

The Java Series

Introduction to Java RMI and CORBA

What are RMI and CORBA for?

Usually, in your application, once you instantiate objects, you can invoke methods on those objects.

Every object of your application runs concurrently in the same memory space and machine.

The generic goal behind RMI and CORBA is to be able to invoke methods on objects running on other machines.

An application running on machine A invokes a method on an object running on machine B.

The method invoked on the object of B, also runs on B, using thus the resources of machine B.

The application on machine A may wait for the method running to finish and eventually capture its return value

What do RMI and CORBA do?

They provide the logistics and conceptualizations so that objects can be accessed across the network.

Pose yourself the problem: **What would I have to do to be able to invoke methods on remote objects?**

There is a bunch of things to take into account:

Referencing Objects. Objects living within the same machine are referenced via memory addresses, etc. Across the network there is a need to be able to reference objects appropriately.

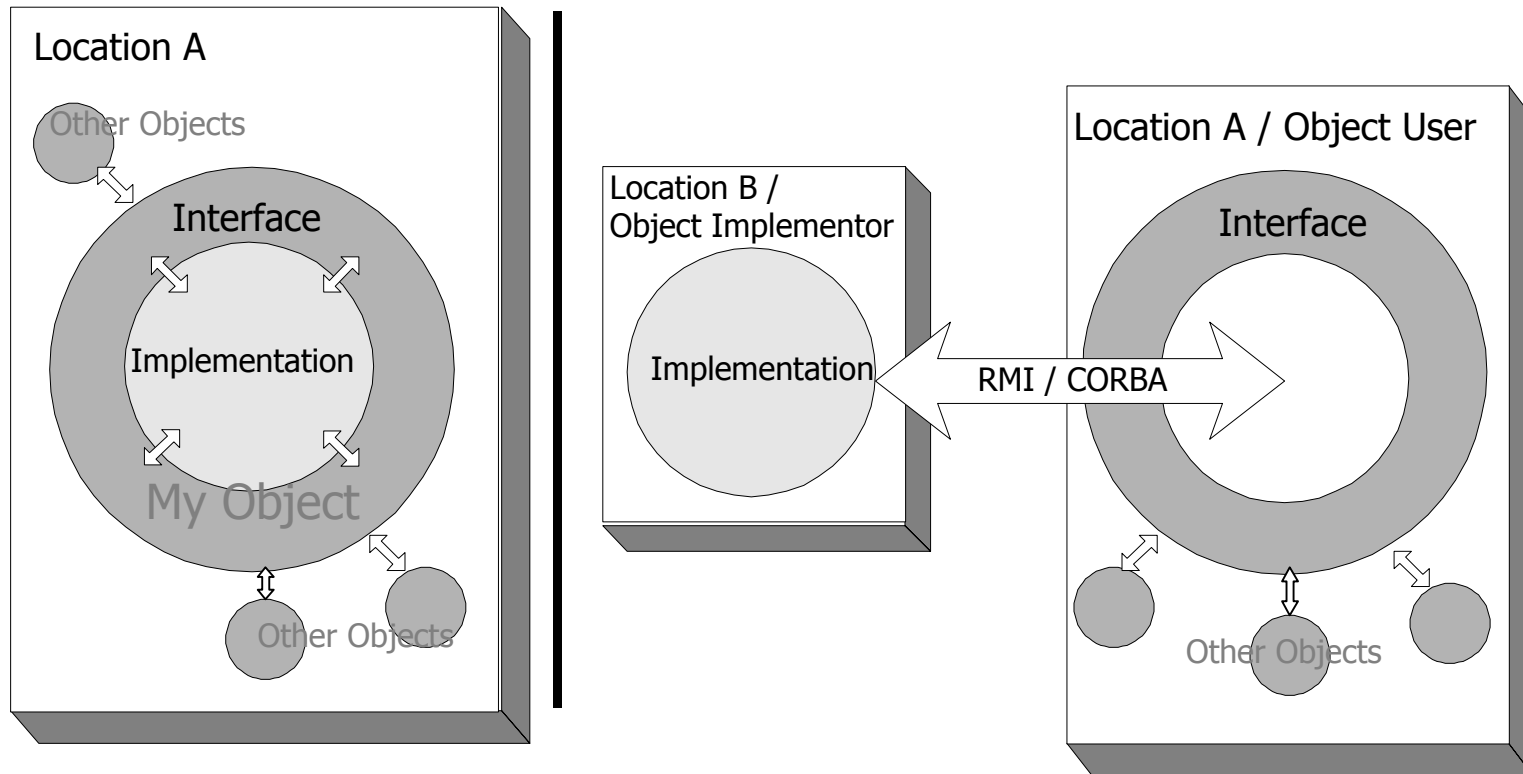
Publishing Objects. Need a way to make an object available remotely.

Network protocols to transport parameters of methods, result values and error conditions.

Languages and machines. Objects available remotely may not know who's going to be invoking methods on them.

The Main Principle

THE PHYSICAL INDEPENDENCE OF AN OBJECT'S
IMPLEMENTATION AND **INTERFACE**.
WHILE KEEPING THE LOGICAL UNIFORMITY



The Interface

The interface is the **CONTRACT** between

- The remote object implementor.

- The remote object user.

The implementor creates an object and its interface. She knows the object will be accessed **ONLY** through the interface. Then, she will give the user:

- The interface definition.

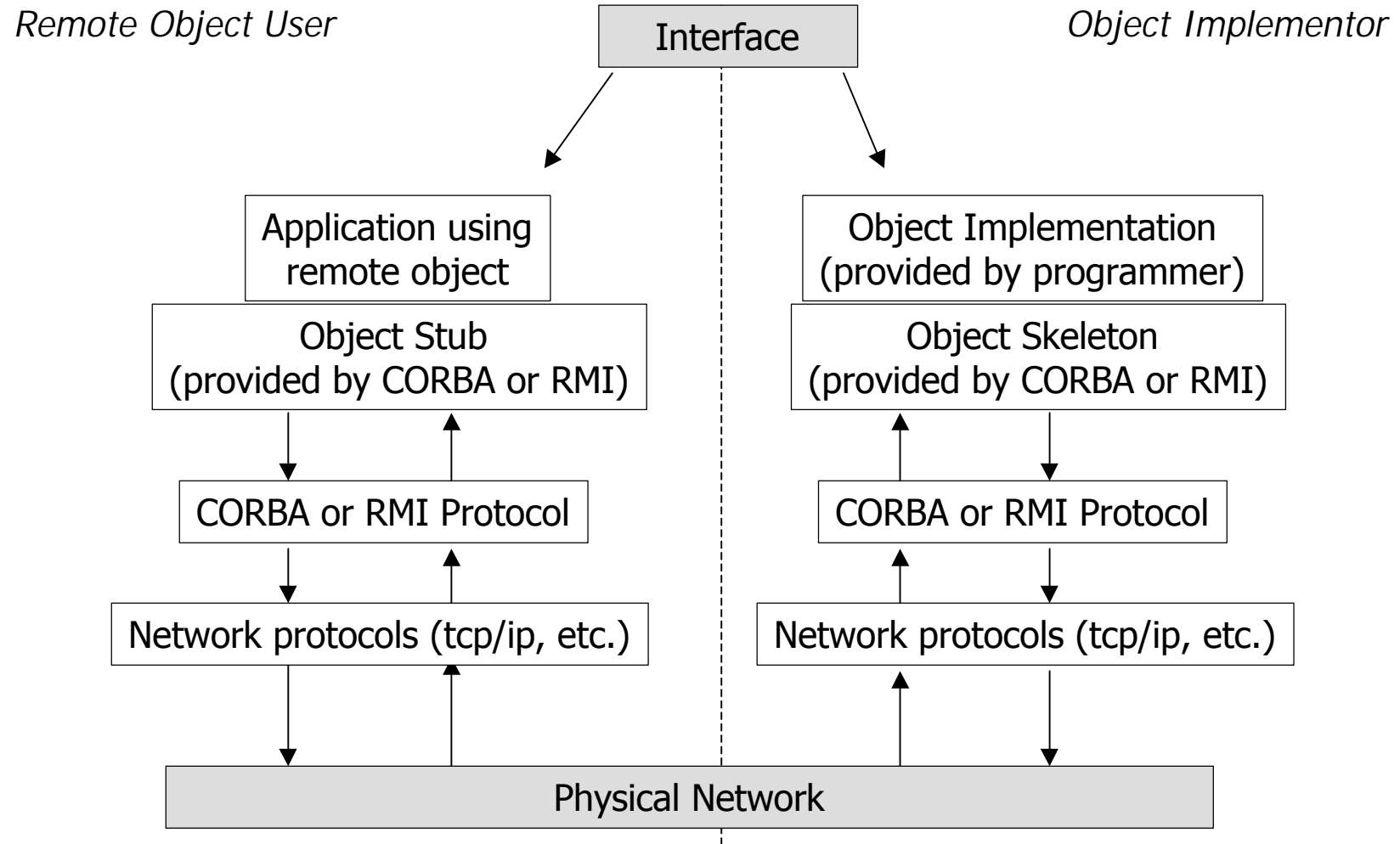
- A way to locate the object.

The user now can:

- Use the interface definition to create her application.

- Locate the object and invoke methods following the interface definition.

How does it work?



How does it work?

The CORBA or RMI libraries provide the object implementor with a local object through which her implementation is made available. This is the **OBJECT SKELETON**.

The CORBA or RMI libraries provide the object user with a **local** object which corresponds to an specific remote object. This is the **OBJECT STUB (Broker)**.

Stubs and skeletons are like representatives or brokers.

Methods are invoked in the object stub as in any other local object. The object stub knows where the remote object lives, contacts the its skeleton object and transfers the parameters of the method invocation. The skeleton in the one who actually invokes the requested method in the object implementation passing on the parameters received from the stub, retrieving the return value or exceptions and transmits them back to the stub.

The stub receives return value or exceptions and gives them to the application.

How does it work?

All network communication is handled by the stubs and skeletons. They talk using a common network protocol.

CORBA's protocol over the Internet is called **IIOP: Internet Inter ORB Protocol**.

The user of a remote object doesn't have to worry about communications, etc. Just invokes methods on the stub as on any other local object. The stub does the work.

The implementor of a remote object just sets the code to be invoked by the skeleton mechanism. Dirty work is done by the skeleton.

RMI

Remote Method Invocation. Developed by Sun. With RMI:

The remote object implementation must be made in Java.

The user application must be made in Java.

The interface definition is just a regular Java interface.

Due to Java portability, there is no assumption on the platform on which an object implementation and the user application are running.

It's designed to be a way to develop easily Java distributed applications.

It's simple and straightforward because everything is kept within Java.

CORBA

Common Object Request Broker Architecture, developed by the OMG (Object Management Group) a consortium of companies and organizations. With CORBA:

There is no assumption on the languages. The implementor and user decide **independently** what platform and language they are going to use, without needing to know what the other uses.

Since the interface is the contract it must be **language neutral**.

So, OMG created IDL (Interface Definition Language), which is used to specify language independent interfaces (contracts).

Implementors and users only share IDL definitions. From an IDL definition they generate code in the language of their choice.

CORBA also provides other functionalities (CORBA Services):

Naming, Transaction, Security, Concurrency, Licensing, Dynamic Resource Discovery, Events, Life Cycle, etc.

RMI vs CORBA

RMI provides a lightweight, Java-specific mechanism for objects to interact across the network AND the tools to make it work. The tools, classes, etc. are included within jdk.

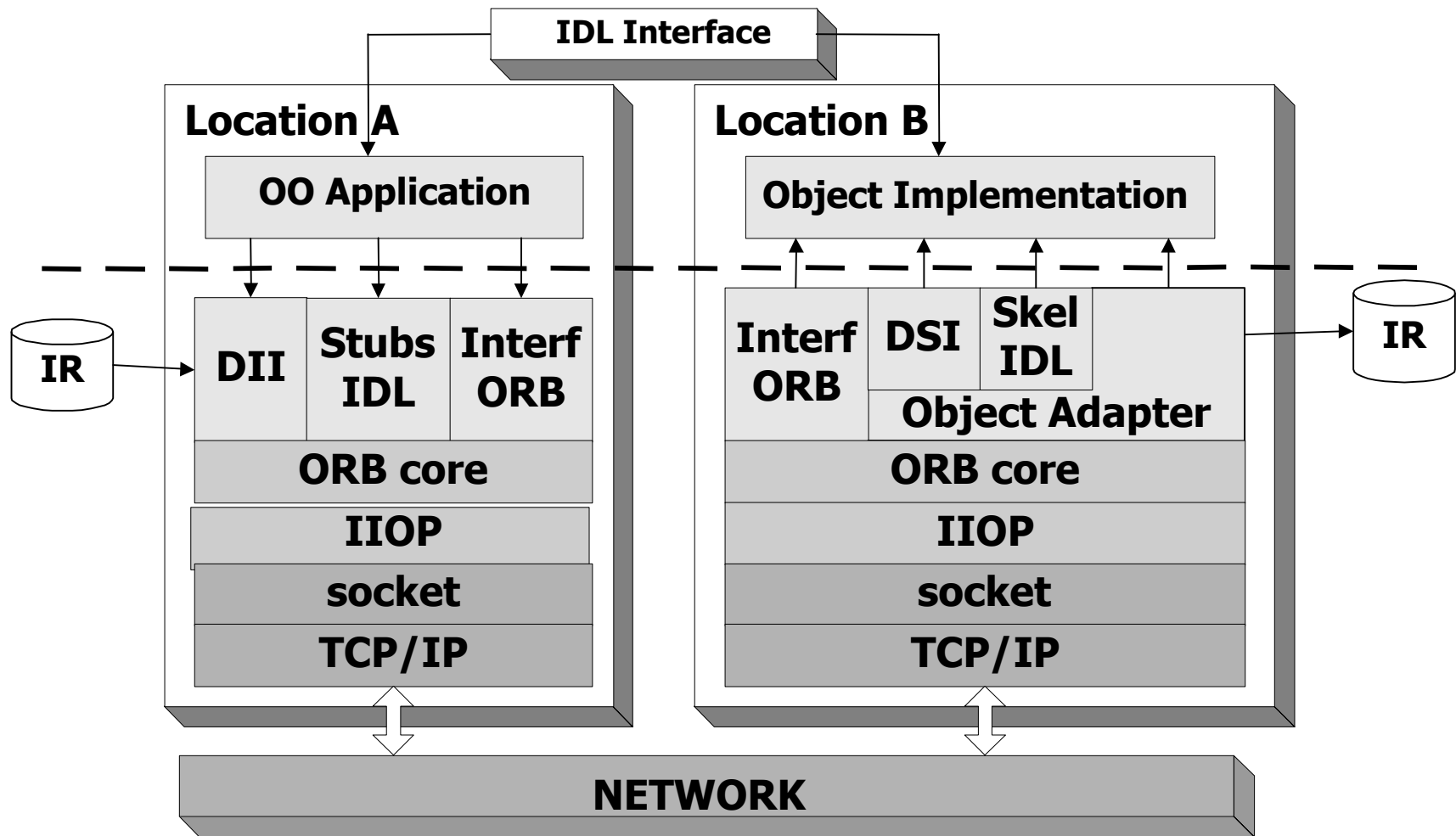
CORBA is a specification and an architecture for the integration of networked objects. Different vendors follow the CORBA specification to provide tools, libraries, etc. for multiple languages and platforms.

The CORBA specification ensures interoperability.

Sun provides a CORBA implementation for Java within the JDK

CORBA and RMI are not compatible.

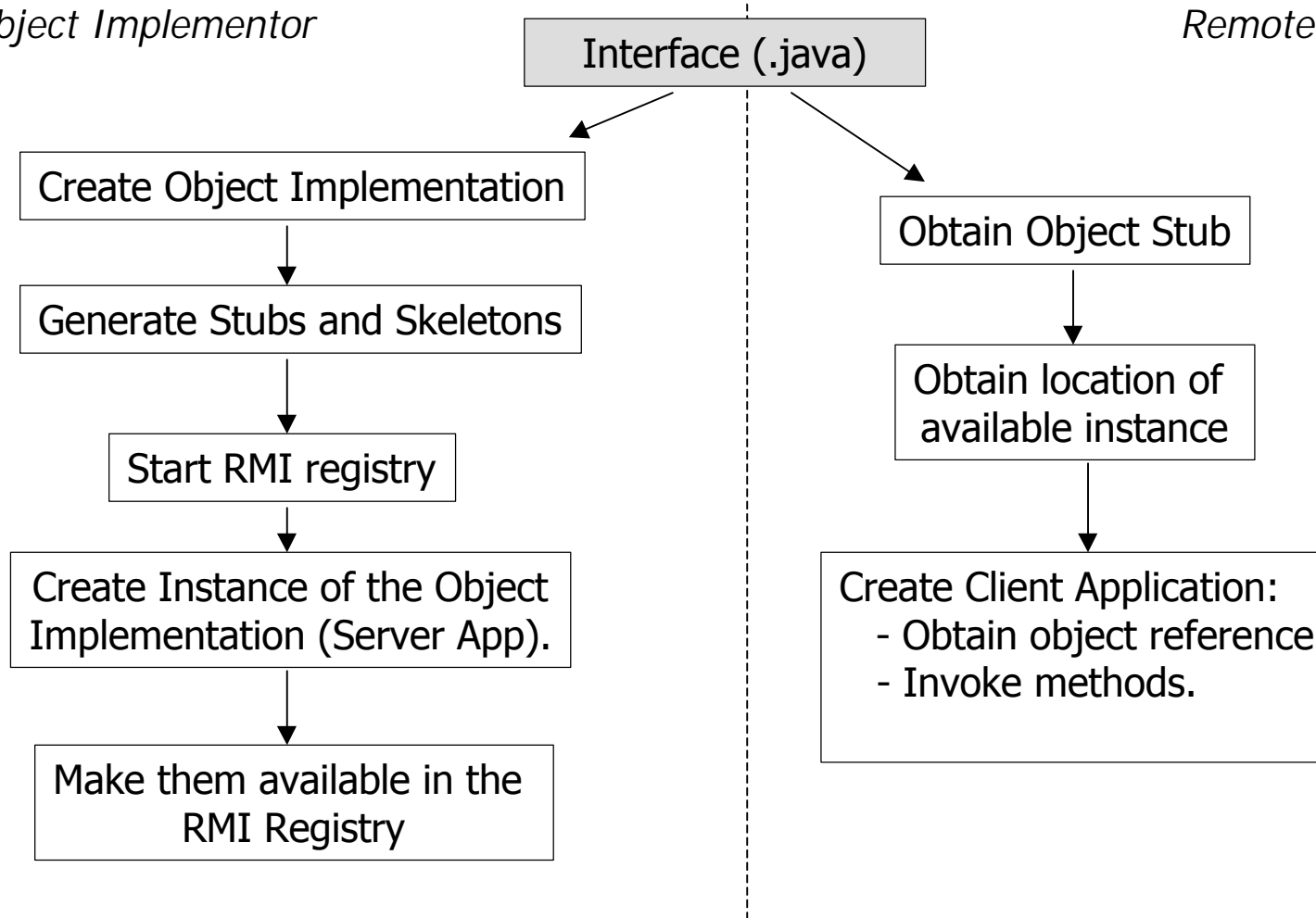
CORBA Architecture



The RMI Process

Object Implementor

Remote Object User



Scenario 1: The Interface

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Compute extends Remote {  
    int executeCalculation (int i) throws RemoteException;  
    String tellmeWhoYouAre (String name) throws RemoteException;  
}
```

Extend the Remote interface



Declare methods available on
remote objects implementing this
interface.
Note the exceptions

Scenario 1: Object Implementation

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

Extends some RMI remote object

```
public class ComputeEngine extends UnicastRemoteObject implements Compute
{
    private String id;
    private int count=0;
    public ComputeEngine(String _id) {
        super();
        id = _id;
    }
    public int executeCalculation(int i) {
        System.out.println("Calculation performed with: "+i);
        count=count+i;
        return count;
    }
    public String tellmeWhoYouAre(String name) {
        String s = new String("Hello, "+name+" I'm Calculator "+id);
        System.out.println("Greeting "+name);
        return s;
    }
}
```

Implements the common interface

Implements the methods required by the interface

Scenario 1: Setting this up

```
# Compile the interface
```

```
javac Compute.java
```

```
# Compile the implementation
```

```
javac ComputeEngine.java
```

```
# Generate stubs & skeletons
```

```
rmic ComputeEngine
```

```
# Start up RMI registry
```

```
rmiregistry&
```


Scenario 1: Registering Instances

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

```
public class Server
{
```

```
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            Compute engine1 = new ComputeEngine("32");
            Naming.rebind("Compute32", engine1);
            Compute engine2 = new ComputeEngine("33");
            Naming.rebind("Compute33", engine2);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Security Manager required by RMI



Create two instances, name and register them.



Now there are two objects implementing the Compute interface available through RMI on this machine. They are named `Compute32` and `Compute33`. Just start the server app: `java Server`

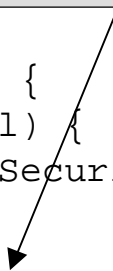
Scenario 1: the Client

```
import java.rmi.*;

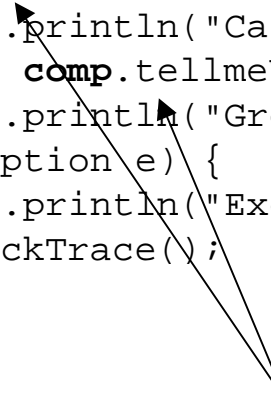
public class Client {

    public static void main (String args[]) {
        if (System.getSecurityManager()==null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//" + args[0] + "/" +args[1]
            Compute comp = (Compute) Naming.lookup(name);
            int i=comp.executeCalculation(1);
            System.out.println("Calculation result is "+i);
            String s = comp.tellmeWhoYouAre("the Client");
            System.out.println("Greeting is "+s);
        } catch (Exception e) {
            System.err.println("Exception in RMI: "+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

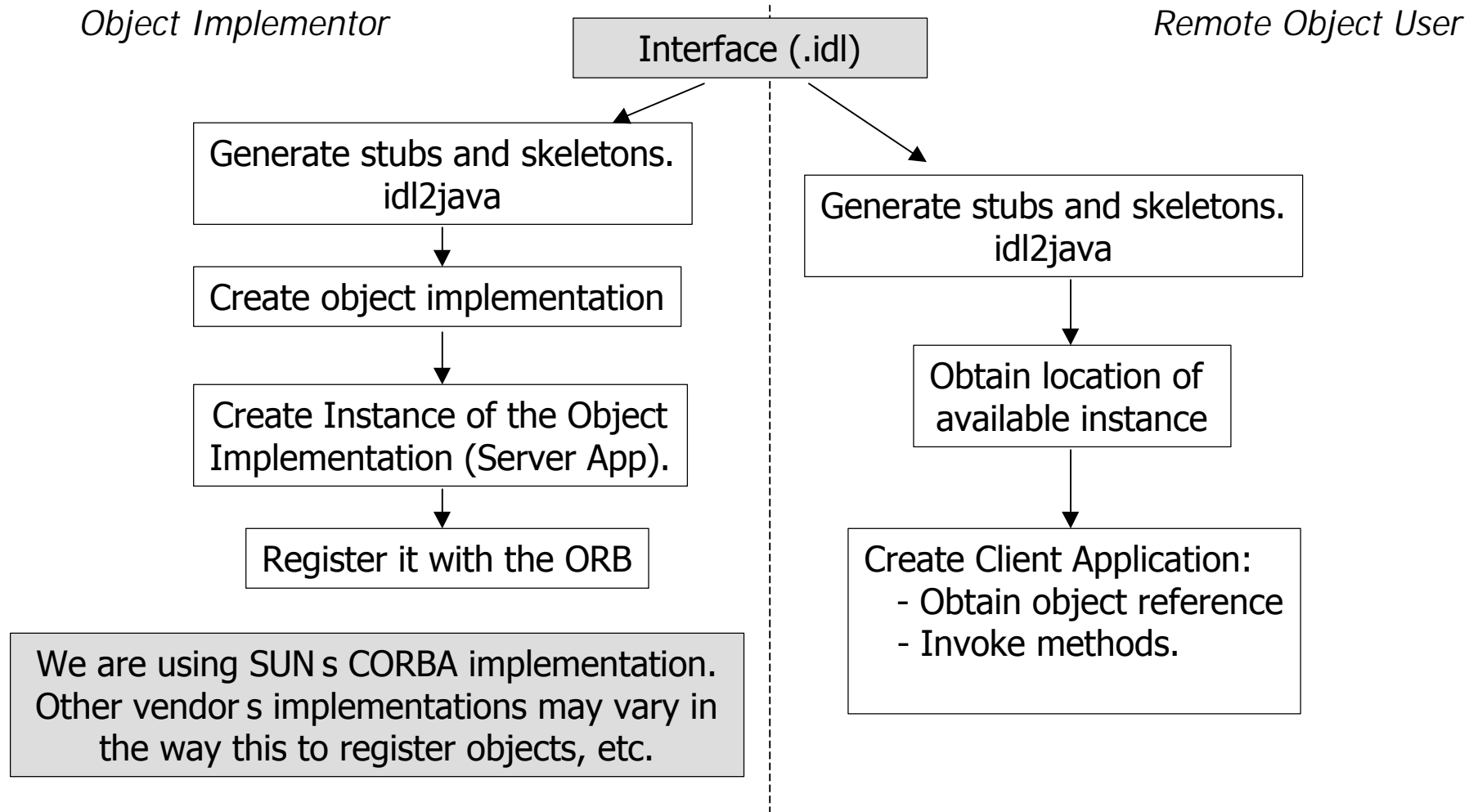
Use RMI classes to obtain a reference



Invoke methods on obtained reference.
Local stub and remote skeleton do the
dirty work for you.



The CORBA process



Scenario 2

We are going to implement the same as in scenario 1 but using CORBA.

We start off from the IDL interface and decide to make both the implementation and the client in Java.

But remember, both decisions are independent, we are just showing how this is done IN THE CASE where everything is in Java.

Our Java implementation object could be accessed from any language understanding CORBA.

Our Java client will be able to access any Compute object accessible through CORBA, no matter what language it is implemented on.

Scenario 2: IDL Interface

```
module Calculations {  
  
    interface Compute  
    {  
        long executeCalculation (in long i);  
        string tellmeWhoYouAre(in string name);  
    };  
};
```

This is IDL syntax. There is a complete specification of the syntax and how it maps to each language.

```
# Compile IDL interface into Java  
idl2java Calculations.idl
```

```
package Calculations;  
public interface Compute  
    extends org.omg.CORBA.Object {  
    int executeCalculation(int i);  
    String tellmeWhoYouAre(String name)  
;  
}
```


This generates some java classes.
Look at them:

- The equivalent Java interface (see types conversion)
- The stubs and skeletons.
- The Helper & Holder classes for further functionality

Scenario 2: Object Implementation

```
import Calculations.*;
import org.omg.CORBA.*;
```

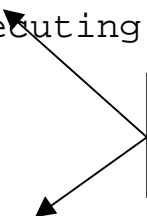
Extend the `_ComputeImplBase` class generated by `idl2java`.



```
public class ComputeImpl extends _ComputeImplBase {
```

```
    private int count = 0;
```

```
    public int executeCalculation(int i) {
        System.out.println("Executing calculation with "+i);
        count=count+i;
        return count;
    }
```



Implement methods as required by the interface definition.

```
    public String tellmeWhoYouAre(String name) {
        System.out.println("Greeting "+name);
        return "Hello "+name+", I'm the server";
    }
```

```
}
```

Scenario 2: Registering Instances

```
import Calculations.*;
import org.omg.CORBA.*;
```

```
public class Server {
    public static void main(String args[]) {
        try{
            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Create an instance and register it with the ORB
            ComputeImpl computeRef = new ComputeImpl();
            orb.connect(computeRef);

            System.out.println("Object IOR is: " + orb.object_to_string(computeRef));
            // Wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized(sync){
                sync.wait();
            }
        } catch(Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Create ORB object (see CORBA architecture)

Create and register object instance

Obtain a stringified reference (IOR)

Wait for connections

Scenario 2: Setting things up

```
# Compile all the idl2java generated classes
```

```
javac Calculations/*.java
```

```
# Compile the implementation
```

```
javac ComputeImpl.java
```

```
# Compile the server
```

```
javac Server.java
```

```
# Start up the server
```

```
java Server
```


Scenario 2: the Client

```
import Calculations.*;
import org.omg.CORBA.*;
```

```
public class Client {
```

```
    public static void main (String args[]) {
        try{
```

```
            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);
```

```
            Compute computeRef = ComputeHelper.narrow(orb.string_to_object(args[0]));
```

```
            // Call the Hello server object and print the results
            System.out.println("Calculation with 1: "+computeRef.executeCalculation(1));
            System.out.println("Greeting response : "+computeRef.tellmeWhoYouAre("Me"));
```

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

Create ORB object (see CORBA architecture)

Obtain object from IOR passed as argument

Cast object to desired class

Use object reference as if it was local

Some Issues

Implementations, Clients and Servers follow the same principles in CORBA and RMI.

RMI Client needs the stub of the implementation.

RMI Server is lighter (rmiregistry takes care of the communications).

Casting in CORBA is done via Helper classes.

We **HAVE TO INHERIT** the implementation from some other class: this may not be desirable in all situations. Solution: the delegation model.

The Delegation Model

In a Delegation Model implementations just implement interfaces but are not required to inherit from any specific class.

Implementations of many interfaces can truly be structured as desired with no restrictions (legacy software, etc.)

Need a couple more of steps or classes to make things work.

Scenario 3: Delegation Based RMI Implementation

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

No inheritance required. We inherit as we want

```
public class ComputeEngine implements Compute
```

```
{
```

```
    private String id;
```

```
    private int count=0;
```

```
    public ComputeEngine(String _id) throws RemoteException {
```

```
        super();
```

```
        id = _id;
```

```
        UnicastRemoteObject.exportObject(this)
```

```
    }
```

```
    public int executeCalculation(int i) {
```

```
        System.out.println("Calculation performed with: "+i);
```

```
        count=count+i;
```

```
        return count;
```

```
    }
```

```
    public String tellmeWhoYouAre(String name) {
```

```
        String s = new String("Hello, "+name+" I'm Calculator "+id);
```

```
        System.out.println("Greeting "+name);
```

```
        return s;
```

```
    }
```

Implements the common interface

MUST Make the object RMI aware

Scenario 4: CORBA delegation

```
# Tell idl2java to generate delegation logistics  
idl2java -ftie Calculations.idl
```

```
# This generates more classes  
ls Calculations
```

```
Calculations/Compute.java  
Calculations/_ComputeImplBase.java  
Calculations/_ComputeTie.java  
Calculations/ComputeHelper.java  
Calculations/_ComputeOperations.java  
Calculations/ComputeHolder.java  
Calculations/_ComputeStub.java
```

Scenario 4: Delegation Based CORBA Implementation

```
import Calculations.*;
// All CORBA applications need these classes.
import org.omg.CORBA.*;

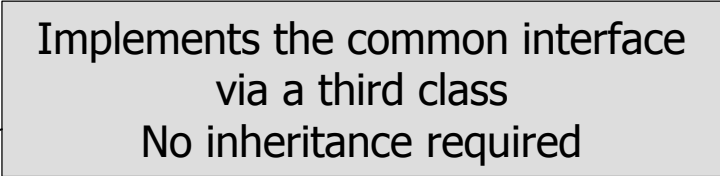
public class ComputeImpl implements _ComputeOperations {

    public int executeCalculation(int i) {
        System.out.println("Executing calculation with "+i);
        return i+1;
    }

    public String tellmeWhoYouAre(String name) {
        System.out.println("Greeting "+name);
        return "Hello "+name+", I'm the server";
    }

}
```

Implements the common interface
via a third class
No inheritance required



Scenario 4: Server Application

```
import Calculations.*;
// All CORBA applications need these classes.
import org.omg.CORBA.*;

public class Server {
    public static void main(String args[]) {
        try{

            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Create the servant and register it with the ORB
            ComputeImpl compute = new ComputeImpl();
            Compute computeRef = new _ComputeTie(compute);
            orb.connect(computeRef);
            System.out.println("Object IOR is: "+ orb.object_to_string(computeRef));

            // Wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized(sync){
                sync.wait();
            }
        }
    }
}
```

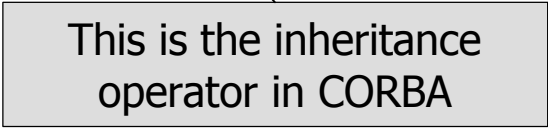
Create an instance of the implementation

Create Tie object from instance

We actually make available the tie object, not the implementation instance itself. TIE Object takes care of invoking our implementation

Scenario 5: Interface Inheritance

```
module Calculations {  
  
    interface Compute  
    {  
        long executeCalculation (in long i);  
        string tellmeWhoYouAre(in string name);  
    };  
  
    interface GreatCompute : Compute {  
        long executeGreatCalculation (in long i);  
    };  
};
```



This is the inheritance
operator in CORBA

Scenario 5

Scenario in CORBA but in RMI is exactly the same game
Decisions on the implementation are independent from
decisions on the interface.

In the implementation we can decide:

ComputeImpl implements Compute: `executeCalculation,`
`tellmeWhoYouAre`

GreatComputeImpl implements GreatCompute and inherits from
ComputeImpl: `executeGreatCalculation.`

Or:

ComputeImpl implements Compute: `executeCalculation,`
`tellmeWhoYouAre`

GreatComputeImpl implements GreatCompute without inheriting
from ComputeImpl: `executeCalculation, tellmeWhoYouAre,`
`executeGreatCalculation`

Scenario 5: Inheritance Implementation

We decide in the implementation to inherit from ComputeImpl. Here we are using **DELEGATION**

```
import Calculations.*;  
import org.omg.CORBA.*;
```

```
public class GreatComputeImpl  
    extends ComputeImpl  
    implements _GreatComputeOperations {
```

```
    public int executeGreatCalculation(int i) {  
        System.out.println("Executing calculation with "+i);  
        return i*2;  
    }
```

```
}
```

But we always implement an interface

and its methods

Scenario 5: Instantiation

```
// Create and initialize the ORB  
ORB orb = ORB.init(args, null);
```

```
// Create the servant and register it with the ORB  
ComputeImpl compute = new ComputeImpl();  
Compute computeRef = new _ComputeTie(compute);  
orb.connect(computeRef);  
System.out.println("Compute Instance is: " + orb.object_to_string(computeRef));
```

```
// Create the servant and register it with the ORB  
GreatComputeImpl greatcompute = new GreatComputeImpl();  
GreatCompute greatcomputeRef = new _GreatComputeTie(greatcompute);  
orb.connect(greatcomputeRef);  
System.out.println("GreatCompute Instance is: "  
    + orb.object_to_string(greatcomputeRef));
```

```
// Wait for invocations from clients  
java.lang.Object sync = new java.lang.Object();  
synchronized(sync){  
    sync.wait();  
}
```

Create one instance of ComputeImpl and make it available through a tie object

Create one instance of GreatComputeImpl and make it available through a tie object

Scenario 5: A Compute Client

```
import Calculations.*;
import org.omg.CORBA.*;

public class InvokeCompute
{
    public static void main (String args[])
    {
        try{

            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);
            Compute computeRef = ComputeHelper.narrow(orb.string_to_object(args[0]));

            // Call the Hello server object and print the results
            System.out.println("Calculation with 1: "+computeRef.executeCalculation(1));
            System.out.println("Greeting response : "+computeRef.tellmeWhoYouAre("Me"));

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

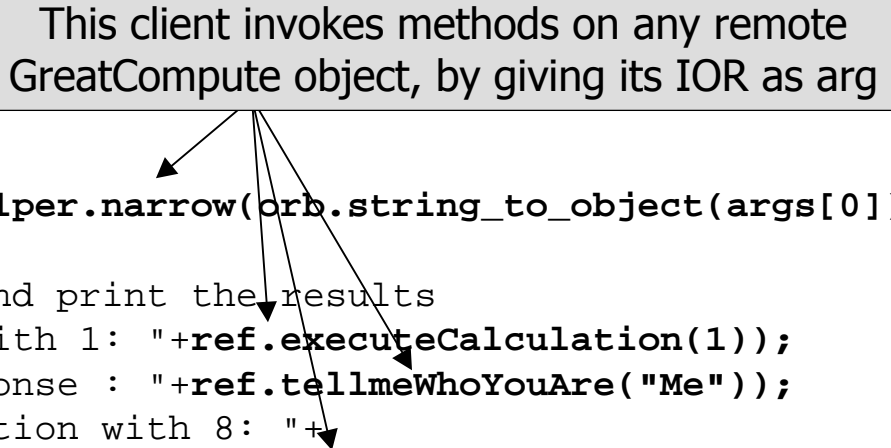
Scenario 5: A GreatCompute Client

```
import Calculations.*;
import org.omg.CORBA.*;
public class InvokeGreatCompute
{
    public static void main (String args[])
    {
        try{
            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);
            GreatCompute ref = GreatComputeHelper.narrow(orb.string_to_object(args[0]));

            // Call the Hello server object and print the results
            System.out.println("Calculation with 1: "+ref.executeCalculation(1));
            System.out.println("Greeting response : "+ref.tellmeWhoYouAre("Me"));
            System.out.println("Great Calculation with 8: "+
                ref.executeGreatCalculation(8));

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

This client invokes methods on any remote GreatCompute object, by giving its IOR as arg

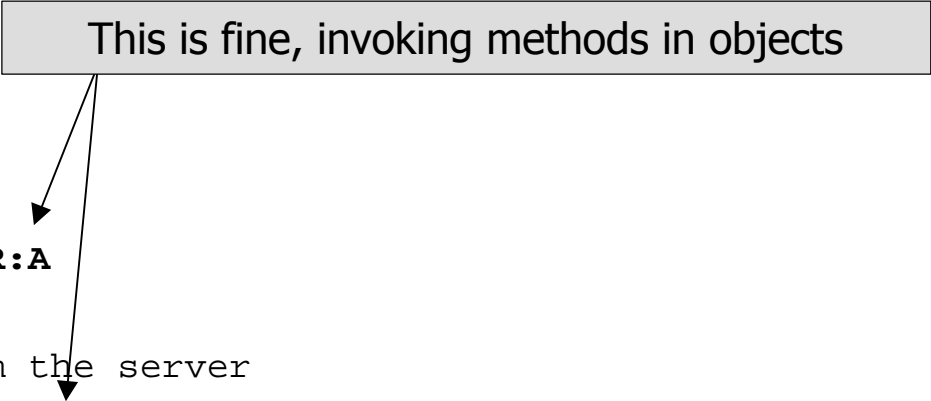


Scenario 5: This is OO!!

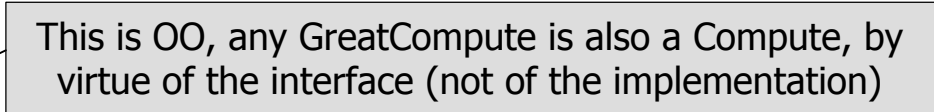
```
# Start the server
[rsplus13]> java Server
Compute instance is: IOR:A
GreatCompute instance is: IOR:B

# On the client:
[solaris]> java InvokeCompute IOR:A
Calculation with 1: 1
Greeting response : Hello Me, I'm the server
[solaris]> java InvokeGreatCompute IOR:B
Calculation with 1: 1
Greeting response : Hello Me, I'm the server
Great Calculation with 8: 16
[solaris]> java InvokeCompute IOR:B
Calculation with 1: 2
Greeting response : Hello Me, I'm the server
[solaris]> java InvokeGreatCompute IOR:A
*** ERROR ***
```

This is fine, invoking methods in objects



This is OO, any GreatCompute is also a Compute, by virtue of the interface (not of the implementation)



But a Compute is not a GreatCompute, so we get an error



Distributed Objects

We are doing distributed objects: clients actually don't have to know where objects are.

Through RMI and CORBA we can design the distribution of objects through the network independently from the clients using them.

Applications are made by assembling objects across the network.

Applications usage of resources is designed by the distribution of objects across machines.

Middleware

CORBA and RMI can also be used as middleware.

Whenever you have proprietary sw or hw to be accessed on a machine create an object on that machine with methods to access the sw or hw, and then make it available through CORBA or RMI.

For instance:

Want to control remotely a lamp connected to a computer?

Create an interface with two methods: `on()`, `off()`

Create an object in the computer the lamp is connected to so that it implements the previous interface.

Make it available through RMI and CORBA.

Clients can invoke remotely the `on()` or `off()` methods.

Summary

RMI and CORBA originate in the same base idea (accessing remote objects).

RMI is a lightweight JAVA specific method:

- Interfaces are created straightaway.

- Mechanisms to transfer code between client and server.

- Simple naming and security mechanisms for remote instances.

CORBA is a complex language independent architecture:

- Interfaces are specified in IDL.

- There is a collection of services to deploy complex applications:

 - Hierarchical naming service for objects (a la DNS) to avoid IORs

 - Dynamical Stubs and Skeletons to make generic clients or servers.

 - Transactions involving multiple objects (like in DBs).

 - Security and authentication to control objects access.

 - Life Cycle for controlling lifetime of objects across the network.

 - etc.

Summary

RMI and CORBA objects are not compatible (IIOP and RMI protocol are not compatible).

Sun and IBM have enabled RMI over CORBA.

An RMI remote object can be accessed as a CORBA object and CORBA objects could be accessed from RMI.

Use RMI if you know you are going to be using only Java (I.e. java Applets and servers) and your application is simple enough.

Don't complicate your life unnecessarily.

Use CORBA for **general solutions** and applications likely to grow in complexity.