Programming Paradigms Tutorials
Multithreading - Java

Threads and Runnables

All modern operating systems support concurrency via processes and threads. Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently, so we can make the most out of the available cores of the CPU.
This is how thread can be created in Java:

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Unfortunately, this concept does not allow to easily control number of threads run at same time and manage or reuse them that why executors and thread pools were created.

Executors

The Concurrency API introduces the concept of an ExecutorService as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually. All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service. Example:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});
```

An ExecutorService provides two methods for that purpose: shutdown() waits for currently running tasks to finish while shutdownNow() interrupts all running tasks and shut the executor down immediately.

In addition to Runnable executors support another kind of task named Callable. Callables are functional interfaces just like runnables but instead of being void they return a value.

Example:

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callables can be submitted to executor services just like runnables. But what about the callables result? Since submit() doesn't wait until the task completes, the executor service cannot return the result of the callable directly. Instead the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

Scheduled Executors

A ScheduledExecutorService is capable of scheduling tasks to run either periodically or once after a certain amount of time has elapsed. Example:

```
ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " +
System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3,
TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1000);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

Exercises

1. What will be printed by first code presented in this instruction and why?

2. Provide two treads which operates on same valuable which represents status of account. One trade does income operations the second one outcome. Do not use synchronization in the program and explain issue which can be observed.

3. Change code from ex. 2 and provide synchronization.

4. Provide solution of Dining Philosophers Problem.

5. Use one of ExecutorServices and print next int number (starting from 0) each second (can be limited to 30 times). Which type of ExecutorService is the best for this task and why?