The learning outcome of this practical session is to understand the internals of a basic executor.

The first two sections are essential.

1 Echo Server with a Thread Pool

Let’s consider the following basic echo TCP server:

```java
int portNumber = 4444;
try {
    ServerSocket serverSocket = new ServerSocket(portNumber);
    Socket clientSocket = serverSocket.accept();
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
}
```

The corresponding client code is:

```java
String hostName = "127.0.0.1";
int portNumber = 4444;
try {
    Socket kkSocket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(kkSocket.getInputStream()));
}
```

Measure the performance of this sequential server by inserting loops in the server and client code. Ideally, the client code is run on a separate machines (to avoid interference) with multiple threads (to load the server as much as possible).
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
    while (true) {
        Socket clientSocket = serverSocket.accept();
        new Thread(() -> {
            try {
                PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
                BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

                String str = in.readLine();
                if (str != null)
                    out.println(str);

            } catch (Exception e) {
            }
            }
        );
    var t1 = new Thread(r);
    var t2 = new Thread(r);
    var st = System.nanoTime();
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println(((System.nanoTime() - st) / 1e9));
}
Build a second alternative relying on a thread pool to process each request. Measure and compare the performance.

```java
Executor exec = Executors.newFixedThreadPool(10);
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
    while (true) {
        Socket clientSocket = serverSocket.accept();
        exec.execute(() -> {
            try {
                PrintWriter out =
                    new PrintWriter(clientSocket.getOutputStream(), true);
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(clientSocket.getInputStream()));
            } {
                String str = in.readLine();
                if (str != null)
                    out.println(str);
                clientSocket.close();
            } catch (Exception e) {}})
        });
    }
}
```

The performance of this third version shows the overhead related to task creations, which is smaller than with thread creations while still less efficient than concurrent execution.

However, the significant benefit of using an executor is that the server is now treating request in parallel. This is crucial to exploit all CPU resources in a practical system when processing requests that are more complex.

### 2 Cached Threads

This objective of this first part is to write a class that implements interface `Executor` and that behaves as the executor returned by `Executors.newCachedThreadPool()`. In particular, calling method `execute` should result in the immediate creation of a new thread that will asynchronously execute the runnable given as an argument.

```java
class DirectExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```
3 Single Thread

The objective of this second part is to similarly mimic the behavior of `Executors.newSingleThreadExecutor()`. In particular, submitting a runnable should result in its asynchronous execution as soon as a uniquely created thread is available.

Your final implementation should make sure that:

- `execute` is thread-safe (multiple threads may call this method simultaneously);
- the thread is not actively checking for a new task, but waiting;
- a specific mechanism ensures that the number of waiting tasks cannot grow indefinitely (in which case a `RejectedExecutionException` should be thrown).

```java
class SerialExecutor implements Executor {
    final Queue<Runnable> tasks = new ConcurrentLinkedDeque<>();
    final int maxBufferCapacity;

    SerialExecutor(int maxBufferCapacity) {
        this.maxBufferCapacity = maxBufferCapacity;
        new Thread(() -> {
            while (true) {
                Runnable r = tasks.poll();
                if (r != null)
                    r.run();
                else {
                    synchronized(tasks) {
                        if (tasks.isEmpty())
                            try {
                                tasks.wait();
                            } catch (InterruptedException e) {} 
                    }
                }
            }
        }).start();
    }

    public void execute(Runnable r) {
        if (tasks.size() > maxBufferCapacity)
            throw new RejectedExecutionException();
        tasks.add(r);
        if (tasks.peek() == r) // not necessary (notification reduction)
            synchronized(tasks) {
                tasks.notify();
            }
    }
}
```
Compare the performance of this executor with the one provided by Java. Measure the time taken to execute the same set of empty tasks. This load test also allows identifying implementation bug (non thread-safe operations).

```java
//var exec = Executors.newSingleThreadExecutor();
var exec = new SerialExecutor(Integer.MAX_VALUE);
var st = System.nanoTime();
for (int i = 0; i < 10_000_000; i++)
    exec.execute(() -> {});
long res;
exec.execute(() -> { res = System.nanoTime(); });
System.out.println((res - st) / 1e9);
```