



Zachodniopomorski  
Uniwersytet  
Technologiczny  
w Szczecinie



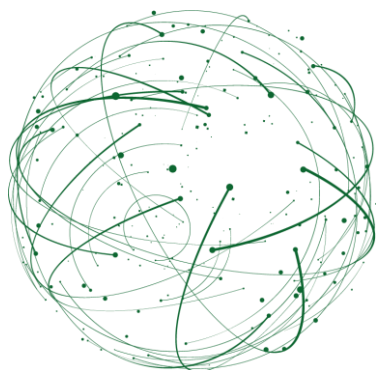
Wydział  
Informatyki



KATEDRA INŻYNIERII OPROGRAMOWANIA

<http://wi.zut.edu.pl/>

*PROJEKTOWANIE OPROGRAMOWANIA/SOFTWARE DESIGN*



# Aplikacje Webowe i Rozproszone

## #02 : Remote Method Invocation (RMI)

*Prowadzący:*

*Krzysztof Kraska*

*email: [kkraska@zut.edu.pl](mailto:kkraska@zut.edu.pl)*

---

Szczecin, 29 marca 2020 r.

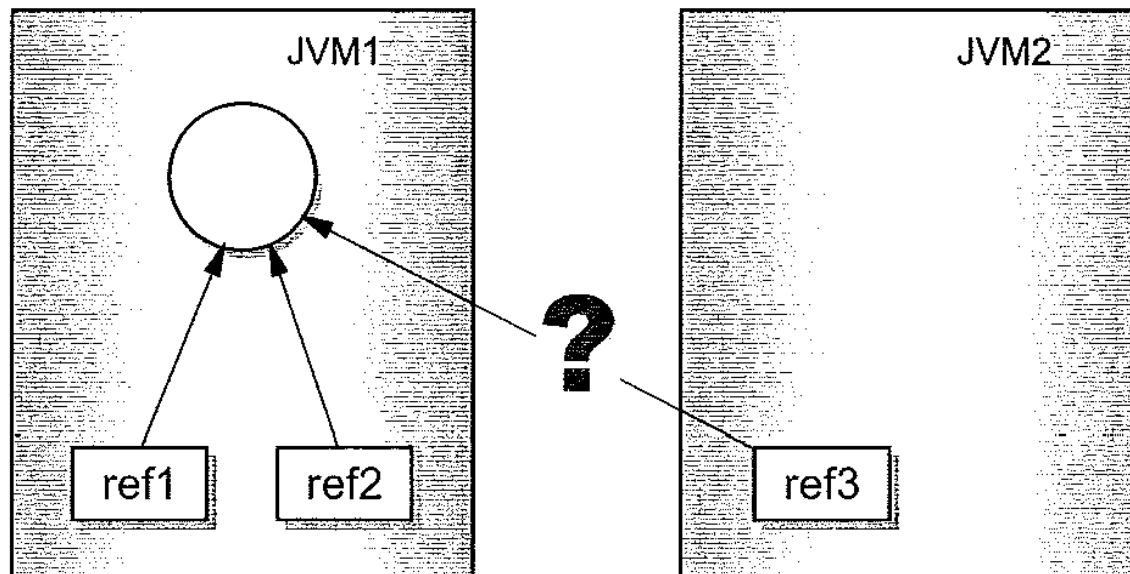
# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Paradigm Shift

---

Move from sharing resources to sharing objects.



RMI only works between Java applications.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

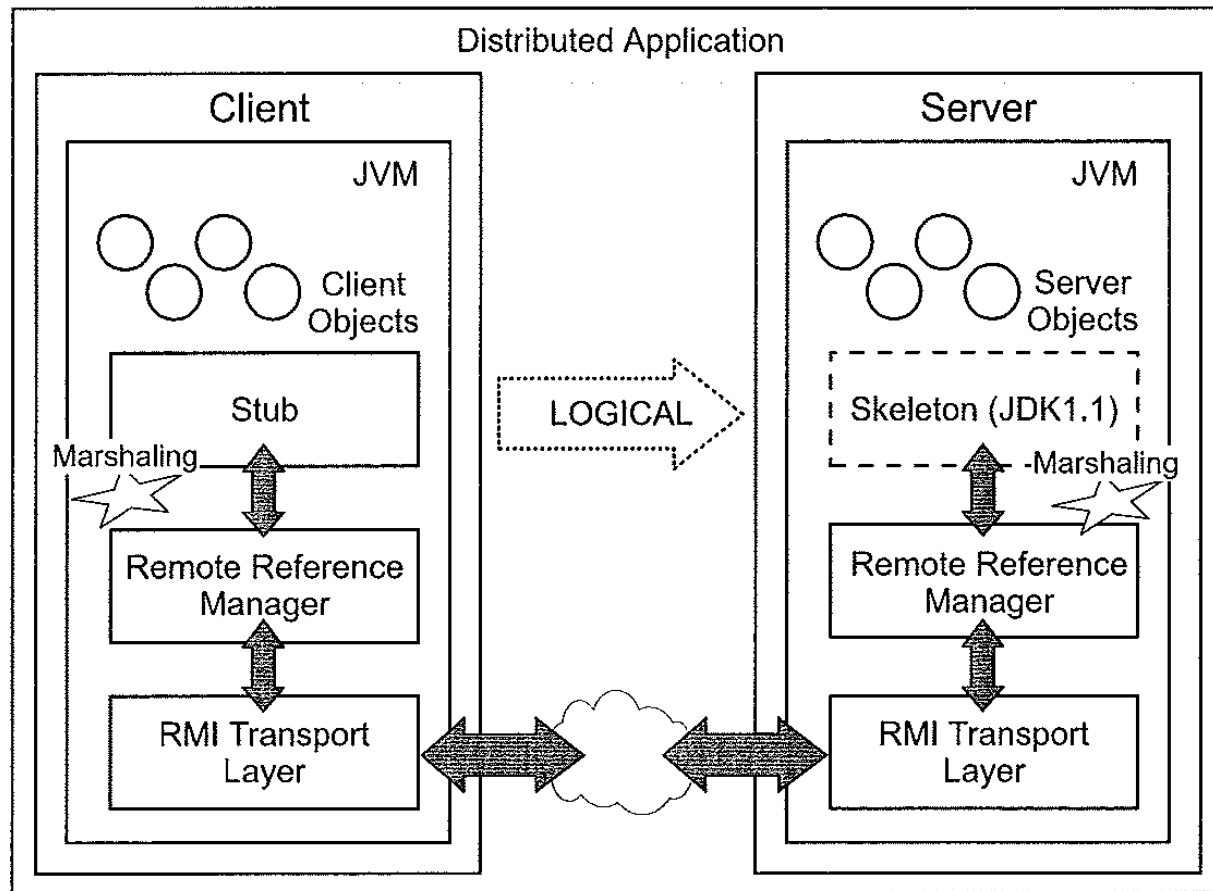
## RMI Architecture

- The RMI API allows access to a remote server object from a client program by making simple method calls on the server object
- Need proxy objects which support a remote interface and provide physical connection and communications
- The RMI Architecture consists of three layers:
  - Stub/Skeleton layer – client-side stubs and server-side skeletons
  - Remote reference layer – interprets and manages references to **remote** objects
  - Transport layer – connection set up and management, also remote object tracking
- RMI/JRMP is Java to Java (Java Remote Method Protocol)

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

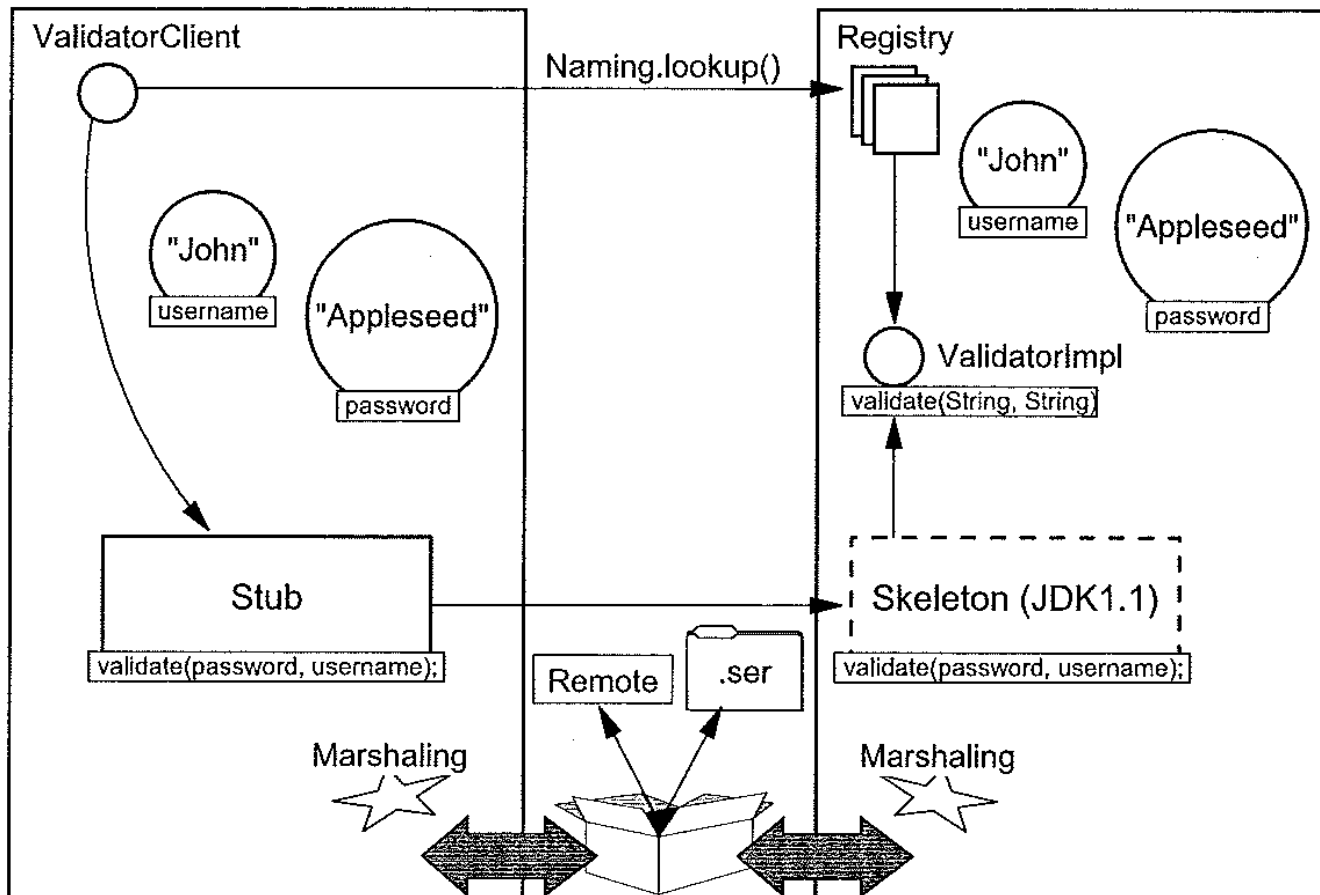
## Remote Method Invocation



# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Communicating with Remote Object



# REMOTE METHOD INVOCATION

## • APLIKACJE WEBOWE I ROZPROSZONE •

If the object is a Remote object, a remote reference for the object is generated, and the reference is marshaled and sent to the remote process.

If the object is serializable it is converted to bytes and sent to the remote process in byte form.

If the method argument is neither remote nor serializable, the argument cannot be sent to the client and a `java.rmi.MarshalException` is thrown.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Remote Interfaces

- The heart of RMI is the definition of a **Remote Interface**
- A remote interface is a Java interface which extends `java.rmi.Remote`
- All methods of a remote interface must include `java.rmi.RemoteException` in their throws list
- Method arguments and return values:
  - **Primitive** – the value is passed
  - **Reference to a local object** – (local reference) an object copy is passed (Serialization)
  - **Reference to a remote object** via a remote interface (remote reference) – a serialized remote reference is passed
- A local reference to a remote object is actually a reference to a proxy (stub) object

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Class Loading

- The `RMIClassLoader` is used to load the stubs of remote objects, remote interfaces, and extended classes of arguments and return values to RMI calls
- Available with JRMP not IIOP as the underlying protocol
- The `RMIClassLoader` looks for bytecode in the following locations:
  - The local `CLASSPATH`
  - The URL that is encoded in the marshal client stream associated with the serialized object
  - The URL specified in the  
`java.rmi.server.codebase` system property
- If the `RMIClassLoader` is required to load classes over the network, a security manager must be in place



# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Dynamic RMI Class Loading

- The ability to dynamically download Java code from any URL to a JVM running in a separate process, usually on a different physical system.
- A Java application running RMI can determine whether to load a class from a remote location, the `RMIClassLoader` is called to do this work.
- The loading of RMI classes is controlled by a number of properties, which can be set when each JVM is run
- The property `java.rmi.server.codebase` is used to specify a URL which points to a `file:`, `ftp:`, or `http:` location that supplies classes for objects that are sent *from* this JVM.
- A security manager and policy file must be used by the client to allow only trusted files to be loaded.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Configurations

---

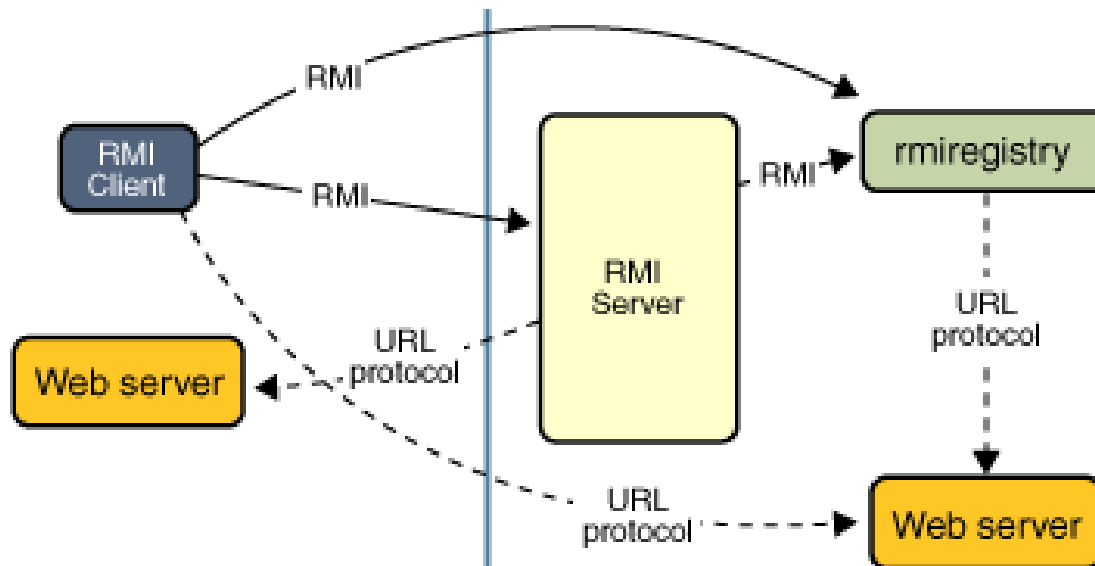
- **Closed Server System** – does not set the `java.rmi.server.codebase` property, and server loads no security manager (server can only load classes from its own CLASSPATH)
- **Bootstrapped System** (client and server) – sets the `java.rmi.server.codebase` property, it must then load a security manager and set the `java.rmi.server.useCodebaseOnly` property `true` which disables loading of any classes from client supplied URLs
- **Dynamic System** (client and server) – server will set the `java.rmi.server.codebase` property, it must then load a security manager which enables loading of all classes from all supplied URLs

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Configurations

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Remote Object Garbage Collection

- When a remotely accessible object (via skeleton) is exported from a server, a stub is serialized to the client, and the stub's host is added to the skeleton's reference set.
- When the stub is garbage collected in a VM, an unreferenced message is sent back to the skeleton's host.
- Thus the remote object will only be garbage collected when there are no **local or remote** references for the object.
- A remote object can implement the `java.rmi.server.Unreferenced` interface and implement its required method, `unreferenced()`. This method is called by the Distributed Garbage Collector when it removes the last remote reference to the object.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Registry

- The current RMI implementation uses a simple remote object registry (name server):

```
rmiregistry <port> // default 1099
```

- Access to the naming services of the registry is provided in the class `java.rmi.Naming`
- Currently, `rmiregistry` is remotely accessible by providing a URL string:

```
rmi://host:port/name
```

- To register a remote object use:

```
void bind(String, Remote) or  
void rebind(String, Remote)
```

- To obtain a server reference use:

```
Remote lookup(String)
```

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Registry

---

A client can enumerate all registered RMI objects by calling:

```
import javax.naming.NameClassPair;
...
Context context = new InitialContext();
Enumeration<NameClassPair> e = context.list("rmi://regserver.mycompany.com");
```

NameClassPair is a helper class that contains both the name of the bound object and the name of its class. For example, the following code displays the names of all registered objects:

```
while (e.hasMoreElements())
    System.out.println(e.nextElement().getName());
```

---

The following code is done to create and get the RMI registry running on the server :

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
...
LocateRegistry.createRegistry(1099);
...
Registry registry = LocateRegistry.getRegistry( );
```

---

Szczecin, 29 marca 2020 r.

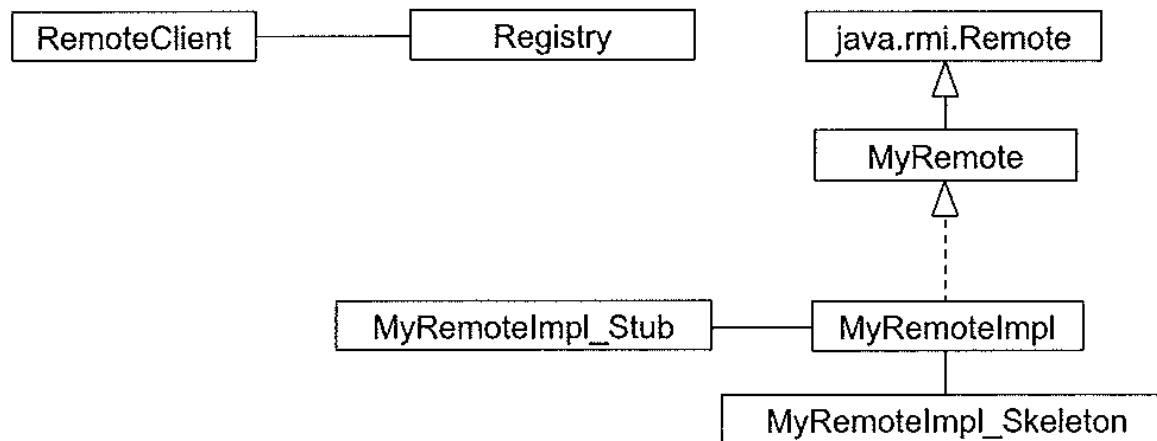
# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Creating a Remote Object

---

1. Extend the Remote Interface.
2. Implement the new interface and bind it to a naming service.
3. Generate stubs and skeletons that are needed for the remote implementations by using the rmic program.
4. Create a client program that will make RMI calls to the server.
5. Start the Registry and run your remote server and client.





# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## RMI Distributed Object Construction

These are the required steps in developing a Closed Server RMI application:

Step 1 - Define interface for the remote class

Step 2 - Create an implementation class for the remote object

Step 3 - Create a server application to host the remote object interface and bind to the rmi registry

Step 4 - Generate stub class using the J2SDK `rmic` tool

Step 5 - Start the RMI registry and the server application on the server machine

Step 6 - Create a client application to access the remote objects

Step 7 - Execute client application



# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 1 - Define the Remote Object Interface

Only methods defined  
in the remote interface  
are available to client.

The java.rmi.Remote interface  
contains no methods of its own;  
it is a marker interface.

The RemoteException class is  
the superclass of most of the  
exceptions that can be thrown  
when RMI is used.

```
import java.rmi.*;  
  
public interface Validator extends Remote(  
    String validate(String aUserName, String aPassword)  
        throws RemoteException;  
)
```

All method arguments and  
return types in remote  
interfaces must implement  
Serializable, or be references  
to remote objects themselves.

# REMOTE METHOD INVOCATION

## • APLIKACJE WEBOWE I ROZPROSZONE •

Any method defined in the remote interface must throw a *RemoteException*. Remote methods depend on many things that are not under our control: for example, the state of the network and other necessary services such as DNS. Therefore, all code in the implementing class has to be in a `try{} block`, or the method must throw the exception. *RemoteException* is the superclass of all remote exceptions. There are 16 exceptions defined in the *java.rmi package*; they are all checked exceptions.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 2 - Create the Implementing Class

UnicastRemoteObject provides a number of methods that make RMI work. Its main job is to marshal and unmarshal remote references to the object.

Marshalling

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

public class ValidatorImpl extends UnicastRemoteObject implements Validator {
    //UnicastRemoteObject implements Remote interface
    Map memberMap;

    public ValidatorImpl() throws RemoteException {
        memberMap = new HashMap();
        memberMap.put("John", "Appleseed"); //could add several records here
    }

    public String validate(String aUserName, String aPassword)
        throws RemoteException {
        if (getMemberMap().containsKey(aUserName) &&
            getMemberMap().get(aUserName).equals(aPassword))
            return "Welcome " + aUserName; //early return if password is valid
        return "Sorry invalid login information!";
    }

    public Map getMemberMap() {
        return memberMap;
    }
}
```

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 2 – Create the Implementing Class

---

Occasionally, you might not want to extend the `UnicastRemoteObject` class, perhaps because your implementation class already extends another class. In that situation, you need to manually instantiate the remote objects and pass them to the static `exportObject` method. Instead of extending `UnicastRemoteObject`, call:

```
UnicastRemoteObject.exportObject(this, 0);
```

in the constructor of the remote object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 3 – Create Server Application

---

```
import java.net.*;
import java.rmi.*;
public class LoginServer{

    public static void main(String args[]){
        try{
            ValidatorImpl aValidator = new ValidatorImpl();
            Naming.rebind("validator", aValidator);
            System.out.println("Login server open for business");
        }catch(RemoteException e){ //catches a RemoteException
            e.printStackTrace();
        }catch(MalformedURLException me){
            System.out.println("MalformedURLException " + me);
        }
    }
}
```

A registry keeps track of the available objects on an RMI server and the names by which they can be requested. The object is added to the registry with the `Naming.bind()` or `Naming.rebind()` methods.

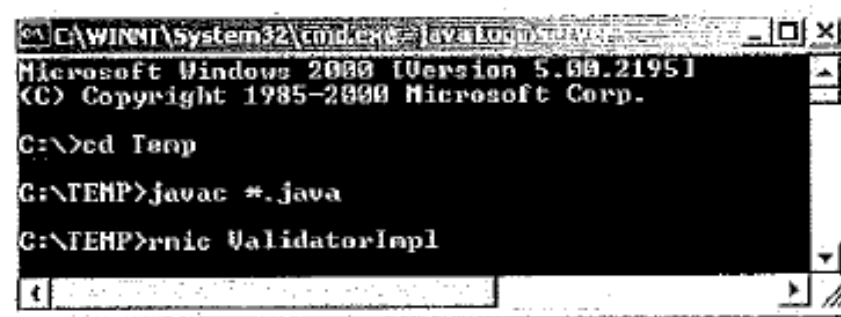
Registry listens on port 1099 unless otherwise specified.

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 4 - Generate the Stubs/Skeletons

- Stub represents the interface on the client  
`ValidateImpl_Stub.class`
- Skeleton bridges the interface to the server  
`ValidateImpl_Skel.class`
- rmic compiler generates stub and skeleton classes  
`C:\Temp>rmic ValidateImpl`



```
C:\WINNT\system32\cmd.exe - java1.4.0_04
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd Temp
G:\TEMP>javac *.java
G:\TEMP>rmic ValidatorImpl
```

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 5 – Start RMI Registry and Server Application

---

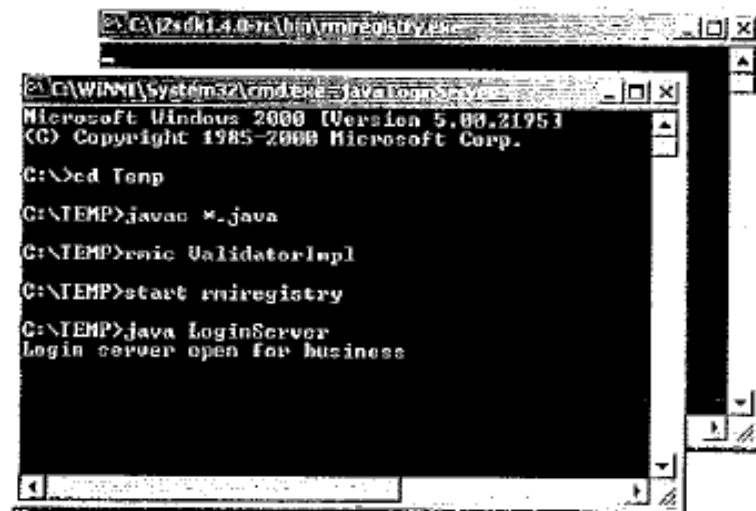
- Start the rmi registry (Naming Registry). It runs in its own window.

```
C:>start rmiregistry
```

- Finally start our own server

```
C:>java LoginServer
```

Outputs: Login Server open for business



# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 6 - Create the Client

Naming.lookup("rmi://objhost.org:port/validator");

protocol hostname registered name

```
import java.rmi.*;
import java.net.*;

public class ValidatorClient{
    public static void main(String args[]){
        if(args.length==0||!args[0].startsWith("rmi:")){
            System.out.println("Usage: java ValidatorClient" +
                               "rmi://host.domain.port/validator username password");
        }
        try{
            Object remote = Naming.lookup(args[0]);
            Validator reply = (Validator)remote;
            System.out.println(reply.validate(args[1], args[2]));
        }catch(MalformedURLException me){
            System.out.println(args[0] + " is not a valid URL");
        }catch(RemoteException rbe){
            System.out.println("Could not find requested object on the server");
        }
    }
}
```

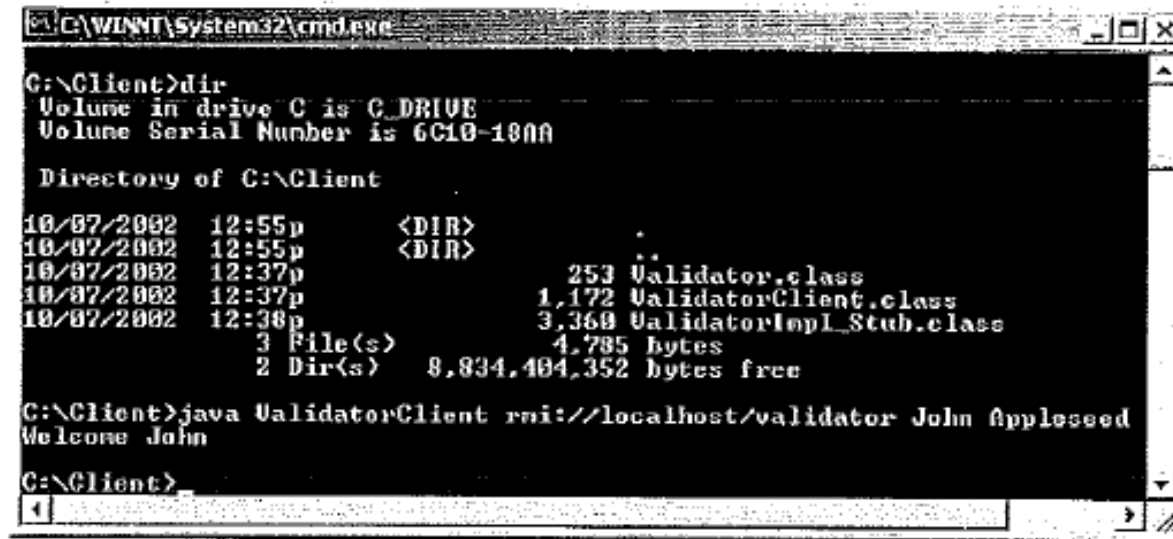


# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Step 7 – Execute Client Application

This is a Closed Server System example, so both the Validator interface and the ValidatorImpl\_stub that was generated have to be in the client JVM's CLASSPATH.



```
C:\WINNT\System32\cmd.exe

C:\Client>dir
Volume in drive C is C_DRIVE
Volume Serial Number is 6C10-1000

Directory of C:\Client

10/07/2002  12:55p    <DIR>          .
10/07/2002  12:55p    <DIR>          ..
10/07/2002  12:37p                253 Validator.class
10/07/2002  12:37p             1,172 ValidatorClient.class
10/07/2002  12:38p             3,360 ValidatorImpl_stub.class
                3 File(s)              4,785 bytes
                2 Dir(s)  8,834,404,352 bytes free

C:\Client>java ValidatorClient rmi://localhost/validator John Applesseed
Welcome John

C:\Client>
```

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Dynamic Client Configuration

To make this example dynamically load the stub, the following changes have to be made:

- ValidatorClient must install a security manager

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

- Policy file must be created with the following permissions

```
grant {  
    permission java.net.SocketPermission "*:1024-", "accept, connect";  
    permission java.io.FilePermission "${}/Temp${}/server${}/-" , "read";  
};
```

- Split the code into client and server subdirectories of Temp directory

```
client dir -> ValidatorClient.class, Validator.class, myPolicy.policy  
server dir -> LoginServer.class, Validator.class, ValidatorImpl.class,  
              ValidatorImpl_Stub.class, (ValidatorImpl_Skel.class)
```

- Using the codebase property, specify the file location containing the stub when executing the server

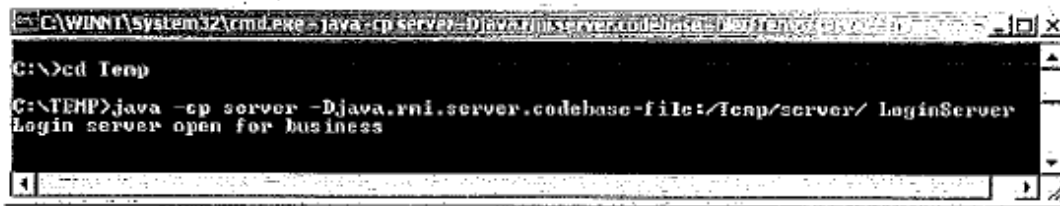
```
java.rmi.server.codebase=file://c:\Temp\server/
```

- Must start rmiregistry from a directory NOT in the classpath

# REMOTE METHOD INVOCATION

• APLIKACJE WEBOWE I ROZPROSZONE •

## Dynamic Client Configuration Example



```
C:\WINNT\System32\cmd.exe - java -cp server -Djava.rmi.server.codebase=file:/Temp/server/ LoginServer
C:\>cd Temp
C:\TEMP>java -cp server -Djava.rmi.server.codebase=file:/Temp/server/ LoginServer
Login server open for business
```

server started  
from c:\Temp



rmiregistry started  
from c:\

client started  
from c:\temp\client



```
C:\WINNT\System32\cmd.exe
C:\TEMP\client>java -Djava.security.policy=myPolicy.policy ValidatorClient rmi:/localhost/validator John Applesseed
Welcome John
C:\TEMP\client>
```

The trailing forward slash at the end of the codebase url  
-Djava.rmi.server.codebase=file:/Temp/server/ signifies that it is a directory. Unless a jar file  
is being referenced, the forward slash must be used.

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## What is a Naming Service?

---

- A service that relates human-friendly names to computer resources
- Name must adhere to a naming convention for that resource
- The name is bound to the actual resource or a resource reference
- Examples:

Resource Type	Name	Binding
Internet host	www.ibm.com	129.42.16.991
Windows file	C:\WINNT\java	Actual file handle
JDBC DataSource	jdbc\Library	Reference to DataSource object

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Some Naming Service Terms

- Context
  - A grouping of name – resource mappings (bindings)
  - A starting point for searching for the resource
  - Follows a naming convention
  - Provides services for adding, searching, removing bindings
- Naming system
  - Definition of:
    - A grouping of related contexts (all follow the same naming convention)
    - Services to manage and manipulate the bindings
- Naming Service
  - Actual code/product that implements the naming system
  - Examples are DNS and the Windows file system

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## What is a Directory Service?

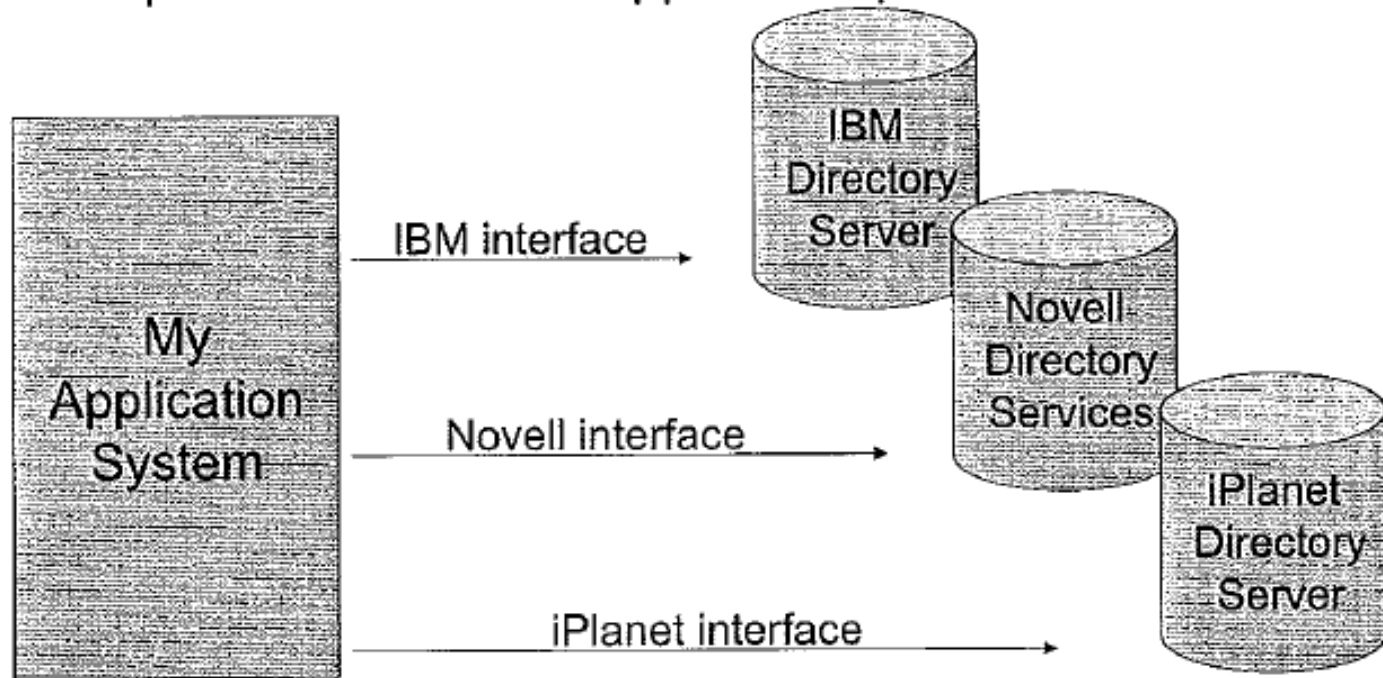
- Extension of a naming service
- Allows attributes for a resource
- Resources are organized in a hierarchical model
- Optimized for READ access
- Examples:
  - A user resource has userid and password attributes
  - A printer resource would have network address and printer option attributes
- Example directory products:
  - IBM Directory Server
  - Novell Directory Services
  - iPlanet Directory Server

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## A Critical Issue with Directory Services

- Each directory has its own interface
- A company may use different directories for different resources
- Developers have to learn/support multiple interfaces





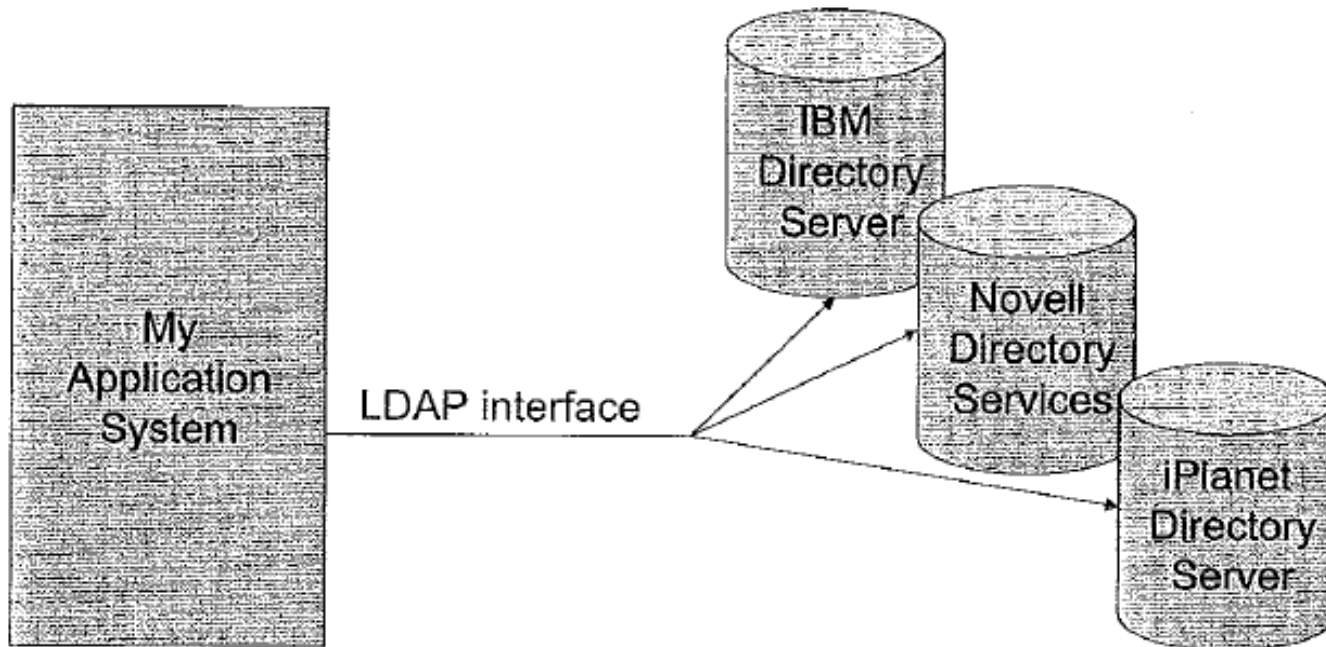
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## LDAP to the Rescue

---

- Lightweight Directory Access Protocol (LDAP)
  - IETF RFC 2251 (LDAP v3)
- Lightweight implementation of Directory Access Protocol (DAP) to access X.500-based directories
- Many directory services also provide an LDAP interface





# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## LDAP

---

- Provides a white pages (lookup by name) and a yellow pages (lookup by information/attributes).
- Resources are organized hierarchically in a directory information tree (DIT).
- A resource is known as an entry or object (not in the OO-sense), and is a leaf on the tree.
- Each entry can have a set of attributes. An attribute can have more than one value (for example, phone number)
- An entry has a unique distinguished name (DN), composed of attribute-value pairs.
- A special system attribute is objectclass, which defines the required attributes for an entry. ObjectClasses can be arranged hierarchically; attributes are additive.
- LDAP v3 specifies a schema concept, which defines the allowed entries and attributes.

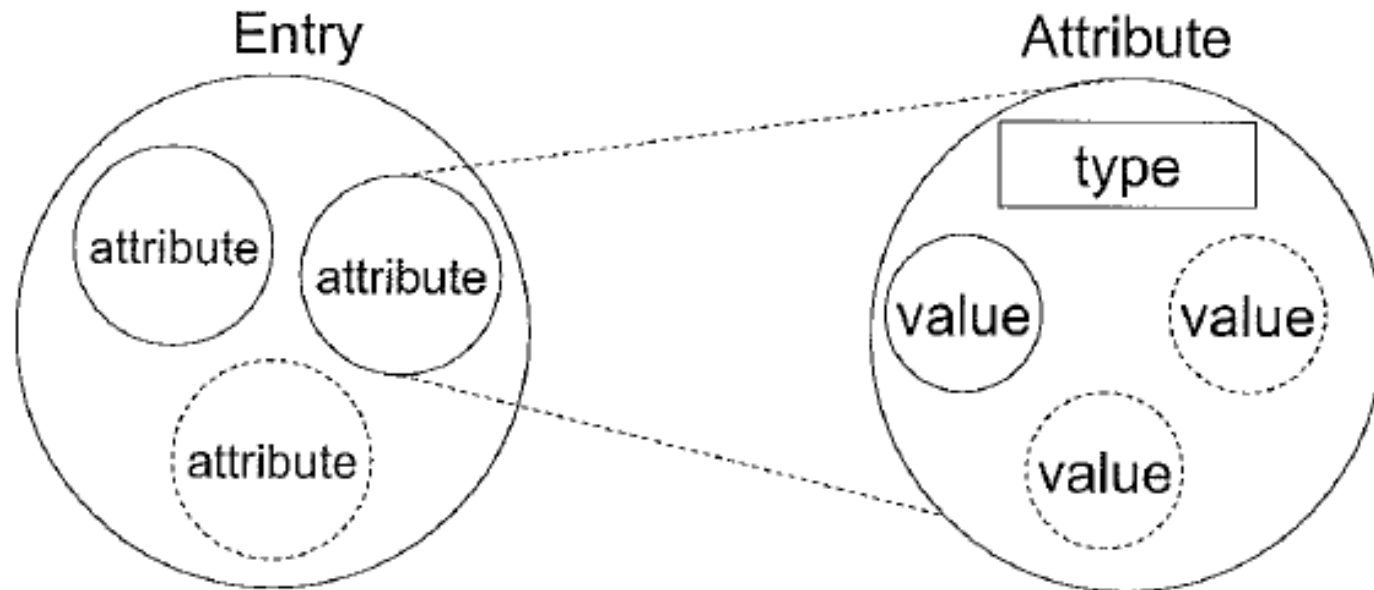
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Entry – Attribute Relationship

---

- An entry has required and possibly optional attributes
- An attribute has a type (syntax) and one or more values

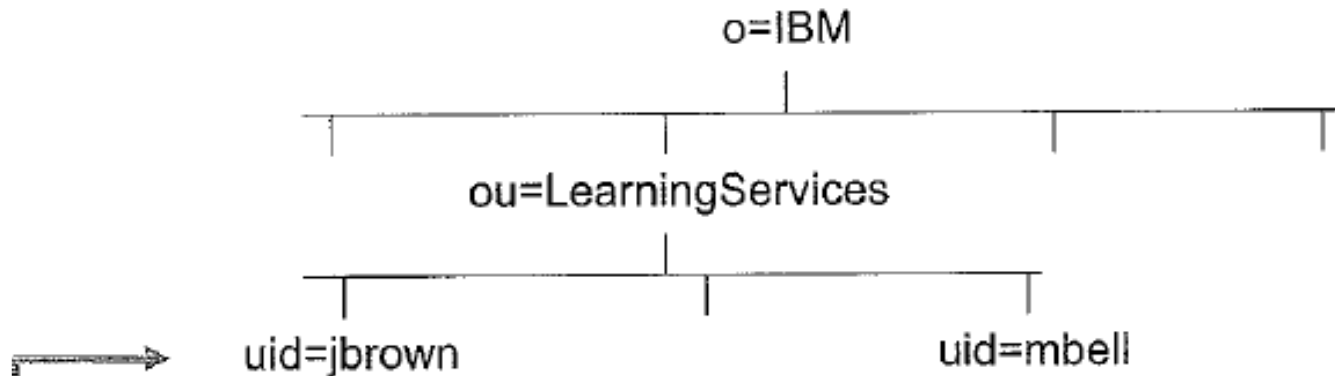


# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## LDAP Directory Entry Example

---



- Distinguished Name:

- uid=jbrown, ou=LearningServices, o=IBM

- Besides the DN, the entry also has required (and optional) attributes:

- cn=Jim Brown, sn=Brown, givenname=Jim, empid=12345,  
telephoneNumber=310-555-1212, telephoneNumber=310-555-9999

- The Relative Distinguished Name (RDN) is the leftmost part of the DN (uid=jbrown in this example)

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## LDAP Operations on the Directory

- Query
  - Search the directory
  - Compare an entry for a specific attribute value
- Update
  - Add an entry to the directory
  - Delete a leaf-node entry from the directory
  - Modify the attributes and values in an entry
  - Move an entry or subtree of entries elsewhere in the DIT
- Authentication
  - Bind (authenticate) between a client and directory
  - Terminate a client-directory connection
  - Abandon an operation outstanding on the directory

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## What Is JNDI?

---

- Java Naming and Directory Interface
  - Provides a Java API to access naming and directory services
  - Also allows storage/retrieval of Java objects
- Widely used by
  - JDBC DataSources
  - Enterprise JavaBeans (EJBs)

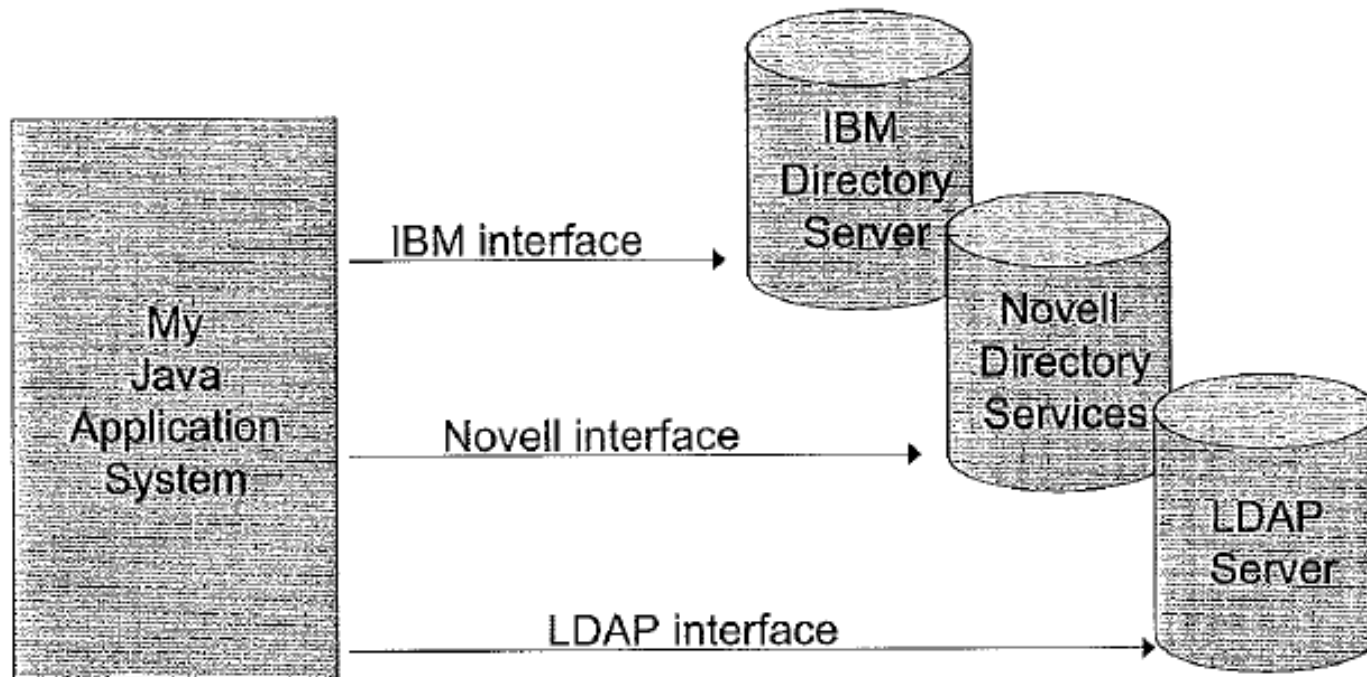
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Why JNDI? (1 of 2)

---

- Java applications accessing the directory services would need to code to each unique interface



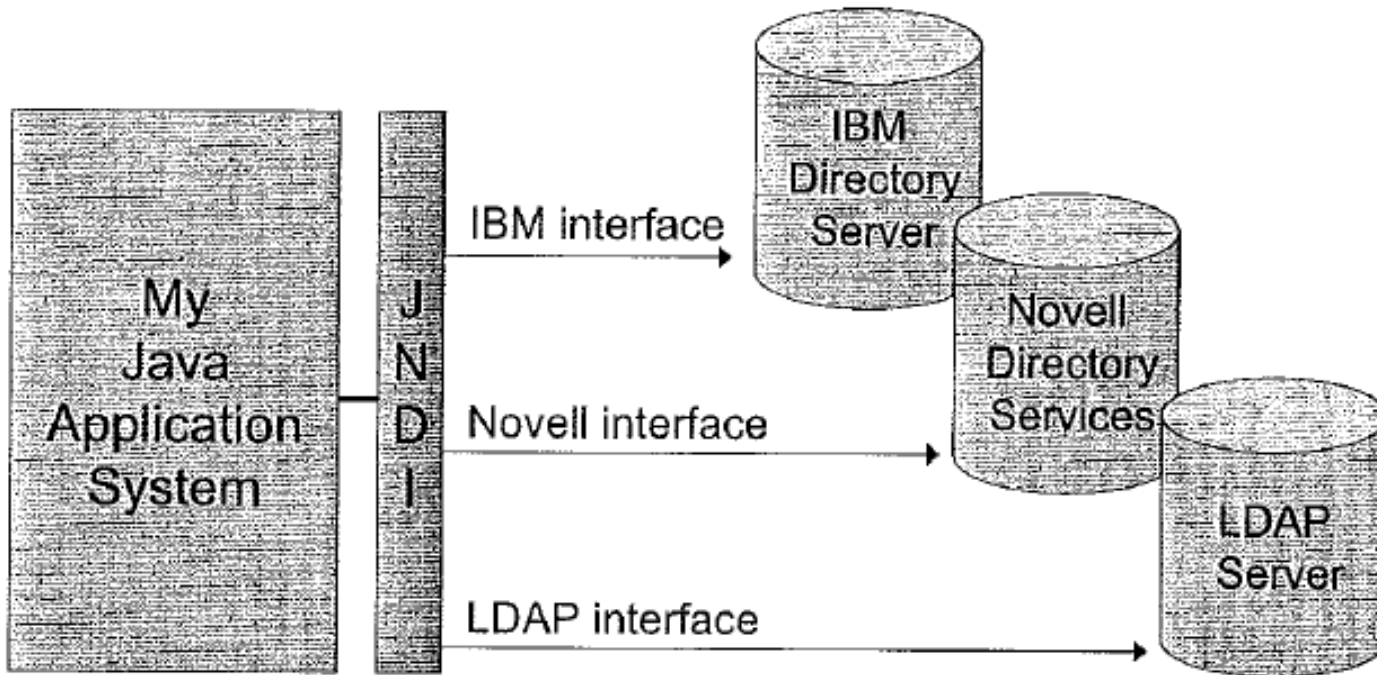
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Why JNDI? (2 of 2)

---

- Java applications using JNDI to access the directory services need to code only to the JNDI interface





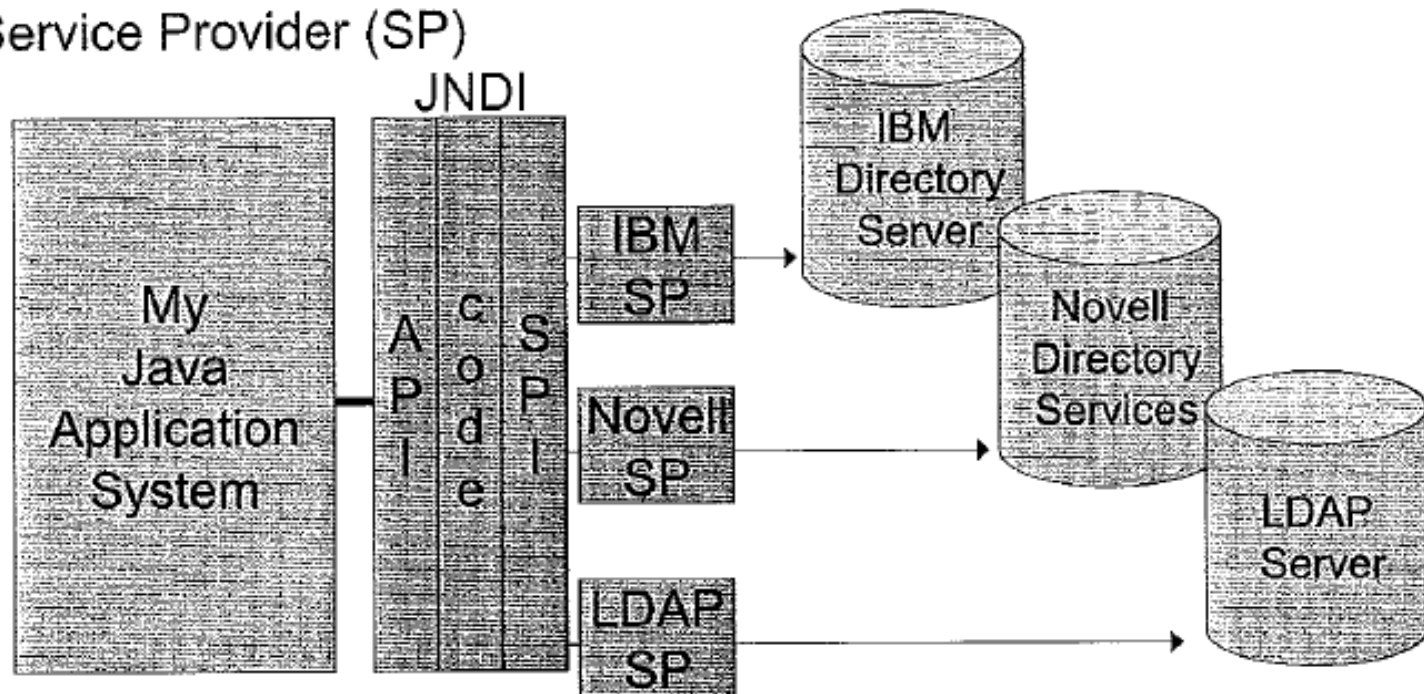
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## JNDI Structure

---

- Java applications using JNDI to access the directory services need to code only to the JNDI interface (API)
- The JNDI implementation code provides a Service Provider Interface (SPI), which allows each directory to plug in its own Service Provider (SP)





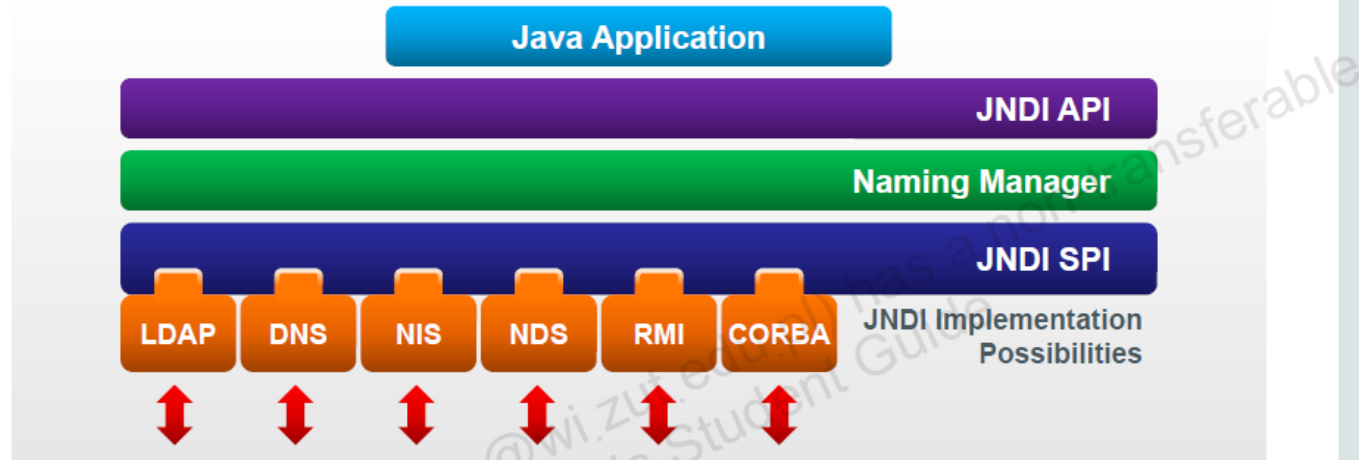
# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## JNDI

The Java Naming and Directory Interface (JNDI) is an API that provides naming and directory functionality to applications.

- Independent of any implementation
- Built on a service provider interface (SPI)
  - SPIs can be plugged in to an API.



ORACLE®

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Szczecin, 29 marca 2020 r.

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Service Provider

---

- Java implementation code that talks a specific directory protocol
- Implements the Context interface at a minimum, and usually the DirContext interface as well
- The JNDI packages from Sun contain Service Providers for
  - LDAP
  - CORBA COS
  - RMI registry
  - File system
- Directory service vendors can supply their own
  - <http://java.sun.com/products/jndi/serviceproviders.html>
- You can write your own.

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## JNDI Setup

---

- Have The JNDI packages available
  - JDK 1.4 already includes the JNDI packages
  - For pre-1.3 JDKs, you can download the packages from Sun
- Install a JNDI-accessible Directory Server
  - The JDK or the vendor supplies a Service Provider
- For Directory Services, define a schema that includes the objectClasses and attributes you need (LDAP)
- Write your application code to use the services!

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## JNDI API - Packages

---

- `javax.naming`
  - Core package for supporting naming services
  - Includes Context interface, InitialContext class, lookup operations, references, and NamingException
- `javax.naming.directory`
  - Adds support for directory services
  - Includes DirContext interface, InitialDirContext class, attribute manipulation operations, and search operations
- `javax.naming.event`
  - Classes and interfaces for directory event notification
- `javax.naming.ldap`
  - Additional directory support for LDAP v3
- `javax.naming.spi`
  - Support for user-or vendor-written service provider code
  - Also has support for accepting/sending Java objects

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Typical JNDI/LDAP Application Programming

- Connect to the directory server
- Bind an object to the server
- Directory entry operations
  - Search
  - Add
  - Modify
  - Delete
- Disconnect from the server

# JAVA NAMING AND DIRECTORY INTERFACE

• APLIKACJE WEBOWE I ROZPROSZONE •

## Connect to the Directory Server

```
// hashtable to hold environment properties for the JNDI context
Hashtable env = new Hashtable();

// name of the service provider class
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
// protocol, hostname, port of the directory server
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
// logon info if needed - default is "none" (anonymous)
//env.put(Context.SECURITY_AUTHENTICATION, "simple");
//env.put(Context.SECURITY_PRINCIPAL, "cn=ROOT");
//env.put(Context.SECURITY_CREDENTIALS, "password");

// starting point for our directory and naming services access
// can use Context if just using the naming service
DirContext dirCtx = null;
try {
    //
    dirCtx = new InitialDirContext(env);
    // more processing...
} catch (javax.naming.NamingException ne) {ne.printStackTrace();}
```

# JAVA NAMING AND DIRECTORY INTERFACE

## • APLIKACJE WEBOWE I ROZPROSZONE •

Context and NamingException are in *javax.naming*, DirContext is in *javax.naming.directory*.

This example is accessing the directory as anonymous, so the sign-on properties are commented out. simple authentication passes a userid/password in clear text. strong allows use of encrypted userids/passwords or certificates, dependent on the directory capabilities.

A good approach (pattern) is to use a singleton for retrieval of the DirContext. Then once the reference is obtained, it can be easily retrieved for subsequent processing.

# JAVA NAMING AND DIRECTORY INTERFACE

## • APLIKACJE WEBOWE I ROZPROSZONE •

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.naming.ldap.InitialLdapContext;
import javax.naming.ldap.LdapContext;

public class ActiveDirectory {
    public static final String  LDAP_HOST_1 = "ni.wi.tuniv.szczecin.pl";
    public static final String  LDAP_HOST_2 = "eta.wi.tuniv.szczecin.pl";
    public static final int     LDAP_PORT = 389;
    public static final String  DOMAIN = "WIPSAD";

    public static LdapContext authenticate(String username, String password,
                                           String domain, String ldapHost, int port) throws NamingException {
        Hashtable<String, String> env = new Hashtable<>();
        env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
        env.put(Context.SECURITY_AUTHENTICATION, "simple");
        env.put(Context.SECURITY_PRINCIPAL, domain + "\\\" + username);
        env.put(Context.SECURITY_CREDENTIALS, password);

        String ldapURL = "ldap://" + ldapHost;
        ldapURL += (port == 0) ? "/" : ( ":" + port + "/" );
        env.put(Context.PROVIDER_URL, ldapURL);

        return new InitialLdapContext(env, null);
    }

    public static void main(String...args) throws NamingException {
        LdapContext ldapContext = authenticate( "kkraska", "*****", DOMAIN, LDAP_HOST_1, LDAP_PORT );
        ldapContext.close();
    }
}
```



# KONIEC

• APLIKACJE WEBOWE I ROZPROSZONE •



**DZIĘKUJĘ  
ZA UWAGĘ!!!**

---

Szczecin, 29 marca 2020 r.