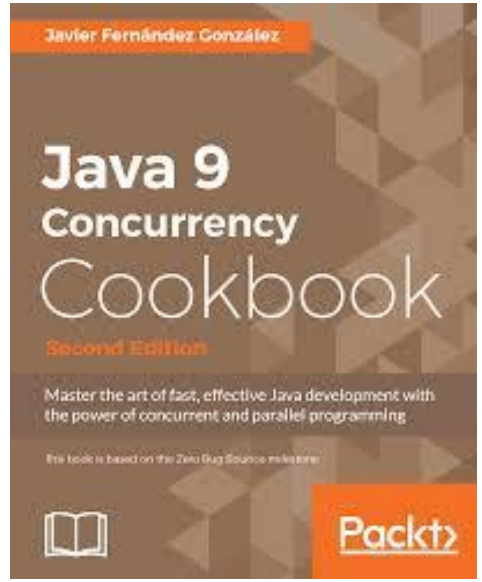
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean



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

➢ Modelul Producator-Consumator

Doua threaduri comunica prin intermediul unui buffer (memorie partajata):

- thread-ul Producator creaza datele si le pune in buffer
- thread-ul Consumator ia datele din buffer si le prelucreaza

Probleme de coordonare:

- Producatorul si consumatorul nu vor accesa bufferul simultan.
- Producatorul nu va pune in buffer date noi daca datele din buffer nu au fost consumate
- Cele doua thread-uri se vor anunta unul pe altul cand starea buferului s-a schimbat

- Modelul Producator-Consumator


```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
    …
  return message;  }

  public synchronized String put(String message) {
    …
  }
```

implementarea buffer-ului:
accesul se face prin metode sincronizate

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

- Thread-ul producator

```
public class PCProducer implements Runnable {
   private PCDrop drop;

   public PCProducer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      String importantInfo[] = {   "Mares eat oats","Does eat oats","Little lambs eat ivy","A kid will eat ivy too"};
      Random random = new Random();

      for (int i = 0;  i < importantInfo.length; i++) {
                                                          drop.put(importantInfo[i]);
                                                          try {
                                                                Thread.sleep(random.nextInt(5000)
                                                             } catch (InterruptedException e) {}
                                                           }
      drop.put("DONE");
   }}
```

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

- Thread-ul consumator

```
import java.util.Random;

public class Consumer implements Runnable {
   private PCDrop drop;

   public  Consumer(PCDrop drop) {
      this.drop = drop;
   }

   public void run() {
      Random random = new Random();
      for (String message = drop.take();  ! message.equals("DONE"); message = drop.take())
                  {
                      System.out.format("MESSAGE RECEIVED: %s%n", message);
                     try {
                           Thread.sleep(random.nextInt(5000));
                          } catch (InterruptedException e) {}
      }
   }
}
```

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

- Modelul Producator-Consumator

- implementarea foloseste *blocuri cu garzi*
- thread-ul este suspendat pana cand
  o anume conditie este satisfacuta

```
public class PCDrop {

   private String message;
   private boolean empty = true;

   public synchronized String take() {
    …
  return message;  }

  public synchronized String put(String message) {
    …
  }
```

```
public synchronized String take() {
      while (empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
       empty = true;  notifyAll();
      return message;
}
```

```
public synchronized void put(String message) {
     while (!empty) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      empty = false;
      this.message = message;
      notifyAll();
   }}
```

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

```
public class ProducerConsumer {
   public static void main(String[] args) {

      PCDrop drop = new PCDrop();

      (new Thread(new Producer(drop))).start();

      (new Thread(new Consumer(drop))).start();
   }
}
```

```
C:\myjava\tutoracle> java producerconsumer/PCExample
MESSAGE RECEIVED: Mares eat oats
MESSAGE RECEIVED: Does eat oats
MESSAGE RECEIVED: Little lambs eat ivy
MESSAGE RECEIVED: A kid will eat ivy too
```

```
interface Lock
interface ReadWriteLock   extends Lock

class Reentrantlock
class ReentrantReadWriteLock

Metode:  lock(), unlock()
```

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

 Lock obLock = new ReentrantLock();
 obLock.lock();
 try {
   //  acceseaza resursa protejata de obLock
} finally {
   obLock.unlock();
 }
```

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, thread-urile pot accesa lacatele in orice ordine; obiectele ReadWriteLock permit accesul simultan pentru mai multe thread-uri.

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

- class ReentrantLock

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

public class ThreadInterference{
   private Integer counter = 0;
   private final ReentrantLock lock = new ReentrantLock();

public static void main (String[] args) {
       ThreadInterference demo = new ThreadInterference();
       Task task1 = demo.new Task();
       Thread thread1 = new Thread(task1);
       Task task2 = demo.new Task();
       Thread thread2 = new Thread(task2);
       thread1.start();
       thread2.start();
   }
  private class Task implements Runnable { …}
  private void perform Task() { …}
}
```

```
   private class Task implements Runnable {
      public void run () {
          for (int i = 0; i < 5; i++) {
              performTask();
          }}}
```

```
private void performTask () {
       lock.lock();
       try{
       int temp = counter;
       counter++;
       System.out.println(Thread.currentThread().getName()
                       + " - before: "+temp+" after:" + counter);
       }
       finally {lock.unlock();}
    }
```

https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-thread-synchronization.html

➢ Interface Condition

- implementeaza metode asemanatoare cu wait(), notify() si notifyall() pentru obiectele din clasa Lock

await(), **cond.await(long time, TimeUnit unit)**
thread-ul current intra in asteptare

signall()
un singur thread care asteapta este trezit

signalAll()
toate thread-urile care asteapta sunt trezite

- Conditiile sunt legate de un obiect Lock! Pot exista mai multe conditii pentru acelasi obiect Lock.

```
Lock obiectLock = new ReentrantLock();
Condition condVar = obiectLock.newCondition();
```

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

➢ Exemplul Producator-Consumator in care folosim obiecte Lock in locul metodelor sincronizate

```
public class PCDrop1 {

   private String message;
   private boolean empty = true;
   private Lock dropLock = new ReentrantLock();
   private Condition condVar = dropLock.newCondition();

   public String take() {
    …
  return message;  }

  public String put(String message) {
    …
  }
```

- Exemplul Producator-Consumator in care folosim obiecte Lock in locul metodelor sincronizate

```
public String take() {
      dropLock.lock();
      try{
      while (empty) {
         try {
            condVar.await();
         } catch (InterruptedException e) {}
      }

      empty = true;
      condVar.signalAll();
      return message;}
      finally { dropLock.unlock(); }
   }
```

```
public  void put(String message) {
    dropLock.lock();
     try{
       while (!empty) {
          try {
             condVar.await();
          } catch (InterruptedException e) {}
       }
       empty = false;
       this.message = message;
       condVar.signalAll();
     }
     finally {dropLock.unlock();}
    }
```

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

➢ Modelul de interactiuni Cititori-Scriitori

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

Class ReadWriteLock

readLock()
intoarce lacatul pentru cititori

writeLock()
intoarce lacatul pentru scriitori

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

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

public class ThreadInterferenceExecRW{
   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();
   }
```

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();}

       }
   }
```

```
C:\myjava\tutoracle>java ThreadInterferenceExecRW
pool-1-thread-1 - before: 0 after:5
pool-1-thread-4 counter:5
pool-1-thread-3 - before: 5 after:10
pool-1-thread-5 counter:10
pool-1-thread-2 counter:10
```

```
C:\myjava\tutoracle>java ThreadInterferenceExecRW
pool-1-thread-1 - before: 0 after:5
pool-1-thread-2 counter:5
pool-1-thread-4 counter:5
pool-1-thread-3 - before: 5 after:10
pool-1-thread-5 counter:10
```

```
C:\myjava\tutoracle>java ThreadInterferenceExecRW
pool-1-thread-1 - before: 0 after:5
pool-1-thread-3 - before: 5 after:10
pool-1-thread-2 counter:10
pool-1-thread-4 counter:10
pool-1-thread-5 counter:10
```

public class Semaphore
extends Object

Semaphore(int permits) // constructor

Implementeaza un semafor cu cantitate (quantity semaphore) care coordoneaza accesul la un numar precizat de resurse

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

sem.acquire();

...  //sectiune critica


sem.release();
```

semaphore.aquire()

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

semaphore.release()

thread-ul care apeleaza release elibereaza accesul la o resursa

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; cream 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 MyATMThread("A");
                 t1.start();
                 …..
                MyThread t4 = new MyATMThread("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:\myjava\tutoracle>java Semaphores
Total available Semaphore permits : 2
C : acquiring lock...
A : acquiring lock...
B : acquiring lock...
D : acquiring lock...
A : got the permit!
C : got the permit!
C : is performing operation 1, available Semaphore permits : 0
A : is performing operation 1, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
A : is performing operation 2, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
A : is performing operation 3, available Semaphore permits : 0
C : releasing lock...
A : releasing lock...
B : got the permit!
D : got the permit!
D : is performing operation 1, available Semaphore permits : 0
B : is performing operation 1, available Semaphore permits : 0
A : available Semaphore permits now: 0
C : available Semaphore permits now: 1
D : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
D : is performing operation 3, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
D : releasing lock...
D : available Semaphore permits now: 1
```

Doua modalitati de a crea obiecte de tip Thread:

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

- abstracta
- folosind metodele clasei Executors

interface Executor

public interface ExecutorService
extends Executor

public class Executors
extends Object

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

Metodele **sthudown()** si **shutdownNow()** sunt folosite pentru terminarea serviciului; **shutdown()** permite thread-urilor deja aflate in executie sa termine; **shutdown()** poate fi folosita impreuna cu **awaitTermination(long timeout, TimeUnit unit)** pentru a limita timpul de asteptare

Crerea unui obiect din clasa Executors

```
ExecutorService  executorService = Executors.newSingleThreadExecutor()
ExecutorService   pool = Executors.newCachedThreadPool()
ExecutorService   poolFixed = Executors.newFixedThreadPool(poolSize)
```

Crearea thread-urilor

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

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

```
public static void main (String[] args)  {
        ThreadInterference demo = new ThreadInterference();
        Task task1 = demo.new Task();
        Thread thread1 = new Thread(task1);

        Task task2 = demo.new Task();
        Thread thread2 = new Thread(task2);

        thread1.start();
        thread2.start();
    }
:
private  class Task implements Runnable {…}
private  synchronized void perform Task() { …}
```

```
public class ThreadInterferenceExec{
    private static Integer counter = 0;

    public static void main (String[] args)  {
        ExecutorService demo = Executors.newCachedThreadPool();
        for(int i=0;i<2;i++) {demo.execute(new Task());}
        demo.shutdown();
    }
```

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

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

```
public class ThreadInterferenceExec{
   private static Integer counter = 0;

   public static void main (String[] args)  {
      ExecutorService demo = Executors.newCachedThreadPool();
      for(int i=0;i<4;i++) {demo.execute(new Task());}
      demo.shutdown();
   }
```

```
C:\myjava\tutoracle>java ThreadInterferenceExec
pool-1-thread-1 - before: 0 after:1
pool-1-thread-1 - before: 1 after:2
pool-1-thread-1 - before: 2 after:3
pool-1-thread-1 - before: 3 after:4
pool-1-thread-1 - before: 4 after:5
pool-1-thread-4 - before: 5 after:6
pool-1-thread-4 - before: 6 after:7
pool-1-thread-4 - before: 7 after:8
pool-1-thread-4 - before: 8 after:9
pool-1-thread-4 - before: 9 after:10
pool-1-thread-3 - before: 10 after:11
pool-1-thread-3 - before: 11 after:12
pool-1-thread-3 - before: 12 after:13
pool-1-thread-3 - before: 13 after:14
pool-1-thread-3 - before: 14 after:15
pool-1-thread-2 - before: 15 after:16
pool-1-thread-2 - before: 16 after:17
pool-1-thread-2 - before: 17 after:18
pool-1-thread-2 - before: 18 after:19
pool-1-thread-2 - before: 19 after:20
```

Se creaza un thread pentru fiecare task
Thread-urile sunt numite
pool-1-thread-k

Metode pentru crearea executorilor:

Executors.newCachedThreadPool() oricate thread-uri
Executors.newFixedThreadPool(int n) n thread-uri
Executors.newSingleThreadExecutor un singur thread

➢ Variabile atomice

sunt implementate folosind instructiuni compare-and-swap care sunt mai rapide

public class AtomicInteger
extends Number

Metode:
get(), set(),
incrementAndGet()
addAndGet(int d)
compareAndSet(int old, int new)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;


public class ThreadInterferenceAtomic{
   private static AtomicInteger counter = new AtomicInteger(0);
   public static void main (String[] args)  {…}

  private static class Task implements Runnable {
            public void run () { counter.incrementAndGet();}
   }}
```

```
public static void main (String[] args)  {
 ExecutorService demo = Executors.newFixedThreadPool(1000);
for(int i=0;i<1000;i++) {demo.execute(new Task());}
       demo.shutdown();
       System.out.println("Final Count is : " + counter.get());
   }
```

https://www.callicoder.com/java-locks-and-atomic-variables-tutorial/

➢ Callable si Future

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

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

```
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)
```

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

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

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

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

public class FutureExample{

 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));
      Thread.currentThread().yield();
      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

ThreadLocalRandom.current().nextInt(1000, 5000)

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

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

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

public class FutureExample{

  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));
       Thread.currentThread().yield();
       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();
    }
```

```
C:\myjava\tutoracle>java FutureExample
Do something else while callable is getting executed
Entered Callable; sleep:3535
Retrieve the result of the future
Hello from Callable
```

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));
        Thread.currentThread().yield();
        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();
    }
```

```
C:\myjava\tutoracle>java FutureExample2
Creating the future
Do something else while callable is getting executed
Task is still not done...
Entered Callable; sleep:3401
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...
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
```

Metoda **invokeAll**

- este apelata pentru o colectie de task-uri care se executa in paralel
- intoarce **o lista de obiecte Future**
- orice apel get() pentru un obiect din colectie, va intoarce numai dupa ce toate s-au terminat

```
List<Callable<String>>   taskList = Arrays.asList(callable, callable, callable);
List<Future<String>>    futures = executorService.invokeAll(taskList);
```

Metoda **invokeAny**

- este apelata pentru o colectie de task-uri care se executa in paralel
- intoarce **primul rezultat** obtinut

```
Callable<String>  task1, task2, task3;
String result = executorService.invokeAny(Arrays.asList(task1, task2, task3))
```

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

public class  FutureExampleAr {
   public static void main(String[] args) throws InterruptedException, ExecutionException {
      ExecutorService executorService = Executors.newFixedThreadPool(5);
       Callable<String> callable = () -> {
                              int ts= ThreadLocalRandom.current().nextInt(1000, 5000);
                              System.out.println("Entered Callable; sleep:"+ts);
                              Thread.sleep(ts);
                              return "Hello from Callable";
                              };
 List<Callable<String>> taskList = Arrays.asList(callable, callable, callable);
 List<Future<String>>  futures = executorService.invokeAll(taskList);

  for(Future<String> future: futures) {System.out.println(future.get());}

      executorService.shutdown();
  } }
```

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

```
C:\myjava\tutoracle>java FutureExampleAr
Entered Callable; sleep:1181
Entered Callable; sleep:4436
Entered Callable; sleep:4333
```

```
C:\myjava\tutoracle>java FutureExampleAr
Entered Callable; sleep:1181
Entered Callable; sleep:4436
Entered Callable; sleep:4333
Hello from Callable
Hello from Callable
Hello from Callable
```