Serializare şi Clonare
RMI

Partea I

Serializare şi clonare

Remote Method Invocation (RMI)

Conţinut

- Serializarea obiectelor
- Serializare şi clonare
- Remote Method Invocation (RMI)

Serializarea obiectelor

- Serializarea: salvarea unui obiect sub forma unei secvenţe de octeţi, care poate fi scrisă într-un fişier, transmisă prin reţea etc.
- Persistenta: obiectul “supravieţuieşte” între mai multe execuţii ale unui program.
- Serializarea este o forma *lightweight* de persistenţă: operaţia nu se face în mod implicit, ci trebuie realizata de programator.

De ce e nevoie de serializare?

- Salvarea “de mână” într-un fişier a valorilor câmpurilor unui obiect nu e suficientă.

- Mecanismul de serializare oferă:
  - Dacă obiectul conţine referinţe la alte obiecte: salvarea automată, recursivă, a tuturor obiectelor necesare (*web of objects*)
  - Salvarea informaţiilor despre clasa obiectului

Interfeţe Java pentru serializare

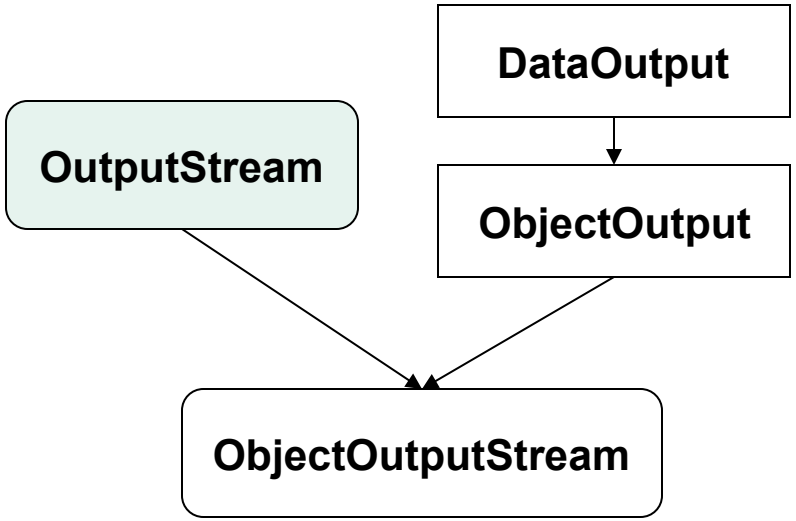
- `java.io.Serializable`
  - Interfaţa care trebuie implementata de o clasa pentru ca instanţele să poată fi serializate.
  - Nu conţine nici o metoda.
- `java.io.Externalizable`
  - Extinde interfaţa `Serializable`
  - Se implementează daca dorim alte operaţii decât cele implicite la serializare.
  - Daca o clasa nu implementează `Serializable` sau `Externalizable` şi se încearcă serializarea unei instantâţe: `NotSerializableException`

```
void writeExternal (ObjectOutput out) throws IOException;

void readExternal (ObjectInput in) throws IOException,
                                          ClassNotFoundException;
```

Clase utilizate pentru serializare

- `java.io.ObjectInputStream, java.io.ObjectOutputStream`



Metode pentru serializare:

```
public void writeObject (Object obj) throws IOException;

public Object readObject () throws ClassNotFoundException,
IOException;
```

Modificatorul `transient`

- Folosit pentru câmpurile pe care nu dorim să le salvăm la serializare.

- Exemple de utilizare:
  - Pentru informaţii confidenţiale (parole);
  - Pentru informaţii care nu au sens pe altă maşină virtuală (referinţe la fire de execuţie).

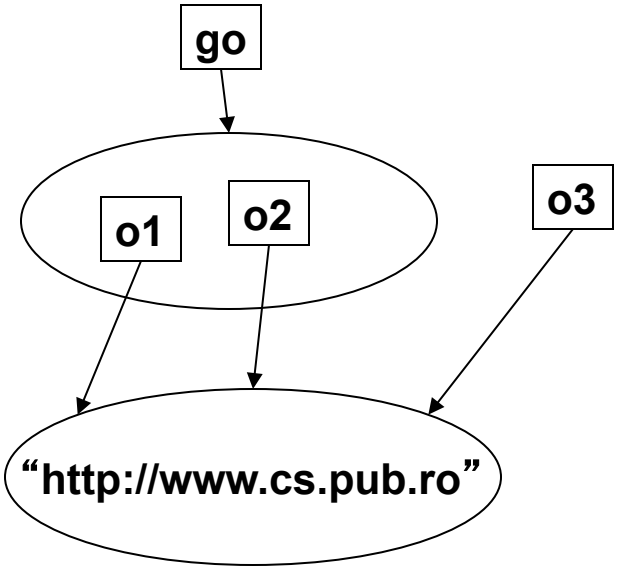
Serializarea informaţiilor despre clasă

- La serializarea unui obiect se salvează şi numele şi signatura clasei acestuia
- La reconstituirea unui obiect serializat se utilizează clasa cu acelaşi nume de pe maşina unde se face de-serializarea.
- Cum se verifică dacă cele două clase sunt compatibile?
  - Default: se tine cont de numele clasei, câmpuri, interfeţele implementate etc.
  - Programatorul poate defini câmpul `serialVersionUID` – se va verifica daca ID-urile coincid:

```
private static long serialVersionUID = 123456789L;
```

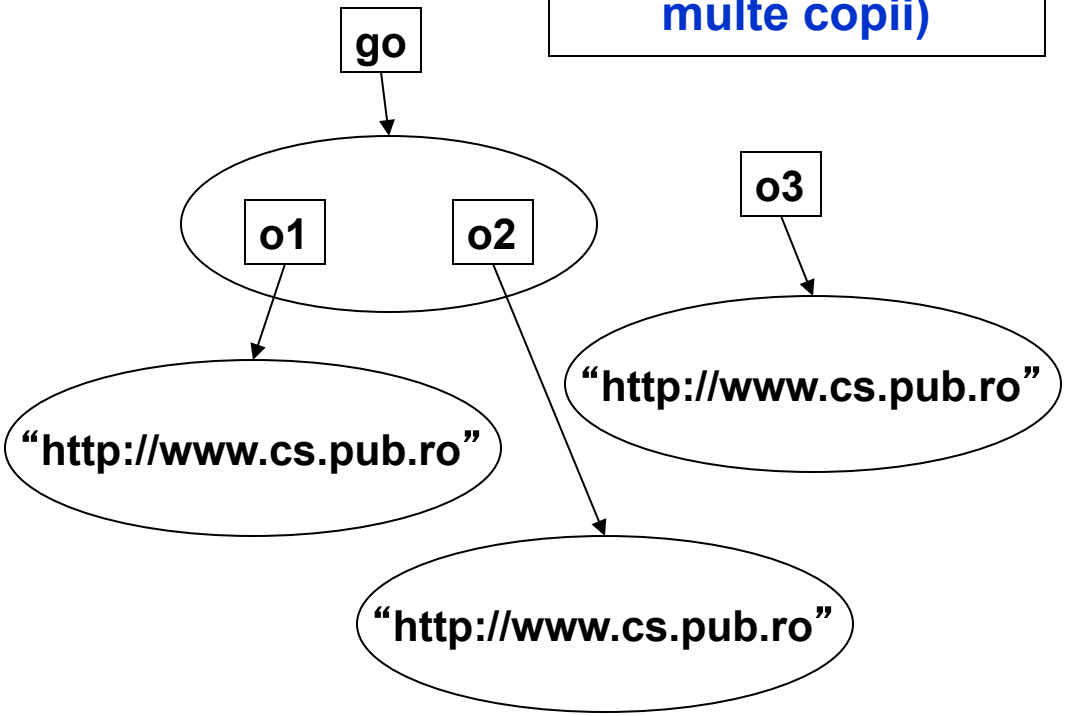
Serializarea grafurilor de obiecte [Ath02]

- Dacă există mai multe referinţe către un obiect, la serializare se fac mai multe copii ale acestuia sau una singură?



Graf initial

Salvare gresita (mai multe copii)



Exemplu – salvarea unui graf de obiecte [Ath02]

```
import java.io.*;
import java.net.*;
public class GrafOb implements Serializable {
  public URL o1;
  public URL o2;
  private static final long serialVersionUID = 241020081745L;
  public GrafOb(URL o1, URL o2) {
    this.o1 = o1;
    this.o2 = o2;
  }
  public String toString() {
    return new String("o1:" + o1 + " o2:" + o2);
  }
}
```

Salvarea unui graf de obiecte [Ath02] (2)

```
class SalvareObiect {
  static GrafOb go;
  public static void main(String args[]) {
    try {
      System.out.println("Salvare obiect");
      FileOutputStream fout = new FileOutputStream("test");
      ObjectOutputStream sout = new ObjectOutputStream(fout);
      URL o1 = new URL("http://www.acs.pub.ro");
      URL o2 = o1, o3 = o1;
      go = new GrafOb(o1, o2);
      sout.writeObject(go); sout.writeObject(o3);
      sout.flush();
      System.out.println("S-a scris obiectul " + go);
      System.out.println("S-a scris obiectul " + o3);
      System.out.println("(go.o1==go.o2)&&(go.o1==o3) este " +
        ((go.o1==go.o2)&&(go.o1==o3)));
      sout.close(); fout.close();
    } catch (Exception e) {
      System.out.println("Eroare: " + e);
    }
  }
}
```

Salvarea unui graf de obiecte [Ath02] (3)

```
class RefacereObiect {
  static GrafOb go;
  public static void main(String args[]) {
    try {
      System.out.println("Restaurare obiect");
      FileInputStream fin = new FileInputStream("test");
      ObjectInputStream sin = new ObjectInputStream(fin);
      go = (GrafOb)sin.readObject();
      System.out.println("S-a citit obiectul: " + go);
      URL o3 = (URL)sin.readObject();
      System.out.println("S-a citit obiectul: " + o3);
      System.out.println("(go.o1==go.o2)&&(go.o1==o3) este " +
        ((go.o1==go.o2)&&(go.o1==o3)));
      sin.close();
      fin.close();
    } catch (Exception e) {
      System.err.println("Eroare: " + e);
    }
  }
}
```

Rezultatele execuţiei exemplului

```
S-a scris obiectul: o1: http://www.acs.pub.ro/ o2: http://www.acs.pub.ro
S-a scris obiectul: http://www.acs.pub.ro/
(go.o1==go.o2)&&(go.o1==o3) este true

Restaurare obiect
S-a citit obiectul: o1: http://www.acs.pub.ro/ o2: http://www.acs.pub.ro
S-a citit obiectul: http://www.acs.pub.ro/
(go.o1==go.o2)&&(go.o1==o3) este true
```

- Concluzie: grafurile de obiecte sunt serializate corect.
- În cadrul unui singur stream de serializare, un obiect este salvat o singura dată.

Soluţii pentru redefinirea operaţiei de serializare (1)

- Pentru a scrie altă metoda de serializare decât cea implicita – în clasa se definesc metodele:

```
private void writeObject(ObjectOutputStream stream) throws
IOException;
private void readObject(ObjectInputStream stream) throws
IOException, ClassNotFoundException;
```

- Aceste metode vor fi utilizate in locul celor implicite
- Pot arunca `java.io.NotSerializableException`
- Din cadrul acestor metode se pot apela metodele default: `defaultReadObject()` si `defaultWriteObject()` din clasele `ObjectInputStream`, `ObjectOutputStream`

Soluţii pentru redefinirea operaţiei de serializare (2)

- Extinderea interfeţei `Externalizable`.
- Daca o clasa extinde `Externalizable`, la serializare se vor apela `writeExternal()/readExternal()` (ignorându-se eventual `writeObject()/ readObject()`).
- Programatorul trebuie sa asigure:
  - Salvarea câmpurilor clasei şi superclaselor
  - Salvarea corecta a grafurilor de obiecte
  - Verificarea că versiunile de clase coincid
- Identitatea clasei este salvată automat.
- La reconstituirea obiectului: este apelat constructorul implicit, apoi `readExternal()`..

Exemplu Externalizable [Eck02] (1)

```
import java.io.*;
import java.util.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in nondefault constructor.
  }
  public String toString() { return s + i; }
  public void writeExternal(ObjectOutput out)
  throws IOException {
    System.out.println("Blip3.writeExternal");
    out.writeObject(s);
    out.writeInt(i);
  }
```

Exemplu Externalizable [Eck02] (2)

```
  public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    s = (String)in.readObject();
    i = in.readInt();
  }
  public static void main(String[] args)
  throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    ObjectOutputStream o = new ObjectOutputStream(
      new FileOutputStream("Blip3.out"));
    System.out.println("Saving object:");
    o.writeObject(b3);
    o.close();
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("Blip3.out"));
    System.out.println("Recovering b3:");
    b3 = (Blip3)in.readObject();
    System.out.println(b3);
  }
}
```

Ce se întâmplă la serializare într-o ierarhie de clase?

- Dacă un obiect este o instanţă a clasei C2, care extinde alta clasă C1:
  - Dacă ambele clase sunt serializabile: se vor salva atât câmpurile din C2, cât şi cele din C1 (în afară de cele `static` şi `transient`).
  - Daca C1 nu e serializabilă: nu se serializează câmpurile acestei clase; se pot scrie funcţii `readObject()/writeObject()` în C2 care să salveze explicit câmpurile din C1.

Conţinut

- Serializarea obiectelor
- Serializare şi clonare
- Remote Method Invocation (RMI)

Utilizarea clonării

- În Java, o atribuire între obiecte setează doar referinţa, nu copiază efectiv obiectele (rezultă un singur obiect cu 2 referinţe).
- Comportament similar pentru obiectele date ca parametri în funcţii
- Clonarea – pentru a copia efectiv obiectele:
  - Interfaţa `java.lang.Cloneable`
  - Definirea `metodei clone()`:

```
public Object clone();
```

Clonarea - probleme

- Dacă un obiect conţine referinţe la mai multe obiecte, acestea trebuiesc şi ele clonate.

- Exemplu: deşi clasa `ArrayList` conţine o metoda de clonare, aceasta nu clonează în mod recursiv elementele array-ului (se copiază doar referinţele).

- Soluţie pentru clonarea obiectelor complexe: serializarea.

Exemplu – Clonare prin serializare [Ath02] (1)

```
import java.io.*;
class Clonare implements Cloneable {
  public Object clone() {
    try {
      ByteArrayOutputStream bout = new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bout);
      out.writeObject(this);
      out.close();
      ByteArrayInputStream bin = new
        ByteArrayInputStream(bout.toByteArray());
      ObjectInputStream in = new ObjectInputStream(bin);
      Object ret = in.readObject();
      in.close();
      return ret;
    } catch (Exception e) {
      System.out.println(e);
      return null;
    }
  }
}
```

Exemplu – Clonare prin serializare [Ath02] (2)

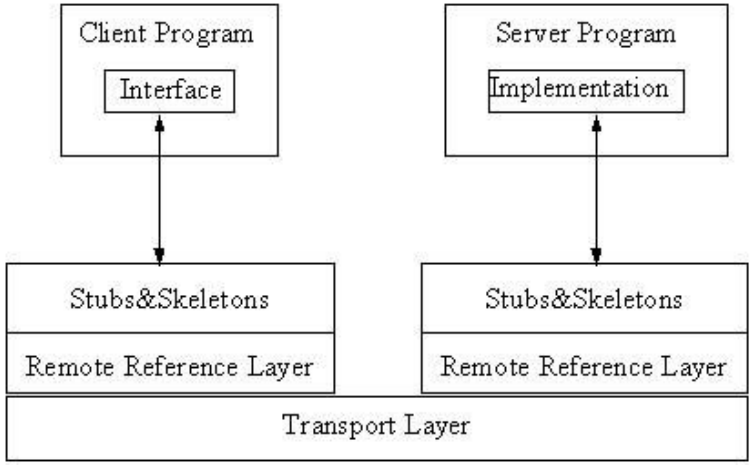
```
class Obiect implements Serializable {
  …
}
class CevaC extends Clonare implements Serializable {
  …
}
public class TestClone {
  public static void main(String[] args) {
    CevaC cc1 = new CevaC("AAAA");
    CevaC cc2 = (CevaC)cc1.clone();
    System.out.println("Info despre cc1");
    cc1.afiseaza();
    cc1.modifica("CCCCCCCCCC");
    System.out.println("Info despre cc1");
    cc1.afiseaza();
    System.out.println("Info despre cc2");
    cc2.afiseaza();
  }
}
```

Conţinut

- Serializarea obiectelor
- Serializare şi clonare
- Remote Method Invocation (RMI)

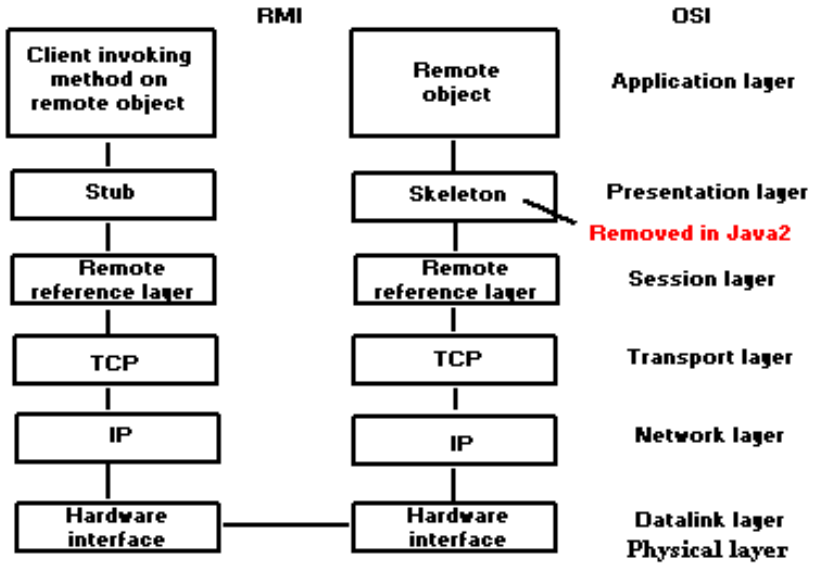
RMI - Introducere

- Extindere a mecanismului RPC
- Se bazează pe modelul client-server
- Mecanism:
  - În maşina virtuală a clientului se foloseşte o clasă care reprezintă server-ul (*stub*)
  - În maşina virtuală a serverului se foloseşte o clasă care reprezintă clientul (*skeleton*)
  - *Stub*-ul şi *skeleton*-ul au rolul de a construi mesajele care se transmit prin reţea



Arhitectura RMI

- Nivele:
  - *stub/skeleton*
  - nivelul referinţelor la distanţă
  - nivelul transportului
- Nivelele sunt independente.
- Obiectele *stub* şi *skeleton* sunt generate pornind de la clasa care implementează serverul.



Condiţii pentru utilizarea RMI

- Mecanismul nu este total transparent.
- Pe maşina clientului trebuie cunoscută interfaţa oferita de server.
- Obiect la distanta (*remote*): obiect aflat pe server, pentru care clientul va apela metodele din interfaţă.
- Obiectele remote sunt diferite de cele obişnuite:
  - `hashCode()` trebuie să producă acelaşi rezultat pentru diferite referiri ale aceluiaşi obiect aflat la distanţă;
  - alt algoritm pentru colectarea memoriei disponibile.

Etape în utilizarea RMI

1. Definirea interfeţei prin care va fi accesat obiectul *remote.*
2. Definirea unei clase care să implementeze interfaţa => obiect *remote.*
3. Generare *stub* şi *skeleton* cu programul rmic.
4. Înregistrarea obiectului *remote* la un serviciu de nume.
5. Clientul interoghează serviciul de nume şi primeşte *stub*-ul.
6. Clientul poate apela metodele obiectului *remote* cu ajutorul *stub*-ului.

Clase şi interfeţe pentru RMI

- Interfaţa `java.rmi.Remote`
  - Trebuie extinsă de orice obiect remote
  - Nu conţine nici o metodă
  - Toate metodele declarate într-o implementare a interfeţei trebuie să anunţe că pot produce excepţia `java.rmi.RemoteException`
- Implementare disponibila pentru `Remote`: `RemoteObject`
  - Clasă abstractă ce poate fi extinsă de obiecte remote
  - Metodele `hashCode()`, `equals()` si `toString()` sunt conforme cu semantica obiectelor la distanţă
  - Subclasa: `RemoteServer`
- Clase ce extind `RemoteServer`: `UnicastRemoteObject`, `Activatable`

Utilitare pentru RMI

- `rmic` – pentru generare de stub şi skeleton:
  - Utilizare: `rmic MyServerImplem`
  - `MyServerImplem` este implementarea pentru interfaţa remote
  - Rezultă `MyServerImplem_Skel.class` şi `MyServerImplem_Stub.class`
- `rmiregistry` – pentru pornirea serviciului de nume
  - Utilizare: `rmiregistry&`
  - Pentru alt port decât cel default (1099):

```
rmiregistry <port>
```

Partea II

RMI: Particularitati, exemple de aplicatii, colectarea memoriei

Conţinut

- Particularităţi RMI
- Exemple şi aplicaţii
- Activarea obiectelor la distanţă
- Colectarea memoriei disponibile
- Avantaje & dezavantaje RMI

Facilitaţi RMI în J2SE 1.5

- Nu mai este necesară utilizarea `rmic` pentru generarea *stub*-ului si *skeleton*-ului
- Generarea se realizeaza automat la apelul metodei `exportObject()` pentru server.
- Utilizarea `rmic` este necesara numai daca vor exista programe client realizate cu versiuni anterioare ale J2SE

Transferul de argumente pentru RMI

- Pentru obiecte *remote*:
  - se transmite un stub
  - pe client trebuie să existe interfaţa implementată de obiect
- Pentru obiecte “obişnuite”:
  - copiere bazată pe serializare
  - clasa trebuie să fie serializabilă
  - dacă pe client nu există clasa respectivă, va fi importată de pe server

Incărcărea dinamică a claselor în RMI

- Adnotarea claselor: adăugarea, la serializare, de informaţii despre locaţia clasei
- Proprietatea **`java.rmi.server.codebase`**
  - specifica un URL de unde se pot încărca clase
  - poate fi utilizata si la client, si la server
  - daca este utilizata la server, clasele vor fi adnotate cu URL-ul specificat
- **`java.rmi.server.useCodebaseOnly`**
  - încărcarea claselor se face numai din locaţia indicată de codebase
- Exemplu:

```
java -Djava.rmi.server.codebase=http://abc.pub.ro/~user/SDir/
clasaDeStartServer
```

Procesul de încărcare dinamică [Gab99]

- La server:
  - adnotarea claselor: dacă clasa a fost încărcată de la o adresa cunoscută sau dacă este specificată proprietatea *codebase*
  - altfel clasa nu va fi adnotată
- La client:
  - Daca *stub*-ul clasei server există local şi este în CLASSPATH, va fi utilizat; altfel este obţinut de la *registry*
  - Pentru alte clase necesare: rezolvarea – proces invers adnotării
  - La rezolvare se iau în considerare în ordine: CLASSPATH-ul local, adnotarea clasei, proprietatea codebase, un încărcător contextual

Conţinut

- Particularităţi RMI
- Exemple şi aplicaţii
- Activarea obiectelor la distanţă
- Colectarea memoriei disponibile
- Avantaje & dezavantaje RMI

Exemplu 1 – obţinerea unei referinţe către un obiect remote [Ath02]

```
import java.rmi.*;
public interface ServerPisicaInterf extends java.rmi.Remote {
  PisicaInterf referintaPisica() throws java.rmi.RemoteException;
}
```

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ServerPisicaImplem extends UnicastRemoteObject
    implements ServerPisicaInterf {
  Pisica p = null;
  ServerPisicaImplem(Pisica p) throws java.rmi.RemoteException {
    super();
    this.p = p;
  }
  public PisicaInter referintaPisica()
    throws java.rmi.RemoteException {
    System.out.println("Se transmite " + p.getClass());
    return p;
  }
}
```

Exemplu 1 – obţinerea unei referinţe către un obiect remote [Ath02] – Implementare client

```
import java.rmi.*;
import java.net.*;
public class ClientRMIPisica {
  static public void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    String unde = "rmi://kermit.cs.pub.ro/ServerPisica";
    try {
      Remote robj = Naming.lookup(unde);
      ServerPisicaInterf spi = (ServerPisicaInterf) robj;
      PisicaInterf pi = (PisicaInterf) spi.referintaPisica();
      System.out.println("S-a obtinut referinta la " +
        pi.getClass());
      PisicaObisnuita po = (PisicaObisnuita) pi.obtinePisica();
      po.afiseaza();
    } catch (Exception e) {
      System.out.println("Eroare " + e);
      System.exit(0);
    }
  }
}
```

Alte aplicaţii RMI

- “transferul de comportament” de la client la server sau invers
- Exemplu: clientul obtine de la server codul pentru calculul unei formule [Ath02]

Exemplu – transfer de comportament de la server la client [Ath02]

```
public interface CalculInterf {
  int efectueazaCalcul(int x, int y);
}
```

```
import java.io.*;
public class CalculUnu implements Serializable, CalculInterf {
  public int efectueazaCalcul(int x, int y) {
    return x+y;
  }
}
```

```
import java.rmi.*;
public interface ServerCalculInterf extends java.rmi.Remote {
  public CalculInterf obtineCalcul() throws java.rmi.RemoteException;
}
```

Exemplu – transfer de comportament de la server la client [Ath02] – Implementare server

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ServerCalcul extends UnicastRemoteObject implements
  ServerCalculInterf {
  ServerCalcul () throws java.rmi.RemoteException {
  }

  public CalculInterf obtineCalcul()
    throws java.rmi.RemoteException  {
    CalculInterf c = new CalculUnu();
    return c;
  }
}
```

Exemplu – transfer de comportament de la server la client [Ath02] – Implementare client

```
import java.net.*;
import java.rmi.*;
public class ClientRMICalcul {
  public static void main(String a[]) {
    CalculInterf ci = null;
    System.setSecurityManager(new RMISecurityManager());
    try {
      Remote robj = Naming.lookup("ServerCalcul");
      System.out.println("S-a obtinut referinta la server " +
        robj.getClass());
      ServerCalculInterf sci = (ServerCalculInterf)robj;
      ci = (CalculInterf)sci.obtineCalcul();
      System.out.println("S-a obtinut referinta la " +
        ci.getClass();
    } catch (Exception e) {…}
    int z = ci.efectueazaCalcul(5, 3);
    System.out.println("Rezultat = " + z);
  }
}
```

Exemplu – transfer de comportament de la client la server [Ath02]

```
import java.rmi.*;
public interface ServerCalculInterf extends java.rmi.Remote {
  public void transmiteCalcul(CalculInterf c) throws
    java.rmi.RemoteException;
  public int obtineRezultat(int x, int y) throws
    java.rmi.RemoteException;
}
```

Exemplu – transfer de comportament de la client la server [Ath02] – Implementare server

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ServerCalcul extends UnicastRemoteObject implements
  ServerCalculInterf {
  CalculInterf c = null;
  ServerCalculImplem() throws java.rmi.RemoteException {
  }

  public void transmiteCalcul(CalculInterf c) throws
    java.rmi.RemoteException {
    this.c = c;
  }

  public int obtineRezultat() throws java.rmi.RemoteException  {
    return c.efectueazaCalcul(x, y);
  }
}
```

Continut

- Particularitati RMI
- Exemple si aplicatii
- Activarea obiectelor la distanta
- Colectarea memoriei disponibile
- Avantaje & dezavantaje RMI

Activarea obiectelor la distanta

- Modelul RMI “clasic”:
  - obiectul *remote* este instantiat la pornirea serverului si “traieste” o perioada nedefinita
  - in cazul caderii serverului obiectul *remote* nu poate fi recuperat
- Modelul cu obiecte activabile
  - obiectele *remote* sunt create doar atunci cand sunt necesare
  - se pot crea referinte persistente la obiecte

Obiecte activabile - termeni specifici

- Obiect activ: obiect remote instantiat si exportat pe alta masina Java
- Obiect pasiv: obiect remote care inca nu a fost instantiat sau exportat
- Referinta la distanta rezolvabila (faulting remote reference) - inclusa in stub-ul obiectului, contine:
  - Un identificator al obiectului – contine informatiile necesare pentru “gasirea” si activarea obiectului
  - O referinta la obiect – este null daca obiectul e pasiv

Entitatile implicate in activare

- Activator
  - Responsabil cu activarea obiectelor
  - Contine o tabela ce face legatura intre clasele obiectelor, URL-ul acestora si eventual alte date necesare la initializare
  - Administreaza masinile virtuale Java in care sunt incarcate obiectele activabile
- Grup de activare
  - Obiectele activate se pot porni in masini virtuale separate
  - Fiecare masina virtuala are un grup de activare – care realizeaza efectiv activarea
  - Daca programatorul nu specifica un grup pentru un nou obiect, acesta se va introduce intr-un grup “default”

Protocolul de activare

- Daca in momentul accesarii unui obiect remote, “faulting reference” este null, este apelat activatorul
- Folosind identificatorul de activare, activatorul determina clasa obiectului, locatia acesteia, datele de initializare si grupul de activare
- Daca grupul de activare exista deja, activatorul ii trimite acestuia o cerere de activare => incarcarea clasei, instantierea unui obiect
- Daca grupul nu exista, va fi creat intr-o noua masina virtuala Java

Clase specifice

- Pachetul `java.rmi.activation`
  - `Activatable` – clasa abstracta de baza pentru obiecte activabile
  - `ActivationGroup` – pentru crearea de noi instante de obiecte activabile
  - `ActivationGroupDesc` – contine informatii necesare pentru crearea / re-createa unui grup de activare
  - `ActivationGroupDesc.CommandEnvironment` – pentru specificarea de proprietati/optiuni
  - `ActivationGroupID` – identificare unica a unui grup
  - `MarshalledObject` – container pentru un obiect transmis ca parametru intr-un apel RMI

Exemplu – Implementare server activabil [Ath02]

```
import java.rmi.activation.*;
import java.rmi.*;

public class ServerPisicaImplemAct extends Activatable implements
  ServerPisicaInterf {
  PisicaObisnuita po = null;

  public ServerPisicaImplemAct (ActivationID id, MarshalledObject
    data) throws RemoteException {
    super(id, 0);
    po = new PisicaObisnuita("Mitzy");
  }
  public PisicaObisnuita obtinePisica() throws RemoteException {
    System.out.println("Se transmite " + po.getClass());
    return po;
  }
  public String numePisica() throws RemoteException {
    return po.cumOCheama();
  }
}
```

Exemplu – Inregistrarea serverului activabil [Ath02]

```
import java.rmi.activation.*;
import java.rmi.*;
import java.util.Properties;

public class Pornire {
  public static void main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());
    Properties prop = new Properties();

    // Creare descriere grup de activare
    ActivationGroupDesc.CommandEnvironment ace = null;
    ActivationGroupDesc grupEx = new ActivationGroupDesc(prop, ace);

    // Determina identificatorul grupului
    ActivationGroupID agi =
      ActivationGroup.getSystem().registerGroup(grupEx);

    // Creare grup de activare
    ActivationGroup.createGroup(agi, grupEx, 0);
```

Exemplu – Inregistrarea serverului activabil (2)

```
    String location = "file:./";
    MarshalledObject data = null;

    // Descriere obiect
    ActivationDesc desc = new ActivationDesc("ServerPisicaImplemAct",
      location, data);

    ServerPisicaImplem spi =
      (ServerPisicaInterf)Activatable.register(desc);
    System.out.println("S-a obtinut referinta la " + spi.getClass();

    //Inregistrare server
    Naming.rebind("ServerPisica", spi);
    System.out.println("S-a anuntat serverul");
    System.exit(0);
  }
}
```

Observatii asupra exemplului

- Din punctul de vedere al clientului nu se schimba nimic daca serverul e activabil
- Nu este obligatoriu ca obiectul activabil sa extinda clasa Activatable: se poate crea un obiect remote si apoi se utilizeaza `Activatable.exportObject()`
- pentru executia secventei de pornire trebuie specificate proprietati referitoare la securitate si la *codebase*:

```
java -D java.security=politica
-D java.rmi.server.codebase=file:./ Pornire
```

Fisierul `politica`:

```
grant {
  permission java.security.AllPermission;
};
```

Continut

- Particularitati RMI
- Exemple si aplicatii
- Activarea obiectelor la distanta
- Colectarea memoriei disponibile
- Avantaje & dezavantaje RMI

Colectarea memoriei disponibile in Java

- Initial: algoritm *mark & sweep*
- Ulterior (Java HotSpot VM): algoritm generational
  - Bazat pe observatia ca cele mai multe obiecte au o perioada de viata foarte scurta
  - Heap-ul e impartit in 3 sectiuni: obiecte permanente / obiecte vechi / obiecte noi
  - Colectarea in sectiunea obiectelor noi (minor collection) se face mult mai des decat colectarea “completa” (major collection)
- J2SE 1.4.1: algoritmi suplimentari, unii optimizati pentru sisteme multi-procesor

Colectarea memoriei disponibile pentru obiectele remote

- Problema: obiectele remote sunt instantiate pe server, dar sunt referite de catre clienti
- Cum se stie pe server ca nici un client nu mai are nevoie de un anumit obiect?
- Solutie: clientii “inchiriaza” obiecte pentru anumite intervale de timp
- Daca obiectul nu este accesat pana la expirarea timpului: poate fi colectat
- Intervalul implicit este de 10 minute

Colectarea memoriei disponibile – mod de implementare

- Pachetul `java.rmi.dgc` – clase pentru *distributed garbage collection*
- Pentru server: interfata DGC – metode `clean(), dirty()`
- Pentru client: thread dedicat care se ocupa de transmisia apelurilor `clean()` si `dirty()` catre server

Propunere de imbunatatire a colectarii memoriei disponibile [Sri05]

- Studiu empiric: s-a constatat ca ciclul de viata al obiectelor *remote* este diferit de cel al obiectelor “obisnuite” (obiectele *remote* au sanse mai mari de a “supravietui” in sectiunea obiectelor noi)
- S-a propus o noua schema pentru garbage collection: o generatie suplimentara pentru obiectele remote (Gen-R)
- Schema propusa a dat rezultate mai bune la simulari decat schemele clasice

Continut

- Particularitati RMI
- Exemple si aplicatii
- Activarea obiectelor la distanta
- Colectarea memoriei disponibile
- Avantaje & dezavantaje RMI

Avantaje RMI

- Orientat-obiect: se pot transmite obiecte ca parametri si ca rezultate
- Permite transmiterea de “comportari”
- Politici de securitate pentru verificarea codului
- Portabilitate
- Posibilitatea utilizarii JNI pentru a converti interfete scrise in alte limbaje la Java
- Colectare de spatiu disponibil

Dezavantaje RMI

- Nu se poate utiliza in medii neomogene
- Nu poate fi utilizat pentru calcul de inalta performanta
- Restrictiile de securitate pot duce la limitarea functionalitatii

Referinte

- [Ath02] Irina Athanasiu, “Java ca limbaj pentru programarea distribuita”, MatrixRom, 2002
- http://www.javacoffeebreak.com/articles/rmi_corba/ - comparatie intre Java RMI si CORBA
- [Eck02] Bruce Eckel, “Thinking in Java” 3rd Edition, 2002, capitolul 12 (se poate downloada de la www.bruceeckel.com)

Referinte suplimentare

- [Gab99] V. Gaburici, I. Athanasiu, “RMI – O abordare pragmatica”, PC Report, noiembrie 1999
- http://my.execpc.com/~gopalan/misc/compare.html - Comparatie intre RMI, CORBA si DCOM
- [Sri05] W. Srisa-an, M. Oey, S. Elbaum, “Garbage Collection in the presence of Remote Objects: An Empirical Study”, conferinta “Distributed Objects and Applications”, 2005
- http://www.javaworld.com/javaworld/jw-03-2003/jw-0307-j2segc.html - J2SE 1.4.1 boosts garbage collection