IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

CONCURENTA IN JAVA

Ioana Leustean

➢ Virtual Threads (Java 23)

"Thread also supports the creation of virtual threads. Virtual threads are typically user-mode threads scheduled by the Java runtime rather than the operating system. Virtual threads will typically require few resources and a single Java virtual machine may support millions of virtual threads. Virtual threads are suitable for executing tasks that spend most of the time blocked, often waiting for I/O operations to complete. Virtual threads are not intended for long running CPU intensive operations."

Thread (Java SE 23 & JDK 23)

Java Downloads | Oracle România

```
public class HelloRunnablev implements Runnable {
public void run() {
      System.out.println("Hello from a thread!");}
public static void main(String args[]) throws InterruptedException {
       HelloRunnablev task = new HelloRunnablev();
       Thread.Builder builder = Thread.ofVirtual().name("MyThread");
       Thread t = builder.start(task);
        t.join(); }}
```

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

➢ Modelul de memorie JAVA

- 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/specs/jls/se23/html/jls-17.html#jls-17.4

➢ 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/specs/jls/se23/html/jls-17.html#jls-17.4

➢ Vizibilitate si Atomicitate

Interactiune dintre thread-uri trebuie sa asigure:

- *Excludere mutuala* - numai un thread executa o sectiune critica
  (o parte in care accesul la resurse trebuie sincronizat)
- *Vizibilitate* – modificarile datelor partajate facute de un thread sunt vizibile celorlalte thread-uri

Metodele sincronizate (si lacatele) asigura ambele proprietati, dar au cost computational mai ridicat.

Metode mai simple:

- variabilele **atomice**: operatiile sunt implementate prin instructiuni compare-and-swap
- variabilele **volatile** : asigura vizibilitatea dar nu si atomicitatea

➢ 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.atomic.AtomicInteger;
public class AtomicCounter {
 private static  AtomicInteger counter = new AtomicInteger();

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter.incrementAndGet();
});

Thread t2 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter.incrementAndGet();}
});

t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter.get());
```

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

➢ Variabile volatile NU asigura atomicitatea

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
 private static volatile int   counter = 0;

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter++;  });

Thread t2 = new Thread(new Runnable() {
                    public void run() {for (int i = 0; i < 1000; i++) counter++;}
});

t1.start(); t2.start();
t1.join(); t2.join();
System.out.println(counter);
```

```
C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
2000

C:\Users\igleu\Documents\DIR\ICLP\ICLP2023\c4-2023\pg4>java VolatileAtomic.java
1979
```

1979

➢ Variabilele **volatile**

sunt folosite atunci cand exista un thread care le actualizeaza si (eventual) mai multe care le citesc,
situatia tipica fiind variabila de control a unui ciclu

```
public class VolatileEx {
static volatile  boolean stop=false;

public static void main(String[] args) throws InterruptedException{
Thread t1 = new Thread(new Runnable() {
                    public void run() {int count =0;
                                        while (!stop) {count++; System.out.println(count); }
                                        System.out.println(count); });

Thread t2 = new Thread(new Runnable() {
                    public void run() {try{Thread.sleep(10);} catch (InterruptedException e) {}
                                        stop=true;}});
t1.start(); t2.start();
 t1.join(); t2.join();
System.out.println("STOP"); }
```

cu **volatile**

```
30606937
STOP
```

fara volatile
nu afiseaza nimic

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

➢ Crearea obiectelor de tip Thread:

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

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

public class Thread
extends Object
implements Runnable
```

- Metoda abstracta
  - folosind clasa Executors

```
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

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

```
ExecutorService  pool = Executors.newSingleThreadExecutor()

pool.execute( instanta Runnable )
```

Crearea unui obiect din clasa **Executors**

Crearea thread-urilor

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 pentru crerea unui obiect din clasa **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 create 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."

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

- Metode pentru crerea unui obiect din clasa **Executors**:
  - **newSingleThreadExecutor()**
  - **newCachedThreadPool()**
  - **newFixedThreadPool(poolSize)**

- Metode:
  - **shutdown()**
    serviciul nu primeste task-uri noi, permite thread-urilor deja aflate in executie sa termine, dar nu asteapta task-urile primate care nu sunt in executie (pentru aceasta trebuie folosit in combinatie cu **awaitTermination()**).
  - **shutdownNow()**
    terminarea serviciului, fara a permite finalizarea executiilor;
  - **awaitTermination(long timeout, TimeUnit unit)**
    pentru a permite finalizarea executiilor, impunand o limita temporara

➢ 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) {.. }}
```

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

➢ Generarea thread-urilor folosind Executors

```
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

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 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
```

➢ 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 {
        Thread.sleep(2000);  // 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<ResultType> 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();       }
```

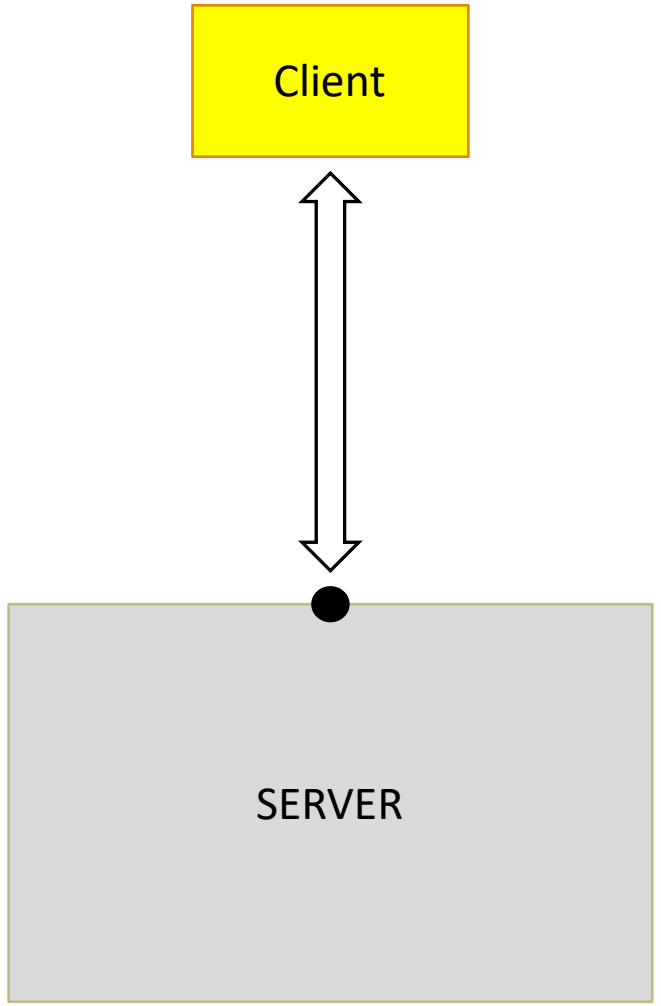
https://www.callicoder.com/java-callable-and-future-tutorial/

```
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();
    }
```

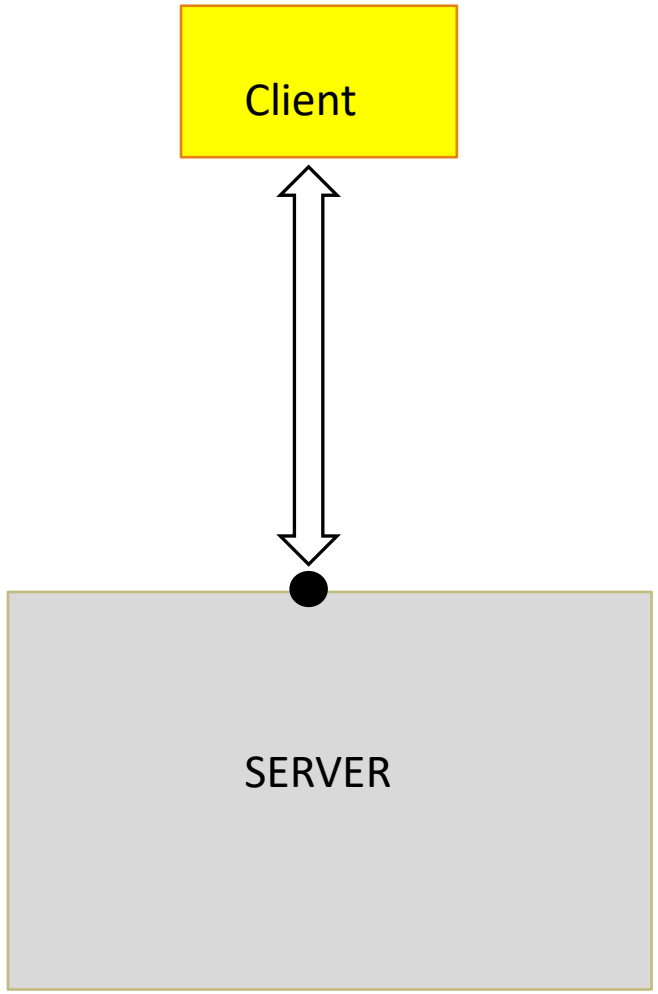
```
public static void main (String[] args) throws Exception{
      ExecutorService pool = Executors.newSingleThreadExecutor();
      int time = ThreadLocalRandom.current().nextInt(1000, 5000);
      System.out.println("Creating the future");

      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**



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



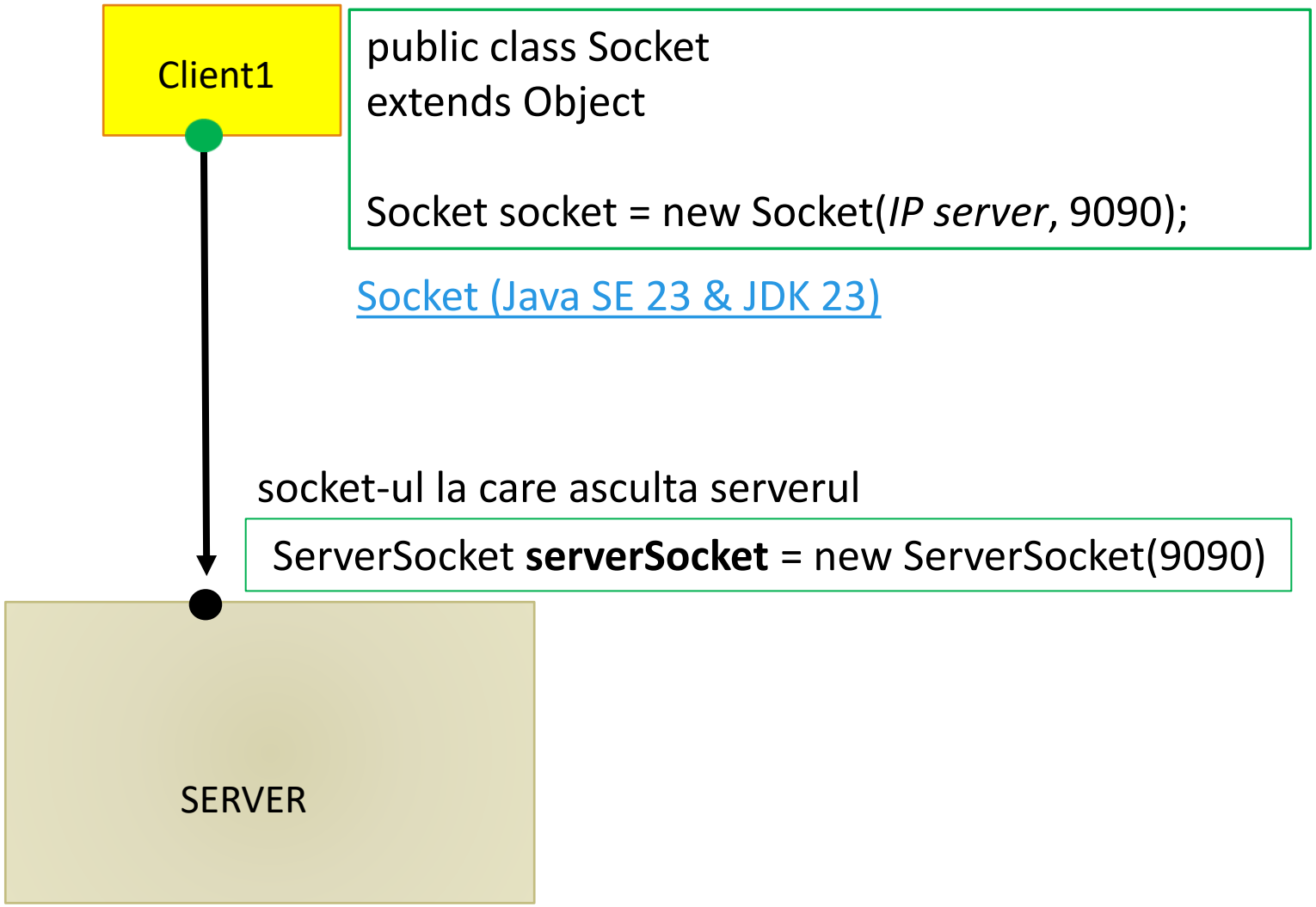
- un socket este un punct final in comunicarea bidirectionala dintre doua programe din aceeasi retea
- un socket are asociat un port

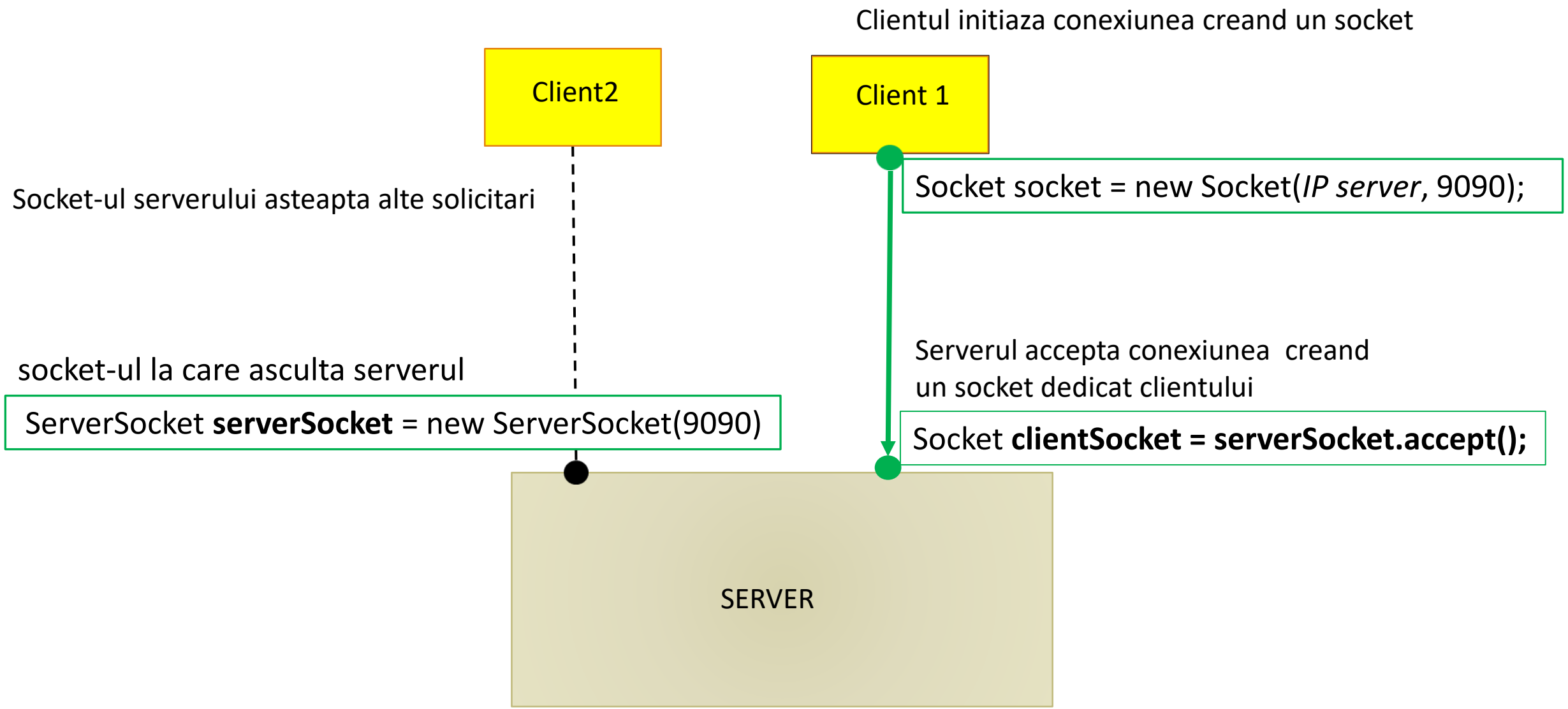
- un socket de server asteapta cererile venite din retea

```
public class ServerSocket
    extends Object

ServerSocket serverSocket = new ServerSocket(9090)
```

ServerSocket (Java SE 23 & JDK 23)

Clientul initiaza conexiunea creand un socket




Clientul initiaza conexiunea creand un socket
Client2
Client 1
Socket-ul serverului asteapta alte solicitari
Socket socket = new Socket(*IP server*, 9090);
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.");
        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);  //mesaj afisat pe propriul canal

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

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

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

```
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);    // citeste un mesaj si il trimite serverului

      String response = in.readLine();   //primeste un mesaj 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);    // citeste un mesaj si il trimite serverului

      String response = in.readLine();   //primeste un mesaj 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();
}}
```

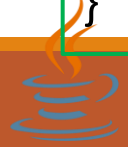
```
>Java Server
Sever is running
Client connected!
From client: buna!
Client connected!
From client: buna din nou!
|
```

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

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

```
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(clientSo
      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
>
```

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 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 Server 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 clientSocket = serverSocket.accept();
      System.out.println("Client connected!");
      Thread clientThread = new Thread (new ServerMT (cSocket));
      clientThread.start(); }}

public Server(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

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

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

public static void main(String args[])
   {  ServerSocket serverSocket = new ServerSocket(909
      System.out.println("Server is running");
      while (true){
      Socket clientSocket = serverSocket.accept();
      System.out.println("Client connected!");
      Thread clientThread = new Thread (new ServerMT
      clientThread.start(); }}

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

  public void run()  {
      BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStre
      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: " +
 }
      clientSocket.close();}}
```

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

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

```
>Java Server
Sever 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 Server 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 clientSocket = serverSocket.accept();
      System.out.println("Client connected!");
      ExecutorService  pool = Executors.newCachedThreadPool();
      pool.execute(new ServerMT (cSocket)); }}

public Server(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

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