IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html

JAVA memory model

- Fiecare thread are propria stiva de executie, heap-ul este comun pentru toate thread-urile.
- Erorile de consistenta a memoriei apar atunci cand thread-uri diferite au vad in mod inconsistent datele comune.
- Accesul la memoria comuna este reglementat de relatia ***happens-before*** care stabileste cand modificarile facute de un thread sunt vizibile altui thread:

  daca actiunea X este in relatie *happens-before* cu actiunea Y atunci
  exita garantia ca thread-ul care executa Y va vedea rezultatele actiunii X

- In absenta relatiei *happens-before* actiunile pot fi reordonate (compiler optimization).

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Happens-before

- un contor c=0 este comun thread-urilor A si B

	A	B
Instructiuni	i1: c++	i2: System.out.println(c)

- nu exista garantia ca B va scrie 1 decat daca i1 *happens-before* i2

- Actiuni care pot crea relatia *happens-before* :
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Exemple: actiunile de sincronizare, accesul variabilelor volatile

https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html

Happens-before

daca actiunea X este in relatie *happens-before* cu actiunea Y atunci
exita garantia ca thread-ul care executa Y va vedea rezultatele actiunii X

- Relatia happens-before este o relatie de ordine partial pe toate actiunile unui program.
- Relatia happens-before este tranzitiva.

Reguli care definesc happens-before

Thread unic: in cadrul aceluiasi thread, relatia *happens-before* este stabilita de ordinea actiunilor in program.
Monitor: orice actiune unlock pe un lacat este in relatia *happens-before* cu orice actiune lock ulterioara pe acelasi lacat.
Variabile volatile: scrierea unei variabile volatile este in relatia *happens-before* cu orice citire ulterioara a variabilei.
Thread.start(): actiunea *thread1.start()* este in relatia *happens-before* cu orice actiune din *thread1*
actiunea de pornire a unui thread este in relatia *happens-before* cu orice alta actiune din thread-ul respective
Thread.join(): orice actiune din *thread1* este in relatia *happens-before* cu orice actiune ulterioara lui *thread1.join()*

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/happens-before.html

Exista reguli care defines relatia happens-before pentru clasele din java.util.concurrent:
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

- ExecutorService: fork-join

```
public class ForkJoinPool
extends AbstractExecutorService
```

Diferenta dintre o piscine din clasa ForkJoinPool si cele create de alte servicii Executor este implementarea unei metode de work-stealing.

newCachedThreadPool(), newFixedThreadPool(n)

- mai multe thread-uri care nu sunt demon si care acceseaza o coada comuna
- sunt indicate pentru task-uri mari in thread-uri separate

newForkJoinPool()

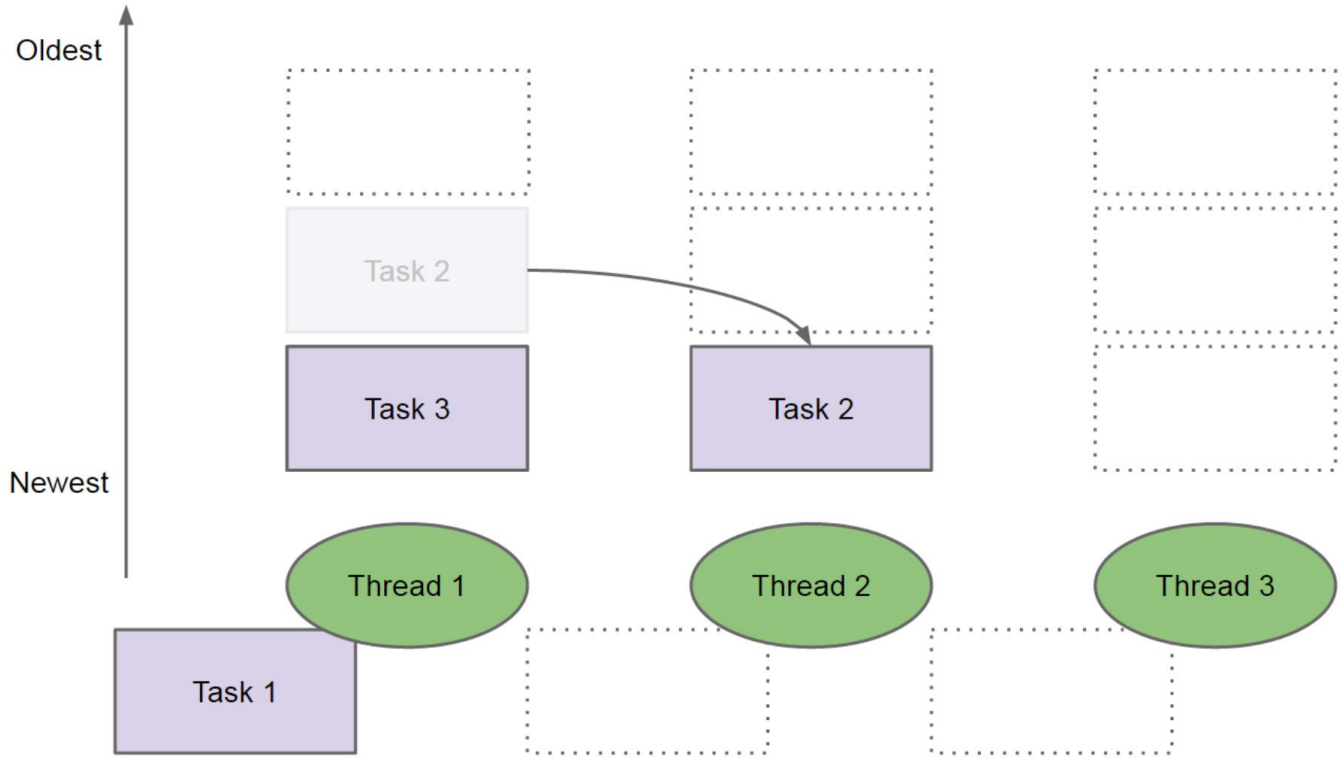
- thread-uri demon, nu se creaza thread-uri noi pentru fiecare subtask,
- fiecare thread din piscina mentine o coada (double-ended queue) de task-uri, thread-urile libere iau task-uri care asteapta in cozile thread-urilor ocupate
- task-uri care presupun executia altor task-uri mai mici

Documentatie:
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinTask.html
https://www.researchgate.net/publication/2609854_A_Java_ForkJoin_Framework

In exemple folosim:
http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html
http://www.baeldung.com/java-fork-join

Daca
- Thread1 executa Task1
- Task1 creaza Task2 si Task3 si
are nevoie de rezultatele lor pentru a continua

atunci
-Thread1 pune in asteptare Task1 si va pune in coada proprie Task2 si Task3
- Thread1 va fi liber pentru a executa un task, celalalt va fi furat de un alt thread liber



Step 3 – Thread 2 steals Task 2

http://www.javacreed.com/java-fork-join-example/

- ExecutorService: fork-join

```
public class ForkJoinPool
extends AbstractExecutorService
```

- Crerea piscinei de threaduri
- Crearea task-ului
- Tarsk-ul este trimis piscinei folosind

invoke - trimite task-ul in executie si intoarce rezultatul
execute, submit – trimit task-ul in executie;
trebuie folosit join pentru a obtine rezultatul

Crearea piscinei de thread-uri

```
ForkJoinPool fjpool = ForkJoinPool.commonPool() // recomandat,
                                                // incearca sa foloseasca toate procesoarele disponibile
ForkJoinPool fjpool =  new ForkJoinPool() // new ForkJoinPool(5)
```

Crerea task-urilor

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>
```

```
public abstract class RecursiveAction
extends ForkJoinTask<Void>
```

```
public abstract class RecursiveTask<V>
extends ForkJoinTask<V>
```

metoda compute
implementeaza actiunea
executata de task

- ExecutorService: fork-join

```
public class ForkJoinRecAc extends RecursiveAction {

public ForkJoinRecAc (long workLoad) {
      this.workLoad = workLoad;
   }


protected void compute() {
……
}
}


ForkJoinPool fjpool =  ForkJoinPool.commonPool()
ForkJoinRecAc fjaction= new ForkJoinRecAc(500);
pool.invoke(fjaction);
```

```
public class ForkJoinRecTk  extends RecursiveTask<T> {

public ForkJoinRecTk (long workLoad) {
      this.workLoad = workLoad;
   }


protected <T> compute() {
……
}
}


ForkJoinPool fjpool =  ForkJoinPool.commonPool()
ForkJoinRecTk  fjtask= new ForkJoinRecTk(500);
<T> result = fjpool.invoke(fjtask);
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

- ExecutorService: fork-join cu Recursive Action (forma generala)

```
public class ForkJoinRecAc extends RecursiveAction {

public ForkJoinRecAc (long workLoad) {
       this.workLoad = workLoad;
   }
protected void compute() {

 if(this.workLoad > limit) { …
                          ForkJoinTask.invokeAll(createSubtasks());
                         }
else {// prelucrata de thread-ul curent}

}

private List<ForkJoinRecAc> createSubtasks() { … }
}
```

invokeAll(Collection<T> tasks)

Trimite in executie toate task-urile (face fork() pe toate task-urile)

https://docs.oracle.com/javase/tutorial/essential/concurrency

- ExecutorService: fork-join cu RecursiveTask (forma generala)

```
public class ForkJoinRecTk extends RecursiveTask <T> {

public ForkJoinRecTk(long workLoad) { this.workLoad = workLoad;}

protected  <T>  compute() {
  if(this.workLoad > limit) { …
                    Collection<ForkJoinTk> futures = ForkJoinTask.invokeAll(createSubtasks());
                    <T>  result1 = 0;
                   for(ForkJoinTk future : futures) {  result += future.join();  }
                   return result1
                              }
else {// prelucrata de thread-ul current
        return result2
}}

private List<ForkJoinRecAc> createSubtasks() { … }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

ExecutorService: fork-join – crearea subtask-urilor

```
private List<MyRecursiveTask> createSubtasks() {
        List<MyRecursiveTask> subtasks =
        new ArrayList<MyRecursiveTask>();

        MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);
        MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Exemplu program: fork/join pool cu RecursiveAction

```
protected void compute() {

 if(this.workLoad > 16) {
        System.out.println(Thread.currentThread().getName()+": Splitting workLoad : " + this.workLoad);

           ForkJoinTask.invokeAll(createSubtasks());
        }

       else {
         System.out.println(Thread.currentThread().getName()+": Doing workLoad myself: " + this.workLoad);
      }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Exemplu program: Exemplu program: fork/join pool cu RecursiveAction

```
C:\myjava\tutoracle>java myforkjoinex/ForkJoinEx
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 500
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 250
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 125
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 62
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 31
ForkJoinPool.commonPool-worker-2: Splitting workLoad : 250
ForkJoinPool.commonPool-worker-2: Splitting workLoad : 125
ForkJoinPool.commonPool-worker-2: Splitting workLoad : 62
ForkJoinPool.commonPool-worker-1: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-3: Splitting workLoad : 125
ForkJoinPool.commonPool-worker-2: Splitting workLoad : 31
ForkJoinPool.commonPool-worker-3: Splitting workLoad : 62
ForkJoinPool.commonPool-worker-1: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-3: Splitting workLoad : 31
ForkJoinPool.commonPool-worker-2: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-3: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-1: Splitting workLoad : 31
ForkJoinPool.commonPool-worker-3: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-2: Doing workLoad myself: 15
ForkJoinPool.commonPool-worker-3: Splitting workLoad : 31
ForkJoinPool.commonPool-worker-1: Doing workLoad myself: 15
```

Exemplu program: ForkJoin pool cu RecursiveTask

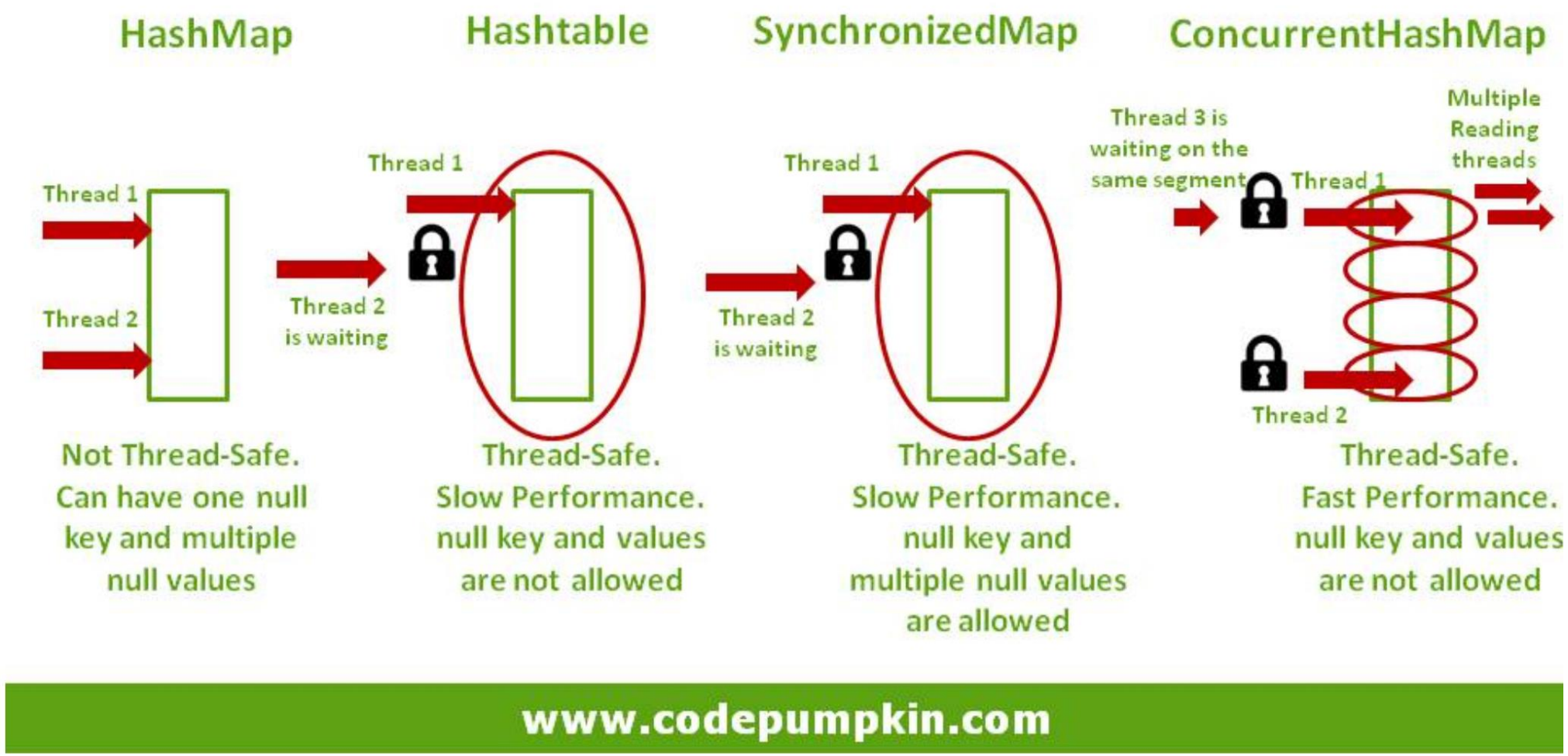
```
protected Long compute() {

if(this.workLoad > 16) {
        System.out.println("Splitting workLoad : " + this.workLoad);

      Collection<MyRecursiveTask> futures = ForkJoinTask.invokeAll(createSubtasks());
        long result = 0;
        for(MyRecursiveTask future : futures) {
           result += future.join();
        }
        return result;

    } else {
        System.out.println("Doing workLoad myself: " + this.workLoad);
        return workLoad * 3;
    }
  }
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

Exemplu program: ForkJoin pool cu RecursiveTask

```
C:\myjava\tutoracle>java recursivetask/ForkJoinExTk
Splitting workLoad : 100
Splitting workLoad : 50
Splitting workLoad : 25
Splitting workLoad : 50
Doing workLoad myself: 12
Splitting workLoad : 25
Splitting workLoad : 25
Doing workLoad myself: 12
Doing workLoad myself: 12
Doing workLoad myself: 12
Doing workLoad myself: 12
Doing workLoad myself: 12
Doing workLoad myself: 12
Splitting workLoad : 25
Doing workLoad myself: 12
Doing workLoad myself: 12
mergedResult = 288
```

Colectii concurente



http://codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap/

➢ public class ConcurrentHashMap<K,V>

- colectia este impartita in fragmente ("bucket") care sunt prelucrate in parallel; colectia poate fi create cu un anume
- nivel de concurenta care implicit este 16; numarul de fragmente prelucrate in paralel este dat de nivelul de concurenta
- cand un thread executa o operatie care blocheaza, este blocat numai fragmentul corespunzator
- actualizarile sunt operatii care blocheaza, regasirile nu blocheaza; este regasita ultima valoare modificata de o actualizare care s-a finalizat
- actiunile dintr-un thread care plaseaza un obiect in colectie sunt in relatie happens-before cu actiunile dintr-un alt thread care urmeaza accesarii/stergerii elementului din colectie
- colectia nu este ordonata, elementele pot fi procesate in paralel in ordini diferite
- piscina de thread-uri este creata cu ForkJoinPool.commonPool()

Metode pentru prelucrarea elementelor colectiei in paralel

forEach(par, …), search(par, …), reduce(par, …)
daca numarul de elemente din colectie este mai mic decat par atunci executia este secventiala

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html

ConcurrentHashMap: forEach

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

public class ConcHashMap{

   public static void main(String[] args) {
ConcurrentHashMap<Integer, String> hashMap = new ConcurrentHashMap<>();
for (int i=1 ; i <= 10; i++){hashMap.put(i, "A"+i);}

hashMap.forEach(3, (k, v) -> System.out.println("key->" + k + " has value-> " + v +",
      by reader thread-> "+ Thread.currentThread().getName()));
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
C:\myjava\tutoracle>java ConcHashMap
key->8 has value-> A8, by reader thread-> ForkJoinPool.commonPool-worker-2
key->4 has value-> A4, by reader thread-> ForkJoinPool.commonPool-worker-1
key->1 has value-> A1, by reader thread-> main
key->5 has value-> A5, by reader thread-> ForkJoinPool.commonPool-worker-1
key->6 has value-> A6, by reader thread-> ForkJoinPool.commonPool-worker-1
key->9 has value-> A9, by reader thread-> ForkJoinPool.commonPool-worker-2
key->7 has value-> A7, by reader thread-> ForkJoinPool.commonPool-worker-1
key->2 has value-> A2, by reader thread-> main
key->10 has value-> A10, by reader thread-> ForkJoinPool.commonPool-worker-2
key->3 has value-> A3, by reader thread-> main
```

ConcurrentHashMap: reduce

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

public class ConcHashMap{

  public static void main(String[] args) {
ConcurrentHashMap<Integer, String> hashMap = new ConcurrentHashMap<>();
for (int i=1 ; i <= 10; i++){hashMap.put(i, "A"+i);}

String result = hashMap.reduce(1,  (k, v) -> {
      System.out.println("Transform: " + Thread.currentThread().getName());
      return k + "=" + v;
            },
                                          (s1, s2) -> {
                              System.out.println("Reduce: " + Thread.currentThread().getName());
                          return s1 + ", " + s2; });

System.out.println("Result: " + result);
}}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
Transform: main
Transform: ForkJoinPool.commonPool-worker-3
Transform: ForkJoinPool.commonPool-worker-2
Transform: ForkJoinPool.commonPool-worker-1
Transform: ForkJoinPool.commonPool-worker-3
Reduce: ForkJoinPool.commonPool-worker-3
Transform: ForkJoinPool.commonPool-worker-2
Transform: ForkJoinPool.commonPool-worker-1
Transform: ForkJoinPool.commonPool-worker-3
Transform: ForkJoinPool.commonPool-worker-2
Reduce: ForkJoinPool.commonPool-worker-2
Reduce: ForkJoinPool.commonPool-worker-2
Transform: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Reduce: ForkJoinPool.commonPool-worker-1
Result: 1=A1, 2=A2, 3=A3, 4=A4, 5=A5, 6=A6, 7=A7, 8=A8, 9=A9, 10=A10
```

Atentie: deoarece ordinea de executie nu este garantata, operatia care acumuleaza rezultatele produse de fiecare intrare trebuie sa fie asociativa iar ordinea in care rezultatele sunt accumulate nu trebuie sa fie importanta.

Exemplu: ConcurrentHashMap care este prelucrata simultan de mai multe thread-uri

- thread-urile cititor parcurg colectia si o afiseaza
- thread-urile scriitor actualizeaza colectia prin introducere si stergere de elemente

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

public class ConcHashMapRW{

   public static void main(String[] args) {
      ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

      new WriterThread(map, "Writer-1").start();
      new WriterThread(map, "Writer-2").start();

      for (int i = 1; i <= 5; i++) {
         new ReaderThread(map, "Reader-" + i).start();
      }
   }}
```

http://javatutorialhq.com/java/util/hashmap-class/putifabsent-method-example

Exemplu (cont): thread-urile care scriu

```
public class WriterThread extends Thread {
   private ConcurrentMap<Integer, String> map;
   private String name;
public WriterThread(ConcurrentMap<Integer, String> map, String threadName) {
      this.map = map; this.name = threadName; }

   public void run() {
      while (true) {
         Integer key = ThreadLocalRandom.current().nextInt(10);  String value = name;
         if(map.putIfAbsent(key, value) == null) {
                  System.out.println(System.currentTimeMillis()+":"+name+" has put["+key+"=>"+value+"]") ;}
          Integer keyToRemove = ThreadLocalRandom.current().nextInt(10);
         if (map.remove(keyToRemove, value)) {
           System.out.println(System.currentTimeMillis()+":"+name+" has removed ["+keyToRemove+"=>"+value+"]");  }

         try {   Thread.sleep(500);} catch (InterruptedException ex) {}
   }}
}
```

Exemplu (cont): thread-urile care citesc

```
public class ReaderThread extends Thread {
   private ConcurrentHashMap<Integer, String> map;
   private String name;

   public ReaderThread(ConcurrentHashMap<Integer, String> map, String threadName) {
      this.map = map; this.name = threadName;}

   public void run() {
      while (true) {
          long time = System.currentTimeMillis();  String output = time + ": " + name + ": ";

       for (Integer key : map.keySet()) {
            String value = map.get(key);
            output += key + "=>" + value + "; ";
          }

          System.out.println(output);

          try { Thread.sleep(300);} catch (InterruptedException ex) {}
      }}
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
C:\myjava\tutoracle>java conchashmaprw/ConcHashMapRW
1525271205368: Writer-1 has put [5 => Writer-1]
1525271205369: Writer-2 has put [6 => Writer-2]
1525271205403: Reader-2: 5=>Writer-1; 6=>Writer-2;
1525271205403: Reader-1: 5=>Writer-1; 6=>Writer-2;
1525271205403: Reader-5: 5=>Writer-1; 6=>Writer-2;
1525271205403: Reader-4: 5=>Writer-1; 6=>Writer-2;
1525271205403: Reader-3: 5=>Writer-1; 6=>Writer-2;
1525271205818: Reader-5: 5=>Writer-1; 6=>Writer-2;
1525271205818: Reader-4: 5=>Writer-1; 6=>Writer-2;
1525271205818: Reader-3: 5=>Writer-1; 6=>Writer-2;
1525271205818: Reader-2: 5=>Writer-1; 6=>Writer-2;
1525271205818: Reader-1: 5=>Writer-1; 6=>Writer-2;
1525271205905: Writer-1 has put [4 => Writer-1]
1525271206126: Reader-5: 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206126: Reader-1: 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206126: Reader-2: 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206126: Reader-4: 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206126: Reader-3: 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206405: Writer-1 has put [3 => Writer-1]
1525271206405: Writer-2 has put [1 => Writer-2]
1525271206427: Reader-4: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206427: Reader-2: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206427: Reader-1: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206427: Reader-5: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206427: Reader-3: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206734: Reader-2: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
1525271206734: Reader-4: 1=>Writer-2; 3=>Writer-1; 4=>Writer-1; 5=>Writer-1; 6=>Writer-2;
```

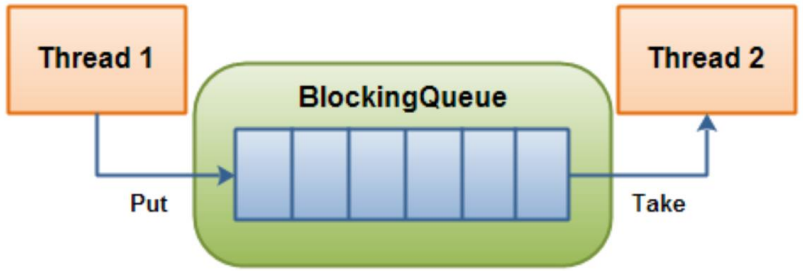
```
C:\myjava\tutoracle>java conchashmaprw/ConcHashMapRW
1525271093659: Writer-1 has put [9 => Writer-1]
1525271093659: Writer-2 has put [3 => Writer-2]
1525271093701: Writer-2 has removed [3 => Writer-2]
1525271093701: Reader-2: 9=>Writer-1;
1525271093701: Reader-3: 9=>Writer-1;
1525271093701: Reader-4: 9=>Writer-1;
1525271093701: Reader-5: 9=>Writer-1;
1525271093701: Reader-1: 9=>Writer-1;
1525271094105: Reader-5: 9=>Writer-1;
1525271094105: Reader-2: 9=>Writer-1;
1525271094105: Reader-3: 9=>Writer-1;
1525271094105: Reader-1: 9=>Writer-1;
1525271094105: Reader-4: 9=>Writer-1;
1525271094205: Writer-1 has put [6 => Writer-1]
1525271094418: Reader-5: 6=>Writer-1; 9=>Writer-1;
1525271094418: Reader-4: 6=>Writer-1; 9=>Writer-1;
1525271094418: Reader-1: 6=>Writer-1; 9=>Writer-1;
1525271094418: Reader-3: 6=>Writer-1; 9=>Writer-1;
1525271094418: Reader-2: 6=>Writer-1; 9=>Writer-1;
1525271094721: Reader-2: 6=>Writer-1; 9=>Writer-1;
1525271094721: Writer-2 has put [2 => Writer-2]
1525271094723: Writer-1 has put [0 => Writer-1]
1525271094721: Reader-5: 6=>Writer-1; 9=>Writer-1;
1525271094721: Reader-3: 6=>Writer-1; 9=>Writer-1;
1525271094721: Reader-4: 6=>Writer-1; 9=>Writer-1;
1525271094721: Reader-1: 6=>Writer-1; 9=>Writer-1;
```

Operatiile de gasire a informatiei nu blocheaza si reflecta ultima actualizare care a fost finalizata.

https://docs.oracle.com/javase/tutorial/essential/concurrency

- Colectii concurente: ArrayBlockingQueue

public class ArrayBlockingQueue<E>



A BlockingQueue with one thread putting into it, and another thread taking from it.

http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

Constructor

ArrayBlockingQueue(int capacity)
este create cu o capacitate fixata care nu poate fi schimbata
ArrayBlockingQueue(int capacity, boolean fair)
cand fair=true thread-urile in asteptare sunt procesate in ordinea FIFO care implicit nu este garantata

Metode

	Throws Exception	Special Value	Blocks	Times Out
Insert	`add(o)`	`offer(o)`	`put(o)`	`offer(o, timeout, timeunit)`
Remove	`remove(o)`	`poll()`	`take()`	`poll(timeout, timeunit)`
Examine	`element()`	`peek()`		

http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html

Exemplu: Producator-Consumator folosind BlockingQueue

```
class Producer implements Runnable {

   private final BlockingQueue<String> sharedQueue;

   public Producer(BlockingQueue<String> sharedQueue) {
      this.sharedQueue = sharedQueue;
   }



public void run() {
      for(int i=0; i<10; i++){
         try {
            System.out.println("Produced: " +
i);Thread.sleep(2000);
            sharedQueue.put("obiect"+i);
         } catch (InterruptedException ex) {}

      }}}
```

```
class Consumer implements Runnable{

   private final BlockingQueue<String> sharedQueue;

   public Consumer (BlockingQueue<String> sharedQueue) {
      this.sharedQueue = sharedQueue;
   }

public void run() {
      while(true){
         try {
            System.out.println("Consumed: "+ sharedQueue.take());
         } catch (InterruptedException ex) {}

      }
   }}
```

http://javarevisited.blogspot.ro/2012/02/producer-consumer-design-pattern-with.html

https://docs.oracle.com/javase/tutorial/essential/concurrency

```
C:\myjava\tutoracle>java PCBlockingQueue
Produced: 0
Produced: 1
Consumed: obiect0
Produced: 2
Consumed: obiect1
Produced: 3
Consumed: obiect2
Consumed: obiect3
Produced: 4
Produced: 5
Consumed: obiect4
Produced: 6
Consumed: obiect5
Produced: 7
Consumed: obiect6
Produced: 8
Consumed: obiect7
Produced: 9
Consumed: obiect8
Consumed: obiect9
```

```
import java.util.concurrent.*

 public class PCBlockingQueue{

    public static void main(String args[]){

     //Creating shared object
     BlockingQueue<String> sharedQueue = new
LinkedBlockingQueue<String> ();

     //Creating Producer and Consumer Thread
     Thread prodThread = new Thread(new Producer(sharedQueue));
     Thread consThread = new Thread(new
Consumer(sharedQueue));

     //Starting producer and Consumer thread
     prodThread.start();
     consThread.start();
    }

}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency