



# **MinCor.NET**

## **Java Connectivity**

Version: 2.4

Middsol GmbH  
mincor@middsol.de

## Contents

1	Introduction .....	3
2	Welcome to the World of Java .....	4
3	Java and CORBA.....	9
4	A Java Connectivity Sample .....	10
5	An EJB Connectivity Sample .....	16
6	Java-to-C# -Mapping.....	24
7	Java Connectivity Tools .....	29
8	Sources and References.....	35

## 1 Introduction

**Middsol's MinCor.NET** bridges the gap between Microsoft's .Net Compact Framework and CORBA. It is primarily a state of the art CORBA implementation for the .Net Compact Framework, enabling full interoperability between .Net and CORBA in both directions. CORBA mobile clients (as well as servers) may now be implemented within any programming language supported by the .Net Compact Framework.

Java comes with its own middleware, the *Remote Method Invocation (RMI)*. RMI is also the core middleware used by *Enterprise Java Beans (EJB)*. Because RMI may use internally the CORBA/IIOP stack as transport layer, it is possible to use CORBA to interoperate with every distributed Java application – not only with genuine CORBA applications written in Java.

This technology is based on the OMG **Java-to-IDL mapping** specification.

**Middsol's Java Connectivity** feature for their product **MinCor.NET** supports this mapping, i.e. .Net code may interoperate with Java RMI applications in a server mode as well as a client mode.

The first part of this documentation covers some general aspects of Java EJBs. The remaining part gives an introduction to MinCor.NET Java Connectivity. For a deeper understanding, please refer to the **tutorial** shipped with MinCor.NET. The starting page (**index.html** in the **MinCor.NET root folder**) provides a guide through all parts of the documentation.

In case you intend to use user defined Java types located in separate assemblies, i.e. assemblies that do not initialize MinCor.NET, you need to load these assemblies explicitly by adding the parameter **-AddAssembly=<assembly list>**. Further configuration details are given in the document MinCor.NET.pdf.

## 2 Welcome to the World of Java

### A View on Java

Coming out of a long tradition of C++ like programming languages, **Suns Java** is a mainstream object-oriented programming language. In the beginning intended for embedded programming (named "Oak" at that time), it started its career in the first half of the last decade primarily on the client side as the basis for the well known "Applets" downloaded in the context of a web page.

Learning from its predecessors (mainly C++ and Smalltalk), Java is a "purely object-oriented" language (i.e. everything is an object). It is strongly typed, with an efficient memory management system ("garbage collection") and it runs on top of a **Java Virtual Machine (JVM)**. The specification of the JVM is published, so that Java was ported to a great variety of platforms. From the point of view of its basic structure, Java and C# are pretty similar.

With its second version ("Java 2") Java was leveraged by some important enhancements, clearly targeting the server side:

- **Java Database Connectivity (JDBC)** for generic access to SQL databases
- **Remote Method Invocation (RMI)** for interprocess communication between several JVMs
- **Java Beans**, a simple component model which enhances Java classes with an inspection mechanism (mainly targeting developers of visual modelling tools);
- **Java Servlet API** as a Java based CGI replacement, enabling web servers to call Java classes hosted within a Servlet Engine without spawning a new process for each call.

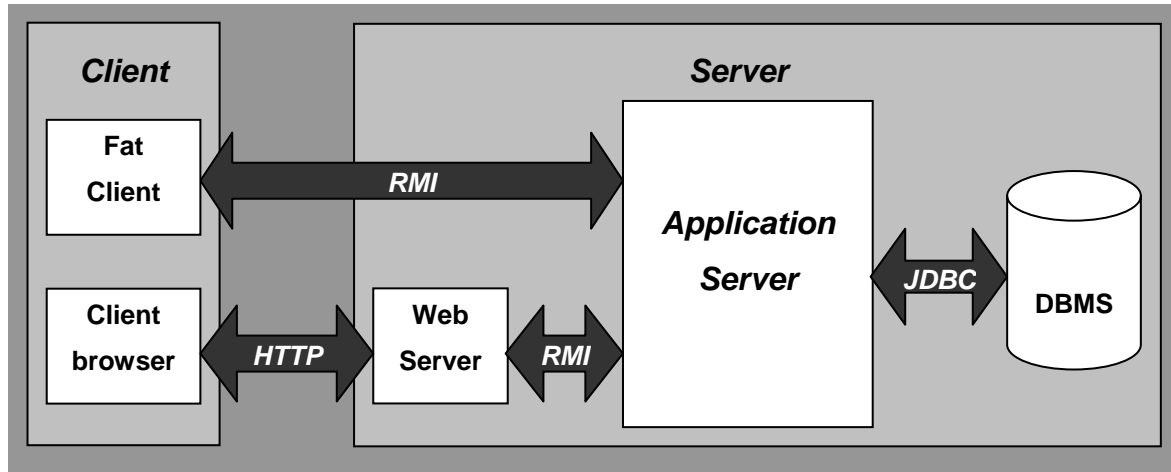
### The J2EE Architecture

The **Java 2 Enterprise Edition (J2EE)** is a broad collection of "enterprise level" APIs. Most of it is freely available from Sun. Third-party supplied components are usually attached via *Service Provider Interfaces (SPI)*, using generic configuration mechanisms. Within J2EE, there are standard APIs for cryptographic algorithms, for transaction processing, for access to

# Java Connectivity

naming and directory services, for accessing *Message Oriented Middleware (MOM)* and messaging in general.

All these APIs are integrated within an embracing architecture, which is commonly associated with the term "**Application Server**" (abbr.: "AppServer"). An application server will usually be embedded within a 3 or 4-Layer Architecture:



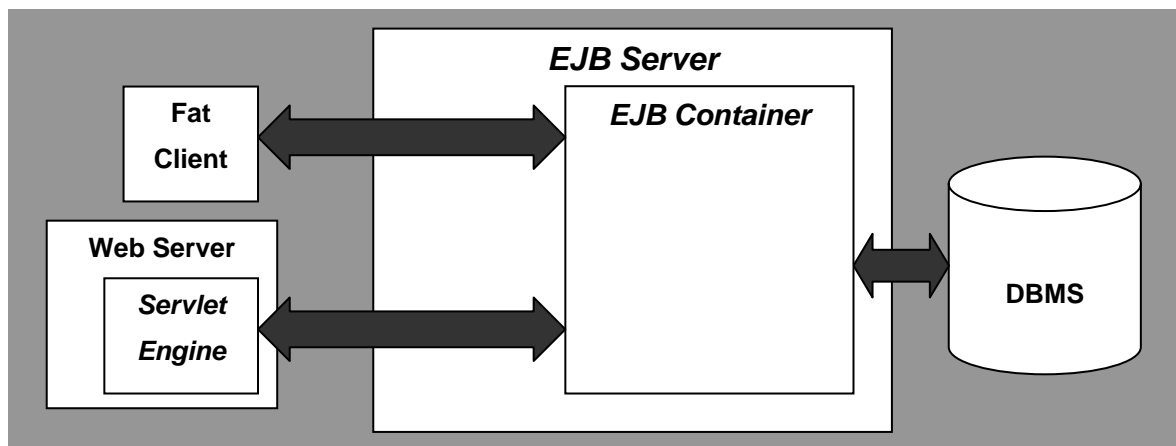
The very core of the AppServer sitting in the center of this architecture is the **EJB Server**, hosting one or more **EJB Container**. Both together (usually, EJB Server and Container are provided by the same vendor, so there is no clear demarcation between both) provide a rich enterprise-oriented infrastructure for – conceptionally fine grained – **Enterprise Java Bean** components (to be further explained within the next section):

- EJBs can be retrieved using a generic naming service implementing the **Java Naming and Directory Interface (JNDI)** API.
- EJBs can communicate via RMI, hence an AppServer may be transparently distributed over several JVMs and is therefore scalable.
- EJBs may work concurrently without being explicitly threadsafe, since the EJB Container takes care of all concurrency issues.
- EJBs may operate transactionally using a generic transaction service implementing the **Java Transaction API (JTA)**. JTA is designed according to the *X/Open XA* standard interface. It supports therefore a standard 2-Phase-Commit protocol and is thus compliant with the *OMG/CORBA Object Transaction Service*. Usually, an EJB Server will provide its own implementation of a Java Transaction Service (JTS), but it may also support any other stand-alone Transaction Monitor.

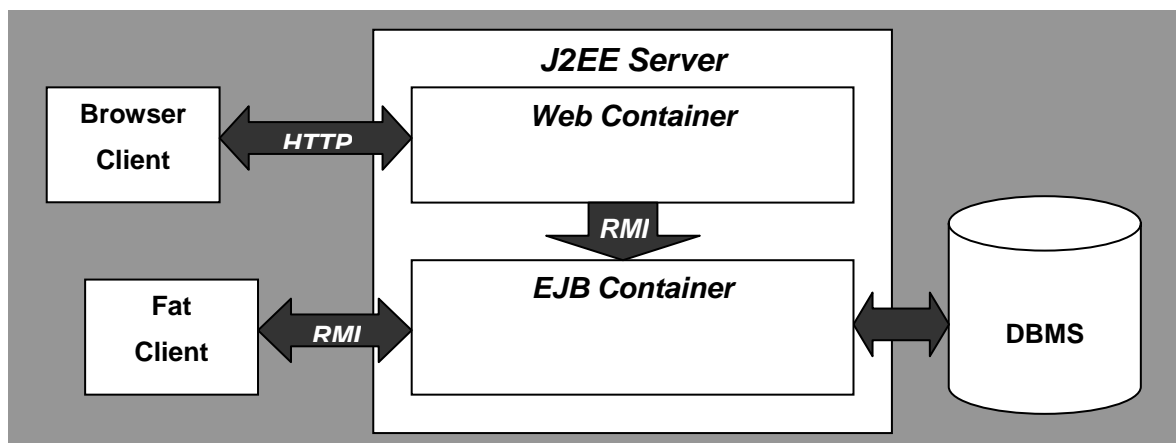
# Java Connectivity

- EJBs are configurable using standardized XML **Deployment Descriptors**, which give, among other things, access to a declarative handling of transactional bean behavior, as well as a declarative handling of persistency and security issues.

The EJB 1.0 standard (the current version is 2.0) defined a pure **EJB Architecture**. The EJB Server did not host any other components than EJBs. So EJB was a mere middle-tier technology for providing business logic:



Since EJB 1.1, **Java Server Pages (JSP)** allow spreading little Java code snippets across a HTML page. The code snippets will be dynamically compiled into standard Servlets hosted by the web server's Servlet Engine. The **J2EE Architecture** allows integration of all components of a web application (EJBs as well as Servlets/JSPs) within one hosting environment:



A short summary of the whole architecture:

- **Business Logic:** the EJB Container contains the EJBs, packaged within *Java Archive (JAR)* files.
- **Presentation Logic:** the Web Container contains web components, i.e. Servlets and JSPs, packaged within *Web Component Archive (WAR)* files.
- **Data Logic:** persistency will be provided by special beans within the EJB Container.
- **Web Application:** The whole web application, i.e. all the EJBs within its JARs, together with all the web components within its WARs, may be packaged within a single *Enterprise Archive (EAR)* file to be deployed to a J2EE Server.

## ***Enterprise Java Beans (EJB)***

Behind J2EE, there is a fine grained component model, different from the usually more coarse grained CORBA components. Though a CORBA component usually embraces a big chunk of business logic (most of the time a whole application on its own), Enterprise Java Beans prefer smaller components, e.g. a single account in a banking application may be represented by a dedicated account EJB, as well the process of generating a consolidated balance over a couple of accounts by a balance EJB.

This concept leads to the following distinction:

- **Entity Beans**, representing fine grained persistent entities. Depending on the preferred architectural pattern, they may contain also the business logic. There are two different ways to approach persistency with Entity Beans:
  - **Bean Managed Persistence (BMP)**, where the Bean class itself is responsible for managing its persistency; usually, a Bean will use JDBC to access the database.
  - **Container Managed Persistence (CMP)**, where the Bean is made persistent using services of the hosting EJB Container (configured within the Deployment Descriptor).
- **Session Beans**, representing a (non persistent) chunk of business process functionality. Session Beans come in two different flavours:
  - **Stateless Session Beans** offering atomar services to the client, and
  - **Stateful Session Beans** realizing a dialogue with the client.

# Java Connectivity

---

- **Message-Driven Beans**, providing entry points to an asynchronous messaging service encapsulated within the *Java Messaging Service (JMS)*.

A common problem with fine grained component models is the management of big amounts of similar objects, e.g. some thousand accounts within a bank application. Usually, one uses the somehow "platonistic" approach of an explicit SELECT statement on a database table to retrieve the intended object, but this is not possible if the database access is encapsulated within a fine grained component – as e.g. within an "AccountBean".

EJB resolves this issue by using the "Home" pattern:

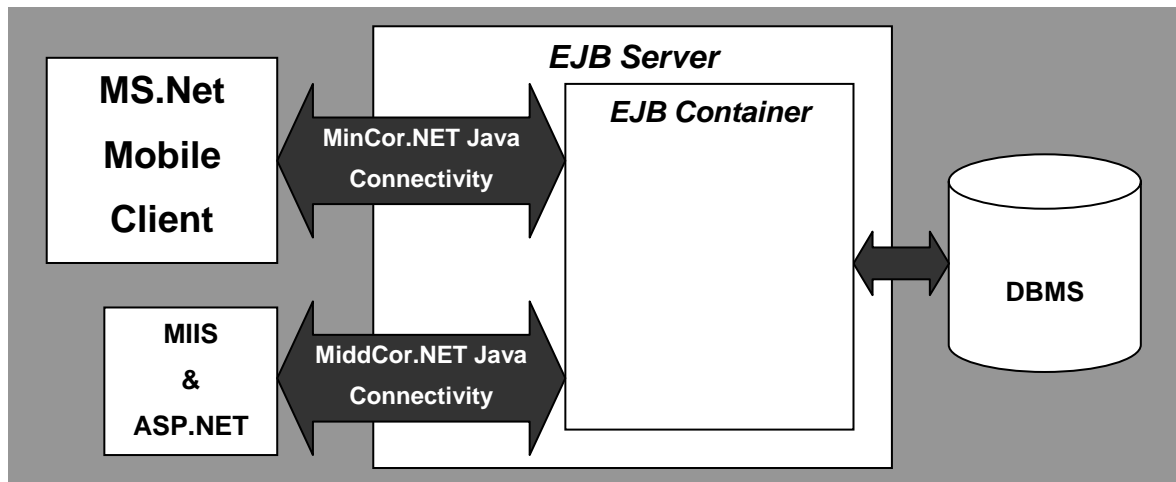
- Each bean (except Message-Driven Beans, which are anonymous endpoints for message queues) has a **Home Interface**. This Home Interface gives access to a couple of lifecycle management methods.
- Each bean has a **Remote Interface** representing its specific interface to the client, once an instance is retrieved using the Home Interface.
- Each bean has a **Bean Class** implementing the Remote Interface business functionality and the Home Interfaces management methods.
- Each Entity Bean has a **Primary Key Class** representing its persistent identity.

Note: The actual EJB will be generated from these classes (provided by the developer) within the – vendor specific – deployment process. Therefore the Bean Class does *not* "syntactically" implement the Remote Interface (using Javas "implements" clause). There is only a rather weak syntactical connection between the parts of the bean.

During deployment, each beans Home Interface will be registered with its name within a naming service. It can be retrieved by any (Java) client using the **Java Naming and Directory Interface (JNDI)**. The Home Interface serves as a factory for the beans instances. Once the client has access to the beans Home Interface, it may create a new instance or (for Entity Beans) retrieve an existing instance.

## **MinCor.NET and EJB**

Using MinCor.NET, the standard EJB architecture may be accessed by Microsoft's .Net framework allowing the implementation of presentation logic native to Windows, i.e. tightly integrated into the most widespread client desktop platform today.



## **3 Java and CORBA**

Though there are striking similarities between CORBA and the J2EE architecture, both are commonly viewed as competitors on the component architecture market. Of course, this point of view seems obvious, due to the fact, that J2EE/EJB is Sun's proprietary architecture, intended to be "*Pure Java*", realizing portability by default on top of the portability of the JVM.

### **RMI over IIOP**

In the beginning, RMI has used only a proprietary communication protocol, the *Java Remote Method Protocol (JRMP)*. Therefore, CORBA and RMI were *not* interoperable. A Java programmer had to decide to go for one or the other in order to realize a distributed Java application. RMI is best suited for a homogeneous environment, whereas CORBA can manage platforms without even an object model. In order to integrate both architectures, Sun layed down a specification for **RMI over IIOP**.

Whereas it is easy to implement a CORBA client embedded within a RMI infrastructure (e.g. to access legacy applications out of an EJB application), the other direction – accessing an EJB Server from a non-Java application – will gain more interest, especially with increasing

investments into EJB development. Due to this fact, Sun made RMI over IIOP with EJB 2.0 a mandatory part for every EJB vendor. IIOP is now the default communication protocol for RMI and EJB.

## ***JNDI and the CORBA Naming Service***

Another hindrance to smooth interaction between CORBA and J2EE/EJB may be seen in the different infrastructure services. Although both architectures define equivalent basic services, especially for naming, transaction handling, and messaging, the likewise access of interfaces may differ, sometimes substantially. This would be especially disadvantageous with respect to the naming service which is absolutely essential for realizing locational transparency within a component architecture.

EJBs use the **Java Naming and Directory Interface (JNDI)**, which provides access to naming and directory services using a generic client API. The API is transparently routed to a specific *Service Provider* encapsulating the selected service. Currently, there are JNDI service provider available for *LDAP*, *NIS*, *Novell NDS*, *SLP* (Service Location Protocol), the *RMI Registry*, the file system, and for *CORBA Cos Naming*.

It is standard for every J2EE/EJB server to offer a CORBA Naming Service on a specified TCP/IP port (for details, please refer to your AppServer vendors documentation). Therefore, every CORBA client application may use the AppServers CORBA Naming Service to retrieve a reference of any registered beans Home Interface, which may be used like any other CORBA object.

## **4 A Java Connectivity Sample**

Let us start with a small "hello world" example providing a Java RMI server which will be called from a .Net client written in C# using MinCor.NET.

(All paths are relative to an arbitrary project root.)

### ***Step 1. Writing the Java RMI Server***

Every Java RMI server needs to provide a special Remote interface, derived from the standard Java interface `java.rmi.Remote`. Every method remotely callable has possibly to throw a `java.rmi.RemoteException`.

# Java Connectivity

---

The Remote interface (Hello\Greetings.java):

```
package Hello;

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

public interface Greetings extends Remote {

    String hello( String name) throws RemoteException;

}
```

Next, we have to implement this remote interface. Therefore, we create a class GreetingsImpl which implements our remote interface.

The implementation (Hello \GreetingsImpl.java) is

```
package Hello;

import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;
import java.util.*;

public class GreetingsImpl extends PortableRemoteObject implements Greetings{

    public GreetingsImpl() throws RemoteException {
        super();
    }

    public String hello( String name ) throws RemoteException {
        System.out.println("Method executed: Greetings.hello(" + name + ")");
        return "Greetings to " + name;
    }

}
```

Finally, we need a server process to host our RMI server object.

This could have been done with a main() method inside the implementation class, but for the sake of a cleaner separation we do this in a separate class, placed in the default package (i.e. located on the project root).

Within the main thread, we will

- instantiate the RMI server object (Hello.GreetingsImpl);

# Java Connectivity

---

- publish its reference via JNDI (placed within the package javax.naming)

Since the server process makes use of the external CORBA Naming Service, Java's distributed garbage collector is not able to remove the newly created object (due to the fact that there is an external reference). Therefore the server process' main thread will not finish, i.e. there is no need to start an event loop.

The Java main server (Server.java) looks as follows:

```
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.rmi.PortableRemoteObject;

import Hello.Greetings;
import Hello.GreetingsImpl;

public class Server {

    public static void main(String[] args) {
        Server oServer = new Server();

        oServer.runServer();
    }

    public void runServer()
    {
        try {
            GreetingsImpl oGreetingsImpl = new GreetingsImpl();

            // publish the reference with the naming service:
            Context initialNamingContext = new InitialContext();
            initialNamingContext.rebind("HelloJavaServer", oGreetingsImpl);

            System.out.println("Hello-Server ready...");

        } catch (Exception e) {
            System.out.println("Trouble: " + e); e.printStackTrace();
        }
    }
}
```

Now we are in a position to compile the whole Java project (provided the local PATH environment variable includes the path to the J2SDKs bin directory).

To do so, we first create a new sub-directory to store our class files. Next, we call the Java compiler for our main server. Because the compiler works incrementally, all referenced but not

yet compiled classes will be compiled as well. We then have to call Javas RMI compiler to generated RMI stubs and skeletons from the freshly created class files.

In a final step, we create a new JAR file which serves as input for the MinCor.NET IDL generator:

```
> mkdir classes
> javac -classpath . -d classes Server.java
> rmic -iiop -classpath .\classes -d classes Hello.GreetingsImpl
> cd classes
> jar cvf ..\Server.jar .
> cd ..
```

## ***Step 2. Generate the IDL***

Having created all Java classes we are now ready to move to CORBA by generating standard IDL code from the Java JAR file using the MinCor.NET IDL generator:

```
> mkdir ..\IDL
> MCjava2idl -cp Server.jar -v -p -d ..\IDL "Hello.Greetings"
```

## ***Step 3. Generate .Net Code form the IDL File***

For the .Net client we introduce a separate project root, "DotNetClient", located parallel to the Java project root and called the MinCor.NET IDL compiler:

```
> mkdir ..\IDL
> MCidl2cs -all -d ..\DotNetClient\Generated ..\IDL\Hello.idl
```

## ***Step 4. Write the .Net Client***

We now create a mobile application and add the following files and references:

- a reference to '**MinCorCF.DLL**', the MinCor.NET ORB runtime located in the MinCor.NET bin directory

# Java Connectivity

---

- a reference to '**JavaBuildInTypesCF.DLL**', the MinCor.NET Java Connectivity runtime located in the MinCor.NET bin directory
- the generated file '**Generated\Hello.cs**'.

Then we have to write the .Net client itself:

At first, we will initialize our ORB with the "corbaloc" address of an existing CORBA Naming Service (preferably the one where our Java RMI server is registered). We assume in this context that the Naming Service runs on a remote server whose address is passed explicitly to the client, listening on port 1050 (it is possible to use another one as well, but take care that changes need to be made consistently throughout the whole example). Then we have to retrieve this service and get a server object reference for it. Finally, we will call our server object.

The .Net clients implementation is given below:

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;

namespace DotNetClient
{
    public class ClientImpl
    {
        // the substring 'localhost' s just a pattern for the replacement with the server address...
        private static string[] m_strORBInit =
            {"-ORBInitRef NameService=corbaloc:iiop:1.2@localhost:1050/NameService"};

        public static void runDemo( FrmClt a_oFrmClt, string a_strIpAddr )
        {
            Hello.Greetings oGreetings;
            Midsol.CORBA.ORB oOrb = null;

            // ...done here:
            m_strORBInit[0] = m_strORBInit[0].Replace("localhost", a_strIpAddr );

            try
            {
                // initialize the Orb:
                oOrb = Midsol.CORBA._ORB.init( m_strORBInit, null );

                // retrieve the Naming Service from the ORB:
                Midsol.CosNaming.NamingContextExt oNsCtx =
                    Midsol.CosNaming.NamingContextExtHelper.narrow(
                        oOrb.resolve_initial_references("NameService") );
            }
        }
    }
}
```

```
try
{
    // retrieve the server object reference from the Naming Service:
    oGreetings = Hello.GreetingsHelper.narrow( oNsCtx.resolve_str("HelloJavaServer" ) );
}
catch(Middsol.CosNaming.NamingContextPackage.NotFound ex)
{
    a_oFrmClt.writeLog( "NS Manager Error:" + ex.why);
    throw new System.Exception();
}

// call the server object:
string strRet = oGreetings.hello( "Middsol" );
a_oFrmClt.writeLog(strRet);

}
catch( Middsol.CORBA.SystemException exSys)
{
    a_oFrmClt.writeLog("Catch Exception from Server (Middsol.CORBA.SystemException).");
    a_oFrmClt.writeLog("ID:" + exSys.ID);
}
finally
{
    // clean up:
    if( oOrb != null)
        oOrb.destroy();
}
}
}
```

## ***Step 5. Run the Example***

Eventually, we may run the whole example:

Start the Sun Naming Service (delivered with the J2SDK 1.4.2) on port 1050 (take care of the fact that the options are case sensitive):

```
> start orbd -ORBInitialPort 1050
```

Start the Java RMI server by using JNDI CosNaming, working with a naming service running on 'localhost' at port '1050':

```
> start java -cp .;Server.jar -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory  
-Djava.naming.provider.url=iiop://localhost:1050 Server
```

At last, copy the .Net mobile client to the target device and start the client.

After providing the correct server address the screen should look as follows:



## 5 An EJB Connectivity Sample

Our next task is to build a small EJB "hello world" example, running in a J2EE Application Server, with a .Net mobile client written in C# using MinCor.NET. Even though an EJB example should not be much more complicated than the "standard" Java RMI example from the last chapter, we have to face the fact that the concrete EJB deployment process is dependent on the AppServer's vendor.

Therefore, this part of the example is not really vendor independent. We will concentrate here on the *Sun ONE Application Server* delivered together with the J2EE SDK 1.4. Refer to your AppServers vendor documentation for adaptation to other J2EE platforms.

## **Step 1. Write a Server EJB**

To create an EJB for deployment within an arbitrary J2EE/EJB server, we have to create a couple of Java files, namely the Home Interface, the Remote Interface, the Bean Class, and an appropriate XML Deployment Descriptor.

All the Java classes and interfaces have to be derived from the appropriate standard parents, located within the package `javax.ejb`. Additionally, we have to throw not only the standard `java.rmi.RemoteException`, but also some other EJB specific exceptions. Beside this, the Java code resembles pretty much our Java RMI example.

We start with the Home Interface, located in our "Hello" package (`Hello\Greetings_Home.java`):

```
package Hello;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface Greetings_Home extends EJBHome {

    public Greetings_Remote create () throws RemoteException, CreateException;

}
```

For our example we implement a simple stateless Session Bean. Accordingly, our Home Interface has to define a `create()` method.

The Remote Interface looks pretty much the same as the RMI Remote interface from the last chapter except for the different parent class (`Hello\Greetings_Remote.java`):

```
package Hello;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Greetings_Remote extends EJBObject {

    public String hello ( String name) throws java.rmi.RemoteException;

}
```

Finally, we need an appropriate implementation for our interfaces.

# Java Connectivity

---

Even though our Home Interface *publishes* only a simple create() method, we have to provide the full spectrum of Session Beans lifecycle management methods due to the bean contract specification. So we provide some almost empty implementations, just writing a short trace message to System.out.

Note that there is no need for these lifecycle methods to declare throwing any specific exceptions, since these methods are only delegates, used locally by code generation during deployment. For the same reason, the bean implementation does *not* syntactically implement its Remote Interface.

Below is the bean implementation (Hello\Greetings\_Bean.java) :

```
package Hello;

import java.rmi.RemoteException;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;

public class Greetings_Bean implements SessionBean {

    public Greetings_Bean () {
        super ();
    }

    // Session Bean contract methods
    private    SessionContext    m_ctx = null;
    public void setSessionContext (SessionContext ctx) {m_ctx = ctx;}

    public void ejbCreate() throws RemoteException, CreateException
        { System.out.println ("ejbCreate  () on " + this); }
    public void ejbRemove ()      { System.out.println ("ejbRemove  () on " + this); }
    public void ejbActivate()     { System.out.println ("ejbActivate () on " + this); }
    public void ejbPassivate ()   { System.out.println ("ejbPassivate() on " + this); }

    // Remote Interface implementation
    public String hello ( String name) throws java.rmi.RemoteException {
        return "Greeting from J2EE Server to " + name;
    };
}
```

# Java Connectivity

---

The last file we need for our EJB is an appropriate Deployment Descriptor describing the bean structure, transactional behavior (here: bean managed), and its unique EJB name. The optional display name is only for use in management tools (ejb-jar.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <enterprise-beans>
    <session>
      <display-name>Hello</display-name>
      <ejb-name>Hello</ejb-name>
      <home>Hello.Greetings_Home</home>
      <remote>Hello.Greetings_Remote</remote>
      <ejb-class>Hello.Greetings_Bean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

The procedure discussed so far is common to all AppServers.

Now we have to take care of the vendor specific deployment procedure, part of which is, for instance, providing the actual name for the naming service. In case of the *Sun ONE AppServer* this is done by a second deployment descriptor (sun-ejb-jar.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Sun ONE Application Server 8.0
EJB 2.1//EN" "http://www.sun.com/software/sunone/appserver/dtds/sun-ejb-jar_2_1-0.dtd">

<sun-ejb-jar>
  <enterprise-beans>
    <unique-id>1</unique-id>
    <ejb>
      <ejb-name>Hello</ejb-name>
      <jndi-name>ejb/Hello</jndi-name>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

Finally, we are ready to compile and deploy our bean.

# Java Connectivity

---

Sun prefers for this task the Ant tool, which is very popular within the Java community. It is a sophisticated tool using XML files as its input. Here is an example (supposing the default path for the *Sun ONE AppServer* was accepted during installation). It is documented here for the mere sake of completeness, so we will not comment on it any further (build.xml):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE project [ <!ENTITY include SYSTEM "c:/Sun/AppServer/samples/common.xml" > ]>

<project name="serialization" default="core" basedir=".">

  <property name="appname" value="Hello"/>
  <property name="sample.home" value="c:/Sun/AppServer/samples"/>

  <property name="app.pkg" value="Hello"/>
  <property name="jar.pkg" value="Hello"/>

  <property name="javadoc.pkgnames" value="Hello.*" />

  <property name="jarDD" value="ejb-jar.xml,sun-ejb-jar.xml"/>

  &include;

  <target name="clean" depends="clean_common"/>
  <target name="init" depends="init_common">
    <condition property="deploy.file" value="${assemble.ear}/${ear}">
      <available file="${assemble.ear}/${ear}"/>
    </condition>
    <property name="deploy.file" value="../${ear}"/>

    <condition property="verify.file" value="${assemble.ear}/${ear}">
      <available file="${assemble.ear}/${ear}"/>
    </condition>
    <property name="verify.file" value="../${ear}"/>
  </target>
  <target name="copy_ear" depends="init">
    <delete file="../${ear}"/>
    <copy file="${assemble.ear}/${ear}" todir=".." />
  </target>
  <target name="compile" depends="compile_common" />
  <target name="javadocs" depends="javadocs_common" />
  <target name="deploy" depends="init, deploy_common"/>
  <target name="undeploy" depends="undeploy_common"/>
  <target name="verify" depends="init, verify_common"/>

  <target name="jar" depends="init,create_ebjjar_common"/>
  <target name="ear" depends="init,jar,create_ear_common"/>
  <target name="core" depends="compile,jar,ear" />
  <target name="all" depends="core,javadocs,verify,deploy"/>

</project>
```

The make process may be started by calling Suns "asant" on the project root (where the build.xml is located).

## **Step 2. Generate the IDL**

As before, we move to CORBA and generate the IDL code from the Java JAR file using the MinCor.NET IDL generator:

```
> mkdir ..\IDL
> set classpath=%classpath%;C:\Sun\AppServer\lib\j2ee.jar
> MCjava2idl -cp %Classpath%;.\ejb\assemble\jar\helloEjb.jar -v -p -d ..\IDL Hello.Greetings_Home
```

## **Step 3. Generate .Net Code form the IDL File**

For the .Net client, we have a project root of its own. Assuming it's a directory "DotNetClient" located parallel to the Java project root, we may now call the MinCor.NET IDL compiler:

```
> mkdir ..\IDL
> MCidl2cs -d ..\DotNetClient\Generated ..\IDL\Hello.idl
```

## **Step 4. Write the .Net Client**

For the .Net client we again create a mobile application adding the following files and references:

- a reference to '**MinCorCF.DLL**', the MinCor.NET ORB runtime located in the MinCor.NET bin directory
- a reference to '**JavaBuildInTypesCF.DLL**', the MinCor.NET Java Connectivity runtime located in the MinCor.NET bin directory
- the generated file '**Generated\Hello.cs**'

The code is very similar to the client for the Java RMI server (see the last chapter) with the only difference that we have to initialize the ORB with the address for the AppServers Naming

# Java Connectivity

---

Service (assuming that a local installation of the *Sun ONE AppServer* with its CORBA Naming Service listening on port 3700 is used) and retrieve the server objects Home Interface reference to create a new Session Bean instance to call.

The .Net client implementation looks as follows:

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;

namespace DotNetClient
{
    public class ClientImpl
    {
        // the substring 'localhost' s just a pattern for the replacement with the server address...
        private static string[] m_strORBInit =
            {"-ORBInitRef NameService=corbaloc:iiop:1.2@localhost:1050/NameService"};

        public static void runDemo( FrmClt a_oFrmClt, string a_strIpAddr )
        {
            Hello.Greetings oGreetings;
            Midsol.CORBA.ORB oOrb = null;

            // ...done here:
            m_strORBInit[0] = m_strORBInit[0].Replace("localhost", a_strIpAddr );

            try
            {
                // initialize the Orb:
                oOrb = Midsol.CORBA._ORB.init( m_strORBInit, null );

                // retrieve the Naming Service from the ORB:
                Midsol.CosNaming.NamingContextExt oNC =
                    Midsol.CosNaming.NamingContextExtHelper.narrow(
                        oOrb.resolve_initial_references("NameService"));

                try
                {
                    // retrieve the server objects Home interface from the Naming Service:
                    Hello.Greetings_Home oGreetingsHome =
                        Hello.Greetings_HomeHelper.narrow( oNC.resolve_str("ejb/Hello"));

                    // create a new server object using the Home interface:
                    oGreetings = oGreetingsHome.create();
                }
                catch(Midsol.CosNaming.NamingContextPackage.NotFound ex)
                {
                    a_oFrmClt.writeLog("NS Manager Error: " + ex.why);
                    throw new System.Exception();
                }
            }

            // call the server object:
        }
    }
}
```

```
        string strRet = oGreetings.hello("Middsol");
        a_oFrmClt.writeLog(strRet);
    }
    catch( Middsol.CORBA.SystemException exSys)
    {
        a_oFrmClt.writeLog("Catch Exception from Server (Middsol.CORBA.SystemException).");
        a_oFrmClt.writeLog("ID:" + exSys.ID);
    }
    finally
    {
        // clean up:
        if( oOrb != null)
            oOrb.destroy();
    }
}
}
```

## **Step 5. Run the Example**

Start the *Sun ONE AppServer* and deploy your bean by copying the *Hello.ear* into the *AppServers* autodeploy directory (from the Java project root):

```
> asadmin start-domain domain1
> copy ..\assemble\ear\Hello.ear c:\Sun\AppServer\domains\domain1\autodeploy
```

At last, copy the .Net mobile client to the target platform and start the client.

After providing the correct server address, the screen should look similar to the following:



## 6 Java-to-C# -Mapping

In this section we will give a brief overview on the Java-to-C# mapping. Most Java types have a direct IDL and therefore a C# counterpart. The basis for this mapping is the original OMG Java-to-IDL-mapping of Java RMI types into CORBA IDL.

### **Basic Data Types**

The following table lists the basic Java data types and their mapping into OMG IDL and C#:

Java	IDL Type	.Net / C#
void	void	void
boolean	boolean	bool
char	wchar	char

# Java Connectivity

---

byte	octet	byte
short	short	short
int	long	int
long	long long	long
float	float	float
double	double	double
String	CORBA::WstringValue	string

## ***Mapping of Java Names***

Java names may contain characters illegal within OMG IDL identifiers, like '\$' or other illegal Unicode characters (i.e. outside ISO Latin 1). These characters will be replaced (according to the OMG Java-to-IDL mapping) with an 'U' followed by 4 upper case hexadecimal characters representing the Unicode value of the replaced character.

Another problem may be a leading underscore, since Java identifiers colliding with reserved OMG IDL keywords are enhanced by a leading underscore within IDL. To avoid possible name clashes with other identifiers resulting from this strategy, a leading underscore within a Java name will be replaced by 'J\_'.

Thus, e.g.:

- **\_name** maps into **J\_name**
- **<Unicode char>** maps into **Uxxxx**

## ***RMI Remote Interfaces***

An Java RMI Remote interface (java.rmi.Remote) is straight-forwardly mapped into an IDL interface, which in turn maps according to the usual MinCor.NET IDL-to-C#-mapping into a .Net interface derived from Midsol.CORBA.Object.

Property accessors (pairs of set/get<property>() methods) will result in standard .Net attributes.

If the Java interface contains overloaded method signatures, these signatures cannot be mapped directly into overloaded .Net/C# signatures, because OMGs IDL does not allow

method overloading. Thus, the OMG Java-to-IDL mapping defines a simple name mangling scheme for overloaded methods.

If a method is unique in a base interface its name will not be mangled within that interface, but if the method is overloaded the name is enhanced within that derived interface by adding the parameter type names to the original method name, e.g.:

- **void hello(boolean b)** maps into **void hello\_\_boolean(bool b)**
- **void hello(int x)** maps into **void hello\_\_long(int x)**
- **void hello(string n)** maps into **void hello\_\_CORBA\_ WStringValue(string n)**

Take care of the fact that the generated name contains the *IDL* types, not the .Net types.

## **Java Serializables**

RMI allows transporting objects by value – provided they are Serializables (i.e. they implement the interface `java.lang.Serializable`). These objects are mapped into IDL ValueTypes and therefore result in C# classes derived from `Middsol.CORBA.portable.StreamableValue` (or `CustomValue`).

By contrast to a pure Java RMI environment, CORBA cannot move the code behind any object functionality together with the object's state (because the target environment of the transport may be on a different platform, implemented by a different programming language) Therefore, a separate implementation for the value type has to be provided on the transports target side – i.e. the C# side – as well.

## **Arrays**

Java arrays map directly into C# arrays.

## **Java Exceptions**

Usually, a Java application will use a more or less elaborated hierarchy of exception classes to realize a fine grained exception handling. OMG IDL does not yet allow subclassing exceptions. Therefore, the OMG Java-to-IDL-mapping defines a mapping of the Java user exception into two classes:

# Java Connectivity

---

- A CORBA value type (which in fact can be derived from another value type) representing the Java user exception. This results on the C# side in a class derived from Midsol.java.lang.\_Exception and Midsol.CORBA.portable.StreamableValue.
- A standard CORBA exception with a "value" attribute containing this value type. The name of this exception will be generated by first removing any trailing "Exception" suffix and secondly, adding a new "Ex" suffix. This results on the C# side in a class derived from the usual Midsol.CORBA.UserException.

Be aware of the fact that for the Java user exception (as for all value types) an implementation has to be provided on the C# side.

## ***Inner Classes***

In Java, a class can contain inner classes. Due to the OMG Java-to-IDL-mapping, such an inner class is mapped into a separate IDL interface with a composite name formed by concatenating the name of the outer class, two underscores, and the name of the inner class, which in turn maps into a standalone C# class, e.g.:

- inner **class TheInner** within **class TheOuter**
  - maps into IDL                    **interface TheOuter\_\_TheInner**
  - maps into .Net                    **class TheOuter\_\_TheInner**

## ***Mapping of Special Classes***

Finally, there are some popular Java classes which either directly map into their .Net counterpart or they are represented by value types with an underlying .Net type.

Java	.Net / C#	underlying .Net/C# type
java.net.URI	System.Uri	System.Uri
java.net.URL	Midsol.java.net.URL	System.Uri
java.net.InetAddress	System.Net.IPHostEntry	System.Net.IPHostEntry
java.net.SocketAddress	System.Net.EndPoint	System.Net.EndPoint

# Java Connectivity

java.net.InetSocketAddress	System.Net.IPEndPoint	System.Net.IPEndPoint
java.sql.Date	Middsol.java.sql.Date	System.DateTime
java.sql.Time	Middsol.java.sql.Time	System.DateTime
java.sql.TimeStamp	Middsol.java.sql.TimeStamp	System.DateTime
java.util.ArrayList	System.Collections.ArrayList	System.Collections.ArrayList
Java.util.BitSet	System.Collections.BitArray	System.Collections.BitArray
java.util.Calender	Middsol.java.util.Calender	<i>generic</i>
java.util.GregorianCalendar	Middsol.java.util.GregorianCalendar	<i>generic</i>
java.util.Date	System.DateTime	System.DateTime
java.util.Hashtable	System.Collections.Hashtable	System.Collections.Hashtable
java.util.HashMap	Middsol.java.util.HashMap	System.Collections.Hashtable
java.util.HashSet	Middsol.java.util.HashSet	System.Collections.ArrayList
java.util.IdentityHashMap	Middsol.java.util.IdentityHashMap	System.Collections.Hashtable
java.util.LinkedHashMap	Middsol.java.util.LinkedHashMap	System.Collections.Hashtable
java.util.LinkedHashSet	Middsol.java.util.LinkedHashSet	System.Collections.ArrayList
java.util.LinkedList	Middsol.java.util.LinkedList	System.Collections.ArrayList
java.util.Locale	System.Globalization.CultureInfo	System.Globalization.CultureInfo
java.util.Properties	Middsol.java.util.Properties	System.Collections.Hashtable
java.util.Random	Middsol.java.util.Random	<i>generic</i>
java.util.SimpleTimeZone	Middsol.java.util.SimpleTimeZone	<i>generic</i>
java.util.Stack	System.Collections.Stack	System.Collections.Stack
java.util.TreeMap	Middsol.java.util.TreeMap	System.Collections.Hashtable
java.util.TreeSet	Middsol.java.util.TreeSet	System.Collections.ArrayList
java.util.Vector	Middsol.java.util.Vector	System.Collections.ArrayList
javax.naming.Name	Middsol.javax.naming.Name	<i>generic</i>
javax.naming.CompositeName	Middsol.javax.naming.CompositeName	System.Collections.ArrayList
javax.naming.CompoundName	Middsol.javax.naming.CompoundName	System.Collections.ArrayList

The underlying .Net type can be accessed by a property with a corresponding name, i.e. an underlying 'ArrayList' is accessed by a property called 'ArrayList'. Some of the mapped Java types do not have a suitable .Net type which could represent this type. These so-called

generic types are essentially data containers which are meant to be used in a Java environment once they are transferred there. If the data needed to be interpreted on the .Net side this should be done with a remote Java interface as delegate.

## ***Built In Java Exceptions***

A large number of Java exceptions and errors have been implemented. Since their use is fairly straight forward and uniform, an explicit listing is not necessary. Each corresponding .Net exceptions carries the same name as the Java exception with Midsol as prefix, e.g. **Midsol.java.lang.ArithmeticException**. In contrast to the specified Java-to-IDL mapping and therefore different to the use of Java user exceptions the Built In Java Exceptions surface directly as .Net exception. An IDL exception as transport container is not used. The **ToString()** method in the base of each exception allows to display reason and stack trace information:

```
try
{
    ...
}
catch( Midsol.java.lang.ArithmeticException ex )
{
    System.Console.WriteLine(ex);
}
```

## ***7 Java Connectivity Tools***

### ***The IDL Generator - MCjava2idl.exe***

The Java-to-IDL Generator can generate a valid OMG IDL code out of any EJB-JAR file, as well as from Java class files (optionally directly scanned from a given directory).

usage: **MCjava2idl [options] [MCP-file | class name list | JAR-/DIR- list]**

Legal options are:

- **-cp <dir1;... | jar1;...>** Class search path of directories and jar files.
- **-d <dir>** Set destination path to <dir>.

- **-h** Print help message.
- **-keep** Do not overwrite existing file.
- **-m** Generate methods for value types.
- **-p** Create a single file per package.
- **-v** Trace compilation stages.

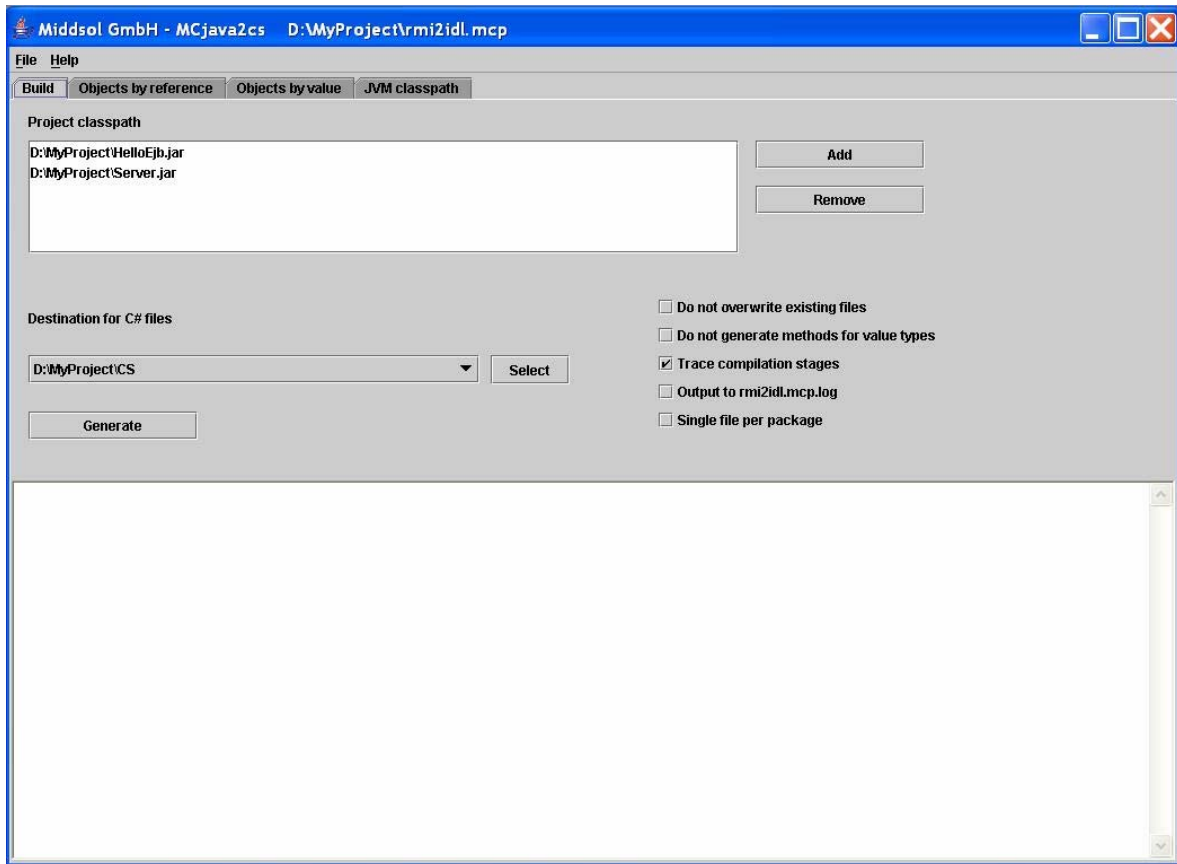
## ***The Graphical Front-end – MCjava2cs.exe***

MCjava2cs.exe represents a graphical front-end which integrates the IDL generator MCjava2idl.exe and the IDL compiler MCidl2cs.exe. This tool allows the user to collect Java JAR files and to select from them all Java RMI interfaces and objects which have to be accessed from the .Net client.

After the startup of the tool, all controls are initially disabled. The user has at first to load or create a MCP file (using the applications 'File' menu), which stores all selections and settings made by the user and serves as project file.

The project MCP file may also be passed manually while calling the command line tool, so that the results of the manual selection process using the graphical front-end can easily be reused within automated make processes.

Picture 1 - The 'Build' Tab



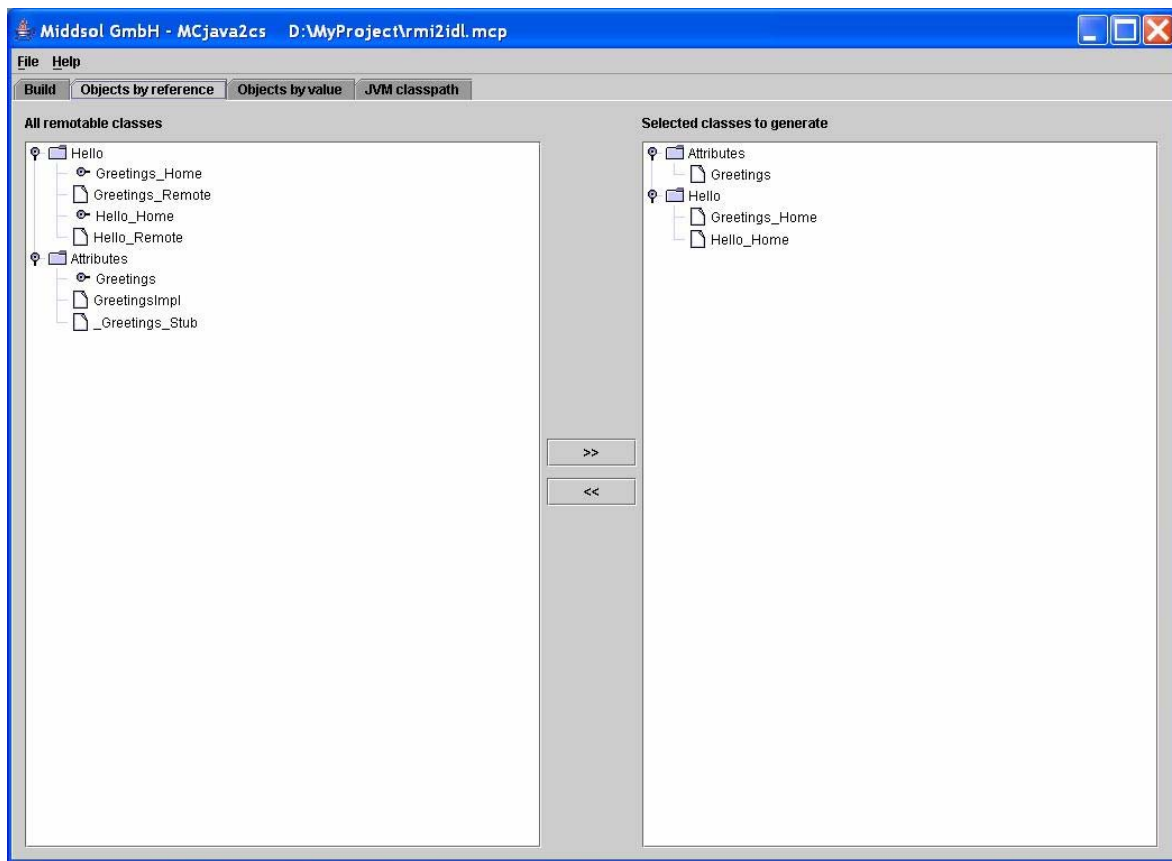
Having loaded a MCP file, the application presents itself like Picture 1:

- **Project classpath** contains all JARs already selected.  
(Use the 'Add' and 'Remove' buttons to manipulate the list.)
- **Destination for C# files** selects a target path for the generated C# files.
- **Generate** starts the generation of C# files.
- Several options:
  - **Do not overwrite existing files**, turns on/off regeneration of already generated files.
  - **Do not generate methods for value types**, turns on/off generation of business method signatures for value types. (Turning on means that only data containers without any methods are generated.)
  - **Trace compilation stages**, turns on/off production of generation process messages.

# Java Connectivity

- **Output to logfile**, turns on/off logging. (The logfile is found in the current project folder and has got the name of the project file with the suffix '.log' added.)
- **Single file per package**, turning on means that only one IDL file for every Java package is created; turning off means that separate IDL files are generated for every Java type.
- The lower box contains messages from the generation process.

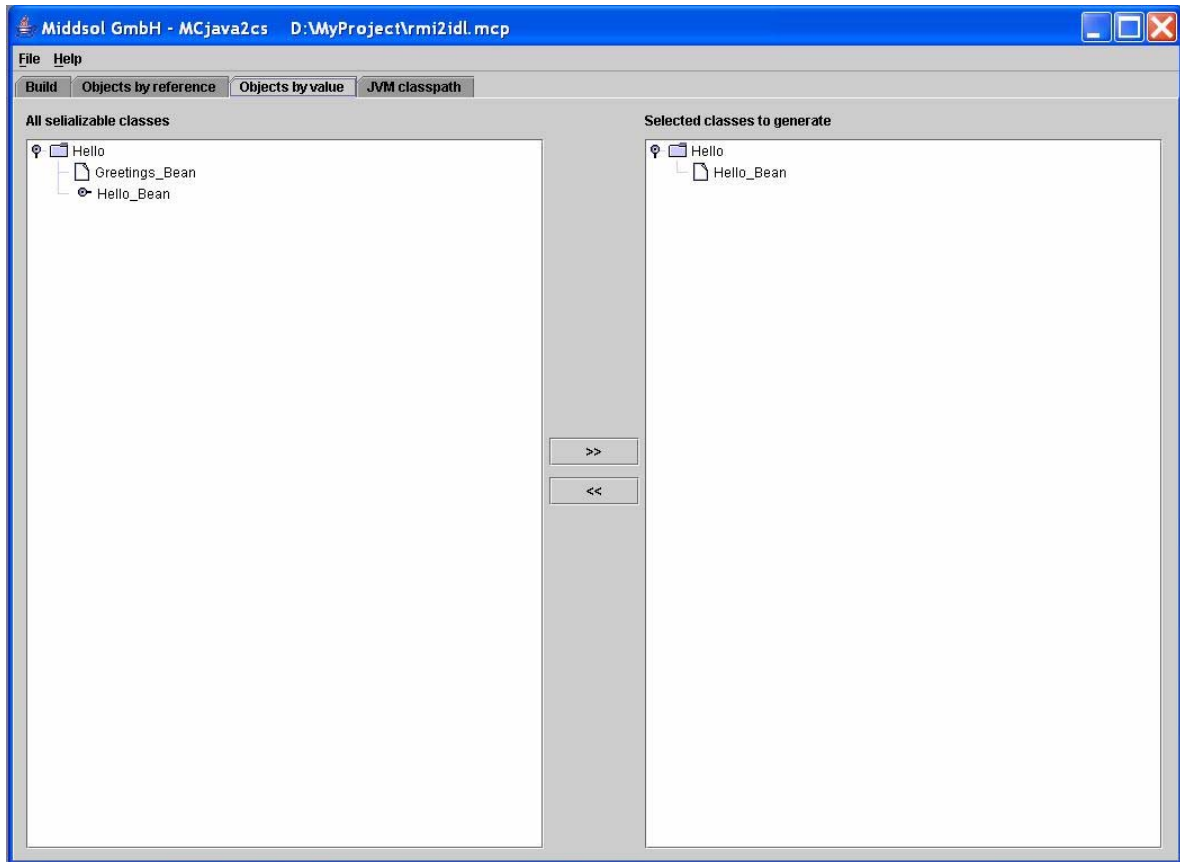
Picture 2 – The 'Objects by reference' Tab



Once there are some JARs within the project classpath, all interfaces within these JARs derived from `java.rmi.Remote`, as well as all classes implementing such interfaces, are listed on the second tab. All those interfaces scheduled for generating C# code from have to be selected explicitly.

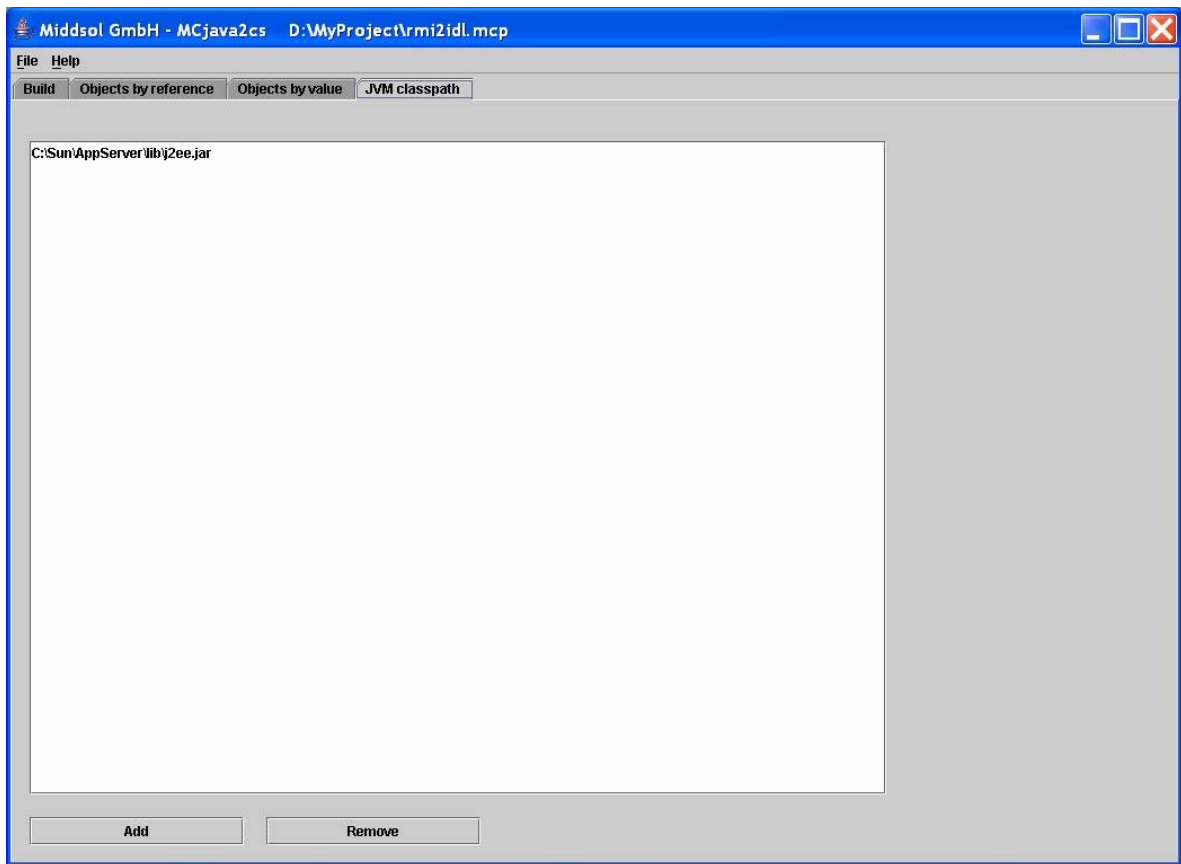
# Java Connectivity

Picture 3 - The 'Objects by value' Tab



On the third tab, all objects implementing `java.lang.Serializable` or `java.lang.Externalizable` within the selected JARs are listed. These are candidates for objects being passed 'by value'. Again all those classes scheduled for generating C# code from have to be selected explicitly.

Picture 4 - The 'JVM classpath' Tab



On the 'JVM classpath' tab the general classpath can be extended. When the project classes contain **EJBs** it has to be ensured that the library **j2ee.jar** (part of the J2EE SDK) is included.

## 8 Java Runtime Version

In order to cope with different Java versions the class **Middsol.Runtime.JavaRuntime** has been introduced. It contains the static property **CurrentRuntimeVersion** ( of type integer ) which allows to set one of the following constants:

- **JAVA\_VERSION\_141** for the Java Runtime Version 1.4.1
- **JAVA\_VERSION\_142** for the Java Runtime Version 1.4.2
- **JAVA\_VERSION\_150** for the Java Runtime Version 1.5.0

The default value is **JAVA\_VERSION\_142**.

## 9 Sources and References

### Web References

- The Sun Java homepage: <http://java.sun.com/>
- The OMG homepage: <http://www.omg.org/>

### Books

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: *Pattern-Oriented Software Architecture – A System of Patterns*, Wiley & Sons, 1996
- Brose, G., Vogel, A., and Duddy, K.: *Java Programming with CORBA*. Wiley & Sons, 2001.
- Dudney, B., Asbury, S., Krozak, J.K., and Wittkopf, K.: *J2EE AntiPatterns*. Wiley & Sons, 2003
- Gamma, E, Helm, R, Johnson, R., and Vlissides, J.: *Design Patterns*. Addison-Wesley, 1995.
- Marinescu, F.: *EJB Design Patterns*. Wiley & Sons, 2002
- Monson-Haefel, R.: *Enterprise JavaBeans*. O'Reilly, 1999
- Roman, E.: *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition*. Wiley & Sons, 1999