

High-level Concurrency API

In the previous sections you got familiar with concurrent programming by using the low-level API such as working with threads (create, start, pause, stop), using locks and synchronized keyword. That's good for the basics and simple usages.

For more advanced tasks, using the low-level API is time-consuming and error-prone. That's why the high-level concurrency API is designed to help programmers easily implement more complex multi-threading tasks. Programmers can focus on the business logic of the tasks rather than getting busy in the low-level details.

The high-level concurrency API is implemented in the following 3 packages:

- `java.util.concurrent`: provides utility classes commonly useful in concurrent programming such as executors, threads pool management, scheduled tasks execution, the Fork/Join framework, concurrent collections, etc.
- `java.util.concurrent.locks`: provides `Lock` and `Condition` implementations that are more advanced than the built-in locking and synchronization mechanism.
- `java.util.concurrent.atomic`: provides data type classes that are safely updated without using locks. For example, an `AtomicInteger` can be atomically incremented or decremented so you can use it as a shared variable without synchronization.

In the previous lessons, you learned how to use `Lock` and `Condition` in the `java.util.concurrent.locks` package. I will talk about atomic classes in the next lesson, and today I focus on helping you get familiar with the concurrent utilities in the `java.util.concurrent` package.

With the support of high-level concurrency API, you can do the following things (but not limited to):

- Executing tasks by multiple threads that are managed in a thread pool. You don't have to manage the thread pool yourself, just choose a kind of pool you want and submit the tasks. This can be done via various implementations of ***Executor***.
- Queuing tasks to be executed sequentially, one after another.
- Running a task that computes a value (`Callable`) and waiting for the result (`Future`). This can be done by using an `ExecutorService`.

- Scheduling a task to be executed after a given delay, or to be executed periodically at a fixed rate or fixed delay. This can be done by using the `ScheduleExecutorService` with `ScheduleFuture`.
- Taking the advantages of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively. This can be done with the support of the Fork/Join framework.

Thread Pool and Executors

* Understanding Thread Pool:

In terms of performance, creating a new thread is an expensive operation because it requires the operating system allocates resources need for the thread. Therefore, in practice thread pool is used for large-scale applications that launch a lot of short-lived threads in order to utilize resources efficiently and increase performance.

Instead of creating new threads when new tasks arrive, a thread pool keeps a number of idle threads that are ready for executing tasks as needed. After a thread completes execution of a task, it does not die. Instead it remains idle in the pool waiting to be chosen for executing new tasks.

You can limit a definite number of concurrent threads in the pool, which is useful to prevent overload. If all threads are busily executing tasks, new tasks are placed in a queue, waiting for a thread becomes available.

That's basically how thread pool works. In practice, thread pool is used widely in web servers where a thread pool is used to serve client's requests. Thread pool is also used in database applications where a pool of threads maintaining open connections with the database.

Implementing a thread pool is a complex task, but you don't have to do it yourself. As the Java Concurrency API allows you to easily create and use thread pools without worrying about the details.

* Understanding Executors:

An `Executor` is an object that is responsible for threads management and execution of `Runnable` tasks submitted from the client code. It decouples the details of thread creation, scheduling, etc from the task submission so you can focus on developing the task's business logic without caring about the thread management details.

That means, in the simplest case, rather than creating a thread to execute a task like this:

```
Thread t = new Thread(new RunnableTask());  
  
t.start();
```

You submit tasks to an executor like this:

```
Executor executor = anExecutorImplementation;  
  
executor.execute(new RunnableTask1());  
  
executor.execute(new RunnableTask2());
```

The Java Concurrency API defines the following 3 base interfaces for executors:

- **Executor**: is the super type of all executors. It defines only one method `execute(Runnable)`.
- **ExecutorService**: is an `Executor` that allows tracking progress of value-returning tasks (`Callable`) via `Future` object, and manages the termination of threads. Its key methods include `submit()` and `shutdown()`.
- **ScheduledExecutorService**: is an `ExecutorService` that can schedule tasks to execute after a given delay, or to execute periodically. Its key methods are `schedule()`, `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`.

You can create an executor by using one of several factory methods provided by the Executors utility class. Here's to name a few:

- **`newCachedThreadPool()`**: creates an expandable thread pool executor. New threads are created as needed, and previously constructed threads are reused when they are available. Idle threads are kept in the pool for one minute. This executor is suitable for applications that launch many short-lived concurrent tasks.
- **`newFixedThreadPool(int n)`**: creates an executor with a fixed number of threads in the pool. This executor ensures that there are no more than `n` concurrent threads at any time. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread becomes available. If any thread terminates due to failure during execution, it will be replaced by a new one. The threads in the pool will exist until it is explicitly shutdown. Use this executor if you want to limit the maximum number of concurrent threads.
- **`newSingleThreadExecutor()`**: creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more

than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another.

- `newScheduledThreadPool(int corePoolSize)`: creates an executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute concurrently.
- `newSingleThreadScheduleExecutor()`: creates a single-threaded executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute sequentially.

In case the factory methods do not meet your need, you can construct an executor directly as an instance of either `ThreadPoolExecutor` or `ScheduledThreadPoolExecutor`, which gives you additional options such as pool size, on-demand construction, keep-alive times, etc.

* A Simple Executor Example:

The following code snippet shows you a simple example of executing a task by a single-threaded executor:

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

public class SimpleExecutorExample {

    public static void main(String[] args) {

        ExecutorService pool = Executors.newSingleThreadExecutor();

        Runnable task = new Runnable() {

            public void run() {

                System.out.println(Thread.currentThread().getName());

            }

        };

        pool.execute(task);

        pool.shutdown();

    }

}
```

```
}
```

As you can see, a Runnable task is created using anonymous-class syntax. The task simply prints the thread name and terminates. Compile and run this program and you will see the output something like this:

```
pool-1-thread-1
```

Note that you should call `shutdown()` to destroy the executor after the thread completes execution. Otherwise, the program is still running afterward. You can observe this behavior by commenting the call to shutdown.

Executing Multiple Tasks by Different Executors

* Using a cached thread pool executor:

Given the following class:

```
public class CountdownClock extends Thread {

    private String clockName;

    public CountdownClock(String clockName) {

        this.clockName = clockName;
    }

    public void run() {

        String threadName = Thread.currentThread().getName();

        for (int i = 5; i >= 0; i--) {

            System.out.printf("%s -> %s: %d\n", threadName,
clockName, i);

            try {

                Thread.sleep(1000);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

    }

}
```

```

    }

}

```

This class represents a countdown clock that counts a number from 5 down to 0, and pause 1 second after every count. Upon running, it prints the current thread name, follows by the clock name and the count number.

Let's create an executor with a cached thread pool to execute 4 clocks concurrently. Here's the code:

```

import java.util.concurrent.*;

public class MultipleTasksExecutorExample {

    public static void main(String[] args) {

        ExecutorService pool = Executors.newCachedThreadPool();

        pool.execute(new CountdownClock("A"));

        pool.execute(new CountdownClock("B"));

        pool.execute(new CountdownClock("C"));

        pool.execute(new CountdownClock("D"));

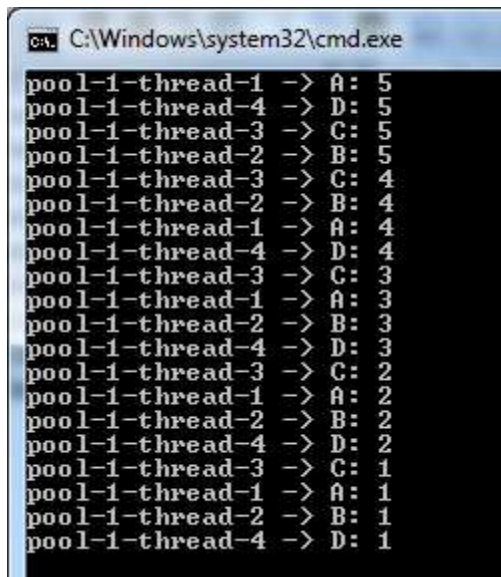
        pool.shutdown();

    }

}

```

Compile and run this program, you will see that there are 4 threads executing the 4 clocks concurrently:



```
C:\Windows\system32\cmd.exe
pool-1-thread-1 -> A: 5
pool-1-thread-4 -> D: 5
pool-1-thread-3 -> C: 5
pool-1-thread-2 -> B: 5
pool-1-thread-3 -> C: 4
pool-1-thread-2 -> B: 4
pool-1-thread-1 -> A: 4
pool-1-thread-4 -> D: 4
pool-1-thread-3 -> C: 3
pool-1-thread-1 -> A: 3
pool-1-thread-2 -> B: 3
pool-1-thread-4 -> D: 3
pool-1-thread-3 -> C: 2
pool-1-thread-1 -> A: 2
pool-1-thread-2 -> B: 2
pool-1-thread-4 -> D: 2
pool-1-thread-3 -> C: 1
pool-1-thread-1 -> A: 1
pool-1-thread-2 -> B: 1
pool-1-thread-4 -> D: 1
```

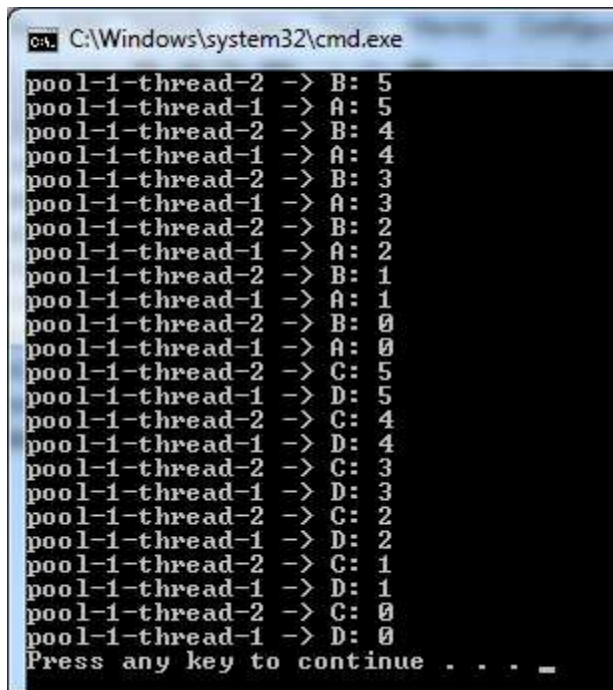
Modify this program to add more tasks e.g. add more 3 clocks. Recompile and run the program again, you will see that the number of threads is as equal as the number of submitted tasks. That's the key behavior of a cached thread pool: new threads are created as needed.

* Using a fixed thread pool executor:

Next, update the statement that creates the executor to use a fixed thread pool:

```
ExecutorService pool = Executors.newFixedThreadPool(2);
```

Here, we create an executor with a pool of maximum 2 concurrent threads. Keep only 4 task (4 clocks) submitted to the executor. Recompile and run the program you will see that there are only 2 threads executing the clocks:



```
C:\Windows\system32\cmd.exe
pool-1-thread-2 -> B: 5
pool-1-thread-1 -> A: 5
pool-1-thread-2 -> B: 4
pool-1-thread-1 -> A: 4
pool-1-thread-2 -> B: 3
pool-1-thread-1 -> A: 3
pool-1-thread-2 -> B: 2
pool-1-thread-1 -> A: 2
pool-1-thread-2 -> B: 1
pool-1-thread-1 -> A: 1
pool-1-thread-2 -> B: 0
pool-1-thread-1 -> A: 0
pool-1-thread-2 -> C: 5
pool-1-thread-1 -> D: 5
pool-1-thread-2 -> C: 4
pool-1-thread-1 -> D: 4
pool-1-thread-2 -> C: 3
pool-1-thread-1 -> D: 3
pool-1-thread-2 -> C: 2
pool-1-thread-1 -> D: 2
pool-1-thread-2 -> C: 1
pool-1-thread-1 -> D: 1
pool-1-thread-2 -> C: 0
pool-1-thread-1 -> D: 0
Press any key to continue . . . _
```

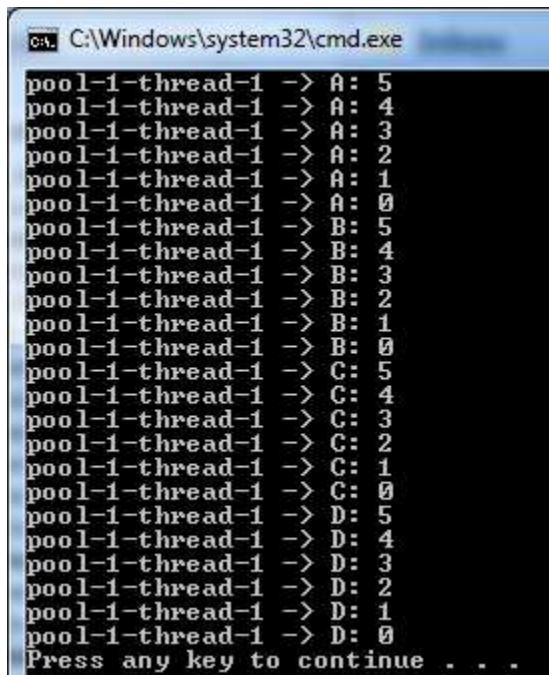
The clocks A and B run first, while the clocks C and D are waiting in the queue. After A and B completes execution, the 2 threads continue executing the clocks C and D. That's the key behavior of a fixed thread pool: limiting the number of concurrent threads and queuing additional tasks.

* Using a single-threaded pool executor:

Let's update the program above to use a single-threaded executor like this:

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

Recompile and run the program, you will see that there's only one thread executing the 4 clocks sequentially:



```
C:\Windows\system32\cmd.exe
pool-1-thread-1 -> A: 5
pool-1-thread-1 -> A: 4
pool-1-thread-1 -> A: 3
pool-1-thread-1 -> A: 2
pool-1-thread-1 -> A: 1
pool-1-thread-1 -> A: 0
pool-1-thread-1 -> B: 5
pool-1-thread-1 -> B: 4
pool-1-thread-1 -> B: 3
pool-1-thread-1 -> B: 2
pool-1-thread-1 -> B: 1
pool-1-thread-1 -> B: 0
pool-1-thread-1 -> C: 5
pool-1-thread-1 -> C: 4
pool-1-thread-1 -> C: 3
pool-1-thread-1 -> C: 2
pool-1-thread-1 -> C: 1
pool-1-thread-1 -> C: 0
pool-1-thread-1 -> D: 5
pool-1-thread-1 -> D: 4
pool-1-thread-1 -> D: 3
pool-1-thread-1 -> D: 2
pool-1-thread-1 -> D: 1
pool-1-thread-1 -> D: 0
Press any key to continue . . .
```

That's the key behavior of a single-threaded executor: queue tasks to execute in order, one after another.

Executing Value-Returning Tasks with Callable and Future

So far we have executed tasks that do not return any value (void). How about tasks those compute and return a value upon completion? Those tasks may take long time to finish, and what if we want to wait for the results?

The `ExecutorService` interface defines a method that allows us to execute such kind of task:

```
< T > Future< T > submit(Callable< T > task)
```

Here, the type parameter `T` is the return type of the task. You submit a task that implements the `Callable` interface which defines only one method as follows:

```
public interface Callable< T > {

    public T call();

}
```

The purpose of the `Callable` interface is similar to `Runnable`, but its method returns a value of type `T`.

Once the task is submitted, the executor immediately returns an object of type `Future` representing the pending results of the task, for example:

```
Callable< Integer > task = new task that returns an Integer value;

Future< Integer > result = executor.submit(task);
```

Then you can invoke the `Future`'s `get()` method to obtain the result upon successful completion. There are two overloads of this method defined as follows:

```
public interface Future< T > {

    T get();

    T get(long timeout, TimeUnit unit);

}
```

The first overload version waits if necessary for the computation to complete and then retrieves its result:

```
Integer value = result.get();
```

This method blocks the current thread to wait until the computation completes and returns the value. In case you want to wait only for a specified amount of time, use the second overload version:

```
Integer value = result.get(2, TimeUnit.SECONDS);
```

This call waits if necessary for at most 2 seconds for the computation to complete, and then retrieves the result if available. If the task takes longer time to complete, the call returns null.

Now, let's see a complete example.

Suppose that we have two tasks: the first calculates the factorial value of N numbers, and the second computes the sum of N numbers.

Implementing the `Callable` interface, the first task is written as follows:

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

public class FactorialCalculator implements Callable< Integer > {

    private int n;

    public FactorialCalculator(int n) {
```

```

        this.n = n;
    }

    public Integer call() {

        int result = 1;

        for (int i = 1; i <= n; i++) {

            result = result * i;

        }

        try {

            Thread.sleep(5000);

        } catch (InterruptedException ex) {

            ex.printStackTrace();

        }

        return result;

    }

}

```

Here we use the `sleep()` method to fake the computation time. And code for the second task:

```

import java.util.concurrent.*;

public class SumCalculator implements Callable< Integer > {

    private int n;

    public SumCalculator(int n) {

        this.n = n;

    }

    public Integer call() {

        int sum = 0;

        for (int i = 1; i <= n; i++) {

```

```

        sum += i;
    }

    try {

        Thread.sleep(2000);

    } catch (InterruptedException ex) {

        ex.printStackTrace();

    }

    return sum;

}
}

```

The following program submits two tasks above to a fixed thread pool executor:

```

import java.util.concurrent.*;

public class CallableAndFutureExample {

    public static void main(String[] args) {

        ExecutorService pool = Executors.newFixedThreadPool(2);

        Future< Integer > sumResult = pool.submit(new
SumCalculator(100000));

        Future< Integer > factorialResult = pool.submit(new
FactorialCalculator(8));

        try {

            Integer sumValue = sumResult.get();

            System.out.println("Sum Value = " + sumValue);

            Integer factorialValue = factorialResult.get();

            System.out.println("Factorial Value = " +
factorialValue);

        } catch (InterruptedException | ExecutionException ex) {

            ex.printStackTrace();

        }

    }

}

```

```

    }

    pool.shutdown();

}

}

```

Run this program and observe the result. The first task, `SumCalculator` takes 2 seconds to complete and you see the result displayed after 2 seconds:

```
Sum Value = 705082704
```

The second task, `FactorialCalculator` takes 5 seconds to complete, so you see the result 3 seconds after the first result:

```
Factorial Value = 40320
```

In addition, the `Future` interface provides methods for checking the completion status:

It's your exercise to experiment with these additional methods.

- `boolean isDone()`: returns `true` if this task completed.
- `boolean isCancelled()`: returns `true` if this task was cancelled before it completed normally.

and for stopping the task:

- `boolean cancel(boolean mayInterruptIfRunning)`: attempts to cancel execution of this task. Returns `false` if the task could not be cancelled, typically because it has already completed normally; `true` otherwise.

Scheduling Tasks to Execute After a Delay or Periodically

Implementations of the `ScheduledExecutorService` interface provide convenient methods for scheduling tasks:

- **`schedule(Callable callable, long delay, TimeUnit unit)`**: executes a `Callable` task after the specified delay. `TimeUnit` can be in `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, etc.
- **`schedule(Runnable command, long delay, TimeUnit unit)`**: Executes a `Runnable` task after a given delay.

- **scheduleAtFixedRate**(Runnable command, long initialDelay, long delay, TimeUnit unit): Executes a periodic task after an initial delay, then repeat after every given period. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.
- **scheduleWithFixedDelay**(Runnable command, long initialDelay, long delay, TimeUnit unit): Executes a periodic task after an initial delay, then repeat after every given delay between the termination of one execution and the commencement of the next.

All these methods return a `ScheduleFuture` object which is a `Future` with an additional method for checking the remaining delay time:

```
long getDelay(TimeUnit unit)
```

And as shown previously, a `ScheduledExecutorService` object can be created via factory methods of the `Executors` utility class:

- **newScheduledThreadPool**(int poolSize): creates a thread pool that can schedule tasks to execute concurrently.
- **newSingleThreadScheduledExecutor**(): creates a single-threaded executor that can schedule tasks to execute sequentially.

Now, let's see some examples.

This is the simplest example that executes a task after 5 seconds:

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

public class SimpleScheduledExecutorExample {

    public static void main(String[] args) {

        ScheduledExecutorService scheduler =
            Executors.newSingleThreadScheduledExecutor();

        Runnable task = new Runnable() {

            public void run() {

                System.out.println("Hi!");

            }

        };

    }
}
```

```

        scheduler.schedule(task, 5, TimeUnit.SECONDS);

        scheduler.shutdown();

    }

}

```

As you can see, this program simply prints the message “Hi!” after a delay of 5 seconds, and terminates.

Next, the following program plays a sound ‘beep’ for every 2 seconds:

```

import java.util.concurrent.*;

public class BeepClock implements Runnable {

    public void run() {

        System.out.print("\007");

    }

    public static void main(String[] args) {

        ScheduledExecutorService scheduler =
            Executors.newSingleThreadScheduledExecutor();

        scheduler.scheduleAtFixedRate(new BeepClock(), 4, 2,
            TimeUnit.SECONDS);

    }

}

```

Notice that, with the execution of periodic tasks, do not call `shutdown()` on the executor because it causes the program to terminate immediately.

The following is a more complex example that uses a pool of 3 threads to schedule 3 count down clocks to execute concurrently:

```

import java.util.concurrent.*;

public class ConcurrentScheduleTasksExample {

    public static void main(String[] args) {

        ScheduledExecutorService scheduler =
            Executors.newScheduledThreadPool(3);
    }
}

```

```

        CountDownClock clock1 = new CountDownClock("A");

        CountDownClock clock2 = new CountDownClock("B");

        CountDownClock clock3 = new CountDownClock("C");

        scheduler.scheduleWithFixedDelay(clock1, 3, 10,
            TimeUnit.SECONDS);

        scheduler.scheduleWithFixedDelay(clock2, 3, 15,
            TimeUnit.SECONDS);

        scheduler.scheduleWithFixedDelay(clock3, 3, 20,
            TimeUnit.SECONDS);

    }

}

```

Here, you can see 3 clocks A, B and C are scheduled to start at the same time, after an initial delay of 3 seconds, but their periodic delay times are different. The following screenshot shows output of this program:

```

C:\Windows\system32\cmd.exe
pool-1-thread-2 -> B: 5
pool-1-thread-1 -> A: 5
pool-1-thread-3 -> C: 5
pool-1-thread-2 -> B: 4
pool-1-thread-3 -> C: 4
pool-1-thread-1 -> A: 4
pool-1-thread-2 -> B: 3
pool-1-thread-3 -> C: 3
pool-1-thread-1 -> A: 3
pool-1-thread-2 -> B: 2
pool-1-thread-3 -> C: 2
pool-1-thread-1 -> A: 2
pool-1-thread-2 -> B: 1
pool-1-thread-3 -> C: 1
pool-1-thread-1 -> A: 1
pool-1-thread-2 -> B: 0
pool-1-thread-3 -> C: 0
pool-1-thread-1 -> A: 0

```

You can use the returned `ScheduleFuture` object to cancel the tasks. This updated version of the program above stops the 3 clocks after 2 minutes, by using another schedule task:

```

import java.util.concurrent.*;

public class ConcurrentScheduleTasksExample {

    public static void main(String[] args) {

```



```

        ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(3);

        CountDownLatch clock1 = new CountDownLatch("A");

        CountDownLatch clock2 = new CountDownLatch("B");

        CountDownLatch clock3 = new CountDownLatch("C");

        Future< ? > f1 = scheduler.scheduleWithFixedDelay(clock1,
3, 10, TimeUnit.SECONDS);

        Future< ? > f2 = scheduler.scheduleWithFixedDelay(clock2,
3, 15, TimeUnit.SECONDS);

        Future< ? > f3 = scheduler.scheduleWithFixedDelay(clock3,
3, 20, TimeUnit.SECONDS);

        Runnable cancelTask = new Runnable() {

            public void run() {

                f1.cancel(true);

                f2.cancel(true);

                f3.cancel(true);

            }

        };

        scheduler.schedule(cancelTask, 120, TimeUnit.SECONDS);

    }

}

```

Recompile and run the program again and observe the result.

* Summary:

Here I summarize the key points you have learned today:

- Know what you can do with the high-level concurrency API.
- Understand the need of thread pool and how it works.
- Understand 3 types of
executors: `Executor`, `ExecutorService` **and** `ScheduledExecutorService`.

- Know how to create different kinds of thread pools via several factory methods of the `Executors` utility class.
- Understand how to execute value-returning tasks with `Callable` and `Future`.
- Know how to schedule tasks to execute after a given delay, or execute periodically after a fixed rate or delay.