Day 3: ThreadPool, Executor f/w, Runnable vs Callable

Executor Framework or ThreadPool:

Consider the following example:

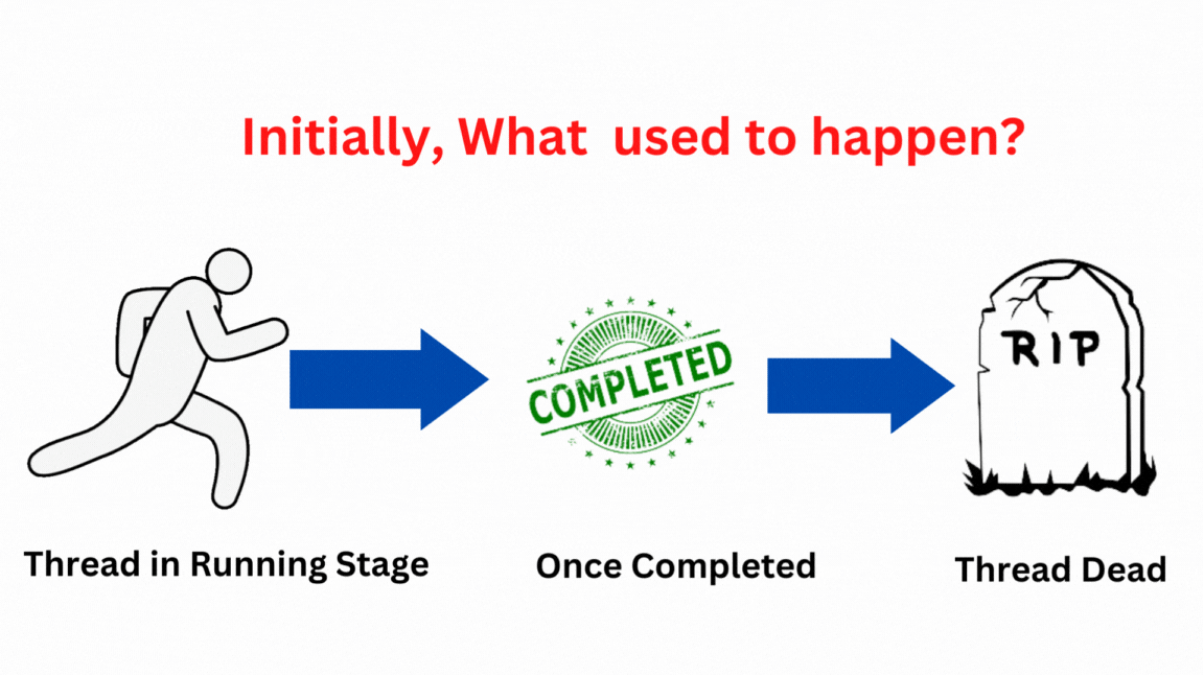
```
Runnable task = () -> System.out.println(“Hello World “ );

Thread thread= new Thread(task);

thread.start();
```

In the above example:

1. A task is an instance of Runnable
2. The task is passed to a new instance of Thread
3. The Thread is launched
4. The thread is created on demand ,by user
5. Once the task is done, thread dies.
6. Thread are expensive resource.



How can we improve the use of Threads ,as a resources ?

By creating pools of ready to use threads ,and using them.

Creating a new thread for every task may creates performance and memory problems, to overcome this we should go for Thread pool.

Thread pool is a pool of already created threads ready to do our tasks.

Thread pool framework is also known as executor framework. this concept is introduced in java5.

Thread pool related API comes in the form of **java.util.concurrent** package.

Without Thread pool, if we have 10 different independent tasks are there then we need to create 10 separate threads.

But using Thread pool concept ,we create a Thread pool of 5 threads and submit all the 10 tasks to this Thread pool.
Here a single thread can perform multiple independent tasks. so that performance will be improved.

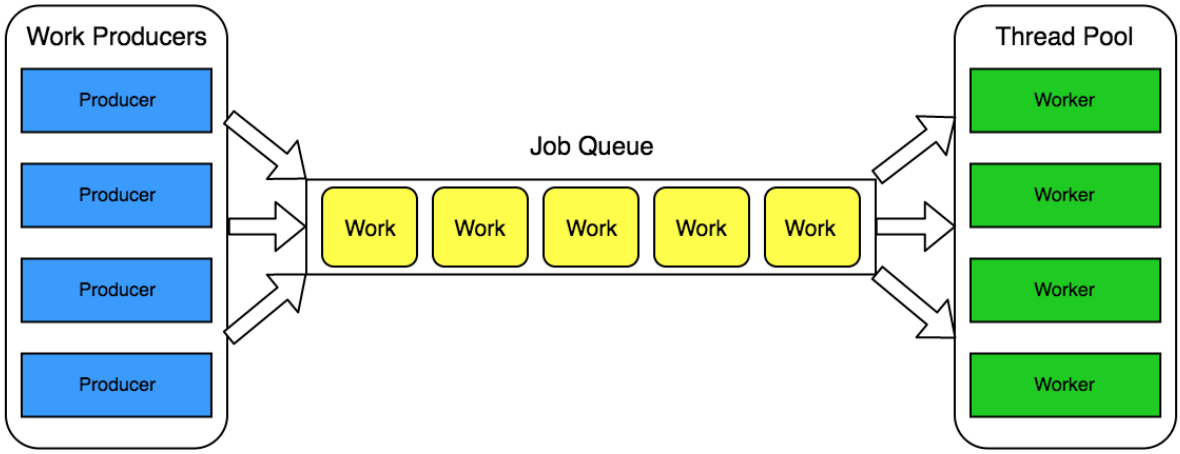
We can create Thread Pool as follows:

```
ExecutorService service=Executors.newFixedThreadPool(3);
```

Here we have created the pool of 3 threads.

After creating the pool we need to submit the tasks to this pool.

```
service.submit(task); // here task is the object of Runnable.
```



We Problem:

Execute 6 tasks by using 3 different threads.

```
class PrintJob implements Runnable{

    String name;
    PrintJob(Sting name){
      this.name=name;
    }

    public void run(){
      System.out.println(name +" job started by Thread :"+Thread.currentThread().getName());
```

```
    try{
        Thread.sleep(5000);
      }catch(InturreptedException e){
          e.printStackTrace();
      }

      System.out.println(name +"..job completed by Thread :"+Thread.currentThread().getName());
    }
}


class Main{
  public static void main(String[] args){

    PrintJob[] jobs={
           new PrintJob("Ram"),
           new PrintJob("Ravi"),
           new PrintJob("Anil"),
           new PrintJob("Shiva"),
           new PrintJob("Pawan"),
           new PrintJob("Suresh")
     };

ExecutorService service = Executors.newFixedThreadPool(3);

for(PrintJob job:jobs){

    service.submit(job);
}

  service.shutdown();//to shutdown the executorService.
}
}
```

[Further Reading]

Callable and Future:

In the case of Runnable tasks, Threads won't return anything after completing the task.

If a thread is required to return some result after execution, then we should use the Callable Interface instead of Runnable.

The callable interface belongs to **java.util.concurrent** package.

The callable interface also contains only one method:

```
public Object call()throws Exception
```

Note:- the object of Callable we can't pass to the normal Thread class constructor, unlike the Runnable object, here we need to use the ExecutorService class help.

if we submit a Callable object to **ExecutorService** object, then after completing the task, thread returns an object of the type **Future**.

The **Future** object can be used to retrieved the result from the Callable tasks.

Example:

```
import java.util.concurrent.*;
 class MyCallable implements Callable{

  int num;

  public MyCallable(int num) {
    this.num = num;
  }

  @Override
  public Object call() throws Exception {

    System.out.println(Thread.currentThread().getName()+" .. is responsible to find the sum of first "+num+" numbers");

    int sum=0;

    for(int i=0;i<=num;i++){
      sum = sum+i;
    }
    return sum;
  }

}


class Main{

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

    MyCallable[] jobs = {

        new MyCallable(10),
        new MyCallable(20),
        new MyCallable(30),
        new MyCallable(40),
        new MyCallable(50),
        new MyCallable(60),

    };

    ExecutorService service=Executors.newFixedThreadPool(3);

    for(MyCallable job:jobs){
      Future f=  service.submit(job);
      System.out.println(f.get());
    }

    service.shutdown();
  }
}
```

Difference Between Runnable and Callable:

Runnable	Callable
If a thread won't returns anything.	If a Thread returns anything
only one method **public void run()**	only one method **public Object call() throws Exception**
return type void	return type is Object
if any exception raise compulsory we need to handle within try catch.	not required to use try-catch
Belongs to java.lang package	Belongs to java.util.concurrent package
from java 1.0 version	from java 1.5 version

Suspending a thread unconditionally:

yield() method:

This yield() method is a static method defined inside the Thread class, it is used to pause the current executing thread for giving the chance to remaining waiting thread of same priority, if there is no any waiting thread or all waiting threads have low priority then same thread will get the chance once again for the execution.

suspend() method:

In order to suspend a thread unconditionally, we make use of non-static method suspend() present in a Thread class.

resume() method:

In order to resume a thread which has been suspended long back we use resume() method, it is also a non-static method defined inside the Thread class.

Note: suspend() and resume() method are deprecated methods. we should not use those methods. we should not use any deprecated methods.

The alternate methods are **wait()** and **notify()**. these methods are the non-static method defined inside the **Object** class.

Difference between yield() and wait() method:

The wait() method will suspend a thread till notify() method is called, whereas yield() method says right now it is not important please give the chance to another thread of same priority.