KAPITEL 1

# RMI – Verteilte Programmierung unter JAVA

**von Anja Austermann**

Da die Sprache Java sehr stark an der Programmierung für das Internet orientiert ist, stellt sie mit RMI einen Mechanismus zur Verfügung, mit dem es dem Programmierer ermöglicht wird, verteilte Programme zu schreiben. Objekte, die sich auf unterschiedlichen Rechnern befinden, können mit Hilfe von RMI über Methodenaufrufe miteinander kommunizieren. JBuilder unterstützt die Generierung und das Compilieren von RMI-Klassen.

Das Prinzip von RMI ist denkbar einfach: Ein Client-Objekt sendet dabei eine Nachricht an den Server, die die aufzurufende Methode sowie die dafür benötigten Parameter enthält. Das Server-Objekt führt die entsprechende Methode aus und schickt das Resultat wieder an den Client zurück.

Zur Realisierung dieses Konzepts kapselt Java die Daten, die über das Netzwerk übermittelt werden sollen, auf dem Client-Rechner in sogenannten »Stubs«. Die Parameter müssen dafür zuerst in einem passenden Format zusammengefaßt werden. Zahlen werden beispielsweise so gespeichert, daß sie sowohl auf Windows- als auch auf Unix- und Solaris-Systemen korrekt gelesen werden können.

Komplizierter ist diese Aktion bei Objekten, beispielsweise bei Strings oder eigenen Klassen. Da Objektreferenzen im Grunde nichts anderes sind als Zeiger auf bestimmte lokale Speicherstellen, kann der Server damit natürlich nicht viel anfangen. Es muß also das komplette Objekt übermittelt werden, wozu Object Serialization, also der gleiche Mechanismus verwendet wird, der mit dem auch Objekte beziehungsweise Objektreferenzen auf einer Diskette oder Festplatte gespeichert werden. Um das zu ermöglichen, müssen Objekte, die als Parameter für RMI-Aufrufe dienen, das Interface *Serializable* aus dem Package *java.io* implementieren.

Diese für die Übermittlung notwendigen Schritte, das Kapseln der Daten in *Stubs* und die eigentliche Datenübermittlung, erledigt Java für Sie, so daß die Arbeit mit verteilten Objekten sich im Prinzip kaum von der mit lokalen Objekten unterscheidet.

Auf dem Server gibt es als Gegenstück zu den Stubs sogenannte »Skeletons«, die dafür zuständig sind, die gewünschte Methode aufzurufen, ihr die Parameter zu übergeben und letztendlich das Ergebnis an den Client zurückzuschicken. Auch hier erspart Java dem Programmierer die Hauptarbeit.

Am besten läßt sich das Prinzip von RMI grafisch verdeutlichen:

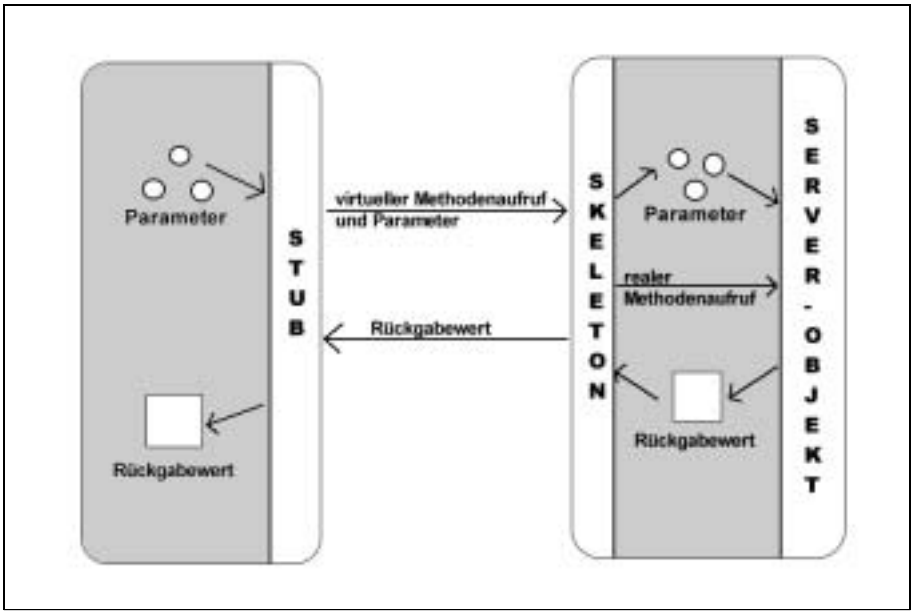


**Abb. 1.1:** Die Funktionsweise von RMI

Soviel zur Theorie. Am anschaulichsten läßt sich an einem kurzen Beispiel erklären, wie RMI grundsätzlich funktioniert und wie man RMI-fähige Objekte programmiert. Auf dem Server wird ein Puffer-Objekt erzeugt, in dem vom Client ein Wert abgelegt und auch wieder ausgelesen werden kann. Auch wenn das Beispiel sehr einfach ist, sind daran die wichtigsten Programmkonstrukte zu erkennen, die für RMI notwendig sind:

# 1.1 Programmierung der Server-Objekte

Für die Verwendung von RMI muß das Package *java.rmi* importiert werden. Auf dem Server, der das Remote-Objekt bereitstellt, wird zusätzlich *java.rmi.server* benötigt.

```
import java.rmi.*;
import java.rmi.server.*;
```

Für jedes Remote-Objekt, das auf einem Server abgelegt werden soll, muß ein Interface existieren, das von *java.Remote* abgeleitet ist. Alle Methoden, die später vom Objekt bereitgestellt werden sollen, werden hier deklariert. Die Implementierung des Interfaces kann später zwar zusätzliche Methoden enthalten, diese können aber nicht mit Hilfe von RMI aufgerufen werden.

Damit andere Objekte auf das Interface zugreifen können, ist dieses als *public* zu deklarieren. Jede Methode kann eine *RemoteException* auslösen, wenn beispielsweise die Verbindung abgebrochen wird oder andere Probleme bei der Kommunikation zwischen Client und Server auftreten. Daher steht diese Exception grundsätzlich bei allen Methoden von Interfaces, die *Remote* implementieren, in der *throws*-Deklaration

```
public interface Buffer extends Remote {
  void set(String Name) throws RemoteException;
  String get() throws RemoteException;
}
```

Auf dem Server muß neben dem Interface natürlich auch dessen Implementierung vorhanden sein. Um die unterschiedlichen Objekte problemlos unterscheiden zu können, empfiehlt es sich, für den Namen dieser Klasse die Endung »-Impl« zu verwenden. Die entsprechende Klasse wird von *UnicastRemoteObject* abgeleitet, das sich im Package *java.rmi.server* befindet. Wie man sieht, wird das Server-Objekt im Grunde genauso programmiert wie jede andere Java-Klasse, mit dem Unterschied, daß jede ihrer Methoden eine *RemoteException* auslösen kann.

```
class BufferImpl extends UnicastRemoteObject implements Buffer {
  public BufferImpl() throws RemoteException {
    Buf = new String("");
  }
```

```
  public void set (String Name) throws RemoteException {
    Buf = Name;
  }

  public String get() throws RemoteException {
    return Buf;
  }
  private String Buf;
}
```

Die Klasse *BufferServer* ist dafür zuständig, daß eine Instanz des *Remote*-Objekts erzeugt wird, und diese unter einem bestimmten Namen im Netzwerk zu erreichen ist. Auch hier entspricht die Endung »-Server« der gängigen Namenskonvention, um die Objekte besser auseinanderhalten zu können. Alternativ zu der Erstellung einer eigenen »Server«-Klasse ist es auch möglich, die *main*-Methode in die »Implementierungs«-Klasse einzubauen.

Jedes RMI-Programm sollte einen Security Manager besitzen, um die Klassen, die dynamisch aus dem Netzwerk geladen werden, zu überwachen, damit keine unangenehmen Überraschungen auftreten.

Weiterhin fällt in dieser Klasse der Aufruf *Naming.rebind("aBuffer", MyBuffer)* auf. Er dient dazu, ein Objekt auf dem Server zu registrieren. Der erste Parameter ist der Name, unter dem das Objekt im Netzwerk zu erreichen sein soll, der zweite eine Referenz auf das entsprechende Objekt.

```
public class BufferServer {
  public static void main (String[] args)  {
    System.out.println("...richte Bufferserver ein...");
    System.setSecurityManager (new RMISecurityManager());
    try {
      BufferImpl MyBuffer = new BufferImpl();
      Naming.rebind("aBuffer", MyBuffer);
      System.out.println("Server eingerichtet");
    } catch exception e) {
      System.out.println("Fehler beim Einrichten des Servers " +
e.getMessage());
    }
  }
}
```

### 1.1.1 RMI-Programme ausführen

Bevor Sie diese Klassen ausführen, sollten Sie auf dem Server einige Einstellungen vornehmen. Auf Ihrem Rechner muß das TCP/IP-Protokoll laufen. Unter Unix- oder Linux wird dies sowieso der Fall sein; wenn Sie Windows benutzen, kann es erforderlich sein, TCP/IP neu einzurichten, falls Sie es nicht bereits für den Zugang zum Internet oder zusammen mit einer Netzwerkkarte installiert haben.

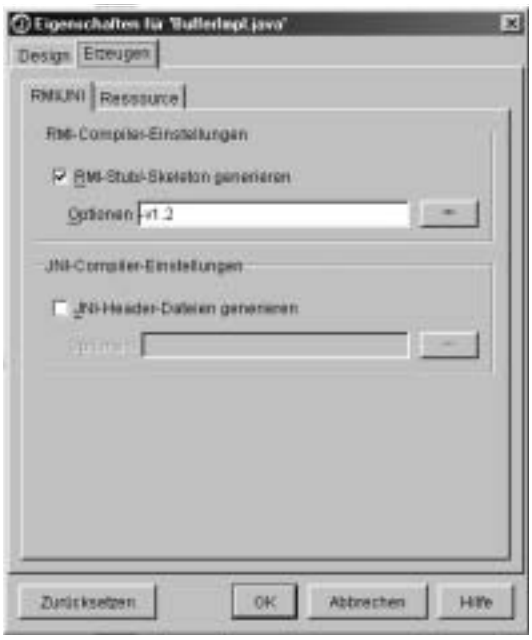


**Abb. 1.2:** Eigenschaften für BufferImpl.java

Außerdem muß die Server-Registry gestartet werden, die einen RMI-Nameserver bereitstellt. Dazu wählen Sie im Menü *Tools* den Menüpunkt *RMI-Registry* aus. Der Prozeß wird im Hintergrund gestartet. So lange die Server-Registry aktiv ist, ist ein Häkchen neben diesem Menüpunkt zu sehen. Sie müssen die Registry neu starten, falls Sie Änderungen an einem der Interfaces vorgenommen haben oder neue Interfaces hinzufügen möchten. Sollten Sie eine Fehlermeldung erhalten, überprüfen Sie, ob die Umgebungsvariable *Classpath* richtig gesetzt ist, damit die Registry in der Lage ist, die betreffenden Klassen zu finden.

Nach der Compilierung ihres Sourcecodes müssen Sie *rmic* ausführen. Mit diesem Befehl erstellt Java selbständig die *Stub*- und *Skeleton*-Klassen, die zur Übermittlung der Daten über ein Netzwerk benötigt werden. Im JBuilder führen Sie *rmic* aus, indem Sie auf die *-Impl*-Klasse, also diejenige Klasse, die von *UnicastRemoteObject* abgeleitet ist, rechtsklicken und den Menüpunkt *Eigenschaften* auswählen. Dort können Sie unter der Registerkarte *Erzeugen* Einstellungen für *rmic* vornehmen.

Wählen Sie dazu *RMI-Stub/-Skeleton generieren* aus. Klicken Sie dann auf den »...«-Button und wählen Sie *Java 2 only* aus oder geben Sie direkt die Option »-v1.2« ein. Damit sorgen Sie dafür, daß Stubs und Skeletons anhand der aktuellen Spezifikation generiert werden.

Wenn Sie jetzt noch einmal auf die *-Impl*-Klasse rechtsklicken und dann *Aktualisieren* auswählen, erzeugt Java zwei neue Klassen mit den Namen *BufferImpl_Skel.class* und *BufferImpl_Stub.class*.

Jetzt können Sie das Programm starten. Da dem Startbefehl insbesondere für die Sicherheitseinstellungen einige Parameter übergeben werden müssen, ist es notwendig, eine neue Startkonfiguration im Startmenü unter dem Menüpunkt *Konfigurationen* zu erstellen:

**Abb. 1.3:** Laufzeiteigenschaften der Klasse rmi.BufferServer

Im Eingabefeld *Name der Konfiguration:* können Sie einen beliebigen Namen eintragen, unter dem Sie die Ausführungsparameter für den Bufferserver ablegen möchten. Als Hauptklasse wählen Sie die *-Server*-Klasse aus. Weiterhin müssen Sie zwei VM-Parameter übergeben: zum einen die Codebase als URL, zum anderen die URL, unter der sich die Policy-Datei befindet. Wichtig ist dabei, daß alle Verzeichnis- und Dateinamen im 8.3-Format angegeben werden, also maximal acht Buchstaben als Dateinamen und drei Buchstaben als Erweiterung haben und ansonsten abgekürzt werden. Angenommen, das Projekt befindet sich in »d:\jbuilder\rmi«, dann müssen folgende Parameter übergeben werden:

```
-Djava.rmi.server.codebase=file:/d:\jbuild~1\rmi\classes/
-Djava.security.policy=file:/d:\jbuild~1\rmi\rmi.policy/
```

Wenn Sie das Programm nicht nur auf dem lokalen Rechner starten möchten und es möglich sein soll, nicht nur über die IP-Adresse, sondern auch über den Hostnamen darauf zuzugreifen, sollten Sie außerdem noch den Parameter

```
-Djava.rmi.server.hostname=mein_hostname
```

mit übergeben.

Damit ist die Programmierung des Servers fürs erste beendet. Um die Implementierung zu testen, benötigen wir noch ein Client-Objekt, das als nächstes geschrieben wird.

## 1.2 Programmierung eines Client-Objekts

Der *BufferClient* erzeugt ein Objekt vom Typ *Buffer*, also vom Interfacetyp und nicht etwa ein Objekt vom Typ *BufferImpl*, wie man vielleicht vermuten könnte. Dem lokalen Objekt wird das entsprechende Objekt auf dem Server zugewiesen. Dazu wird die Methode *lookup* der Klasse *Naming* benötigt, die zu einer URL und einem Objektnamen das passende Objekt vom Typ *Remote* zurückliefert, das dann nur noch in ein Objekt vom Typ *Buffer* konvertiert werden muß. Das ist durch explizites Typecasting möglich, da *Buffer* das Interface *Remote* implementiert und jedes *Buffer*-Objekt somit auch ein Objekt vom Typ *Remote* ist.

Da die *get*- und die *set*-Methode des Interfaces *Buffer* Remote Exceptions auslösen können, müssen diese vom Programm abgefangen werden. In diesem Beispiel geschieht das Exception-Handling durch die Ausgabe einer

Fehlermeldung, die den Typ der Exception ausgibt, in komplexeren Programmen werden Sie vermutlich für die möglichen unterschiedlichen Exceptions eigene Fehlerbehandlungsroutinen einführen müssen.

```
public class BufferClient {
  public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    String url = "rmi://127.0.0.1/";
    try {
      Buffer b1 = (Buffer)Naming.lookup(url + "aBuffer");
      b1.set("Hello World! - was sonst???");
      System.out.println(b1.get());
    } catch (Exception e) {
      System.out.println ("Fehler: " + e);
    }
  }
}
```

## 1.3 Sicherheitseinstellungen mit Security Policies

Im Sourcecode der Klassen *BufferServer* und *BufferClient* wurde ein SecurityManager festgelegt. Es handelt sich hierbei um einen SecurityManager vom Typ *RMISecurityManager*. Das Sicherheitskonzept von Java kennt viele verschiedene Security Manager. Neben *RMISecurityManagern* kommen auch beispielsweise solche für Applets vor, mit denen dann bestimmte Rechte vergeben oder entzogen werden können.

Der *RMISecurityManager* sorgt dafür, daß keine unberechtigten Zugriffe erfolgen, aber bis jetzt weiß er noch gar nicht, was er tun soll, weil ihm dazu noch ein Policy-File fehlt. Im Policy-File wird angegeben, auf welchen Ports Clients Verbindungen aufbauen dürfen und auf welche Dateien der Zugriff gestattet werden soll. Dabei wird zwischen lesendem, schreibendem, ausführendem und löschendem Zugriff unterschieden.

Im folgenden ein Beispiel für ein Policy-File für einen RMI-Server. In diesem Beispiel werden zuerst die Ports 1024 bis 65535 – also alle unprivilegierten Ports – freigegeben. Falls man bestimmte Zugriffsarten nicht benötigt, wie hier im Beispiel jegliche Arten von Dateizugriffen, werden diese in der Policy-Datei einfach nicht erwähnt. Sie sind dann automatisch verboten

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect,accept";
};
```

Sie erzeugen das Policy-File, indem Sie auf die Projektdatei rechtsklicken und *Dateien/Packages hinzufügen* auswählen. Im Stammverzeichnis des Projekts können Sie dann die leere Datei »rmi.policy« erzeugen, die oben gezeigten Zeilen einfügen und speichern.

### 1.3.1 3...2...1...start!

Auch für den Client wird eine neue Start-Konfiguration erzeugt, die genauso aussieht wie die für den Server, mit dem Unterschied, daß hier als Hauptklasse die *-Server*-Klasse ausgewählt wird.

Wenn Sie *rmiregistry* gestartet haben und jetzt auf den Pfeil neben dem Start-Icon klicken und zunächst die Server- und dann die Clientklasse starten, sehen Sie, wenn Sie die Codebase und die Security-Policy korrekt festgelegt haben, auf dem Bildschirm die Ausgabe

```
"Hello World! - was sonst???"
```

Zusammenfassend kann man festhalten:

Um RMI-Objekte zu programmieren, werden auf dem Server und auf dem Client bestimmte Klassen benötigt.

Auf dem Server sind das:

- *public Klasse*: Ein Interface, das die Methoden des betreffenden Objekts enthält und Tochterklasse von *Remote* ist.
- *KlasseImpl*: Eine Klasse, die von *UnicastRemoteObject* abgeleitet ist und das Interface *Klasse* implementiert.
- *public KlasseServer*: Eine Klasse, die eine statische Methode *main()* besitzt und dafür zuständig ist, eine Instanz von *KlasseImpl* zu erzeugen und auf dem Server zu registrieren.

Auf dem Client sind das:

- *public Klasse*: Das gleiche Interface wie auf dem Server.
- *public KlasseClient*: Eine Klasse, die in ihrer *main()*-Methode eine Instanz der Remote-Objekte auf dem Client-Rechner erzeugt und ihre Methoden aufruft.

Die Server-Registry muß mit Hilfe des Programms *rmiregistry* gestartet werden.

Das Programm *rmic* erzeugt die erforderlichen Stub- und Skeleton-Klassen.

# 1.4 Aktivierbare Objekte

Mit Java 1.1 mußten Objekte, die für einen Client zur Verfügung gestellt werden sollten, die ganze Zeit auf dem Server laufen, wie es im ersten Beispiel der Fall ist. Seit Java 2 ist es auch möglich, Server-Programme nur bei Bedarf zu starten. Zuständig dafür ist der RMI-Daemon *rmid*, der im Hintergrund auf RMI-Anfragen wartet und dann die passenden Klassen und Methoden aufruft.

Damit eine Server-Applikation über *rmid* aufgerufen werden kann, muß sie etwas anders aufgebaut sein als die Remote-Objekte, die Sie bisher kennengelernt haben.

An Stelle des Packages *java.rmi.server* wird hier *java.rmi.activation* importiert. Dort befinden sich die Klassen, die Sie benötigen, um aktivierbare Objekte zu programmieren.

Am Interface für das betreffende Remote-Objekt ändert sich nichts. Auch der Code für die Client-Applikation bleibt unverändert, da die automatische Aktivierung von Objekten einzig und allein Sache des Servers ist.

## 1.4.1 Eine »Implementations«-Klasse für aktivierbare Objekte

Die ersten Änderungen werden an der Implementations-Klasse vorgenommen, die jetzt von *Activatable* und nicht mehr von *UnicastRemoteObjekt* abgeleitet wird.

Bei dem Beispiel mit der *Buffer*-Klasse sieht die Klassendeklaration also folgendermaßen aus:

```
public class BufferImpl extends Activatable implements Buffer
```

Weiterhin muß die Klasse einen Konstruktor mit zwei Parametern besitzen. Der erste ist vom Typ *ActivationID* und dient dazu, das erzeugte Objekt für *rmid* identifizierbar zu machen, der zweite hat den Typ *MarshalledObject. MarshalledObjects* sind Objekte, deren Daten schon für den Transport über ein Netzwerk zusammengefaßt sind.

Der Konstruktor für unsere *Buffer*-Klasse hat demnach die folgende Form:

```
public BufferImpl(ActivationID id, MarshalledObject o) throws RemoteException
{
  super(id, 0);
}
```

## 1.4.2 Programmierung einer »Setup«-Klasse

Statt einer »Server«-Klasse wird für aktivierbare Objekte eine »Setup«-Klasse geschrieben. Sie hat die Aufgabe, Informationen über die aktivierbare »Implementations«-Klasse an *rmid* weiterzugeben und eine Instanz der Klasse und ihren Namen beim Server zu registrieren. Danach kann sich die »Setup«-Klasse selbst beenden. Hier liegt auch der Hauptunterschied zu der bisherigen »Server«-Klasse, die nach dem Start ihrer *main()*-Methode nicht terminiert, sondern permanent weiterläuft.

Neben *java.rmi* und *java.rmi.activation* muß zusätzlich die Klasse *java.util.Properties* importiert werden.

In der *main()*-Methode der Klasse *BufferSetup* wird der Activation-Descriptor für das *Remote*-Objekt erstellt. Er stellt *rmid* die wichtigsten Informationen über das betreffende Objekt zur Verfügung, die für die Aktivierung gebraucht werden. Daher wird er zur Registrierung des Objekts auf dem Server benötigt.

Wie auch beim vorherigen Beispiel benötigen Sie wieder einen *SecurityManager*, der überwacht, ob das *Remote*-Objekt sicherheitsrelevante Operationen durchführt.

```
System.setSecurityManager(new RMISecurityManager());
```

Als nächstes erstellen wir eine *ActivationGroup*. Dafür wird zuerst ein *Properties*-Objekt erzeugt, das Informationen über das Betriebssystem, die Java-Umgebung und Benutzerdaten mit Hilfe der Methode *getProperties()* aus der Klasse *System* zugewiesen bekommt. Dabei wird *clone()* verwendet, um ein eigenständiges Objekt zu erhalten und nicht nur eine Referenz auf die tatsächlichen Systemeinstellungen. Diese Informationen werden benutzt, um eine Instanz von *ActivationGroupDesc* zu erzeugen, deren zugehörige Gruppe beim *ActivationSystem* registriert wird. Die Registrierung liefert eine *ActivatonGroupID* zurück. Sie hat die Funktion, die betreffende Objektgruppe im Netzwerk eindeutig zu identifizieren. Nachdem die Gruppe registriert ist, kann sie mit *createGroup* tatsächlich erzeugt werden.

```
Properties props = (Properties) System.getProperties().clone();
ActivationGroupDesc.CommandEnvironment command = null;
ActivationGroupDesc activationgroup = new ActivationGroupDesc(properties,
                                                              command);
ActivationGroupID id = ActivationGroup.getSystem().registerGroup(
                                                   activationgroup);
ActivationGroup.createGroup(id, activationgroup, 0);
```

Im nächsten Schritt soll ein ActivationDescriptor ezeugt werden. Dazu muß zunächst der Pfad festgelegt werden, wo die zum Projekt gehörenden class-Dateien liegen.

Denken Sie immer daran, den Verzeichnisnamen mit »/« abzuschließen, sonst kann *rmid* die Dateien nicht finden.

```
String location = "file:/Verzeichnis mit den class-Dateien/";
```

Die Klasse *ActivationDescriptor* besitzt verschiedene Konstruktoren. In diesem Beispielprogramm wird folgender Konstruktor benutzt:

```
ActivationDesc(String className, String location, MarshalledObject data)
```

*ClassName* ist der Name der *-Impl*-Klasse. *location* wurde bereits erzeugt, so daß nur noch *data* fehlt. Dieser Parameter faßt Daten, die eventuell für die Initialisierung des *Remote*-Objekts benötigt werden, zusammen und stellt sie dem Server zur Verfügung. Da in diesem Fall keine besonderen Daten benötigt werden, wird *data* mit dem Wert *null* initialisiert. Da jetzt alle benötigten Parameter vorhanden sind, kann der *ActivationDescriptor* für unser Buffer-Objekt fertiggestellt werden.

```
MarshalledObject data = null;
ActivationDesc desc = new ActivationDesc("activation.BufferImpl",
                                         location, data);
```

Um eine neue Instanz des Buffer-Objekts zu erzeugen und beim Server zu registrieren, wird der eben erstellte *ActivationDescriptor* benötigt. Danach kann man dem neuen *Buffer*-Objekt genau so wie einem nicht-aktivierbaren Objekt mit Hilfe der Methode *rebind(String s, Object o)* aus der Klasse *Naming* einen Namen zuweisen, über den es im Netzwerk gefunden werden kann.

```
Buffer MyBuffer = (Buffer)Activatable.register(desc);
Naming.rebind("BufferImpl", MyBuffer);
```

So sieht die komplette Setup-Klasse für unseren *ActivatableBuffer* aus:

```
import java.rmi.*;
import java.rmi.activation.*;
import java.security.CodeSource;
import java.util.Properties;
import java.net.*;

public class BufferSetup {
  public static void main (String[] args)  {
    System.out.println("...richte Bufferserver ein...");
```

```
    try {
      System.setSecurityManager (new RMISecurityManager());
      Properties properties = ((Properties)System.getProperties().clone());
      ActivationGroupDesc.CommandEnvironment command = null;
      ActivationGroupDesc activationgroup = new ActivationGroupDesc(
                                                properties, command);
      ActivationGroupID id = ActivationGroup.getSystem().registerGroup(
                                                      activationgroup);
      ActivationGroup.createGroup(id, activationgroup, 0);
      String location = "file:/d:\\jbuild~1\\activation\\activation/";
      MarshalledObject data = null;
      ActivationDesc descriptor = new ActivationDesc("activation.BufferImpl",
                                                     location, data);
      Buffer MyBuffer = (Buffer)Activatable.register(descriptor);
      Naming.rebind("aBuffer", MyBuffer);
      System.out.println("Server eingerichtet");
    } catch (Exception e) {
      System.out.println("Fehler beim Einrichten des Servers " +
                         e.getMessage());
    }
  }
}
```

## 1.4.3 Sicherheit für aktivierbare Objekte

Die Policy-Datei für aktivierbare Objekte sieht etwas komplizierter aus als die für normale RMI-Objekte:

```
grant {
  permission java.net.SocketPermission  "*:1024-65535",
                                        "accept, connect, listen";
  permission java.util.PropertyPermission "*", "read,write";
  permission com.sun.rmi.rmid.ExecOptionPermission "*";
  permission java.lang.RuntimePermission "*";
};
```

Neben der bereits im letzten Beispiel beschriebenen *SocketPermission* muß jetzt zusätzlich eine *PropertyPermission* eingerichtet werden, die es erlaubt, lesend und schreibend auf die System-Properties zuzugreifen, sowie eine *ExecOptionPermission* und eine *RuntimePermission*, die das Ausführen von Programmen ermöglichen.

### 1.4.4 Starten

Genau wie im letzten Beispiel kann jetzt eine neue Startkonfiguration erzeugt werden, die die *Setup*-Klasse als Hauptklasse enthält.

Weiterhin soll es möglich sein, den RMI-Daemon *rmid* aufzurufen, der auf Anfragen reagiert und automatisch die richtigen RMI-Objekte aktiviert. Dazu richten Sie einen neuen Menüpunkt im Tools-Menü ein, indem Sie dort auf *Tools konfigurieren* klikken:

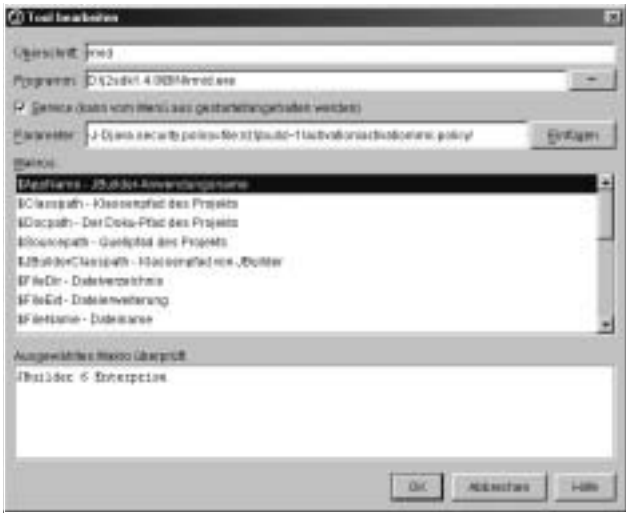

**Abb. 1.4:** rmid einrichten

Es wird zunächst ein Name für das Programm vergeben. Unter *Programm* geben Sie den Pfad zur Datei »rmid.exe« an. Außerdem wird die Security Policy als Parameter übergeben.

Jetzt kann *rmid* direkt aus dem Tools-Menü gestartet werden. Danach können Sie die Dateien *BufferSetup* und *BufferClient* aufrufen und erhalten dann dieselbe Ausgabe wie im ersten Beispielprogramm.