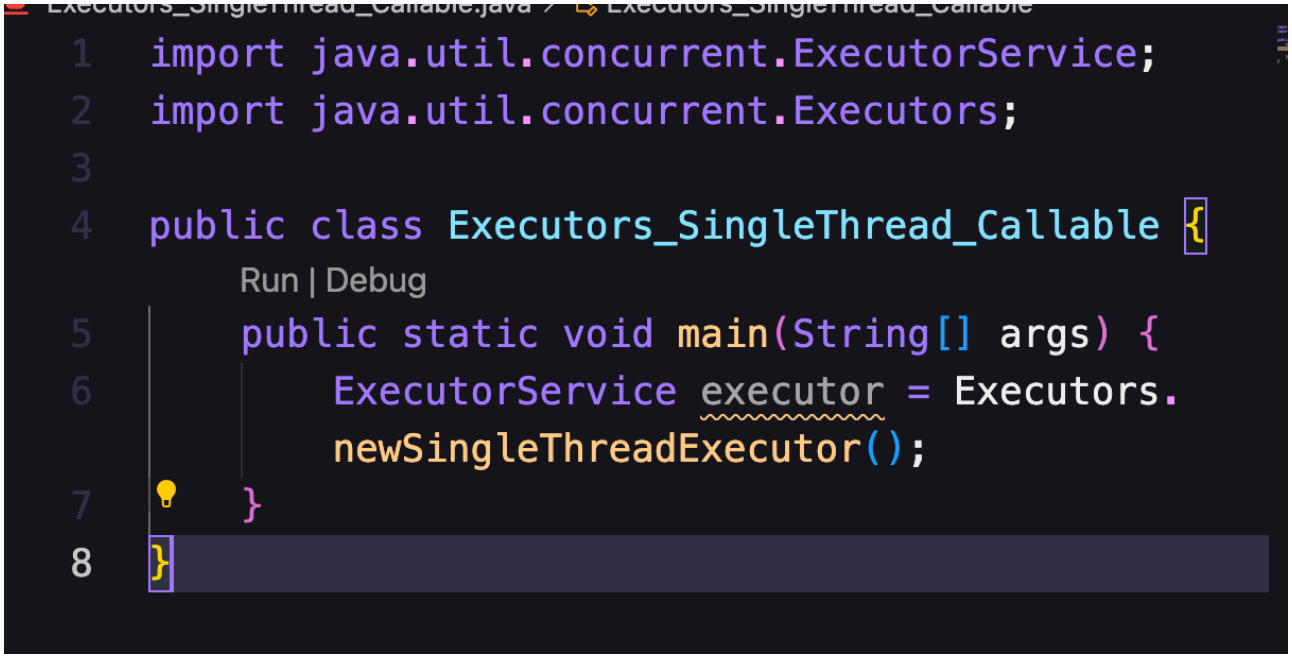
Programação_Concorrente-Studies/ Aula06: Executores:

- Como a classe Thread não é mais tão utilizada, atualmente se utilizam executores, e é sobre isso que iremos estudar agora.

- Criando um executor singleThread:



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Executors_SingleThread_Callable {
    Run | Debug
    public static void main(String[] args) {
        ExecutorService executor = Executors.
        newSingleThreadExecutor();
    }
}
```

- Dessa forma criamos um executor SingleThread, e para executarmos ele precisamos passar para ele uma Runnable da seguinte forma:

```
Run | Debug
public static void main(String[] args) {
    ExecutorService executor = Executors.
    newSingleThreadExecutor();
    executor.execute(new Tarefa());
}

public static class Tarefa implements
Runnable {
    @Override
    public void run() {
        System.out.println(x:"Tarefa
        executada");
    }
}

}
```

- Utilizando o método execute(new Tarefa) criamos uma nova Thread com o Runnable e logo ele executa o seu método run.
- Diferente de Thread o execute pode executar mais de uma vez uma tarefa da seguinte forma:

```
executor.execute(new Tarefa());
executor.execute(new Tarefa());
executor.execute(new Tarefa());
executor.execute(new Tarefa());
```

- Porém é de extrema importância que após criar um executor nós chamarmos o método executor.shutdown(), para dessa forma finalizar a execução da Thread finalizando o executor.

```
executor.execute(new Tarefa());
executor.shutdown();
}
```

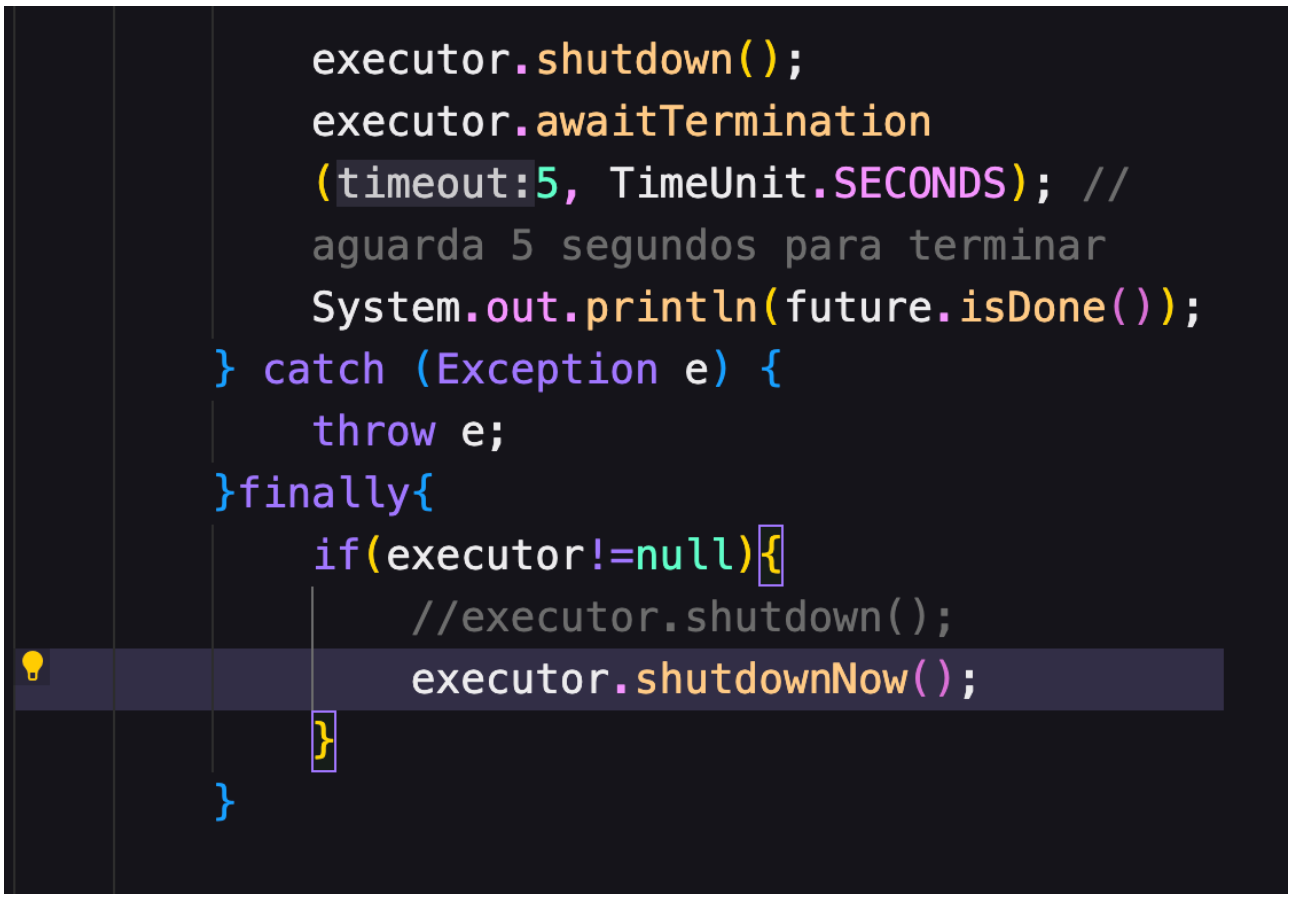
- Porém oq pode ocorrer é que quando utilizamos o shutdown, ele termine a execução antes de todas as Tarefas internas da Thread sejam realizadas, dessa forma para que isso não ocorra devemos utilizar o método awaitTermination();

```
executor.execute(new Tarefa());
executor.awaitTermination(timeout:5,
TimeUnit.SECONDS); // aguarda 5
segundos para terminar
executor.shutdown();
```

- Porém há uma forma correta de tratar esse await com o shutdown, para isso vamos necessitar de um bloco try_catch:

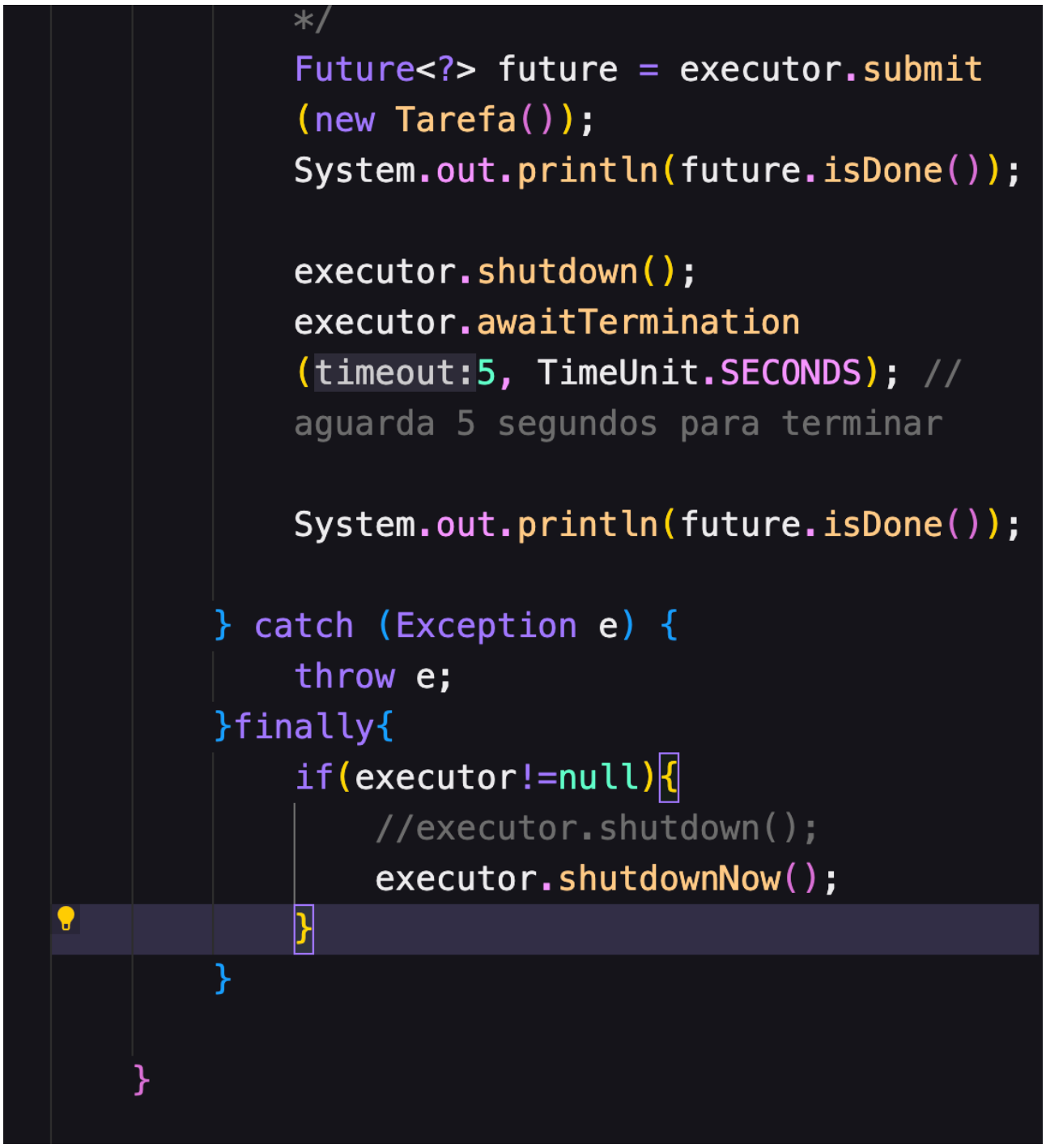
```
ExecutorService executor=null;
try {
    executor = Executors.
    newSingleThreadExecutor();
    executor.execute(new Tarefa());
    executor.awaitTermination
    (timeout:5, TimeUnit.SECONDS); //
    aguarda 5 segundos para terminar
} catch (Exception e) {
    throw e;
}finally{
    if(executor!=null){
        executor.shutdown();
    }
}
```

- Dessa forma garantimos o tratamento de erros e finalizamos o programa no momento correto.
- Para o melhor funcionamento e tratamento do nosso código é recomendado se utilizar o tratamento de fechamento de tempo com o awaitTermination() e o shutdown() dentro do bloco try e no bloco finally apenas utilizar o shutdownNow():



- Agora iremos ver sobre o método submit, que é uma alternativa mais interessante do que o método execute:

- Para o método submit() ele funciona como um submit, a diferença é que ele nós dá um retorno, ou seja eu consigo receber em uma variável um controle da execução do meu executor, dessa forma terei métodos a mais de controle.
- O objeto em questão que recebemos é um objeto do tipo Future, que tem métodos como o isDone(), que me retorna se o meu executor terminou ou não de executar as suas tarefas, abaixo temos um exemplo de uso do submit com o future:



- Agora com o uso do future ele nós abre mais oportunidades de trabalho com executores, um exemplo é o uso de Callables ao invés de Runnables.
- Callables são um tipo parecido com Runnables, porém elas podem retornar valores para assim serem acessados de fora da Thread, dessa forma podemos fazer com que nossa Thread trabalhe com algo e no final nós retorne esse valor, aqui abaixo temos um exemplo do uso do Callable com o nosso future:

```
    public static class Tarefa implements
    Callable<String> {

        @Override
        public String call() throws Exception {
            String name = Thread.currentThread
            ().getName();
            int nextInt = new Random().nextInt
            (bound:1000);
            return "Hello World! " + name + " -
            " + nextInt;
        }
    }

}
```

```
        Future<?> future = executor.submit
        (new Tarefa());
        System.out.println(future.get());
```

- Aqui temos um Callable que retorna uma String e acessamos e recebemos essa String por meio do método do objeto Future chamado future.get();
- É importante ressaltar que quando usamos o .get() no nosso future, ele aguarda até que a tarefa finalize, portanto quando usamos o .get() para obter o valor do Callable, o shutdown e o awaitTermination já não tem mais tanto sentido pois o get já espera que a tarefa finalize para retornar o valor, porém o shutdownNow() que aplicamos no finally do programa ainda é necessário para acabar com a Thread após toda execução do programa.

```
System.out.println(future.isDone());

System.out.println(future.get());

/*
executor.shutdown();
executor.awaitTermination(5,
TimeUnit.SECONDS); // aguarda 5
segundos para terminar
*/

System.out.println(future.isDone());
```

- Aqui temos um exemplo em que o primeiro isDone() retornará quase sempre false e o segundo quase sempre true, a n ser que a tarefa seja executada bem rápido antes mesmo do primeiro isDone() ser executado, tendo assim uma chance de ambos retornarem true.

- Uso do timeout no get, uma alternativa para definir o tempo máximo de espera que o get vai aguardar a execução total da tarefa:

```
System.out.println(future.get
(timeout:1, TimeUnit.SECONDS));
```

- Nesse caso o nosso get aguarda apenas 1 segundo a execução da tarefa e caso essa tarefa demore mais que isso, o nosso get() lançará uma exeção da seguinte forma:

```java
    public static class Tarefa implements
    Callable<String> {

        @Override
        public String call() throws Exception {
            Thread.sleep(millis:1000);
            String name = Thread.currentThread
            ().getName();
            int nextInt = new Random().nextInt
            (bound:1000);
            return "Hello World! " + name + " -
            " + nextInt;
        }
    }

}
```

- Adicionamos um sleep para que a exceção ocorra:

```
false
Exception in thread "main" java.util.concurrent.TimeoutException
        at java.util.concurrent.FutureTask.get(FutureTask.java:205)
        at Executors_SingleThread_Callable.main(Executors_SingleThread_Callable.java:26)
marcus_cs_pereira@MacBook-Pro-de-Marcus Aula06 %
```

- Dessa forma o get() aguardou até no máximo 1 segundo até que a tarefa terminasse, porém como a tarefa ultrapassou esse tempo limite ele lançou uma exceção.
- O get on timeout será utilizado geralmente em casos de produção de sistemas mais complexos, em que não sabemos exatamente o quão demorada essa tarefa pode ser, acabando por atrasar o sistema aguardando a finalização de uma tarefa extremamente demorada.

Criando executores MultiThread:

- A primeira forma de se criar um executor é utilizando o ExecutorService e instanciar o nosso objeto executor com o método newFixedThreadPool(valor);

- Esse método faz com que eu crie um executor que vai utilizar 4 threads e ele pode escolher qual delas ele vai utilizar para realizar alguma tarefa.

```java
public class Executores_MultiThread {
    Run | Debug
    public static void main(String[] args) throws
    InterruptedException, ExecutionException {
        ExecutorService executor = Executors.
        newFixedThreadPool(nThreads:4);
        Future<String> future = executor.submit(new
        Tarefa());
        System.out.println(future.get());
    }

    public static class Tarefa implements
    Callable<String> {

        @Override
        public String call() throws Exception {
            Thread.sleep(millis:1000);
            String name = Thread.currentThread().getName
            ();
            int nextInt = new Random().nextInt
            (bound:1000);
            return "Hello World! " + name + " - " +
            nextInt;
        }
    }
```

- Fazendo da seguinte forma conseguimos obter um retorno da nossa execução através do callable que retorna um objeto future, e agora vamos analisar esse retorno:

```
utores_MultiThread
Hello World! pool-1-thread-1 - 763
```

- Dessa forma podemos ver qual das Threads da pool o nosso executor usou para realizar essa tarefa, mas oq acontece se eu colocar mais tarefas para esse mesmo executor trabalhar

```
Run | Debug
9   public static void main(String[] args) throws
    InterruptedException, ExecutionException {
10    ExecutorService executor = Executors.
      newFixedThreadPool(nThreads:4);
11    Future<String> f1 = executor.submit(new Tarefa
      ());
12    Future<String> f2 = executor.submit(new Tarefa
      ());
13    Future<String> f3 = executor.submit(new Tarefa
      ());
14    System.out.println(f1.get());
15    System.out.println(f2.get());
16    System.out.println(f3.get());
17    executor.shutdown();
```

- Dessa forma estamos direcionando mais tarefas ao nosso executor

```
Hello World! pool-1-thread-1 - 989
Hello World! pool-1-thread-2 - 39
Hello World! pool-1-thread-3 - 469
marcus_cs_pereira@MBP-de-Marcus Aula06 %
```

- E percebemos que o retorno que temos é de cada uma dessas tarefas sendo executada em Threads diferentes da nossa pool.
- A manipulação do pool limita a quantidade de Threads que o nosso executor vai usar para trabalhar

- Agora vamos utilizar o método .newCachedThreadPool();

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

- A diferença do cached pro fixed: o fixed trabalha apenas com o número de Thread que limitamos a ele, já o cached, sempre que o executor necessitar de uma Thread para executar uma tarefa ele cria, porém caso existam threads que ele criou para executar uma tarefa anterior e elas já tiverem acabado essa tarefa, o executor usa essa thread “antiga” para executar a nova tarefa.

- Agora vamos aprender a executar várias tarefas utilizando o invokeAll:

```
18        executor = Executors.newCachedThreadPool();
19        Tarefa t1 = new Tarefa();
20        Tarefa t2 = new Tarefa();
21        Tarefa t3 = new Tarefa();
22        Tarefa t4 = new Tarefa();
23
24        List<Future<String>> list = executor.
          invokeAll(Arrays.asList(t1, t2, t3, t4));
25
26        for (Future<String> future : list) {
27          System.out.println(future.get());
28        }
```

- Primeiro criamos diversas tarefas, e por meio do metodo invokeAll chamamos elas para serem executadas, como tempo um cachedThreadPool, ele vai criar o número de Threads necessárias para executar as tarefas, nesse caso 4.
- É importante ressaltar que nosso método retorna o Future de cada uma das nossas tarefas, por isso necessitamos recebe-lós como uma List de Futures.

```
List<Tarefa> tarefas = new ArrayList<>();
for (int i = 0; i < 10; i++) {
  tarefas.add(new Tarefa());
}

List<Future<String>> list = executor.
invokeAll(tarefas);
```

- Outra forma de passar e armazenar as tarefas.

- Agora vamos utilizar o invokeAny()

```
String string = executor.invokeAny(tarefas);
System.out.println(string);
```

- O invokeAny tambem recebe a lista de tarefas porém ele retorna o resultado de apena uma, geralmente o resultado da tarefa que foi realizada mais rapidamente ou da primeria tarefa da lista.

Criando Executores Scheduled:

- Executores Scheduled são executores que realizam tarefas com parametros de agendamento:

```
14  public class Executores_Scheduled {
        Run | Debug
15      public static void main(String[] args) throws
        Exception {
16      ScheduledExecutorService executor=null;
17
18        try {
19          executor = Executors.newScheduledThreadPool
            (corePoolSize:3);
20          ScheduledFuture<String> future = executor.
            schedule(new Tarefa(), delay:2, TimeUnit.
            SECONDS);
21          System.out.println(future.get());
22
23          executor.shutdown();
24        } catch (Exception e) {
25          throw e;
26        }finally{
27          executor.shutdownNow();
28        }
```

- Primeiramente criamos o nosso executor e determinamos sua pool size, após podemos utilizar o método .schedule(passando uma tarefa, o delay para realizar essa tarefa, e por ultimo a unidade de tempo desse delay)
- Podemos também chamar esse método de schedule com o método run na nossa Runnable ao invés do método Callable

- Agora vamos ver como Executamos as tarefas a cada X segundos:

```
executor.scheduleAtFixedRate(new Tarefa(),
initialDelay:0, period:1, TimeUnit.SECONDS);
```

- Algumas coisas são importantes nesse método, primeiro definimos a tarefa que estamos chamando nesse caso tarefa do tipo run, logo após declaramos o delay para ele executar pela primeira vez a tarefa, depois declaramos o intervalo de tempo entre a execução de cada tarefa, e por ultimo determinamos a unidade de tempo.
- Além disso é necessário remover o executor.shutdown do bloco try para que ele n pare essa execução loopada.
- Se atente ao uso do Sleep em Threads usando esse método de agendar e executar tarefas, caso o sleep demora menos que o tempo colocado no period do scheduleFixedRate(), ele

continua com o tempo do period determinado pelo programados, porém quando colocamos um tempo de sleep maior ele sobrepõe o period, e passa a ser o tempo de execução entre cada tarefa.

- Agora vamos ver a execução de tarefas com intervalo de tempo defenido entre elas:

```
executor.scheduleWithFixedDelay(new Tarefa
(), initialDelay:0, delay:1, TimeUnit.
SECONDS);
```

- A unica diferença é que agora determinamos o delay entre cada execução de Tarefa, nesse caso agora ele espera a tarefa ser terminada pela Thread e ao fim da tarefa ele conta esse tempo, então n importa se o Thread.Sleep seja maior ou menor que o delay, ele vai esperar esse tempo entre cada execução, obviamente somado ao tempo do sleep.

#Concorrent