);  //Create thread class object
10         th.start();
11         System.out.println("Thread End..!");
12     }
13
14 }
15
```

Main.java Thread2.java ×

```
1
2  public class Thread2 implements Runnable
3  {
4
5      @Override
6      public void run()
7      {
8          for(int i=0;i<5;i++)
9          {
10             System.out.println("Thread2 : "+i);
11         }
12
13     }
14
15 }
```

Problems @ Javadoc Declaration Console ×

```
<terminated> Main [Java Application] /snap/eclipse/73/plu
Thread Start..!
Thread End..!
Thread2 : 0
Thread2 : 1
Thread2 : 2
Thread2 : 3
Thread2 : 4
```

- The question is, why are there two methods for creating Thread?
    - Extending the Thread class: In this approach, you create a new class that extends the Thread class and overrides the run() method. This method provides a straightforward way to define a thread by creating a subclass of the Thread class.
    - Implementing the Runnable interface: In this method, you create a class that implements the Runnable interface and defines the run() method. The advantage here is that Java allows a class to implement multiple interfaces, so implementing Runnable doesn't limit your class from extending other classes.
    - Both approaches serve the purpose of creating threads, but the choice between them often depends on the specific requirements and design considerations of the application.
    - The primary reason for having these two methods is to provide flexibility and better support for object-oriented design principles. Implementing the Runnable interface allows for more flexible structuring of classes and code reuse since Java doesn't support multiple inheritances.
- Instead of creating a new class for the implementation of the Runnable interface and then overriding the run method, we can use the lambda function.

Main.java

```
1
2  public class Main
3  {
4
5  //  public static void main(String[] args)
6  //  {
7  //      System.out.println("Thread Start..!");
8  //      Thread2 th2 = new Thread2();
9  //      Thread th = new Thread(th2);  //Create thread class object
10 //      th.start();
11 //      System.out.println("Thread End..!");
12 //  }
13 //------------------Upper code is for normal Thread call, below code is for Lambda function-----
14     public static void main(String args[])
15     {
16         System.out.println("Lambda Function Thread is Start");
17         Thread th = new Thread(() -> {              //Compiler create runnable interface and provide implemention of run
18                                                    //method of this for loop
19
20             for(int i=0;i<5;i++)
21             {
22                 System.out.println("Thread2 : "+i);
23             }
24         });
25         th.start();
26     }
27 }
28
```

-

```
Problems  @ Javadoc  Declaration  Console ×
<terminated> Main [Java Application] /snap/eclipse/73/plugins/org.ec
Lambda Function Thread is Start
Thread2 : 0
Thread2 : 1
Thread2 : 2
Thread2 : 3
Thread2 : 4
```

-

Synchronization (Ref DemoSynchronization)

- Threads share the same memory space, i.e., they can share resources (objects)
- There might be a condition that a sharable resource should be accessed by only one thread. To achieve this, “Synchronization” between threads is necessary.
    - E.g., Multiple threads are used to view the booking status of the movie theater, but at the time of booking, only one thread should book the ticket.

- For Implementation purposes, I have taken a Stack example, where I have created a Stack class with all Stack operations, ref the following code.

*Main.java | Stack.java

```
1
2  public class Stack
3  {
4      private int[] array;
5      private int stackTop;
6      public Stack(int capacity)
7      {
8          array = new int[capacity];
9          stackTop = -1;
10     }
11     public boolean isEmpty()
12     {
13         return stackTop<0;
14     }
15     public boolean isFull()
16     {
17         return stackTop>=array.length-1;
18     }
19     public boolean push(int element)
20     {
21         if(isFull())
22         {
23             return false;
24         }
25         ++stackTop;
26         try
27         {
28             Thread.sleep(1000);
29         }
30         catch (Exception e)
31         {
32
33         }
34         array[stackTop] = element;
35         return true;
36
37     }
```

```
    public int pop()
    {
        if(isEmpty())
        {
            return Integer.MIN_VALUE;
        }
        int ele = array[stackTop];
        array[stackTop] = Integer.MIN_VALUE;
        try
        {
            Thread.sleep(1000);
        }
        catch(Exception e)
        {

        }
        stackTop--;
        return ele;
    }
}
```

Main.java | Stack.java

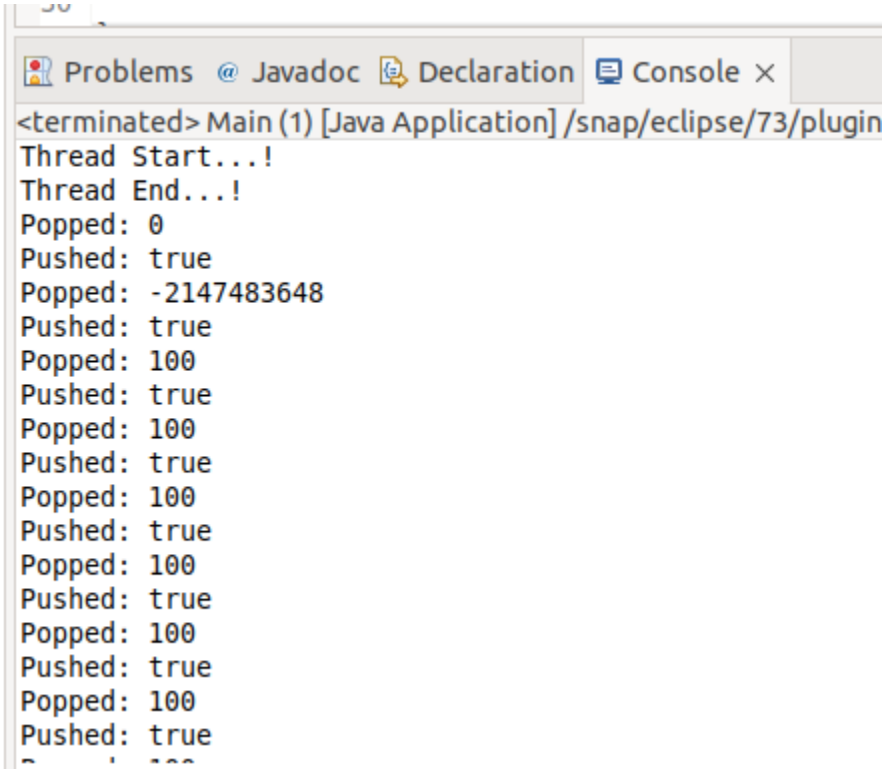
```
1
2  public class Main {
3
4      public static void main(String[] args)
5      {
6          Stack stack = new Stack(5);
7          // following is Lambda function for creating Thread uisng Runnable Interface, you can see lambda function
8          //is nothing but implementation of run method. Normally we create new class and implement Runnable interface
9          // but here instead of we directly created new Thread with Runnable interface with give run method implementation
10
11         System.out.println("Thread Start...!");
12         new Thread(() -> {
13             int counter = 0;
14             while(++counter<10)
15             {
16                 System.out.println("Pushed: "+stack.push(100));
17             }
18         }).start();
19
20         new Thread(() ->{
21             int counter = 0;
22             while(++counter<10)
23             {
24                 System.out.println("Popped: "+stack.pop());
25             }
26         }).start();
27
28         System.out.println("Thread End...!");
29     }
30
31 }
32
```

Now, here you can see that we have created two threads: one is pushing an element into the stack, and the other one is popping an element from the stack. Both threads are running concurrently, and we do not know which Thread the JVM will start.
Please note that both threads are using a single stack, and there might be a possibility of the stack overflowing or underflowing or the code running successfully.

In one case, we got this error:

```
<terminated> Main (1) [Java Application] /snap/eclipse/73/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.8.v202
Thread Start...!
Thread End...!
Popped: 0
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Popped: -2147483648
Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 5
        at Stack.push(Stack.java:34)
        at Main.lambda$0(Main.java:16)
        at java.base/java.lang.Thread.run(Thread.java:833)
```

In one case, we got this output:



Even though we handle errors properly, we still encounter an 'Out of Index' error.
The reason is that we are using two threads that share the 'Stack' as a resource. In certain cases, the 'pop' method attempts to push an element. Since both threads are running in parallel, one thread might decrement the index value to -1 while the other is pushing an element. Consequently, when attempting to push an element, it encounters a -1 index, resulting in an 'array index out of bounds' error.

Here, we do not allow both threads to run in parallel. We can say that “Stack.java” class is not thread-safe

In the Stack.java file, both the Push and Pop methods are responsible for modifying the 'stackTop' variable. We need to make changes to these methods so that only one thread is allowed to enter, even if the currently running thread is in sleep mode.

We can utilize the concept of a 'Lock'. In this concept, a thread acquires the 'Lock' from the JVM/Scheduler, then it locks the method and performs all the necessary operations. Once the task is completed, the thread hands over the lock back to the JVM/Scheduler, not to another thread.

We can utilize a 'Synchronized' lock, which essentially blocks access. We can encapsulate the affected code within this block.

```
    public boolean push(int element)
    {
        synchronized {
            if(isFull())
            {
                return false;
            }
            ++stackTop;
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {

            }
            array[stackTop] = element;
            return true;
        }

    }
    public int pop()
    {
        synchronized {
            if(isEmpty())
            {
                return Integer.MIN_VALUE;
            }
            int ele = array[stackTop];
            array[stackTop] = Integer.MIN_VALUE;
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {

            }
            stackTop--;
            return ele;
        }
    }
}
```

However, we encountered an error because we were unsure which entity will access that synchronized block. To address this, we need to pass a 'wrapper class,' such as an object, into this block for identification. You can create any object like String, Integer, etc. it will accept it.

Note: Primitive datatype won't allow, it will give an error

Main.java Stack.java ×

```
1
2  public class Stack
3  {
4      private int[] array;
5      private int stackTop;
6      Object lock;
7      public Stack(int capacity)
8      {
9          array = new int[capacity];
10         stackTop = -1;
11         lock = new Object();
12     }
13     public boolean isEmpty()
14     {
15         return stackTop<0;
16     }
17     public boolean isFull()
18     {
19         return stackTop>=array.length-1;
20     }
21     public boolean push(int element)
22     {
23         synchronized (lock) {
24             if(isFull())
25             {
26                 return false;
27             }
28             ++stackTop;
29             try
30             {
31                 Thread.sleep(1000);
32             }
33             catch (Exception e)
34             {
35
36             }
37             array[stackTop] = element;
38             return true;
39         }
40
41     }
```

```
    public int pop()
    {
        synchronized(lock) {
            if(isEmpty())
            {
                return Integer.MIN_VALUE;
            }
            int ele = array[stackTop];
            array[stackTop] = Integer.MIN_VALUE;
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {

            }
            stackTop--;
            return ele;
        }
    }
}
```

Object lock; → is the only lock used by both methods.
Let's consider we have threads named t1, t2, t3, and t4. The 'push' method is currently locked by t2, and if t1 attempts to access the 'push' method while t2 has acquired the lock, t1 will have to wait. Furthermore, neither t1, t3, nor t4 can access the 'pop' method because the same lock is shared between both methods. Only when t2 exits, will other threads be able to access either of the methods.

At any given time, only a single thread can access both methods.

If we create two different locks for both methods, and suppose the 'pop' method is taken by t2, then the 'push' method can be taken by either t1, t3, or t4 because both methods use different locks.
Object lock1;
Object lock2;

Main.java | Stack.java

```
1
2  public class Stack
3  {
4      private int[] array;
5      private int stackTop;
6      Object lock1;
7      Object lock2;
8      public Stack(int capacity)
9      {
10         array = new int[capacity];
11         stackTop = -1;
12         lock1 = new Object();
13         lock2 = new Object();
14     }
15     public boolean isEmpty()
16     {
17         return stackTop<0;
18     }
19     public boolean isFull()
20     {
21         return stackTop>=array.length-1;
22     }
23     public boolean push(int element)
24     {
25         synchronized (lock1) {
26             if(isFull())
27             {
28                 return false;
29             }
30             ++stackTop;
31             try
32             {
33                 Thread.sleep(1000);
34             }
35             catch (Exception e)
36             {
37
38             }
39             array[stackTop] = element;
40             return true;
41         }
42
43     }
```

```
    public int pop()
    {
        synchronized(lock2) {
            if(isEmpty())
            {
                return Integer.MIN_VALUE;
            }
            int ele = array[stackTop];
            array[stackTop] = Integer.MIN_VALUE;
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {

            }
            stackTop--;
            return ele;
        }
    }
}
```

If we use two locks, it could potentially allow two threads to access these methods simultaneously, leading to the same issue we initially encountered. For this specific use case, it's essential to use the same lock for both methods. This means that if one thread is using the 'push' method, no other thread can access the 'pop' method at that time, and vice versa.

```
21      }
22      public boolean push(int element)
23      {
24          synchronized (lock) {
25              if(isFull())
26              {
27                  return false;
28              }
29              ++stackTop;
30              try
31              {
32                  Thread.sleep(1000);
33              }
34              catch (Exception e)
35              {
36
37              }
38              array[stackTop] = element;
39              return true;
40          }
41
42      }
43      public int pop()
44      {
45          synchronized(lock) {
46              if(isEmpty())
47              {
48                  return Integer.MIN_VALUE;
49              }
50              int ele = array[stackTop];
51              array[stackTop] = Integer.MIN_VALUE;
52              try
53              {
54                  Thread.sleep(1000);
55              }
56              catch(Exception e)
57              {
58
59              }
60              stackTop--;
61              return ele;
62          }
63      }
```

We have made specific code blocks synchronized, not the entire method. It's important to note that when we intend to synchronize certain sections of code, we must use a synchronized block and pass a lock object.

When you intend to synchronize the entire function, the keyword 'synchronized' is used at the beginning of the function. In this scenario, there's no need to mention any lock object explicitly.

All synchronized method take “This” keyword means current instance as “Lock”

```
    }
    public synchronized boolean push(int element)
    {
        if(isFull())
        {
                return false;
        }
        ++stackTop;
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {

        }
        array[stackTop] = element;
        return true;


    }
    public synchronized int pop()
    {

        if(isEmpty())
        {
            return Integer.MIN_VALUE;
        }
        int ele = array[stackTop];
        array[stackTop] = Integer.MIN_VALUE;
        try
        {
            Thread.sleep(1000);
        }
        catch(Exception e)
        {

        }
        stackTop--;
        return ele;

    }
}
```

But you have a question: in this case, the method is synchronized, so which lock is it using? At compile time, the entire code inside the function is wrapped using the 'this' lock.

```
    public  int pop()
    {

        synchronized (this)
        {
            if(isEmpty())
            {
                return Integer.MIN_VALUE;
            }
            int ele = array[stackTop];
            array[stackTop] = Integer.MIN_VALUE;
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {

            }
            stackTop--;
            return ele;
        }

    }
}
```

One more question: If your method is 'static,' you can't add the 'synchronized' keyword to the method. The solution is, that when you have a static method, you can make the entire method synchronized. In that case, use a synchronized block and pass 'class_name.class' as a parameter within the block instead of an object.

```
    }
    public static  int pop()
    {

        synchronized (Stack.class)
        {
            if(isEmpty())
            {
                return Integer.MIN_VALUE;
            }
            int ele = array[stackTop];
            array[stackTop] = Integer.MIN_VALUE;
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception e)
            {

            }
            stackTop--;
            return ele;
        }
```

//ignore error here

Synchronized Methods :

• While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.

• This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object.

• Such a method can invoke other synchronized methods of the object without being blocked.

• The non-synchronized methods of the object can always be called at any time by any thread.

Rules of Synchronization :

• A thread must acquire the object lock associated with a shared resource before it can enter the shared resource.

• The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with it.

• If a thread cannot immediately acquire the object lock, it is blocked, i.e., it must wait for the lock to become available.

• When a thread exits a shared resource, the runtime system: that the object lock is also relinquished. If another thread is waiting for this object lock, it can try to acquire the lock in order to gain access to the shared resource.

• It should be made clear that programs should not make any assumptions about the order in which threads are granted ownership of a lock.

Volatile Keyword (ref Volatile_KeyWord)

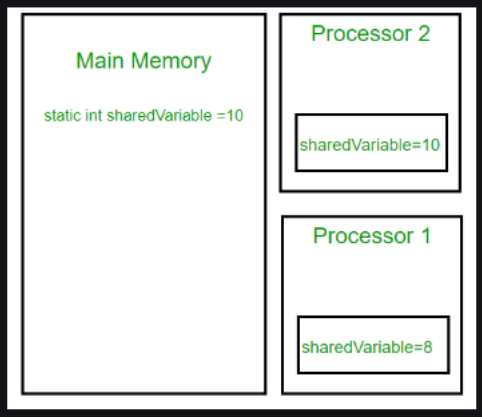
Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread-safe. Thread-safe means that a method or class instance can be used by multiple threads at the same time without any problem. Consider the below example.

```
class SharedObj
{
   // Changes made to sharedVar in one thread
   // may not immediately reflect in other thread
   static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

The below diagram shows that if two threads are run on different processors, then the value of **sharedVariable** may be different in different threads.

The below diagram shows that if two threads are run on different processors, then the value of **sharedVariable** may be different in different threads.



Note that writing of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called sequential consistency). Although most modern hardware provides good cache coherence, therefore, most probably the changes in one cache are reflected in another but it's not a good practice to rely on hardware to 'fix' a faulty application.

```
class SharedObj
{
   // volatile keyword here makes sure that
   // the changes made in one thread are
   // immediately reflect in other thread
   static volatile int sharedVar = 6;
}
```

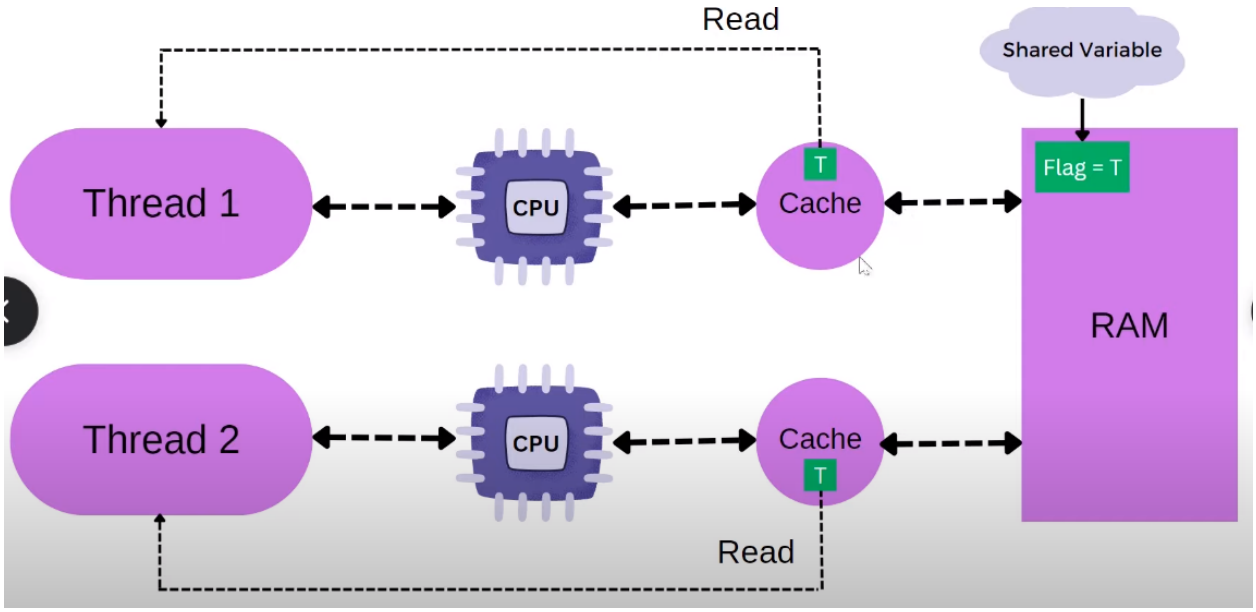
Note that volatile should not be confused with the static modifier. static variables are class members that are shared among all objects. There is only one copy of them in the main memory.

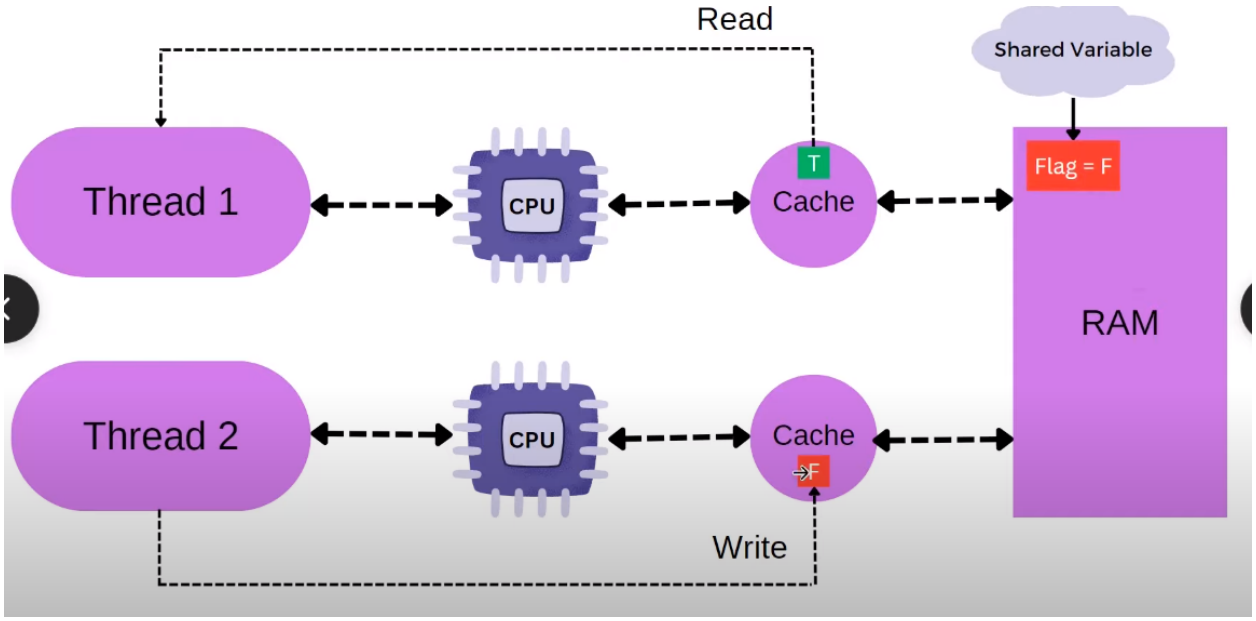
volatile vs synchronized: Before we move on let's take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

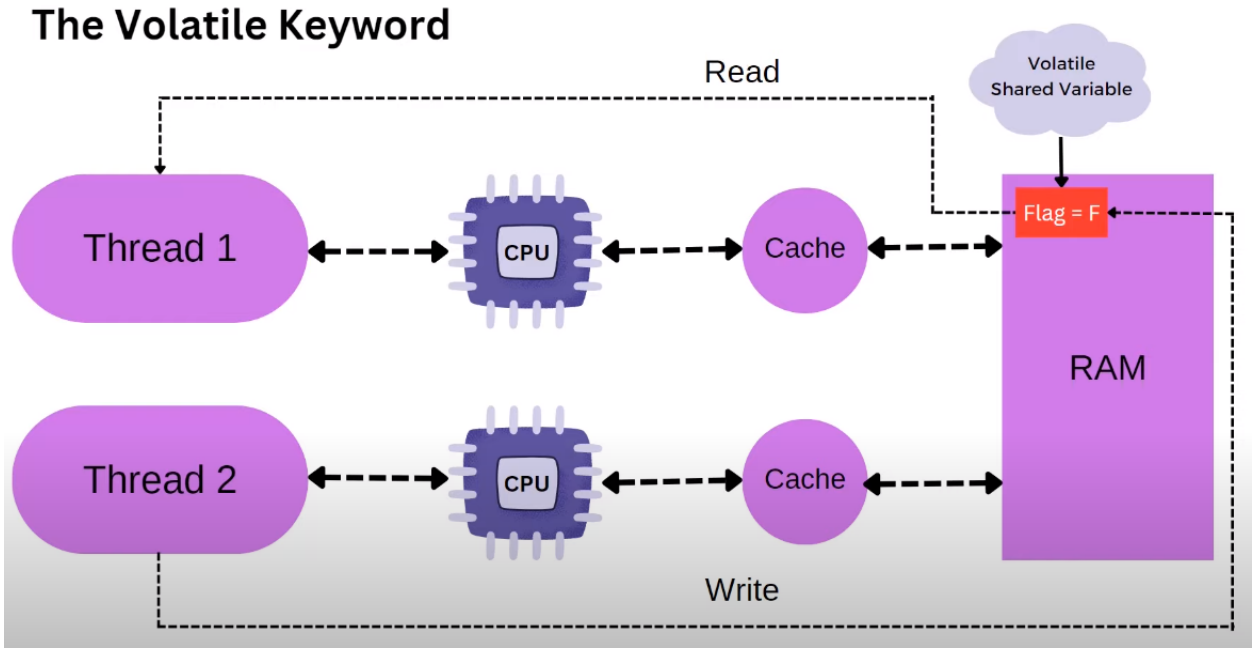
Java's synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modify the value of the shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other threads trying to enter the block at the same time will be blocked and put to sleep.

In some cases, we may only desire visibility and not atomicity. The use of synchronized in such a situation is overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of the volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, the use of volatile is limited to a very restricted set of cases as most of the times atomicity is desired. For example, a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is really a compound read-modify-write sequence of operations that must execute atomically.





After using volatile keyword, both thread access variable from MM



VolatileTest_with_Keyword.java × VolatileTest_withoutKeyword.java

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

public class VolatileTest_with_Keyword {
    private static final Logger LOGGER = Logger.getLogger(VolatileTest_with_Keyword.class.getName());
    private static volatile int MY_INT = 0;

    public static void main(String[] args) {
        new ChangeListener().start();
        new ChangeMaker().start();
    }

    static class ChangeListener extends Thread {
        @Override
        public void run() {
            int local_value = MY_INT;
            while (local_value < 5) {
                if (local_value != MY_INT) {
                    LOGGER.log(Level.INFO, "Got Change for MY_INT : {0}", MY_INT);
                    local_value = MY_INT;
                }
            }
        }
    }

    static class ChangeMaker extends Thread {
        @Override
        public void run() {
            int local_value = MY_INT;
            while (MY_INT < 5) {
                LOGGER.log(Level.INFO, "Incrementing MY_INT to {0}", local_value + 1);
                MY_INT = ++local_value;
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Problems @ Javadoc Declaration Console ×

```
<terminated> VolatileTest_with_Keyword [Java Application] /snap/eclipse/73/plugins/org.eclipse.jus
Dec 27, 2023 8:16:09 PM VolatileTest_with_Keyword$ChangeMaker run
INFO: Incrementing MY_INT to 1
Dec 27, 2023 8:16:09 PM VolatileTest_with_Keyword$ChangeListener run
INFO: Got Change for MY_INT : 1
Dec 27, 2023 8:16:10 PM VolatileTest_with_Keyword$ChangeMaker run
INFO: Incrementing MY_INT to 2
Dec 27, 2023 8:16:10 PM VolatileTest_with_Keyword$ChangeListener run
INFO: Got Change for MY_INT : 2
Dec 27, 2023 8:16:10 PM VolatileTest_with_Keyword$ChangeMaker run
INFO: Incrementing MY_INT to 3
Dec 27, 2023 8:16:10 PM VolatileTest_with_Keyword$ChangeListener run
INFO: Got Change for MY_INT : 3
Dec 27, 2023 8:16:11 PM VolatileTest_with_Keyword$ChangeMaker run
INFO: Incrementing MY_INT to 4
Dec 27, 2023 8:16:11 PM VolatileTest_with_Keyword$ChangeListener run
INFO: Got Change for MY_INT : 4
Dec 27, 2023 8:16:11 PM VolatileTest_with_Keyword$ChangeMaker run
INFO: Incrementing MY_INT to 5
Dec 27, 2023 8:16:11 PM VolatileTest_with_Keyword$ChangeListener run
INFO: Got Change for MY_INT : 5
```

VolatileTest_with_Keyword.java | VolatileTest_withoutKeyword.java ×

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

public class VolatileTest_withoutKeyword {
    private static final Logger LOGGER = Logger.getLogger(VolatileTest_withoutKeyword.class.getName());
    private static int MY_INT = 0;

    public static void main(String[] args) {
        new ChangeListener().start();
        new ChangeMaker().start();
    }

    static class ChangeListener extends Thread {
        @Override
        public void run() {
            int local_value = MY_INT;
            while (local_value < 5) {
                if (local_value != MY_INT) {
                    LOGGER.log(Level.INFO, "Got Change for MY_INT : {0}", MY_INT);
                    local_value = MY_INT;
                }
            }
        }
    }

    static class ChangeMaker extends Thread {
        @Override
        public void run() {
            int local_value = MY_INT;
            while (MY_INT < 5) {
                LOGGER.log(Level.INFO, "Incrementing MY_INT to {0}", local_value + 1);
                MY_INT = ++local_value;
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Problems | Javadoc | Declaration | Console ×

VolatileTest_withoutKeyword [Java Application] /snap/eclipse/73/plugins/org.eclipse.justj.openjdk.h

```
Dec 27, 2023 8:17:35 PM VolatileTest_withoutKeyword$ChangeMaker run
INFO: Incrementing MY_INT to 1
Dec 27, 2023 8:17:35 PM VolatileTest_withoutKeyword$ChangeMaker run
INFO: Incrementing MY_INT to 2
Dec 27, 2023 8:17:36 PM VolatileTest_withoutKeyword$ChangeMaker run
INFO: Incrementing MY_INT to 3
Dec 27, 2023 8:17:36 PM VolatileTest_withoutKeyword$ChangeMaker run
INFO: Incrementing MY_INT to 4
Dec 27, 2023 8:17:37 PM VolatileTest_withoutKeyword$ChangeMaker run
INFO: Incrementing MY_INT to 5
```

Producer Consumer Patter problem Thread (Ref Producer Consumer Problem)

In the following example:
For synchronization, I used the same lock in both methods.

BlockingQueue.java

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3
4  public class BlockingQueue
5  {
6      private Queue<Integer> q;
7      private int capacity;
8      public BlockingQueue(int cap)
9      {
10         cap = capacity;
11         q = new LinkedList<>();
12     }
13     public boolean add(int item)
14     {
15         synchronized (q)
16         {
17             if(q.size()==capacity)
18             {
19
20             }
21             q.add(item);
22         }
23
24     }
25     public int remove()
26     {
27         synchronized (q)
28         {
29             if(q.size()==0)
30             {
31
32             }
33             int num = q.poll();
34             return num;
35         }
36     }
37 }
38
```

If capacity is full or empty, what will our thread do?

Problem: Suppose one thread comes and calls the remove() method. It will lock both methods, preventing other threads from entering that code until the thread completes its execution. However, the problem arises when the queue size is zero. In this scenario, the thread has already acquired the lock, but due to the queue size being '0', it is unable to remove elements from the queue. Consequently, it is unable to complete its task and becomes stuck there, waiting for the queue to contain any elements in the future. Unfortunately, this situation does not resolve because the add() method is also blocked due to the lock. It is a deadlock case.

Solution: remove calling thread has to wait when the size==0 condition occurs, it waits until other thread adds the element.
Here we can use "wait()", "notify()" , and "notifyAll()" method of object class.

BlockingQueue.java

```java
import java.util.LinkedList;
import java.util.Queue;

public class BlockingQueue
{
    private Queue<Integer> q;
    private int capacity;
    public BlockingQueue(int cap)
    {
        cap = capacity;
        q = new LinkedList<>();
    }
    public boolean add(int item)
    {
        synchronized (q)
        {
            if(q.size()==capacity)
            {
                try {
                    q.wait();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            q.notifyAll();                      // notify all thread who are waiting()
            q.add(item);
            return true;
        }

    }
```

```
    public int remove()
    {
        synchronized (q)
        {
            if(q.size()==0)
            {
                try {                   // q.wait until some will add element,and revoke notifyAll()
                    q.wait();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            int num = q.poll();
            q.notifyAll();
            return num;
        }
    }
}
```

Here is a problem scenario involving three threads: t1, t2, and t3. Assume the queue is already full. Now, both thread t1 and thread t2 attempt to push items and wait for space in the queue. Both t1 and t2 send notifications to all threads, indicating that someone should remove an item.

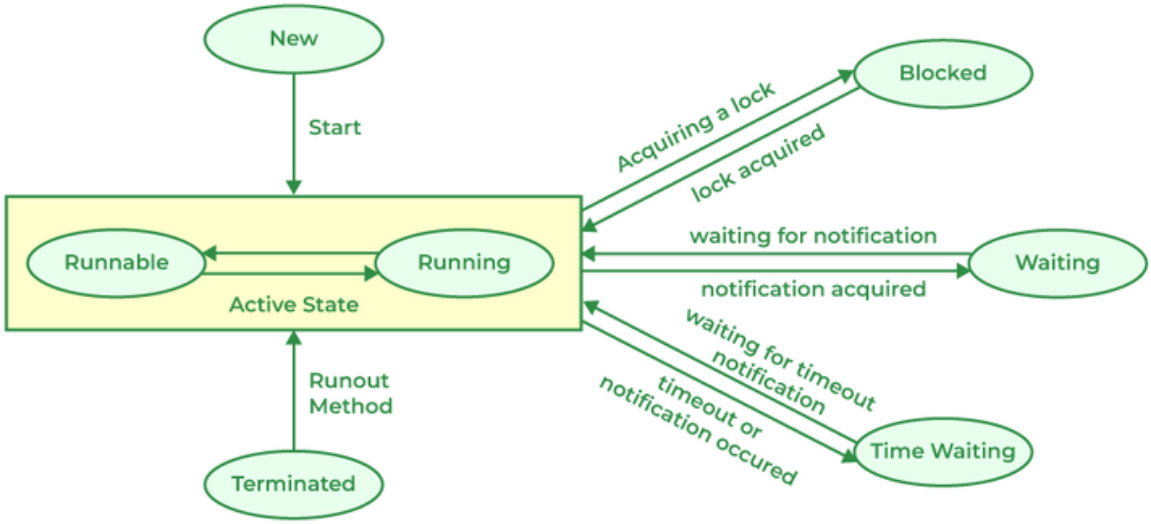
After some time, thread t3 successfully pops an element and notifies all waiting threads, including t1 and t2. Both t1 and t2 attempt to add elements, but due to the limited space in the queue, only one of them will succeed. In this situation, if the queue becomes full again after inserting an element, thread t2 should wait. Instead of directly calling the wait() method again, you can utilize a while loop to handle this scenario effectively.

BlockingQueue.java

```java
import java.util.LinkedList;

public class BlockingQueue
{
    private Queue<Integer> q;
    private int capacity;
    public BlockingQueue(int cap)
    {
        cap = capacity;
        q = new LinkedList<>();
    }
    public boolean add(int item)
    {
        synchronized (q)
        {
            while(q.size()==capacity)
            {
                try {
                    q.wait();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            q.notifyAll();                      // notify all thread who are waiting()
            q.add(item);
            return true;
        }

    }
    public int remove()
    {
        synchronized (q)
        {
            while(q.size()==0)
            {
                try {                   // q.wait until some will add element,and revoke notifyAll()
                    q.wait();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            int num = q.poll();
            q.notifyAll();
            return num;
        }
    }
}
```

Lifecycle and States of a Thread in Java (Ref: Thread_States)

- A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:
    1. New State
    2. Runnable State
    3. Blocked State
    4. Waiting State
    5. Timed Waiting State
    6. Terminated State



- ---

- New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.
- Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.
- Blocked: The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
- Waiting state: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

- Timed Waiting: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
- Terminated State: A thread terminates because of either of the following reasons:
    - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
    - Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.
-

Test.java

```java
// Java program to demonstrate thread states
class thread implements Runnable {
    public void run()
    {
        // moving thread2 to timed waiting state
        try {
            Thread.sleep(1500);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(
            "State of thread1 while it called join() method on thread2 -"
            + Test.thread1.getState());
        try {
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Test implements Runnable {
    public static Thread thread1;
    public static Test obj;

    public static void main(String[] args)
    {
        obj = new Test();
        thread1 = new Thread(obj);

        // thread1 created and is currently in the 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() method on it - "
            + thread1.getState());
    }
```

```
    public void run()
    {
        thread myThread = new thread();
        Thread thread2 = new Thread(myThread);

        // thread1 created and is currently in the 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() method on it - "
            + thread2.getState());

        // moving thread1 to timed waiting state
        try {
            // moving thread1 to timed waiting state
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(
            "State of thread2 after calling .sleep() method on it - "
            + thread2.getState());

        try {
            // waiting for thread2 to die
            thread2.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(
            "State of thread2 when it has finished it's execution - "
            + thread2.getState());
    }
}
```

```
64          // moving thread1 to timed waiting state
```

Problems @ Javadoc Declaration Console ×

```
<terminated> Test [Java Application] /snap/eclipse/73/plugins/org.eclipse.justj.ope
State of thread1 after creating it - NEW
State of thread1 after calling .start() method on it - RUNNABLE
State of thread2 after creating it - NEW
State of thread2 after calling .start() method on it - RUNNABLE
State of thread2 after calling .sleep() method on it - TIMED_WAITING
State of thread1 while it called join() method on thread2 -WAITING
State of thread2 when it has finished it's execution - TERMINATED
```

Explanation of the above Program
When a new thread is created, the thread is in the NEW state. When the start() method is called on a thread, the thread scheduler moves it to the Runnable state. Whenever the join() method is called on a thread instance, the current thread executing that statement will wait for this thread to move to the Terminated state. So, before the final statement is printed on the console, the program calls join() on thread2 making the thread1 wait while thread2 completes its execution and is moved to the Terminated state. thread1 goes to the Waiting state because it is waiting for thread2 to complete its execution as it has called join on thread2.

Java Thread Priority in Multithreading (ref Thread_Priority)

- As we already know java being completely object-oriented works within a multithreading environment in which thread scheduler assigns the processor to a thread based on the priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.
- Priorities in threads is a concept where each thread is having a priority which in layman's language one can say every object is having priority here which is represented by numbers ranging from 1 to 10.
    - The default priority is set to 5 as excepted.
    - Minimum priority is set to 1.
    - Maximum priority is set to 10.
- Here 3 constants are defined in it namely as follows:
    - public static int NORM_PRIORITY
    - public static int MIN_PRIORITY
    - public static int MAX_PRIORITY

ThreadDemo.java

```java
// Java Program to Illustrate Priorities in Multithreading
// via help of getPriority() and setPriority() method

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

// Main class
class ThreadDemo extends Thread {

    // Method 1
    // run() method for the thread that is called
    // as soon as start() is invoked for thread in main()
    public void run()
    {
        // Print statement
        System.out.println("Inside run method");
    }

    // Main driver method
    public static void main(String[] args)
    {
        // Creating random threads
        // with the help of above class
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        // Thread 1
        // Display the priority of above thread
        // using getPriority() method
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // Thread 1
        // Display the priority of above thread
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // Thread 3
        System.out.println("t3 thread priority : "
                           + t3.getPriority());
```

-

```
        // Setting priorities of above threads by
        // passing integer arguments
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw
        // IllegalArgumentException

        // 2
        System.out.println("t1 thread priority : "
                           + t1.getPriority());

        // 5
        System.out.println("t2 thread priority : "
                           + t2.getPriority());

        // 8
        System.out.println("t3 thread priority : "
                           + t3.getPriority());

        // Main thread

        // Displays the name of
        // currently executing Thread
        System.out.println(
            "Currently Executing Thread : "
            + Thread.currentThread().getName());

        System.out.println(
            "Main thread priority : "
            + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);

        System.out.println(
            "Main thread priority : "
            + Thread.currentThread().getPriority());
    }
}
```

```
Problems  Javadoc  Declaration  Console ×
<terminated> ThreadDemo [Java Application] /snap/eclipse/73/plugin
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

- If two threads have the same priority then we can't expect which thread will execute first. It depends on the thread scheduler's algorithm(Round-Robin, First Come First Serve, etc)
- If we are using thread priority for thread scheduling then we should always keep in mind that the underlying platform should provide support for scheduling based on thread priority.

Project : Ref ATM_Project

In the first scenario, I designed a simple project where two threads, named 'Nilesh' and 'Priyanshu', attempt to withdraw money from a bank. However, in this initial setup, there is no proper synchronization between the threads. Consequently, there is a possibility that one of them will attempt to withdraw money even when the bank balance is at 0.

Account.java | ClientTest.java × | AccountHolder.java

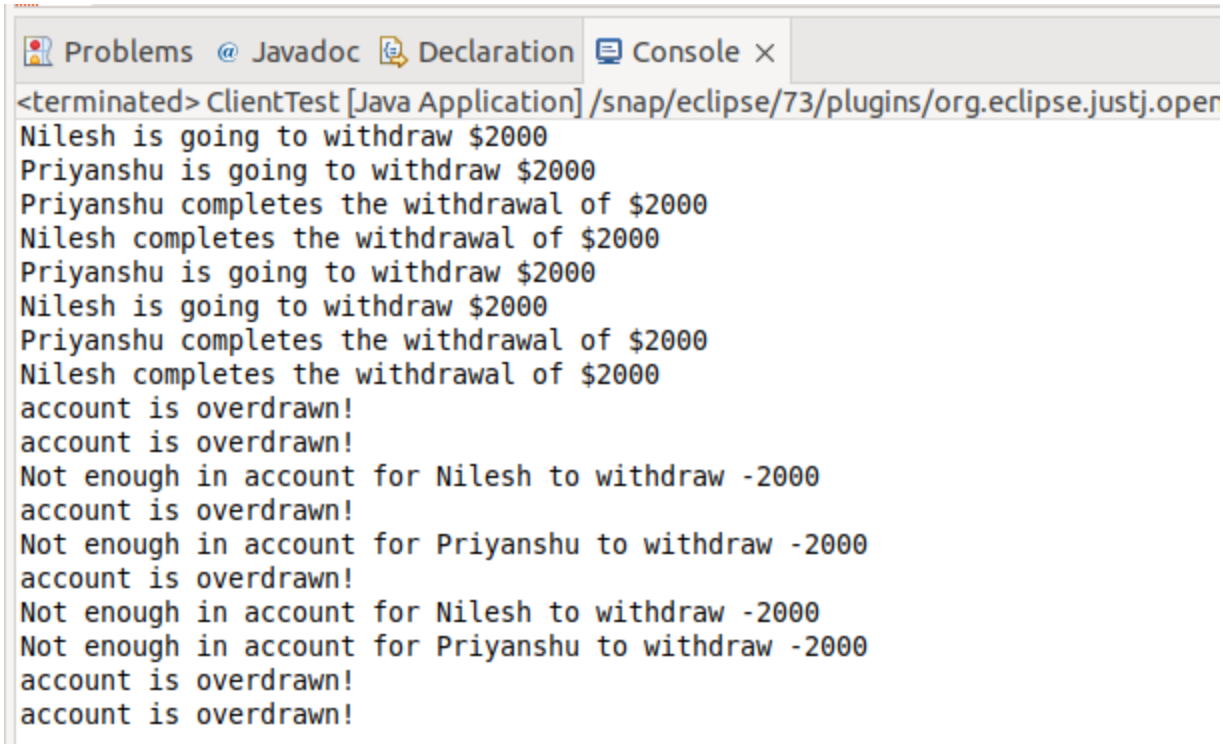
```
1
2  public class ClientTest
3  {
4
5      public static void main(String[] args)
6      {
7
8          Account account = new Account();
9          AccountHolder accountHolder = new AccountHolder(account);
10         Thread t1 = new Thread(accountHolder);
11         Thread t2 = new Thread(accountHolder);
12         t1.setName("Nilesh");
13         t2.setName("Priyanshu");
14
15         t1.start();
16         t2.start();
17     }
18 }
```

Account.java | ClientTest.java | AccountHolder.java

```java

public class Account
{
    private int balance = 6000;

    public int getBalance()
    {
        return balance;
    }
    public void withdraw(int amount)
    {
        balance = balance - amount;
    }
}

```

Account.java | ClientTest.java | AccountHolder.java

```java
public class AccountHolder implements Runnable
{
    private Account account;

    public AccountHolder(Account account)
    {
        this.account = account;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 4; i++)
        {
            makeWithdrawal(2000);
            if (account.getBalance() < 0)
            {
                System.out.println("account is overdrawn!");
            }
        }
    }

    private void makeWithdrawal(int withdrawAmount)
    {
        if (account.getBalance() >= withdrawAmount)
        {
            System.out.println(Thread.currentThread().getName() + " is going to withdraw $"+withdrawAmount);
            try
            {
                Thread.sleep(3000);
            } catch (InterruptedException ex) {
            }
            account.withdraw(withdrawAmount);
            System.out.println(Thread.currentThread().getName() + " completes the withdrawal of $"+withdrawAmount);
        }
        else
        {
            System.out.println("Not enough in account for " + Thread.currentThread().getName() + " to withdraw "
                    + account.getBalance());
        }
    }
}
```



In the second scenario, I implemented proper synchronization between the threads (I did change only in "AccountHolder class") ,
After synchronozation, if one thread is in makeWithdrawal() method, that case other thread won't enter. So it handle access of other thread if amount is 0.

```
private synchronized void makeWithdrawal(int withdrawAmount)
```

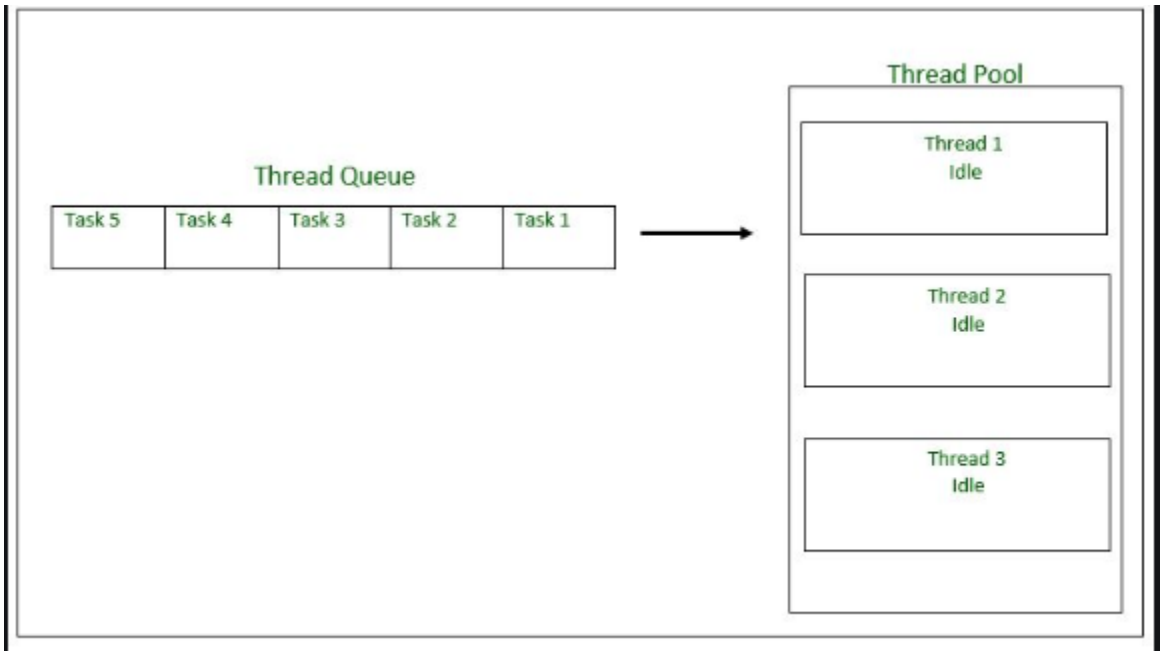
Account.java | ClientTest.java | AccountHolder.java

```java
public class AccountHolder implements Runnable
{
    private Account account;

    public AccountHolder(Account account)
    {
        this.account = account;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 4; i++)
        {
            makeWithdrawal(2000);
            if (account.getBalance() < 0)
            {
                System.out.println("account is overdrawn!");
            }
        }
    }

    private synchronized void makeWithdrawal(int withdrawAmount)
    {
        if (account.getBalance() >= withdrawAmount)
        {
            System.out.println(Thread.currentThread().getName() + " is going to withdraw $"+withdrawAmount);
            try
            {
                Thread.sleep(3000);
            } catch (InterruptedException ex) {
            }
            account.withdraw(withdrawAmount);
            System.out.println(Thread.currentThread().getName() + " completes the withdrawal of $"+withdrawAmount);
        }
        else
        {
            System.out.println("Not enough in account for " + Thread.currentThread().getName() + " to withdraw "
                    + account.getBalance());
        }
    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> ClientTest [Java Application] /snap/eclipse/73/plugins/org.eclipse.jus
Nilesh is going to withdraw $2000
Nilesh completes the withdrawal of $2000
Nilesh is going to withdraw $2000
Nilesh completes the withdrawal of $2000
Nilesh is going to withdraw $2000
Nilesh completes the withdrawal of $2000
Not enough in account for Nilesh to withdraw 0
Not enough in account for Priyanshu to withdraw 0
Not enough in account for Priyanshu to withdraw 0
Not enough in account for Priyanshu to withdraw 0
Not enough in account for Priyanshu to withdraw 0
```

Thread Pools in Java (Ref Thread_Pool)

- Problem :
    - Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.
    - Since active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.
    - Solution for this “Thread Pool”
- What is ThreadPool in Java?
    - A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.
    - Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.
    - Java provides the Executor framework which is centered around the Executor interface,
    - They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
    - To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it.
    - ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.
- 

Thread Pool Initialization with size = 3 threads. Task Queue = 5 Runnable Objects

- Methods() available:
  - **newFixedThreadPool(int)** : Creates a fixed size thread pool
  - **newCachedThreadPool()** : Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available
  - **newSingleThreadExecutor()** : Creates a single thread

Test.java × Task.java

```
1  import java.util.concurrent.ExecutorService;
2  import java.util.concurrent.Executors;
3
4  public class Test
5  {
6      // Maximum number of threads in thread pool
7      static final int MAX_T = 3;
8      public static void main(String[] args)
9      {
10         // creates five tasks
11         Runnable r1 = new Task("task 1");
12         Runnable r2 = new Task("task 2");
13         Runnable r3 = new Task("task 3");
14         Runnable r4 = new Task("task 4");
15         Runnable r5 = new Task("task 5");
16
17         // creates a thread pool with MAX_T no. of
18         // threads as the fixed pool size(Step 2)
19         ExecutorService pool = Executors.newFixedThreadPool(MAX_T);
20
21         // passes the Task objects to the pool to execute (Step 3)
22         pool.execute(r1);
23         pool.execute(r2);
24         pool.execute(r3);
25         pool.execute(r4);
26         pool.execute(r5);
27
28         // pool shutdown ( Step 4)
29         pool.shutdown();
30     }
31
32 }
33
```

Test.java | Task.java

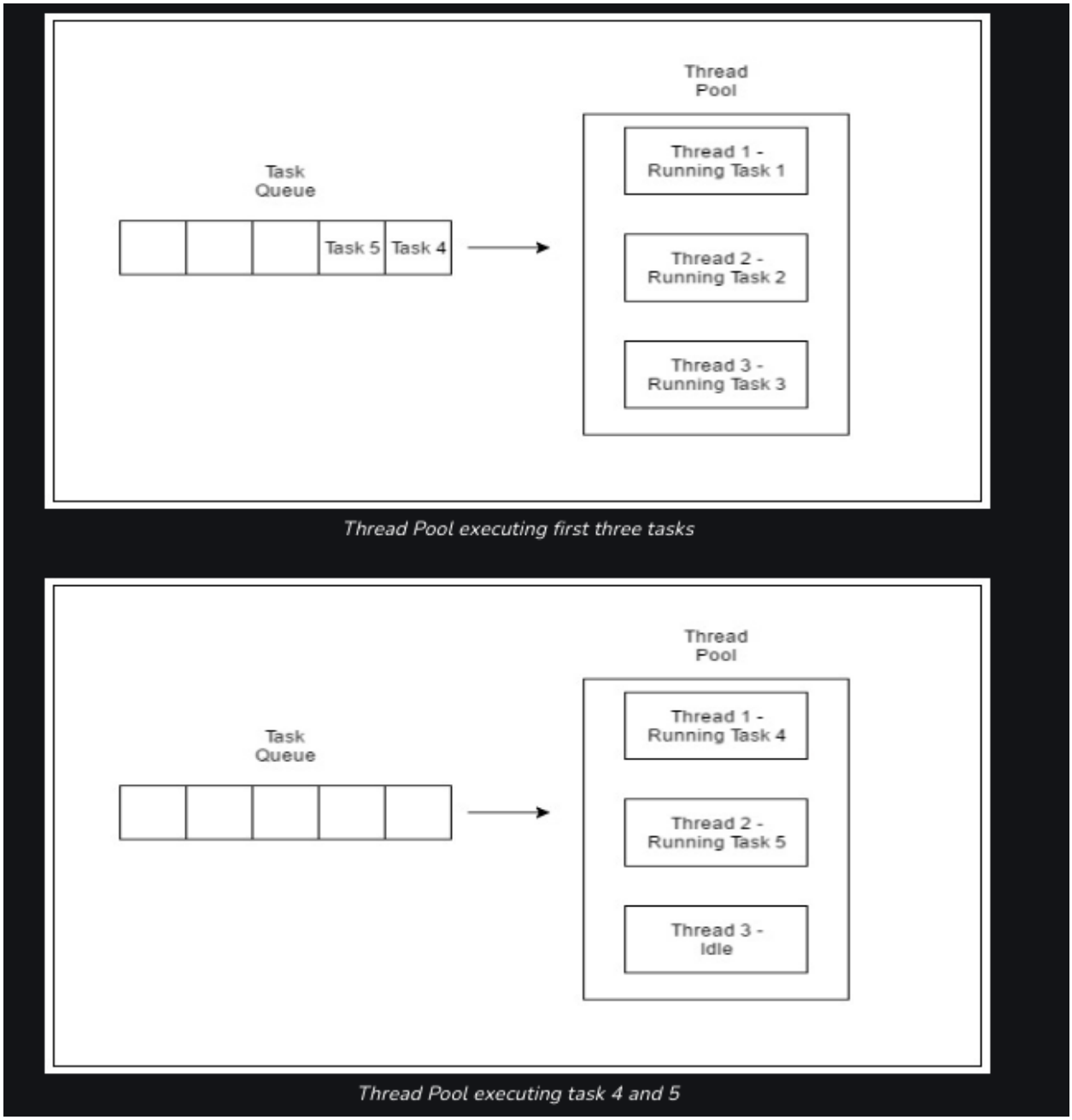
```java
import java.text.SimpleDateFormat;
import java.util.Date;
public class Task implements Runnable
{
    private String name;
    public Task(String s)
    {
        this.name = s;
    }
    // Prints task name and sleeps for 1s
    // This Whole process is repeated 5 times
    public void run()
    {
        try
        {
            for (int i = 0; i<=5; i++)
            {
                if (i==0)
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Initialization Time for"
                            + " task name - "+ name +" = " +ft.format(d));
                    //prints the initialization time for every task
                }
                else
                {
                    Date d = new Date();
                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");
                    System.out.println("Executing Time for task name - "+
                            name +" = " +ft.format(d));
                    // prints the execution time for every task
                }
                Thread.sleep(1000);
            }
            System.out.println(name+" complete");
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

```
5     private String name;
```

Problems @ Javadoc Declaration Console ×

```
<terminated> Test (1) [Java Application] /snap/eclipse/73/plugins/org.eclipse.justj.oper
Initialization Time for task name - task 1 = 03:46:59
Initialization Time for task name - task 2 = 03:46:59
Initialization Time for task name - task 3 = 03:46:59
Executing Time for task name - task 2 = 03:47:00
Executing Time for task name - task 3 = 03:47:00
Executing Time for task name - task 1 = 03:47:00
Executing Time for task name - task 2 = 03:47:01
Executing Time for task name - task 3 = 03:47:01
Executing Time for task name - task 1 = 03:47:01
Executing Time for task name - task 2 = 03:47:02
Executing Time for task name - task 3 = 03:47:02
Executing Time for task name - task 1 = 03:47:02
Executing Time for task name - task 2 = 03:47:03
Executing Time for task name - task 3 = 03:47:03
Executing Time for task name - task 1 = 03:47:03
Executing Time for task name - task 3 = 03:47:04
Executing Time for task name - task 1 = 03:47:04
Executing Time for task name - task 2 = 03:47:04
task 1 complete
task 3 complete
task 2 complete
Initialization Time for task name - task 4 = 03:47:05
Initialization Time for task name - task 5 = 03:47:05
Executing Time for task name - task 4 = 03:47:06
Executing Time for task name - task 5 = 03:47:06
Executing Time for task name - task 5 = 03:47:07
Executing Time for task name - task 4 = 03:47:07
Executing Time for task name - task 5 = 03:47:08
Executing Time for task name - task 4 = 03:47:08
Executing Time for task name - task 5 = 03:47:09
Executing Time for task name - task 4 = 03:47:09
Executing Time for task name - task 5 = 03:47:10
Executing Time for task name - task 4 = 03:47:10
task 5 complete
task 4 complete
```

- Output explanation:
    - You can see in the “Test.class” file we created “5 Task” and assigned a name for each.
    - I created a pool of size 3 using “`ExecutorService`”
    - Using Poll trying to execute all 5 tasks even though our poll size is 3,
    - In the background, only 3 tasks were assigned to the poll you can see the output t1,t2,t3 assign first.
    - All 3 tasks complete their task -> execution of for loop
    - After completing all three tasks, the remaining task i.e. t4,t5 assigned again to poll and execute loop.



- Risks in using Thread Pools:
    - Deadlock : While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
    - Thread Leakage :Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
    - Resource Thrashing :If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.
- Important Points
    - Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.
    - Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.
    - The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
    - One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.
    - You can restrict maximum number of threads that can run in JVM, reducing chances of JVM running out of memory.
    - If you need to implement your loop to create new threads for processing, using ThreadPool will help to process faster, as ThreadPool does not create new Threads after it reached it’s max limit.
    - After completion of Thread Processing, ThreadPool can use the same Thread to do another process(so saving the time and resources to create another Thread.)
-

Joining Threads in Java (Ref Join_Thread)

- java.lang.Thread class provides the join() method which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then "t.join()" will make sure that t is terminated before the next instruction is executed by the program.
- If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.
- join(): It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.

Note : join() always mentioned in try catch block, it may throw the exception.

Main.java × Demo.java

```
1
2  public class Main
3  {
4      public static void main(String args[])
5      {
6          Demo obj1 = new Demo("Thread 1");
7          Demo obj2 = new Demo("Thread 2");
8          Demo obj3 = new Demo("Thread 3");
9          Thread t1 = new Thread(obj1);
10         Thread t2 = new Thread(obj2);
11         Thread t3 = new Thread(obj3);
12
13         t1.start();  //thread 1 is Start
14
15         try
16         {
17             t1.join();  // next thread start after thread 1 is terminated
18         }
19         catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22         t2.start();
23         try
24         {
25             t2.join();  // next thread start after thread 2 is terminated
26         }
27         catch (InterruptedException e) {
28             e.printStackTrace();
29         }
30         t3.start();
31     }
32 }
```

Main.java | Demo.java

```java

public class Demo implements Runnable
{
    private String name;
    Demo(String s)
    {
        this.name = s;
    }
    @Override
    public void run()
    {
        for(int i=0;i<3;i++)
        {
            try
            {
                Thread.sleep(500);
                System.out.println("Current Thread: at "+i+" "+name+ " is start");
            }

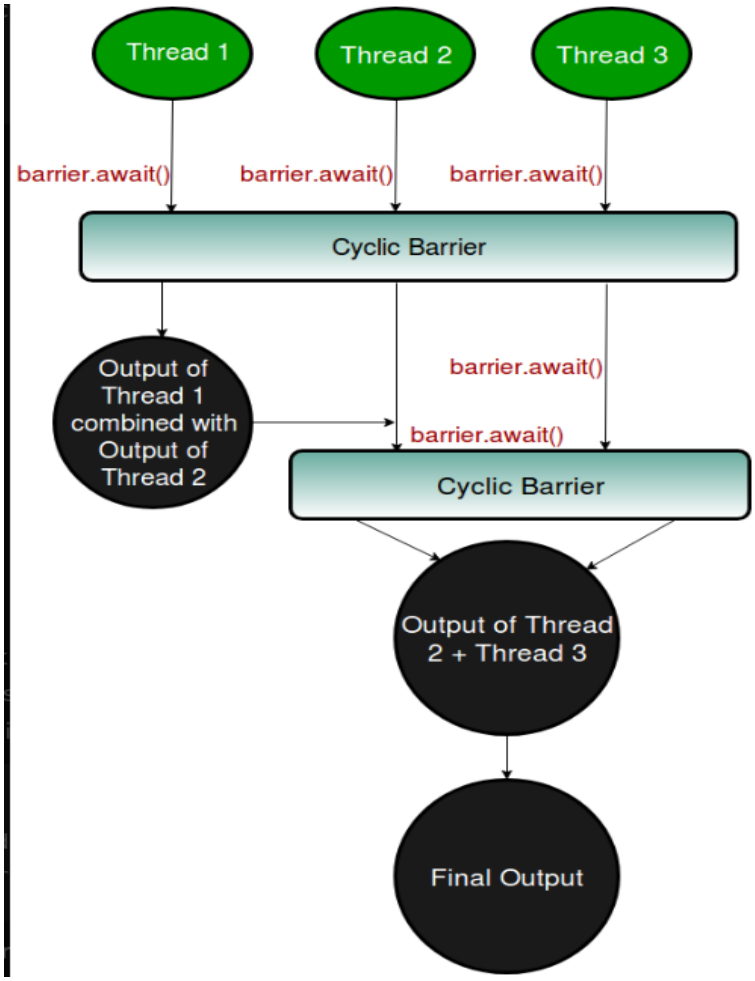
            catch(Exception ex)
            {
                System.out.println("Exception has" + " been caught" + ex);
            }
        }
        System.out.println(name+" is Dead");
    }
}

```

Problems | Javadoc | Declaration | Console

```
<terminated> Main (3) [Java Application] /snap/eclipse/73/plugins/org
Current Thread: at 0 Thread 1 is start
Current Thread: at 1 Thread 1 is start
Current Thread: at 2 Thread 1 is start
Thread 1 is Dead
Current Thread: at 0 Thread 2 is start
Current Thread: at 1 Thread 2 is start
Current Thread: at 2 Thread 2 is start
Thread 2 is Dead
Current Thread: at 0 Thread 3 is start
Current Thread: at 1 Thread 3 is start
Current Thread: at 2 Thread 3 is start
Thread 3 is Dead
```

CyclicBarrier and CountDownLatch

- **CyclicBarrier** is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.
- 

- Once the number of threads that called await() equals numberOfThreads, the barrier then gives a way for the waiting threads.
- ```
  Runnable action = ...
  //action to be performed when all threads reach the barrier;
  CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads, action);
  ```
  ```

CyclicBarrier_Test.java × Computation1.java Computation2.java

```
1  import java.util.concurrent.BrokenBarrierException;
2  import java.util.concurrent.CyclicBarrier;
3
4  public class CyclicBarrier_Test implements Runnable
5  {
6      public static CyclicBarrier newBarrier = new CyclicBarrier(3);
7
8      public static void main(String[] args)
9      {
10         //Parent Thread
11         CyclicBarrier_Test obj = new CyclicBarrier_Test();
12         Thread t1 = new Thread(obj);
13         t1.start();
14     }
15     public void run()
16     {
17          System.out.println("Number of parties required to trip the barrier = "+ newBarrier.getParties());
18          System.out.println("Sum of product and sum = " + (Computation1.product + Computation2.sum));
19
20         // objects on which the child thread has to run
21          Computation1 c1 = new Computation1();
22          Computation2 c2 = new Computation2();
23          Thread t2 = new Thread(c1);
24          Thread t3 = new Thread(c2);
25          t2.start();
26          t3.start();
27
28          try {
29             CyclicBarrier_Test.newBarrier.await();
30         } catch (InterruptedException e) {
31             e.printStackTrace();
32         } catch (BrokenBarrierException e) {
33             e.printStackTrace();
34         }
35
        // barrier breaks as the number of thread waiting for the barrier
           // at this point = 3
           System.out.println("Sum of product and sum = " + (Computation1.product +
           Computation2.sum));

           // Resetting the newBarrier
           newBarrier.reset();
           System.out.println("Barrier reset successful");
    }

}
```

CyclicBarrier_Test.java | Computation1.java | Computation2.java

```java
import java.util.concurrent.BrokenBarrierException;

public class Computation1 implements Runnable
{
    public static int product = 0;
    public void run()
    {
        product = 10 * 20;
        try {
            CyclicBarrier_Test.newBarrier.await();  // call await method
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

CyclicBarrier_Test.java | Computation1.java | Computation2.java

```java
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class Computation2 implements Runnable
{
    public static int sum = 0;
    public void run()
    {
        // check if newBarrier is broken or not
        System.out.println("Is the barrier broken? - " + CyclicBarrier_Test.newBarrier.isBroken());
        sum = 200 + 300;
        try {
            CyclicBarrier_Test.newBarrier.await(3000, TimeUnit.MILLISECONDS);
            // number of parties waiting at the barrier
            System.out.println("Number of parties waiting at the barrier "+
            "at this point = " + CyclicBarrier_Test.newBarrier.getNumberWaiting());
        }
        catch (InterruptedException  | BrokenBarrierException  | TimeoutException e)
        {
            e.printStackTrace();
        }

    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> CyclicBarrier_Test [Java Application] /snap/eclipse/73/plugins/o
Number of parties required to trip the barrier = 3
Sum of product and sum = 0
Is the barrier broken? - false
Number of parties waiting at the barrier at this point = 0
Sum of product and sum = 700
Barrier reset successful
```

- Explanation of Output:
    - In this scenario, I've established three classes: the main class along with Computation1 and Computation2. Each class implements the Runnable interface.
    - Computation1 computes the product of numbers within its run method, while Computation2 computes the sum of numbers within its run method.
    - The main class also implements the Runnable interface. In its run method, it creates two threads—one for Computation1 and another for Computation2.
    - A barrier of 3 is set, ensuring synchronization until all three threads (main and both implementations) reach a certain point simultaneously. At this point, the output requires the addition of both classes' outputs, and this combined output is printed.
    - Within the run method of the main thread, I invoke the threads for both Computation1 and Computation2.

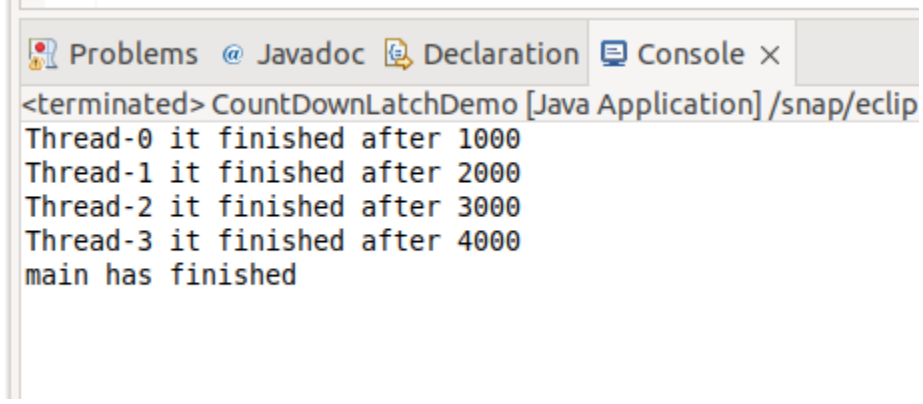
- **CountDownLatch,** is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.
- Working of CountDownLatch:
    - When we create an object of CountDownLatch, we specify the number of threads it should wait for, all such 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, the waiting task starts running.

CountDownLatchDemo.java × Worker.java

```java
package Count_Down;

import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo
{
    public static void main(String args[])
                    throws InterruptedException
    {
        // Let us create task that is going to
        // wait for four threads before it starts
        CountDownLatch latch = new CountDownLatch(4);

        // Let us create four worker
        // threads and start them.
        Worker w1 = new Worker(1000, latch, "Worker 1");
        Worker w2 = new Worker(2000, latch, "Worker 2");
        Worker w3 = new Worker(3000, latch, "Worker 3");
        Worker w4 = new Worker(4000, latch, "Worker 4");
        Thread t1 = new Thread(w1);
        Thread t2 = new Thread(w2);
        Thread t3 = new Thread(w3);
        Thread t4 = new Thread(w4);

        //Start all threads
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        // The main task waits for four threads
        latch.await();

        // Main thread has started
        System.out.println(Thread.currentThread().getName() +" has finished");
    }
}
```

CountDownLatchDemo.java | Worker.java

```java
package Count_Down;

import java.util.concurrent.CountDownLatch;

public class Worker implements Runnable
{
    private int delay;
    private CountDownLatch latch;
    private String name;
    public Worker(int delay, CountDownLatch latch, String n)
    {
        this.name = n;
        this.delay = delay;
        this.latch = latch;
    }
    public void run()
    {
        try
        {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(Thread.currentThread().getName()
                               + " it finished after "+delay);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

- 

- The main thread waits until all four threads have finished their execution.
- Facts about CountDownLatch:
    - Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.

    - The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
    - countDown() method decrements the count and await() method blocks until count == 0
- What is difference between them ?
    - In CountDownLatch, only the main thread waits for other threads to complete their execution, whereas, in CyclicBarrier, Each worker threads wait for each other to complete their execution. Let's understand this by the following Example

      CountDownLatch:

      - Consider an IT world scenario where the manager divided modules between development teams (A and B) and wants to assign them to the QA team for testing only when both teams complete their task.
      - Here manager thread works as the main thread and the development team works as the worker thread. The manager thread waits for the development team's thread to complete their task. Once developer teams complete their tasks, they will inform the manager thread, and then the manager thread assign modules to the QA team.

      CyclicBarrier:

      - Consider the same scenario where the manager divided modules between development teams (A and B). He goes on leave. He asked both teams to wait for each other to complete their respective tasks and once both teams are done, assign it to the QA team for testing.
      - Here manager thread works as the main thread and the development team works as the worker thread. Development team threads wait for other development team threads after completing their task.
    -

JVM Shutdown Hook (Ref ShutDownHook)

- Shutdown Hooks are a special construct that allows developers to plug in a piece of code to be executed when the JVM is shutting down. This comes in handy in cases where we need to do special clean up operations in case the VM is shutting down.
- Handling this using the general constructs such as making sure that we call a special procedure before the application exits (calling System.exit(0)) will not work for situations where the VM is shutting down due to an external reason (ex. kill request from O/S), or due to a resource problem (out of memory). As we will see soon, shutdown hooks solve this problem easily, by allowing us to provide an arbitrary code block, which will be called by the JVM when it is shutting down.
- From the surface, using a shutdown hook is downright straightforward. All we have to do is simply write a class that extends the java.lang.Thread class, and provide the logic that we want to perform when the VM is shutting down, inside the public void run() method. Then we register an instance of this class as a shutdown hook to the VM by calling Runtime.getRuntime().addShutdownHook(Thread) method. If you need to remove a previously registered shutdown hook, the Runtime class provides the removeShutdownHook(Thread) method as well.

ShutDownHook.java

```
1
2  public class ShutDownHook {
3
4      public static void main(String[] args)
5      {
6          Runtime.getRuntime().addShutdownHook(new Thread()
7                  {
8                      public void run()
9                      {
10                         System.out.println("Shut Down Hook is Running..!");
11                     }
12                 });
13         System.out.println("Application is Terminating");
14     }
15
16 }
17
```

-

```
Problems  @ Javadoc  Declaration  Console ×
<terminated> ShutDownHook [Java Application] /snap/eclipse/73/p
Application is Terminating
Shut Down Hook is Running..!
```

-
- When we run the above code, you will see that the shutdown hook is getting called by the JVM when it finishes the execution of the main method.

- 2nd Example
  - ShutDownHook2.java

```
1
2  public class ShutDownHook2
3  {
4      public static void main(String args[])
5      {
6          Runtime current = Runtime.getRuntime();
7          Demo obj = new Demo();
8          current.addShutdownHook(obj);
9          for(int i = 1; i <= 10; i++)
10         {
11             System.out.println("2 X " + i + " = " + 2 * i);
12         }
13         System.out.println("Main Thread Operation done time to close JVM..!");
14
15     }
16 }
17
```

  - Demo.java

```
1
2  public class Demo extends Thread
3  {
4      public void run()
5      {
6          System.out.println("Clean up code is running..!");
7          System.out.println("In Shut Down Hook");
8      }
9  }
10
```

  - Output

```
<terminated> ShutDownHook2 [Java Application] /snap/eclipse/7
2 X 2 = 4
2 X 3 = 6
2 X 4 = 8
2 X 5 = 10
2 X 6 = 12
2 X 7 = 14
2 X 8 = 16
2 X 9 = 18
2 X 10 = 20
Main Thread Operation done time to close JVM..!
Clean up code is running..!
In Shut Down Hook
```

  - The first operation of the main thread executes, and after it completes, no other threads are waiting. Consequently, the JVM initiates shutdown. Prior to this shutdown, the "Shut_Down_Hook" is invoked to clean up the code.