IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

➢ Interfata Lock

interface Lock

class ReentrantLock

Metode:
lock(), unlock(), tryLock()

Lock vs **synchronized**

- **syncronized** acceseaza lacatul intern al resursei si impune o programare structurata: primul thread care detine resursa trebuie sa o si elibereze

- obiectele din clasa **Lock** nu acceseaza lacatul resursei ci **propriul lor lacat**, permitand mai multa flexibilitate

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

➢ Interface ReadWriteLock

```
interface Lock
interface ReadWriteLock   extends Lock
class ReentrantReadWriteLock
```

- mentine o pereche de lacate: unul pentru citire si unul pentru scriere
- lacatul pentru citire poate fi detinut de mai multe thread-uri simultan, daca nu exista o solicitare pentru scriere; lacatul pentru scriere poate fi detinut de un singur thread
    - metoda **readLock()** intoarce lacatul pentru cititori
    - metoda **writeLock()** intoarce lacatul pentru scriitori
- implementare: class ReentrantReadWriteLock

➢ Modelul de interactiune Cititori-Scriitori (Reader-Writers)

- Mai multe threaduri au acces la o resursa.
- Unele thread-uri scriu (writers), iar altele citesc (readers).
- Resursa poate fi accesata simultan de mai multi cititori.
- Resursa poate fi accesata de un singur scriitor.
- Resursa nu poate fi accesata simultan de cititori si de scriitori

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

public class ReaderWriter {
   private static Integer counter = 0;
   private static ReadWriteLock lock = new ReentrantReadWriteLock();

   public static void main(String[] args) {

      (new Thread(new TaskW())).start();
      (new Thread(new TaskR())).start();
      (new Thread(new TaskR())).start();
      (new Thread(new TaskW())).start();
      (new Thread(new TaskR())).start();
      (new Thread(new TaskR())).start();
      (new Thread(new TaskR())).start();
      (new Thread(new TaskW())).start();
      }
```

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

```
private static class TaskW implements Runnable {
    public void run () {

        lock.writeLock().lock();
     try{
     int temp = counter;
     for (int i=0;i<5;i++) {counter++;
     Thread.currentThread().sleep(1);}
     System.out.println(Thread.currentThread().getName() +
                                                          " - before: "+temp+" after:" + counter);}
     catch (InterruptedException e){}
     finally {
      lock.writeLock().unlock();}

     }
  }
```

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

```
private static class TaskR implements Runnable {
      public void run () {

            lock.readLock().lock();

       try{
       System.out.println(Thread.currentThread().getName() + " counter:" + counter);}

        finally {
        lock.readLock().unlock();}

       }}}
```

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

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ReaderWriter
Thread-0 - before: 0 after:5
Thread-0 - before: 0 after:5
Thread-1 counter:5
Thread-3 - before: 5 after:10
Thread-4 counter:10
Thread-2 counter:10
Thread-6 counter:10
Thread-5 counter:10
Thread-7 - before: 10 after:15

C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ReaderWriter
Thread-0 - before: 0 after:5
Thread-4 counter:5
Thread-2 counter:5
Thread-1 counter:5
Thread-3 - before: 5 after:10
Thread-7 - before: 10 after:15
Thread-6 counter:15
Thread-5 counter:15
```

➢ Semafor cu cantitate (quantity semaphore)

```
public class Semaphore
extends Object

Semaphore(int permits)  // constructor
```

- implementeaza un semafor cu cantitate
  care coordoneaza accesul la un numar precizat de resurse

- metoda **aquire()**
  thread-ul care apeleaza aquire cere accesul la o resursa;
  daca nu sunt resurse, thread-ul este blocat

- metoda **release()**
  thread-ul care apeleaza release elibereaza accesul la o resursa

```
Semaphore sem = new Semaphore(n);

sem.acquire();

      ...  //sectiune critica


sem.release();
```

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

```
public class Semaphore
extends Object

Semaphore(int permits)  // constructor

Semaphore sem = new Semaphore(n);

sem.acquire();

      ...  //sectiune critica
sem.release();
```

Diferenta dintre un obiect construit cu **Semaphore(1)** si unul din clasa **Lock** este urmatoarea:

- lacatul intern al obiectului din clasa **Semaphore** este eliberat de orice thread care face **release**
- lacatul intern al obiectului din clasa **Lock** este eliberat numai de thread-ul care il detine

Varianta **Semaphore(int permits, true)** thread-urile care asteapta sa faca aquire sunt FIFO

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

Exemplu:

- un semafor coordoneaza accesul la 2 resurse
- exista 4 thread-uri care cer accesul la resursa
- dupa ce primeste accesul, fiecare thread executa 3 task-uri, apoi elibereaza resursa

```
public class Semaphores{

    static Semaphore semaphore = new Semaphore(2);

    static class MyThread extends Thread {

        // thread-ul va face aquire, va executa task-urile, apoi va face release
    }

public void main(String[] args) { …}
}
```

```
public static void main(String[] args)   {
        MyThread t1 = new MyThread("A"); t1.start();
          …..
        MyThread t4 = new MyThread("D"); t4.start();
}
```

http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/

```
static class MyThread extends Thread {
        String name = "";
        MyThread(String name) { this.name = name;}

public void run()  {
   try {
          semaphore.acquire();
                try {

              for (int i = 1; i <= 3; i++) {
                 System.out.println(name + " : is performing operation " + i }
                                  Thread.sleep(1000);}

                 } finally { semaphore.release();}

    }  catch (InterruptedException e) {}
}}
```

Atentie! **aquire** pune thread-urile in asteptare, deci poate arunca o exceptie

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

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java Semaphores
B : is performing operation 1
A : is performing operation 1
B : is performing operation 2
A : is performing operation 2
B : is performing operation 3
A : is performing operation 3
C : is performing operation 1
D : is performing operation 1
C : is performing operation 2
D : is performing operation 2
C : is performing operation 3
D : is performing operation 3
```

➢ Crearea obiectelor de tip Thread:

- Metoda directa
  - ca subclasa a clasei Thread
  - implementarea interfetei Runnable

- Metoda abstracta
  - folosind clasa Executors

➢ Interfata ExecutorService

```
interface Executor

public interface ExecutorService
extends Executor

public class Executors
extends Object
```

- Serviciul Executor asigura crearea si managementul unei piscine de thread-uri.

```
ExecutorService  executorService = Executors.newSingleThreadExecutor()
```

Crerea unui obiect din clasa **Executors**

```
executorService.execute( instanta Runnable )
pool.execute(instant Runnable)
```

Crearea thread-urilor

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

interface Executor
public interface ExecutorService extends Executor
public class Executors extends Object

Metode:

- **shutdown()**

inchide serviciul, dar permite thread-urilor deja aflate in executie sa termine;

- **shutdownNow()**

  terminarea serviciului, fara a permite finalizarea executiilor;

- **awaitTermination(long timeout, TimeUnit unit)**

  pentru a permite finalizarea executiilor, impunand o limita temporara

Metode pentru crerea unui obiect din clasa **Executors**:

- **newSingleThreadExecutor()**
- **newCachedThreadPool()**
- **newFixedThreadPool(poolSize)**

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

➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public class Interference  implements Runnable {
 static Integer counter = 0;

       public void run () {
          for (int i = 0; i < 5; i++) {
             performTask();
          }}

private synchronized void performTask () {
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread()
                          .getName() + " - before: "+temp+" after:" + counter);}
public static void main (String[] args) {.. }}
```

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

➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public static void main (String[] args) {
      Thread thread1 = new Thread(new Interference());
      Thread thread2 = new Thread(new Interference());
      thread1.start(); thread2.start();
      thread1.join(); thread2.join();  }
```

```
Thread-1 - before: 1 after:2
Thread-1 - before: 2 after:3
Thread-0 - before: 0 after:1
Thread-1 - before: 3 after:4
Thread-0 - before: 4 after:5
Thread-1 - before: 5 after:6
Thread-0 - before: 6 after:7
Thread-1 - before: 7 after:8
Thread-0 - before: 8 after:9
Thread-0 - before: 9 after:10
```

➢ Generarea thread-urilor folosind Executors

```
public static void main (String[] args) {
      Thread thread1 = new Thread(new Interference());
      Thread thread2 = new Thread(new Interference());
      thread1.start(); thread2.start();
      thread1.join(); thread2.join();  }
```

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

public static void main (String[] args) throws InterruptedException  {

      ExecutorService demo = Executors.newCachedThreadPool();
      for(int i=0;i<2;i++) {demo.execute(new Interference());}
      demo.shutdown();
}
```

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

```
public static void main (String[] args) throw InterruptedException  {

    ExecutorService demo = Executors.newCachedThreadPool();
    for(int i=0;i<4;i++) {demo.execute(new Interference());}
    demo.shutdown();
}
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java InterferenceE
pool-1-thread-2 - before: 1 after:3
pool-1-thread-1 - before: 0 after:3
pool-1-thread-1 - before: 5 after:6
pool-1-thread-3 - before: 3 after:4
pool-1-thread-4 - before: 2 after:3
pool-1-thread-1 - before: 6 after:7
pool-1-thread-2 - before: 4 after:5
pool-1-thread-1 - before: 9 after:10
pool-1-thread-4 - before: 8 after:9
pool-1-thread-3 - before: 7 after:8
pool-1-thread-4 - before: 12 after:13
pool-1-thread-1 - before: 11 after:12
pool-1-thread-2 - before: 10 after:11
pool-1-thread-4 - before: 14 after:15
pool-1-thread-3 - before: 13 after:14
pool-1-thread-4 - before: 16 after:17
pool-1-thread-2 - before: 15 after:16
pool-1-thread-3 - before: 17 after:18
pool-1-thread-3 - before: 19 after:20
pool-1-thread-2 - before: 18 after:19
```

Thread-urile sunt numite pool-1-thread-k

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

Exemplu: ReaderWriter - generarea thread-urilor folosind Executors

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

public class ReaderWriterE{
   private static Integer counter = 0;
   private static final ReadWriteLock lock = new ReentrantReadWriteLock();

   public static void main (String[] args)  {
      ExecutorService demo = Executors.newCachedThreadPool();
      demo.execute(new TaskW());
      demo.execute(new TaskR());
      demo.execute(new TaskW());
      demo.execute(new TaskR());
      demo.execute(new TaskR());
      demo.shutdown();
   }
```

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ReaderWriterE
pool-1-thread-1 - before: 0 after:5
pool-1-thread-6 counter:5
pool-1-thread-4 - before: 5 after:10
pool-1-thread-3 counter:10
pool-1-thread-2 counter:10
pool-1-thread-5 counter:10
pool-1-thread-7 - before: 10 after:15

C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java ReaderWriterE
pool-1-thread-1 - before: 0 after:5
pool-1-thread-3 counter:5
pool-1-thread-4 - before: 5 after:10
pool-1-thread-2 counter:10
pool-1-thread-5 counter:10
pool-1-thread-7 - before: 10 after:15
pool-1-thread-6 counter:15
```

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

➢ Callable si Future

```
public interface Runnable {
    public void run();
}
```

executa un thread

```
public interface Callable<ResultType> {
    ResultType call() throws Exception;
}
```

intoarce rezultatul executiei unui thread

```
Callable<String> callable = new Callable<String>() {

public String call() throws Exception {
        // Perform some computation
        Thread.sleep(2000);
        return "Return some result";
    }};

ExecutorService exec=Executor.newSingleThreadExecutor
Future<ResultType> future = exec.submit(callable)
```

un obiect **Callable** intoarce un obiect **Future**

https://www.callicoder.com/java-callable-and-future-tutorial/

➢ Callable si Future

```
Callable<String> callable = new Callable<String>() {

public String call() throws Exception {
        // Perform some computation
        Thread.sleep(2000);
        return "Return some result";
    }};

public static void main (String[] args) throws Exception{

ExecutorService exec=Executor.newSingleThreadExecutor
Future<ResultType> future = exec.submit(callable)
…
}
```

Callable reprezinta o executie **asincrona**, al carei rezultat este recuperate cu ajutorul unui obiect **Future**

Exemplu:

- implementarea unei instante a clasei Callable care intoarce un <String>
- instanta va fi folosita pentru a crea un obiect Future

```
private static class TaskCallable implements Callable<String> {
        private static int ts;
        public  TaskCallable (int ts) {this.ts = ts;}

        public String call () throws InterruptedException {
            System.out.println("Entered Callable; sleep:"+ts);
            Thread.sleep(ts);
            return "Hello from Callable";
        }
    }
```

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String>  futureEx =executorService.submit(newTaskCallable(time));
```

https://www.callicoder.com/java-callable-and-future-tutorial

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

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

public class CallableFuture{

 public static void main (String[] args) throws Exception{

      ExecutorService demo = Executors.newSingleThreadExecutor();
      int time = ThreadLocalRandom.current().nextInt(1000, 5000);
      System.out.println("Creating the future");
      Future<String>  futureEx = demo.submit(new TaskCallable(time));
      System.out.println("Do something else while callable is getting executed");
      Thread.currentThread().sleep(time);
      System.out.println("Retrieve the result of the future");
      String result = futureEx.get();
      System.out.println(result);
      demo.shutdown();
   }
```

recomandat pentru a genera valori aleatoare in aplicatii concurente

https://www.callicoder.com/java-callable-and-future-tutorial/

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

➢ Future

- **ExecutorService.submit()** intoarce imediat, returnand un obiect Future.
  Din acest moment se pot executa diferite task-uri in parallel cu cea executata de obiectul Future.

- Rezultatul returnat de obiectul Future este obtinut apeland **future.get()**.

- Metoda **get()** a obiectelor Future va bloca thread-ul care o apeleaza pana cand se returneaza obiectului Future; daca task-ul executat de obiect este anulat, metoda
  get() arunca exceptie.

- Metoda **isDone()** a obiectelor Future poate fi apelata pentru a vedea daca obiectul si-a terminat de executat task-ul.

```
public static void main (String[] args) throws Exception{
        ExecutorService demo = Executors.newSingleThreadExecutor();
        int time = ThreadLocalRandom.current().nextInt(1000, 5000);
        System.out.println("Creating the future");
        Future<String>  futureEx = demo.submit(new TaskCallable(time));
        System.out.println("Do something else while callable is getting executed");

        while(!futureEx.isDone()) {
            System.out.println("Task is still not done...");
            Thread.sleep(200);
        }
        System.out.println("Retrieve the result of the future");
        String result = futureEx.get();
        System.out.println(result);
        demo.shutdown();
    }
```

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

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java CallableFuture
Creating the future
Do something else while callable is getting executed
Task is still not done...
Entered Callable; sleep:3089
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Retrieve the result of the future
Hello from Callable
```

Callable reprezinta o executie **asincrona**,
al carei rezultat este recuperate cu ajutorul
unui obiect **Future**

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

```
C:\Users\igleu\Documents\DIR\ICLP22\Curs 2022\Java2022\pg>java CallableFuture
Creating the future
Do something else while callable is getting executed
Task is still not done...
Entered Callable; sleep:2601
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Task is still not done...
Retrieve the result of the future
Hello from Callable
```

Callable reprezinta o executie **asincrona**,
al carei rezultat este recuperate cu ajutorul
unui obiect **Future**

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