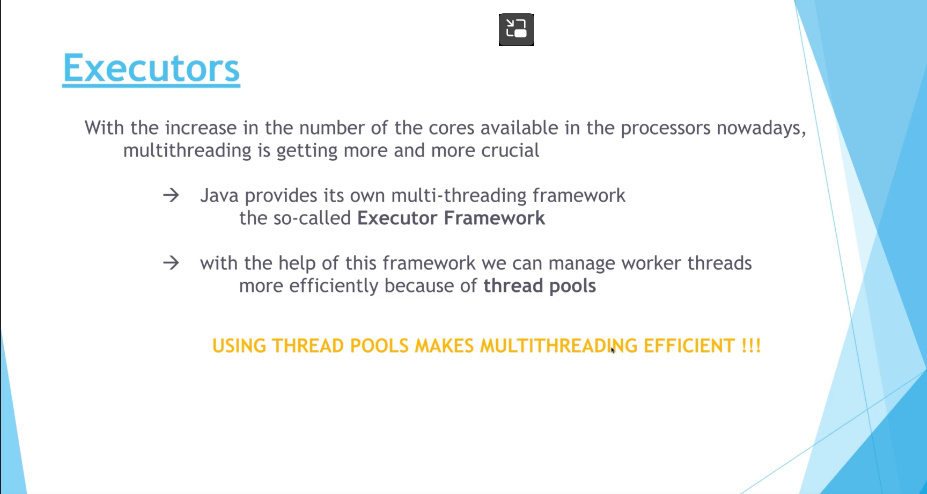
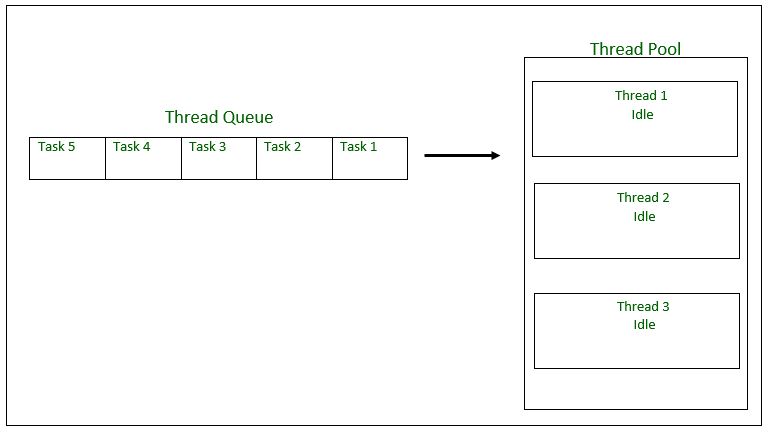
**Executor Framework**



* Java are propriul sau framework pentru multithreading, numit **Executor Framework**
* Anume acest thread ne permite sa administram threadurile cu **thread pools**
* **Thread Pool** – design pattern pentru a rula mai multe threaduri concurent, reprezentand un grup de worker threaduri care sunt executatet de mai multe ori pentru diferite taskuri.
* **Intr**-un Thread Pool, un numar fix de threaduri e creat
* Ideea e ca, de ex, un web server trebuie sa preia multe requesturi de la useri, si o solutie buna e sa cream cate un thread pentru fiecare request. Dar, problema e ca crearea si distrugerea threadului ar consuma mai multe resurse decat prelucrarea requestului
* Daca JVM creaza mult prea multe threaduri, sistemul poate ramane fara memorie
* Iata de ce, e necesar de a limita numarul de threaduri ce sunt create
* Thread Pool creaza initial cateva threaduri, fara a fi nevoie de un request, si foloseste aceste threaduri deja create in loc de a tot crea unele noi mereu. Asa, se evita crearea threadurilor, alocarea la cache, a stack, distruerea iar si iar. Asta se face doar odata pentru fiecare thread, si cand un thread executa alt task, cache si stack al sau pur si simplu e curatat si atat, nu realocat.
* Asa cand requestul vine, nu trebuie sa se creeze un nou thread, dar se va folosi cel deja existent
* Apoi, aceste threaduri odata ce termina executia, nu vor fi sterse, ci refolosite
* Thread pool foloseste Quee:





* 

**Executors**

* **SingleThreadExecutor** – are doar un singur thread care este creat pentru fiecare task. Fiecare task va folosi acest thread, nu va tot fi creat unul nou. Totusi, daca threadul termina executia cumva si nu e folosit ceva timp, el e sters si se va crea altul
* **FixedThreadPool(n)** – Creaza un thread pool cu n threads. Daca sunt mai multe taskuri ca threaduri, aceste taskuri sunt stocate intr-o LinkedBlockingQuee
* **CachedThreadPool** – nr. de threaduri nu este limitat. Daca toate threadurile sunt ocupate, si un nou task vine, pool va crea un nou thrread. Daca threadul e inactiv pentru 60 de secunde, e sters. Asta e folosit pentru short parallel tasks. Trebuie sa ne asiuram ca cachedThreadPool nu va crea prea multe threaduri
* **ScheduledExecutor –** este folosit pentru a executa o operatie la intervale rehulate de timp.

**Code**

* Pentru a crea un Executor, facem asa:

**ExecutorService executor = Executors.newTIPEXECUTOR;**

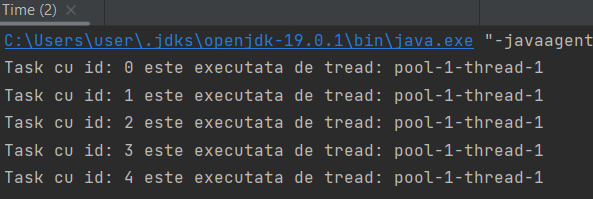
apoi pentru a da la executie un task, facem asa:

**executor.execute(new Object de tip Runnable)**

* Aici facem un SingleThreadExecutor

**ExecutorService service = Executors.newSingleThreadExecutor();**

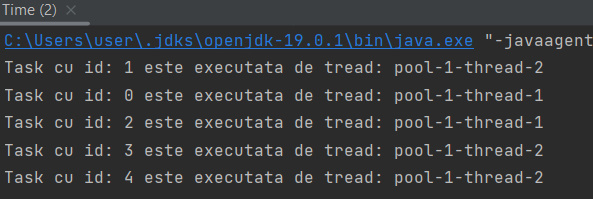
class Time {  
 public static void main(String[] args) throws InterruptedException {  
 ExecutorService executorService = Executors.*newSingleThreadExecutor*();  
 for(int i = 0;i<5;i++){  
 executorService.execute(new Run(i));  
 }  
 }  
}  
class Run implements Runnable{  
 public int id;  
  
 public Run(int id){  
 this.id = id;  
 }  
 @Override  
 public void run() {  
 System.*out*.println("Task cu id: "+id+" este executata de thread: "+Thread.*currentThread*().getName());  
 try {  
 Thread.*sleep*(1500);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}



Vedem ca a fost folosit acelasi thread.

* Acum, daca fom face un asa ThreadPool cu **FixedT**h**readPool**:

ExecutorService executorService = Executors.*newFixedThreadPool*(2);





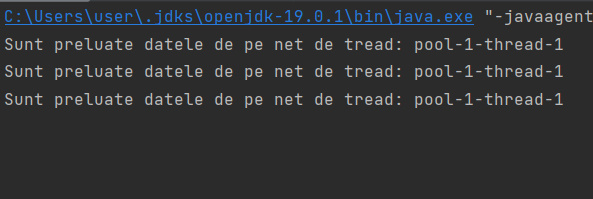
Asa am creat un ThreadPool creat din 2 Threaduri. Vedem ca intai au fost executate 2 taskuri de catre 2 threaduri, apoi dupa ce s-au executat au fost executate inca 2 taskuri tot de aceleasi threaduri si asa pana la final.

De fapt, cand un thread din ThreadPool termin cu un task, el preia un alt task si tot asa.

* Aici cream un **ScheduledExecutor**
* **ScheduledExecutorService service = new Executors.newScheduledThreadPool(Nr de threaduri)**
* .schedule(object, time, TimeUnit.MILISECONDS) – executia taskului se va face o sinura data peste **time** milisecunde
* .scheduleAtFixedRate(obj,timeAfter,timeCicle,TimeUnit)

timeAfter – peste atata timp va incepe executia la Thread

* timeCicle – odata in atata timp taskul este executat din nou
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
   ScheduledExecutorService service = Executors.*newScheduledThreadPool*(1);  
   //service.schedule(new Run(),1000, TimeUnit.MILLISECONDS);  
   service.scheduleAtFixedRate(new Run(),2000,5000,TimeUnit.*MILLISECONDS*);  
   }  
  }  
  class Run implements Runnable{  
    
   @Override  
   public void run() {  
   System.*out*.println("Sunt preluate datele de pe net de tread: "+Thread.*currentThread*().getName());  
   }  
  }





Vedem ca taskul a fost executat in fiecare 5 secunde iar si iar

**Stopping Executors**

* Indiferent de ce tip de executor folosim, apare problema ca desi s-au executat toate taskurile, el nu a fost oprit si continua sa tina proramul deschis
* .**shutDown()** – pur si simplu i spune la Executor ca nu mai poate lua taskuri noi, dar va termina executia celora pe care deja le-a inceput
* .**shutDownNow**() – i spune la Executor ca nu mai poate lua taskuri si incearca sa le intrerupa pe cele care deja ruleaza. Atentie ca poate arunca exceptii si sa nu aiba efect.

Iata de ce, ne asiuram sa avem asta in catch

@Override  
public void run() {  
 System.*out*.println("Task cu id: "+id+" este executata de tread: "+Thread.*currentThread*().getName());  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
}



**Runnable vs Callable interface**

* Runnable si Callable sunt foarte asemanatoare, dar diferenta cea mai mare este ca Runnable nu poate returna vreo valoare, dar Callable poate returna o valoare de la thread
* Nu putem crea direct un thread cu callable!
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
    
   Thread thread = new Thread(new Run(1));  
   }  
  }  
  class Run implements Callable<String>{  
    
   public int id;  
    
   public Run(int id) {  
   this.id = id;  
   }  
    
   @Override  
   public String call() throws Exception {  
   Thread.*sleep*(1000);  
   return "Thread with id "+id+" has finished";  
   }  
  }

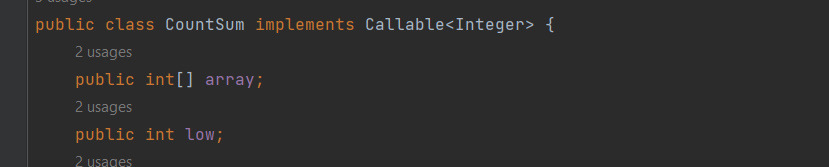


* Pentru a putea face asta, avem nevoie sa cream un obiect de tip **FutureTask**(clasa data implementeaza si Runnable si Future) si sa trimitem obiectul de tip callable lui, apoi cu el cream threadul:
* FutureTask futureTask = new FutureTask(new Run(1));  
  Thread thread = new Thread(futureTask);  
  thread.start();  
  System.*out*.println(futureTask.get());

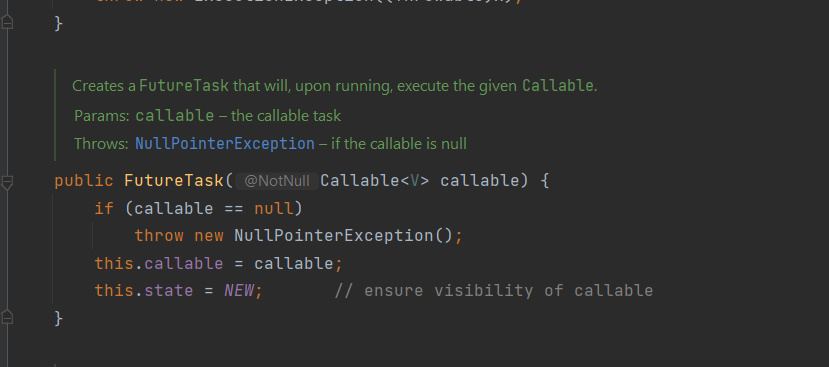
**El va pastra valoarea returnata de t**hread, si o obtinem cu get()

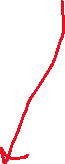
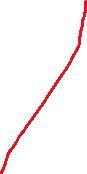
Clasa Thread are si un asa constructor **Thread(FutureTask);**

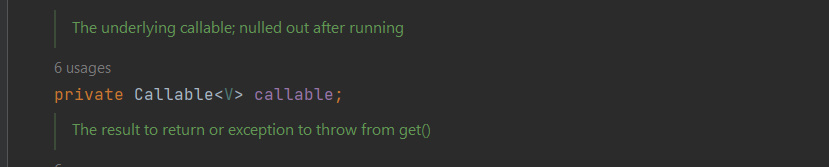
* .get() returneaza anume un obiect de tip setat de noi cand am implementant Callable<>
* FutureTask are in constructor un parametru de tip Callable<>, si in acest generic daca de ex am pus Integer, pai si FutureTask va pune ca generic al sau Integer, fara ca noi sa scriem asta cand cream obiectul de tip FutureTask.



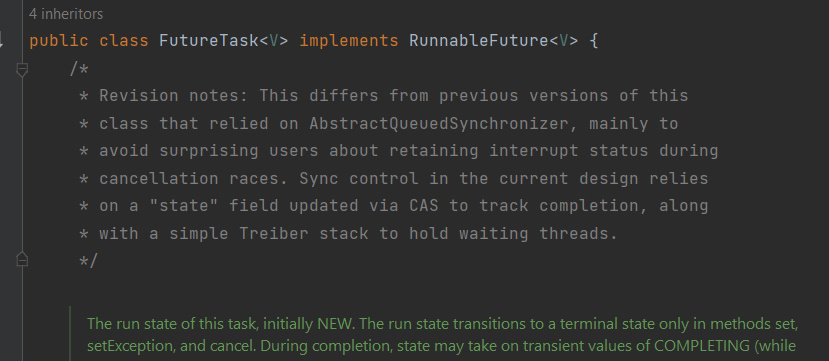


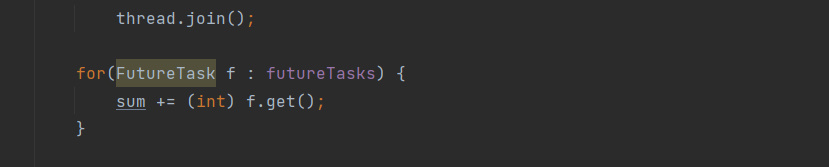








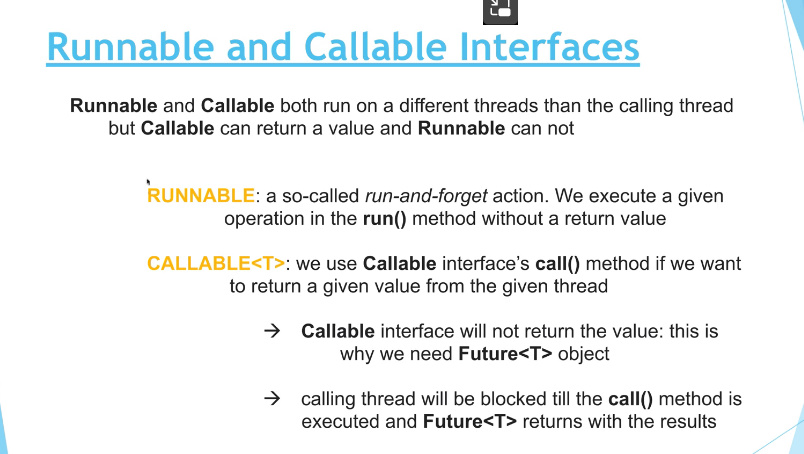
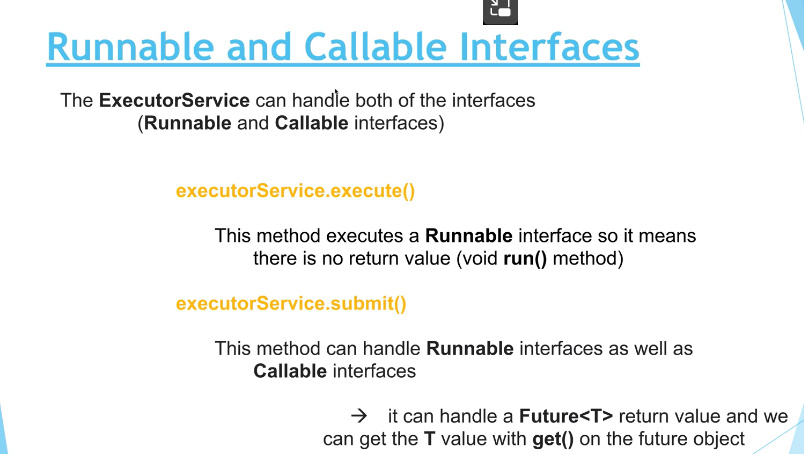






* **La crearea obiectului de tip FutureTask nu ne trebuie generic!Putem sa il punem, dar nu are sens.**
* **Clasa Thread nu accepta obiecte ce implementeaza Callable pentru a nu implementa Generics, asa cum Generics e consumator de tip si resurse, si mai bine lasa pe mana la alt obiect setarea valorii si generics**
* Folosim **(int) f.get()** deoarece la runtime nu se stie care e generics ce va fi trimis obiectului de tip FutureTask de catre cel ce implementeaza Callable
* **Dar,clasele care implementeaza ExecutorService au direct o metoda ce returneaza Future a t**hread, si aia e **submit()**

ExecutorService service = Executors.*newFixedThreadPool*(2);  
for(int i = 0;i<10;i++){  
 Future<String> future = service.submit(new Run(i));  
 System.*out*.println(future.get());  
}

* 
* Dar Callable interface nu returneaza valoarea explicit, ci cu un Future<T> de asta avem nevoie de un Future<T>
* 
* ExecutorService interface are 2 metode de executare a threadului:
* **.execute()** - poate executa doar obiecte de tip Runnable
* **.submit()** – poate executa si obiecte de tip Runnable, si Callable si returneaza valoarea in Future<T> object.