

CS 02



BITS Pilani
Pilani Campus

Course Name : Middleware Technologies CSIW ZG524



Start Recording



IMP Note to Students

- It is important to know that just login to the session does not guarantee the attendance.
- Once you join the session, continue till the end to consider you as present in the class.
- IMPORTANTLY, you need to make the class more interactive by responding to Professors queries in the session.
- **Whenever Professor calls your number / name ,you need to respond, otherwise it will be considered as ABSENT**

Textbooks



T1: Letha Hughes Etzkorn - Introduction to middleware _ web services, object components, and cloud computing- Chapman and Hall_CRC (2017).

T2: William Grosso - Java RMI (Designing & Building Distributed Applications)

R1: Gregor Hohpe, Bobby Woolf - Enterprise Integration Patterns_ Designing, Building, and Deploying Messaging Solutions -Addison-Wesley Professional (2003)

R2: MongoDB in Action

Note: In order to broaden understanding of concepts as applied to Indian IT industry, students are advised to refer books of their choice and case-studies in their own organizations



Evaluation Components

Evaluation Component	Name	Type	Weight	Duration	Schedule
EC – 1	Quiz I, II & III	Individual / Take-home	15%		Pre/Post Mid-Sem
EC – 2	Assignment/ Laboratory Exercises	Practical	10%		TBA
EC – 3	Mid-Semester Examination	Closed Book	30%	2 Hrs.	TBA
EC – 4	End-Semester Examination	Open Book	45%	3 Hrs.	TBA



Modular Structure

No	Title of the Module
M1	Introduction and Evolution
M2	Enterprise Middleware
M3	Middleware Design and Patterns
M4	Middleware for Web-based Application and Cloud-based Applications
M5	Specialized Middleware



BITS Pilani
Pilani Campus

CS2: CORBA - Common Object Request Broker Architecture



BITS Pilani
Pilani Campus

CS1: Introduction and Evolution

Links for practicals



➤ CORBA Product Profiles

➤ <http://www.puder.org/corba/matrix/>

➤ AI-Trader

➤ <http://www.puder.org/aitrader/>

➤ Oracle WebLogic Server

➤ <https://www.oracle.com/in/java/weblogic/>

➤ Java IDL: The "Hello World" Example

➤ <https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlExample.html>

➤ CS 441 - CORBA Example - <https://www.cs.uic.edu/~i441/CORBA/index.html>

➤ Java RMI:

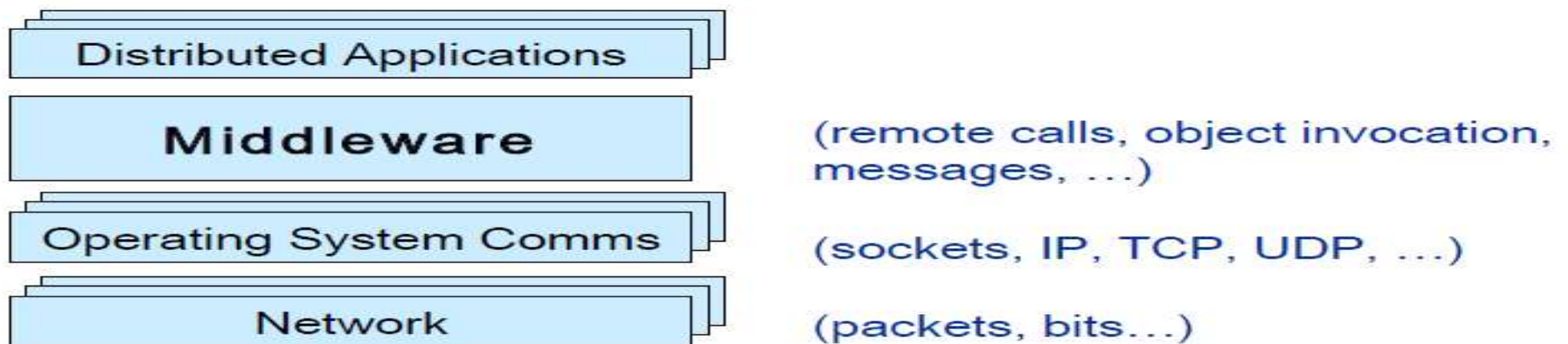
➤ https://www.tutorialspoint.com/java_rmi/java_rmi_application.htm

➤ <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html>

What is Middleware?



- Layer between OS and distributed applications
- Hides complexity and heterogeneity of distributed system
- Bridges gap between low-level OS communications and programming language abstractions
- Provides common programming abstraction and infrastructure for distributed applications
- Overview at: <http://www.middleware.org>



What is Middleware?



- Middleware provides support for (some of):
 - Naming, Location, Service discovery, Replication
 - Protocol handling, Communication faults, QoS
 - Synchronization, Concurrency, Transactions, Storage
 - Access control, Authentication

- Middleware dimensions:

- Request/Reply	vs. Asynchronous Messaging
- Language-specific	vs. Language-independent
- Proprietary	vs. Standards-based
- Small-scale	vs. Large-scale
- Tightly-coupled	vs. Loosely-coupled components

Common Forms of MW



- Sockets
- Remote Procedure Calls
- Distributed Object-Oriented Components (Ex: ORB)
- Message Oriented Middleware (Message Queues/Enterprise Message Bus etc.)
- Service Oriented Architectures
- Web services (Arbitrary / RESTful)
- SQL-oriented data access
- Embedded middleware
- Cloud Computing



BITS Pilani
Pilani Campus

CS2: CORBA - Common Object Request Broker Architecture



Agenda

- CORBA
- CORBA vs RMI
- Inter-ORB Protocols
- Object Bus and References
- CORBA Naming Services
- CORBA Object Services
- JAVA CORBA Facility

CORBA



- The Common Object Request Broker Architecture (CORBA) is a standard architecture for a distributed objects system.
- CORBA is designed to allow distributed objects to interoperate in a heterogeneous environment, where objects can be implemented in different programming language and/or deployed on different platforms
- CORBA is not itself a distributed objects facility; instead, it is a set of protocols.

CORBA

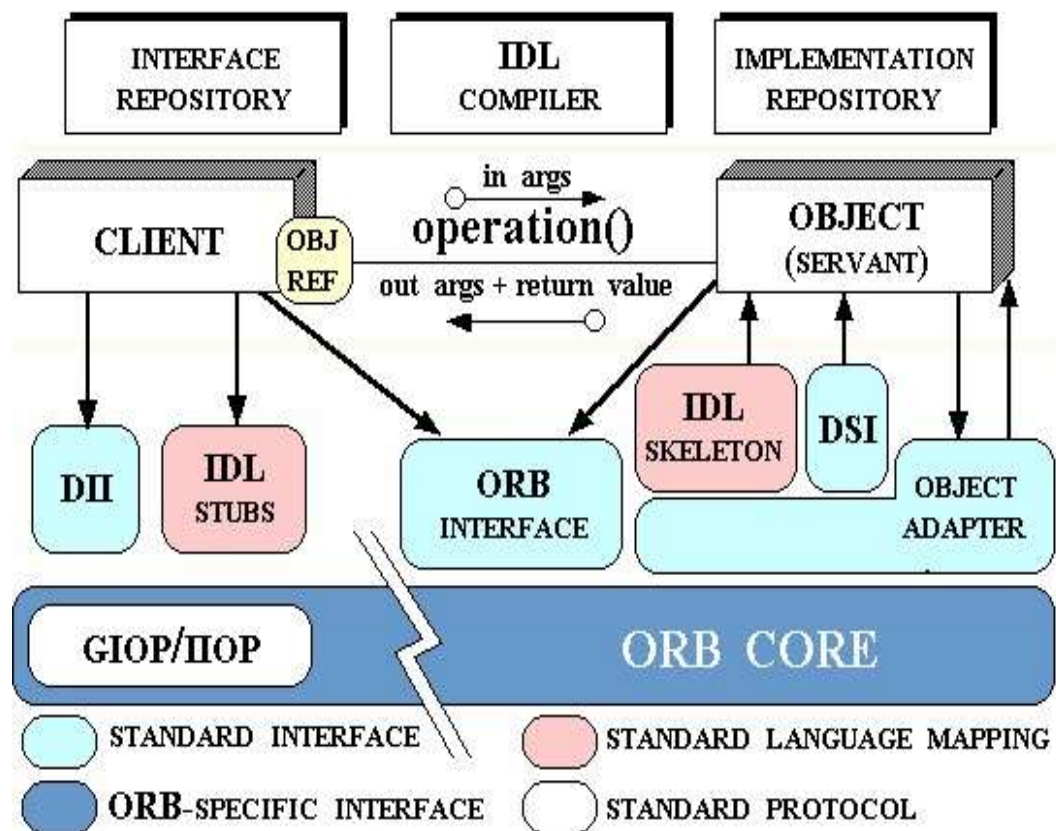


- A distributed object facility which adhere to these protocols is said to be CORBA-compliant, and the distributed objects that the facility support can interoperate with objects supported by other CORBA-compliant facilities.
- CORBA is a very rich set of protocols.
- We will focus on the key concepts of CORBA related to the distributed objects paradigm.
- We will also study a facility based on CORBA: the Java IDL.

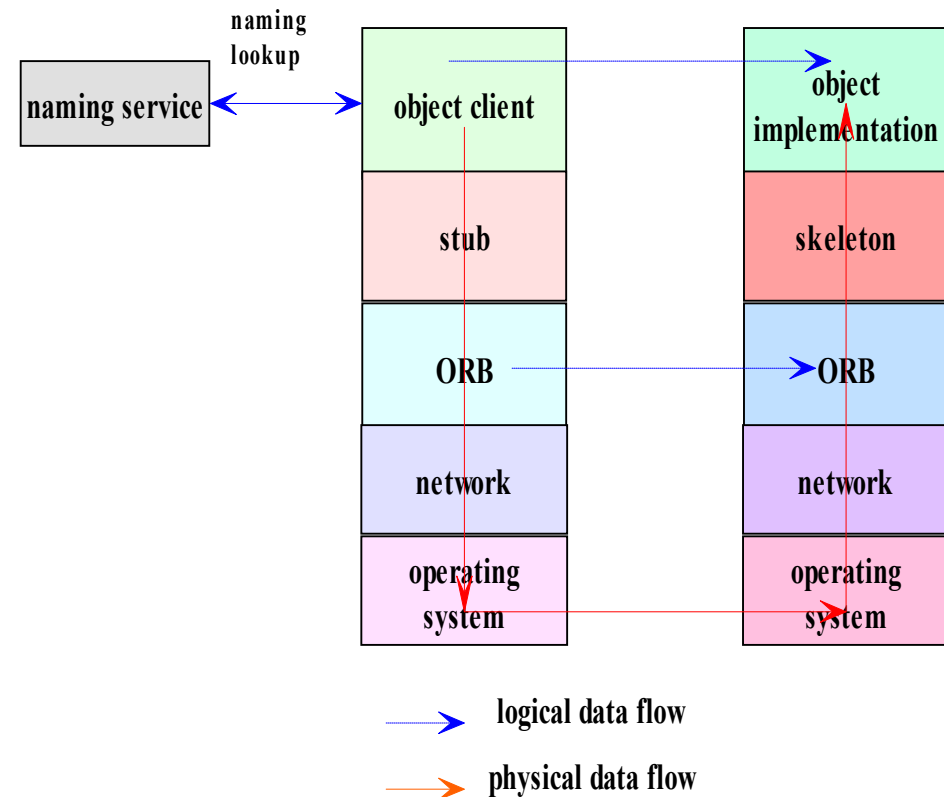
CORBA Architecture and Workflow



Architecture



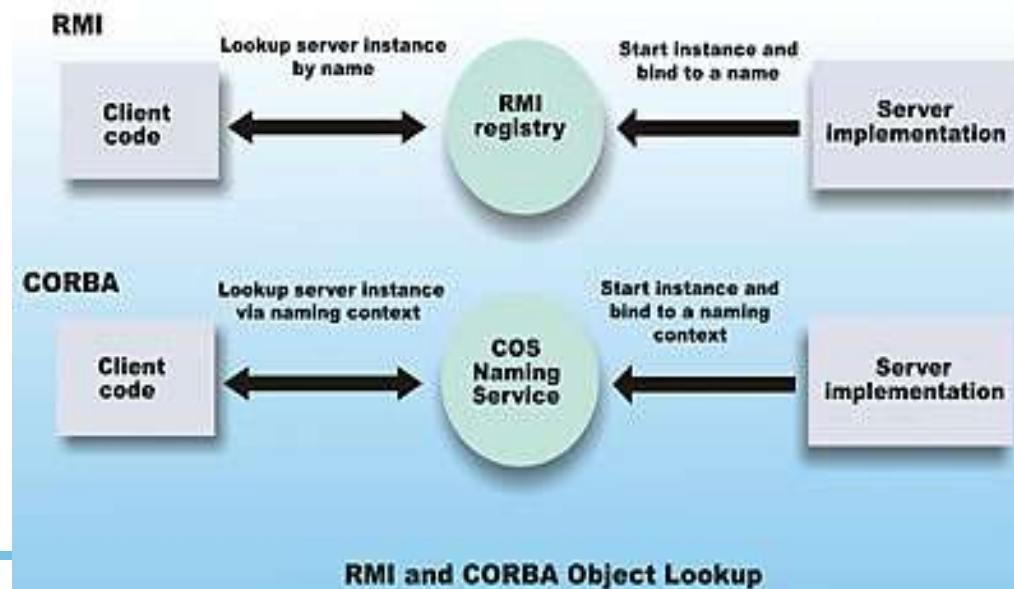
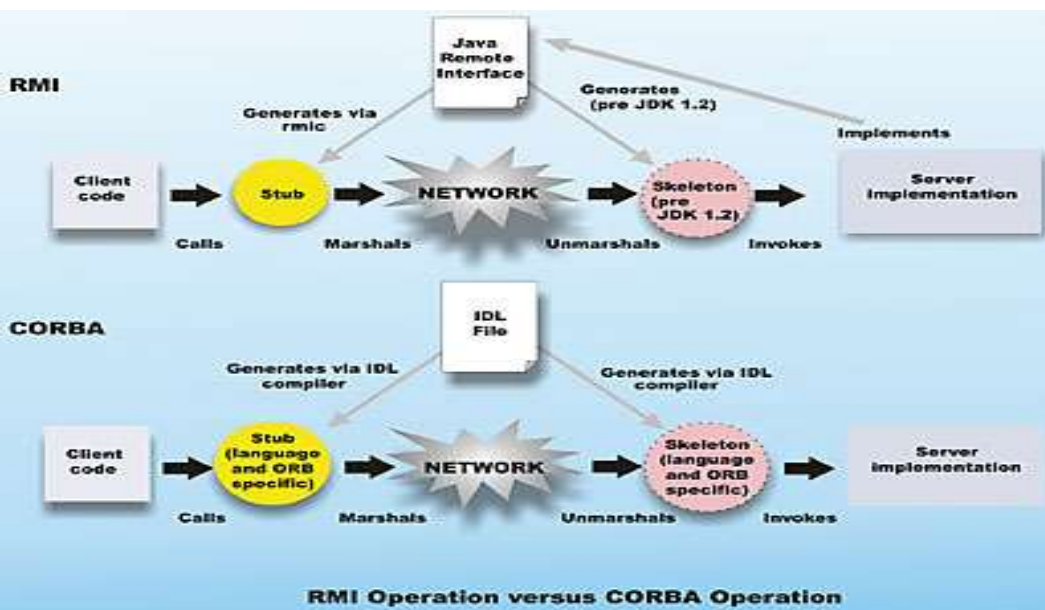
Basic Workflow



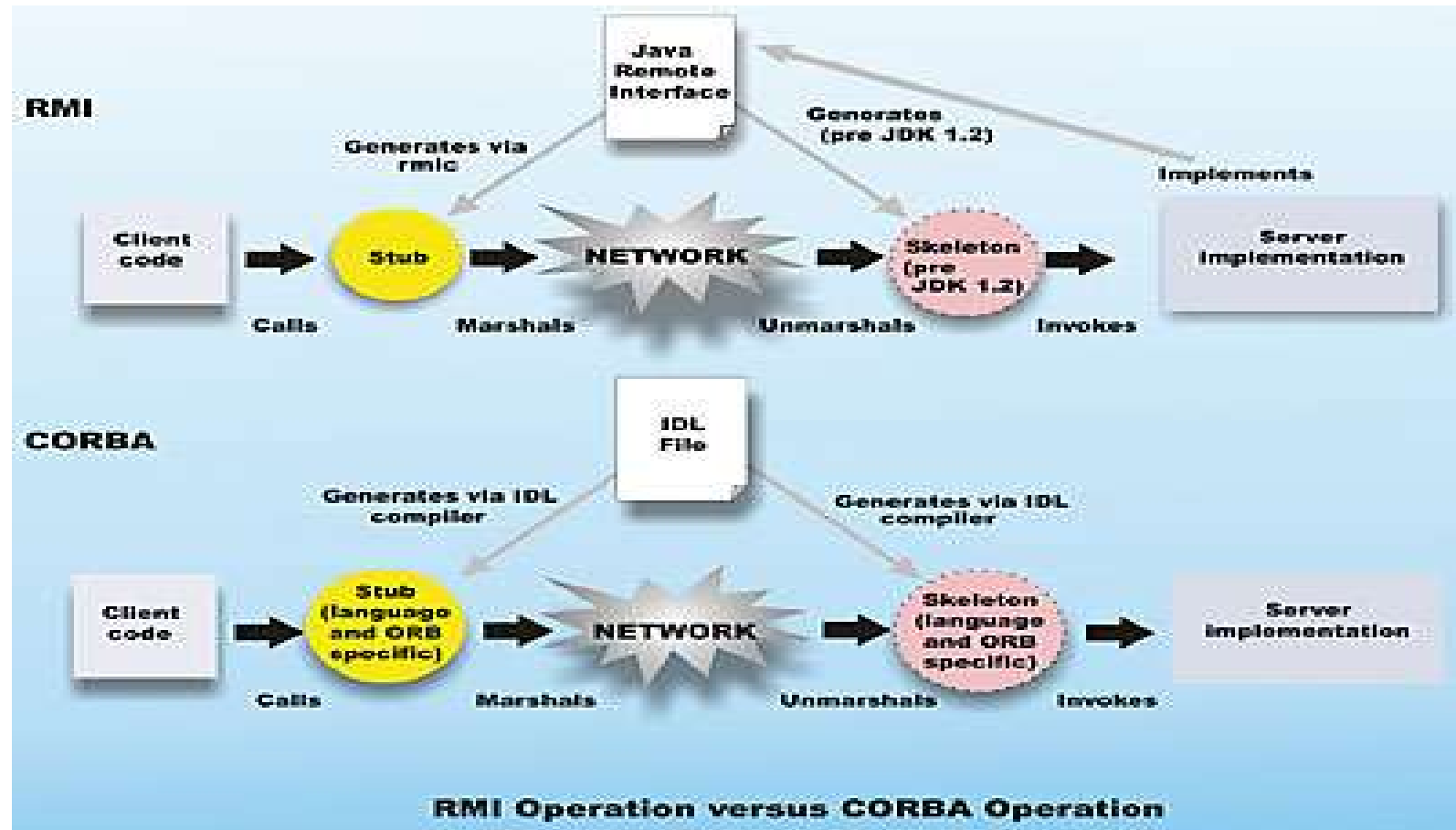
CORBA vs RMI



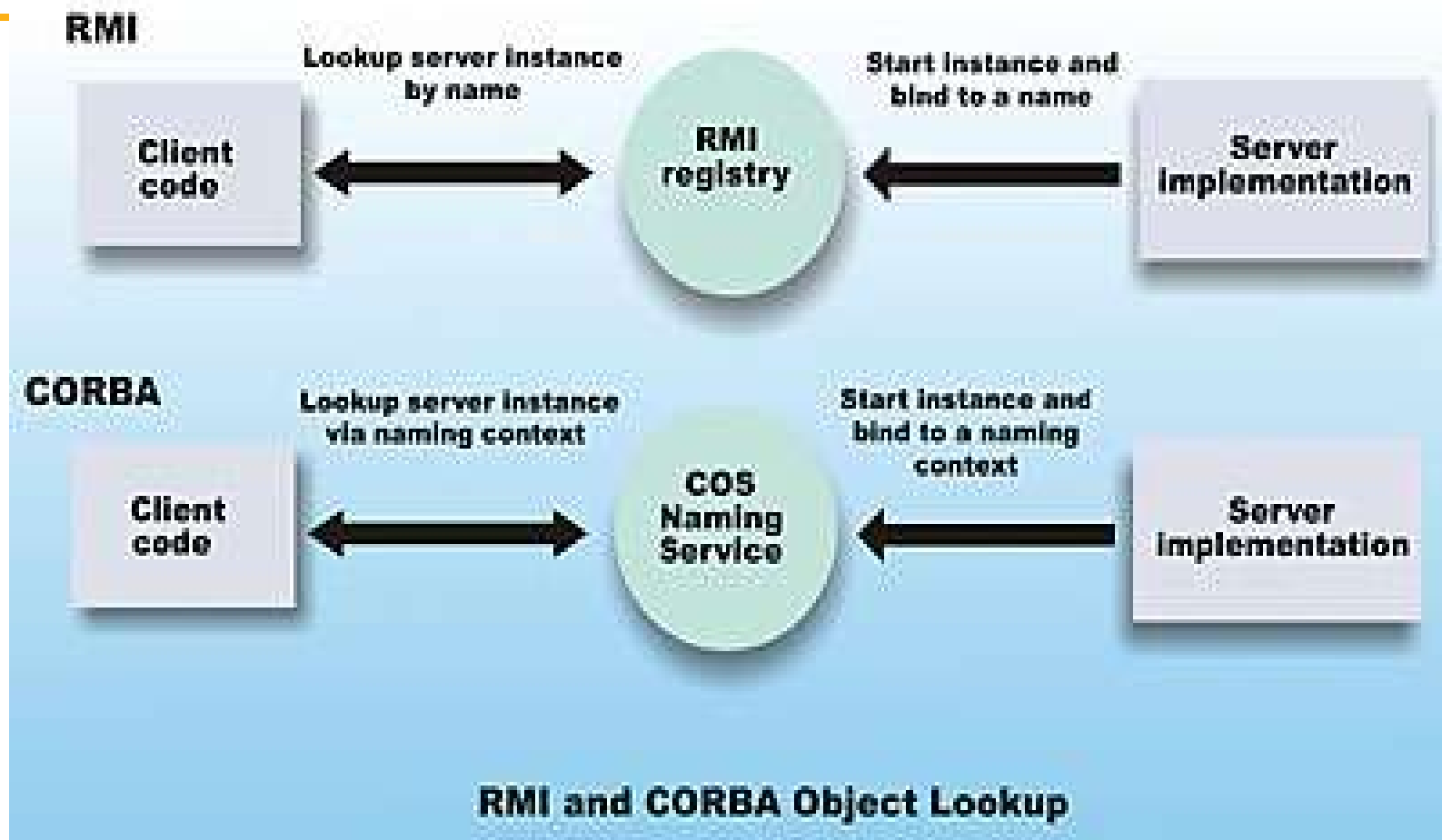
- CORBA differs from the architecture of Java RMI in one significant aspect:
 - RMI is a proprietary facility developed by Sun Microsystems, Inc., and supports objects written in the Java programming language only.
 - CORBA is an architecture that was developed by the Object Management Group (OMG), an industrial consortium.



CORBA vs RMI



CORBA vs RMI



CORBA Object Interface



- A distributed object is defined using a software file similar to the remote interface file in Java RMI.
- Since CORBA is language independent, the interface is defined using a universal language with a distinct syntax, known as the CORBA Interface Definition Language (IDL).
- The syntax of CORBA IDL is similar to Java and C++.
- However, object defined in a CORBA IDL file can be implemented in a large number of diverse programming languages, including C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLScript.

CORBA Object Interface



- For each of these languages, OMG has a standardized mapping from CORBA IDL to the programming language.
- This mapping helps - a compiler can be used to process a CORBA interface to generate the proxy files needed to interface with an object implementation or an object client written in any of the CORBA-compatible languages.

Mapping of OMG IDL Statements to C++

https://docs.oracle.com/cd/E13203_01/tuxedo/tux80/cref/member.htm

CORBA Addressing and Bindings Explained with an Industry Use Case



- Overview of CORBA (Common Object Request Broker Architecture)
- CORBA is a middleware standard defined by the Object Management Group (OMG) for enabling distributed object-oriented communication across different platforms, programming languages, and networks.
- It allows applications to communicate seamlessly, even if they are built using different technologies.
- One of the critical aspects of CORBA is Addressing and Bindings, which define how clients locate and communicate with remote objects.

CORBA Addressing and Bindings Explained with an Industry Use Case



- CORBA Addressing and Bindings
- Addressing in CORBA
- Addressing in CORBA refers to how clients find and refer to remote objects within a distributed environment. CORBA uses Object References (IOR - Interoperable Object References) to uniquely identify objects in a network.
- Types of Addressing in CORBA:
 - 1. Direct Addressing (Explicit Reference)
 - - Clients obtain the object reference explicitly (e.g., from a naming service or configuration file).
 - - Example: Using an Interoperable Object Reference (IOR) stored in a string format.

CORBA Addressing and Bindings Explained with an Industry Use Case



- 2. Indirect Addressing (Using Services)
 - - Clients use a Naming Service or Trader Service to discover objects dynamically.
 - - Example: CORBA Naming Service provides hierarchical object registration and lookup.
- 3. Persistent Addressing
 - - Objects are stored persistently and retrieved when required.
 - - Example: Banking applications store a user's account reference for future transactions.
- 4. Transient Addressing
 - - Object references exist only for the duration of the application's execution.
 - - Example: A video streaming service's session reference expires after logout.



CORBA Addressing and Bindings Explained with an Industry Use Case

- Bindings in CORBA
- Binding refers to how a client establishes a connection with a CORBA object before making a request.
- Types of Bindings in CORBA:

Types of Bindings in CORBA:

Binding Type	Description	Use Case Example
Static Binding	The client and server are compiled together with predefined object references.	A factory automation system where sensors communicate with a pre-registered central controller.
Dynamic Binding	The client discovers the object reference at runtime using a Naming Service or IOR.	An e-commerce website fetching available inventory dynamically from different warehouse servers.

CORBA Addressing and Bindings Explained with an Industry Use Case



- How Binding Works in CORBA:
- 1. Client gets an object reference (directly or via Naming Service).
- 2. The client invokes a method on the object reference, and CORBA handles communication.
- 3. CORBA's ORB (Object Request Broker) marshals the request and sends it over the network.
- 4. The server's ORB unmarshals the request, invokes the method, and sends the response back.

CORBA Addressing and Bindings Explained with an Industry Use Case



- Industry Use Case: CORBA in Banking Transactions
- Scenario:
 - A global banking system needs to handle secure transactions between branches worldwide. The system requires:
 - - Interoperability (different banking software stacks communicate).
 - - Scalability (millions of customers and transactions).
 - - Security (transactions must be encrypted and reliable).
- How CORBA Addressing and Binding Help in Banking Transactions:
 - 1. Addressing:
 - - Each banking server has a unique object reference for customer accounts.
 - - The CORBA Naming Service allows clients (banking applications) to find account objects dynamically.
 - - The system uses persistent addressing to ensure that an account reference remains valid across sessions.

CORBA Addressing and Bindings Explained with an Industry Use Case



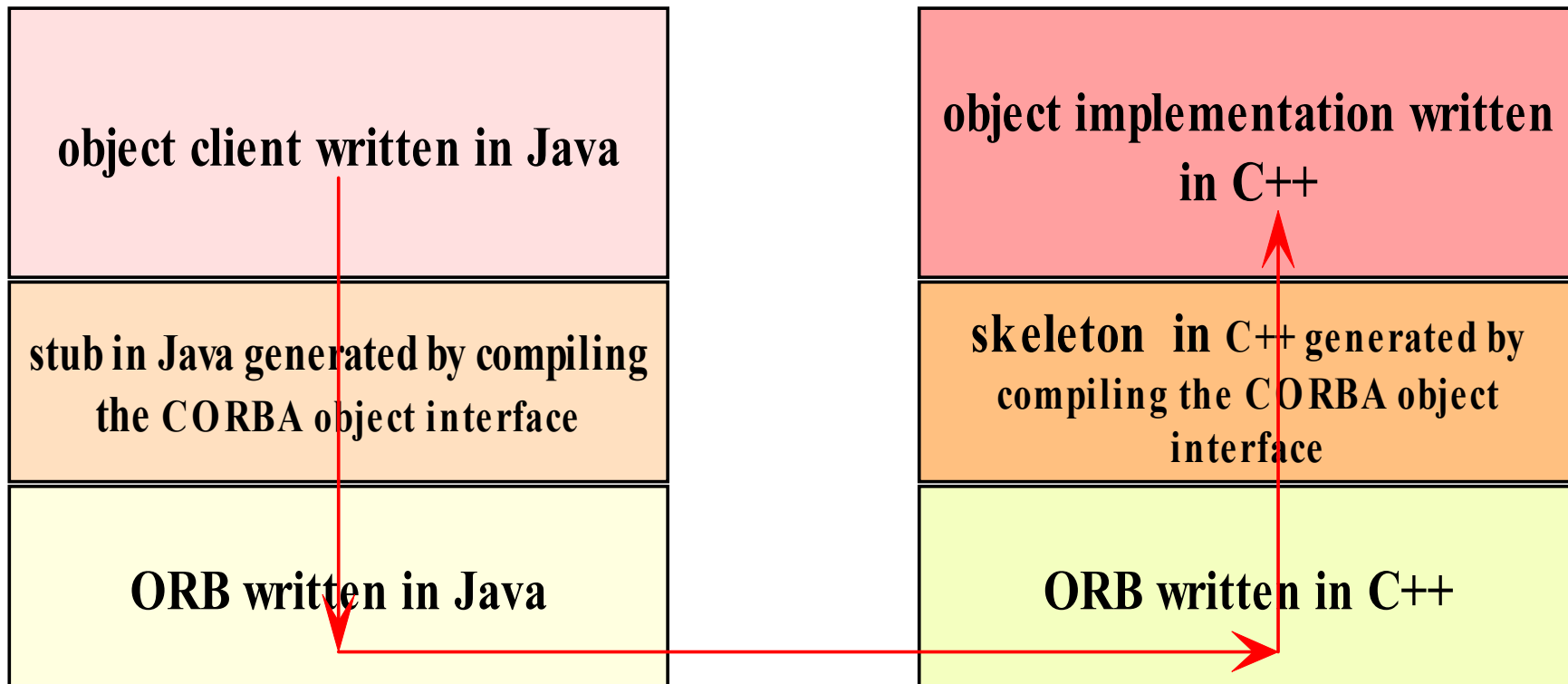
- Binding:
 - - When a customer initiates a transaction, the banking application dynamically binds to the bank's transaction-processing service.
 - - The Interoperable Object Reference (IOR) ensures secure communication with the correct branch.
 - - The client ORB and server ORB communicate securely to process the transaction.
- Process Flow:
 - 1. The user logs into the banking portal (client).
 - 2. The client application queries the CORBA Naming Service to locate the account service.
 - 3. CORBA binds dynamically to the correct banking server.
 - 4. The transaction is executed securely via ORB communication.
 - 5. The server returns the result, confirming success or failure.

CORBA Addressing and Bindings Explained with an Industry Use Case



- Benefits in the Banking System:
 - Seamless Communication between different banking software stacks.
 - Scalability to handle millions of transactions per second.
 - Security via encrypted communication.
 - Fault Tolerance by using CORBA's built-in load balancing and failover mechanisms.
- Conclusion
- CORBA's addressing and bindings provide a robust mechanism for distributed applications to communicate seamlessly.
- Industries like banking, telecommunications, and aerospace use CORBA to handle complex, cross-platform communication.

Cross-language CORBA application

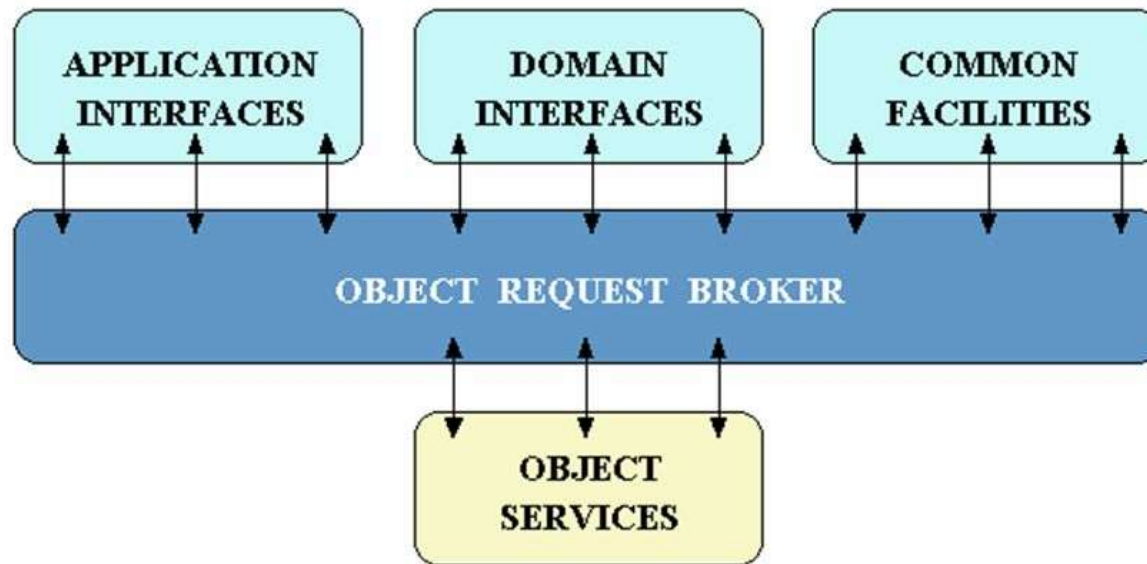


Inter-ORB Protocols



- To allow ORBs to be interoperable, the OMG(Object Management Group) specified a protocol known as the General Inter-ORB Protocol (GIOP), a specification which “provides a general framework for protocols to be built on top of specific transport layers.”
- A special case of the protocol is the Inter-ORB Protocol (IIOP), which is the GIOP applied to the TCP/IP transport layer.

CORBA



OMG Reference Model architecture

Inter-ORB Protocols



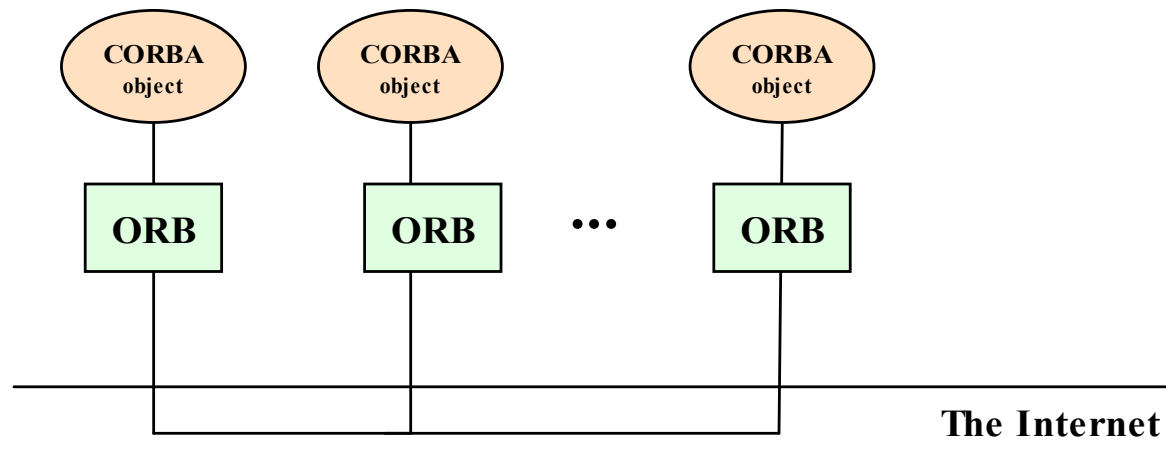
➤ The IIOP specification includes the following elements:

1. Transport management requirements specifies the connection and disconnection requirements, and the roles for the object client and object server in making and unmaking connections.
 2. Definition of common data representation: a coding scheme for marshalling and unmarshalling data of each IDL data type
 3. Message formats: different types of message format are defined.
 4. The messages allow clients to send requests to object servers and receive replies.
-
1. A client uses a Request message to invoke a method declared in a CORBA interface for an object and receives a reply message from the server.

Object Bus



- An ORB which adheres to the specifications of the IIOP may interoperate with any other IIOP-compliant ORBs over the Internet.
- This gives rise to the term “object bus”, where the Internet is seen as a bus that interconnects CORBA objects.
- There are a large number of proprietary as well as experimental ORBs available:

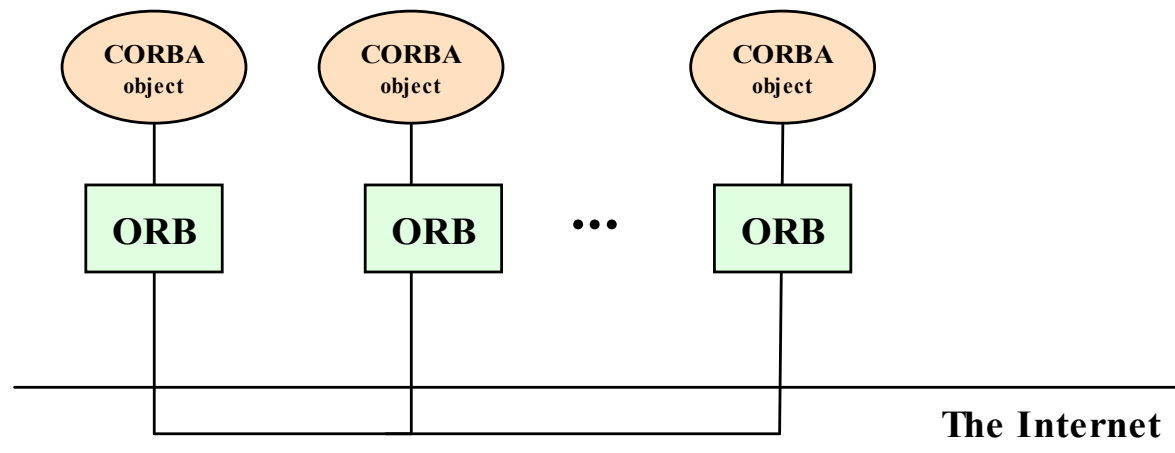


Object Bus



➤ (See CORBA Product Profiles, <http://www.puder.org/corba/matrix/>)

- Orbix IONA
- Borland Visibroker
- PrismTech's OpenFusion
- Web Logic Enterprise from BEA
- Ada Broker from ENST
- Free ORB

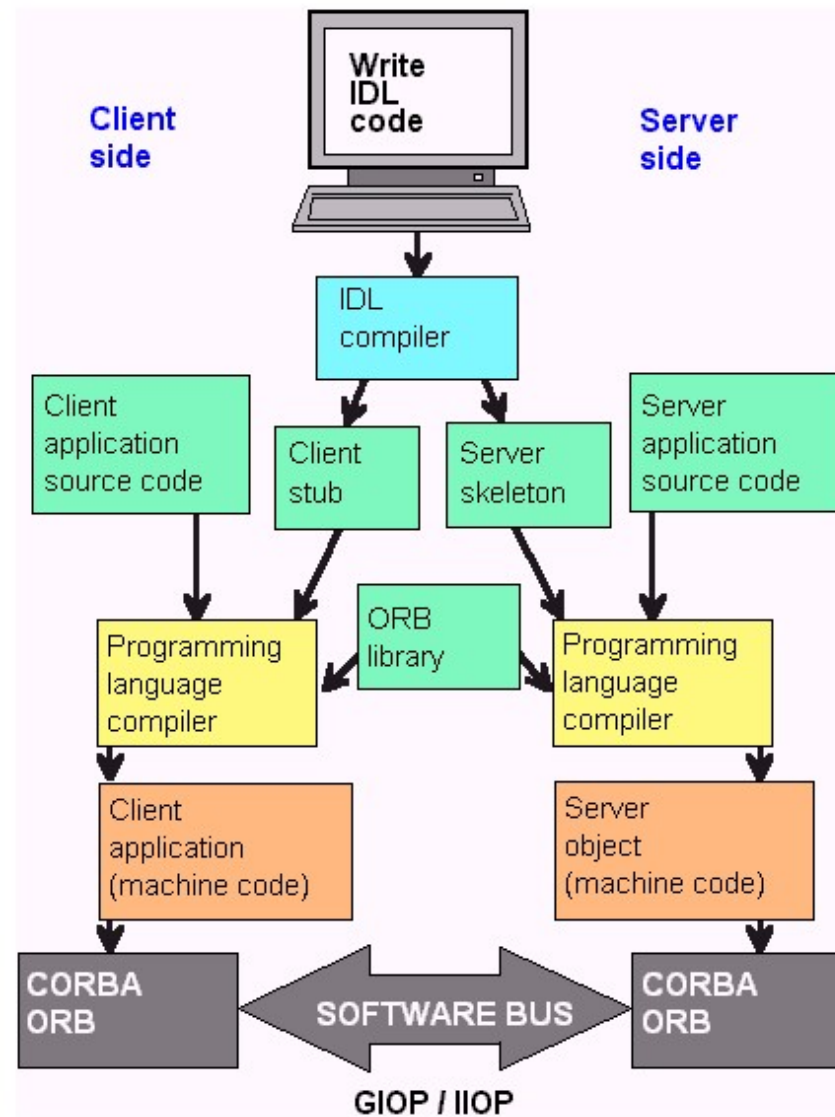


Object Servers, Clients and Reference



- As in Java RMI, a CORBA distributed object is exported by an object server, similar to the object server in RMI.
- An object client retrieves a reference to a distributed object from a naming or directory service, to be described, and invokes the methods of the distributed object.
- As in Java RMI, a CORBA distributed object is located using an object reference. Since CORBA is language-independent, a CORBA object reference is an abstract entity mapped to a language-specific object reference by an ORB, in a representation chosen by the developer of the ORB.
- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the Interoperable Object Reference (IOR) protocol.

CORBA



Interoperable Object Reference (IOR)



- An ORB compatible with the IOR protocol will allow an object reference to be registered with and retrieved from any IOR-compliant directory service.
- CORBA object references represented in this protocol are called Interoperable Object References (IORs).
- An IOR is a string that contains encoding for the following information:
 - The type of the object.
 - The host where the object can be found.
 - The port number of the server for that object.
 - An object key, a string of bytes identifying the object.
 - The object key is used by an object server to locate the object.

Interoperable Object Reference (IOR)



- The following is an example of the string representation of an IOR [5]:
- *IOR:0000000000000000d49444c3a677269643a312e3000000000000000100000000000004c0001000000000015756c472612e6475626c696e2e696f6e612e696500000963000000283a5c756c7472612e6475626c696e2e696f6e612e69653a677269643a303a3a49523a67726964003a*
- The representation consists of the character prefix “IOR:” followed by a series of hexadecimal numeric characters, each character representing 4 bits of binary data in the IOR.

CORBA Naming Service



- CORBA specifies a generic directory service.
- The Naming Service serves as a directory for CORBA objects, and, as such, is platform independent and programming language independent.
- The Naming Service permits ORB-based clients to obtain references to objects they wish to use.
- It allows names to be associated with object references.
- Clients may query a naming service using a predetermined name to obtain the associated object reference.

CORBA Naming Service

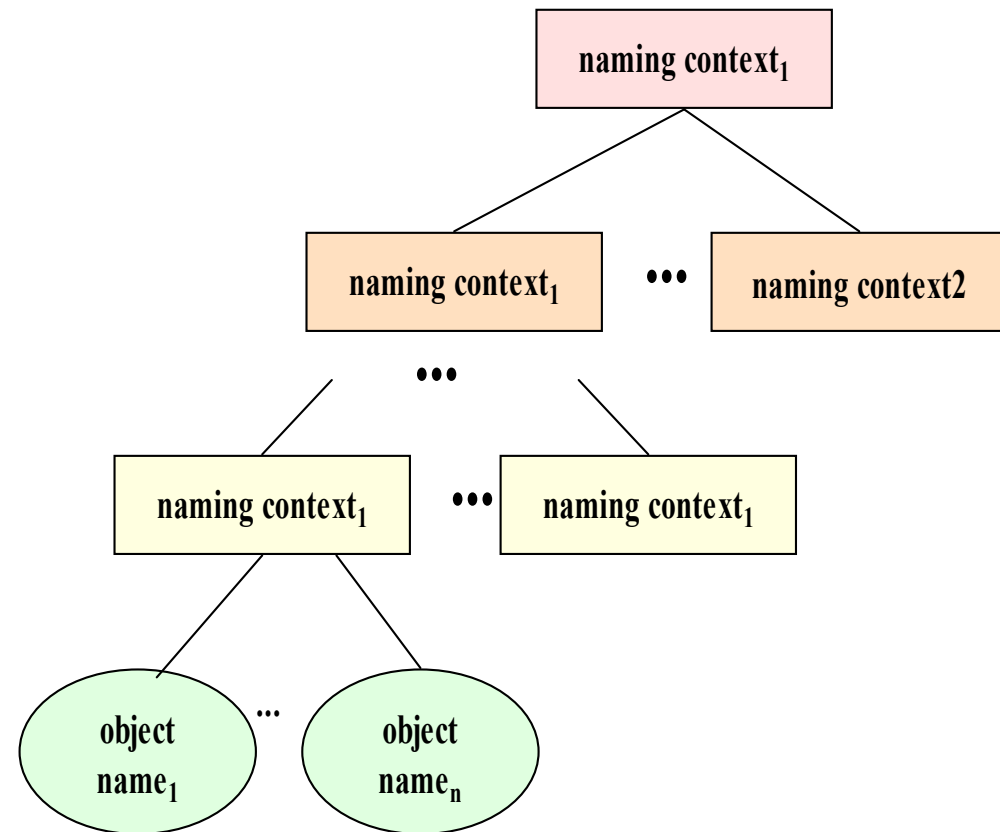


- To export a distributed object, a CORBA object server contacts a Naming Service to bind a symbolic name to the object
- The Naming Service maintains a database of names and the objects associated with them.
- To obtain a reference to the object, an object client requests the Naming Service to look up the object associated with the name
- (This is known as resolving the object name.)

CORBA Naming Service



- The API for the Naming Service is specified in interfaces defined in IDL, and includes methods that allow servers to bind names to objects and clients to resolve those names.
- To be as general as possible, the CORBA object naming scheme is necessary complex.
- Since the name space is universal, a standard naming hierarchy is defined in a manner similar to the naming hierarchy in a file directory



CORBA Naming Context



- A naming context corresponds to a folder or directory in a file hierarchy, while object names correspond to a file.
- The full name of an object, including all the associated naming contexts, is known as a compound name.
- The first component of a compound name gives the name of a naming context, in which the second component is accessed.
- This process continues until the last component of the compound name has been reached.

CORBA Naming Context



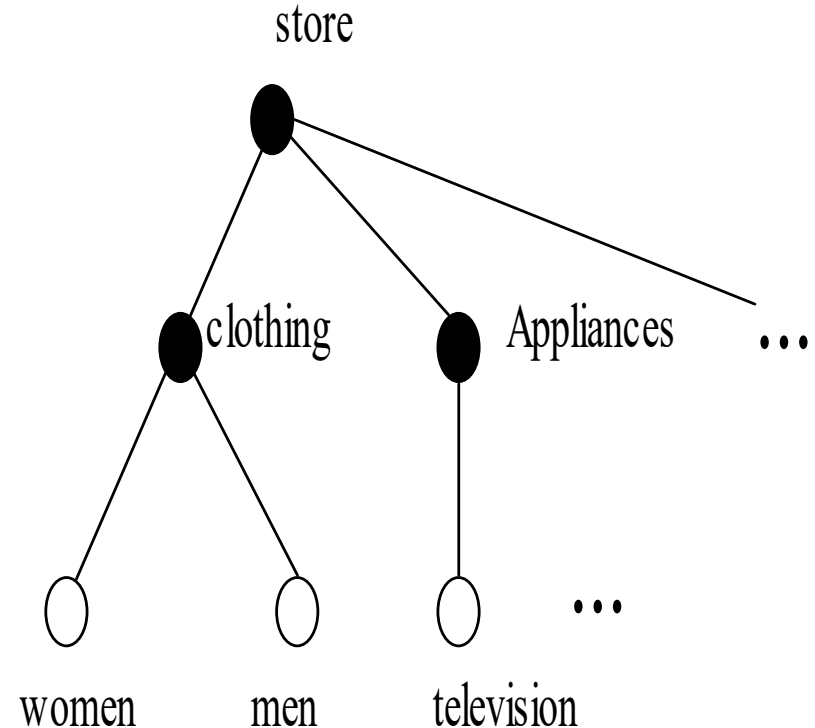
- Naming contexts and name bindings are created using methods provided in the Naming Service interface.
- CORBA Object Name: The syntax for an object name is as follows:
 <naming context > ...<naming context><object name>
 where the sequence of naming contexts leads to the object name.

Example of a naming hierarchy



- As shown, an object representing the men's clothing department is named store.clothing.men, where store and clothing are naming contexts, and men is an object name.
- The **Interoperable Naming Service (INS)** is a URL-based naming system based on the CORBA Naming Service, it allows applications to share a common initial naming context and provide a URL to access a CORBA object.

<https://aws.amazon.com/api-gateway>
<https://aws.amazon.com/api-gateway/pricing/>
<https://aws.amazon.com/api-gateway/features/>



CORBA Object Services



- CORBA specify services commonly needed in distributed applications, some of which are:
 - Naming Service:
 - Concurrency Service:
 - Event Service: for event synchronization;
 - Logging Service: for event logging;
 - Scheduling Service: for event scheduling;
 - Security Service: for security management;
 - Trading Service: for locating a service by the type (instead of by name);

CORBA Object Services



- CORBA specify services commonly needed in distributed applications, some of which are:
 - Time Service: a service for time-related events;
 - Notification Service: for events notification;
 - Object Transaction Service: for transactional processing.
- Each service is defined in a standard IDL that can be implemented by a developer of the service object, and whose methods can be invoked by a CORBA client.

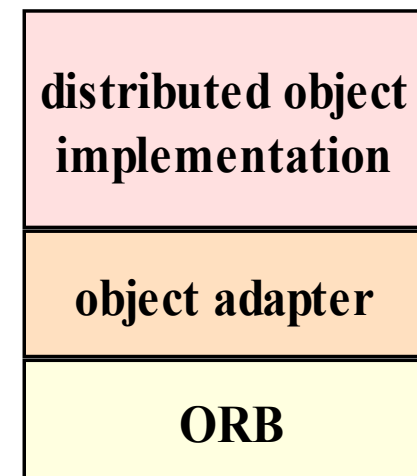
[A First CORBA Application](#)

https://docs.oracle.com/cd/F49540_01/DOC/java.815/a64683/corba3.htm

Object Adapters



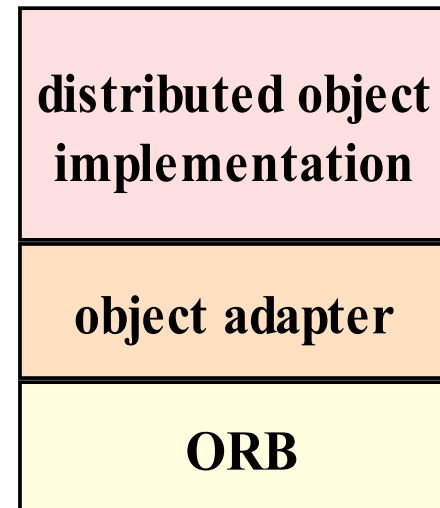
- In the basic architecture of CORBA, the implementation of a distributed object interfaces with the skeleton to interact with the stub on the object client side.
- As the architecture evolved, a software component in addition to the skeleton was needed on the server side: an object adapter.
- An object adapter simplifies the responsibilities of an ORB by assisting an ORB in delivering a client request to an object implementation.
- When an ORB receives a client's request, it locates the object adapter associated with the object and forwards the request to the adapter.



Object Adapters



- The adapter interacts with the object implementation's skeleton, which performs data marshalling and invoke the appropriate method in the object.
- There are different types of CORBA object adapters.
- The **Portable Object Adapter, or POA**, is a particular type of object adapter that is defined by the CORBA specification.
- An object adapter that is a POA allows an object implementation to function with different ORBs, hence the word portable.



Portable Object Adapters



- Portable Object Adapter (POA) in CORBA: Examples & Exercises
- 1. What is the Portable Object Adapter (POA)?
- The Portable Object Adapter (POA) is a CORBA component that manages object implementations and request dispatching.
- It allows developers to control object activation, request handling, and object lifespan.

Portable Object Adapters



- POA Features & Lifecycle
- Key Features of POA:
 - - Persistent and transient objects (controls object lifecycle).
 - - Multi-threading and request processing policies (handles multiple requests efficiently).
 - - Object Activation Policies (implicit vs. explicit activation).
 - - Servant Managers (allows lazy object creation).
- POA Lifecycle in CORBA:
 - 1. Create a POA (from the ORB).
 - 2. Create a servant (implementation of the CORBA object).
 - 3. Activate the servant (bind to an object reference).
 - 4. Process requests (handle client calls).
 - 5. Deactivate servant (cleanup).

Portable Object Adapters



- Example: Implementing POA in CORBA
- Let's implement a simple CORBA server-client application using POA.
- Step 1: Define the IDL (Interface Definition Language)
- ```
module Banking {
 interface Account {
 float getBalance();
 };
};
```
- This defines a `Banking::Account` interface that provides a `getBalance()` method.

# Java IDL – Java's CORBA Facility



- IDL is part of the Java 2 Platform, Standard Edition (J2SE).
- The Java IDL facility includes a CORBA Object Request Broker (ORB), an IDL-to-Java compiler, and a subset of CORBA standard services.
- In addition to the Java IDL, Java provides a number of CORBA-compliant facilities, including RMI over IIOP, which allows a CORBA application to be written using the RMI syntax and semantics.
- Key Java IDL Packages
  - package org.omg.CORBA – contains interfaces and classes which provides the mapping of the OMG CORBA APIs to the Java programming language
  - package org.omg.CosNaming - contains interfaces and classes which provides the naming service for Java IDL
  - org.omg.CORBA.ORB - contains interfaces and classes which provides APIs for the Object Request Broker.

# Java IDL Tools



- Java IDL provides a set of tools needed for developing a CORBA application:
  - **idlj** - the IDL-to-Java compiler (called `idl2java` in Java 1.2 and before)
  - **orbd** - a server process which provides Naming Service and other services
  - **servertool** – provides a command-line interface for application programmers to register/unregister an object, and startup/shutdown a server.
  - **tnameserv** – an older Transient Java IDL Naming Service whose use is now discouraged.

# A Java IDL Application example



- **Compiling the IDL file (using Java)**
- The IDL file should be placed in a directory dedicated to the application.
- The file is compiled using the compiler `idlj` using a command as follows:  
***idlj -fall Hello.idl***
- The **-fall** command option is necessary for the compiler to generate all the files needed.
- In general, the files can be found in a subdirectory named <some name>App when an interface file named <some name>.idl is compiled.



# A Java IDL Application example



- If the compilation is successful, the following files can be found in a HelloApp subdirectory:
  - ***HelloOperations.java***
  - ***Hello.java***
  - ***HelloHelper.java***
  - ***HelloHolder.java***
  - ***\_HelloStub.java***
  - ***HelloPOA.java***
- These files require no modifications.
- The CORBA Interface file Hello.idl
  - 01. module HelloApp***
  - 02. {***
  - 03. interface Hello***
  - 04. {***
  - 05. string sayHello();***
  - 06. oneway void shutdown();***
  - 07. };***
  - 08. };***

# The \*Operations.java file



- There is a file HelloOperations.java found in HelloApp/ after you compiled using idlj
- It is known as a Java operations interface in general
- It is a Java interface file that is equivalent to the CORBA IDL interface file (Hello.idl)
- You should look at this file to make sure that the method signatures correspond to what you expect.

**HelloApp/HelloOperations.java :** The file contains the methods specified in the original IDL file: in this case the methods *sayHello( )* and *shutdown()*.

```
01. package HelloApp;
04. /**
05. * HelloApp/HelloOperations.java
06. * Generated by the IDL-to-Java compiler (portable),
07. * version "3.1" from Hello.idl
08. */
09.
10. public interface HelloOperations
11. {
12. String sayHello ();
13. void shutdown ();
14. } // interface HelloOperations
```

**HelloApp/Hello.java:** The signature interface file combines the characteristics of the Java operations interface (HelloOperations.java) with the characteristics of the CORBA classes that it extends.

```
01. package HelloApp;
03. /**
04. * HelloApp/Hello.java
05. * Generated by the IDL-to-Java compiler (portable),
06. * version "3.1" from Hello.idl
07. */
09. public interface Hello extends HelloOperations,
10. org.omg.CORBA.Object,
11. org.omg.CORBA.portable.IDLEntity
12. { ...
13. } // interface Hello
```

# The \*Operations.java file



## ➤ **HelloHelper.java, the Helper class:**

- The Java class HelloHelper provides auxiliary functionality needed to support a CORBA object in the context of the Java language.
- In particular, a method, narrow, allows a CORBA object reference to be cast to its corresponding type in Java, so that a CORBA object may be operated on using syntax for Java object.

## ➤ **HelloHolder.java, the Holder class:**

- The Java class called HelloHolder holds (contains) a reference to an object that implements the Hello interface.
- The class is used to handle an out or an inout parameter in IDL in Java syntax ( In IDL, a parameter may be declared to be out if it is an output argument, and inout if the parameter contains an input value as well as carries an output value.)

## ➤ **\_HelloStub.java:**

- The Java class HelloStub is the stub file, the client-side proxy, which interfaces with the client object.
- It extends org.omg.CORBA.portable.ObjectImpl and implements the Hello.java interface.



# The \*Operations.java file

## ➤ **HelloPOA.java, the server skeleton:**

- The Java class HelloImplPOA is the skeleton, the server-side proxy, combined with the portable object adapter.
- It extends `org.omg.PortableServer.Servant`, and implements the `InvokeHandler` interface and the `HelloOperations` interface.

## ➤ **Server Side classes:**

- On the server side, two classes need to be provided: the servant and the server.
- The servant, `HelloImpl`, is the implementation of the `Hello IDL` interface; each `Hello` object is an instantiation of this class.

# The Application



## The Servant - HelloApp/HelloImpl.java

// The servant -- object implementation -- for the Hello example. Note that this is a subclass of HelloPOA, whose source file is generated from the // compilation of Hello.idl using j2idl.

```
import HelloApp.*;
import org.omg.CosNaming.*;
import java.util.Properties; ...
class HelloImpl extends HelloPOA {
 private ORB orb;
 public void setORB(ORB orb_val) {
 orb = orb_val;
 }
 // implement sayHello() method
 public String sayHello() {
 return "\nHello world !!\n";
 }

 // implement shutdown() method
 public void shutdown() {
 orb.shutdown(false);
 }
} //end class
```

## The Servant - HelloApp/HelloServer.java

```
public class HelloServer {
 public static void main(String args[]) {
 try{
 // create and initialize the ORB
 ORB orb = ORB.init(args, null);
 // get reference to rootpoa & activate the POAManager
 POA rootpoa =
 (POA)orb.resolve_initial_references("RootPOA");
 rootpoa.the_POAManager().activate();
 // create servant and register it with the ORB
 HelloImpl helloImpl = new HelloImpl();
 helloImpl.setORB(orb);
 // get object reference from the servant
 org.omg.CORBA.Object ref =
 rootpoa.servant_to_reference(helloImpl);
 // and cast the reference to a CORBA reference
 Hello href = HelloHelper.narrow(ref);
```

# The Application



## The Servant - HelloApp/HelloServer.java

```
// and cast the reference to a CORBA reference
Hello href = HelloHelper.narrow(ref);
// get the root naming context
// NameService invokes the transient name service
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
// Use NamingContextExt, which is part of the
// Interoperable Naming Service (INS) specification.
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
// bind the Object Reference in Naming
String name = "Hello";
NameComponent path[] = ncRef.to_name(name);
ncRef.rebind(path, href);
System.out.println ("HelloServer ready and waiting ...");
// wait for invocations from clients
orb.run();
```

# The Object Client Application



- A client program can be a Java application, an applet, or a servlet.
- The client code is responsible for creating and initializing the ORB, looking up the object using the Interoperable Naming Service, invoking the narrow method of the Helper object to cast the object reference to a reference to a Hello object implementation, and invoking remote methods using the reference.
- The object's sayHello method is invoked to receive a string, and the object's shutdown method is invoked to deactivate the service.

## The Servant - HelloApp/HelloClient.java

```
// A sample object client application.
import HelloApp.*;
import org.omg.CosNaming.*; ...
public class HelloClient{
 static Hello helloImpl;
 public static void main(String args[]){
 try{
 ORB orb = ORB.init(args, null);
 org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
 NamingContextExt ncRef =
 NamingContextExtHelper.narrow(objRef);
 helloImpl =
 HelloHelper.narrow(ncRef.resolve_str("Hello"));
 System.out.println(helloImpl.sayHello());
 helloImpl.shutdown();
 }
 }
}
```

# Compiling and Running a Java IDL application



1. Create and compile the Hello.idl file on the server machine:  
`idlj -fall Hello.idl`
2. Copy the directory containing Hello.idl (including the subdirectory generated by idlj) to the client machine.
3. In the HelloApp directory on the client machine: create HelloClient.java. Compile the \*.java files, including the stubs and skeletons (which are in the directory HelloApp):  
`javac *.java HelloApp/*.java`
4. In the HelloApp directory on the server machine:
  1. Create HelloServer.java. Compile the .java files:
  2. `javac *.java HelloApp/*.java`
5. On the server machine: Start the Java Object Request Broker Daemon, orbd, which includes a Naming Service.  
To do this on Unix: `orbd -ORBInitialPort 1050 -ORBInitialHost servermachinename&`  
To do this on Windows:  
`start orbd -ORBInitialPort 1050 -ORBInitialHost servermachinename`



# Compiling and Running a Java IDL application



5. On the server machine, start the Hello server, as follows:

```
java HelloServer -ORBInitialHost <nameserver host name> -ORBInitialPort 1050
```

6. On the client machine, run the Hello application client. From a DOS prompt or shell, type:

```
java HelloClient -ORBInitialHost nameserverhost -ORBInitialPort 1050
```

all on one line.

Note that nameserverhost is the host on which the IDL name server is running. In this case, it is the server machine.

7. Kill or stop orbd when finished. The name server will continue to wait for invocations until it is explicitly stopped.
8. Stop the object server

# RMI (Remote Method Invocation)



- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.
- The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.
- Understanding stub and skeleton
  - RMI uses stub and skeleton object for communication with the remote object.
  - A **remote object** is an object whose method can be invoked from another JVM.
  - Let's understand the stub and skeleton objects:

# RMI (Remote Method Invocation)



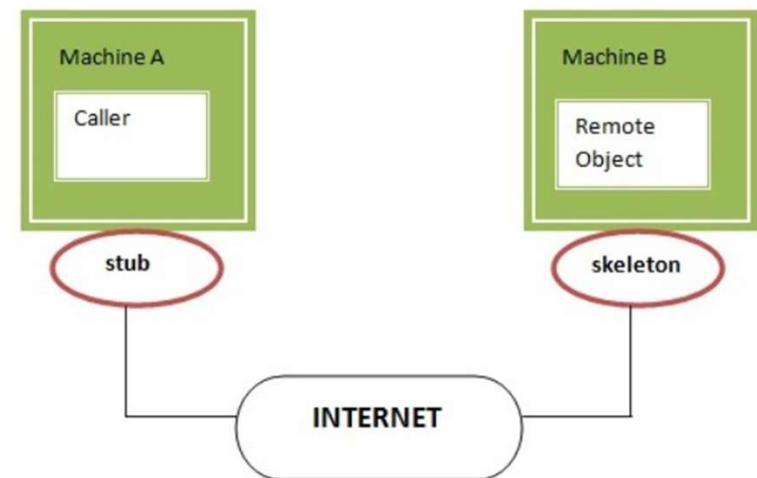
- Stub

- The stub is an object, acts as a gateway for the client side.
- All the outgoing requests are routed through it.
- It resides at the client side and represents the remote object.
- When the caller invokes method on the stub object, it does the following tasks:
  - It initiates a connection with remote Virtual Machine (JVM),
  - It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
  - It waits for the result
  - It reads (unmarshals) the return value or exception, and
  - It finally, returns the value to the caller.

# RMI (Remote Method Invocation)



- **Skeleton**
- The skeleton is an object, acts as a gateway for the server side object.
- All the incoming requests are routed through it.
- When the skeleton receives the incoming request, it does the following tasks:
  - It reads the parameter for the remote method
  - It invokes the method on the actual remote object, and
  - It writes and transmits (marshals) the result to the caller.



# RMI (Remote Method Invocation)



- Java RMI Example
- The is given the 6 steps to write the RMI program.
  1. Create the remote interface
  2. Provide the implementation of the remote interface
  3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
  4. Start the registry service by rmiregistry tool
  5. Create and start the remote application
  6. Create and start the client application

# RMI (Remote Method Invocation)



## RMI Example

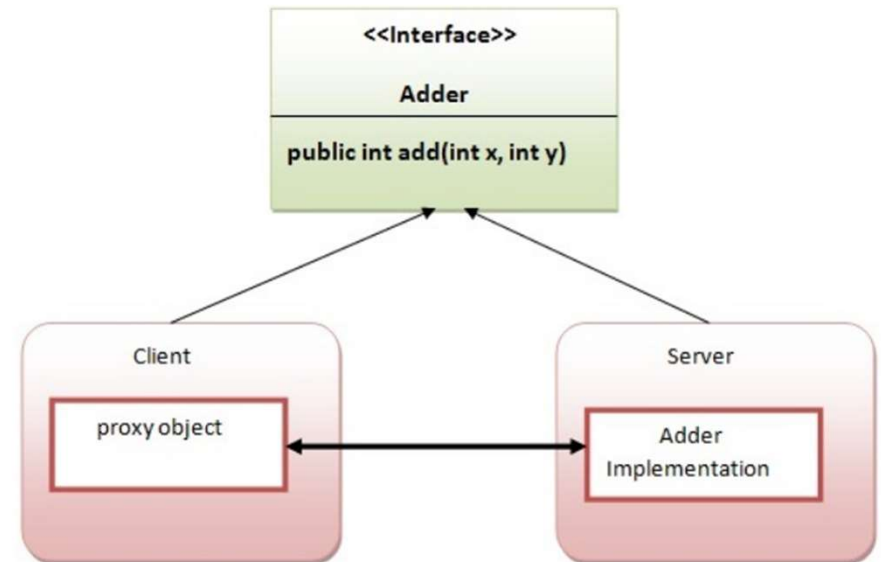
In this example, we have followed all the 6 steps to create and run the rmi application.

The client application need only two files, remote interface and client application.

In the rmi application, both client and server interacts with the remote interface.

The client application invokes methods on the proxy object, RMI sends the request to the remote JVM.

The return value is sent back to the proxy object and then to the client application.



# RMI (Remote Method Invocation)



## 1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface.

Here, we are creating a remote interface that extends the Remote interface.

There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;
public interface Adder extends Remote{
 public int add(int x,int y)throws RemoteException;
}
```

# RMI (Remote Method Invocation)



- 2) Provide the implementation of the remote interface
- Now provide the implementation of the remote interface.
- For providing the implementation of the Remote interface, we need to
- Either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class
- In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.
- **import** java.rmi.\*;
- **import** java.rmi.server.\*;
- **public class** AdderRemote **extends** UnicastRemoteObject **implements** Adder{
- AdderRemote()**throws** RemoteException{
- **super**();
- }
- **public int** add(**int** x,**int** y){**return** x+y;}
- }



# RMI (Remote Method Invocation)



- 3) create the stub and skeleton objects using the rmic tool.
- Next step is to create stub and skeleton objects using the rmi compiler.
- The rmic tool invokes the RMI compiler and creates stub and skeleton objects.
- `rmic AdderRemote`
- 4) Start the registry service by the rmiregistry tool
- Now start the registry service by using the rmiregistry tool.
- If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.
- `rmiregistry 5000`



# RMI (Remote Method Invocation)

- 5) Create and run the server application
- Now rmi services need to be hosted in a server process.
- The Naming class provides methods to get and store the remote object.
- The Naming class provides 5 methods.

|                                                                                                                                                                                |                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
| <pre>public static java.rmi.Remote lookup(java.lang.String) throws<br/>java.rmi.NotBoundException, java.net.MalformedURLException,<br/>java.rmi.RemoteException;</pre>         | It returns the reference of the remote object.                                |
| <pre>public static void bind(java.lang.String, java.rmi.Remote) throws<br/>java.rmi.AlreadyBoundException, java.net.MalformedURLException,<br/>java.rmi.RemoteException;</pre> | It binds the remote object with the given name.                               |
| <pre>public static void unbind(java.lang.String) throws java.rmi.RemoteException,<br/>java.rmi.NotBoundException, java.net.MalformedURLException;</pre>                        | It destroys the remote object which is bound with the given name.             |
| <pre>public static void rebind(java.lang.String, java.rmi.Remote) throws<br/>java.rmi.RemoteException, java.net.MalformedURLException;</pre>                                   | It binds the remote object to the new name.                                   |
| <pre>public static java.lang.String[] list(java.lang.String) throws<br/>java.rmi.RemoteException, java.net.MalformedURLException;</pre>                                        | It returns an array of the names of the remote objects bound in the registry. |

# RMI (Remote Method Invocation)



In this example, we are binding the remote object by the name sonoo.

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

# RMI (Remote Method Invocation)



- 6) Create and run the client application
- At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object.
- In this example, we are running the server and client applications, in the same machine so we are using localhost.
- If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;
public class MyClient{
 public static void main(String args[]){
 try{
 Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
 System.out.println(stub.add(34,4));
 }catch(Exception e){}
 }
}
```

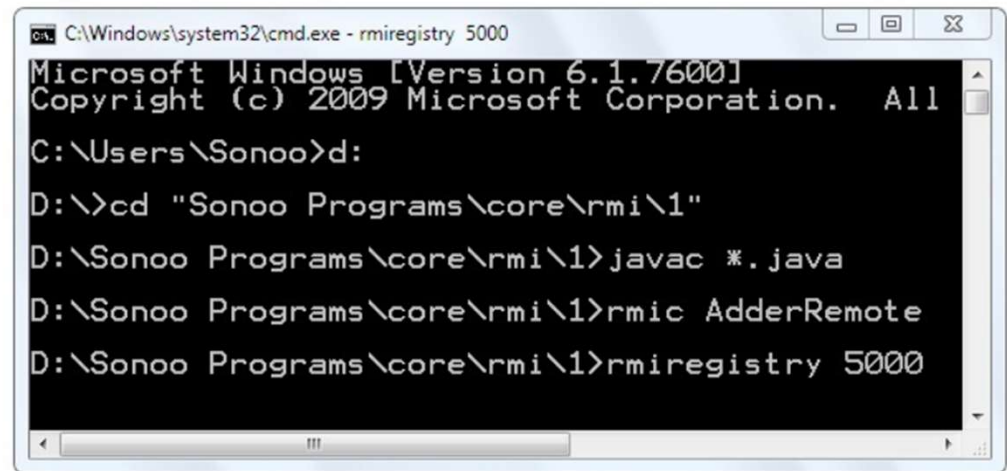
# RMI (Remote Method Invocation)



For running **this** rmi example,

- 1) compile all the java files  
`javac *.java`
- 2) create stub and skeleton object by rmic tool  
`rmic AdderRemote`
- 3) start rmi registry in one command prompt  
`rmiregistry 5000`
- 4) start the server in another command prompt  
`java MyServer`
- 5) start the client application in another command prompt  
`java MyClient`

Output of this RMI example



```
C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```

# RMI (Remote Method Invocation)



Output of this RMI example

```
C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```

```
C:\Windows\system32\cmd.exe - java MyServer
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyClient
38
D:\Sonoo Programs\core\rmi\1>
```

# Summary



1. The key topics introduced with CORBA are:
  1. The basic CORBA architecture and its emphasis on object interoperability and platform independence
  2. Object Request Broker (ORB) and its functionalities
  3. The Inter-ORB Protocol (IIOP) and its significance
  4. CORBA object reference and the Interoperable Object Reference (IOR) protocol
  5. CORBA Naming Service and the Interoperable Naming Service (INS)
  6. Standard CORBA object services and how they are provided.
  7. Object adapters, portable object Adapters (POA) and their significance.
2. The key topics introduced with Java IDL are:
  1. It is part of the Java TM 2 Platform, Standard Edition (J2SE)
  2. Java packages are provided which contain interfaces and classes for CORBA support
  3. Tools provided for developing a CORBA application include idlj (the IDL compiler) and orbd (the ORB and name server)
  4. An example application Hello
  5. Steps for compiling and running an application.
  6. Client callback is achievable.
3. CORBA toolkits and Java RMI are comparable and alternative technologies that provide distributed objects. An application may be implemented using either technology. However, there are tradeoffs between the two.



# Thank You





STOP REEC