**Fork-Join Framework**

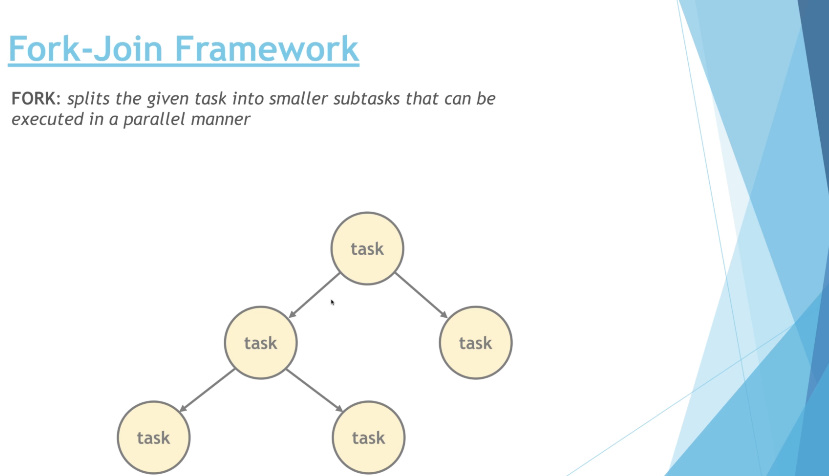
* **Fork-Join Framework** este o implementare a interfetei ExecutorService pentru parallel execution
* Nu e asa de usor de implementat parallel execution, si de asta acest framework ne ajuta
* El ne ajuta sa facem un algoritm paralel
* Nu trebuie sa ne tot batem capul cu locks cu el
* Este un mecanism divide and conquer
* Taskurile largi pot fi divizate in mai mici si combinam la urma solutiile pentru a obtine o solutie finala
* **Subtaskurile trebuie sa fie independente ca sa fie executate paralel**
* **Conceptul de baza al acestui Framework este de a diviza un task in mici taskuri pana cand aceste subtaskuri sunt destul de simple pentru a fi rezolvate fara a mai fi divizate**
* **Unele probleme care necesita foarte multe operatii sau care contin multe date, pot fi rezolvate prin a diviza problema in taskuri, si odata divizata, fiecare task sa rezolve bucata lui de problema in mod secvential, nu paralel!**
* Daca folosim parallelization, asta nu inseamna ca nu vom folosi si abordarea secventiala. Abordarea paralela e doar pentru a crea subtaskuri, adica de a diviza o problema in subprobleme, mai apoi deja subproblemele adesea si se rezolva cu algoritmi secventiali, dar evident mai rapid, asa cum fiecare task are doar o bucata din problema.

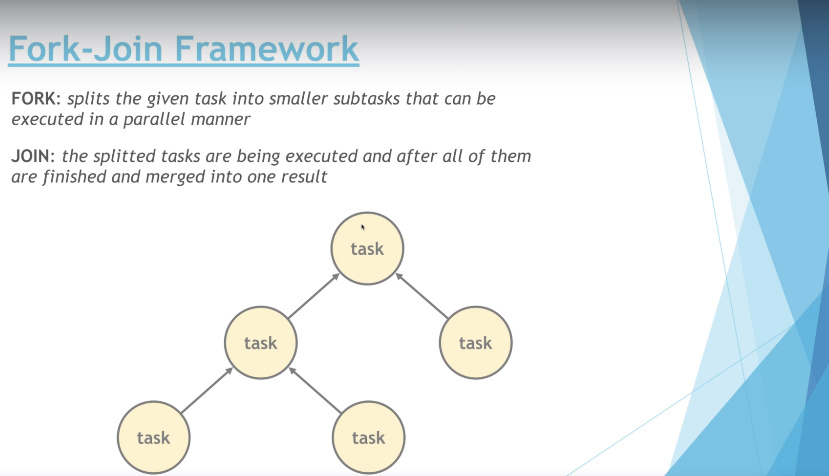
**Classes and Interfaces**

* **RecursiveTask<T>(clasa abstracta)** -un task cu return. returneaza un generic type T.
* Toate taskurile pe care vrem sa le executam in paralel si sa returneze ceva trebuie sa fie subclase ale RecursiveTask
* Trebuei suprascrisa metode **compute()** care returneaza solutia problemei
* **RecursiveAction(clasa abstracta)** – un task, dar fara return
* **ForkJoinPool** - este un thread pool pentru a executa fork-join tasks
* work-stealing – Daca un thread nu mai are taskuri de executat, el va fura taskuri de la threadurile mai ocupate.Taskurile sunt distribuite catre toate threadurile din thread pool
* **Un task nu este echivalent cu un thread!**
* Deci, cand cream de ex un RecursiveTask sau un RecursiveAction, ei bine aceste taskuri nu vor fi niste threaduri noi, ci vor rula in threaduri deja create, ca in executors
* **Task** – este o simpla multime de instructiuni ce ruleaza asyncronizat,
* ForkJoinPool creaza un numbar fix de threaduri, deobicei egal cu nr de cores. Anume acest threaduri executa toate taskurile
* Folosirea a prea multe threaduri poate fi o problema de performanta, dar folosirea a multe taskuri nu este, asa cum ele sunt executate de un nr. constant de threaduri
* Anume asa atingem **Load Balacing**. Adica, toate threadurile vor executa taskurile in timp egal aproximativ, asa cum nici unul nu va termina si nu va sta, dar va prelua de la altele taskuri

**Fork and Join operations**

* **Fork** – a diviza un task in subtaksuri mai mici care pot fi executate in mod paralel



* **Join** – subtaskurile ce au rezultat din task de baza sunt executate si dupa ce toate sunt executate, solutiile lor sunt imprenate in una singura

**Methods**

* **fork()** – executa in mod **asynchronizat** taskurile in pool. Putem folosi RecursiveTask<T> sau RecusrsiveAction ca sa o apelam. O folosim pentru a starta un task
* **join**() – returneaza rezultatul calcului cand e gata in mod **asynchronizat si face ca un task sa astepte pana altul termina**
* **invoke**(action) – executa taskul + asteapta + return rezultatul de indata ce termina, deci in mod **synchronizat. Metoda este pentru a incepe rularea taskurilor in thread pool**

**Exemplu RecursiveAction**

public class SimpleRecursiveAction extends RecursiveAction {  
  
 private int simulatedWork;  
  
 public SimpleRecursiveAction(int simulatedWork) {  
 this.simulatedWork = simulatedWork;  
 }  
  
 @Override  
 protected void compute() {  
 if(simulatedWork > 100){  
 System.*out*.println("Parallel execution and split the task");  
 SimpleRecursiveAction action1 = new SimpleRecursiveAction(simulatedWork/2);  
 SimpleRecursiveAction action2 = new SimpleRecursiveAction(simulatedWork/2);  
  
 action1.fork();  
 action2.fork();  
  
 action1.join();  
 action2.join();  
 }  
 }  
}

.join() – adesea e folosit cu fork()

Dar, asta poate fi inlocuit:

action1.fork();  
 action2.fork();  
  
 action1.join();  
 action2.join();

cu asta:

*invokeAll*(action1,action2);

Metoda asta tot va astepta ca aceste taskuri sa termine executia

Putem folosi apoi join() pentru a returna, nu e nicio problema.

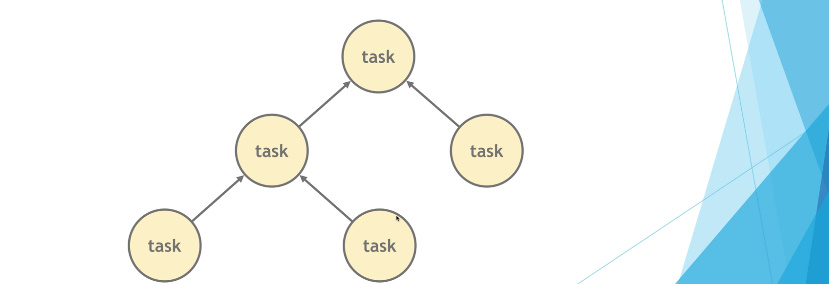
* Main:
* public class MyApp {  
   public static void main(String[] args){  
   SimpleRecursiveAction action = new SimpleRecursiveAction(600);  
   ForkJoinPool forkJoinPool = new ForkJoinPool();  
   forkJoinPool.invoke(action);  
   }  
  }
* ForkJoinPool forkJoinPool = new ForkJoinPool();

Constructori:

**ForkJoinPool(nr of threads)**

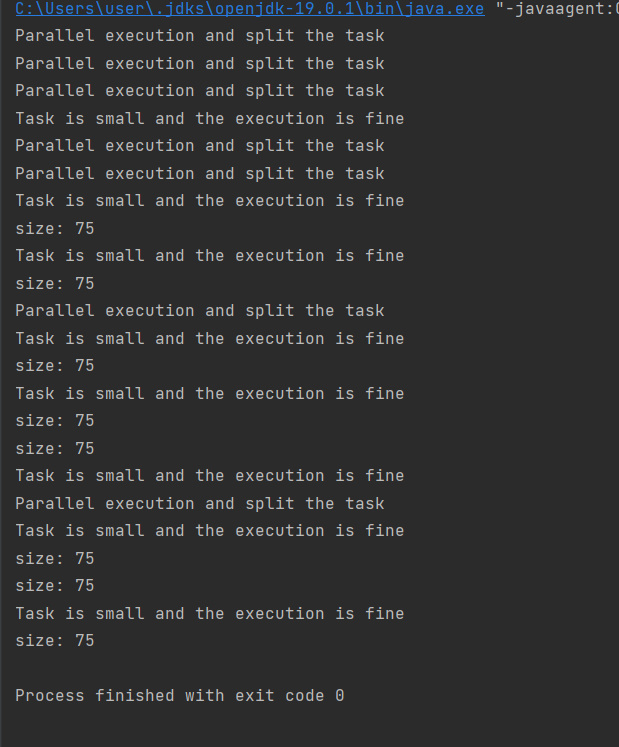
**ForkJoinPool()** – creaza atatea threaduri cate cores exista intr-un thread pool unde vor fi executate taskurile

* SimpleRecursiveAction action = new SimpleRecursiveAction(600);





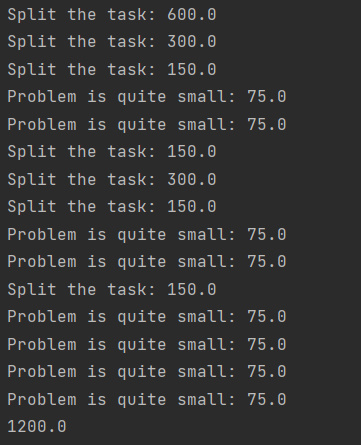
* Threadurile sunt din thread pool sunt executate in paralel



**Exemplu RecursiveTask<T>**

public class SimpleRecursiveTask extends RecursiveTask<Double> {  
  
 private Double num;  
  
 public SimpleRecursiveTask(Double num) {  
 this.num = num;  
 }  
  
 @Override  
 protected Double compute() {  
 if(num > 100){  
 System.*out*.println("Split the task: "+num);  
 SimpleRecursiveTask task1 = new SimpleRecursiveTask(num/2);  
 SimpleRecursiveTask task2 = new SimpleRecursiveTask(num/2);  
  
 task1.fork();  
 task2.fork();  
  
 Double solution = 0.0;  
 solution += task1.join();  
 solution += task2.join();  
  
 return solution;  
 }  
 else{  
 System.*out*.println("Problem is quite small: "+num);  
 return 2\*num;  
 }  
 }  
}

public class MyApp {  
 public static void main(String[] args){  
 ForkJoinPool pool = new ForkJoinPool();  
 SimpleRecursiveTask simpleRecursiveTask = new SimpleRecursiveTask(10000.0);  
 System.*out*.println(pool.invoke(simpleRecursiveTask));  
 }

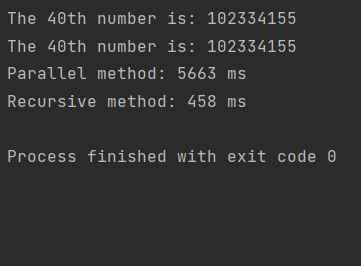


* .**join**() – se va asigura ca taskul curent va astepta pana cel invocat termina si va returna valoarea taskului
* .**invoke(task)** – ForkJoinPool executa taskul si returneaza valoarea lui

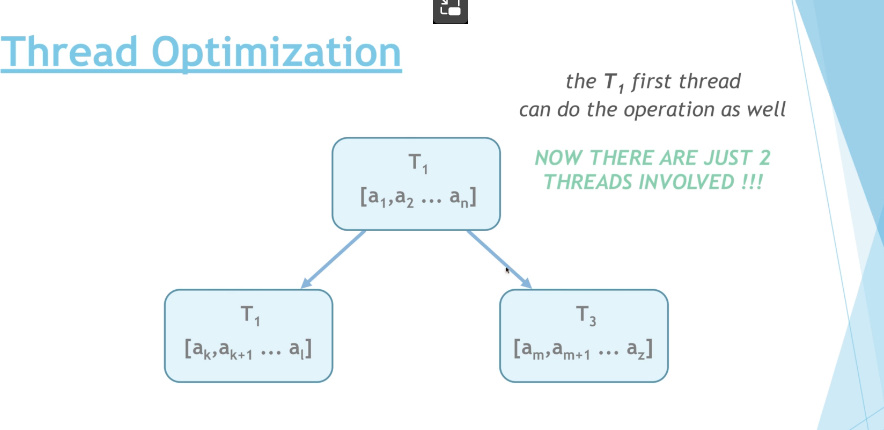
**Thread optimization**

* Problema lui Fibbonaci rezolvata cu parallel execution este mult mai lenta chiar si decat cea utilizata cu recursia:
* public class Fibonacci extends RecursiveTask<Integer> {  
   private final int nth;  
    
   public Fibonacci(int nth) {  
   this.nth = nth;  
   }  
    
   @Override  
   protected Integer compute() {  
   int sum = 0;  
   if(nth > 1){  
   Fibonacci f1 = new Fibonacci(nth-1);  
   Fibonacci f2 = new Fibonacci(nth-2);  
    
   f1.fork();  
   f2.fork();  
   sum = f1.join() + f2.join();  
   }  
   else if(nth == 1){  
   return 1;  
   }  
   else if(nth==0){  
   return 0;  
   }  
    
   return sum;  
   }  
  }

public class MyApp {  
 public static void main(String[] args){  
 ForkJoinPool forkJoinPool = new ForkJoinPool();  
 long count1 = 0;  
 long count2 = 0;  
 int nr = 40;  
  
 count1 = System.*currentTimeMillis*();  
 Fibonacci f = new Fibonacci(nr);  
  
  
 System.*out*.printf("The %dth number is: %d",nr,f.invoke());  
 f.join();  
 count1 = System.*currentTimeMillis*() - count1;  
 System.*out*.println();  
  
 count2 = System.*currentTimeMillis*();  
 System.*out*.printf("The %dth number is: %d",nr,new OldFib().count(nr));  
 count2 = System.*currentTimeMillis*() - count2;  
  
 System.*out*.println("\nParallel method: "+count1+" ms");  
 System.*out*.println("Recursive method: "+count2+" ms");  
  
 }  
}  
  
class OldFib{  
 public int count(int nr){  
 if(nr <= 1)  
 return nr;  
 return count(nr-1) + count(nr - 2);  
 }  
}

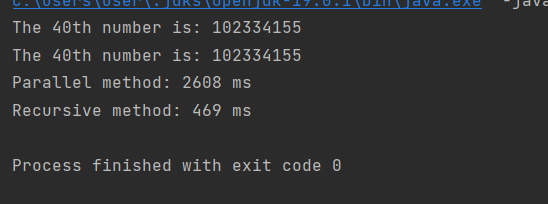


* Ideea e ca se creaza foarte multe taskuri ce vor rula in diferite threaduri, dar am putea executa direct un task tot in acelasi thread.
* Primul task, care e in thread1, oricum va trebui sa astepte pana celelalte termina, deci si el ar putea face ceva, ca sa mai micsoram din threaduri:



@Override  
protected Integer compute() {  
 int sum = 0;  
 if(nth > 1){  
 Fibonacci f1 = new Fibonacci(nth-1);  
 Fibonacci f2 = new Fibonacci(nth-2);  
  
 // f1.fork();  
 f2.fork();  
 sum = f1.compute() + f2.join();  
 }  
 else if(nth == 1){  
 return 1;  
 }  
 else if(nth==0){  
 return 0;  
 }  
  
 return sum;  
}

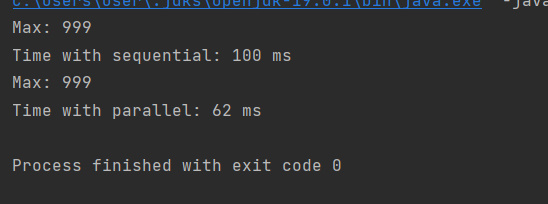
f1 nu va fi un task separat si la sigur nu va rula intr-un thread separat, dar va rula exact in acelasi thread care a executat metoda, si asa vom economisi in taskuri si threadurile nu vor comunica atata intre ele. Deci, va rula taskul f2 si apoi dupa ce el va termina, taskul curent va executa f1 ce va fi in acelasi thread, spre deosebire de f2 care a rulat in alte threaduri.



**Max Number Problem(vezi in project)**

* In cazul problemei date, metoda parallela e mai buna doar daca avem un numar enorm de date.
* Totusi, mai si vedem ca depinde mult de cate elemente se iau per array per task:

if(highIndex - lowIndex <30000){



Dar in cazul asta:

if(highIndex - lowIndex <3000){

