IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

➢ Crearea obiectelor de tip Thread:

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

public class Thread
extends Object
implements Runnable
```

- Metoda directa
  - ca subclasa a clasei Thread
  - implementarea interfetei Runnable
- Metoda abstracta
  - folosind interfata Executor

```
interface Executor

public interface ExecutorService
extends Executor

public class Executors
extends Object
```

- Framework-ul **Executor**

interface Executor
public interface ExecutorService
extends Executor

public class Executors
extends Object

- ExecutorService asigura crearea si managementul unei piscine de thread-uri.

```
ExecutorService  pool = Executors.newCachedThreadPool()

pool.execute( instanta Runnable )
```

Crearea thread-urilor

crerea unui piscine de thread-uri folosind o metoda a clasei **Executors (**care intoarce o instanta a interfetei **ExecutorService)**

https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/util/concurrent/Executors.html

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

- Metode ale clasei **Executors**:

- **newSingleThreadExecutor()**

"Creates an Executor that uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.)**"**

  - Un thread (normal) executa un singur task, dar un thread creat cu aceasta metoda poate executa secvential o serie de task-uri.

- **newCachedThreadPool()**

"Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks."

- **newFixedThreadPool(poolSize)**

"Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most n Threads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available."

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

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

- Metode ale clasei **Executors**:
  - **newSingleThreadExecutor()**
  - **newCachedThreadPool()**
  - **newFixedThreadPool(poolSize)**

- Metode ale interfetei **ExecutorService**
  - **shutdown()**
    serviciul nu primeste task-uri noi, dar executa task-urile deja primite
  - **shutdownNow()**
    terminarea serviciului, fara a permite finalizarea executiilor
  - **awaitTermination(long timeout, TimeUnit unit)**
    pentru a permite finalizarea executiilor, impunand o limita temporara

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

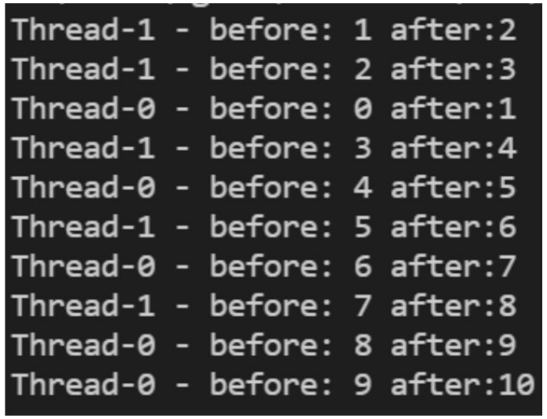
➢ Metode sincronizate

- doua thread-uri care incrementeaza acelasi contor

```
public class Task  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) {.. }}
```

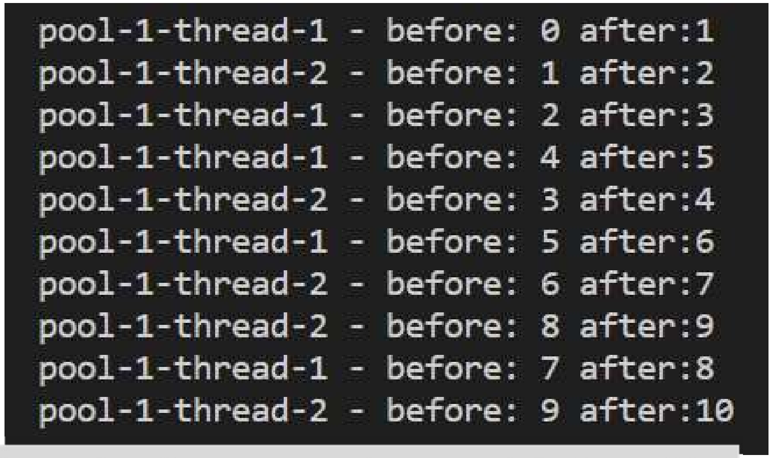
- Generarea thread-urilor folosind Executor

```
public static void main (String[] args) {
      Thread thread1 = new Thread(new Task());
      Thread thread2 = new Thread(new Task());
      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
```

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

public static void main (String[] args)  {

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

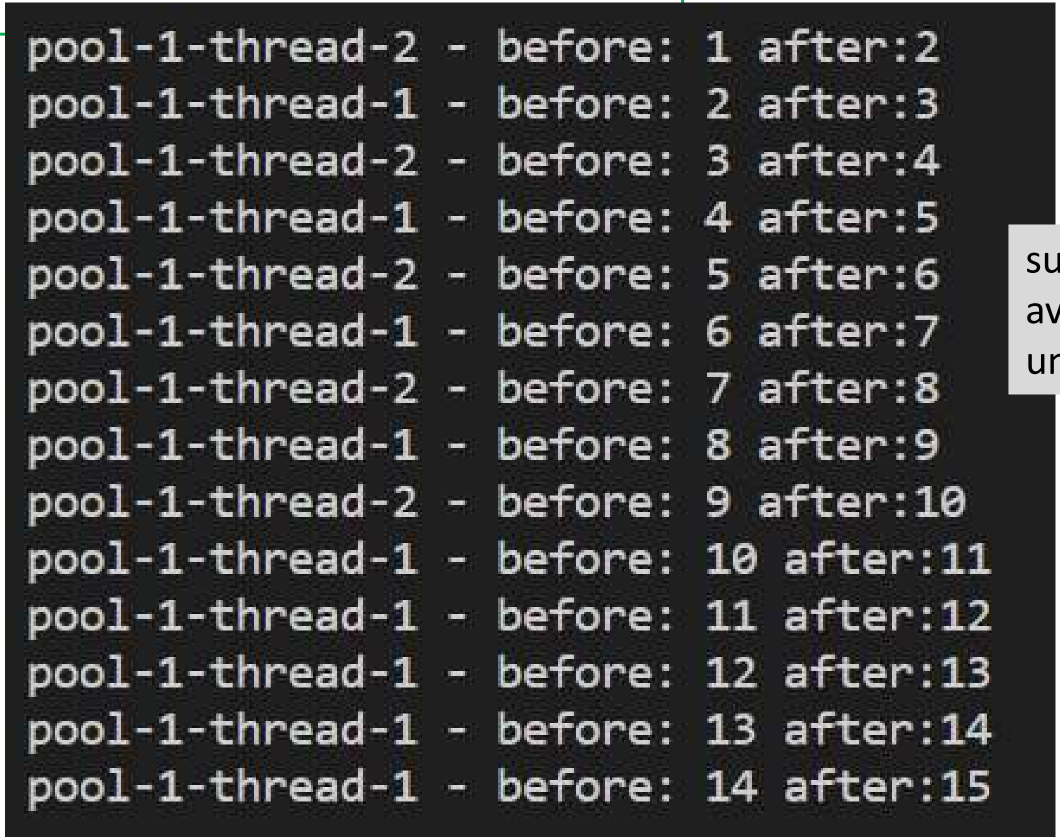


```
pool-1-thread-1 - before: 0 after:1
pool-1-thread-2 - before: 1 after:2
pool-1-thread-1 - before: 2 after:3
pool-1-thread-1 - before: 4 after:5
pool-1-thread-2 - before: 3 after:4
pool-1-thread-1 - before: 5 after:6
pool-1-thread-2 - before: 6 after:7
pool-1-thread-2 - before: 8 after:9
pool-1-thread-1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

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

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

```
  public static void main (String[] args)  {
      ExecutorService pool = Executors.newFixedThreadPool(2);
      for(int i=0;i<3;i++) {pool.execute(new Task());}
      demo.shutdown();
}
```

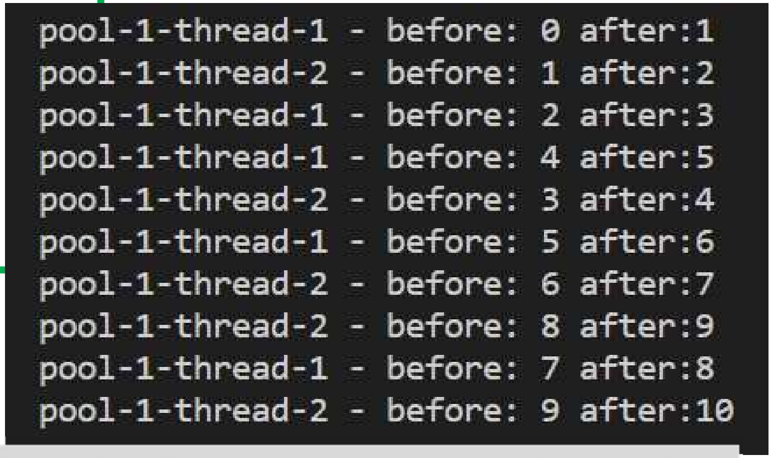


```
pool-1-thread-2 - before: 1 after:2
pool-1-thread-1 - before: 2 after:3
pool-1-thread-2 - before: 3 after:4
pool-1-thread-1 - before: 4 after:5
pool-1-thread-2 - before: 5 after:6
pool-1-thread-1 - before: 6 after:7
pool-1-thread-2 - before: 7 after:8
pool-1-thread-1 - before: 8 after:9
pool-1-thread-2 - before: 9 after:10
pool-1-thread-1 - before: 10 after:11
pool-1-thread-1 - before: 11 after:12
pool-1-thread-1 - before: 12 after:13
pool-1-thread-1 - before: 13 after:14
pool-1-thread-1 - before: 14 after:15
```

sunt create 2 thread-uri, dar
avem 3 task-uri, deci
un thread executa 2 task-uri

➢ shutdown() cu awaitTermination()

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

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


      ExecutorService  pool = Executors.newCachedThreadPool();
      for(int i=0;i<2;i++) {pool.execute(new Task());}
      pool.shutdown();
       try {
          if (!pool.awaitTermination(3500, TimeUnit.MILLISECONDS)) {
                         pool.shutdownNow();  }
      } catch (InterruptedException e) { pool.shutdownNow();}
 }
```

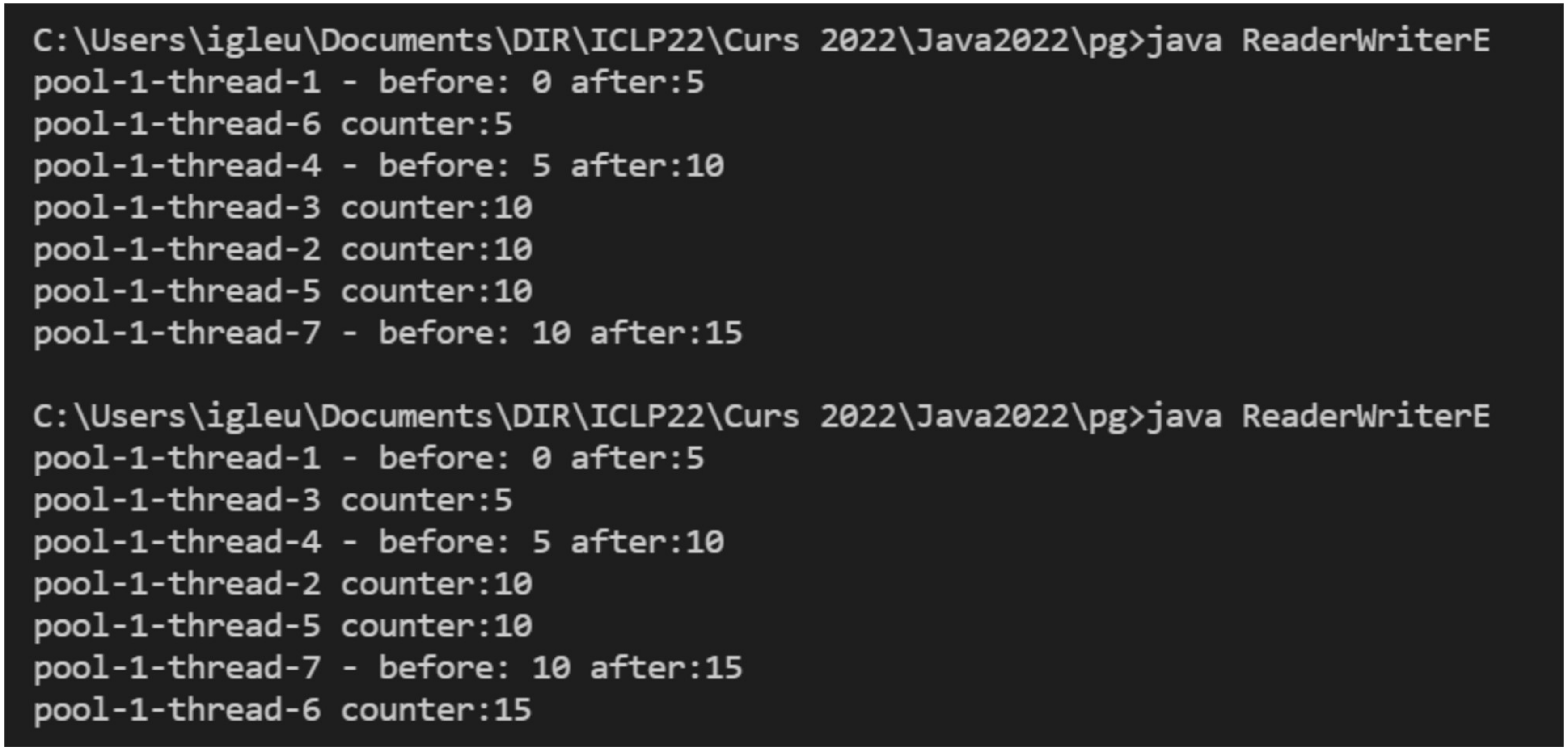


```
pool-1-thread-1 - before: 0 after:1
pool-1-thread-2 - before: 1 after:2
pool-1-thread-1 - before: 2 after:3
pool-1-thread-1 - before: 4 after:5
pool-1-thread-2 - before: 3 after:4
pool-1-thread-1 - before: 5 after:6
pool-1-thread-2 - before: 6 after:7
pool-1-thread-2 - before: 8 after:9
pool-1-thread-1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

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

Exemplu: ReaderWriter - generarea thread-urilor folosind Executor

```
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 pool = Executors.newCachedThreadPool();
        pool.execute(new TaskW());
        pool.execute(new TaskR());
        pool.execute(new TaskW());
        pool.execute(new TaskR());
        pool.execute(new TaskR());
        pool.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<ResultType> callable = new Callable<ResultType>() {

public String call() throws Exception {
        // executie care dureaza
      return 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<String>  future = exec.submit(callable) ;
…
}
```

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

➢ Executie asincrona

- 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";
        }
   }
```

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

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

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

- 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 este anulat sau thread-ul current este intrerupt, metoda get() arunca exceptii.

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

```
import java.util.concurrent.*;
public class CallableFuture{

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

        ExecutorService pool = Executors.newSingleThreadExecutor();

        int time = ThreadLocalRandom.current().nextInt(1000, 5000);

        System.out.println("Creating the future");
        Future<String>  futureEx = pool.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);

        pool.shutdown();        }
```

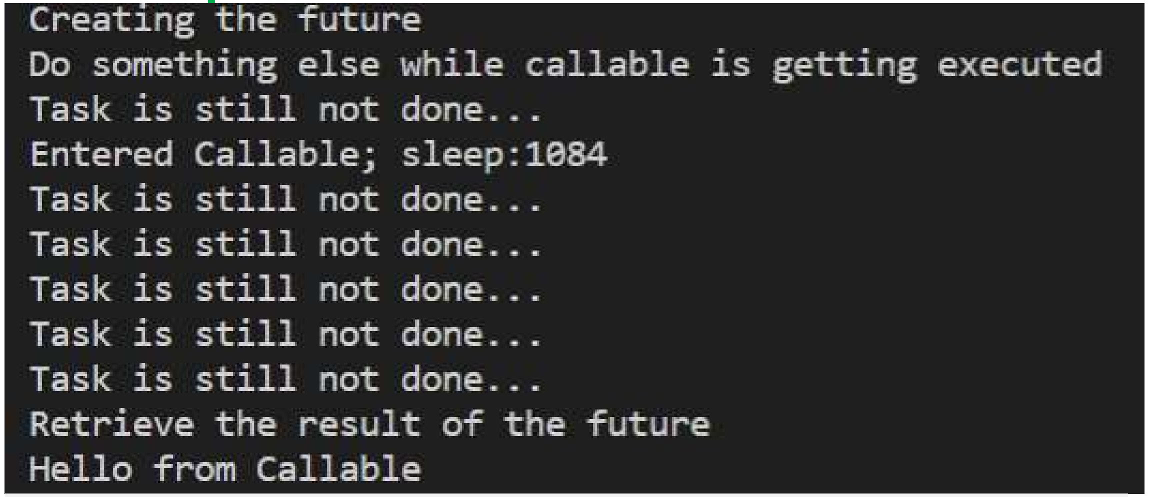
https://www.callicoder.com/java-callable-and-future-tutorial/

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

```
public static void main (String[] args) throws Exception{
    ExecutorService pool = Executors.newSingleThreadExecutor();
    int time = ThreadLocalRandom.current().nextInt(1000, 5000);
    System.out.println("Creating the future");

    Future<String>  futureEx = pool.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);

    pool.shutdown();
  }
```

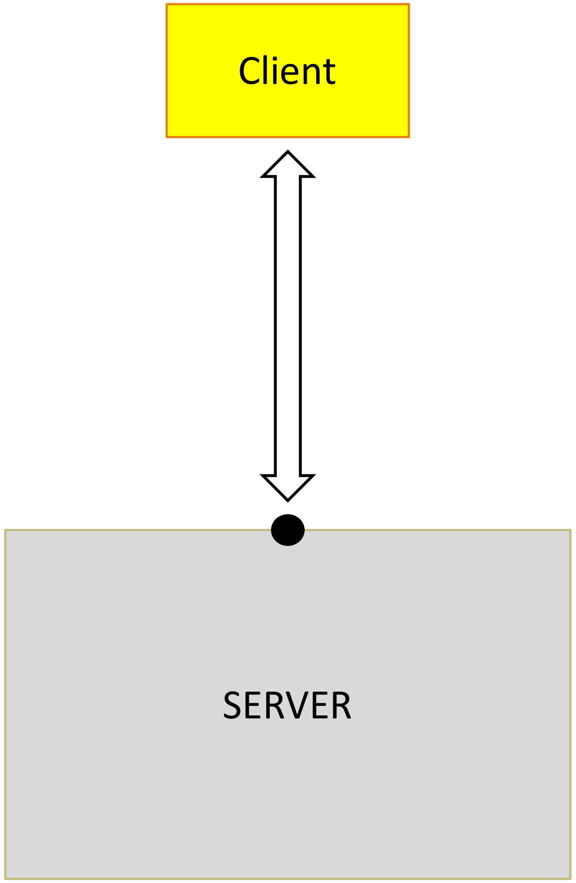
https://docs.oracle.com/javase/tutorial/essential/concurrency

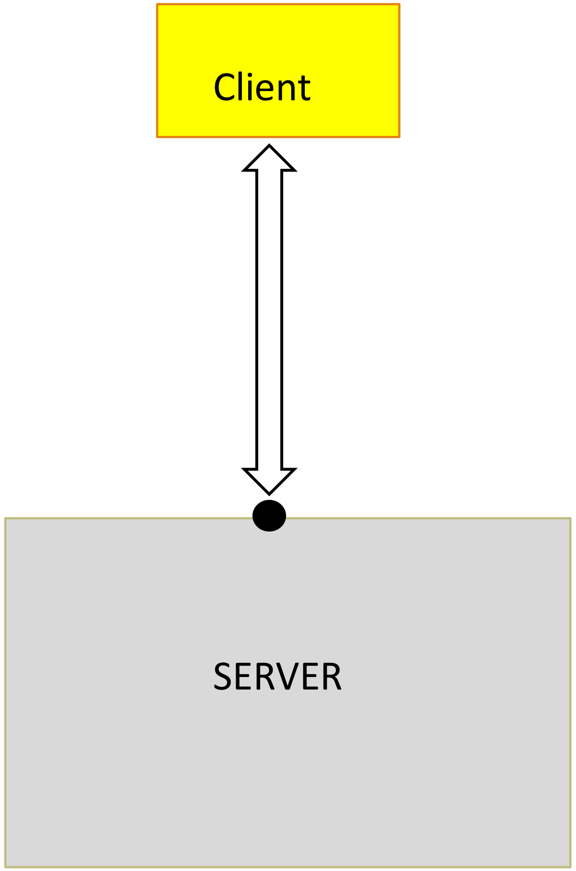
```
public static void main (String[] args) throws Exception{
        ExecutorService pool = Executors.newSingleThreadExecutor();
        int time = ThreadLocalRandom.current().nextInt(1000, 5000);
        System.out.println("Creating the future");

        Future<String>  futureEx = pool.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);

        pool.shutdown();
    }
```



```
Creating the future
Do something else while callable is getting executed
Task is still not done...
Entered Callable; sleep:1084
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**


Client
SERVER

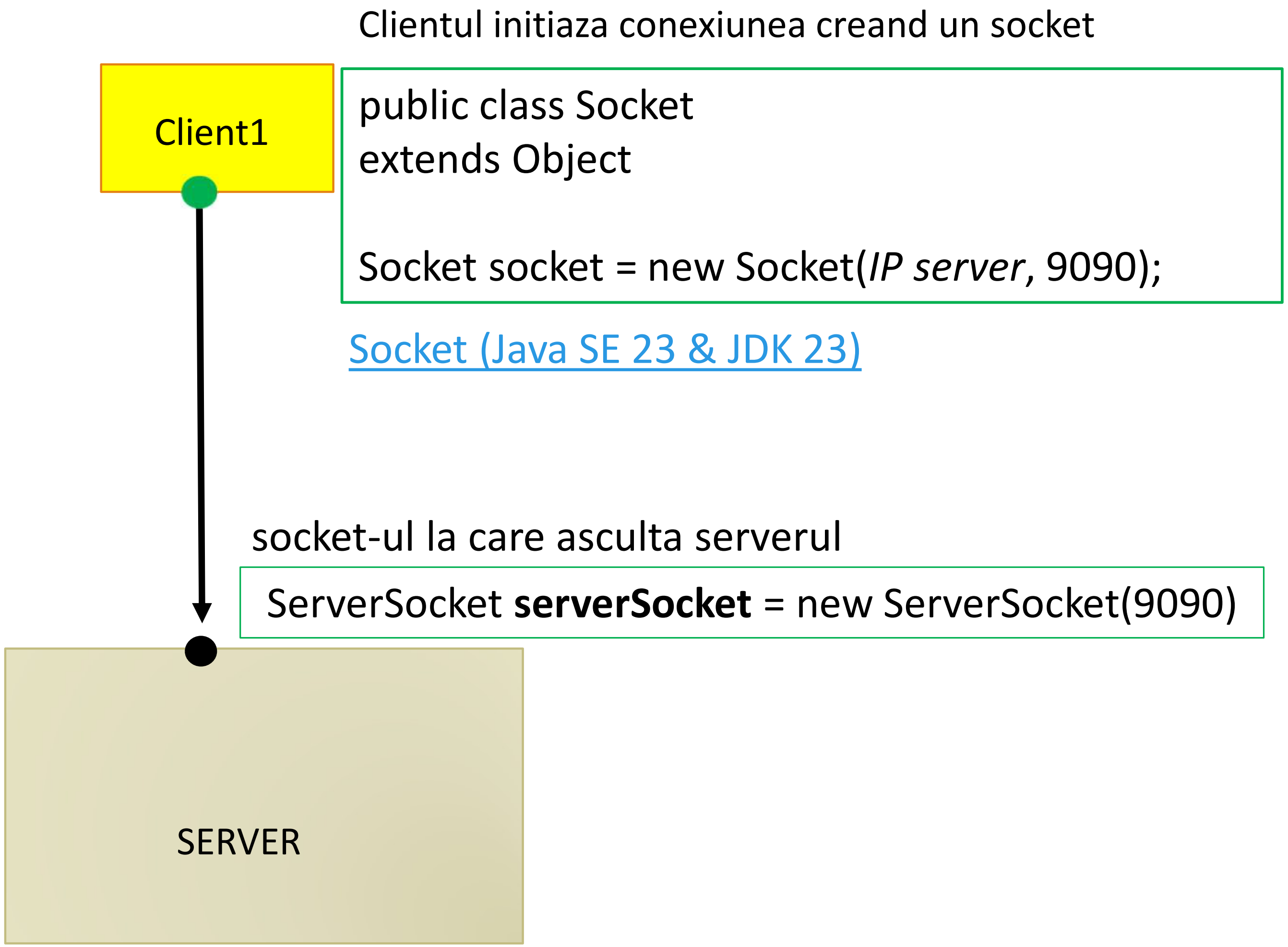


- un socket este un punct final in comunicarea bidirectionala dintre doua programe din aceeasi retea
- un socket are asociat un port

● un socket de server asteapta cererile venite din retea

```
public class ServerSocket
    extends Object

ServerSocket serverSocket = new ServerSocket(9090)
```

ServerSocket (Java SE 23 & JDK 23)



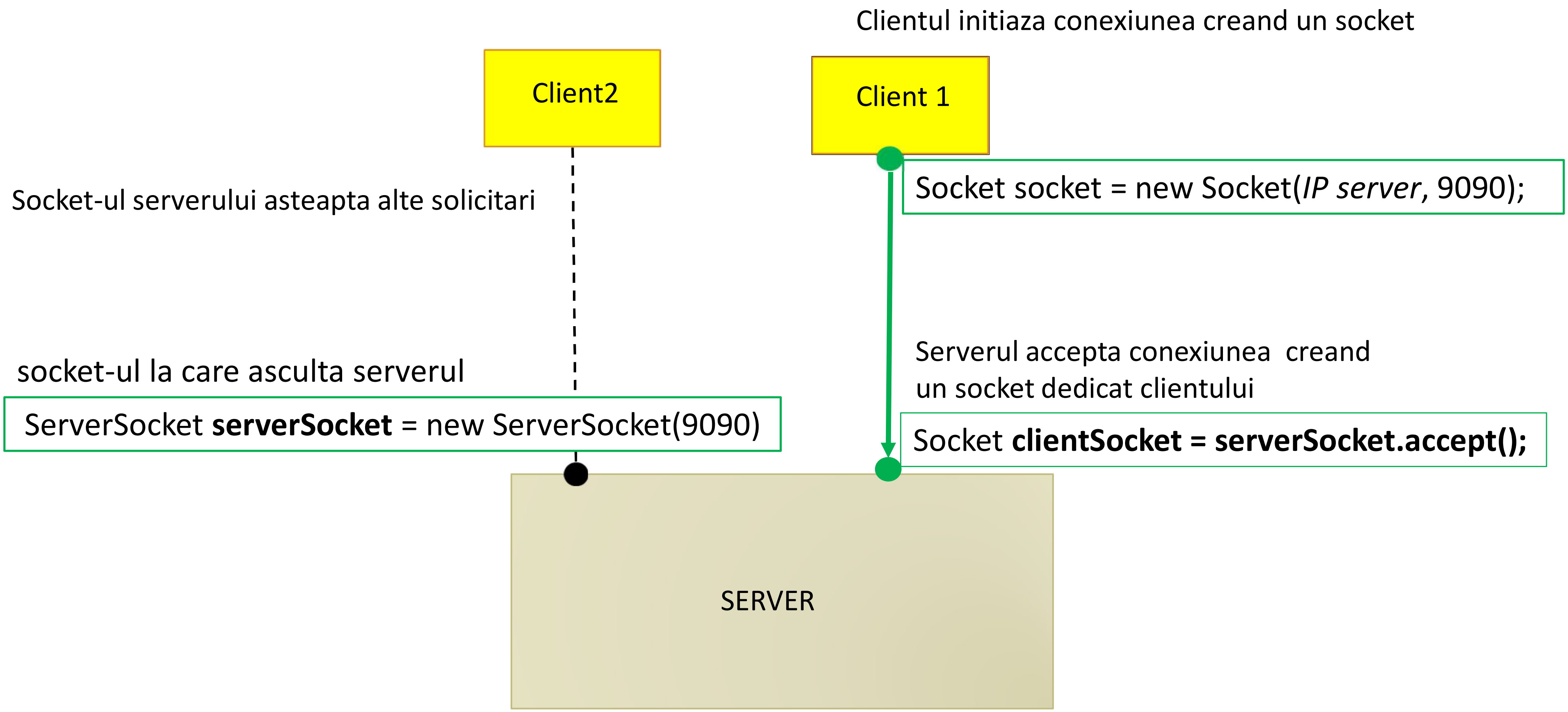
Clientul initiaza conexiunea creand un socket

```
public class Socket
extends Object

Socket socket = new Socket(IP server, 9090);
```

[Socket (Java SE 23 & JDK 23)]()

socket-ul la care asculta serverul

```
ServerSocket serverSocket = new ServerSocket(9090)
```



Clientul initiaza conexiunea creand un socket

Client2

Client 1

Socket socket = new Socket(*IP server*, 9090);

Socket-ul serverului asteapta alte solicitari

socket-ul la care asculta serverul

ServerSocket **serverSocket** = new ServerSocket(9090)

Serverul accepta conexiunea creand
un socket dedicat clientului

Socket **clientSocket = serverSocket.accept();**

SERVER

```
public class Server {
   public static void main(String args[])
   {  ServerSocket serverSocket = new ServerSocket(9090);
      Socket clientSocket = serverSocket.accept();
                                                                    // comunicarea cu clientul
      BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
... }
```

prin **in** si **out** stabilesc canalele de comunicare

```
public class Client {
   public static void main(String args[])
   { Socket socket = new Socket("localhost", 9090);
                                                                    // comunicarea cu serverul
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
     BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
...}
```

https://www.geeksforgeeks.org/how-to-create-a-simple-tcp-client-server-connection-in-java/
Learning Network Programming with Java , R. M. Reeese, 2015

```
public class Server {
    public static void main(String args[])
    {  ServerSocket serverSocket = new ServerSocket(9090);
        System.out.println("Server is running.");              //mesaj afisat pe propriul canal
        Socket clientSocket = serverSocket.accept();
        System.out.println("Client connected!");             //mesaj afisat pe propriul canal

        BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

         String message = in.readLine();
         System.out.println("From client: " + message);  //mesaj afisat pe propriul canal

         out.println("Message received!");             // mesaj trimis clientului

        clientSocket.close();
        serverSocket.close();
}
```

```
public class Client {
   public static void main(String args[]) throws IOException
   { Socket socket = new Socket("localhost", 9090);

      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

       Scanner clientin = new Scanner(System.in);
       String message = clientin.nextLine();
       out.println(message);    // mesaj trimis serverului

       String response = in.readLine();   // mesaj primit de la server
       System.out.println("From server: " + response);

       socket.close();
}
```

```
>Java Server
Sever is running.
Client connected!
From client: buna!
```

```
>java Client
buna!
From server: Message received!
```

```
public class Client {
   public static void main(String args[]) throws IOException
   { Socket socket = new Socket("localhost", 9090);

      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

       Scanner clientin = new Scanner(System.in);
       String message = clientin.nextLine();
       out.println(message);    // mesaj trimis serverului

       String response = in.readLine();   // mesaj primit de la server
       System.out.println("From server: " + response);

       socket.close();
}
```

```
>Java Server
Sever is running.
Client connected!
From client: buna!
>
```

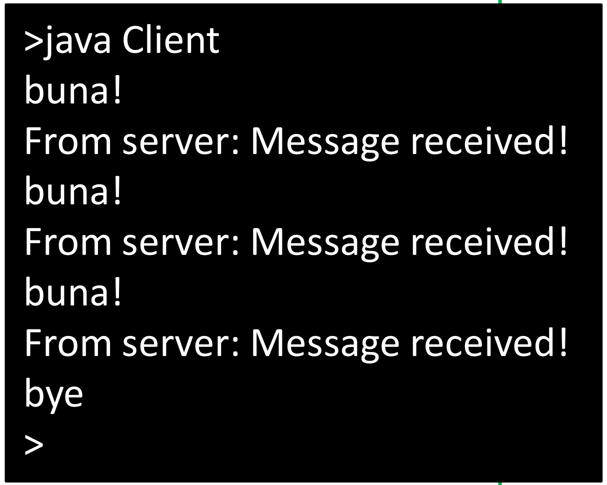
```
>java Client
buna!
From server: Message received!
>java Client
Exception ....
```

```
public class Server {
   public static void main(String args[]) throws IOException
   {  ServerSocket serverSocket = new ServerSocket(9090);
      System.out.println("Server is running");
   while (true){
      Socket clientSocket = serverSocket.accept();
     System.out.println("Client connected!");

      BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

       String message = in.readLine();
       System.out.println("From client: " + message);
       out.println("Message received!");

    clientSocket.close();
}}
```

```
>Java Server
Sever is running
Client connected!
From client: buna!
Client connected!
From client: buna din nou!
|
```

```
>java Client
buna!
From server: Message received!
>java Client
buna din nou!
From server: Message received!
>
```

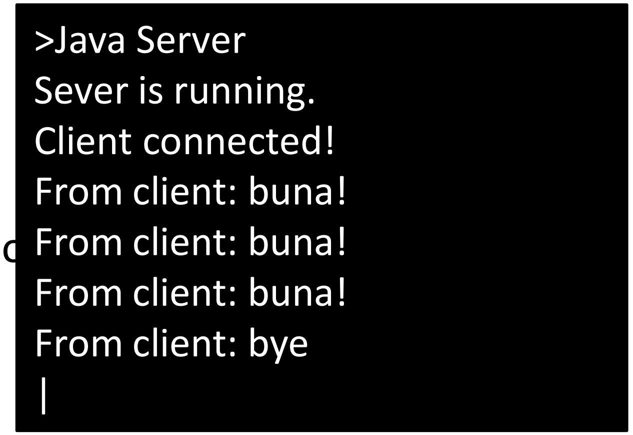
```
public class Client {
   public static void main(String args[]) throws IOException
   { Socket socket = new Socket("localhost", 9090);

      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

       Scanner clientin = new Scanner(System.in);
       String message = clientin.nextLine();

      while (!message.equals("bye")) {
       // Receive response from the server
       String response = in.readLine();
       System.out.println("From server: " + response);
       message = clientin.nextLine();
       out.println(message);
   }
       socket.close();
}
```



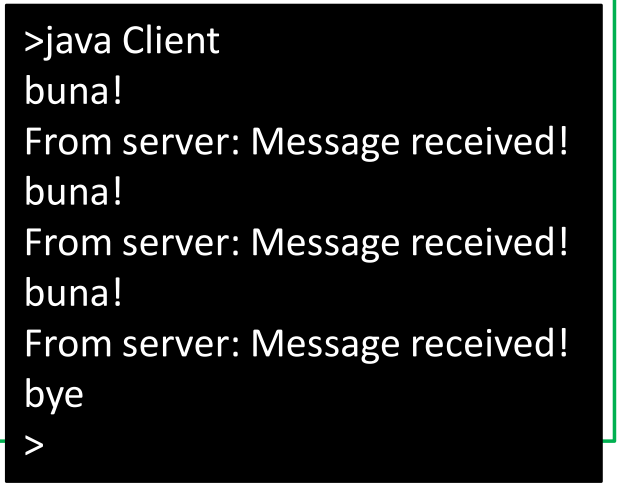
```
>java Client
buna!
From server: Message received!
buna!
From server: Message received!
buna!
From server: Message received!
bye
>
```

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

```
public class Server {
   public static void main(String args[]) throws IOException
   {  ServerSocket serverSocket = new ServerSocket(9090);
      System.out.println("Server is running");
   while (true){
      Socket clientSocket = serverSocket.accept();
     System.out.println("Client connected!");
      BufferedReader in = new BufferedReader(new InputStreamReader(clientSc
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

       String message = in.readLine();
       System.out.println("From client: " + message);
       while (! message.equals("bye")) {
             out.println("Message received!");
              message = in.readLine();
             System.out.println("From client: " + message);
            }
    clientSocket.close();
}}
```



```
>Java Server
Sever is running.
Client connected!
From client: buna!
From client: buna!
From client: buna!
From client: bye
|
```



```
>java Client
buna!
From server: Message received!
buna!
From server: Message received!
buna!
From server: Message received!
bye
>
```

```
public class Server {
   public static void main(String args[]) throws IOException
   {  ServerSocket serverSocket = new ServerSocket(9090);
      System.out.println("Server is running");
   while (true){
      Socket clientSocket = serverSocket.accept();
      System.out.println("Client connected!");
      BufferedReader in = new BufferedReader(new In                                InputStream()));
      PrintWriter out = new PrintWriter(clientSocket.g

       String message = in.readLine();
       System.out.println("From client: " + message);
       while (! message.equals("bye")) {
              out.println("Message received!");
               message = in.readLine();
              System.out.println("From client: " + message);
            }
    clientSocket.close();
}}
```

```
>java Client
buna!
From server: Message received!
buna!
From server: Message received!
bye
>
```

```
>java Client
hi!
From server: Message received!
```

```
>Java Server
Sever is running.
Client connected!
From client: buna!
From client: buna!
From client: bye
Client connected!
From client: hi!
|
```

Clientii sunt serviti secvential!

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

```
public class ServerMT implements Runnable {
private Socket clientSocket;

public static void main(String args[])
    {  ServerSocket serverSocket = new ServerSocket(9090);
        System.out.println("Server is running");
        while (true){
        Socket cSocket = serverSocket.accept();
        System.out.println("Client connected!");
        Thread clientThread = new Thread (new ServerMT (cSocket));
        clientThread.start(); }}

 public ServerMT(Socket s){this.clientSocket =s;}

  public void run()  {
        BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
         String message = in.readLine();  System.out.println("From client: " + message);
         while (! message.equals("bye")) {
          out.println("Message received!");  message = in.readLine(); System.out.println("From client: " + message);
           }
        clientSocket.close();}}
```

main si **run** trebuie scrise cu try-catch

am creat **cate un thread** pentru fiecare client

```
public class ServerMT implements Runnable {
private Socket clientSocket;

public static void main(String args[])
   {  ServerSocket serverSocket = new ServerSocket(9090);
      System.out.println("Server is running");
      while (true){
      Socket cSocket = serverSocket.accept();
      System.out.println("Client connected!");
      Thread clientThread = new Thread (new ServerMT (cSocket));
      clientThread.start(); }}

 public ServerMT(Socket s){this.clientSocket =s;}

  public void run()  {
      BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
       String message = in.readLine();  System.out.println("From client: " + message);
       while (! message.equals("bye")) {
 out.println("Message received!");  message = in.readLine(); System.out.println("From client: " + mes
  }
      clientSocket.close();}}
```

```
>java Client
hi!
From server: Message received!
hi!
From server: Message received!
bye
>
```

```
>java Client
buna!
From server: Messag
buna!
From server: Message received!
buna!
From server: Message received!
bye
>
```

```
a Server
er is running.
Client connected!
From client: buna!
From client: buna!
Client connected!
From client: hi!
From client: buna!
From client: hi!
From client: bye
From client: bye
```

Clientii sunt serviti **concurent**!

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

```
public class ServerMT implements Runnable {
private Socket clientSocket;

public static void main(String args[])
   {  ServerSocket serverSocket = new ServerSocket(9090);
       System.out.println("Server is running");
       ExecutorService  pool = Executors.newCachedThreadPool();

       while (true){
       Socket cSocket = serverSocket.accept();
       System.out.println("Client connected!");
       pool.execute(new ServerMT (cSocket)); }}

 public ServerMT(Socket s){this.clientSocket =s;}

  public void run()  {
       BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
       PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
        String message = in.readLine();  System.out.println("From client: " + message);
        while (! message.equals("bye")) {
         out.println("Message received!");  message = in.readLine(); System.out.println("From client: " + message);}
        clientSocket.close();}}
```

main si **run** trebuie scrise cu try-catch

am creat o piscina **de thread-uri**

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

➢ Fork-Join Framework

```
public class ForkJoinPool
extends AbstractExecutorService
```

```
class AbstractExecutorService
extends Object
implements ExecutorService
```

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

Java 9 Concurrency Cookbook

> The core of the fork/join framework is formed by the following two classes: ForkJoinPool: This class implements the ExecutorService interface and the work-stealing algorithm. It manages the worker threads and offers information about the status of the tasks and their execution.
>
> ForkJoinTask: This is the base class of the tasks that will execute in the ForkJoinPool. It provides the mechanisms to execute the fork() and join() operations inside a task and the methods to control the status of the tasks. Usually, to implement your fork/join tasks, you will implement a subclass of three subclasses of this class: RecursiveAction for tasks with no return result, RecursiveTask for tasks that return one result, and CountedCompleter for tasks that launch a completion action when all the subtasks have finished.

Documentatie:
ForkJoinPool (Java SE 23 & JDK 23)
https://www.researchgate.net/publication/2609854_A_Java_ForkJoin_Framework

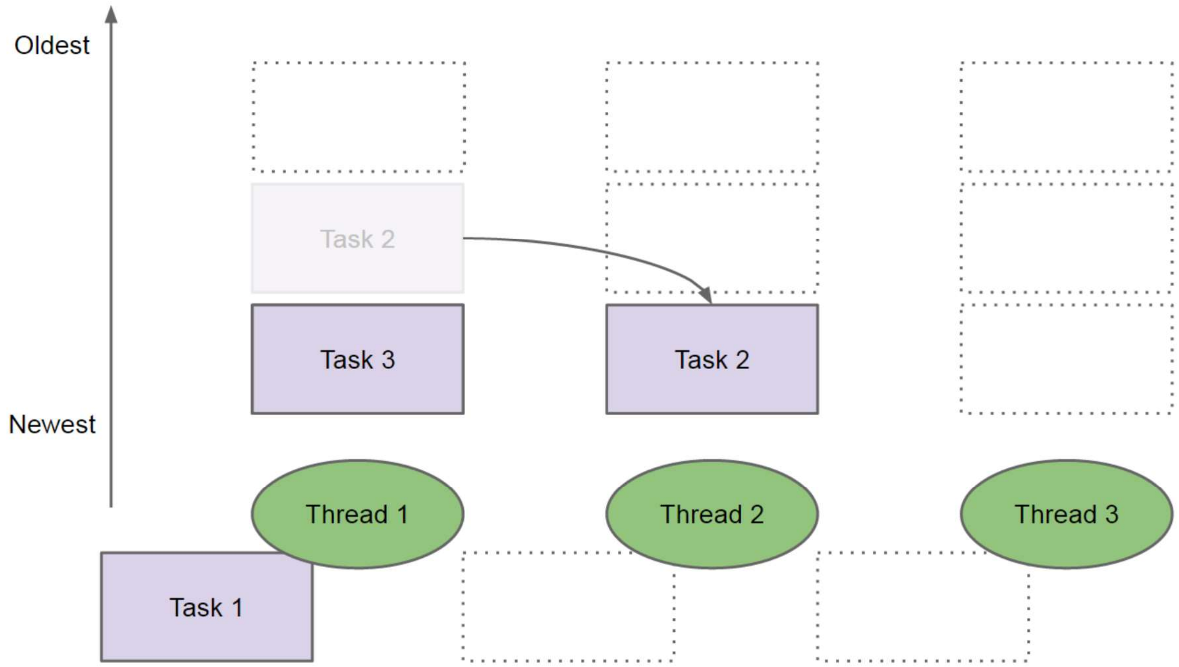
Daca

- 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/

Fork-Join Framework

```
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

- **Fork-Join Framework: ForkJoinTask<V>**

```
public class ForkJoinRecAc extends RecursiveAction {

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

protected void compute() {
……
}}

public static void main (String[] args){
ForkJoinPool fjpool =  ForkJoinPool.commonPool(4)
ForkJoinRecAc fjaction= new ForkJoinRecAc(workLoad);
pool.invoke(fjaction);}
```

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

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

protected <T> compute() {
……
}}

public static void main (String[] args){
ForkJoinPool fjpool =  ForkJoinPool.commonPool(4)
ForkJoinRecTk  fjtask= new ForkJoinRecTk(workLoad);
<T> result = fjtask.invoke(fjtask);}
```

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

➢ **Fork-Join Framework** cu **RecursiveAction** (forma generala)

```
public class MyRecursiveAction extends RecursiveAction {

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

 if (this.workLoad > limit) {…
                                List<MyRecursiveAction> subtasks = new ArrayList<MyRecursiveAction>();
                                 subtasks.addAll(createSubtasks());
                                 invokeAll(subtasks);}
                               }
else {// prelucrata de thread-ul curent}

}
private List<MyRecursiveAction> createSubtasks() {
        List<MyRecursiveAction> subtasks =  new ArrayList<MyRecursiveAction>();
…. }
```

invokeAll(Collection<T> tasks)

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

Exemplu**: crearea subtask-urilor**

```
private List<MyRecursiveAction>  createSubtasks() {

        List<MyRecursiveAction> subtasks =  new ArrayList<MyRecursiveAction>();

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

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

        return subtasks;
    }
```

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

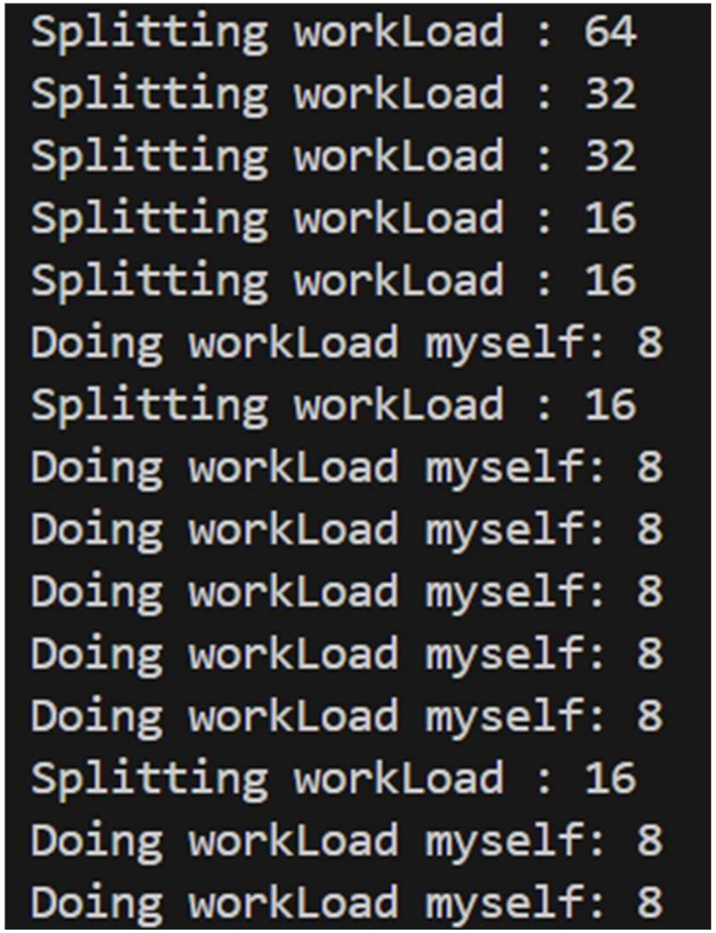
Exemplu: **Fork-Join Framework cu RecursiveAction**

```
protected void compute() {

if  (this.workLoad > 15) {

          System.out.println("Splitting workLoad : " + this.workLoad);

          List<MyRecursiveAction> subtasks = new ArrayList<MyRecursiveAction>();

          subtasks.addAll(createSubtasks()); invokeAll(subtasks);}


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

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

Exemplu: **Fork-Join Framework** cu **RecursiveAction**

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

    ForkJoinPool forkJoinPool = new ForkJoinPool(4);

    MyRecursiveAction myRecursiveAction = new MyRecursiveAction(64);

    forkJoinPool.invoke(myRecursiveAction);
  }}
```



```
Splitting workLoad : 64
Splitting workLoad : 32
Splitting workLoad : 32
Splitting workLoad : 16
Splitting workLoad : 16
Doing workLoad myself: 8
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
```

- **Fork-Join Framework cu RecursiveTask <T>** (forma generala)

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

public MyRecursiveTask (long workLoad) {
        this.workLoad = workLoad;
    }
protected  T compute() {

 if (this.workLoad > limit) {.....
                                List<MyRecursiveTask> subtasks = new ArrayList<MyRecursiveTask>();
                                 subtasks.addAll(createSubtasks());
                                 invokeAll(subtasks);
                                 return joinresult(subtasks);}

else {// prelucrata de thread-ul current;
           return result}}

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

joinresult(subtasks) calculeaza rezultatul rezultatele subtaskurilor se obtin cu **subtask.get()**

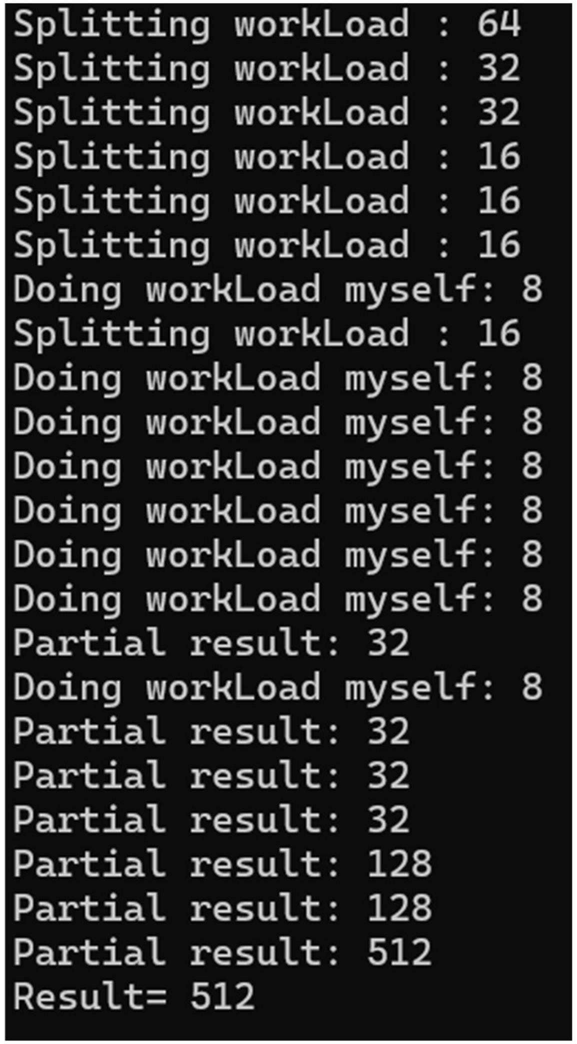
Exemplu: **Fork-Join Framework** cu **RecursiveTask<V>**

```
protected Integer compute() {
if(this.workLoad > 15) {
      System.out.println("Splitting workLoad : " + this.workLoad);
      List<MyRecursiveTask> subtasks = new ArrayList<MyRecursiveTask>();
      subtasks.addAll(createSubtasks());
      invokeAll(subtasks);

      int result = 0;
         try{
         for(MyRecursiveTask subtask : subtasks) {  result = result +  2* subtask.get();}
         System.out.println("Partial result: " + result);
         } catch (InterruptedException | ExecutionException e) {};
         return result;   }

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

joinresult(subtasks) calculeaza rezultatul
rezultatele subtaskurilor se obtin cu subtask.get()

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

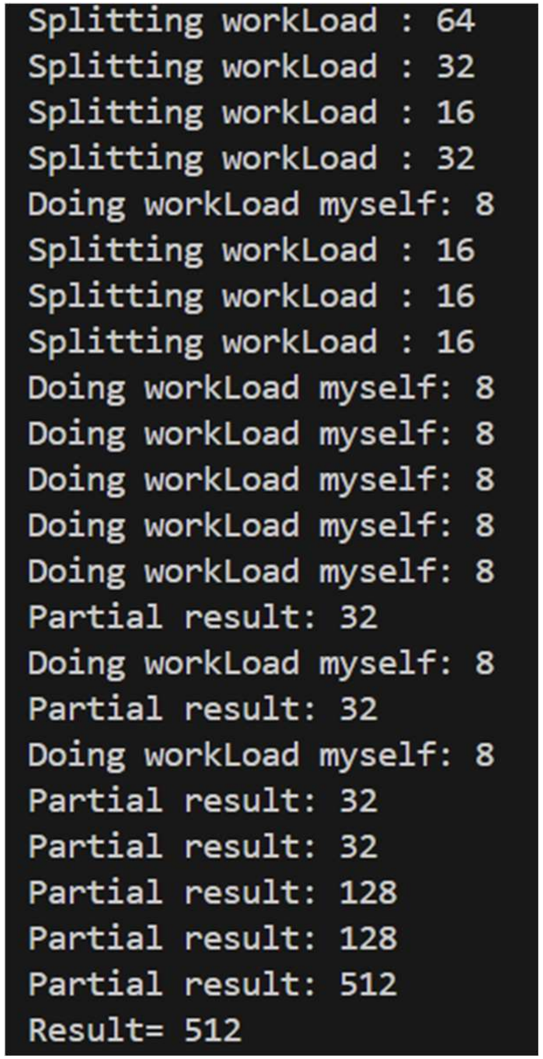
Exemplu: **Fork-Join Framework** pool cu **RecursiveTask**



```
Splitting workLoad : 64
Splitting workLoad : 32
Splitting workLoad : 32
Splitting workLoad : 16
Splitting workLoad : 16
Splitting workLoad : 16
Doing workLoad myself: 8
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Partial result: 32
Doing workLoad myself: 8
Partial result: 32
Partial result: 32
Partial result: 32
Partial result: 128
Partial result: 128
Partial result: 512
Result= 512
```

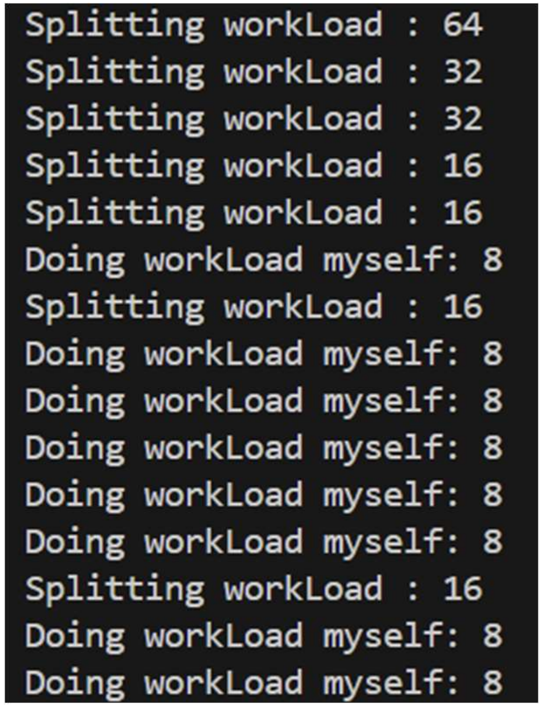
```
public static void main (String[] args){

    ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();

    MyRecursiveTask myRecursiveTask = new MyRecursiveTask(64);

    int res = forkJoinPool.invoke(myRecursiveTask);

    System.out.println("Result= " + res);}
```

Exemplu program: ForkJoin pool cu **RecursiveTask**



```
Splitting workLoad : 64
Splitting workLoad : 32
Splitting workLoad : 16
Splitting workLoad : 32
Doing workLoad myself: 8
Splitting workLoad : 16
Splitting workLoad : 16
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Partial result: 32
Doing workLoad myself: 8
Partial result: 32
Doing workLoad myself: 8
Partial result: 32
Partial result: 32
Partial result: 128
Partial result: 128
Partial result: 512
Result= 512
```

ForkJoin pool cu **RecursiveAction**



```
Splitting workLoad : 64
Splitting workLoad : 32
Splitting workLoad : 32
Splitting workLoad : 16
Splitting workLoad : 16
Doing workLoad myself: 8
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Doing workLoad myself: 8
Splitting workLoad : 16
Doing workLoad myself: 8
Doing workLoad myself: 8
```