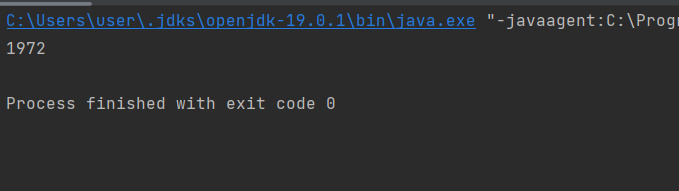
**Collections**

* Cea mai mare parte din Data Structure din Collections framework nu sunt thread safe
* Thread safe sunt:
* vector
* stack
* hashTable(nu e prea bine sincronizat)
* Anume asta si e problema cu Collections framework, nu e facut pentru multithreading:
* class Time {  
   public static void main(String[] args) throws InterruptedException, ExecutionException {  
    
   List<Integer> integerList = new ArrayList<>();  
    
   Thread thread1 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   for(int i = 0;i<1000;i++)  
   integerList.add(i);  
   }  
   });  
   Thread thread2 = new Thread(new Runnable() {  
   @Override  
   public void run() {  
   for(int i = 0;i<1000;i++)  
   integerList.add(i);  
   }  
   });  
   thread1.start();  
   thread2.start();  
    
   thread1.join();  
   thread2.join();  
    
   System.*out*.println(integerList.size());  
   }  
  }

Totusi, size la lista adesea nu e 2000:



Cauze:

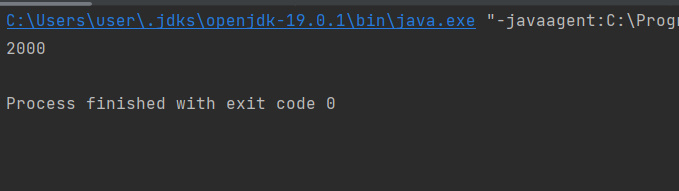
* ambele threaduri se poate intampla sa fie executate paralel pe diferite cores si fiecare are cache propriu si deci au propriile date despre lista, si ambele returneaza datele, dar se va lua doar la unul dintre ele, si evident ca oricare nu ar fi, in loc sa se mai puna 2 iteme, se va pune doar 1, asa cum ambele au adauat doar un item si nu stiu ca si celalalt a adauat, ca au doar cache vechi
* Un thread a luat datele despre lista in cache, dar nu a reusit sa returneze noua lista cu datele ei ,ca a fost oprit de alt thread, ce a returnat lista, si apoi si threadul oprit returneaza lista lui deja, si iese ca lista de la al 2 dispare

**Collections methods pentru a rezolva problema**

* Collections ofera cateva metode pentru a crea anumite structuri de date sincronizate(tread safe)
* **Collections.synchronizedCollection(new OriceCeExtindeCollection());**
* **Collections.synchronizedList(new OriceCeExtindeList());** - deci creaza o structura de date sincronizata, dar anume una ce extinde interfata List
* **Collections.synchronizedMap(new OriceCeExtindeMap());**
* **Collections.synchronizedSet(new OriceCeExtindeSet());**

Deci, daca vom crea ArrayList asa:

List<Integer> integerList = Collections.*synchronizedList*(new ArrayList<>());

Problema dispare, asa cum numai un thread o executa

* Aceste metode creaza un Monitor(Intrinsic) lock pentru aceste structuri de date, si asa fiecare thread il preia, apoi il elibereaza.
* Dar aceste metode anume si de asta nu sunt prea eficiente, din cauza la Intrinsic Lock
* Intrinsic Lock va face ca niciun thread sa nu poata accesa aceasta structura de date, si de ex un thread poate adaua ceva, dar alt thread vrea pur si simplu sa preia size la lista, dar nu va putea, caci primul thread deja a blocat intrinsic lock si al 2 thread trebuie sa astepte, chiar daca vrea sa execute cu totul alta metoda din structura de date
* SOLUTIA este de a folosi **Concurent Collections**

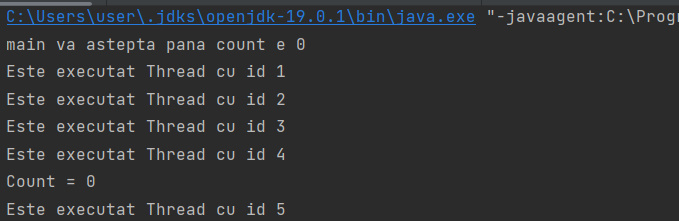
**CountDownLatch**

* task != thread
* Este folosit pentru a **sincroniza** **un thread(sau mai multe)** fortandul sa astepte pana un set de alte taskuri executa un set de operatii
* Putem oferi un initial count la un obiect de tip **CountDownLatch** si orice task care cheama **await()** la acest obiect va fi blocat pana count atinge 0
* **countDown()** – reduce count cand task **termina jobul** sau
* **CountDownLatch** este facut in asa fel incat count nu poate fi resetat!
* Daca vrem sa resetam count, folosim **CyclicBarrier** in schimb
* Task care cheama **countDown**() nu sunt blocate cand apeleaza metoda. Doar task care cheama await() este blocat pana count ajune la 0
* Un exemplu de folosire la **CountDownLatch** este cand divizam o problema in N taskuri independente si cream un CountDownLatch cu o valoare N
* Cand fiecare task termina , el apeleaza countDown() la latch. Taskurile care asteapta cheama await() la latch pentru a tine latch blocat pana cand termina
* CountDownLatch arata cate taskuri trebuie sa se execute, si odata ce numarul e ajuns, el termina. Poate face cu await() asa ca alte taskuri sa astepte pana altele termina executia lor
* CountDownLatch nu este cel care executa threadurile, dar pur si simplu are sarcina de a se asiura cu metoda await() ca un alt thread va astepta pana altele isi termina treaba si atat.
* **CountDownLatch(Nr de taskuri)** – constructorul ia nr. de taskuri ce trebuie executate;

class Time {  
 public static void main(String[] args) throws InterruptedException {  
 CountDownLatch countDownLatch = new CountDownLatch(5);  
  
 for(int i = 0;i<5;i++)  
 new Thread(new Run(countDownLatch,i+1)).start();  
 System.*out*.println("main va astepta pana count e 0");  
 countDownLatch.await();  
 System.*out*.println("Count = 0");  
 }  
}  
  
class Run implements Runnable{  
 CountDownLatch latch;  
 int id;  
  
 public Run(CountDownLatch latch, int id) {  
 this.latch = latch;  
 this.id = id;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(1000\*id);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("Este executat Thread cu id "+id);  
 latch.countDown();  
 }  
}

* Vedem ca toate threadurile au fost executate deodata
* metoda countDown() trebuie pusa la final, de altfel, se va intampla ca va fi scazut count prea devreme
* await() va face ca main thread sa stea in waitin state pana count = 0
* Daca count nu va ajune sa fie 0, thread main nu va iesi niciodata din wait
* Daca de ex. punem mai putine taskuri in constructor de cat vor fi, nu e nicio problema, pur si simplu await() va face ca main thread sa iasa din wait inainte ca celelalte threaduri sa termine

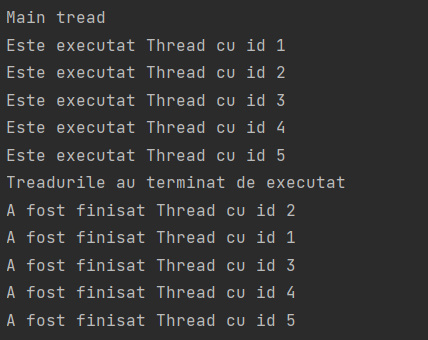
CountDownLatch countDownLatch = new CountDownLatch(4);



**CyclicBarrier**

* Este folosit in situatiile in care vrem sa cream un set de taskuri care sunt executate in mod concurent si sa se asiure ca ele se asteapta unele pe altele.
* CycleBarrier face ca mai multe taskuri sau threaduri sa se astepte unele pe altele pana ele toate termina
* CycleBarrier poate modifica valaorea la count, cu reset()
* Odata ce CountDownLatch a ajuns la 0, el e oprit, dar CyclicBarrier poate fi refolosit iar si iar.
* Cel mai mare avantaj la CyclicBarrier e ca are **o bariera de actiune**, adica un runnable care va rula automat cand count ajune la 0. Asta si inseamna Barrier,
* **new CyclicBarrier(N, Runnable) – acest Runnable e un t**hread ce va rula cand count = 0
* **reset() o vom folosi dupa ce deja CyclicBarrier a termina cu t**hreadurile ca sa il refolosim iar si ca sa fie reinitializat cu nr. de threaduri setat de noi

class Time {  
 public static void main(String[] args) throws InterruptedException {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(5, new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Treadurile au terminat de executat");  
 }  
 });  
  
 ExecutorService service = Executors.*newFixedThreadPool*(5);  
 for(int i = 0;i<5;i++)  
 service.execute(new Run(i+1,cyclicBarrier));  
 System.*out*.println("Main tread ");  
 }  
}  
  
class Run implements Runnable{  
 int id;  
 CyclicBarrier cyclicBarrier;  
  
 public Run(int id, CyclicBarrier cyclicBarrier) {  
 this.id = id;  
 this.cyclicBarrier = cyclicBarrier;  
 }  
  
 @Override  
 public void run() {  
 try {  
 Thread.*sleep*(2000\*id);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("Este executat Thread cu id "+id);  
 try {  
 cyclicBarrier.await();  
 } catch (InterruptedException | BrokenBarrierException e) {  
 throw new RuntimeException(e);  
 }  
 System.*out*.println("A fost finisat Thread cu id "+id); }  
}

****

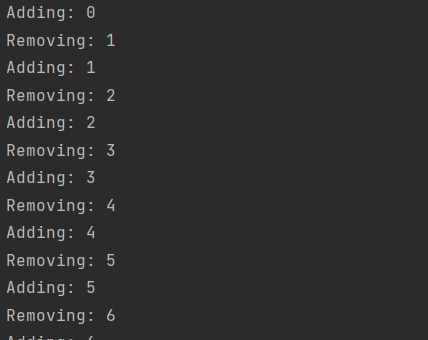
* **main tread a fost executat fara a tine cont de celelalte taskuri din CycleBarrier**
* **Vedem ca intai s-a executat primul task , apoi el a intrat in wait la un moment dat, in acest timp s-a executat si al 2 si tot a intrat in wait si tot asa. La final, cand al 5 tread a dat wait(), a ajuns count la 0 si s-a executat task ce l-am dat in constructor si toate treadurile au fost trezite dupa aceia si au continuat mai departe.**

**Blocking Queue**

* Queue nu este o structura sincronizata
* **BlockingQueue** – interfata care reprezinta o queue ce este tread safe
* Ea ne ajuta anume cu faptul ca Collections.synchronizedList blocheaza intrinsic lock la tot obiectul, dar BlockinQueue nu, ci doar asupra la metoda executata. Deci, un thread poate adau**g**a iteme si tot in acest timp altul poate ster**g**e, fara a intra in waitin**g** state
* put() – adaua iteme
* take() – returneaza head items si il sterge
* Clase ce implementeaza Blockin**g**Queue:
* **LinkedBlockingQueue – nu trebuie de setat in constructor cate elemente va putea pastra**
* **ArrayBlockingQueue – trebuie neaparat de setat in constructor cate elemente va putea pastra!**
* **DelayQueue**

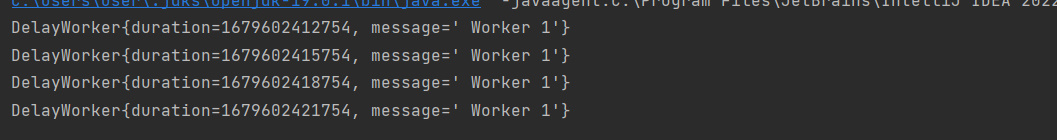
......

class Time {  
 public static void main(String[] args) throws InterruptedException {  
 LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<>();  
 Run1 run1 = new Run1(queue);  
 Run2 run2 = new Run2(queue);  
  
 Thread thread1 = new Thread(run1);  
 Thread thread2 = new Thread(run2);  
 thread1.start();  
 thread2.start();  
 }  
}  
  
class Run1 implements Runnable{  
 BlockingQueue<Integer> blockingQueue;  
  
 public Run1(BlockingQueue<Integer> blockingQueue) {  
 this.blockingQueue = blockingQueue;  
 }  
  
 @Override  
 public void run() {  
 int i = 0;  
 while (true){  
 System.*out*.println("Adding: "+i);  
 try {  
 i++;  
 blockingQueue.put(i);  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}  
class Run2 implements Runnable{  
 BlockingQueue<Integer> blockingQueue;  
  
 public Run2(BlockingQueue<Integer> blockingQueue) {  
 this.blockingQueue = blockingQueue;  
 }  
  
 @Override  
 public void run() {  
 int i = 0;  
 while (true){  
 try {  
 System.*out*.println("Removing: "+blockingQueue.take());  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 }  
 }  
}



**DelayQueue**

* Implementeaza si **BlockingQueue** si **Delayed** interface
* **BlockingQueue** tot e thread safe, si deci poate fi accesat de diferite threaduri si nu are doar un Intrinsit Lock, deci diferite threaduri pot accesa diferite metode
* DelayQueue pastreaza obiecte care implementeaza interfata **Delayed**, dar nu permite accesarea lor pana un oarecare delay a expirat
* Deci, nici un element din DelayQueue nu poate fi accesat pana delay setat de noi nu a expirat. Adica, delay aici e o perioada de timp care trebuie sa treaca ca noi sa putem accesa itemele din queue
* Nu putem adauga iteme nulle
* Coada e sortata in asa fel incat elementul din **head**(primul) are un delay de cea mai scurta perioada de timp, adica elementele incep a fi puse in queue propriu zis cand delay expira
* **DelayQueue poate pastra doar obiecte care implementeaza interfata *Delayed*, deoarece anume ea e cea care are metode pentru a compara delay la obiectele din lista si de a seta acest delay:**
* **int compareTo(Delayed o)** – anume asta e metoda folosita de un DelayQueue pentru a compara delay la obiectele din el si a le sorta dupa delay. Daca delay la elementul care o apeleaza e mai mic ca la obiectul parametru, returnam -1, daca mai mare 1, daca egale 0
* **long getDelay(TimeUnit unit)** – converteste atributul pe care noi l-am creat pentru delay in long.Folosim return
* ATENTIE! DelayQueue va adauga un item doar atunci cand getDelay va returna un numar egal cu 0 sau negativ. Asta e din cauza ca DelayQueue intruna executa getDelay pentru fiecare element din queue, si cand valoarea la metoda returna e negativa, inseamna ca deja s-a trecut de timpul necesar,iata de ce cand setam duration de noi si returnam delay cu metoda, folosim si **System.currentTimeMilis()**
* Evident, avem nevoie si de un atribut propriu pentru timp. Noi il setam cum vrem.
* Daca nici un delay nu e expirat, nu exista nici head si **poll**() va returna null mereu
* Daca un thread acceseaza un element si delay a lui nu e expirat, acel thread va intra in waiting state pana un delay expira
* size() – returneaza numarul de iteme cu delay expirate si neexpirate
* put() – adaugae item
* take() – returneaza head si o sterge
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
    
   BlockingQueue<DelayWorker> queue = new DelayQueue<>();  
   queue.put(new DelayWorker(3000," Worker 1"));  
   queue.put(new DelayWorker(6000," Worker 1"));  
   queue.put(new DelayWorker(9000," Worker 1"));  
   queue.put(new DelayWorker(12000," Worker 1"));  
    
   while(!queue.isEmpty())  
   System.*out*.println(queue.take());  
   }  
  }  
    
  class DelayWorker implements Delayed{  
    
   private long duration;  
   private String message;  
    
   public DelayWorker(long duration, String message) {  
   this.duration = duration + System.*currentTimeMillis*();  
   this.message = message;  
   }  
    
   @Override  
   public long getDelay(TimeUnit unit) {  
   return duration - System.*currentTimeMillis*();  
   }  
    
   @Override  
   public int compareTo(Delayed o) {  
   if(duration < ((DelayWorker)o).getDuration())  
   return -1;  
   else if(duration > ((DelayWorker) o).getDuration())  
   return 1;  
   return 0;  
   }  
    
   public long getDuration() {  
   return duration;  
   }  
    
   public void setDuration(long duration) {  
   this.duration = duration;  
   }  
    
   public String getMessage() {  
   return message;  
   }  
    
   public void setMessage(String message) {  
   this.message = message;  
   }  
    
   @Override  
   public String toString() {  
   return "DelayWorker{" +  
   "duration=" + duration +  
   ", message='" + message + '\'' +  
   '}';  
   }  
  }

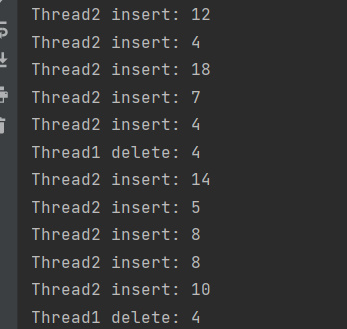


Vedem ca elementele din DelayQueue au putut fi accesate abea dupa ce a trecut delay setat

* Atentie cum setam delay! Metoda getDelay se executa regulat la fiecare element pentru a vedea daca nu cumva s-a trecut de timpul maxim. Daca am returna un numar constant, ca 3000, pai obiectul din queue nu va fi niciodata disponibil, caci el va crede mereu ca mai trebuie sa astepte 3 secunde
* Iata de ce, folosim si System.currentTimeMilis()
* Sa zicem ca folosim minute. Duration setata de noi am vrea sa fie 5 minute, asa dar, daca de ex acum e ora 10:00, vom avea 10:00 : 5min = 10 :05, deci la 10:05 delay expira
* Acum, ca sa stim cand a expirat delay, in metoda getDelay facem asa ca ea sa returneze ceva egal cu 0 sau < 0 cand trece de delay, si sa zicem ca acum e ora 10:05, deci 10:05-10:05(setat de noi) = 0, si asa obiectul e disponibil ca delay a expirat

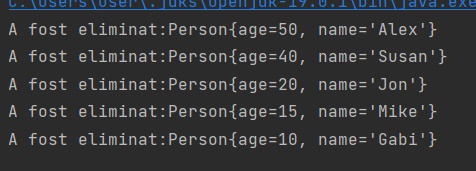
**PriorityBlockingQueue**

* Implementeaza **BlockingQueue** interface
* E exact ca **PriorityQueue**, doar ca threadsafe
* Ea determina in ce ordine vor fi puse obiectele in queue, si de asta iar trebuie sa implementam Comparable cu compareTo() pentru a defini ordinea
* null items nu sunt acceptate
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
   BlockingQueue<Integer> blockingQueue = new PriorityBlockingQueue<>();  
    
    
   Thread thread1 = new Thread(()->  
   {  
   try {  
   while (true) {  
   System.*out*.println("Thread1 delete: " + blockingQueue.take());  
   Thread.*sleep*(5000);  
   }  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   });  
   Thread thread2 = new Thread(()->  
   {  
   try {  
   Random random = new Random();  
   while (true) {  
   int i = random.nextInt(1,20);  
   blockingQueue.put(i);  
   System.*out*.println("Thread2 insert: " + i);  
   Thread.*sleep*(1000);  
   }  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   });  
    
   thread1.start();  
   thread2.start();  
    
   }  
  }

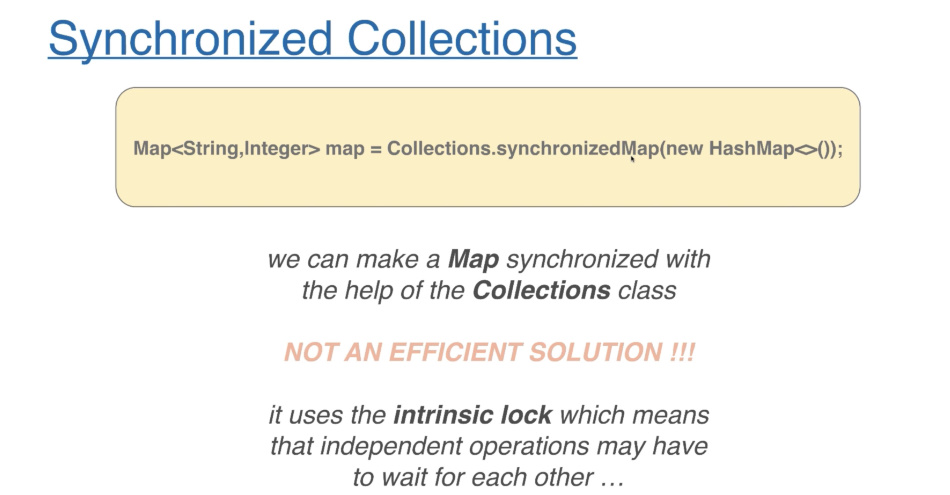


Vedem ca elementele puse sunt automat puse dupa prioritate, si Integer are metoda sa de compareTo(). Deci, vedem ca mereu stergerea se face la elementul cel mai mic, care are prioritate mai mica daca e Integer.

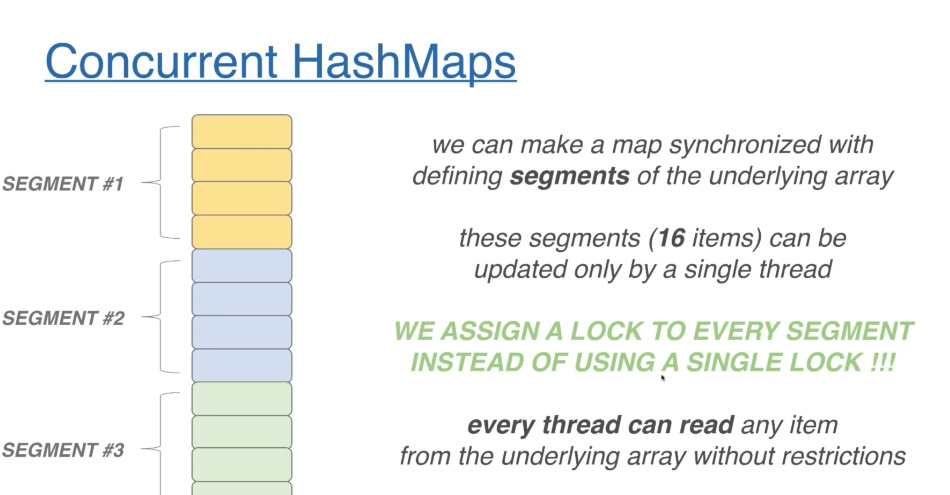
* Putem noi defini un obiect al nostru(sortam invers, cei mai batrani in fata)
* import java.util.Comparator;  
  import java.util.Random;  
  import java.util.concurrent.\*;  
    
  class Time {  
   public static void main(String[] args) throws InterruptedException {  
   BlockingQueue<Person> queue = new PriorityBlockingQueue<>();  
   Thread thread1 = new Thread(()->  
   {  
   try {  
   queue.put(new Person(50,"Alex"));  
   queue.put(new Person(20,"Jon"));  
   queue.put(new Person(15,"Mike"));  
   queue.put(new Person(40,"Susan"));  
   queue.put(new Person(10,"Gabi"));  
    
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
    
    
   });  
   Thread thread2 = new Thread(() -> {  
   try {  
   while (!queue.isEmpty())  
   System.*out*.println("A fost eliminat:"+queue.take());  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   });  
   thread1.start();  
   thread2.start();;  
   }  
  }  
    
  class Person implements Comparable<Person> {  
   private int age;  
   private String name;  
    
   public Person(int age, String name) {  
   this.age = age;  
   this.name = name;  
   }  
    
   public int getAge() {  
   return age;  
   }  
    
   public void setAge(int age) {  
   this.age = age;  
   }  
    
   public String getName() {  
   return name;  
   }  
    
   public void setName(String name) {  
   this.name = name;  
   }  
    
   @Override  
   public String toString() {  
   return "Person{" +  
   "age=" + age +  
   ", name='" + name + '\'' +  
   '}';  
   }  
    
   @Override  
   public int compareTo(Person p) {  
   if(age < p.getAge())  
   return 1;  
   else if(age > p.getAge())  
   return -1;  
   return 0;  
   }  
  }



**ConcurrentMaps**



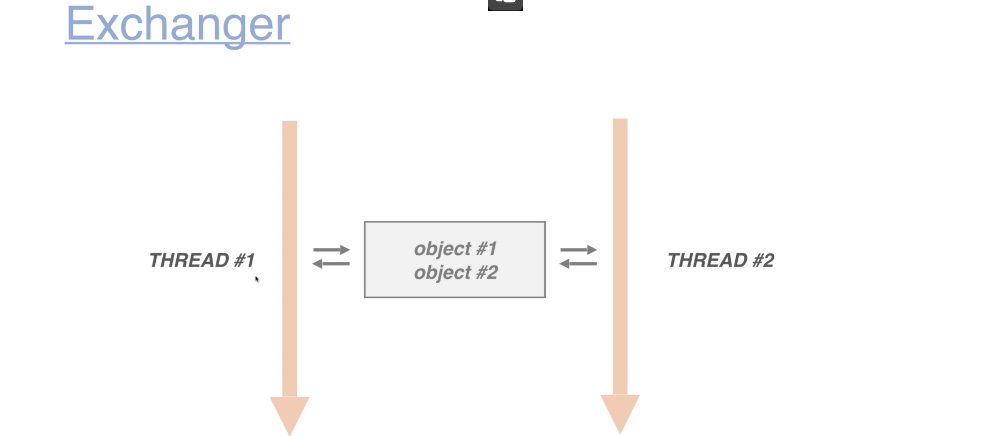
* Avem iar problema ca un thread blocheaza tot obiectul
* Concurent Map ne ajuta aici:



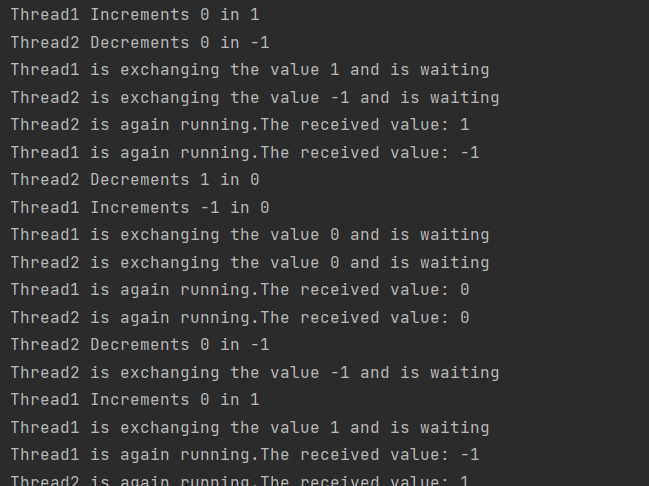
* Deci putem crea un Lock per segment, si segmentele le putem crea cat de mari vrem
* Un segment are in mod normal 16 iteme.
* Deci, fiecare thread ce manipuleaza un segment, il blocheaza anume pe el
* ConcurrentMap are implementare ConcurrentHashMap
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
    
   ConcurrentMap<Integer,String> map = new ConcurrentHashMap<>();  
   Thread thread1 = new Thread(  
   () -> {  
   map.put(1,"Item1");  
   map.put(2,"Item2");  
   map.put(3,"Item3");  
   try {  
   Thread.*sleep*(5000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   map.put(4,"Item4");  
   map.put(5,"Item5");  
   map.put(6,"Item6");  
   }  
   );  
   Thread thread2 = new Thread(  
   () -> {  
   System.*out*.println(map.size());  
   try {  
   Thread.*sleep*(6000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println(map.size());  
   }  
   );  
   thread1.start();  
   thread2.start();  
    
   }  
  }

**Exchanger**

* Cu ajutorul la **Exchanger**, 2 sau mai multe threaduri pot face schimb de obiecte intre ele
* In mod normal, fiecare thread are propria memorie, si propriul stack si cache. Deci, variabilele pastrate de un thread nu sunt disponibile altor threaduri, dar trebuie cumva sa putem sa le facem cunoscute si altor threaduri,adica sa trimitem valorile lor la nevoie, si anume aici ne ajuta exchanger

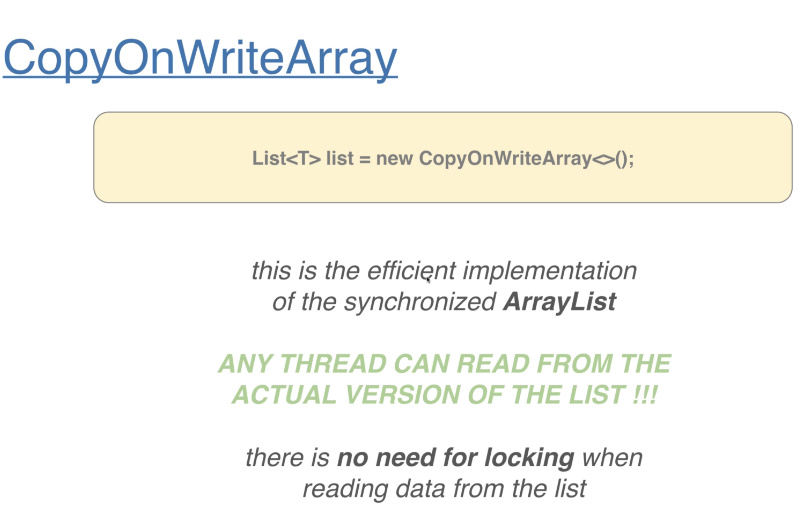


* De ex, in schema de sus Thread1 impartaseste Object1 cu Thread2 si Thread2 impartaseste Object2 cu Thread1
* Cel mai important in Exchanger e ca daca Thread1 vrea sa trimita ceva lui Thread2 cu Exchanger si Thread2 e in waiting state, sau nu raspunde, Thread1 va intra in blocking state pana Thread2 raspunde
* **Fara Exchanger e imposibil sa impartasim datele in multithreading!**
* Impartasirea datelor se face anume prin obiecte, deci daca Thread1 va vrea sa trimita ceva date lui Thread1, va folosi object1 si Thread2 va folosi object1 ca sa trimita date lui Thread1.
* **exchange**(Object) – impartasirea obiectelor se face prin apelarea metodei date de unul dintre threaduri
* Deci, exchange(object) va trimite acest object catre un alt Thread care tot va chema metoda exchange, asa cum ea returneaza catre threadul ce a chemat-o valoarea trimisa de alt thread care tot a apelat-o si a folosit valoarea returnata de ea si apoi a returnat alta inapoi.
* Daca un thread cheama exchange si nu e nici un alt Thread care tot sa o cheme, primul thread nu mai iese din blocking state
* Rolul lui Exchanger este de a se asigura ca un thread trimite datele si intra in waiting state pana altul preia datele si trimtite altele inapoi, si asta prin metoda exchange()
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
   Exchanger<Integer> exchanger = new Exchanger<>();  
   Thread thread1 = new Thread(new Run1(exchanger));  
   Thread thread2 = new Thread(new Run2(exchanger));  
   thread1.start();  
   thread2.start();  
   }  
  }  
  class Run1 implements Runnable{  
   public int counter;  
   public Exchanger<Integer> exchanger;  
   public Run1(Exchanger<Integer> exchanger) {  
   this.exchanger = exchanger;  
   }  
   @Override  
   public void run() {  
   while(true){  
   System.*out*.println("Thread1 Increments "+counter+" in "+(++counter));  
   System.*out*.println("Thread1 is exchanging the value "+counter+" and is waiting");  
   try {  
   counter = exchanger.exchange(counter);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println("Thread1 is again running.The received value: "+counter);  
   try {  
   Thread.*sleep*(2000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
   }  
  }  
  class Run2 implements Runnable{  
   public int counter;  
   public Exchanger<Integer> exchanger;  
   public Run2(Exchanger<Integer> exchanger) {  
   this.exchanger = exchanger;  
   }  
    
   @Override  
   public void run() {  
   while(true){  
   System.*out*.println("Thread2 Decrements "+counter+" in "+(--counter));  
   System.*out*.println("Thread2 is exchanging the value "+counter+" and is waiting");  
   try {  
   counter = exchanger.exchange(counter);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   System.*out*.println("Thread2 is again running.The received value: "+counter);  
   try {  
   Thread.*sleep*(2000);  
   } catch (InterruptedException e) {  
   throw new RuntimeException(e);  
   }  
   }  
   }  
  }

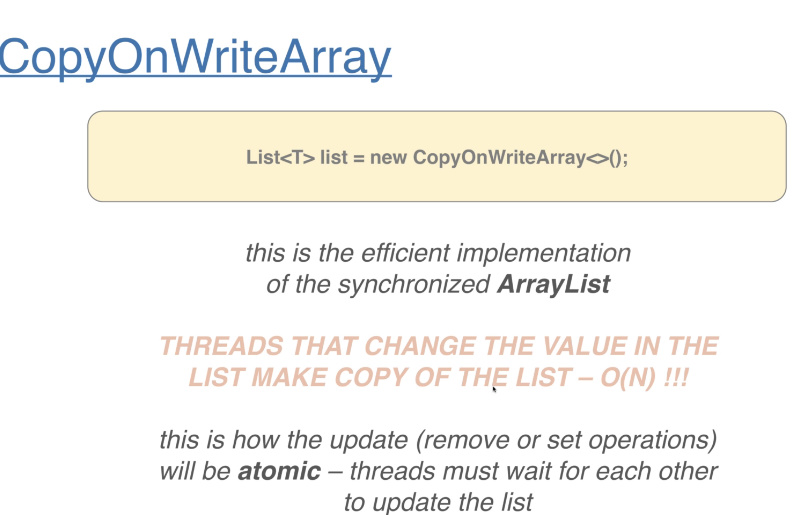


* Threadurile sunt executate exact asa ca in mod normal, adica unul se poate opri putin ca altul sa ruleze si apoi continua si tot asa
* Metoda exchange() trimite datele si face ca un thread sa fie in waiting state pana altul raspunde si el cu exchange()

**CopyOnWriteArrayList**

* Implementarea unui ArrayList cu Collections.synchronizedList(new ArrayList()) nu e prea buna, si cea cu GroupingQueue tot e rea, caci trebuie sa setam un nr initial de element
* CopyOnWriteArray e pentru a crea un synchronzied ArrayList
* Este mult mai rapida ca Vector
* 

Cand un thread citeste valori din array, celelalte tot vor putea sa le citeasca in acelasi timp

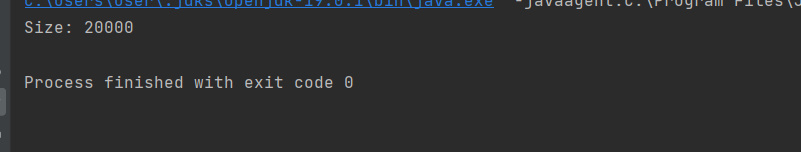
* 

Totusi, daca un thread modifica cumva vreo valoare, celelalte threaduri care tot vor sa modifice trebuie sa astepte pana celalalt termina. **Daca threadul incepe sa faca modificarea si e intrerupt de time slicing alghoritm, nici un alt thread nu va putea face vreo modificare la date pana celalalt nu termina de facut modificare si nu pune noile sale date din cache in lista**.

* Cand un thread vrea sa modifice o valoare, el face in cache o copie la array, modifica valoarea si apoi transfera noile date in array list
* daca threadurile ar putea face modificari in acelasi timp, ar fi mari probleme.

class Time {  
 public static void main(String[] args) throws InterruptedException {  
 List<Integer> arrayList = new CopyOnWriteArrayList<Integer>();  
 arrayList.add(1);  
 arrayList.add(2);  
 arrayList.add(3);  
 arrayList.add(4);  
 arrayList.add(5);  
 arrayList.add(6);  
  
 Thread thread1 = new Thread(new Writer("Thread1",arrayList,2));  
 Thread thread2 = new Thread(new Writer("Thread2",arrayList,3));  
 Thread thread3 = new Thread(new Reader(arrayList));  
  
 thread1.start();  
 thread2.start();  
 thread3.start();  
  
 }  
}  
  
class Writer implements Runnable{  
 String name;  
 List<Integer> arrayList;  
 int sleep;  
  
 public Writer(String name, List<Integer> arrayList,int sleep) {  
 this.name = name;  
 this.arrayList = arrayList;  
 this.sleep = sleep;  
 }  
  
 @Override  
 public void run() {  
 Random random = new Random();  
 while(true){  
 int index = random.nextInt(arrayList.size());  
 int value = random.nextInt(100);  
 System.*out*.println(name + " is modifying index "+index+" in value "+value);  
 arrayList.set(index,value);  
 try {  
 Thread.*sleep*(1000\*sleep);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}  
  
class Reader implements Runnable{  
  
 List<Integer> arrayList;  
  
 public Reader(List<Integer> arrayList) {  
 this.arrayList = arrayList;  
 }  
  
 @Override  
 public void run() {  
 while(true){  
 System.*out*.println(arrayList);  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
}

class Time {  
 public static void main(String[] args) throws InterruptedException {  
 List<Integer> arrayList = new CopyOnWriteArrayList<Integer>();  
  
 Thread thread1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<10000;i++)  
 arrayList.add(i);  
 }  
 });  
 Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for(int i = 0;i<10000;i++)  
 arrayList.add(i);  
 }  
 });  
  
 thread1.start();  
 thread2.start();  
 thread1.join();  
 thread2.join();  
 System.*out*.println("Size: "+arrayList.size());  
 }  
}



* Deci, ambele threaduri adaugau iteme treptat, adica unul si apou altul si tot asa.
* Ideea e ca threadurile facea copii ale listei la ele in cache, modificau lista la ele in cache si apoi copiau datele in lista din RAM
* Dar, CopyOnWriteArrayList se asigura ca niciun Thread sa nu poata copia cache din lista daca alt Thread nu a reusit sa dea update la lista din RAM desi a modificat-o la el in cache.De ex, daca thread1 a copiat lista la el, a adaugat un element, dar nu a reusit sa returneze lista creata, ca a fost oprit, thread2 nu va putea face copy la lista din RAM pana thread1 nu o va returna.
* **Threadurile/CPU pastreaza in cache nu doar referinta la un obiect, ci chiar o copie a obiectului!**
* class Time {  
   public static void main(String[] args) throws InterruptedException {  
   RunTest runTest = new RunTest();  
   Thread thread1 = new Thread(()->{  
   for(int i = 0;i<10000;i++)  
   runTest.a++;  
   });  
   Thread thread2 = new Thread(()->{  
   for(int i = 0;i<10000;i++)  
   runTest.a++;  
   });  
   thread1.start();  
   thread2.start();  
   thread1.join();  
   thread2.join();  
   System.*out*.println(runTest.a);  
   }  
  }  
  class RunTest {  
   int a = 0;  
    
  }

