ID2212 Network Programming with Java
Lecture 6

Distributed Objects. Java IDL (CORBA) and Java RMI

Leif Lindbäck, Vladimir Vlassov
KTH/ICT/SCS

HT 2015

Outline

- **Revisited: Distributed Computing**
  - **Architectures**
  - **Implementation Approaches**
- **Basics of a Distributed Object Architecture**
- **Java IDL (CORBA)**
- **Java RMI: Remote Method Invocation**

Review: Architectures of Distributed Applications

- Two-tier architecture: Clients and Servers
- Three-tier architecture:
  - First tier: clients with GUI
  - Middle tier: business logic
  - Third tier: System services (databases)
- Peer-to-peer architecture: Equal peers

Existing Implementation Approaches

- **Message passing via sockets**
- **RPC: Remote Procedure Calls**
- **Distributed objects (RMI)**
  - **DCOM: Distributed Component Object Model (Microsoft, homogeneous implementation)**
  - **CORBA: Common Object Request Broker Architecture (OMG, heterogeneous)**
  - **Java RMI (Oracle, homogeneous)**
  - **Enterprise Java Beans (EJB) – Distributed component architecture for building integrated enterprise services**

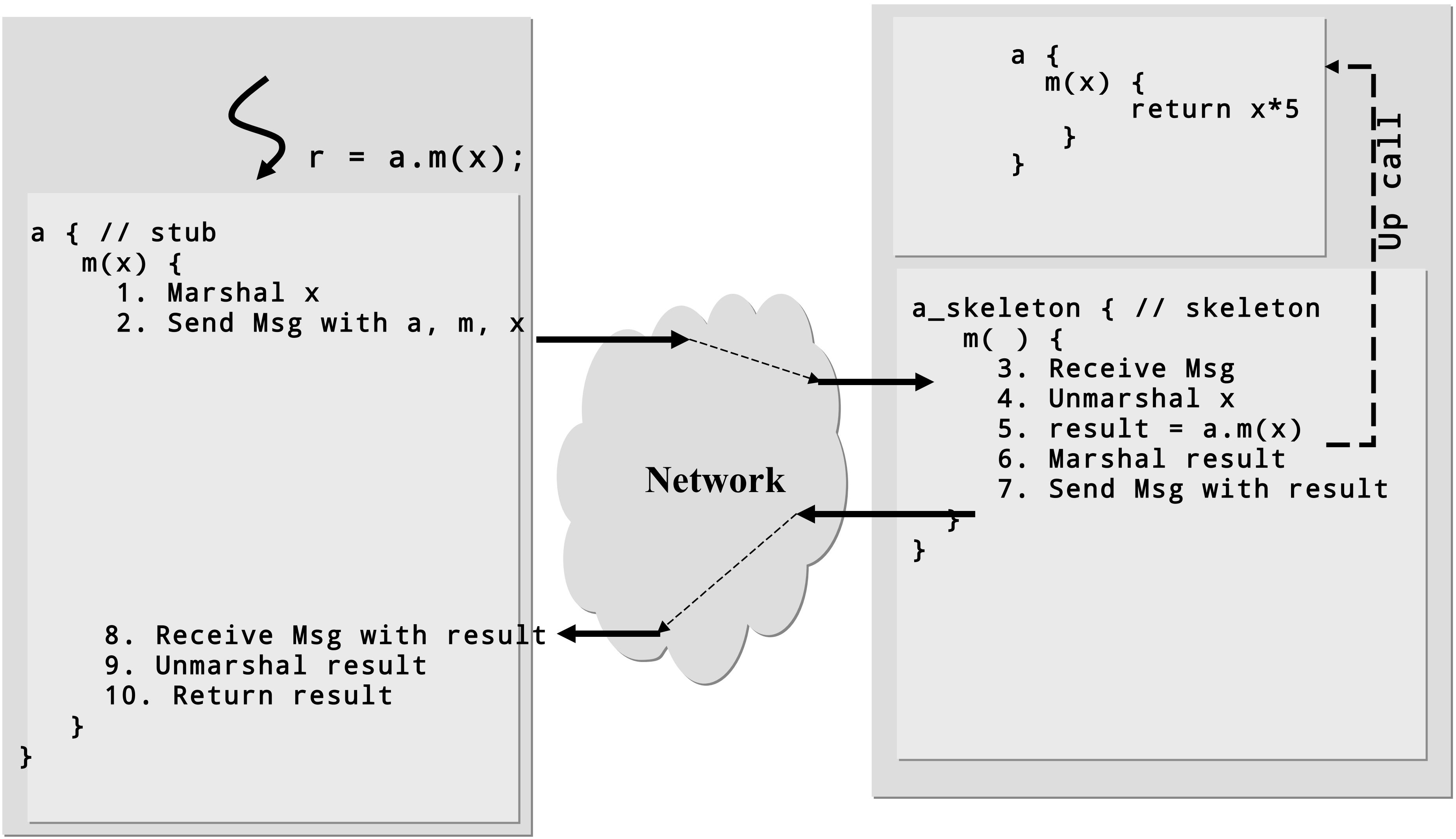
Motivation for RPC and RMI

- **Message passing over socket connections is somewhat low level for distributed applications**
  - **Typically, client/server interaction is based on a request/response protocol**
  - **Requests are typically mapped to procedures or method invocations on objects located on the server**
- **A better approach for client/server applications is to use**
  - **Remote Procedure Calls**
    - **Rendezvous (like in ADA, Concurrent C)**
  - **Remote Method Invocation – in OO environment**

Remote Method Invocation (RMI)

- *Remote method invocation* (RMI) is the mechanism to invoke a method in a remote object
  - the object-oriented analog of RPC in a distributed OO environment, e.g. OMG CORBA, Java RMI, DCOM
  - RPC allows calling procedures over a network
  - RMI invokes object's methods over a network
- Location transparency: invoke a method on a stub like on a local object (via stack)
- Location awareness: the stub makes remote call across a network and returns a result via stack

Remote Method Invocation



Parameter Passing

- Parameters are passed in an RMI message and not via a local stack
  - data of primitive types are passed by values
  - objects are passed either by values (replication) or by references
- Objects can be heterogeneous
  - different implementation languages
  - different target virtual machines and operation systems
- Different representations of primitive types
  - convert data representation across different implementation
- Composite Types (e.g., structures, objects)
  - need to be flattened and reconstructed (marshal / unmarshal)

Marshaling/Unmarshaling

- **Marshaling:**
  - done by client (i.e., caller)
  - packing the parameters into a message
  - flatten structures
  - perform representation conversions if necessary
  - also done by server (i.e., callee) for results
- **Unmarshaling:**
  - done by receiver of message to extract parameters or results

Stubs and Skeletons

- **Encapsulate marshaling and communication**
  - Enable application code in both client and server to treat call as local
- ***Stub*** **is a proxy for the real object on the client**
  - represents the real object as a local object on the client
  - contains information to locate the real object
  - implements original interface with the same method signatures but the methods perform remote calls to the real object
- ***Skeleton*** **is on the server**
  - receives, unmarshals parameters
  - calls original routine on the real object
  - marshals and sends result (data, acknowledgment or exception) to the client

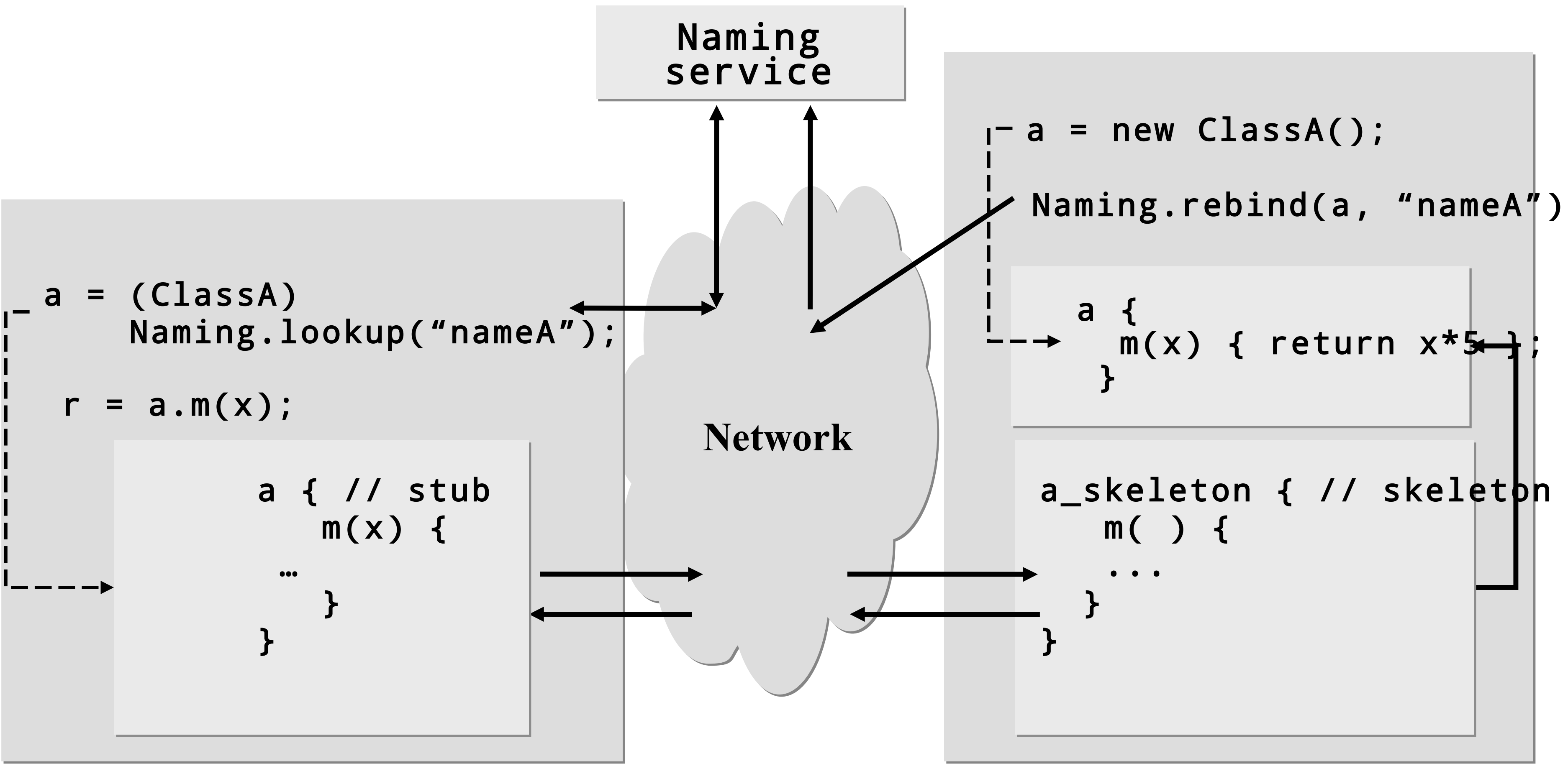
Synchronous versus Asynchronous Invocation

- **Void methods** do not require a result to be sent to the caller
- **Asynchronous invocation**
  - The method locally invoked on the stub immediately returns and the calling thread proceeds as soon as the request is on its way to the remote object
  - The request is executed by the underlying layer in a separate thread
  - Problem: exceptions
- **Synchronous invocation**
  - The calling thread is suspended waiting for the remote invocation to complete (for the invoked method to return)
  - The calling thread proceeds as soon as it gets acknowledgement from the remote object

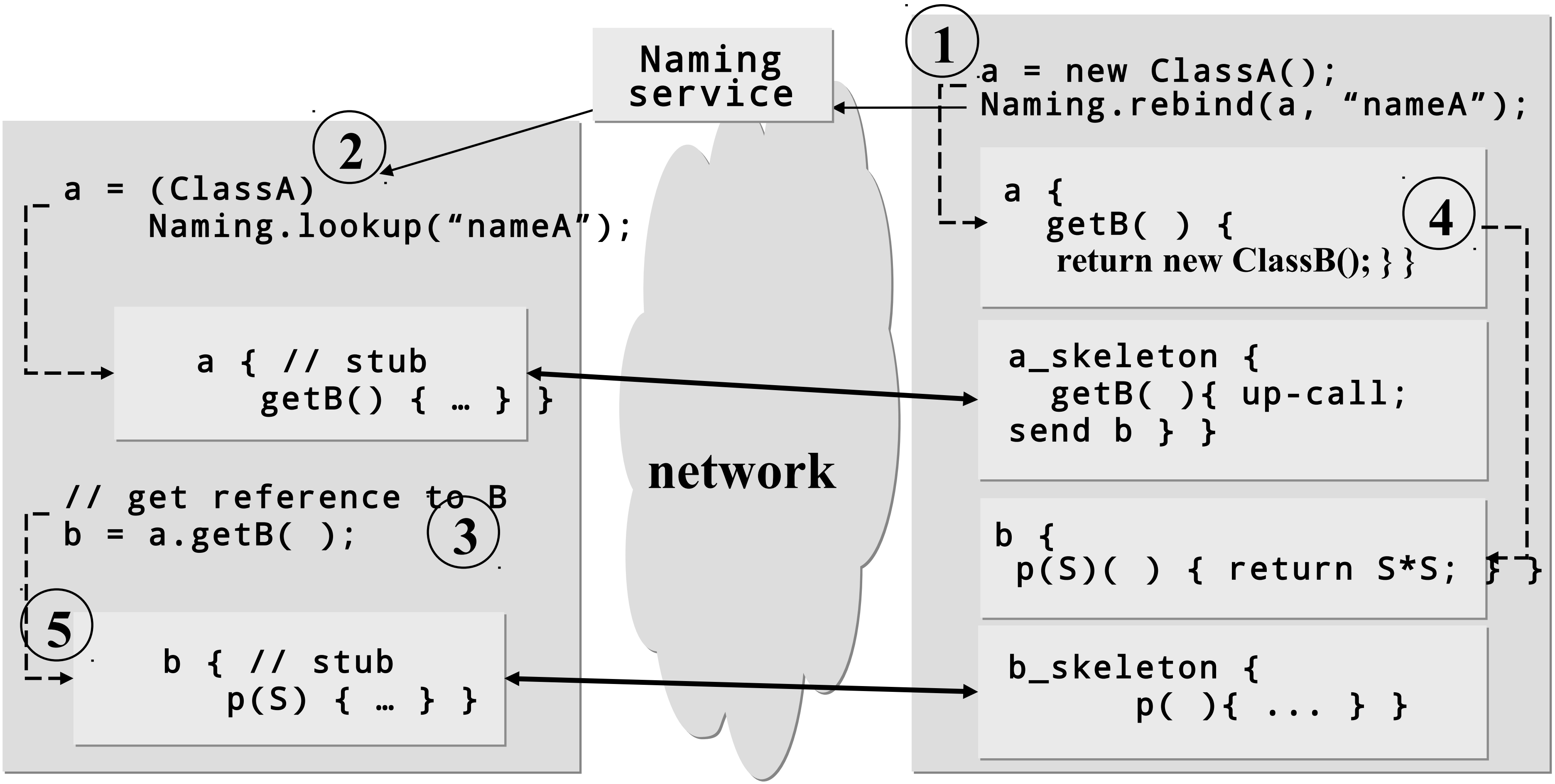
Locating Objects

- **How does the caller get a reference to the remote object, i.e. stub?**
- **One approach is to use a distributed *Naming Service*:**
  - **Associate a unique name with a remote object and bind the name to the object at the Naming Service.**
    - **The name must be unique in current context.**
    - **The record typically includes name, class name, object reference**
    - **The object reference contains location information.**
  - **The object name is used by the client to lookup the Naming Service for the object reference (stub).**
  - **Problem of the primary reference: How does the client locate the Naming Service? – configuration issues**
- **Another way to get a reference to a remote object is to get it as *a parameter or a return* in remote method invocation**
- **Third way: to make a reference (*IOR: Interoperable Object Reference*) and store/send it in a file**

Use of the Naming Service



Remote Reference in Return



Separate Interface from Implementation. Interface Definition Language (IDL)

- **A remote object is remotely accessed via its remote interfaces.**
- **Objects can be heterogeneous**
  - **different implementation languages**
  - **different target virtual machines and operation systems**
- **Separate interface definition from implementation:**
  - **Implementation may change, as long as the interface is respected**
- ***Interface Definition Language (IDL)***
  - **Describe interface for RMI (when using CORBA)**

Generating Stubs and Skeleton. IDL Mappings

- **Where do Stubs and Skeletons come from?**
  - writing (un)marshaling code is bugprone
  - communication code has many details
  - structure of code is very mechanical
- **Answer:**
  - Stubs and Skeletons can be generated from IDL definitions
- **Mapping from IDL to OO-language**
  - generates code for Stubs and Skeletons
  - IDL to Java, C++, Smalltalk, COBOL, Ada
  - Allows cross language invocations

Java IDL (CORBA)

Reference implementation of OMG CORBA in Java for Java

`org.omg.CORBA`

Four Components of OMA (Object Management Architecture)

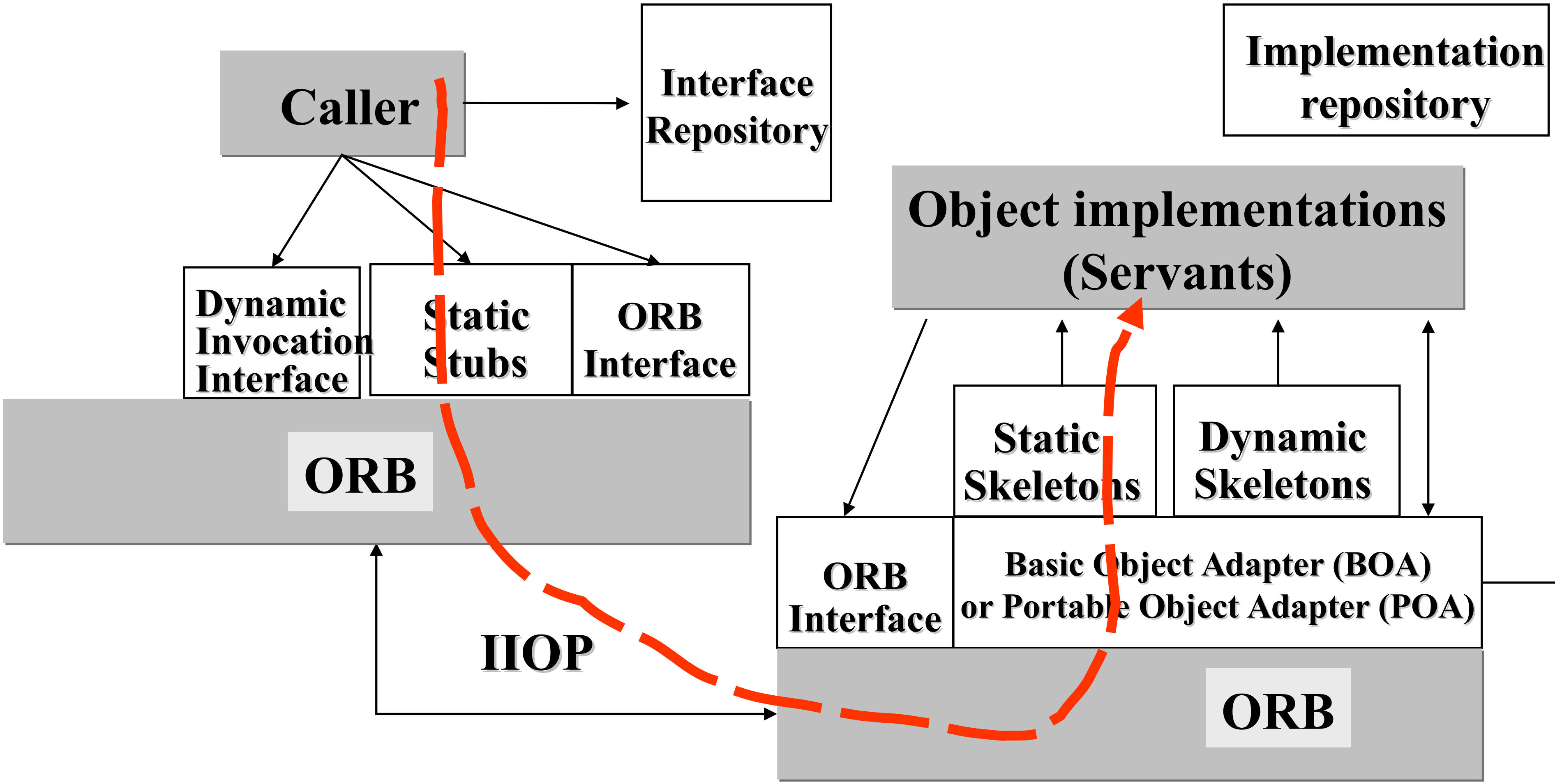
- By the Object Management Group (OMG) consortium that operates since 1989. See: http://www.omg.org

1. *Object Model* (Glossary of terms)
   - Concepts: class, object, attribute, method, inheritance, etc.
   - UML (Unified Modeling Language) is a standard for object modeling.
2. *CORBA* (Common Object Request Broker Architecture)
   - A mechanism for communication between objects
   - Specification, related APIs and tools
   - Object Request Broker (ORB) is implementation of CORBA

Four Components of OMA (cont)

3. ***CORBA Services***
   – Horizontal services common for any objects: Naming, Security, Life Cycle, Transactions, Events, etc.
4. ***CORBA Facilities***
   – High level functionality for integrating objects
     • User interface: drag-and-drop, compound documents
     • System Management
     • Task Management / Workflow
   – Vertical services supporting particular industries
     • Finance, Oil and Gas Exploration, Telecommunications (TMN/TINA-C) , 10 other
       – TMN is Telecommunications Management Network;
       – TINA-C is Telecommunications Information Networking Architecture Consortium

The Architecture of CORBA



CORBA Anatomy

- ***ORB: Object Request Broker***
  - makes it possible for CORBA objects to communicate with each other by connecting objects making requests (clients) with objects servicing requests (servants).
- ***BOA: Basic Object Adapter***
  - accepts call requests (as a meta-call),
  - instantiates objects,
  - initiates up-calls on skeletons,
  - manages the Implementation Repository,
  - different ORB vendors have completely different implementation of BOA
- ***POA: Portable Object Adapter***
  - like BOA but portable between different ORB products

(cont'd) CORBA Anatomy

- **A *stub* on the client side provides a static interface to remote object services.**
  - **resolves the remote object's location**
  - **performs remote method invocation via a local ORB**
    - **Sends the object reference, the method name and parameters to the destination ORB (skeleton) by using IIOP (Internet Inter-ORB Protocol)**
    - **Receives and unmarshals data in return**
- **A *skeleton* on the server side performs up-calls on a real object**
  - **transforms the call and parameters into the required format and calls the object**
  - **marshals result (or exception) and sends it over ORB connection.**

ORB Protocols

- **CORBA 2.0 defines standard protocols:**
- **GIOP: General Inter ORB Protocol**
  - **Defines standard message format**
- ***IIOP: Internet Inter ORB Protocol***
  - **IIOP is the implementation of GIOP over TCP/IP**
  - **IIOP-to-HTTP gateway and HTTP-to-IIOP gateway allow CORBA clients to access Web resources and Web clients to access CORBA resources.**
- **ESIOP: Environment Specific Inter ORB Protocol**
  - **Allows ORBs to run on top of other standards (such as DCE: Distributed Computing Environment consisting of standard APIs: naming, DFS, RPC, etc.)**

IDL: Interface Definition Language

- *IDL* is a purely declarative language: interface declarations
- An IDL interface describes the attributes and methods (operations) that are exported on the ORB.
  - An interface can have several implementations.
  - An object can implement several interfaces.
- IDL-to-language compilers are based on mapping from IDL to the language (Java, C++, Smalltalk, COBOL, Ada)
- A compiler generates
  - An interface(s),
  - A stub (a client proxy for remote calls),
  - A skeleton (a server proxy for translating incoming calls to up-calls)

IDL Concepts

- ***Interface***
  - Similar to a class, but only defines the interface of an object, without information on its representation in memory
- ***Operation***
  - Similar to a method or member function
  - The direction of parameter must be specified: in, out, inout
- ***Attribute***
  - Does not define an attribute in memory
  - Defines two operations for getting and setting the value
  - readonly is used to suppress the function setting the value

Basic Data Types

- No int type
- No pointer type
- IDL types are defined in terms of their semantics

IDL	Java
`short`	`short`
`long`	`int`
`unsigned short`	`short`
`unsigned long`	`int`
`float`	`float`
`double`	`double`
`char`	`char`
`boolean`	`boolean`
`octet`	`byte`
`any`	`class any`
`string`	`String`

Complex Types

- **Build complex types from basic types in IDL:**
  - `struct`, `enum`, `union`, `typedef`
  - `array` – **fixed length collection**
  - `sequence` – **variable length collection**
  - `Object` – **reference to an IDL object (proxy)**
- **Mapping to Java**
  - `sequence` **and** `array` **are mapped to the Java** `array` **type.**
  - `enum`, `struct`, **and** `union` **are mapped to a final Java class that implements the semantics of the IDL type.**
  - **For example, array of bytes can be defined as:**

```
typedef sequence <octet> bytes;
bytes getBytes(in string from) raises(cannotget);
```

- **The Java class generated should have the same name as the original IDL type.**

Passing Parameters and Returns

- CORBA sends all types across the network by value, except objects
  - Objects are passed by reference
  - A proxy is constructed on the receiving end
- The OMG added a new specification called "Pass-by-Value"
  - Include Object by Value mapping
  - Initiators were Sun and IBM
  - Motivation: support for object migration and replication
  - RMI over IIOP

Java IDL (`org.omg.CORBA`)

- **Java IDL is a reference implementation of CORBA in Java**
- **Oracle delivers Java IDL in the JDK**
  - IDL-to-Java compiler
  - Multi-protocol ORB (classes)
  - Support for Java clients and servers (Name service, etc.)
- **Java IDL is not a sophisticated product on the server side:**
  - Doesn't have CORBA scalability and security features
  - No CORBA Services except of Naming
- **Java IDL will be useful on the client**
  - Avoid downloading the ORB client

Other Implementations of CORBA

- **CORBA platforms from Progress Software**
  - **http://web.progress.com/en/Product-Capabilities/corba.html**
- **The Micro Focus's solution for CORBA Technology (VisiBroker)**
  - **http://www.microfocus.com/products/visibroker/index.aspx**
- **CORBA typically comes as a part of an enterprise (application) server**

Developing a Distributed Application with Java IDL

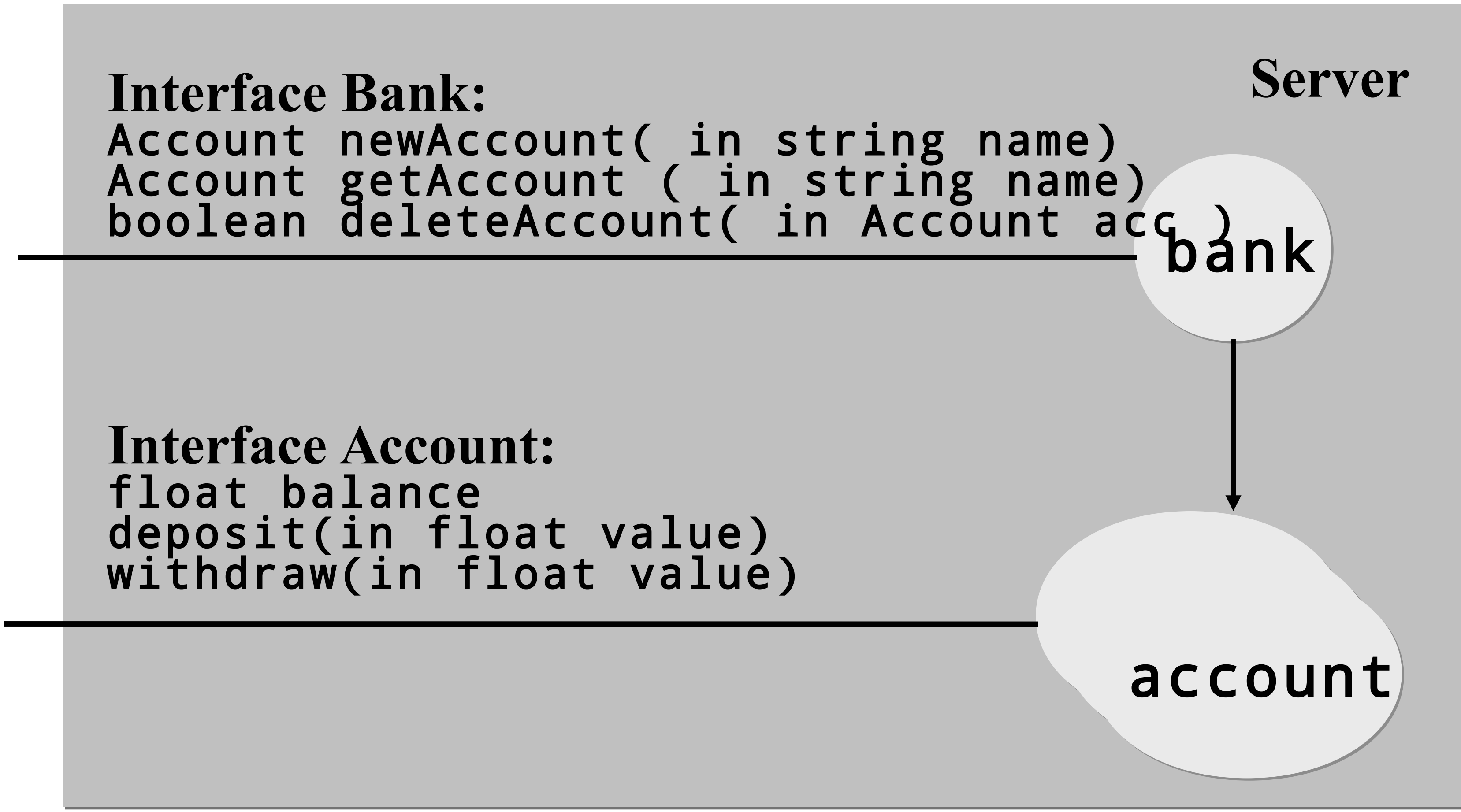
1. **Define interfaces with IDL**
2. **Compile the interfaces using `idlj`, which generates the Java bindings for a given IDL file.**
3. **Develop an implementation for the interfaces (servants)**
4. **Develop a server (a container for servants) that initializes ORB and creates the servants**
5. **Develop a client**
6. **Compile the client, the servants and the server (using `javac`)**
7. **Start the Naming Service `tnameserv`, which is the Common Object Services (COS) Name Service**
8. **Start the server**
9. **Run the client**

Step 1. Sample IDL Interfaces

```
module bankidl {
    interface Account {
     readonly attribute float balance;
     exception rejected { string reason; };
     void deposit(in float value) raises (rejected);
     void withdraw(in float value) raises (rejected);
    };

    interface Bank {
     exception rejected { string reason; };
     Account newAccount( in string name) raises
                                          (rejected);
     Account getAccount ( in string name);
     boolean deleteAccount( in string acc );
    };
};
```

The IDL Interfaces (cont'd)



Step 2. Compiling IDL to Java

- The IDL to Java compiler (`idlj`) generates:
  - Interfaces:
    - `Bank.java`, `Account.java`
  - Stubs for the client side:
    - `_BankStub.java`,`_AccountStub.java`
  - Skeletons for the server side:
    - When using BOA (backwards compatible to Java SE 1.4) `BankImplBase.java`,`AccountImplBase.java`
    - When using POA: `BankPOA.java, AccountPOA.java`
    - Implementations of the interfaces should extend the skeletons.
  - Helpers used to narrow a remote reference to its remote interface:
    - `BankHelper`, `AccountHelper`

Step 3. Implementing The Interfaces.

- A servant is a class that implements the interface(s) generated by a IDL to Java compiler.
- The servant class may extend an appropriate skeleton (implementation base) class, for example:

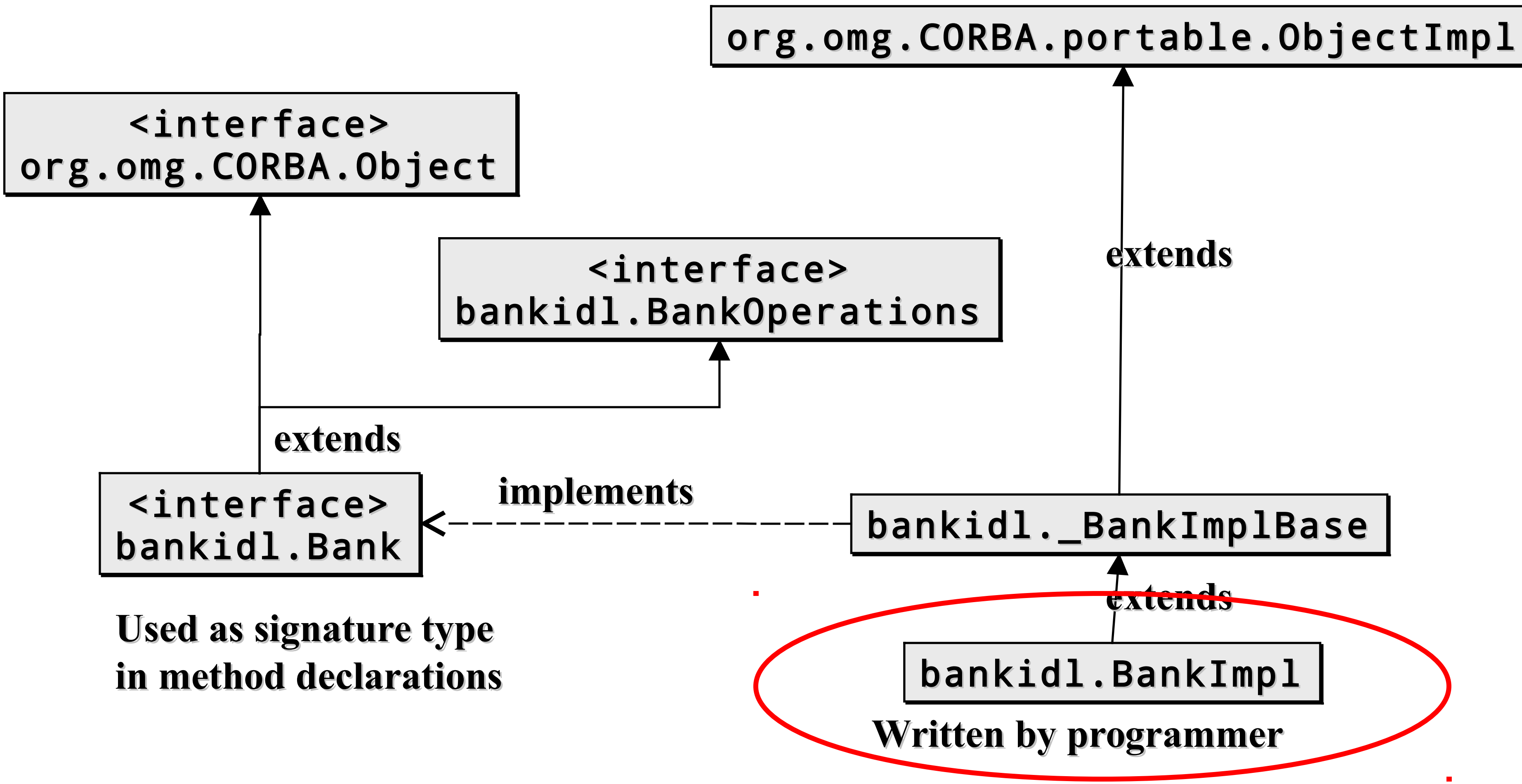
```
public class BankImpl extends
_BankImplBase
```

or (when using POA)

```
public class BankImpl extends BankPOA
```

  – In this way the servant implements the interface and encapsulates the skeleton that accepts (remote) calls

Inheritance Structure



Step 2. Bank Implementation

```
public class BankImpl extends _BankImplBase {
  private String bankname = null;
  private Hashtable accounts = new Hashtable();
  public BankImpl(String name) {
    super();
    bankname = name;
  }
  public Account newAccount(String name) throws rejected {
    AccountImpl account = (AccountImpl) accounts.get(name);
    if (account != null) {
      throw new rejected("Rejected: Account for: "
                         + name + " already exists");
    }
    account = new AccountImpl(name);
    accounts.put(name, account);
    return (Account)account;
  }
  public Account getAccount(java.lang.String name) {
    return (Account) accounts.get(name);
  }
  public boolean deleteAccount(String name) {
    AccountImpl account = (AccountImpl) accounts.get(name);
    if (account == null) {
      return false;
    }
    accounts.remove(name);
    return true;
  }
}
```

Step 3. Account Implementation

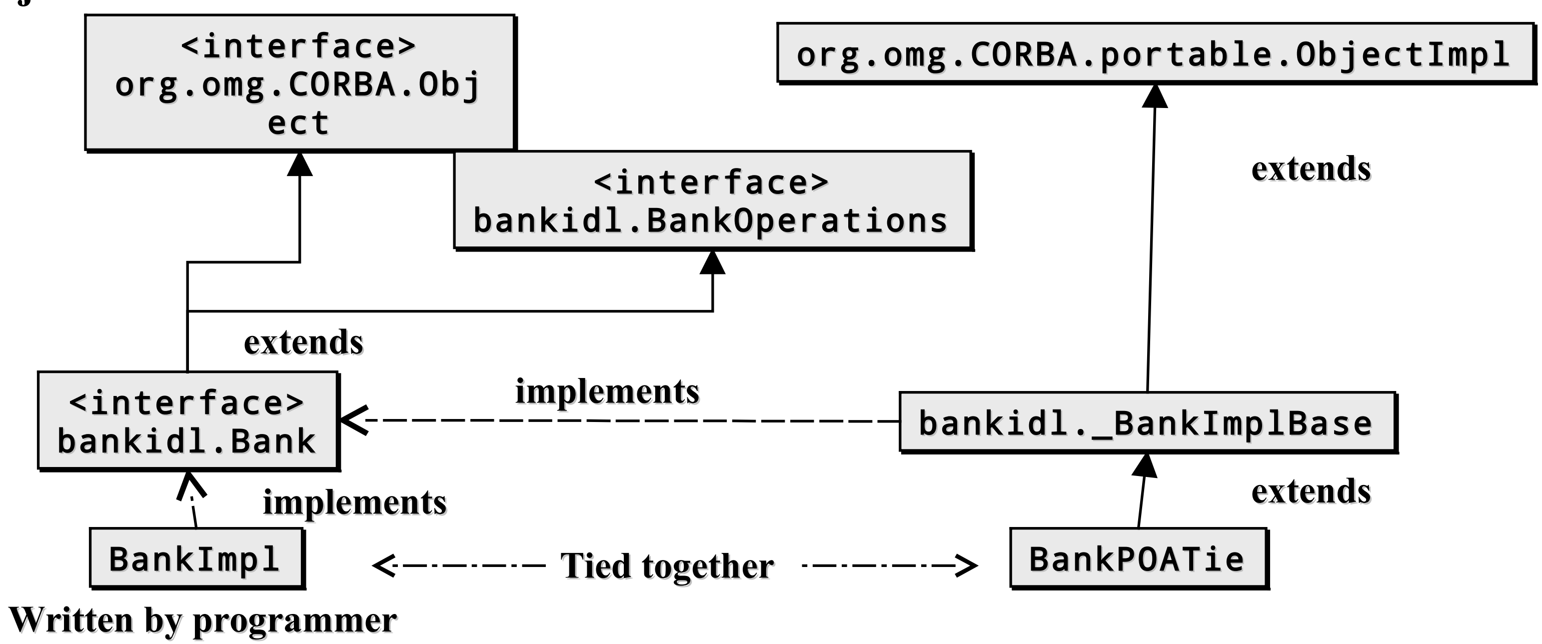
```
package bankidl;
import bankidl.AccountPackage.rejected;
public class AccountImpl extends _AccountImplBase {
  private float balance = 0;
  private String name = null;

  public AccountImpl(java.lang.String name) {
    super();
    this.name = name;
  }
  public void deposit(float value) throws rejected {
    if (value < 0) {
      throw new rejected("Rejected: Illegal value: ” +
                         value);
    }
    balance += value;
  }
  public void withdraw(float value) throws rejected {
    if (value < 0) {
      throw new rejected("Rejected:  Illegal value: ” +
                         Value);
    }
    if ((balance - value) < 0) {
      throw new rejected("Rejected: Negative balance”));
    }
    balance -= value;
  }
  public float balance() {
    return balance;
   }
}
```

Le

Inheritance Structure with POATie. The Tie Delegation Model.

- An IDL to Java compiler can generate a `<interface>POATie` class that extends the skeleton.
- The implementation class may inherit from a different class and implement the remote interface.
- Remote calls received by the tie object are directed to the implementation object.



Design Options (1/2)

- Choose an ORB implementation that suits you (price, efficiency, etc.)
- Use either POA (standard Portable Object Adapter) or BOA (non-standard Basic Object Adapter, which could be more efficient)
  1. To generate both client and server-side POA bindings, use
     `idlj -fall My.idl`
     - Generates `MyPOA.java` given an interface `My` defined in `My.idl`.
     - You must implement `My` in a class that must inherit from `MyPOA`.
  2. To generate BOA bindings backwards compatible to JDK 1.4, use
     `idlj -fall -oldImplBase My.idl`
     - Generates `_MyImplBase.java` given an interface `My` defined in My.idl.
     - You must implement `My` in a class that must inherit from `_MyImplBase`.

Design Options (2/2)

Use a tie class when it is not convenient or possible to have your implementation class inherit from either of the skeletons `MyPOA` or `_MyImplBase`.

```
idlj -fallTIE My.idl
```

- **Generates the tie class**

– **Wrap your implementation within `My_Tie`.**

– **For example:**

```
MyImpl myImpl = new MyImpl ();
My_Tie tie = new My_Tie (myImpl);
orb.connect (tie);
```

Step 4. Server

(Using BOA, Backwards compatible to JDK 1.4)

```
package bankidl;
import org.omg.*;
import org.omg.CORBA.ORB;
public class Server {
  public static void main(String args[]) {
    if (args.length != 3) {
      System.out.println(
        "usage: java Server <bankname> <-ORBInitialPort port>");
      System.exit(1);
    }
    try {
      ORB orb = ORB.init(args, null);
      BankImpl bankRef = new BankImpl(args[0]);
      orb.connect(bankRef);
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references( "NameService" );
      NamingContext ncRef = NamingContextHelper.narrow(objRef);
      NameComponent nc = new NameComponent(args[0], "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, bankRef);
      orb.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

Step 5. Client

```
package bankidl;
import org.omg.CosNaming.*;
import org.omg.CORBA.ORB;
public class SClient {
  static final String USAGE = "java bankidl.SClient <bank> “ +
                               “<client> <value> “ +
                               “<-ORBInitialPort port>";
  Account account;
  Bank bankobj;
  String bankname = "SEB";
  String clientname = "Vladimir Vlassov";
  float value = 100;

  public static void main(String[] args) {
    if ((args.length > 0) && args[0].equals("-h")) {
      System.out.println(USAGE);
      System.exit(0);
    }
    new SClient(args).run();
  }
```

Step 5. Client (cont'd)

```
public SClient(String[] args) {
  if (args.length > 2) {
    try {
      value = (new Float(args[2])).floatValue();
    } catch (NumberFormatException e) {
      System.out.println(USAGE);
      System.exit(0);
    }
  }
  if (args.length > 1)  clientname = args[1];
  if (args.length > 0)  bankname = args[0];
  try {
    ORB orb = ORB.init(args, null);
    org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);
    NameComponent nc = new NameComponent(bankname, "");
    NameComponent[] path = {nc};
    bankobj = BankHelper.narrow(ncRef.resolve(path));
  } catch (Exception se) {
    System.out.println("The runtime failed: " + se);
    System.exit(0);
  }
  System.out.println("Connected to bank: " + bankname);
}
```

Step 5. Client (cont'd)

```
   public void run() {
     try {
       account = bankobj.getAccount(clientname);
       if (account == null) {
         account = bankobj.newAccount(clientname);
       }
       account.deposit(value);
       System.out.println(clientname + "'s account: $" +
                                  account.balance());
     } catch (org.omg.CORBA.SystemException se) {
       System.out.println("The runtime failed: " + se);
       System.exit(0);
     } catch (bankidl.AccountPackage.rejected e) {
       System.out.println(e.reason);
       System.exit(0);
     } catch (bankidl.BankPackage.rejected e) {
       System.out.println(e.reason);
       System.exit(0);
     }
   }
 }
```

Locating Objects

- Using Name Service
  - The server creates the Bank object with the specified name, e.g. “Nordea”, and makes it persistent (ready).
  - To obtain the object reference, the client via the ORB contacts the Name Service of Java IDL, which is started with the following command:

```
tnameserv -ORBInitialPort 1050
```

- Using Interoperable Object References (IOR)
  - Server can store an object’s IOR (Interoperable Object Reference) as a string to a file.
  - Client can then fetch the reference from the file via a web server.

Server Using IOR

```
package bankidl;
import org.omg.CORBA.ORB;
import java.io.*;
public class Serverl {
  public static final String USAGE =
            "usage: java bankidl.Serverl bankname dir";
  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println(USAGE); System.exit(1);
    }
    try {
      ORB orb = ORB.init(args, null);
      BankImpl bankRef = new BankImpl(args[0]);
      orb.connect(bankRef);
      File dir = new File(args[1]);
      if (!dir.exists()) {
        dir.mkdir();
      }
      String filename = dir + Character.toString(File.separatorChar) +
                        args[0] + ".ior";
      File file = new File(filename);
      file.createNewFile();
      file.deleteOnExit();
      FileWriter writer = new FileWriter(file);
      writer.write(orb.object_to_string(bankRef));
      writer.close();
      orb.run();
    } catch (Exception e) {
      System.out.println(USAGE); System.exit(1);
    }
  }
}
```

Client Using IOR

```
public class Clientl {
  static final String USAGE =
                "java bankidl.Client url <-ORBInitialPort port>";
  Bank bankobj;
  String bankname = "SEB";
  public static void main(String[] args) {
    if ((args.length > 0) && args[0].equals("-h")) {
      System.out.println(USAGE); System.exit(0);
    }
    new Clientl(args).run();
  }
  public Clientl(String[] args) {
    if (args.length < 1) {
      System.out.println(USAGE); System.exit(1);
    }
    try {
      URL bankURL = new URL(args[0]);
      BufferedReader in = new BufferedReader(
                          new InputStreamReader(
                          (InputStream)bankURL.getContent()));
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
                   orb.string_to_object(in.readLine());
      bankobj = BankHelper.narrow(objRef);
    } catch (Exception se) {
      System.out.println("The runtime failed: " + se);
      System.exit(0);
    }
    System.out.println("Connected to bank: " + bankname);
  }
```

Java RMI (Remote Method Invocation)

`java.rmi`

Java RMI

- Java RMI is a Java native ORB (object request broker)
- The Java RMI facility allows applications or applets running on different JVMs, to interact with each other by invoking remote methods.
  - Remote reference (stub) is treated as local object.
  - Method invocation on the reference causes the method to be executed on the remote JVM.
  - Serialized arguments and return values are passed over network connections.
  - Uses Object streams to pass objects "by value".

Some RMI Classes and Interfaces

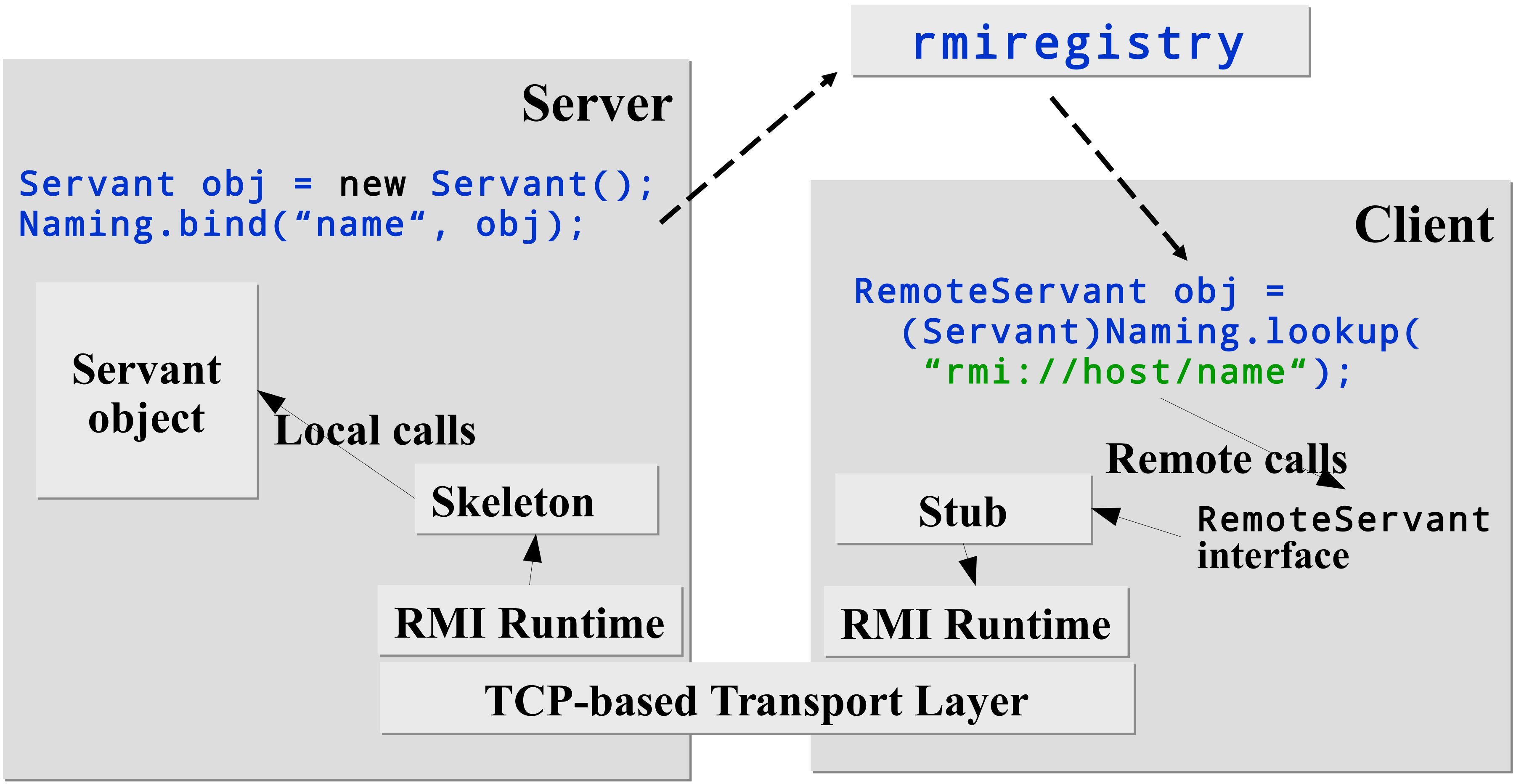
- `java.rmi.Remote`
  - Interface that indicates interfaces whose methods may be invoked from a non-local JVM -- remote interfaces.
- `java.rmi.Naming`
  - The RMI Naming Service client that is used to bind a name to an object and to lookup an object by name at the name service `rmiregistry`.
- `java.rmi.RemoteException`
  - The common superclass for a number of communication-related RMI exceptions.
- `java.rmi.server.UnicastRemoteObject`
  - A class that indicates a non-replicated remote object.
  - Exports servant to RMI runtime.
  - Handles interaction between servant and RMI runtime.

Developing a Distributed Application with Java RMI

- **Typical steps:**
  1. **Define a remote interface(s) tha extends `java.rmi.Remote`.**
  2. **Develop a class (a.k.a. servant class) that implements the interface.**
  3. **Develop a server class that provide a container for servants, i.e. creates the servants and registers them at the Naming Service.**
  4. **Develop a client class that gets a reference to a remote object(s) and calls its remote methods.**
  5. **Compile all classes and interfaces using `javac`.**
  6. **(*optional*) Generate stub classes for classes with Remote interfaces using `rmic`**

     Since JDK 1.5, stubs are generated dynamically.
  7. **Start the Naming service `rmiregistry`**
  8. **Start the server on a server host, and run the client on a client host.**

Architecture of a Client-Server Application with Java RMI



Declaring and Implementing a Remote Interface (1/2)

- A remote interface must extend the `java.rmi.Remote`
  - Each method must throw `java.rmi.RemoteException`
- A class may implement one or several remote interface
  - The class should extend the `UnicastRemoteObject` class or must be exported via the static call `UnicastRemoteObject.exportObject(Remote obj)`

Declaring and Implementing a Remote Interface (2/2)

- An object of the class that implements the remote interface is called a ***servant***.
  - A servant is created by a server. The local RMI runtime is started when the server exports the servant.
  - The servant and the server can be encapsulated into one class (typically, a primary class).
- A ***stub*** and a ***skeleton*** are generated from a servant class by the JDK.

The Naming Service `rmiregistry`. The Naming Client `Naming`

- A Remote object can be registered with a specified name at the Naming service, `rmiregistry`, provided in J2SE.
  - A registered object can be pointed to by a URL of the form `rmi://host:port/objectName`
  - The URL indicates host/port of rmiregistry – default `localhost:1099`.
- The `Naming` class provides a static client of the RMI registry.

- A server binds a name to an object:

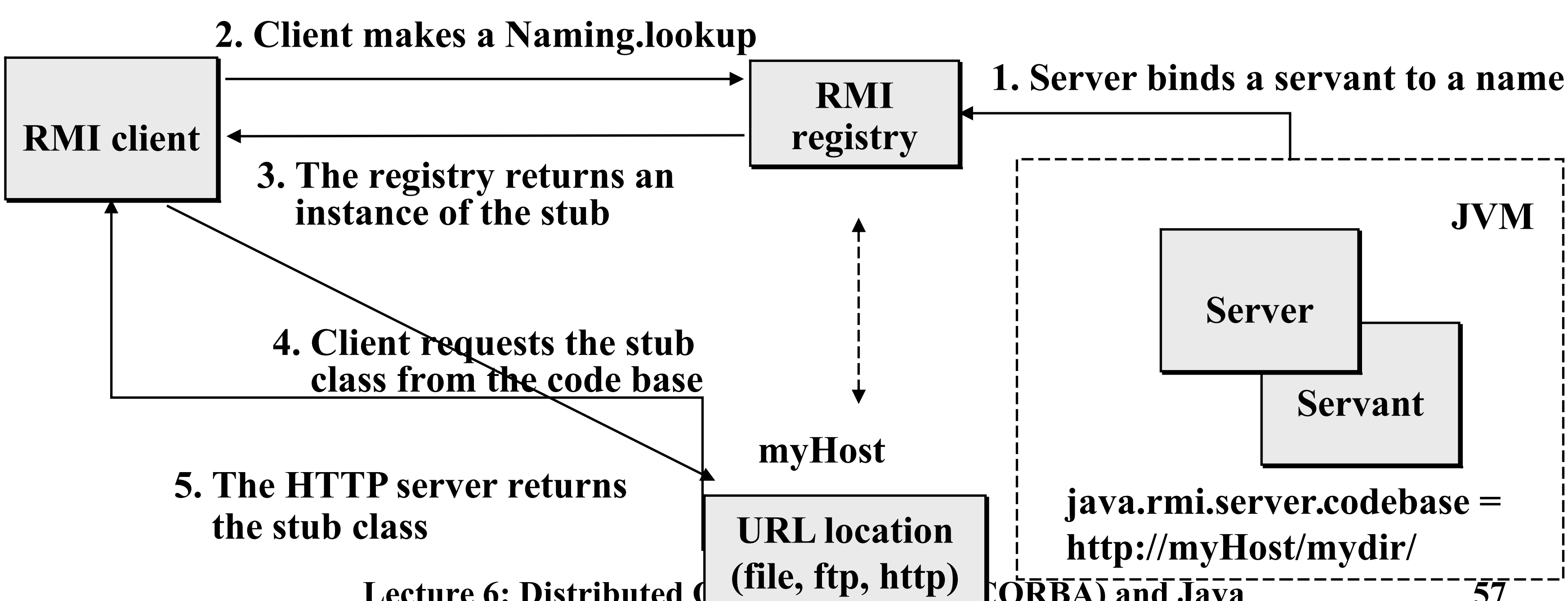
```
try {
   Bank bankobj = new
    BankImpl("CityBank");
  Naming.rebind("rmi://" + host + ":"
    + port + "/CityBank", bankobj);
  System.out.println(bankobj + " is
    ready.");
} catch (Exception e) {
    e.printStackTrace();
}
```

- A client looks up a remote reference:

```
String bankURL =
    "rmi://theHost/CityBank";
try {
 bankobj = (Bank)
    Naming.lookup(bankURL);
} catch (Exception e) {
  System.out.println("The runtime
    failed: "+ e);
  System.exit(0);
}
```

Loading Stub Classes

- Stubs are dynamically loaded when needed either from the local file system or from the network using the URL specified on server side using the `java.rmi.server.codebase` property.
  - The property can be set in a command line of an application, for example:
    - `-Djava.rmi.server.codebase=http://webvector/export/`
    - See: https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/javarmiproperties.html



Starting rmiregistry programmatically

- **Before rebind/bind**

```
try {
  LocateRegistry.getRegistry(1099).list();
} catch (RemoteException e) {
  LocateRegistry.createRegistry(1099);
}
```

Parameters and Returns in Java RMI

- Primitive data types and non-remote `Serializable` objects are passed by values.
  - If an object is passed by value, it is cloned at the receiving JVM, and its copy is no longer consistent with the original object.
  - The class name collision problem. Versioning.
- Remote objects are passed by references.
  - A remote reference can be returned from a remote method. For example:

```
try {
    // lookup for the bank at rmiregistry
    Bank bankobj = (Bank)Naming.lookup(bankname);
    // create a new account in the bank
    Account account = bankobj.newAccount(clientname);
    account.deposit(value);
} catch (Rejected e) { handle the exception }
…
```

  - A remote object reference can be passed as a parameter to a remote method.

Example: A Bank Manager

- An application that controls accounts.
- Remote interfaces:
  - `Account` – deposit, withdraw, balance;
  - `Bank` – create a new account, delete an account, get an account;
- Classes that implement the interfaces:
  - `BankImpl` – a bank servant class that implements the `Bank` interface used to create, delete accounts;
  - `AccountImpl` – a account servant class that implements the `Account` interface to access accounts.

Bank and Account Remote Interfaces

- **The Bank interface:**

```
package bankrmi;
import java.rmi.*;
import bankrmi.Account;
import bankrmi.RejectedException;
public interface Bank extends Remote {
  public Account newAccount(String name) throws RemoteException,
                                                RejectedException;
  public Account getAccount(String name) throws RemoteException;
  public boolean deleteAccount(String name) throws RemoteException;
  public String[] listAccounts() throws RemoteException;
}
```

- **The Account interface**

```
package bankrmi;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Account extends Remote {
  public float getBalance() throws RemoteException;
  public void deposit(float value) throws RemoteException,
                                          RejectedException;
  public void withdraw(float value) throws RemoteException,
                                           RejectedException;
}
```

A Fragment of the Bank Implementation

```
package bankrmi;
import java.rmi.*;
import java.util.*;
public class BankImpl extends UnicastRemoteObject implements Bank {
  private String bankName;
  private Map<String, Account> accounts = new HashMap<String, Account>();
  public BankImpl(String bankName) throws RemoteException {
    super();
    this.bankName = bankName;
  }
  public synchronized Account newAccount(String name) throws RemoteException,
                                                             RejectedException {
    AccountImpl account = (AccountImpl) accounts.get(name);
    if (account != null) {
      throw new RejectedException("Rejected: Bank: " + bankName +
                                  " Account for: " + name +
                                  " already exists: " + account);
    }
    account = new AccountImpl(name);
    accounts.put(name, account);
    return account;
  }
  public synchronized Account getAccount(String name) {
    return accounts.get(name);
  }
  public synchronized String[] listAccounts() {
    return accounts.keySet().toArray(new String[1]);
  }
  ...
}
```

The Account Implementation

```
package bankrmi;
import java.rmi.*;
public class AccountImpl extends UnicastRemoteObject implements Account {
  private float balance = 0;
  private String name;
  public AccountImpl(String name) throws RemoteException {
    super();
    this.name = name;
  }
  public synchronized void deposit(float value) throws RemoteException,
                                                       RejectedException {
    if (value < 0) {
      throw new RejectedException("Rejected: Account " + name +
                                  ": Illegal value: "+value);
    }
    balance += value;
  }
  public synchronized void withdraw(float value) throws RemoteException,
                                                        RejectedException {
    if (value < 0) {
      throw new RejectedException("Rejected: Account " + name +
                                  ": Illegal value: "+value);
    }
    if ((balance - value) < 0) {
      throw new RejectedException("Rejected: Account " + name +
                                  ": Negative balance on withdraw: " +
                                  (balance – value));
    }
    balance -= value;
  }
  public synchronized float getBalance() throws RemoteException {
    return balance;
  }
}
```

The Server

```
package bankrmi;
public class Server {
  private static final String USAGE =
                              "java bankrmi.Server <bank_rmi_url>";
  private static final String BANK = "Nordea";
  public Server(String bankName) {
    try {
      Bank bankobj = new BankImpl(bankName);
      java.rmi.Naming.rebind(bankName, bankobj);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public static void main(String[] args) {
    if (args.length > 1 || (args.length > 0 &&
        args[0].equalsIgnoreCase("-h"))) {
      System.out.println(USAGE);
      System.exit(1);
    }
    bankName = (args.length > 0) ? args[0] : BANK;
    new Server(bankName);
  }
}
```

A Fragment of a Simple Client

```
package bankrmi;
import bankrmi.*;
import java.rmi.*;
public class SClient {
  static final String USAGE = "java Client <bank_url> <client> <value>";
  String bankname = “Noname";
  String clientname = “Noname";
  float value = 100;
  public SClient(String[] args) {
    //... Read and parse command line arguments (see Usage above)
    try {
      Bank bankobj = (Bank) Naming.lookup( bankname );
      Account account = bankobj.newAccount( clientname );
      account.deposit( value );
      System.out.println (clientname + "'s account: $" + account.balance());
    } catch (Rejected e) {
      System.out.println(e); System.exit(0);
    } catch (Exception se) {
      System.out.println("The runtime failed: " + se);
      System.exit(0);
    }
  }
  public static void main(String[] args) {
    new SClient(args);
  }
}
```

Integrating Java RMI with CORBA

- **RMI is an all-Java solution**
  - A good programming model
- **CORBA is an enterprise distributed architecture**
  - A programming model not designed specifically for Java
  - A mature middleware infrastructure
- **RMI can run on top of IIOP**
  - The OMG adds a new specification called ”Pass-by-Value”
  - See: http://download.oracle.com/javase/7/docs/technotes/guides/rmi-iiop/index.html
  - Most of services in Java EE application server implementations use either RMI or RMI/IIOP for communication