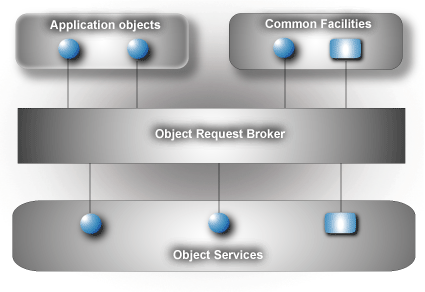
**What is CORBA?**

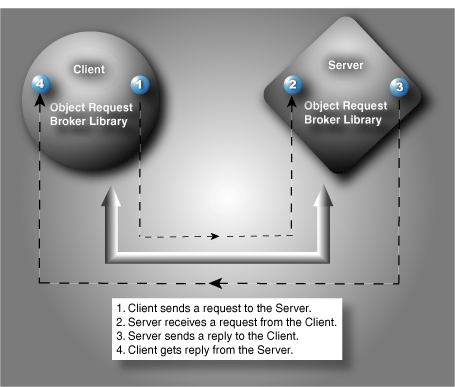
The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects. CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages, and operating systems. CORBA is essentially a design specification for an Object Request Broker (ORB), where an ORB provides the mechanism required for distributed objects to communicate with one another, whether locally or on remote devices, written in different languages, or at different locations on a network.

The CORBA Interface Definition Language, or IDL, allows the development of language and location-independent interfaces to distributed objects. Using CORBA, application components can communicate with one another no matter where they are located, or who has designed them. CORBA provides the location transparency to be able to execute these applications.

CORBA is often described as a "software bus" because it is a software-based communications interface through which objects are located and accessed. The illustration below identifies the primary components seen within a CORBA implementation.



Data communication from client to server is accomplished through a well-defined object-oriented interface. The Object Request Broker (ORB) determines the location of the target object, sends a request to that object, and returns any response back to the caller. Through this object-oriented technology, developers can take advantage of features such as inheritance, encapsulation, polymorphism, and runtime dynamic binding. These features allow applications to be changed, modified and re-used with minimal changes to the parent interface. The illustration below identifies how a client sends a request to a server through the ORB:



**Interface Definition Language (IDL)**  
A cornerstone of the CORBA standards is the Interface Definition Language. IDL is the OMG standard for defining language-neutral APIs and provides the platform-independent delineation of the interfaces of distributed objects. The ability of the CORBA environments to provide consistency between clients and servers in heterogeneous environments begins with a standardized definition of the data and operations constituting the client/server interface. This standardization mechanism is the IDL, and is used by CORBA to describe the interfaces of objects.

IDL defines the modules, interfaces and operations for the applications and is not considered a programming language. The various programming languages, such as Ada, C++, C#, or Java, supply the implementation of the interface via standardized IDL mappings.

**Application Development Using ORBexpress**  
The basic steps for CORBA development can be seen in the illustration below. This illustration provides an overview of how the IDL is translated to the corresponding language (in this example, C++), mapped to the source code, compiled, and then linked with the ORB library, resulting in the client and server implementation.

The basic steps for CORBA development include:

 **Create the IDL to Define the Application Interfaces**  
The IDL provides the operating system and programming language independent interfaces to all services and components that are linked to the ORB. The IDL specifies a description of any services a server component exposes to the client. The term "IDL Compiler" is often used, but the IDL is actually translated into a programming language.

** **Translate the IDL**  
An IDL translator typically generates two cooperative parts for the client and server implementation, stub code and skeleton code. The stub code generated for the interface classes is associated with a client application and provides the user with a well-defined Application Programming Interface (API). In this example, the IDL is translated into C++.

 **Compile the Interface Files**Once the IDL is translated into the appropriate language, C++ in this example, these interface files are compiled and prepared for the object implementation.

 **Complete the Implementation**If the implementation classes are incomplete, the spec and header files and complete bodies and definitions need to be modified before passing through to be compiled. The output is a complete client/server implementation.

** **Compile the Implementation**Once the implementation class is complete, the client interfaces are ready to be used in the client application and can be immediately incorporated into the client process. This client process is responsible for obtaining an object reference to a specific object, allowing the client to make requests to that object in the form of a method call on its generated API.

 **Link the Application**Once all the object code from steps three and five have been compiled, the object implementation classes need to be linked to the C++ linker. Once linked to the ORB library, in this example, ORBexpress, two executable operations are created, one for the client and one for the server.

 **Run the Client and Server**The development process is now complete and the client will now communicate with the server. The server uses the object implementation classes allowing it to communicate with the objects created by the client requests.

In its simplest form, the server must perform the following:

* Create the required objects.
* Notify the CORBA environment that it is ready to receive client requests.
* Process client requests by dispatching the appropriate servant.

**Interoperability**  
The first version of CORBA provided the IDL and standard mappings to just a few languages, and as the CORBA standard has matured, CORBA 2.0 added more language bindings (particularly C++ and Java) as well as General Inter-ORB Protocol (GIOP). When a client calls a CORBA operation, the client ORB sends a GIOP message to the server. The server ORB converts this request into a call on the server object and then returns the results in a GIOP reply. This standard transfer syntax, specified by the Object Management Group, allows the interoperability of ORB-to-ORB interaction and is designed to work over any transport protocol meeting a minimal set of assumptions.

When GIOP is sent over TCP/IP, it is called Internet Inter ORB Protocol (IIOP). IIOP is designed to allow different ORB vendors to interoperate with one another. An example of this interoperability occurs when there is communication between an enterprise designed ORB, and a smaller real-time application, utilizing a real-time ORB.

[**Object Management Group (OMG)**](http://www.omg.org/)  
The OMG is a non-profit consortium created in 1989 to promote the theory and practice of object technology for the development for distributed operating systems. The goal is to provide a common architectural framework for object-oriented applications based on widely available interface specifications. With a membership of over 800 members, representing large and small companies within the computer industry, OMG leads the specification development efforts of CORBA, OMG IDL, IIOP, OMA, UML, MOF, and CWM specifications.

The OMG does not produce software or implementation guidelines, only the specifications to which OMG members respond to in Request For Information (RFI) and Requests for Proposals (RFP). By managing these specifications, the OMG supports the adoption process for the member companies interested in advancing the uses and applications of distributed object-oriented computing.

# CORBA-Example

A simple CORBA implementation using Java

## Instructions

### 1. Write the IDL file

The IDL file defines the interface that will be used by the client and server for communicating and passing objects.  
When the IDL file gets compiled, it will produce a number of files, known as the stub and skeleton:

* The stub is used by the client to communicate with the server
* The skeleton is used by the server to communicate with the client
* The stub and skeleton communicate with the ORB server to facilitate the remote procedure call

The module in the IDL file will correspond to the package and directory in which the Java code will be generated

Echo.idl

module EchoApp {

interface Echo {

string echoString();

};

};

### 2. Generate the stub and skeleton code

There is an idlj program that comes with the JDK for generating the stub and skeleton code in Java

idlj –fall sum.idl

The following files are generated by the idlj program:

* \_EchoStub.java
* Echo.java
* EchoHelper.java
* EchoHolder.java
* EchoOperations.java
* EchoPOA.java

### 3. Write the server code

The server program will inherit from the EchoPOA class that was generated as part of the idlj program.  
The EchoPOA class implements the EchoOperations interface

* This interface contains the methods we defined in the Echo.idl file, but standardized to Java.

We create an [EchoServer.java](https://github.com/johnngugi/CORBA-Example/blob/master/src/EchoServer.java) class that extends the abstract EchoPoa class and then implement the methods contained in it

We create a main method in [Server.java](https://github.com/johnngugi/CORBA-Example/blob/master/src/Server.java) to communicate with the object request broker (ORB), registering the server with the ORB so clients are able to find it.

### 4. Write the client code

The client program [Client.java](https://github.com/johnngugi/CORBA-Example/blob/master/src/Client.java) needs to get a reference to the ORB then resolve the name of the server object it would like to invoke.

* This is ECHO-SERVER in this case

After getting an instance of a Server object from the server, it can invoke methods on it just like a method within its own JVM.

### 5. Compile the code

1. Compile the stub and skeleton from the directory that contains the IDL file.

Windows

javac EchoApp\\*.java

Linux

javac EchoApp/\*.java

1. Generate a JAR file from the compiled stub and skeleton.

Windows

jar cvf echoapp.jar EchoApp\\*.class

Linux

jar cvf echoapp.jar EchoApp/\*.class

1. Compile the server and client classes

Windows

javac -classpath .;echoapp.jar Server.java EchoServer.java Client.java

Linux

javac -classpath .:echoapp.jar Server.java EchoServer.java Client.java

### 6. Running the application

1. Start the ORB server

orbd -ORBInitialPort 1050 -ORBInitialHost localhost

1. Start the server application

java Server -ORBInitialPort 1050 -ORBInitialHost localhost

1. Start the client application

java Client -ORBInitialPort 1050 -ORBInitialHost localhost

If everything was compiled correctly the output should be:

Hello World!!!!!!!