JAVA RMI

- Erlaubt den Aufruf von Javamethoden in der Umgebung einer anderen virtuellen Maschine
- Die andere virtuelle Maschine kann auf einem anderen Host laufen (Server)
- Ein Daemonprozess auf dem Server registriert alle Objekte, die Dienste anbieten und »lauscht« auf einem Port (default:1099)
- Der Client kann Methoden des Servers aufrufen, so als ob es eigene Methoden wären.
- Dabei können Parameter übergeben und Returnwerte zurückgegeben werden (es werden Kopien übergeben).
- Applets können RMI Aufrufe nur an den Host, von dem sie geladen worden sind, senden
- Eine Minimalapplikation benötigt folgende Teile
  - ein Remote Interface
  - ein Remote Object, das den Server darstellt
  - ein Java Client (Applet/Applikation)
  - ggf. html File, das das Applet lädt
- Werkzeuge, die benötigt werden
  - Javac
  - java
  - ggf. Browser
  - rmregistry (unix), start rmregistry(WIN) um den Server zu registrieren
  - rmic um Skeleton- und Stubclasses zu erzeugen

1. Remote Interface

```
public interface Hello extends java.rmi.Remote
{
    String syaHello() thows java.rmi.RemoteException;
}
```

- muß public sein, sonst kann das RemoteObject (auf dem Server ?) nicht geladen werden.
- muß von java.rmi.remote erben.
- Muß `thows java.rmi.RemoteException` enthalten
- Wird ein remote Objekt als Parameter oder Returnwert übegeben, muß es als RemoteInterface deklariert werden.

## 2. Implementation class

```
package RMI.Hello;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;

public class HelloImpl extends UnicastRemoteObject
                       implements Hello
{
  private String Name;

  public HelloImpl(String s) throws RemoteException
  {
    super();
    Name=s;
  }

  public String sayHello() throws RemoteException
  {
    return "Hello from RMI-Test";
  }

  public static void main(String args[])
  {
    System.setSecurityManager(new RMISecurityManager()
    {
      public void checkConnect(String host,int port){}
      public void checkAccept(String host, int port){}
    });

    try
    {
       HelloImpl obj=new HelloImpl("HelloServer");
       Naming.rebind("//beck2/HelloServer",obj);
       System.out.println("Helloserver bound in registry");
    }
    catch (Exception e)
    {
       System .out.println("HelloImpl err: "+e.getMessage());
       e.printStackTrace();
    }
  }
}
```

- muss das remote Interface implementieren
- muss einen Constructor definieren
- muss eine Implementation für die remote aufzurufenden Methoden enthalten
- muss einen speziellen Securitymanager implementieren
- es muss eine Instanz des Remoteobjektes erzeugt werden
- das remote objekt muss registriert werden
- indem das remote objekt von UnicastRemoteObject abgeleitet wird, beinhaltet es die notwendige Funktionalität zur Kommunikation. UnicastRemoteObject ist abgeleitet von RemoteServer und RemoteObject
- Wird ein remote object nicht von UnicastRemoteObject abgeleitet, so muss die Methode UnicastRmoteObject.exportObject aufgerufen werden

- Der Constructor ruft super() auf, somit wird der Costructor der Superclass ausgeführt. Dies bewirkt das Exportieren des remote objects, indem es auf einem anonymen Port auf ankommende Aufrufe wartet.
- Der Construktor muß java.rmi.RemoteException „thrown“

- Für jede romote aufzurufende Methode wird eine Methode implementiert.
- Muss ebenfalls java.rmi.RemoteException „thrown“
- Argumente können von jedem beliebigen einfachen javaTyp sein oder von Klassen, wenn sie das Interface serializable implementiert haben. Lokale Objekte werden dabei kopiert (außer static und transient fields), remote objekts (lokal bzgl. Server) werden als Referenzen übergeben.

Die main Methode

- Die main methode des Servers muß einen Securitymanager instazieren und installieren
- Es kann der vordefinierte RMISecurityManager oder ein eigener SecurityManager sein.
- Ohne den Securitymanager ist RMI-Arbeit nicht möglich
- Die main Methode muß eine Instanz von RemoteObject erzeugen, der Constructor exportiert dann das Objekt, dh, es wartet auf eingehende Aufrufe

Registrieren des Remote Objektes

- Das RMI System stellt einen Registrierungsmechanismus bereit, so daß remote objkets unter einer URL der Form //host/object adressierbar sind.
- Dazu muß ein Programm rmiregistry laufen
- über Naming.rebind wird ein remote object registriert
- In der Url wird kein Protokoll angegeben, es kann eine Portnummer angegeben werden, normalerweise wird 1099 verwendet
- bind oder unbind Operationen können nur von Programmen auf dem Host, auf dem rmiregistry läuft, vergenommen werden, so kann kein Client von außen Services hinzufügen oder entfernen
- rmiregistry muß immer, wenn sich eine Implementation geändert hat beendet und neu getsartet werden.

Applet mit RMI

- Das Applet erhält ein Objekt des Servers über Naming.lookup. Es bekommt dabei aber immer nur das Stub objekt geliefert.
- Bei Applets wird getCodeBase und GetHost verwendet, um die URL zusammenzubauen.
- Der Typ des Objektes ist das von RemoteInterface abgeleitete Interface
- Im Beisiel wird über obj.sayHello die remote Methode aufgerufen

Stubs und Skeletons

- rmic generiert die stub- und skeleton Dateien
- dabei ist ggf. -d . anzugeben, das Verzeichnis, in das die erzeugten Classfiles abgelegt werden
- client benötigt Stub, Server Skel

```
import java.util.*;

public interface RDir extends java.rmi.Remote
{
  public Vector getDir(String Dir) throws
java.rmi.RemoteException;
}
```

---

```
import java.io.*;

import java.util.*;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;

public class RDirImpl extends UnicastRemoteObject
                        implements RDir
{
  static int i=0;
  private String Name;

  public RDirImpl(String s) throws RemoteException
  {
    super();
    Name=s;
  }
  public static void main(String args[])
  {
    System.setSecurityManager(new RMISecurityManager()
     {
       public void checkConnect(String host,int port){}
       public void checkRead(FileDescriptor fd){}
       public void checkRead(String file){}
       public void checkAccept(String host, int port){}
     });
    try
    {
       RDirImpl obj=new RDirImpl("HelloServer");
       Naming.rebind("//"+args[0]+"/RDirServer",obj);
       System.out.println
                      ("RDirserver bound in registry");
    }
    catch (Exception e)
    {
       System .out.println
           ("HelloImpl err: "+e.getMessage());
       e.printStackTrace();
    }
  }
```

```
  public Vector getDir(String Dir)
                throws RemoteException
  {
    File f=new File(Dir);
    Vector v=new Vector();
    System.out.println("Req :"+Dir);
    System.out.println("File:"+f.toString());
    try
    {
       if (f.isDirectory())
        {
          String[] files=f.list();
          for(int i=0;i<files.length;i++)
          {
            System.out.println(files[i]);
            v.add(files[i]);
          }
        }

        else
        try
        {
          FileInputStream fis=new FileInputStream(f);
          byte[] data=new byte[f.length()];
          fis.read(data);
          System.out.write(data);
          v.add(new String(data));
        }
        catch (FileNotFoundException e)
        {System.out.println(e);}
    }
    catch (Exception e)
    {System.out.println(e);}

    return v;
  }


}
```

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

public class RDirCl
{
 Vector v=null;
 public RDirCl(String host,String Dir)
 {
  try
  {
    RDir obj =(Rdir)Naming.lookup
                ("//"+host+"/RDirServer");
    v=obj.getDir(Dir);

    System.out.println(v.toString());

  }
  catch(Exception e)
  {
        System.out.println("RDirException:\n"+e);
        e.printStackTrace();
  }
 }

//  public RFile getRDir(){return ()f;}

 public static void main(String args[])
 {
   RDirCl H=new RDirCl(args[0],args[1]);
   System.out.println("Message:"+H.getRDir().toString());
 }
}
```