Medii de proiectare și programare

2021-2022
Curs 6

Conținut curs 6

- Networking si threading in Java (cont.)
- Exemplu Mini-Chat (networking)
- Networking si threading in C#

Networking în Java

- **`java.net.Socket`** deschide o conexiune TCP din partea clientului.

```
public Socket(String host, int port) throws UnknownHostException,
                                            IOException
public Socket(InetAddress host, int port) throws IOException
```

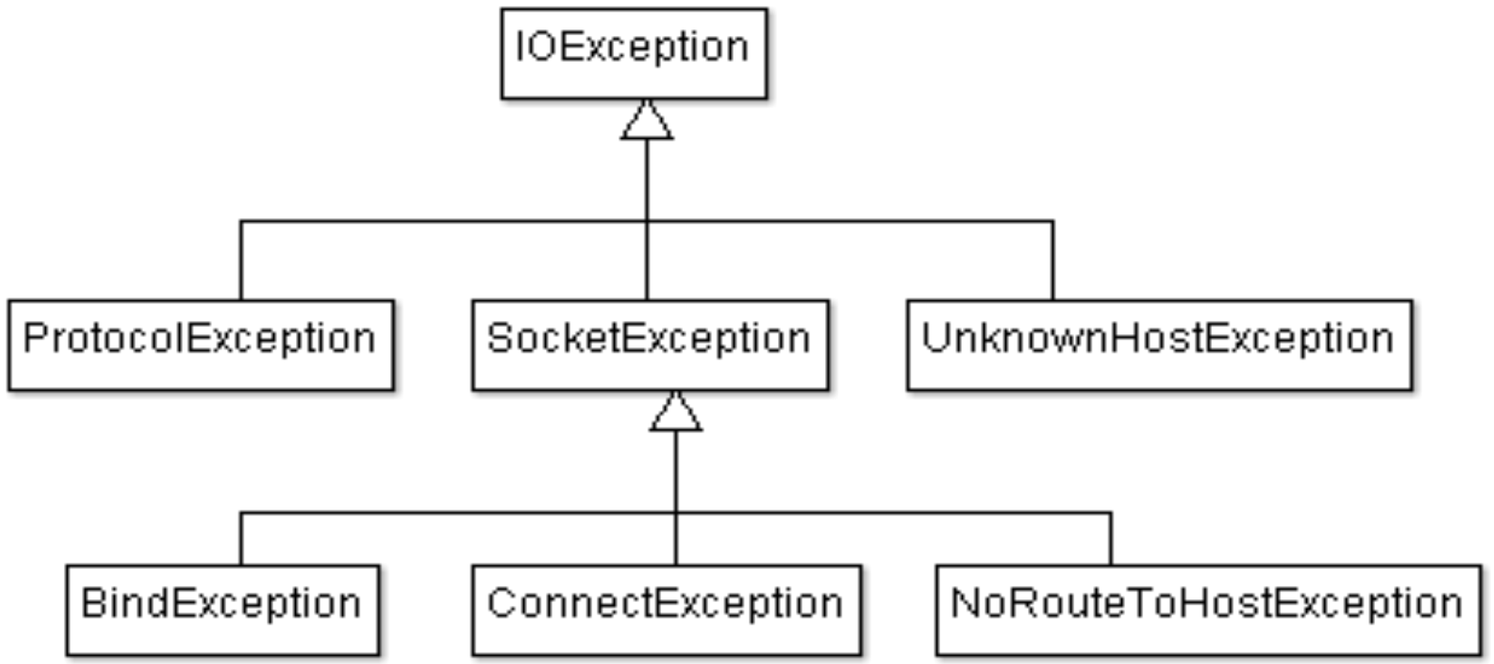
- Metode:

```
public int getPort()
public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException
public void close() throws IOException
```

Networking în Java

```
try (Socket connection=new Socket(“172.30.106.5”, 5555)){
    //processing code
}catch(UnknownHostException e){
   //…
}catch(IOException  e){
   //…
}
```

Excepții


IOException
ProtocolException
SocketException
UnknownHostException
BindException
ConnectException
NoRouteToHostException

Threading în Java

- Două modalități de definire a unui thread:
  - Extinderea clasei **`Thread`** și redefinirea metodei **`run.`**
  - Implementarea interfeței **`Runnable`** și definierea metodei **`run`**.
- Crearea unui thread se face prin intermediul clasei **`Thread`**

```
public Thread()
public Thread(Runnable target)
```

- Pornirea execuției unui thread se face prin apelul metodei **`start`** din clasa **`Thread`**:

```
public void start()
```

Sincronizarea threadurilor

- Instrucțiunea **synchronized**

```
synchronized(locker_obj){
 //code to execute
}
```

- Sincronizarea unei metode:

```
public synchronized void methodA();
```

- Yielding: un thread renunța la CPU alocat și permite execuția altui thread:

```
public static void yield( );
```

```
public void run( ) {
  while (true) {
    // Time and CPU consuming thread's work…,
    Thread.yield( );
  }
}
```

Utilități Java Concurrency

- Java 5 a introdus utilități pentru concurență - un framework extensibil care permite crearea containerelor de thread-uri și cozi sincronizate (eng. *blocking queues*):

  - **`java.util.concurrent`**: Tipuri utile în programarea concurentă (ex. executors)

  - **`java.util.concurrent.atomic`**: Programare concurentă avansată

  - **`java.util.concurrent.locks`**: Mecanisme de blocare avansate, mai performante decât notify/wait.

Taskuri Java

- Un obiect *task* Java este un obiect a cărui clasa implementează interfața **`java.lang.Runnable`** (*taskuri runnable*) sau interfața **`java.util.concurrent.Callable`** (*taskuri callable*).

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

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

- Metoda **`call()`** poate returna o valoare și poate arunca excepții (checked).

Execuția taskurilor Java

- Interfața Executor - execuția taskurilor runnable:

```
public interface Executor{
  void execute(Runnable command)
}
```

- **`ScheduledThreadPoolExecutor, ThreadPoolExecutor`**

- Dezavantaje:
  - Se axeaza doar pe **`Runnable`**. Metoda **`run()`** nu returnează nici o valoare. Este dificilă returnarea unei valori ca și rezultat al execuției taskului.
  - Nu oferă posibilitatea monitorizării progresului execuției unui task runnable care se execută (se execută încă?, anulat? execuția s-a încheiat?)
  - Nu poate executa mai multe taskuri.
  - Nu ofera posibilitatea opririi unui executor.

ExecutorService

- **`java.util.concurrent.ExecutorService`** soluția pentru problemele apărute la interfața **`Executor`**.
- Este implementat folosind un container de threaduri (eng. *thread pool*).

```
public interface ExecutorService extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  <T> Future<T> submit(Callable<T> task);
  <T> Future<T> submit(Runnable task,  T result);
  <T> List<Future<T>> invokeAll(Collection<? extends  Callable<T>> tasks);
  <T> T invokeAny(Collection<? extends Callable<T>> tasks);
//alte metode
}
```

- **`ScheduledThreadPoolExecutor, ThreadPoolExecutor`**

Executorul trebuie oprit după terminarea execuției, altfel aplicația nu își va termina execuția.

Interfața Future

- Un obiect de tip Future reprezintă rezultatul unui calcul asincron.
- Rezultatul este numit *future* pentru că de obicei nu va fi disponibil decât la un moment în viitor.
- Are metode pentru: anularea execuției unui task, obținerea rezultatului execuției, determinarea dacă un task și-a încheiat execuția.

```
public interface Future<V>{
  boolean isCancelled();
  boolean isDone();
  boolean cancel(boolean mayInterruptIfRunning)
  V get() throws InterruptedException, ExecutionException;
  //alte metode …
}
```

Clasa Executors

- Clasa **`Executors`** conține metode statice care returnează obiecte de tip **`ExecutorService`**:
  - **`newFixedThreadPool(int nThreads): ExecutorService`**
  - **`newSingleThreadExecutor(): ExecutorService`**
  - **`newCachedThreadPool():ExecutorService`**
  - **`newWorkStealingPool(): ExecutorService`**

Colecții concurente

- Colecții folositoare în programarea concurentă.
- Începând cu versiunea 1.5
- Interfața **`BlockingQueue`**:
  - Coadă - conține metode care așteaptă ca coadă să devină nevidă la scoaterea unui element, respectiv așteaptă eliberarea spațiului la adăugarea unui element.
  - Implementările BlockingQueue au fost proiectate și implementate pentru a fi folosite în situații de tip producător-consumator.
  - **`ArrayBlockingQueue`**, **`LinkedBlockingQueue`**, **`PriorityBlockingQueue`**, etc.
- Interfața **`BlockingDeque`**:
  - Extinde **`BlockingQueue`** și oferă suport pentru operații de tip FIFO și LIFO.
  - **`LinkedBlockingDeque`**
- Interfața **`ConcurrentMap:`**
  - Subinterfață a **`java.util.Map`**
  - **`ConcurrentHashMap, ConcurrentSkipListMap.`**

Exemplu BlockingQueue

- Producător-Consumator simplu cu BlockingQueue

```
//ambele threaduri au referință la obiectul messages

//inițializarea
private BlockingQueue<String> messages=new
   LinkedBlockingQueue<String>();

//Producator
try {
     messages.put(message);
} catch (InterruptedException e) {
     e.printStackTrace();
}

//Consumator
String message = messages.take();
```

Actualizare GUI

- Interfețele grafice (JavaFX, Swing) folosesc obiecte de tip Component.
- Aceste obiecte pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.

```
Platform.runLater(new Runnable() {                    //JavaFX
      @Override
       public void run() {
         //codul care modifica informatia de pe interfata grafica
         label.setText("New text …");
       }
 });
```

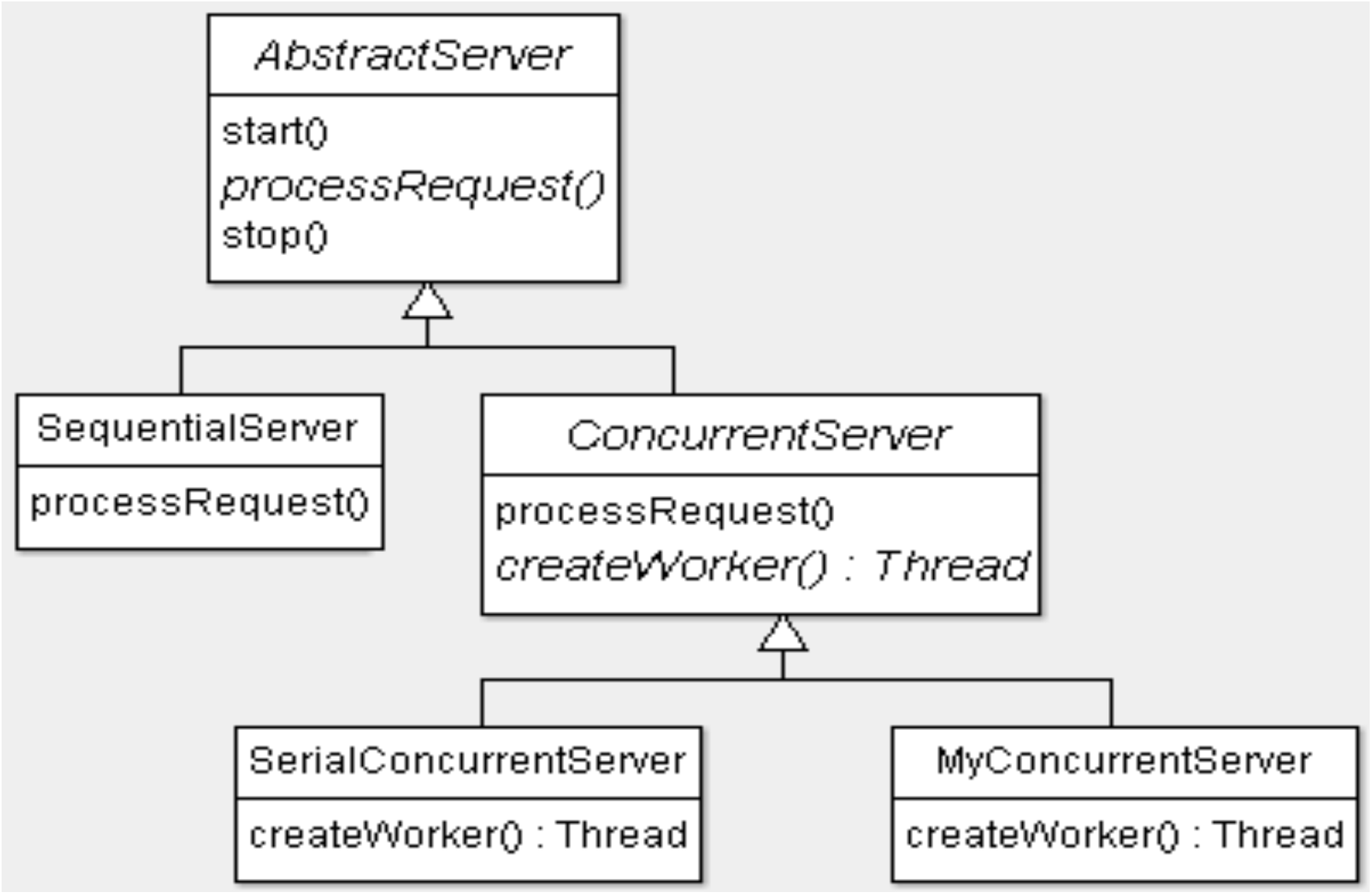
//sau, folosind funcții lambda

```
Platform.runLater(() -> {                        //JavaFX
         //codul care modifica informatia de pe interfata grafica
         label.setText("New text …");
 });
```

Exemplu Java

- O aplicație simplă client/server:
  - Serverul așteaptă conexiuni.
  - Clientul se conectează la server si îi trimite un text.
  - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

Server Template

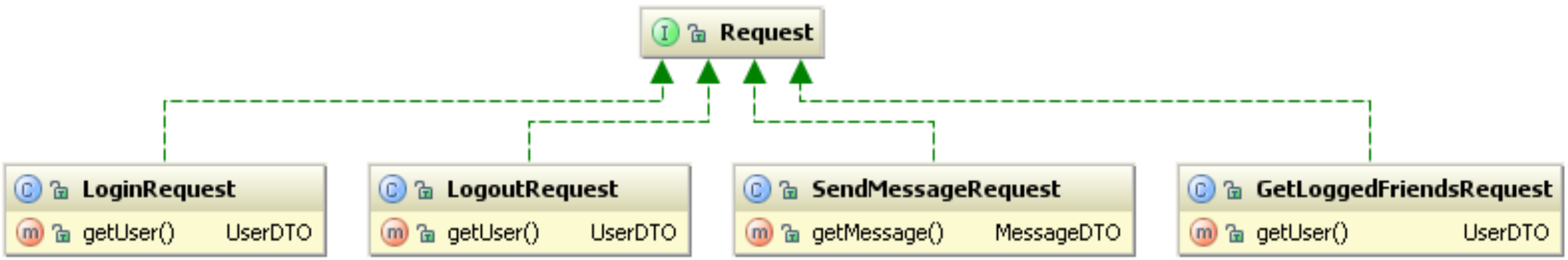


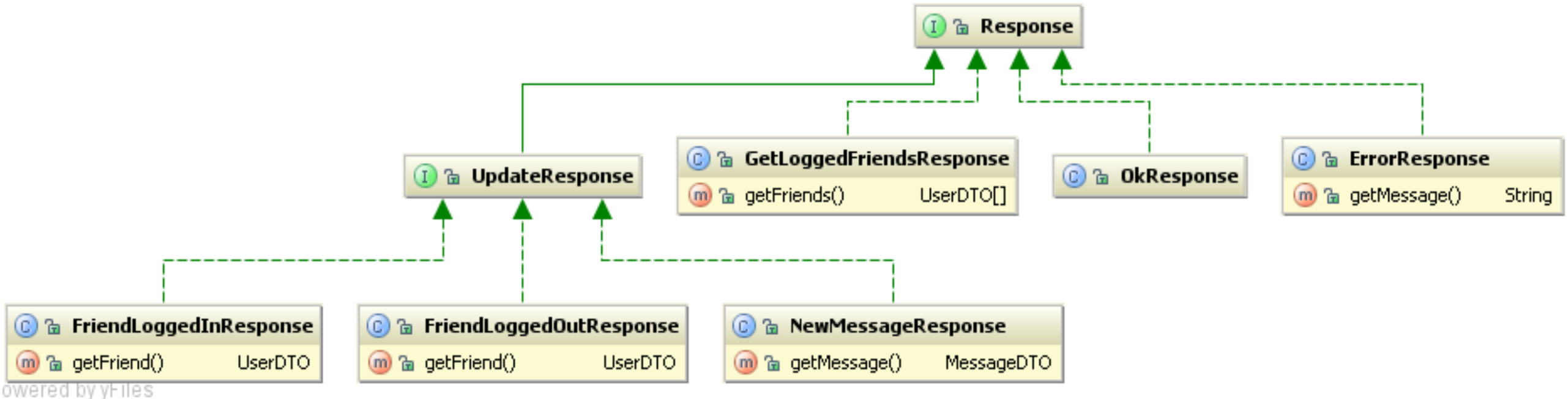
Mini-Chat

- Proiectare

- Implementare Java

Mini-chat Object Protocol





Mini-chat Rpc Protocol