

Medii de proiectare și programare

2019-2020

Curs 5

Conținut curs 5

- Networking si threading în Java
- Networking și threading în C#
- Exemplu Mini-Chat

Networking în Java

- `java.net` - pachetul conține clase pentru comunicarea TCP/UDP prin rețea.
- TCP: `Socket` și `ServerSocket`.
- UDP: `DatagramPacket`, `DatagramSocket` și `MulticastSocket`.
- Clasa `InetAddress` reprezintă o adresă IP:
 - `Inet4Address`: pentru adrese IPv4 (32 bits).
 - `Inet6Address`: pentru adrese IPv6 (128 bits).

```
InetAddress localhost=InetAddress.getLocalHost();
```

```
InetAddress googAdr=InetAddress.getByName("www.google.com");
```

- `InetSocketAddress` asociere între o adresă IP și un port:

```
InetSocketAddress(InetAddress addr, int port);
```

```
InetSocketAddress(String hostname, int port);
```

Networking in Java

- **ServerSocket** reprezintă clasa corespunzătoare serverului care așteaptă conexiuni TCP.

- Constructori/Metode:

```
public ServerSocket(int port) throws BindException, IOException
```

```
public ServerSocket( ) throws IOException //not bind yet, since Java 1.4
```

```
//binds a server to a port
```

```
public void bind(SocketAddress endpoint) throws IOException
```

```
//blocks and waits for clients
```

```
public Socket accept( ) throws IOException
```

```
//closes the server
```

```
public void close() throws IOException
```

Networking in Java

```
ServerSocket server=null;
try{
    server=new ServerSocket(5555) ;
    while(keepProcessing){
        Socket client=server.accept() ;
        //processing code
    }
}catch(IOException ex){
    //...
}finally{
    if(server!=null){
        try{
            server.close() ;
        }catch(IOException ex){...}
    }
}
```

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
```

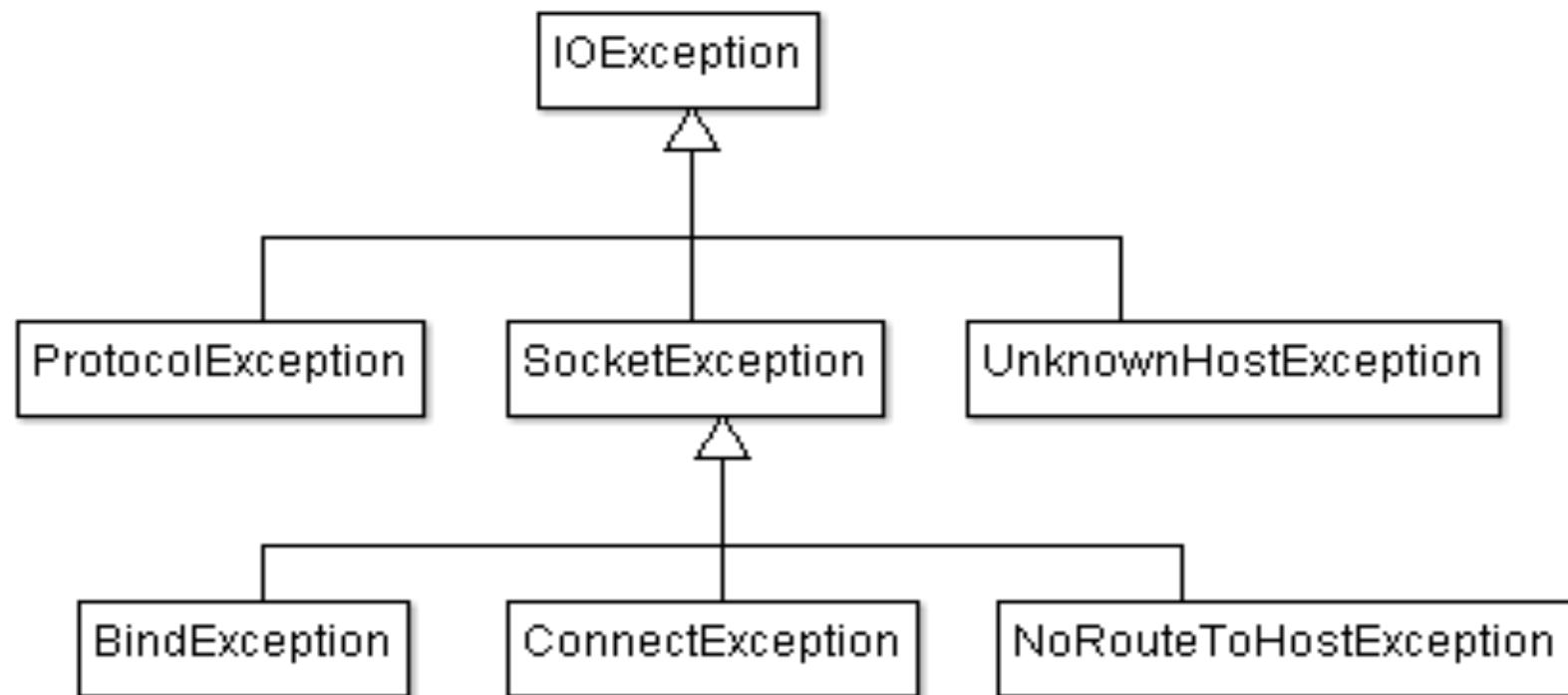
```
public OutputStream getOutputStream() throws IOException
```

```
public void close() throws IOException
```

Networking in Java

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

Excepții



Threading în Java

- Două modalități de definire a unui thread:
 - Extinderea clasei **Thread** și redefinirea metodei **run**.
 - Implementarea interfeței **Runnable** și definirea 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 clasă 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 axează 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 oferă 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 folosite 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 ...");
    }
});
```

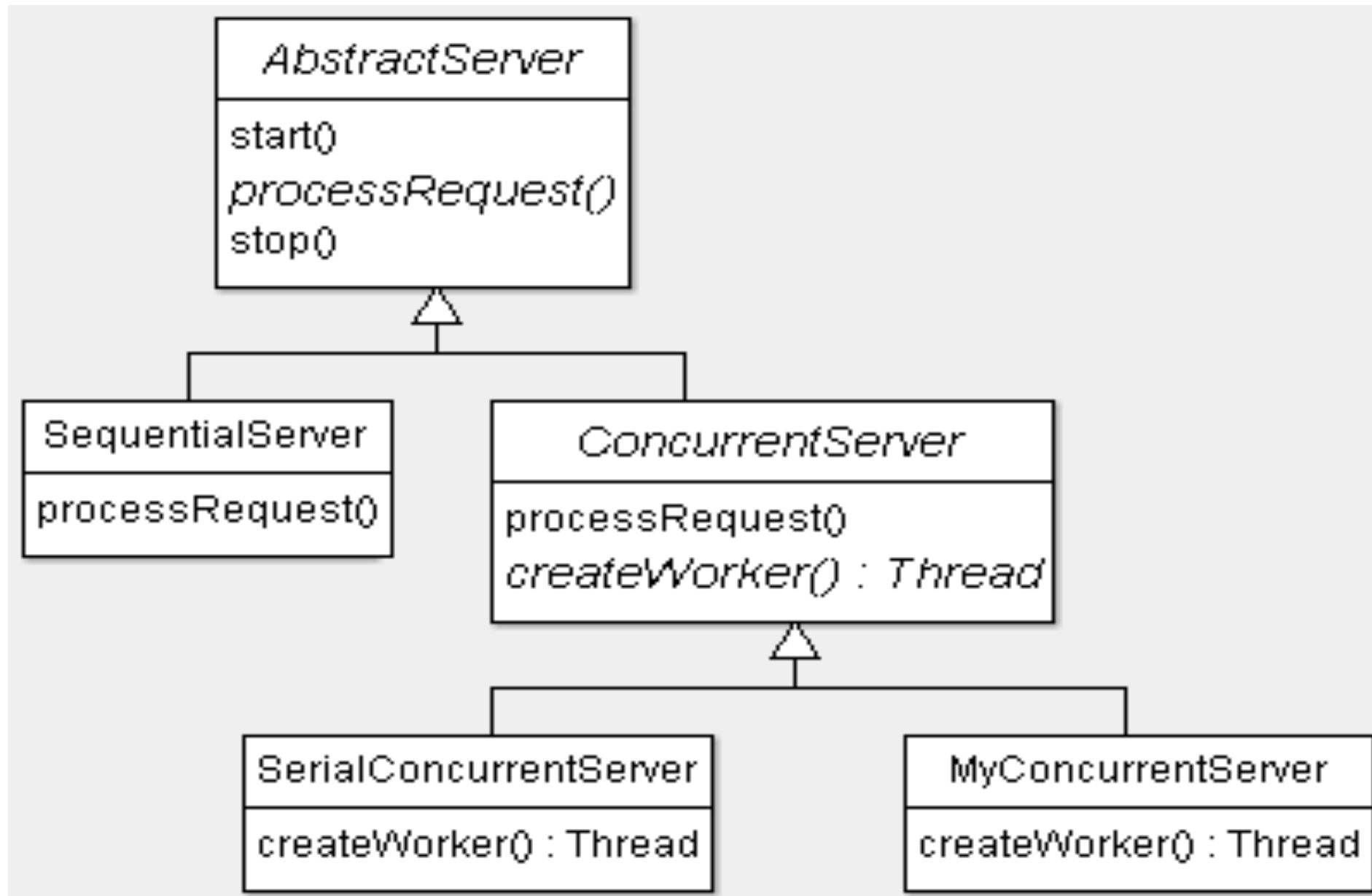
//sau, folosind funcții lambda

```
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 și îi trimite un text.
 - Serverul returnează textul scris cu litere mari, la care adaugă data și ora la care a fost primit textul.

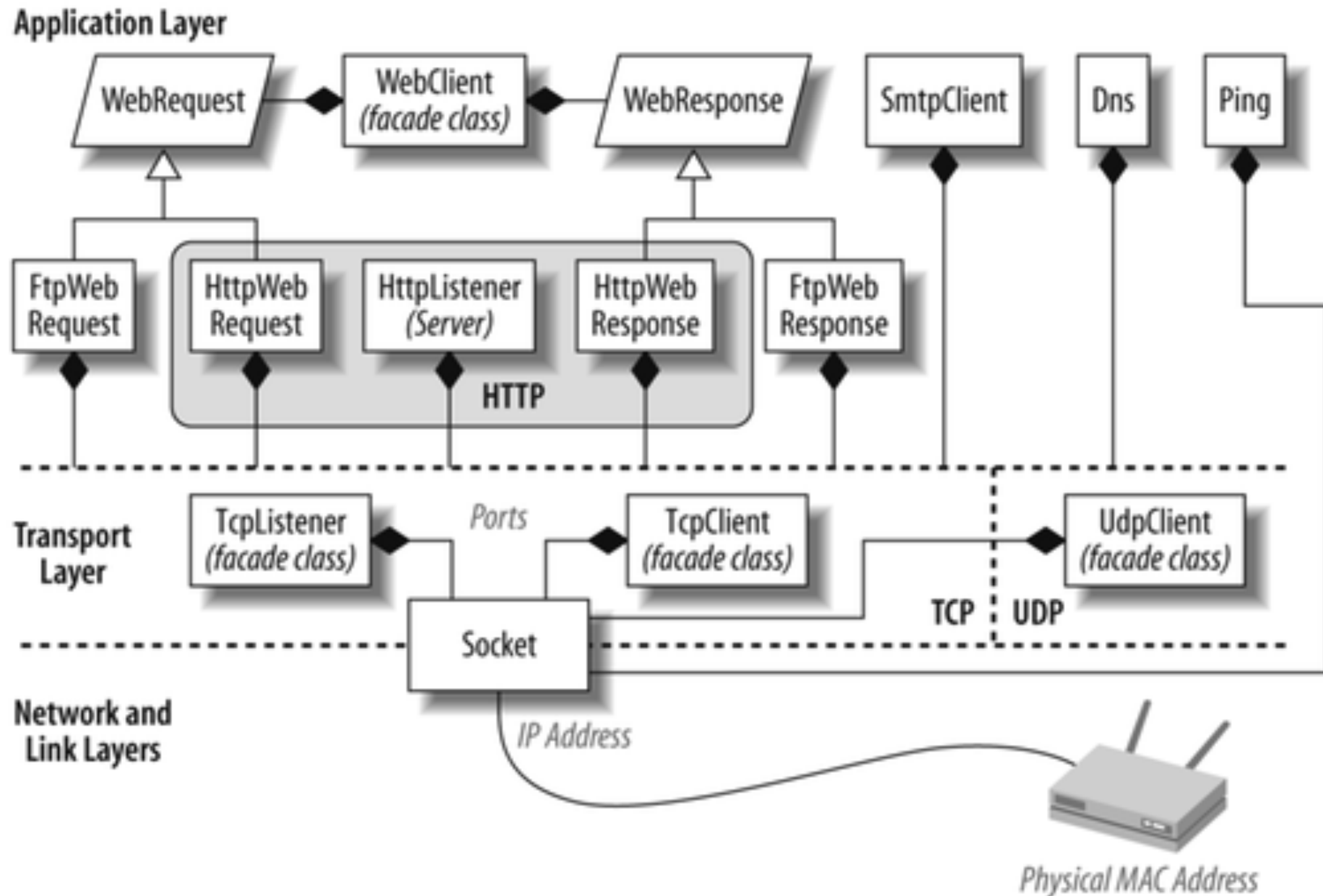
Server Template



Networking în C#

- .NET conține clase pentru comunicarea prin rețea folosind protocoale standard cum ar fi HTTP, TCP/IP și FTP.
- Spațiul de nume **System.Net.***:
 - **WebClient** fațade pentru operații simple de download/upload folosind HTTP sau FTP.
 - **WebRequest** și **WebResponse** pentru operații HTTP și FTP complexe.
 - **HttpListener** pentru implementarea unui HTTP server.
 - **SmtpClient** pentru construirea și trimiterea mesajelor folosind SMTP.
 - **TcpClient**, **UdpClient**, **TcpListener**, și **Socket** pentru acces direct la nivelul rețea.

Networking in C#



Networking în C#

- Clasa `IPAddress` din `System.Net` reprezintă o adresă IPv4 (32 bits) sau IPv6 (128 bits).

```
IPAddress a1 = new IPAddress (new byte[] { 172, 30, 106, 5 });
```

```
IPAddress a2 = IPAddress.Parse ("172.30.106.5");
```

```
IPAddress a3 = IPAddress.Parse  
    (" [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]"); //IPv6
```

- O asociere între o adresă IP și un port este reprezentată folosind clasa `IPEndPoint`:

```
IPAddress a = IPAddress.Parse ("172.30.106.5");
```

```
IPEndPoint ep = new IPEndPoint (a, 55555); // Port 55555
```

```
Console.WriteLine (ep.ToString()); // 172.30.106.5:55555
```

- Porturile: 1 – 65535.
- Porturile dintre 49152 și 65535 nu sunt rezervate oficial.

System.Net.Sockets Namespace

- Clasele **TcpClient**, **TcpListener** și **UdpClient** încapsulează detaliile creării conexiunilor de tip TCP și UDP.
- **Socket** implementează interfața Berkeley socket.
- **SocketException** excepția aruncată când apare o eroare la comunicarea prin socket.
- **NetworkStream** streamul folosit pentru comunicarea prin rețea.

TcpListener

- TCP server:

```
TcpListener listener = new TcpListener (<ip address>, port);  
listener.Start();  
while (keepProcessingRequests)  
    using (TcpClient c = listener.AcceptTcpClient( ))  
        using (NetworkStream n = c.GetStream( ))    {  
            // Read and write to the network stream...  
        }  
listener.Stop ();
```

- **TcpListener** necesită adresa IP la care va aștepta conexiunile clienților (dacă calculatorul are două sau mai multe plăci de rețea).
 - **IPAddress.Any** ascultă pe toate adresele IP locale (sau singura).
- **AcceptTcpClient** blochează execuția până când se conectează un client.

TcpClient

- Client Tcp:

```
using (TcpClient client = new TcpClient (<address>, port))  
    using (NetworkStream n = client.GetStream( ))  
    {  
        // Read and write to the network stream...  
    }
```

- **TcpClient** încearcă crearea conexiunii în momentul creării obiectului folosind adresa IP și portul specificate.
- Constructorul blochează execuția până la stabilirea conexiunii.

NetworkStream

- **NetworkStream** comunicare bidirecțională pentru transmiterea și recepționarea datelor după stabilirea unei conexiuni.
- Methods:
 - **Read**
 - **Close**
 - **Write**
 - **Seek**
 - **Flush**
- Properties:
 - **CanRead, CanWrite**
 - **Socket**
 - **DataAvailable**
 - **Length**

Threading în C#

- Spatiul de nume `System.Threading` clase și interfețe pentru programarea concurentă:
 - Clasa `Thread`.
 - Delegate: `ThreadStart`, `ParameterizedThreadStart`.
 - Sincronizare: `lock`, `Monitor`, `Mutex`, `Semaphore`, `EventHandles`.
- Delegates: reprezintă metoda executată de un thread.

```
public delegate void ThreadStart();
```

```
public delegate void ParameterizedThreadStart(Object obj);
```

- Clasa `Thread`: crearea unui thread, setarea priorității, obținerea informațiilor despre statusul unui thread.

```
public Thread(ThreadStart start);
```

```
public Thread(ParameterizedThreadStart start);
```

Threading in C#

```
class Program {
    static void Main(string[] args) {
        Worker worker=new Worker();
        Thread t1=new Thread(new ParameterizedThreadStart(static_run));
        Thread t2=new Thread(new ThreadStart(worker.run));
        t1.Start("a");
        t2.Start();
    }
    static void static_run(Object data) {
        for(int i=0;i<26;i++) { Console.Write("{0} ",data); }
    }
}

class Worker {
    public void run() {
        for(int i=0;i<26;i++) Console.Write("{0} ",i);
    }
}

//a a a a a a a a a a 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 a a a a a a a a a a a a a a a a
```

Sincronizarea threadurilor

- Diferite tipuri:
 - *Blocarea exclusivă*: doar un singur thread poate executa o porțiune de cod la un moment dat.
 - `lock`, `Mutex`, and `SpinLock`.
 - *Blocarea nonexclusivă*: limitarea concurenței.
 - `Semaphore` and `ReaderWriterLock`.
 - *Semnalizarea*: un thread poate bloca execuția până la primirea unei notificări de la unul sau mai multe threaduri.
 - `ManualResetEvent`, `AutoResetEvent`, `CountdownEvent` ȘI `Barrier`.

Sincronizarea threadurilor –Blocarea

- Instrucțiunea `lock`:

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

- `locker_obj` object de tip referință.
- Doar un singur thread poate obține accesul la un moment dat. Dacă mai multe threaduri încearcă să obțină accesul, ele sunt puse într-o coadă și primesc accesul pe baza regulii “primul venit-primul servit”.
- Dacă un alt thread a obținut deja accesul, threadul curent nu își continuă execuția până nu obține accesul.

Sincronizarea threadurilor - Signaling

- *Event wait handles* - construcții simple pentru semnalizare:
 - **EventWaitHandle** - reprezintă un eveniment pentru sincronizarea threadurilor. Unul sau mai multe threaduri blochează execuția folosind un **EventWaitHandle** până când un alt thread apelează metoda **Set** permițând execuția unuia sau mai multor threaduri aflate în așteptare.
 - **AutoResetEvent**, **ManualResetEvent**
 - **CountdownEvent** (Framework 4.0)
- **AutoResetEvent** notifică un thread aflat în așteptare de apariția unui eveniment (doar un singur thread).
 - **Set()** - eliberează un thread aflat în așteptare
 - **WaitOne()** - threadul așteaptă apariția unui eveniment

Observații:

1. Dacă **Set** este apelată când nici un thread nu se află în așteptare, handle -ul așteaptă până când un thread apelează metoda **WaitOne**.
2. Apelarea metodei **Set** de mai multe ori când nici un thread nu este în așteptare nu va permite mai multor threaduri obținerea accesului când apelează metoda **WaitOne**.

Sincronizarea threadurilor - Signaling

- **ManualResetEvent** (asemănător **AutoResetEvent**) – notifică toate threadurile aflate în așteptare la apariția unui eveniment.

- Crearea unui event wait handle:

- Constructori:

```
AutoResetEvent waitA=new AutoResetEvent(false);
```

```
AutoResetEvent waitA=new AutoResetEvent(true); //calls Set
```

```
ManualResetEvent waitM=new ManualResetEvent(false);
```

- Clasa **EventWaitHandle**

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

```
var manual = new EventWaitHandle (false, EventResetMode.ManualReset);
```

- Distrugerea unui wait handle:

- Apelul metodei **Close** pentru eliberarea resurselor sistemului de operare.
- Ștergerea referințelor pentru a permit garbage collector-ului distrugerea obiectului.

Exemplu Signaling

```
class WaitHandleExample
{
    static EventWaitHandle waitHandle = new AutoResetEvent (false);
    static void Main()
    {
        new Thread (Worker).Start();
        Thread.Sleep (1000);    // Pause for a second...
        waitHandle.Set();    // Wake up the Worker.
    }
    static void Worker()
    {
        Console.WriteLine ("Waiting...");
        waitHandle.WaitOne();    // Wait for notification
        Console.WriteLine ("Notified");
    }
}
```

Task-uri C#

- Limitările threadurilor:
 - Se pot transmite ușor date unui thread, dar nu se poate obține la fel de ușor rezultatul execuției unui thread.
 - Dacă execuția threadului aruncă o excepție, tratarea excepției și retransmiterea ei este mai dificil de implementat.
 - Nu se poate seta ca un thread să execute altceva când și-a încheiat execuția.
- Un *Task* C# reprezintă o operație concurentă care poate fi (sau nu) executată folosind threaduri.
 - Taskurile pot fi compuse.
 - Pot folosi un container de threaduri pentru a reduce timpul necesar pornirii execuției.
- Tipul **Task** a fost introdus începând cu Framework 4.0 ca făcând parte din biblioteca pentru programare paralelă.
- **System.Threading.Tasks** namespace.

Pornirea execuției unui Task

- Framework 4.5 - Metoda statică **Task.Run** (pornirea execuției unui task folosind threaduri) - parametru de tip **Action** delegate:

```
Task.Run (() => Console.WriteLine ("Ana")) ;
```

- Framework 4.0 - Metoda statică **Task.Factory.StartNew**:

```
Task.Factory.StartNew (() => Console.WriteLine ("Ana")) ;
```

- Implicit taskurile folosesc threaduri din containere deja create.
- Folosirea metodei **Task.Run** - similar cu execuția explicită folosind threaduri:
 - **new Thread** (() => **Console.WriteLine** ("Ana")).**Start**() ;
- **Task.Run** returnează un obiect **Task** care poate fi folosit pentru monitorizarea progresului.
- Nu este necesară apelarea metodei **Start**.

Obținerea rezultatului execuției

- **Task<TResult>** subclasă a clasei **Task** care permite returnarea rezultatului execuției.
- **Task<TResult>** poate fi obținut apelând **Task.Run** folosind un delegate de tip **Func<TResult>** (sau o expresie lambda compatibilă).
- Rezultatul poate fi obținut folosind proprietatea **Result**.
- Dacă taskul nu și-a încheiat execuția, apelul proprietății **Result** va bloca execuția threadului curent până la terminarea execuției taskului:

```
Task<int> task = Task.Run(()=>{int x=2; return 2*x; });  
int result = task.Result;    // Blocks if not already finished  
Console.WriteLine (result);    // 4
```

Taskuri și excepții

- Taskurile propagă excepțiile (threadurile nu).
- Dacă un task aruncă o excepție, excepția este rearuncată către codul care apelează metoda `Wait()` a clasei `Task` sau care apelează proprietatea `Result` a clasei `Task<TResult>`:
- Excepția va fi inclusă într-o excepție de tip `AggregateException` de către CLR:

// Start a Task that throws a NullReferenceException:

```
Task task = Task.Run (() => { throw null; });

try{
    task.Wait();
}catch (AggregateException aex){
    if (aex.InnerException is NullReferenceException)
        Console.WriteLine ("Null!");
    else throw;
}
```

Actualizare GUI

- Interfețele grafice de tip Windows Forms folosesc obiecte de tip Control.
- Aceste obiecte pot fi modificate (actualizate, șterse, etc) doar de threadul care le-a creat.
- Nerespectarea acestei reguli are rezultate neașteptate sau aruncă excepții.
- Dacă se dorește apelarea unui membru (metoda, atribut, proprietăți) a obiectului X creat într-un thread Y, cererea trebuie transmisă threadului Y (folosind metoda **Invoke** sau **BeginInvoke** a obiectului X).

Actualizare GUI

- **Invoke** și **BeginInvoke** au un parametru de tip delegate care referă metoda corespunzătoare obiectului de tip Control care se dorește a se executa.
- **Invoke** execuție sincronă: execuția apelantului este blocată până la actualizarea controlului.
- **BeginInvoke** execuție asincronă: execuția apelantului continuă, iar cererea este pusă într-o coadă (corespunzătoare evenimentelor de la tastatură, mouse, etc) și se va executa ulterior.

Exemplu

//1. definirea unei metode pentru actualizarea unui ListBox

```
private void updateListBox(ListBox listBox, IList<String> newData) {  
    listBox.DataSource = null;  
    listBox.DataSource = newData;  
}
```

//2. definirea unui delegate care va fi apelat de GUI Thread

```
public delegate void UpdateListBoxCallback(ListBox list, IList<String>  
data);
```

//3. în celălalt thread se transmite metoda care actualizeaza ListBox:

```
list.Invoke(new UpdateListBoxCallback(this.updateListBox), new Object[]  
{list, data});
```

or

```
list.BeginInvoke(new UpdateListBoxCallback(this.updateListBox), new  
Object[]{list, data});
```

Exemplu C#

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

Mini-chat

- Proiectați și implementați o aplicație client-server pentru un mini-chat având următoarele funcționalități:
 - *Login*. După autentificarea cu succes, o nouă fereastră se deschide în care sunt afișați toți prietenii *online* ai utilizatorului și o listă cu mesajele trimise/primate de utilizator. De asemenea, toți prietenii online văd în lista lor că utilizatorul este *online*.
 - *Trimiterea unui mesaj*. Un utilizator poate trimite un mesaj text unui prieten care este online. După trimiterea mesajului, prietenul vede automat mesajul în fereastra lui.
 - *Logout*. Toți prietenii online ai utilizatorului văd în lista lor că utilizatorul nu mai este *online*.