

# Aplicații integrate pentru întreprinderi

## Laborator 5

09.11.2010

### Proiectarea de aplicații distribuite folosind mecanismul RMI

**Scopul laboratorului** îl reprezintă înțelegerea mecanismului RMI și distribuirea unei aplicații în funcționalități implementate în componente capabile să ruleze în mașini virtuale diferite, având un comportament vizibil „la distanță”.

1. Ce este RMI ?
2. Arhitectura mecanismului RMI
3. Noțiunea de clasă/obiect serializabil(ă)
4. Clase și interfețe „la distanță”
5. Programarea unui server
6. Programarea unui client

#### 1. Ce este RMI ?

**RMI** (Remote Method Invocation) este un mecanism care extinde funcționalitatea RPC<sup>1</sup> permițând invocarea unei metode a unui obiect care există într-un alt spațiu de adresă, ce poate fi pe aceeași mașină sau pe una diferită. RMI implementează comunicația la distanță dintre programe scrise în Java.

Obiectivul RMI s-a concentrat pe a pune la dispoziția programatorilor mijloacele pentru a dezvolta aplicații distribuite în același fel în care ar scrie aplicații nedistribuite (folosind o sintaxă și o structură semantică asemănătoare), încât folosirea de obiecte distribuite să fie similară cu utilizarea de obiecte locale, diferențele fiind sintetizate mai jos:

Concept	Obiect local	Obiect „la distanță”
<b>definirea obiectului</b>	se face într-o clasă Java	definirea comportamentului (exportat) al obiectului „la distanță” se face printr-o interfață care trebuie să extindă interfața <code>java.rmi.Remote</code> ;
<b>implementarea obiectului</b>	se face în clasa Java corespunzătoare	comportamentul obiectului accesat „la distanță” este executat printr-o clasă care implementează interfața
<b>crearea obiectului</b>	instanța unui obiect nou se face prin operatorul <code>new</code>	o instanță nouă pentru un obiect aflat „la distanță” se face folosind operatorul <code>new</code> pe mașina gazdă (clientul nu poate crea direct obiectul „la distanță”)
<b>accesul obiectului</b>	se realizează direct printr-o variabilă de tip referință	un obiect „la distanță” este accesat printr-o referință care indică spre o delegare a implementării interfeței

<sup>1</sup> RPC = Remote Procedure Call (apel de metodă la distanță), mecanism din care RMI a fost inspirat, extinzând funcționalitatea pe care acesta o pune la dispoziție. RMI este un mecanism RPC orientat obiect.

<b>referințe</b>	o referință indică direct spre obiectul din heap corespunzător	o referință „la distanță” indică spre un obiect delegat (stub) alocat local în heap; obiectul delegat conține informații care permite conectarea la obiectul „la distanță” care conține implementarea metodelor
<b>referințe active</b>	un obiect este considerat „activ” dacă există ce puțin o referință către el	într-un mediu distribuit unde mașinile virtuale pot manifesta erori fatale sau conexiunea poate fi pierdută, o referință este activă pentru un obiect la distanță dacă accesarea se face într-o anumită perioadă de timp (de închiriere); dacă pentru un obiect s-a renunțat (în mod explicit) la toate referințele sau toate referințele au expirat, obiectul „la distanță” este disponibil pentru modulul garbage collection distribuit.
<b>finalizarea</b>	metoda <code>finalize()</code> (în caz că este definită) se apelează înainte ca memoria aferentă obiectului să fie dezallocată (prin garbage collector)	dacă obiectul „la distanță” implementează interfața <code>Unreferenced</code> , metoda definită de această interfață este apelată când referințele la obiect sunt distruse
<b>garbage collection</b>	un obiect spre care nu mai există referințe (locale) devine candidat pentru garbage collector (pentru dezallocarea memoriei aferente)	există un modul garbage collector distribuit care lucrează cu modulul garbage collector local, apelându-se atunci când nu există referințe realizate „la distanță” și toate referințele locale pentru obiectul accesat „la distanță” au fost distruse
<b>excepții</b>	un program trebuie să trateze toate excepțiile (runtime sau alt tip)	pentru asigurarea robusteții aplicațiilor distribuite, programele trebuie să trateze excepțiile <code>RemoteException</code> care ar putea fi aruncate

## 2. Arhitectura mecanismului RMI

Proiectarea mecanismului RMI a vizat implementarea unui model de obiecte distribuite în Java care să se integreze în limbajul de programare și totodată să fie compatibil cu modelul de obiecte locale.

Aplicațiile RMI sunt compuse, de cele mai multe ori, din două programe separate, server și client.

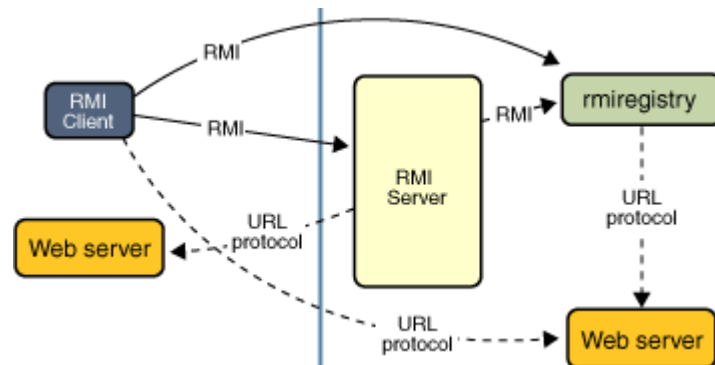
Un **server** crează niște obiecte la distanță, face accesibile referințele spre obiecte și așteaptă clienții să invoce metodele pe care le-au definit.

Un **client** obține o referință spre unul sau mai multe astfel de obiecte la distanță care se găsesc pe server și apoi invocă metodele pe ele.

RMI oferă un mecanism prin care serverul și clientul comunică și transferă informația în ambele sensuri. O astfel de aplicație este denumită cel mai frecvent **aplicație (cu obiecte) distribuite**.

Aplicațiile distribuite trebuie să realizeze următoarele operații: localizarea obiectelor la distanță, comunicarea cu obiectele la distanță, încărcarea definițiilor de clase pentru obiectele care sunt obținute.

1. *localizarea obiectelor la distanță* se poate face prin înregistrarea obiectelor „la distanță” la serviciul de numire RMI (rmiregistry)<sup>2</sup>;
2. *comunicarea cu obiectele la distanță* este realizată transparent de mecanismul RMI; pentru programator, comunicarea la distanță se face similar cu invocarea metodelor locale;
3. *încărcarea definițiilor de clase pentru obiectele care sunt obținute* este posibilă concomitent cu transmiterea datelor obiectelor, având în vedere faptul că obiectele pot fi comunicate bidirecțional între client și server;



Un exemplu de aplicație RMI

În exemplul de mai sus, serverul folosește serviciul de nume (rmiregistry) pentru a înregistra un obiect pe care apoi clientul îl găsește invocând același serviciu. Ulterior, clientul apelează o metodă a obiectului „la distanță”. Definițiile de clase sunt obținute pentru web de pe un server web (existent), în ambele direcții (de la server spre client și de la client spre server).

Unul dintre avantajele principale<sup>3</sup> puse la dispoziție de mecanismul RMI este obținerea definiției clasei unui obiect care nu este definit în mașina virtuală Java a clientului<sup>4</sup>. Transmiterea obiectului se face prin clasa sa, astfel încât comportamentul nu se modifică atunci când acesta este transmis către altă mașină virtuală. Astfel, noi atribute și comportamente sunt introduse într-o mașină virtuală aflată la distanță, îmbogățind astfel comportamentul aplicației.

Ca orice aplicație Java, o aplicație distribuită folosind mecanismul RMI conține interfețe (pentru a declara metode) și clase (pentru a implementa metodele definite în interfețe și, posibil, pentru a implementa metode noi). Unele implementări se pot găsi doar pe unele mașini virtuale. Obiectele invocate între mașini virtuale diferite sunt denumite obiecte „la distanță” (*eng.* remote objects).

Un obiect „la distanță” implementează o interfață (vizibilă „la distanță”) având următoarele caracteristici:

- extinde interfața `java.rmi.Remote`;
- fiecare metodă definită în interfață aruncă excepția `java.rmi.RemoteException`, în plus față de alte excepții;

Obiectele „la distanță” sunt tratate diferit (față de obiectele locale) prin mecanismul RMI atunci când sunt transmise de la o mașină virtuală la alta.

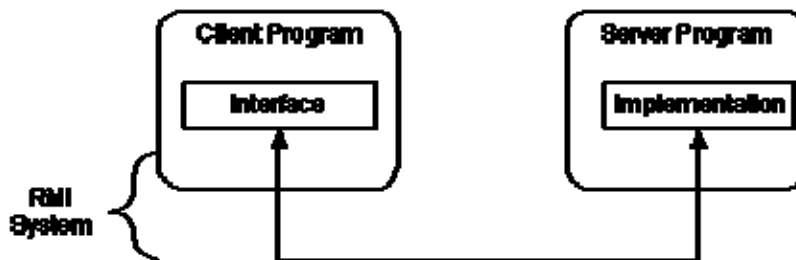
<sup>2</sup> Alternativ, o aplicație poate transfera obiecte „la distanță” prin intermediul unor invocări ale unor metode.

<sup>3</sup> Este în același timp și una dintre funcționalitățile unice ale mecanismului RMI.

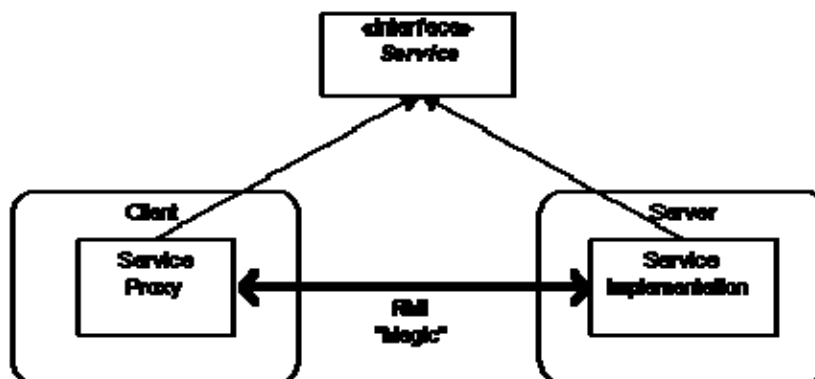
<sup>4</sup> Toate atributele și metodele unui obiect, disponibile până acum într-o singură mașină virtuală, pot fi transmise către o alta, posibil la distanță.

În loc să se facă o copie a implementării obiectului în mașina virtuală (client), mecanismul RMI transmite pentru obiectul la distanță un ciot (*eng.* stub) care are rolul de delegat (*eng.* proxy), reprezentant local al obiectului și reprezintă referința pentru acesta pe mașina virtuală (client). Astfel, se va invoca metoda pe ciotul local care este responsabil pentru execuția metodei invocate pentru obiectul la distanță.

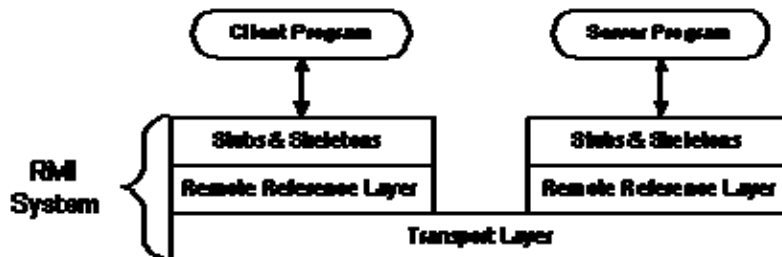
Un ciot pentru un obiect la distanță implementează același set de interfețe pe care le implementează însuși obiectul, proprietate ce permite ciotului să poată fi convertit la oricare dintre interfețele implementate de obiectul la distanță. Totuși, doar metodele definite în interfață sunt disponibile pentru a fi apelate de mașina virtuală (client).



Arhitectura RMI (1): Clientul cunoaște interfața obiectului „la distanță”, implementată pe server



Arhitectura RMI (2): Clientul dispune de un obiect delegat (*eng.* proxy) care implementează aceeași interfață ca obiectul „la distanță”, astfel încât pot fi apelate toate metodele obiectului



Arhitectura RMI (3): Obiectul delegat este format dintr-un ciot care conține metodele propriu-zise ale obiectului la distanță și o referință către acesta

**Etapele dezvoltării unei aplicații distribuite** folosind mecanismul RMI constau în:

1. proiectarea și implementarea componentelor aplicației distribuite; Proiectarea presupune stabilirea componentelor locale și ale componentelor accesibile la distanță.
  - a. *definirea interfețelor la distanță*, prin specificarea metodelor care pot fi apelate de client (nu și a implementării lor !); trebuie precizate tipurile obiectelor care vor fi folosite ca parametri și valori întoarse pentru metodele folosite<sup>5</sup>;
  - b. *implementarea obiectelor „la distanță”*; obiectele la distanță implementează una sau mai multe interfețe, unele fiind accesibile doar local; trebuie implementate și clasele (locale) care sunt folosite ca parametri sau valori întoarse pentru oricare dintre aceste metode;
  - c. *implementarea clientului* poate fi realizată oricând după definirea interfețelor la distanță, chiar și după configurarea obiectelor la distanță
2. compilarea surselor se face apelând compilatorul javac atât pentru interfețele la distanță, implementarea lor, alte clase pe server, cât și pentru clasele client<sup>6</sup>;
3. „publicarea” claselor pentru a fi accesibile prin rețea (este vorba despre interfețele la distanță și tipurile asociate precum și definițiile claselor care trebuie descărcate pe client sau pe server), de regulă, prin intermediul unui server web;
4. pornirea aplicației în ordinea: serviciul de nume RMI (rmiregistry), serverul și clientul.

Aplicațiile distribuite bazate pe mecanismul RMI sunt de regulă aplicații bazate pe comportamente căci sarcinile sunt încăcate dinamic în mașina virtuală (server) fără a exista cunoștințe în prealabil despre clasele care implementează sarcinile respective.

### 3. Noțiunea de clasă/obiect serializabil(ă)

Tehnologia RMI folosește mecanismul de serializare a obiectelor din Java pentru a transmite obiecte prin valoare între mașini virtuale diferite. Pentru ca un obiect să fie considerat serializabil, clasa sa trebuie să implementeze interfața `java.io.Serializable`.

Există două tipuri de clase care pot fi folosite cu mecanismul RMI:

1. clase Remote ale căror instanțe pot fi folosite la distanță, obiectele instanță ale acestei clase putând fi accesate în două moduri:
  - a. în spațiul de adrese în care a fost creat, obiectul fiind utilizat ca orice alt obiect (local);
  - b. în alte spații de adrese, obiectul va fi utilizat prin intermediul unui identificator, care impune unele limitări în accesarea sa comparativ cu metoda anterioară, dar care în linii generale permite folosirea obiectului ca și acum ar fi accesat local;
2. clase Serializable ale căror instanțe pot fi copiate între spații de adrese.

---

<sup>5</sup> Dacă nu se folosesc tipuri de bază ci tipuri ale unor obiecte definite de utilizator, acestea trebuie specificate de asemenea în acest moment.

<sup>6</sup> Crearea claselor ciot prin apelarea compilatorului `rmic`, necesară până la versiunile mai mici de Java 5, nu mai este obligatorie.

Dacă un obiect serializabil este dat ca parametru (sau valoare întoarsă) pentru un apel de metodă la distanță, valoarea obiectului va fi copiat dintr-un spațiu de adrese în altul. De asemenea, dacă un obiect la distanță este transmis ca parametru (sau valoare întoarsă), identificatorul său va fi copiat dintr-un spațiu de adrese în altul.

Majoritatea claselor standard sunt serializabile, deci o subclasă ale oricăroră dintre acestea este de asemenea serializabilă. Orice informație dintr-o clasă serializabilă ar trebui să fie serializabilă.

Folosirea unui obiect serializabil în apelul unei metode la distanță este foarte simplă. Se specifică obiectul dat ca parametru sau valoare întoarsă, tipul acestuia trebuind să fie o clasă serializabilă, fiind necesar ca atât clientul cât și serverul să aibă acces la definiția clasei serializabile care este utilizat, descărcarea definițiilor claselor serializabile de pe o mașină virtuală pe alta trebuind să fie specificată prin politici de securitate.



În cele ce urmează, implementarea unei aplicații distribuite folosind mecanismul RMI va fi ilustrată pe cazul particular al unui program cu un comportament asemănător cu Doodle ([www.doodle.com](http://www.doodle.com)).

## Doodle®

Doodle™ este o aplicație care permite planificarea unor activități în funcție de necesități specificate în mod particular.

În cadrul laboratorului, vom implementa o aplicație care simulează această funcționalitate, având ca scop alocarea de intervale de timp pentru studenții care vor să prezinte teme la „*Aplicații Integrate pentru Întreprinderi*”.

Aplicația va avea arhitectura client-server și are definite semnăturile pentru metodele pe care va trebui să le implementați.

Se specifică niște date calendaristice cu intervale orare aferente în care este posibilă prezentarea temelor, informații reținute de către aplicația server (`ServiciuProgramare.java`).

Fiecare student (aplicația client – `ObțineProgramare.java`) face o cerere pentru o perioadă de timp preferată (având orice durată). Dacă perioada de timp se încadrează în orarul specificat și intervalul dorit este liber (nu a fost alocat deja altui student), atunci cererea este aprobată și intervalul marcat ca ocupat.

Informațiile cu privire la orar sunt specificate într-un fișier text pe server (`orar.txt`) și încărcate în momentul lansării în execuție a acestuia. Nu se cere menținerea persistenței datelor. Cererile sunt specificate în linia de comandă având același format cu care au fost precizate în fișier: `ZZ LL AAAA O1 M1 O2 M2`.

#### 4. Clase și interfețe „la distanță”

Protocolul care permite ca serverul să fie interogat cu cereri de către client care să fie rulate apoi de server iar rezultatul să fie întors la client este realizat prin intermediul unei interfețe care descrie funcționalitatea care poate fi accesată la distanță.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Programare extends Remote
{
    public boolean ProgramareInterval (Interval i)
                                                throws RemoteException;
}
```

Prin extinderea interfeței `java.rmi.Remote` se specifică faptul că metodele declarate de interfața `Programare` vor putea fi apelate din altă mașină virtuală Java. Orice obiect care implementează interfața `Programare` va putea fi considerat un obiect „la distanță”.

`ProgramareInterval (Interval i)` (singura metodă definită în interfața `Programare`) este o metodă care va putea fi apelată „la distanță”, prin urmare trebuie să poată arunca excepții de tip `java.rmi.RemoteException`<sup>7</sup>.

Se observă că metoda `ProgramareInterval` primește un parametru de tip `Interval` și întoarce un rezultat de tip `boolean`. Pentru că `Interval` nu reprezintă un tip de bază, el trebuie definit tot acum.

```
import java.io.Serializable;
import java.util.GregorianCalendar;

public class Interval implements Serializable
{
    private static final long serialVersionUID = 1024L;

    GregorianCalendar inceput;
    GregorianCalendar sfarsit;

    public Interval ()
    {
        inceput = new GregorianCalendar();
        sfarsit = new GregorianCalendar();
    }

    public Interval
        (GregorianCalendar inceput, GregorianCalendar sfarsit)
    {
        this.inceput = inceput;
        this.sfarsit = sfarsit;
    }
}
```

---

<sup>7</sup> O astfel de excepție este aruncată de sistemul RMI dintr-un apel de metodă la distanță pentru indicarea situației în care a intervenit o eroare de comunicare sau de protocol. Codul sursă care apelează metode „la distanță” trebuie să trateze o astfel de excepție (prin prinderea ei sau aruncarea mai departe).

```
public GregorianCalendar obtineInceput ()
{
    return inceput;
}

public GregorianCalendar obtineSfarsit ()
{
    return sfarsit;
}
}
```

Câmpul `serialVersionUID` e folosit în procesul de deserializare al obiectelor încât dacă atributul obținut nu corespunde cu cel specificat în definiția clasei, este aruncată o excepție de tipul `InvalidClassException`.

Obiectele de tip `Interval` vor fi transmise între client și server astfel încât ele trebuie să fie serializabile (asigurând astfel consistența informațiilor reținute în obiect indiferent de formatul de reprezentare din mașinile virtuale). De aceea, clasa `Interval` implementează interfața `Serializable` (din pachetul `java.io`) și pune la dispoziție un constructor de copiere și două metode de tip `get`<sup>8</sup>.

Compilarea claselor se face în mod obișnuit:

```
> javac Programare.java Interval.java
```

Deoarece vom avea nevoie de cele două clase atât pe client cât și pe server este bine să le includem într-o arhivă de tip `jar` pe care să o adăugăm în `classpath` atât la compilare cât și la rulare.

```
> jar cvf programare.jar Programare.class Interval.class
```

## 5. Programarea unui server

Serverul trebuie să implementeze interfața „la distanță” vizibilă în rețea astfel încât comportamentul obiectelor care vor fi accesate de client să fie definit (obiectele „la distanță” sunt obținute prin instanțierea clasei `Server`). Pe server pot fi definite și alte metode (în afara celei impuse de interfață) care vor putea fi accesate local, între care constructorul clasei și metoda `main`.

Argumentele sau valorile întoarse din metodele „la distanță” (definite în interfață) pot fi de orice tip, inclusiv obiecte locale, obiecte „la distanță” sau chiar tipuri primitive de date. Astfel, orice entitate având orice tip poate fi transmisă către sau de la o metodă „la distanță” cât timp aceasta este o instanță a unui tip care este fie primitiv, fie obiect „la distanță” sau obiect local serializabil (așa cum s-a precizat și mai sus). Alte tipuri de obiecte, cum ar fi fire de execuție sau descriptori de fișier, având sens doar în spațiul de adrese în care rulează, nu pot să fie utilizați ca argumente sau valori întoarse în metode „la distanță”. Totuși, majoritatea claselor din pachetele `java.lang` și `java.util` implementează interfața `java.io.Serializable`.

Obiectele „la distanță” sunt transmise prin referință (un ciot care este delegatul la nivelul clientului și care implementează setul complet de interfețe accesibile prin rețea specificate de obiect).

---

<sup>8</sup> Metodele nu mai trebuie să arunce excepția `java.rmi.RemoteException` pentru că nu este extinsă interfața `Remote`.



Sunt vizibile doar metodele definite în interfața „la distanță” pentru obiectul respectiv. Metodele definite pentru obiect care nu sunt definite în interfață nu vor putea fi accesate la distanță.

Orice modificare realizată asupra stării obiectului prin apelurile la distanță sunt reflectate în obiectul original (de pe server).

Obiectele locale sunt transmise prin valoare (este realizată o copie folosind mecanismul de serializare). Sunt copiate toate câmpurile care nu au precizat atributul `static` sau `transient`<sup>9</sup>. Orice modificare realizată în starea obiectului sunt reflectate doar în copie, dar nu și în instanța originală. Similar, schimbările asupra stării obiectului în instanța originală nu vor fi reflectate în copie.

```
public class ServiciuProgramare implements Programare
{
    ...

    public ServiciuProgramare()
    {
        super(); // instructiune ce poate fi omisa
        ...
    }

    public boolean ProgramareInterval (Interval interval)
                                                throws RemoteException
    {
        boolean rezultat = false;
        // TO DO
        return rezultat;
    }

    ...

    public static void main (String[] args)
    {
        if (System.getSecurityManager() == null)
        {
            System.setSecurityManager(new SecurityManager());
        }

        try
        {
            String nume = "ServiciuProgramare";
            Programare serviciu = new ServiciuProgramare();
            Programare delegat =
                (Programare)
                UnicastRemoteObject.exportObject (serviciu, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind (nume, delegat);
            System.out.println ("legare realizata cu succes");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

---

<sup>9</sup> Modul în care este realizat comportamentul implicit de serializare poate fi modificat.

Metoda `main` este cea mai complexă, fiind folosită pentru a porni serverul, realizând totodată și inițializările necesare pentru a pregăti serverul să accepte conexiuni dinspre clienți. Metoda nu poate fi apelată din altă mașină virtuală, pentru că nu extinde interfața `Remote`. Fiind statică, ea este asociată cu clasa și nu cu obiectul.

1. Este creat și instalat **un sistem de securitate** care protejează accesul la resursele sistemului de cod sursă (descărcat) care rulează în mașina virtuală Java. Acest sistem de securitate stabilește dacă codul sursă poate să acceseze sistemul local de fișiere sau poate realiza alte operații privilegiate.

În situația în care aplicația distribuită care folosește tehnologia RMI nu dispune de un sistem de securitate, nu vor fi descărcate clase (altele decât din sistemul de fișiere specificat în `classpath`) pentru obiectele primite ca argumente sau valori întoarse.

Se asigură totodată faptul că operațiile realizate de codul descărcat sunt conforme cu o politică de securitate.

2. Se creează o **instanță** a clasei `ServiciuProgramare` și este exportată spre RMI folosindu-se instrucțiunea `UnicastRemoteObject.exportObject`<sup>10</sup> astfel încât obiectul să poată fi apelat de către clienți de la distanță. La întoarcerea din execuția acestei metode, obiectul poate procesa apeluri „la distanță”.

Este întors un ciot (*eng.* stub) care are tipul `Programare`, deci tipul interfeței și nu al clasei (pentru că ciotul implementează doar interfața „la distanță”).

3. Înainte ca un client să poată invoca metodele unui obiect „la distanță”, trebuie să obțină referința spre acesta, operație ce poate fi realizată în același fel în care este obținută o referință spre un obiect într-o aplicație (ca valoare întoarsă pentru o metodă sau ca un câmp al unei structuri de date care conține referința).

Mecanismul RMI pune la dispoziție un tip particular de obiect „la distanță” serviciul de nume (RMI registry), pentru găsirea de referințe către alte obiecte de acest tip prin specificarea unui nume. Un astfel de sistem este în mod obișnuit folosit doar pentru găsirea primului obiect solicitat de client, putând ajuta la găsirea altor obiecte.

Interfața `java.rmi.registry.Registry` conține specificațiile pentru **înregistrarea** și găsirea de obiecte „la distanță” în serviciul de nume. Clasa `java.rmi.registry LocateRegistry` pune la dispoziție metode statice pentru identificarea unei referințe la distanță într-o rețea anume (specificată prin adresă și port). Metodele creează referința către obiectul „la distanță” conținând adresa de rețea specificată fără a se realiza comunicare la distanță<sup>11</sup>.

După ce obiectul „la distanță” a fost înregistrat de sistemul de nume, clienții de pe orice mașină îl pot căuta după nume, îi pot obține referința și pot apela metodele „la distanță” de pe el<sup>12</sup>.

Metoda `rebind` reprezintă un apel la distanță către serviciul de nume RMI și poate avea ca rezultat aruncarea unei excepții de tip `RemoteException`.

---

<sup>10</sup> Cel de-al doilea argument al metodei `exportObject` reprezintă un port TCP utilizat pentru ascultarea apelurilor metodelor la distanță de către client. Valoarea 0, folosită în exemplul de față specifică utilizarea unui port anonim, ales de RMI sau de sistemul de operare.

<sup>11</sup> Clasa `java.rmi.registry LocateRegistry` pune la dispoziție metode statice pentru crearea unui serviciu de nume nou în mașina virtuală Java.

<sup>12</sup> Serverele de pe o mașină pot partaja același serviciu de nume sau fiecare aplicație (de tip server) poate să își creeze și să utilizeze propriul său sistem de nume.

Metoda `LocateRegistry.getRegistry` este apelată fără parametri, obținându-se o referință la serviciul de nume de pe mașina locală și portul 1099, definit implicit<sup>13</sup>.

Când se face apelul la distanță către serviciul de nume, se obține un obiect de tip ciot (în loc de copia obiectului) pentru că obiectele care implementează interfețe de tip `Remote` rămân în cadrul mașinii virtuale Java în care au fost create. Astfel, atunci când un client realizează o căutare în serviciul de nume al serverului, se întoarce o copie a obiectului de tip ciot, obiectele „la distanță” fiind transmise deci prin referință.

Din motive de securitate, o aplicație poate (re)construi sau distruge legătura dintre o referință a unui obiect la distanță și serviciul de nume care rulează pe aceeași mașină, prevenind situația în care o altă mașină ar înlătura sau suprascrise intrările într-un serviciu de nume.

Operația de căutare (lookup) poate fi realizată de pe orice mașină, locală sau „la distanță”.

Excepțiile ce pot fi aruncate de metoda `main` sunt de tip `RemoteException`, generate fie de metoda `UnicastRemoteObject.exportObject` sau de apelul `rebind`. Se poate realiza recuperarea din eroare (în caz că o excepție a fost aruncată) prin reapelarea metodei care a generat-o sau prin folosirea unui alt server.

După realizarea acestor operații, metoda `main` se încheie. Nu este necesară definirea unui fir de execuție care să țină serverul în starea de rulare, pentru că atât timp cât există o referință către un obiect de tip `ServiciuProgramare` într-o mașină virtuală (locală sau la distanță), el nu va fi distrus de garbage collector. Mecanismul RMI menține activ procesul asociat obiectului `ServiciuProgramare`, pentru că există o legătură către el în serviciul de nume, accesibilă la distanță. Obiectul va fi distrus atunci când legătura dintre el și serviciul de nume va fi eliberată și nu vor mai exista clienți la distanță care să aibă referințe către el.

Compilarea claselor se face în mod obișnuit:

```
> javac -cp programare.jar ServiciuProgramare.java
```

Lansarea în execuție nu poate fi realizată decât după:

- pornirea serviciului de nume, care permite clienților să obțină o referință către un obiect la distanță<sup>14</sup>
  - Windows

```
> start rmiregistry [port]
```
  - Linux

```
> rmiregistry [port] &
```
- specificarea unei politici de securitate, într-un fișier numit `policy[.txt]`, codul sursă obținând permisiunile de care are nevoie pentru a putea rula:

```
grant codeBase "file:///<cale_absoluta>\\Server" {
    permission java.security.AllPermission;
};
```

Toate permisiunile sunt acordate claselor din calea locală a aplicației pentru că acesta are un grad de încredere limitat, dar nu se acordă drepturi pentru codul sursă descărcat din alte mașini virtuale.

---

<sup>13</sup> Dacă serviciul de nume a fost creat pe un alt port, se folosește o metodă supraîncărcată care primește un parametru de tip `int` care specifică portul.

<sup>14</sup> Parametrul port este opțional, el poate să nu fie menționat, caz în care serviciul de nume folosește portul implicit 1099.

Lansarea în execuție a serverului se face prin comanda<sup>15</sup>:

```
> java -cp .;programare.jar
-Djava.rmi.server.codebase=file:///<cale_absoluta>/programare.jar
-Djava.rmi.server.hostname=localhost
-Djava.security.policy=policy
ServiciuProgramare
```

Proprietatea `java.rmi.server.codebase` specifică locația de unde pot fi descărcate definiții pentru clasele provenind de la server<sup>16</sup>.

Proprietatea `java.rmi.hostname` specifică numele mașii sau adresa care urmează a fi completată în obiectele de tip `ciot` care vor fi exportate. Aceeași valoare va fi folosită de clienți când vor încerca să apeleze metodele la distanță. Implicit, implementarea RMI folosește adresa IP indicată de `java.net.InetAddress.getLocalHost`. Totuși, câteodată o astfel de adresă nu este întotdeauna potrivită pentru toți clienții și un nume (al mașinii) este mai adecvat. Pentru ca numele să fie vizibil de toți clienții, trebuie configurată proprietatea `java.rmi.server.hostname`.

Proprietatea `java.rmi.policy` specifică politica care conține permisiunile ce vor fi acordate.

## 6. Programarea unui client

Codul care apelează metodele unui obiect de tip `Programare` trebuie să obțină o referință către acesta.

Ulterior, va crea un obiect de tip `Interval` (eventual din parametrii specificați în linie de comandă) pe care îl va transmite ca argument metodei `ProgramareInterval`.

1. Ca și în cazul serverului, inițial se va instala un **sistem de securitate**, necesar pentru că în procesul de obținere a referinței către obiectul `ciot` corespunzător obiectului la distanță se poate întâmpla să se descarce definiții de clase de la server, lucru care este posibil doar în condițiile existenței unui sistem de securitate.

2. Clientul va **căuta** obiectul „la distanță” specificând un nume (același folosit de server pentru a înregistra obiectul la serviciul de nume) – în acest sens, e folosită de asemenea metoda `LocateRegistry.getRegistry` pentru a obține referința la sistemul de nume RMI care rulează pe mașina server, parametrul cu care este apelat reprezentând numele mașinii<sup>17</sup> pe care rulează obiectul de tip `Programare`. Apoi este utilizată metoda `lookup` pentru a căuta obiectul la distanță prin nume în sistemul de nume de pe mașina server.

3. Se va crea un obiect de tip `Interval` (eventual din parametrii specificați în linia de comandă) și se va **apela metoda la distanță** `ProgramareInterval`, afișându-se rezultatul obținut.

---

<sup>15</sup> Pe Linux, caracterul `;` va fi înlocuit cu `:`.

<sup>16</sup> Dacă în locul unei arhive se specifică o ierarhie de directoare, aceasta trebuie încheiată prin caracterul `'`.

<sup>17</sup> În mod implicit, se consideră că sistemul de nume RMI ascultă pe portul 1099. În plus, poate fi folosită o metodă supraîncărcată care să specifice și portul, prin specificarea unui parametru de tip `int`.

```

public class ObtineProgramare
{
    ...

    public static void main (String[] args)
    {
        if (System.getSecurityManager() == null)
        {
            System.setSecurityManager(new SecurityManager());
        }

        try
        {
            String nume = "ServiciuProgramare";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Programare programare =
                (Programare)registry.lookup (nume);

            Interval interval = new Interval (...);

            if (programare.ProgramareInterval(interval))
                System.out.println ("succes");
            else
                System.out.println ("insucces");

        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Compilarea claselor se face în mod obișnuit:

```
> javac -cp programare.jar ObtineProgramare.java
```

Lansarea în execuție nu poate fi realizată decât după:

- pornirea serviciului de nume RMI pe mașina server
- lansarea în execuție a aplicației server ServiciuProgramare
- specificarea unei politici de securitate, într-un fișier numit `policy[.txt]`, codul sursă obținând permisiunile de care are nevoie pentru a putea rula:

```

grant codeBase "file:///<cale_absoluta>\\Client" {
    permission java.security.AllPermission;
};

```

Lansarea în execuție a serverului se face prin comanda:

```

> java -cp .;programare.jar
-Djava.rmi.server.codebase=file:///<cale_absoluta>/
-Djava.security.policy=policy
ObtineProgramare <adresa> 09 11 2010 8 00 10 00

```

Se observă că prin proprietatea `java.rmi.server.codebase` s-a precizat locația unde clientul își definește clasele, iar prin proprietatea `java.security.policy` s-a specificat fișierul care conține permisiunile acordate pentru diferite coduri sursă.

## Activitate de laborator

(9p) 1. Completați în mod corespunzător clasele `ServiciuProgramare.java` și `ObtineProgramare.java` astfel încât să se constituie într-o aplicație distribuită funcțională.

(1p) 2. Formați o echipă împreună cu un coleg și rulați cele două componente din mașini virtuale Java diferite (aveți grijă ca adresele celor două mașini să fie în același spațiu, astfel încât să fie vizibile între ele). Pe o mașină se va rula serverul, iar pe altă mașină clientul care va face diverse cereri.

(2p) 3. Să se implementeze o metodă `Remote` care să afișeze intervalele ocupate.

(1p) 4. Să se implementeze o metodă `Remote` care să permită anularea rezervării pentru un interval (nu se va face verificarea ca același client care a făcut cererea de rezervare să realizeze și anularea).