The Common Object Request Broker Architecture (CORBA)

based on slides by M. L. Liu

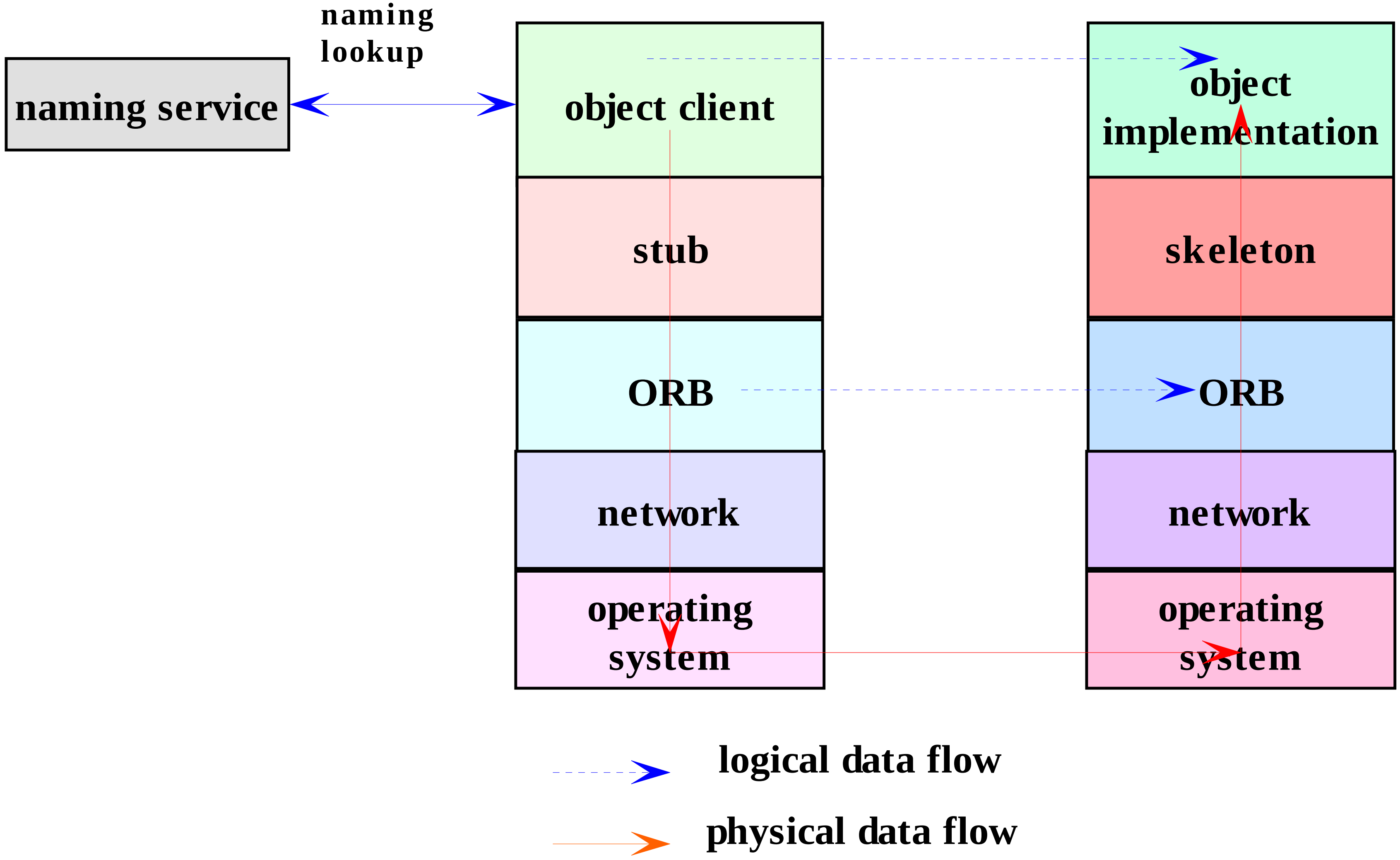
CORBA

- 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 heterogenous environment, where objects can be implemented in different programming language and/or deployed on different platforms

CORBA vs. Java 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 langugage only.
  - CORBA is an architecture that was developed by the Object Management Group (OMG), an industrial consortium.

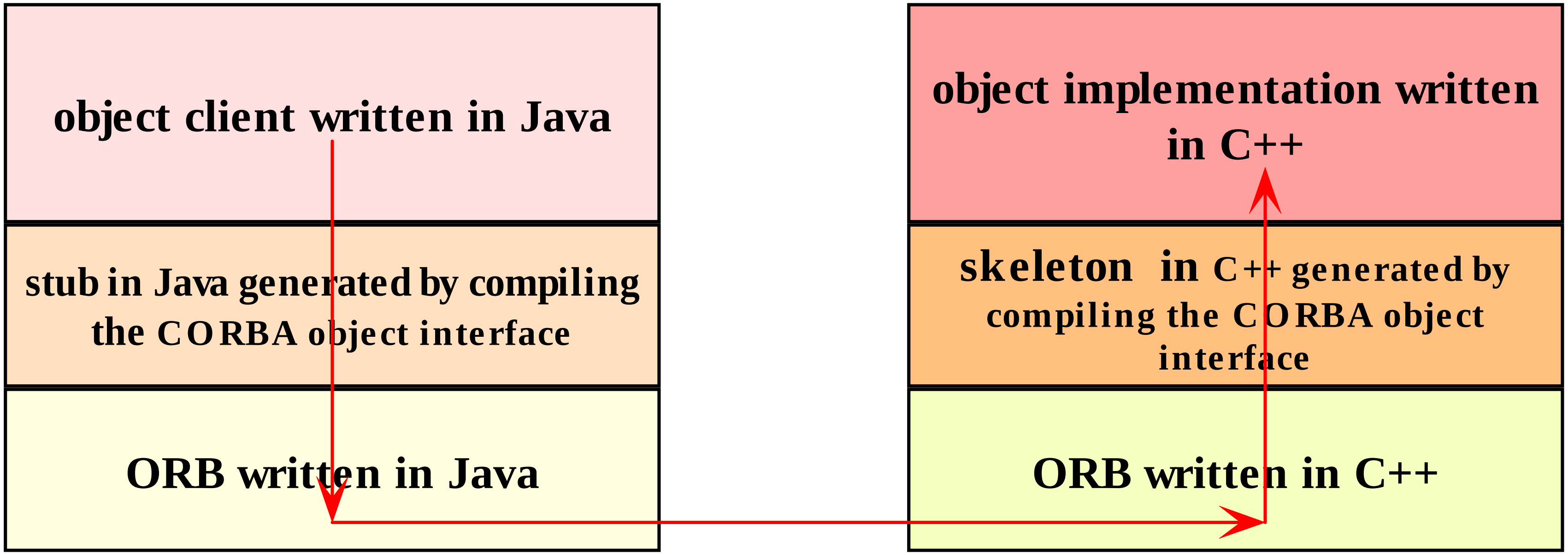
The Basic Architecture



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.
- For each of these languages, OMG has a standardized mapping from CORBA IDL to the programming language, so that 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.

Cross-language CORBA application



object client written in Java

stub in Java generated by compiling the CORBA object interface

ORB written in Java

object implementation written in C++

skeleton in C++ generated by compiling the CORBA object interface

ORB written in C++

ORB Core Feature Matrix

ORB Core Feature Matrix

http://www.jetpen.com/~ben/corba/orbmatrix.html

Inter-ORB Protocols

- To allow ORBs to be interoperable, the OMG 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.

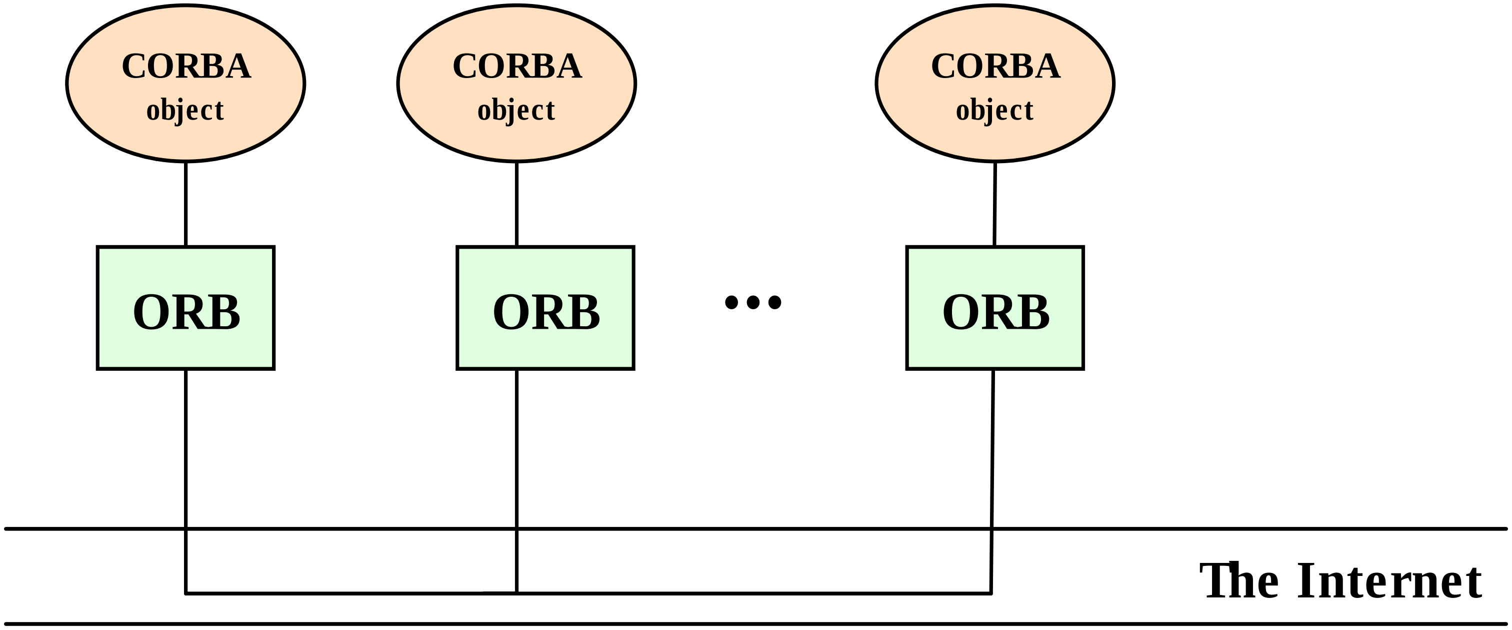
Inter-ORB Protocols

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. The messages allow clients to send requests to object servers and receive replies. 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



ORB products

There are a large number of proprietary as well as experimental ORBs available:

(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 ORBs

Object Servers and Object Clients

- 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.

CORBA Object References

- 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.

Interoperable Object Reference (IOR)

- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the ***Interoperable Object Reference*** (***IOR***) protocol.
- 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*** (***IOR***s).

Interoperable Object Reference (IOR)

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:000000000000000d49444c3a677269643a312e3000000
0000000000100000000000000004c0001000000000015756c74
72612e6475626c696e2e696f6e612e6965000009630000002
83a5c756c7472612e6475626c696e2e696f6e612e69653a67
7269643a303a3a49523a67726964003a
```

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

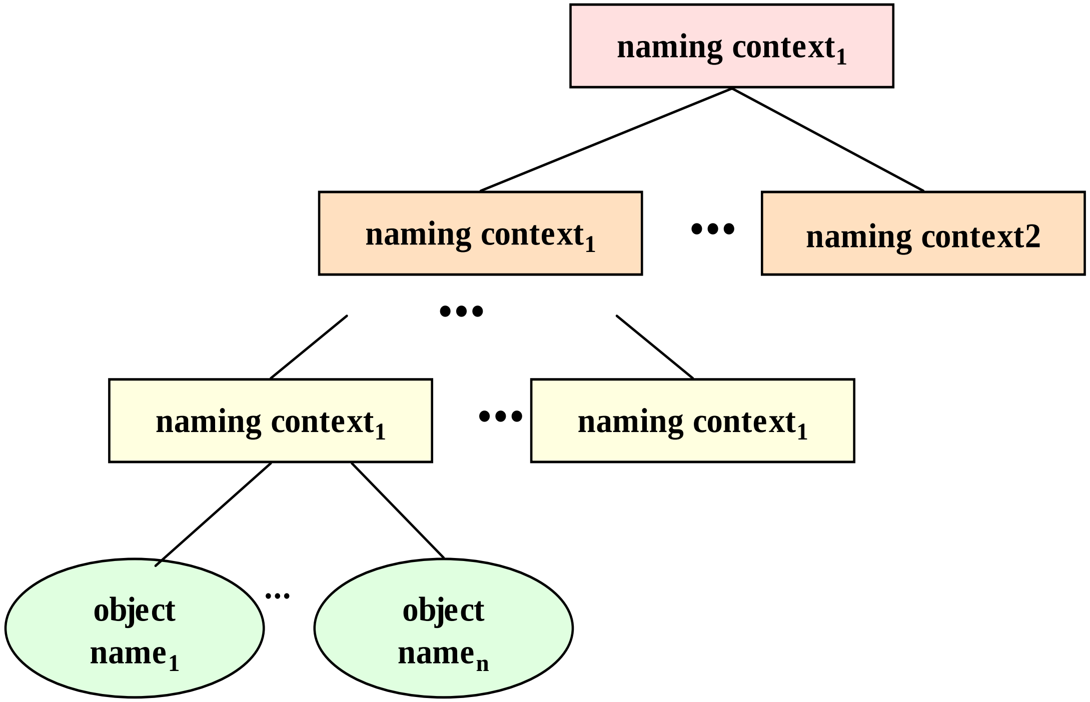
- CORBA specifies a generic directory service. The ***Naming Service*** serves as a directory for CORBA objects, and, as such, is platform independent and programming language independent.
- The Naming Service permits ORB-based clients to obtain references to objects they wish to use. It allows names 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.)
- 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.

CORBA Naming Service

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



A Naming Context

- A naming context correspond to a folder or directory in a file hierarchy, while object names corresponds 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.
- Naming contexts and name bindings are created using methods provided in the Naming Service interface.

A CORBA object name

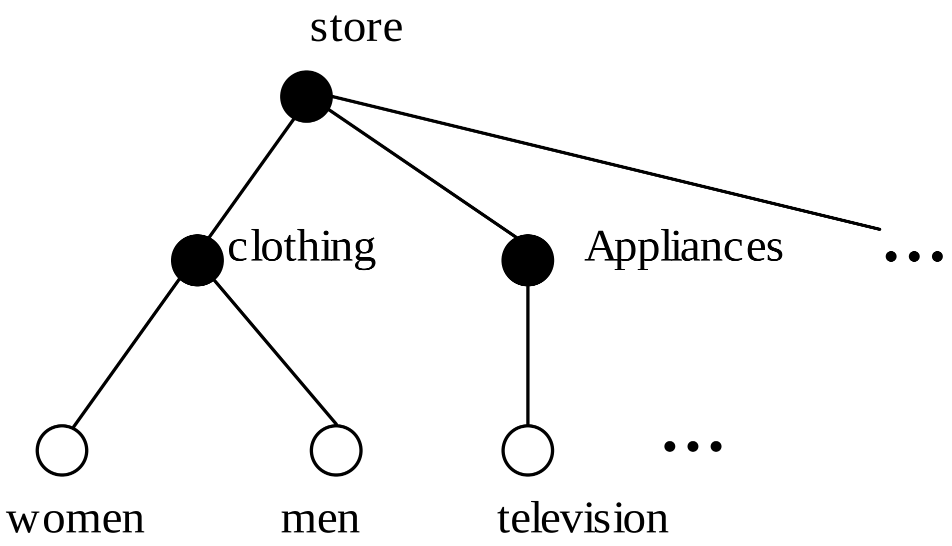
The syntax for an object name is as follows:

```
<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.



Interoperable Naming Service

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.

CORBA Object Services

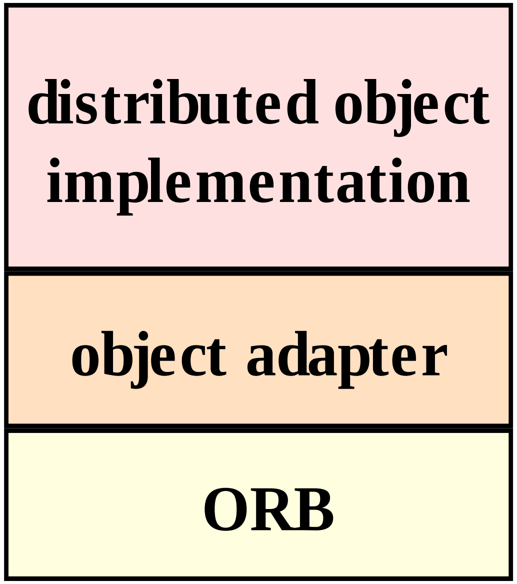
CORBA specify services commonly needed in distributed applications, some of which are:

- ***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);
- ***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.

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**.



distributed object implementation

object adapter

ORB

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.
- The adapter interacts with the object implementation’s skeleton, which performs data marshalling and invoke the appropriate method in the object.

The ***Portable Object Adapter***

- 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.

The Java IDL (Java 1.4 version)

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 olderTransient Java IDL Naming Service whose use is now discouraged.

A Java IDL application example

The CORBA Interface file **Hello.idl**

```
module HelloApp
{
  interface Hello
  {
  string sayHello();
  oneway void shutdown();
  };
};
```

Compiling the IDL file (using Java 1.4)

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.

If the compilation is successful, the following files can be found in a ***HelloApp*** subdirectory:

HelloOperations.java

HelloHelper.java

_HelloStub.java

Hello.java

HelloHolder.java

HelloPOA.java

These files require no modifications.

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***().

```
package HelloApp;
package HelloApp;
/**
* HelloApp/HelloOperations.java
* Generated by the IDL-to-Java compiler (portable),
*  version "3.1" from Hello.idl
*/

public interface HelloOperations
{
  String sayHello ();
  void shutdown ();
} // 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.

```
package HelloApp;
/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable),
* version "3.1" from Hello.idl
*/
public interface Hello extends HelloOperations,
  org.omg.CORBA.Object,
  org.omg.CORBA.portable.IDLEntity
{ …
} // interface Hello
```

HelloHelper.java, the Helper class

- The Java class `HelloHelper (Figure 7d)` 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 (Figure 7e)` 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*** `(Figure 7e)` 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.

HelloPOA.java, the server skeleton

- The Java class ***HelloImplPOA*** (Figure 7f) 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.

The application

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 Servant - HelloApp/HelloImpl.java

```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 server - 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);
```

HelloApp/HelloServer.java - continued

```
// 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.

```
// 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
```

Compiling and Running a Java IDL application

4. In the ***HelloApp*** directory on the server machine:

- Create ***HelloServer.java***. Compile the `.java` files:

  javac *.java HelloApp/*.java

- 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.

Compiling and Running a Java IDL application

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.

Summary-1

- You have been introduced to
  - the **Common Object Request Broker Architecture** (**CORBA**), and
  - a specific CORBA facility based on the architecture: **Java IDL**

Summary-2

- The key topics introduced with CORBA are:
  - The basic CORBA architecture and its emphasis on object interoperability and platform independence
  - Object Request Broker (ORB) and its functionalities
  - The Inter-ORB Protocol (IIOP) and its significance
  - CORBA object reference and the Interoperable Object Reference (IOR) protocol
  - **CORBA Naming Service** and the **Interoperable Naming Service (INS)**
  - Standard CORBA **object services** and how they are provided.
  - **Object adapters**, portabl**e object Adapters** (**POA**) and their significance.

Summary-3

- The key topics introduced with **Java IDL** are:
  - It is part of the Java ^{TM} 2 Platform, Standard Edition (J2SE)
  - Java packages are provided which contain interfaces and classes for CORBA support
  - Tools provided for developing a CORBA application include ***idlj*** (the IDL compiler) and ***orbd*** (the ORB and name server)
  - An example application Hello
  - Steps for compiling and running an application.
  - Client callback is achievable.
- CORBA tookits and Java RMI are comparable and alternative technologies that provide distributed objects. An applicaton may be implemented using either technology. However, there are tradeoffs between the two.