Threads and Thread Pool

Java Thread Scheduling

- A Java virtual machine is required to implement a preemptive, priority-based scheduler among its various threads.
  - This means that each thread in a Java program is assigned a certain priority, a positive integer that falls within a well-defined range. This priority can be changed by the developer.
- The Java virtual machine never changes the priority of a thread, even if the thread has been running for a certain period.
- The priority value is important because the contract between the Java virtual machine and the underlying operating system is that the operating system must generally choose to run the Java thread with the highest priority.

Notes on Thread Priorities

1. Remember that all the threads carry normal priority when a priority is not specified.
2. Priorities can be specified from 1 to 10. 10 being the highest, 1 being the lowest priority and 5 being the normal priority.
3. Remember that the thread with highest priority will be given preference in execution. But there is no guarantee that it will be in running state the moment it starts.
4. Always the currently executing thread might have the higher priority when compared to the threads in the pool who are waiting for their chance.
5. It is the thread scheduler which decides what thread should be executed.
6. t.setPriority() can be used to set the priorities for the threads.
7. Remember that the priorities should be set before the threads start method is invoked.
8. You can use the constants, MIN_PRIORITY, MAX_PRIORITY and NORM_PRIORITY for setting priorities.

```java
public class YieldExample
{
   public static void main(String[] args)
   {
      Thread producer = new Producer();
      Thread consumer = new Consumer();

      producer.setPriority(Thread.MIN_PRIORITY); //Min Priority
      consumer.setPriority(Thread.MAX_PRIORITY); //Max Priority

      producer.start();
      consumer.start();
   }
}
```

```java
class Producer extends Thread
{
   public void run()
   {
      for (int i = 0; i < 5; i++)
      {
         System.out.println("I am Producer : Produced Item " + i);
         Thread.yield();
      }
   }
}
```

```java
class Consumer extends Thread
{
   public void run()
   {
      for (int i = 0; i < 5; i++)
      {
         System.out.println("I am Consumer : Consumed Item " + i);
         Thread.yield();
      }
   }
}
```

Output

“without” yield() method

```
I am Consumer : Consumed Item 0
I am Consumer : Consumed Item 1
I am Consumer : Consumed Item 2
I am Consumer : Consumed Item 3
I am Consumer : Consumed Item 4
I am Producer : Produced Item 0
I am Producer : Produced Item 1
I am Producer : Produced Item 2
I am Producer : Produced Item 3
I am Producer : Produced Item 4
```

“with” yield() method

```
I am Producer : Produced Item 0
I am Consumer : Consumed Item 0
I am Producer : Produced Item 1
I am Consumer : Consumed Item 1
I am Producer : Produced Item 2
I am Consumer : Consumed Item 2
I am Producer : Produced Item 3
I am Consumer : Consumed Item 3
I am Producer : Produced Item 4
I am Consumer : Consumed Item 4
```

join() method

- The join() method of a Thread instance can be used to "join" the start of a thread's execution to the end of another thread's execution so that a thread will not start running until another thread has ended.
- If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.

```
public final void join() throws InterruptedException
```

Example

```
public class Threadjoiningmethod extends Thread{
          public void run(){
           for(int i=1;i<=4;i++){
            try{
             Thread.sleep(500);
            }catch(Exception e){System.out.println(e);}
           System.out.println(i);
           }
          }
         public static void main(String args[]){
             Threadjoiningmethod th1=new Threadjoiningmethod ();
             Threadjoiningmethod th2=new Threadjoiningmethod ();
             Threadjoiningmethod th3=new Threadjoiningmethod ();
          th1.start();
          try{
           th1.join();
          }
          catch(Exception e){
              System.out.println(e);
              }

          th2.start();
          th3.start();
          }
         }
```

```
1
2
3
4
1
1
2
2
3
3
4
4
```

What is thread pool?

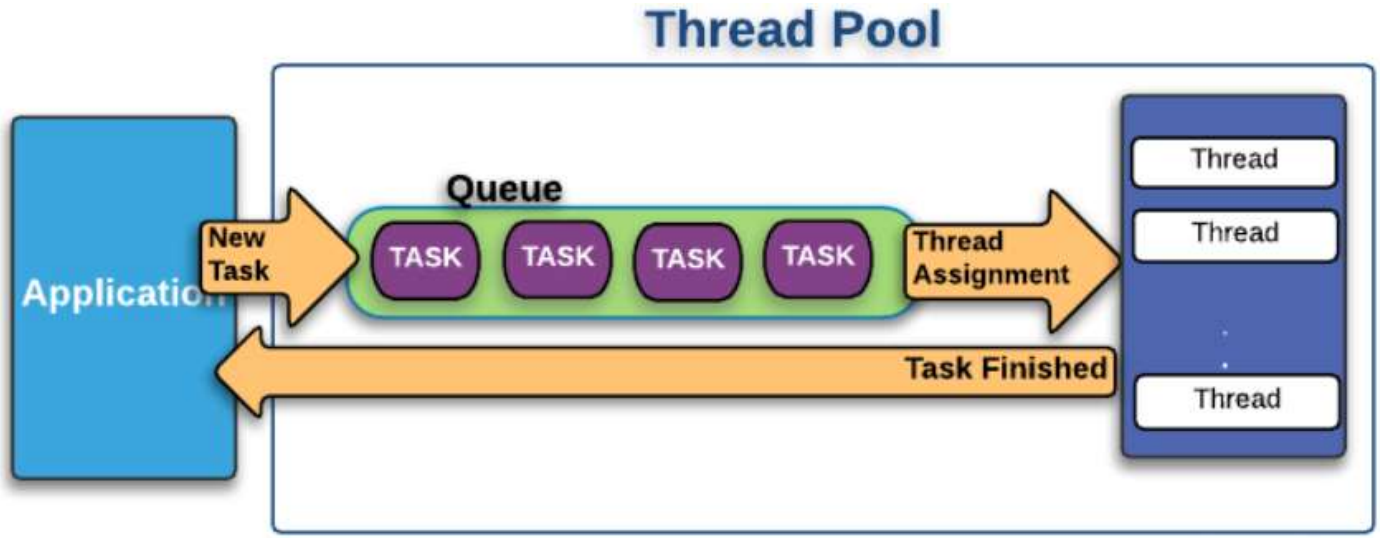
- A thread pool is a collection of pre-initialized threads.
- Generally, the size of the collection is fixed, but it is not mandatory.
- It facilitates the execution of N number of tasks using the same threads.
- If there are more tasks than threads, then tasks need to wait in a queue like structure (FIFO – First in first out).
- When any thread completes its execution, it can pickup a new task from the queue and execute it. When all tasks are completed, the threads remain active and wait for more tasks in the thread pool.

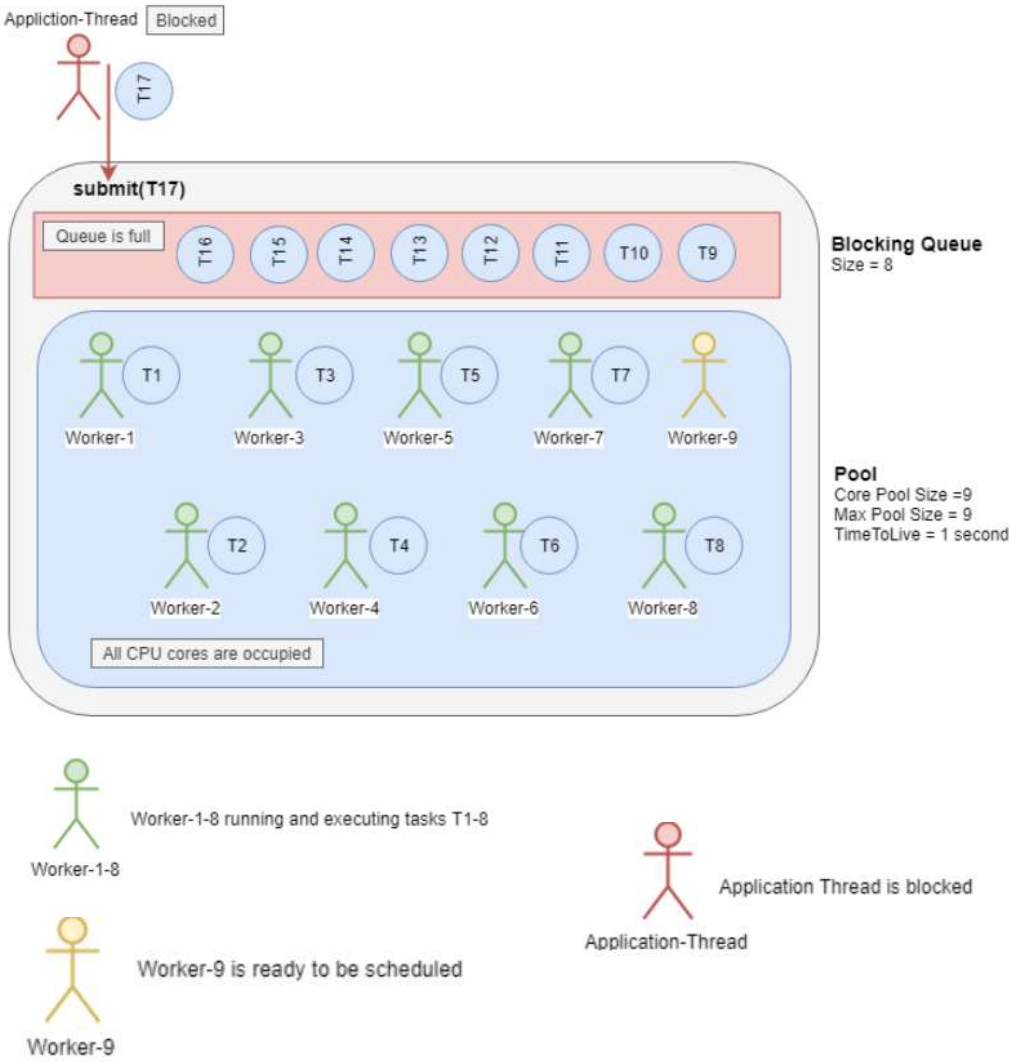
What is thread pool?

- Thread pools are the ones that use the previous threads to perform current tasks.
- Through this, we can solve the problem of resource thrashing and memory wasting.
- It saves the time of thread creation, and hence the response is very nice.
- Threads are mapped to system-level- threads in Java that directly refer to operating system resources.

Thread Pool

A watcher keep watching queue (usually BlockingQueue) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them.





ThreadPool

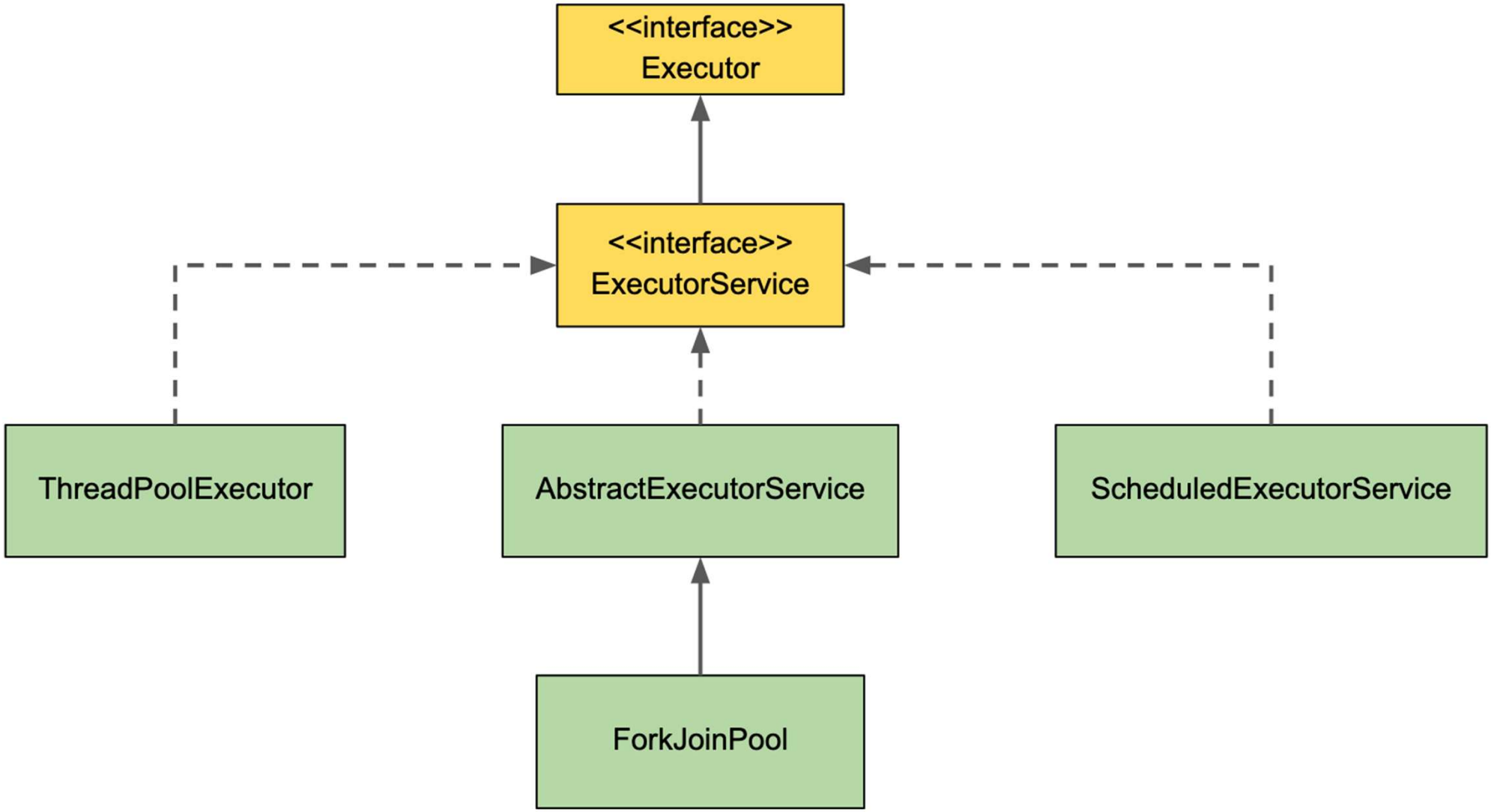
1. Application thread submitted 16 tasks to the thread pool
2. 8 worker threads in the pool have taken 8 tasks from the queue and are executing the tasks
3. Remaining tasks are queued as workers are busy
4. The 9th worker thread is ready to be scheduled. But it cannot run yet as all the CPU cores are occupied by Workers 1-8
5. Remaining 8 tasks are queued in the Blocking Queue. Effectively, the application is allowed to submit 8+8=16 tasks to the thread pool. After this application thread is blocked.
6. When application thread tried submitting the 17th task, application thread is blocked by the queue as the queue is full

ThreadPoolExecutor

- Since Java 5, the Java concurrency API provides a mechanism Executor framework. This is around the Executor interface, its sub-interface ExecutorService, and the ThreadPoolExecutor class that implements both interfaces.
- ThreadPoolExecutor separates the task creation and its execution.
- With ThreadPoolExecutor, you only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads.
- The Executor framework relieved Java application developers from the responsibility of creating and managing threads.

Executors, Executor and executor service

- Java provides an executor framework centered around the Executor interface. Its sub-interface is executor services, and the class names thread pool executor, implementing both interfaces.
- With the help of the executor, we can handle the running object and send them to the executor to execute these things.
- The main focus is on the task performance by the threads.


<<interface>>
Executor
<<interface>>
ExecutorService
ThreadPoolExecutor
AbstractExecutorService
ScheduledExecutorService
ForkJoinPool

Executor in Java

- Java implements a thread pool based on an executor that is present in java.util.concurrent package.
- It is an interface that is used for executing tasks in a thread pool.
- Executor interface provides a method named execute().

```
public void execute(Runnable object)
```

- This method executes the runnable task.

ExecutorService in Java

- ExecutorService is a sub-interface of executor interface that is used for managing and controlling tasks.
- It is also present in java.util.concurrent package.
- ExecutorService interface in Java provides four useful methods.
  - shutdown(): used to shut down the executor but allow tasks in the executor to complete. Once the executor shuts down, it cannot accept new tasks.
  - shutdownNow(): used to shut down the executor immediately even though there are unfinished tasks in the pool. It also returns the list of uncompleted tasks.
  - isShutdown(): This method returns true if the executor has been shut down.
  - isTerminated(): This method returns true if all the tasks in the pool are terminated.

Types of Thread Pool Executors

- Thread Pool Executor separates
  - the task of creating and maintaining the thread task through its lifecycle,
  - so the developer doesn't need to focus on maintaining the task executor rather than on the actual business use case.
- Thread Pool Executor class implements both the ExecutorService interface and its parent Executor as well.
- We can create following 5 types of thread pool executors with pre-built methods in java.util.concurrent.Executors interface.

1. Fixed thread pool executor
2. Cached thread pool executor
3. Scheduled thread pool executor
4. Single thread pool executor
5. Work stealing thread pool executor

1. Fixed Size Thread Pool Executor:

- This is the most common thread pool executor.
- This thread pool executor needs to be configured with the initial nos. of threads and the total nos. of threads.
- So during startup, this executor starts with the minimal nos. of threads and depends upon the increasing load it increases the nos. of threads till the maximum no. configured.
- After that limit, if a new task arrives it will wait in the waiting queue unless any thread gets free from its task.

```
ThreadPoolExecutor executor  = (ThreadPoolExecutor) Executors.newFixedThreadPool(5);
```

2. Cached Thread Pool Executor:

- Creates a thread pool that creates the thread on demand.
- unlike fixed size,it creates the new threads if a new task arrives and there are no existing free threads available in the pool.
- Creating of new threads is not limited, it is only limited till memory supports creating new threads.
- any thread is created but is in an idle state for more than 60 seconds, the thread will get terminated from the thread pool

ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();

3. Scheduled Thread Pool Executor:

- It is a kind of fixed thread pool executor with pre-defined nos. of threads available
- the scheduled thread pool is configured to execute the task at a certain time or after a certain time period.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10);
```

4. Single Thread Pool Executor:

- This kind of thread pool executor contains a single thread to handle the tasks.
- This kind of executor is useful when the incoming task is a very less amount in a particular time frame or the result of a task is not needed ASAP
- in these cases, we can think of creating this single thread pool executor to let a single thread to take care all the tasks one by one.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor();
```

5. Work Stealing Thread Pool Executor:

- This kind of thread pool works on the principle of parallelism of a task by forking a task and then joining at the end.
- It maintains enough threads to support multi-processor processing.
- a task can be forked into multiple tasks and it will be shared in a pool which is accessed by a different processor.
- So, once a task is pushed into this pool and there is a thread available in another processor may take up (steal) the task and continue with it.

Example – Create task

```
import java.util.concurrent.*;

class DemoTask implements Runnable {
    private String name;

    public DemoTask(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void run() {
        try {
            System.out.println("Executing : " + name);
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Example - Execute tasks with thread pool executor

```
public class ThreadPoolEx1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(2);

        for (int i = 1; i <= 5; i++)
        {
            DemoTask task = new DemoTask("Task " + i);
            System.out.println("Created : " + task.getName());

            executor.execute(task);
        }
        executor.shutdown();

    }

}
```

The given program creates 5 tasks and submit to the executor queue. The executor uses two threads to execute all tasks.

Difference between Executor, ExecutorService and Executors class

- Executor, ExecutorService, and Executors are part of Java's Executor framework which provides thread pool facilities to Java applications.
- Since the creation and management of Threads are expensive and the operating system also imposes restrictions on how many Threads an application can spawn, it's a good idea is to use a pool of threads to execute tasks in parallel, instead of creating a new thread every time a request comes in.
- This not only improves the response time of the application but also prevents resource exhaustion errors like "java.lang.OutOfMemoryError: unable to create new native thread".

Difference

- The main difference between Executor, ExecutorService, and Executors class is that Executor is the core interface which is an abstraction for parallel execution.
- It separates tasks from execution, this is different from java.lang.Thread class which combines both task and its execution.
- ExecutorService is an extension of the Executor interface and provides a facility for returning a Future object and terminate or shut down the thread pool. Once the shutdown is called, the thread pool will not accept new tasks but complete any pending task.
- ExecutorService also provides a submit() method which extends Executor.execute() method and returns a Future.

Difference

- One of the key differences between Executor and ExecutorService interface is that the former is a parent interface while ExecutorService extends Executor, i.e., it's a sub-interface of Executor.
- Executor defines execute() method which accepts an object of the Runnable interface, while submit() method can accept objects of both Runnable and Callable interfaces.
- The execute() method doesn't return any result, its return type is void but the submit() method returns the result of computation via a Future object.