ASSIGNMENT 2

GENERAL

The Java platform includes a package of concurrency utilities. These are classes that are designed to be used as building blocks in building concurrent classes and applications.
Handling the design complexity for advanced use-cases demands developers to extend the functionality of these built-in concurrency utilities.

the following document describes specific limitations of the Java's concurrency designs, explains the need for a new type of asynchronous operation, specifies the requirements and a concise technical background. Read this document thoroughly before getting started with your solution.

BACKGROUND

JAVA THREADS

According to Javadoc, a thread is a thread of execution in a program. The Java Virtual Machine (JVM) allows a Java application to have multiple threads of execution running concurrently. The *Thread* class supports the creation of platform threads that are typically mapped 1:1 to kernel threads, scheduled by the operating system.

The JVM implements platform threads as wrappers around operating system (OS) threads and will usually have a large stack and other resources that are maintained by the operating system**.** The *Thread* class defines 6 constructors that declare a *Runnable* as parameter.
Runnable is an interface representing an operation that does not return a value. This operation may be executed in a separate thread, i.e., asynchronously using the run() method.

PRIORITY-BASED TASK SCHEDULING

The Java Virtual machine (JVM) schedules threads using a preemptive, priority-based policy.
Every thread has a priority – Threads with higher priority are executed in preference to threads with lower priority. When code running in a thread creates a new *Thread* object, the new thread has its initial priority set automatically equal to the priority of the creating thread.
If a *thread* was created using a different *ThreadGroup,* the priority of the newly created thread is the smaller of priority of the thread creating it and the maximum permitted priority of the thread group.

If a thread with a higher priority than the currently running thread enters the RUNNABLE state, the scheduler preempts the executing thread schedules the thread with the higher priority to run.
The Scheduler may also invoke a different thread to run if the currently running thread changes state from RUNNABLE to a different state such as BLOCKED, WAITING or TERMINATED.

A user may set the priority of a thread using the method: public final void setPriority(int newPriority).
The setPriority method changes the priority of a thread. For platform threads, the priority is set to the smaller of the specified newPriority and the maximum permitted priority of the thread group.

BUILT-IN LIMITATIONS

Java enables developers to set the priority of a thread, but not the *Runnable* operation it executes. Tightly coupling the operation with the execution path that runs it creates major drawback when using an executor such as a *ThreadPoolExecutor:* the collection of threads in an executor is defined by a *ThreadFactory*. By default, it creates all threads with the same priority and non-daemon status.

Moreover, if we wish to execute a returning value operation, for example using the *Callable<V>* interface, there are no constructors in the *Thread* class that get a *Callable<V>* as parameter and we ought to use an Executor of some type, such as a *ThreadPoolExecutor*.

OBJECTIVE

Your goal is to create two new types that extend the functionality of Java's Concurrency Framework:

1. A **generic task with** a *Type* that returns a result and may throw an exception.
   Each task has **a priority** used for scheduling, inferred from the integer value of the task's T*ype.*
2. A custom thread pool class that defines a method for submitting a generic task as described in the section 1 to a priority queue, and a method for submitting a generic task created by a *Callable<V>* and a *Type*, passed as arguments.

See partial API for both classes in the requirements section.
Use the *Type* Enum to descript a *Task*'s type:

```
public enum TaskType {
  COMPUTATIONAL(1){
    @Override
    public String toString(){return "Computational Task";}

  },
  IO(2){
    @Override
    public String toString(){return "IO-Bound Task";}
  },
  OTHER(3){
    @Override
    public String toString(){return "Unknown Task";}
  };
```

```java
    private int typePriority;

    private TaskType(int priority){
        if (validatePriority(priority)) typePriority = priority;
        else
            throw new IllegalArgumentException("Priority is not an integer");
    }

    public void setPriority(int priority){
        if(validatePriority(priority)) this.typePriority = priority;
        else
            throw new IllegalArgumentException("Priority is not an integer");
    }

    public int getPriorityValue(){
        return typePriority;
    }

    public TaskType getType(){
        return this;
    }

    /**
     * priority is represented by an integer value, ranging from 1 to 10
     * @param priority
     * @return whether the priority is valid or not
     */
    private static boolean validatePriority(int priority){
        if (priority < 1 || priority >10) return false;
        return true;
    }
}
```

REQUIREMENTS

The *Task* class

3. Represents a **task with a *TaskType*** and may return **a value of some type**
4. It may throw an exception if unable to compute the result
5. instances are potentially **executed by another thread** in one of the threads in *CustomExecutor*
6. It may be submitted to the queue in *CustomExecutor*
7. The ***TaskType*** member is used to return an integer value, representing the instance's priority.
8. Exposes public factory methods for safe creation
9. Creation of a *Task* instance requires passing at least:
   An operation that may run **asynchronously with a return value**
10. It is also possible to pass A ***TaskType*** as argument for instance creation.
11. **Natural order** for *Task* instances ought to be determined **by the priority** of the ***TaskType*** member when using ordered data structures

12. Consider constructor chaining for readability and clean code
13. You may **determine** the class **definition**, method **signatures** and **implementation**, access modifiers, additional data structures and attributes
14. Consider **Object-Oriented Design principles** such as S.O.L.I.D and other best practices we've learnt throughout the course
15. Your design will be evaluated as well as implementation

The *CustomExecutor* class

16. An Executor that asynchronously computes ***Task* instances**
17. User may submit:
    - A *Task* instance
    - An operation that may return a value. It will then be **used for creating a *Task* instance**
    - An operation that may return a value and a *TaskType.* It will then be **used for creating a *Task* instance**
18. Consider method chaining for readability and clean code
19. The pool's **queue** should maintain elements according to their **natural order.**
20. Unlike *Thread* scheduling, where threads with higher priority are executed in preference to threads with lower priority, ordered data structures maintain elements low natural order values **in precedence to elements with greater** natural order values
    An operation that may run **asynchronously with a return value**
21. Priority in ordered data structures - maintain Task instances according to **the integer value** of the TaskType member
22. Maintain the maximum priority of Task instances in the queue at any given time:
    1. Create a method that returns the maximum priority in the queue in O(1) time & space complexity
    2. This method may not access the queue to query the current maximum priority
23. Set the number of **threads to keep in the pool**, **even** if **they are idle** to be **half the** number of **processors** available for the Java Virtual Machine (JVM)
24. Set the maximum number of threads to allow in the pool to be on **the number of processors** available for the Java Virtual Machine (JVM) minus 1
25. when the number of threads is greater than the core, this is the maximum time that excess idle threads will wait **300 milliseconds** for new tasks before terminating
26. After finishing of all tasks submitted to the executor, or if an exception is thrown, terminate the executor
27. You may **determine** the class **definition**, method **signatures** and **implementation**, access modifiers, additional data structures and attributes
28. Consider **Object-Oriented Design principles** such as S.O.L.I.D and other best practices we've learnt throughout the course
29. Your design will be evaluated as well as implementation

JUNIT EXAMPLE

The following is a **partial** JUnit. The ***var*** keyword is used to infer the type of the variable without explicitly declaring it, according to its local context

I've used it for obfuscation purposes (so it would be more **difficult for you to reverse-engineer** the code from the following test :-)

```
public class Tests {
    public static final Logger logger = LoggerFactory.getLogger(Tests.class);

    @Test
    public void partialTest(){
        CustomExecutor customExecutor = new CustomExecutor();
        var task = Task.createTask(()->{
            int sum = 0;
            for (int i = 1; i <= 10; i++) {
                sum += i;
            }
            return sum;
        }, TaskType.COMPUTATIONAL);

        var sumTask = customExecutor.submit(task);

        final int sum;
        try {
            sum = sumTask.get(1, TimeUnit.MILLISECONDS);
        } catch (InterruptedException | ExecutionException | TimeoutException e) {
            throw new RuntimeException(e);
        }

        logger.info(()-> "Sum of 1 through 10 = " + sum);

        Callable<Double> callable1 = ()-> {
            return 1000 * Math.pow(1.02, 5);
        };

        Callable<String> callable2 = ()-> {
            StringBuilder sb = new StringBuilder("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
            return sb.reverse().toString();
        };


        var priceTask = customExecutor.submit(()-> {
            return 1000 * Math.pow(1.02, 5);
        }, TaskType.COMPUTATIONAL);

        var reverseTask = customExecutor.submit(callable2, TaskType.IO);

        final Double totalPrice;
        final String reversed;
        try {
            totalPrice = priceTask.get();
            reversed = reverseTask.get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
        logger.info(()-> "Reversed String = " + reversed);
        logger.info(()->String.valueOf("Total Price = " + totalPrice));

        logger.info(()-> "Current maximum priority = " +
```

```
        customExecutor.getCurrentMax());
        customExecutor.gracefullyTerminate();
    }
}
```

TEST OUTPUT (PARTIAL)

Dec 26, 2022 12:28:56 AM Tests partialTest
INFO: Sum of 1 through 10 = 55

Dec 26, 2022 12:28:56 AM Tests partialTest
INFO: Reversed String = ZYXWVUTSRQPONMLKJIHGFEDCBA

Dec 26, 2022 12:28:56 AM Tests partialTest
INFO: Total Price = 1104.0808032

Dec 26, 2022 12:28:56 AM Tests partialTest
INFO: Current maximum priority = 2

ADDITIONAL INSTRUCTIONS

1. Consider how developers might extend your classes (either by inheritance or by composition)
2. Consider how developers might use your classes, for example sorting *Task* instances or maintaining instances in hash-based data-structures
3. Same submission instructions in part 1 apply
4. Add a PDF/README in your repo:

- Generate a Class Diagram in IntelliJ including dependencies
- Describe the design and development considerations and provide techniques/patterns you employed
- Explain the difficulties you have encountered and how you handled them
- Explain how the proposed design contributed to enhance the flexibility, performance, and maintainability of your code

GOOD LUCK