```
myserver.host.com 45
```

- For Solaris OS or Linux:

```
java -cp
/home/distributedcomputing/workspace/src;
/home/distributedcomputing/workspace/classes/calculate.jar -
Djava.rmi.server.codebase=http://myserver/~client/classes/ -
Djava.security.policy=client.policy client.CalculateClient
myserver.host.com 45
```

Common Object Request Broker Architecture (CORBA)

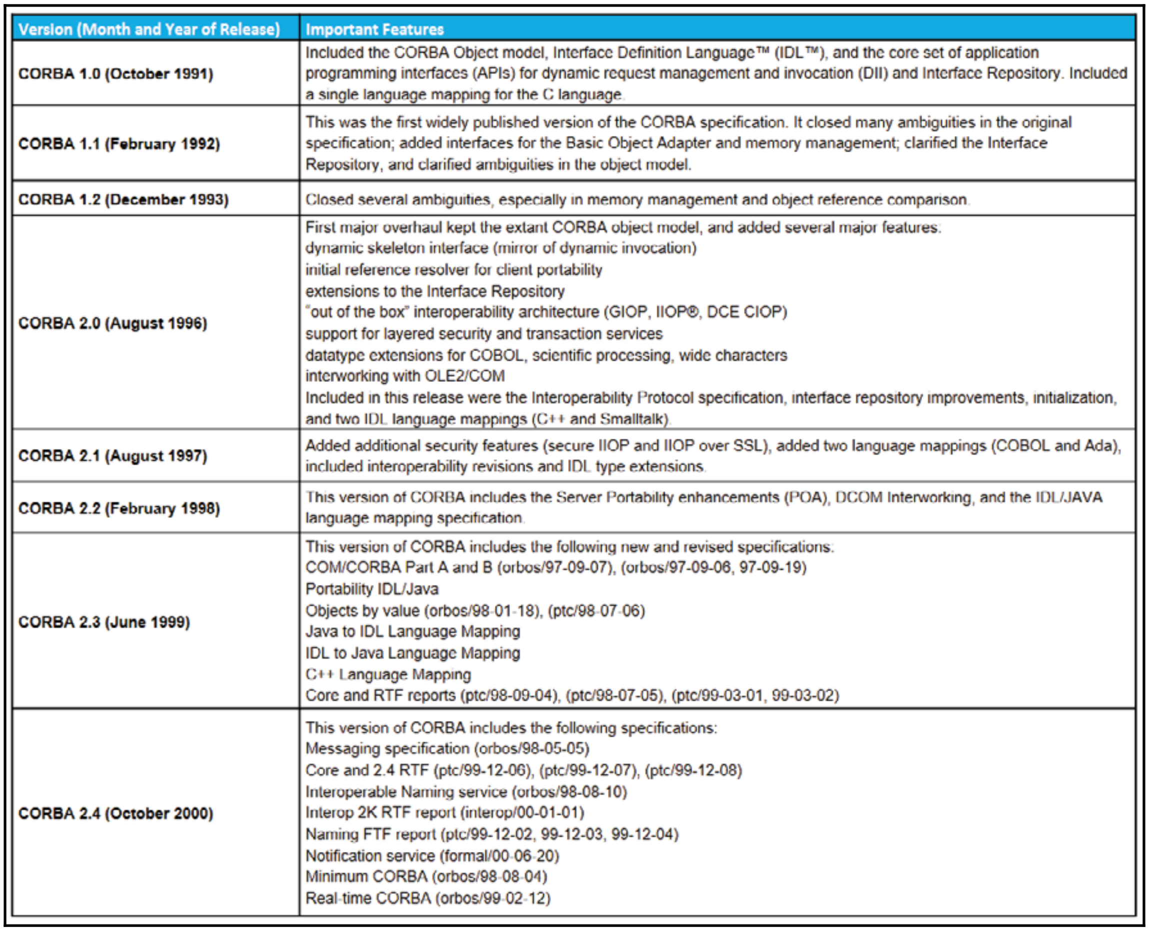
CORBA is the acronym for Common Object Request Broker Architecture. It is an open source, vendor-independent architecture and infrastructure developed by the **Object Management Group** (**OMG**) to integrate enterprise applications across a distributed network. OMG is a nonprofit global software association that sets the distributed object computing standards. CORBA specifications provide guidelines for such integration applications, based on the way they want to interact, irrespective of the technology; hence, all kinds of technologies can implement these standards using their own technical implementations.

When two applications/systems in a distributed environment interact with each other, it is often true that there are quite a few unknowns between those applications/systems, including the technology they are developed in (such as Java/ PHP/ .NET), the base operating system they are running on (such as Windows/Linux), or system configuration (such as memory allocation). They communicate mostly with the help of each other's network address or through a naming service. Due to this, these applications end up with quite a few issues in integration, including content (message) mapping mismatches. As discussed in the socket/remote method invocation programming section, the publisher may not always know the receiver details well.

An application developed based on CORBA standards with standard **Internet Inter-ORB Protocol (IIOP)**, irrespective of the vendor that develops it, should be able to smoothly integrate and operate with another application developed based on CORBA standards through the same or different vendor. This rule is true even if the applications are run on different operating systems or servers or developed with a different technology and connected over the distributed network.

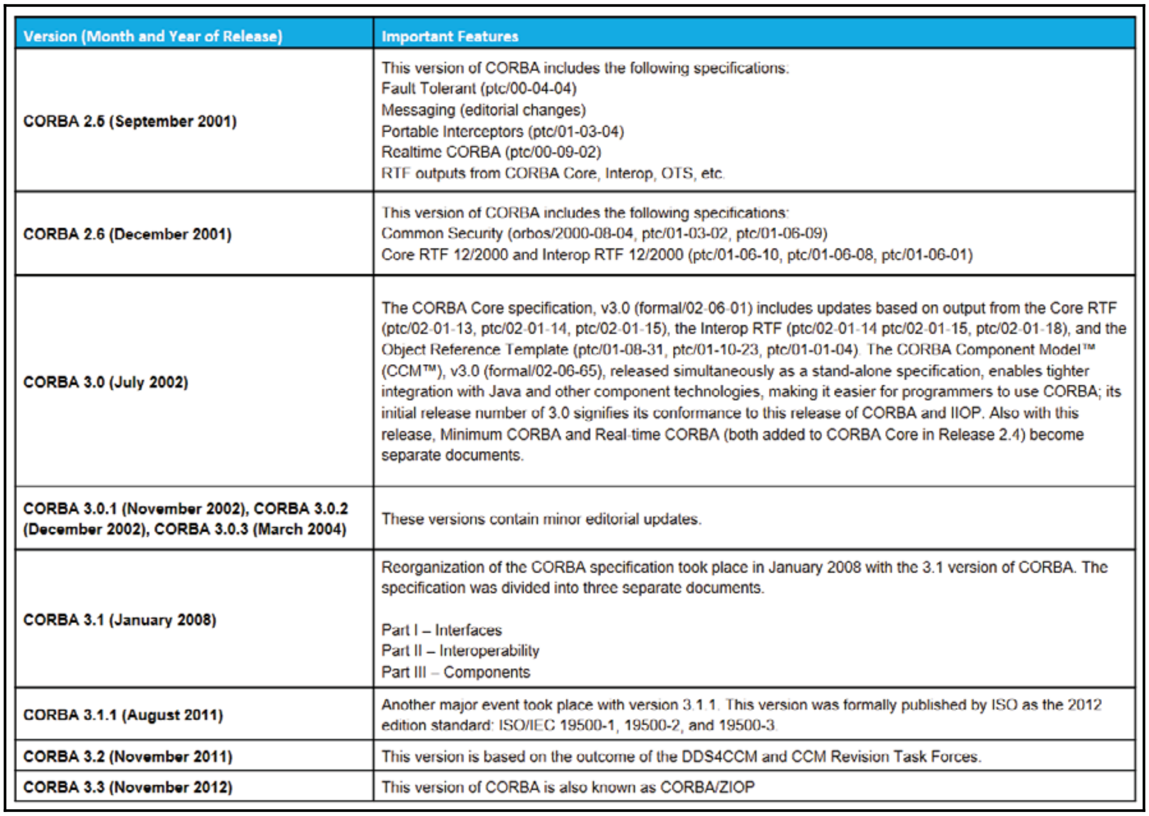
CORBA standards

The following are the CORBA standards published by the OMG group version-wise (as advised in CORBA's official website at `http://www.corba.org/`):



Version (Month and Year of Release)	Important Features
CORBA 1.0 (October 1991)	Included the CORBA Object model, Interface Definition Language™ (IDL™), and the core set of application programming interfaces (APIs) for dynamic request management and invocation (DII) and Interface Repository. Included a single language mapping for the C language.
CORBA 1.1 (February 1992)	This was the first widely published version of the CORBA specification. It closed many ambiguities in the original specification; added interfaces for the Basic Object Adapter and memory management; clarified the Interface Repository, and clarified ambiguities in the object model.
CORBA 1.2 (December 1993)	Closed several ambiguities, especially in memory management and object reference comparison.
CORBA 2.0 (August 1996)	First major overhaul kept the extant CORBA object model, and added several major features: dynamic skeleton interface (mirror of dynamic invocation) initial reference resolver for client portability extensions to the Interface Repository "out of the box" interoperability architecture (GIOP, IIOP®, DCE CIOP) support for layered security and transaction services datatype extensions for COBOL, scientific processing, wide characters interworking with OLE2/COM Included in this release were the Interoperability Protocol specification, interface repository improvements, initialization, and two IDL language mappings (C++ and Smalltalk).
CORBA 2.1 (August 1997)	Added additional security features (secure IIOP and IIOP over SSL), added two language mappings (COBOL and Ada), included interoperability revisions and IDL type extensions.
CORBA 2.2 (February 1998)	This version of CORBA includes the Server Portability enhancements (POA), DCOM Interworking, and the IDL/JAVA language mapping specification.
CORBA 2.3 (June 1999)	This version of CORBA includes the following new and revised specifications: COM/CORBA Part A and B (orbos/97-09-07), (orbos/97-09-06, 97-09-19) Portability IDL/Java Objects by value (orbos/98-01-18), (ptc/98-07-06) Java to IDL Language Mapping IDL to Java Language Mapping C++ Language Mapping Core and RTF reports (ptc/98-09-04), (ptc/98-07-05), (ptc/99-03-01, 99-03-02)
CORBA 2.4 (October 2000)	This version of CORBA includes the following specifications: Messaging specification (orbos/98-05-05) Core and 2.4 RTF (ptc/99-12-06), (ptc/99-12-07), (ptc/99-12-08) Interoperable Naming service (orbos/98-08-10) Interop 2K RTF report (interop/00-01-01) Naming FTF report (ptc/99-12-02, 99-12-03, 99-12-04) Notification service (formal/00-06-20) Minimum CORBA (orbos/98-08-04) Real-time CORBA (orbos/99-02-12)

From 2001 onward, CORBA has given us standards that are related to security as well:



Version (Month and Year of Release)	Important Features
CORBA 2.5 (September 2001)	This version of CORBA includes the following specifications: Fault Tolerant (ptc/00-04-04) Messaging (editorial changes) Portable Interceptors (ptc/01-03-04) Realtime CORBA (ptc/00-09-02) RTF outputs from CORBA Core, Interop, OTS, etc.
CORBA 2.6 (December 2001)	This version of CORBA includes the following specifications: Common Security (orbos/2000-08-04, ptc/01-03-02, ptc/01-06-09) Core RTF 12/2000 and Interop RTF 12/2000 (ptc/01-06-10, ptc/01-06-08, ptc/01-06-01)
CORBA 3.0 (July 2002)	The CORBA Core specification, v3.0 (formal/02-06-01) includes updates based on output from the Core RTF (ptc/02-01-13, ptc/02-01-14, ptc/02-01-15), the Interop RTF (ptc/02-01-14 ptc/02-01-15, ptc/02-01-18), and the Object Reference Template (ptc/01-08-31, ptc/01-10-23, ptc/01-01-04). The CORBA Component Model™ (CCM™), v3.0 (formal/02-06-65), released simultaneously as a stand-alone specification, enables tighter integration with Java and other component technologies, making it easier for programmers to use CORBA; its initial release number of 3.0 signifies its conformance to this release of CORBA and IIOP. Also with this release, Minimum CORBA and Real-time CORBA (both added to CORBA Core in Release 2.4) become separate documents.
CORBA 3.0.1 (November 2002), CORBA 3.0.2 (December 2002), CORBA 3.0.3 (March 2004)	These versions contain minor editorial updates.
CORBA 3.1 (January 2008)	Reorganization of the CORBA specification took place in January 2008 with the 3.1 version of CORBA. The specification was divided into three separate documents. Part I – Interfaces Part II – Interoperability Part III – Components
CORBA 3.1.1 (August 2011)	Another major event took place with version 3.1.1. This version was formally published by ISO as the 2012 edition standard: ISO/IEC 19500-1, 19500-2, and 19500-3.
CORBA 3.2 (November 2011)	This version is based on the outcome of the DDS4CCM and CCM Revision Task Forces.
CORBA 3.3 (November 2012)	This version of CORBA is also known as CORBA/ZIOP

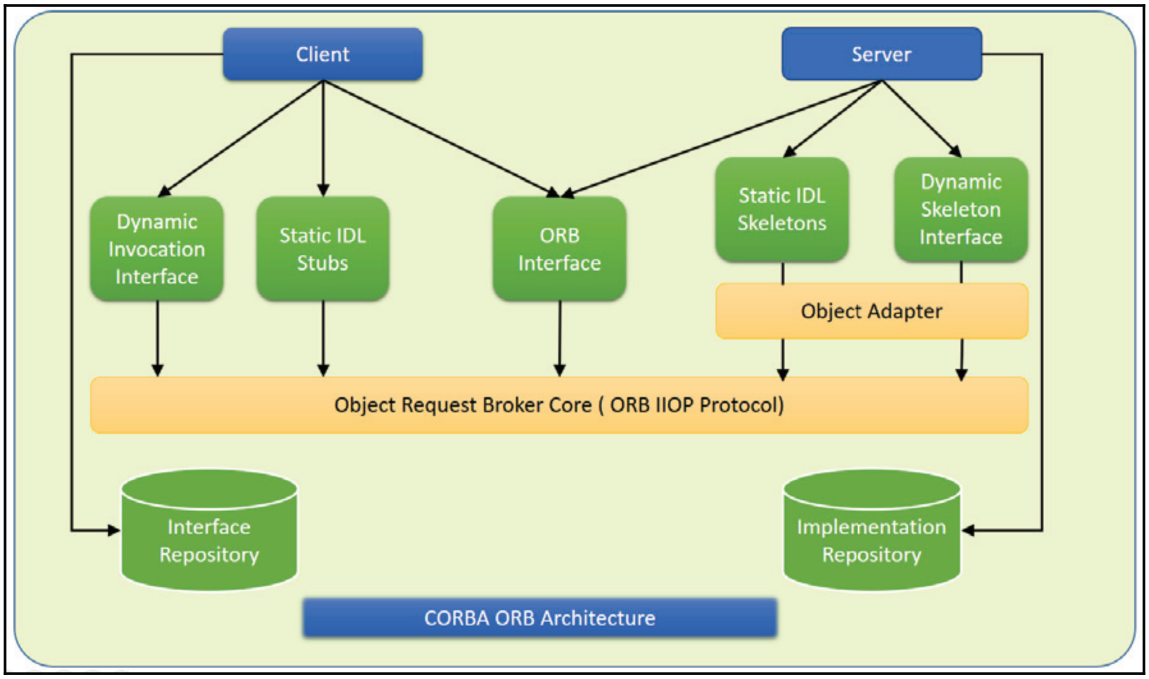
CORBA standards help with the integration of versatile systems, from mainframes to mini computers and from Unix systems to handhelds and embedded systems. CORBA is useful when there are huge connecting systems with good hit rate and when you expect high stability and a reliable system.

Except legacy applications, most of the applications follow common standards when it comes to object modeling, for example. All applications related to, say, "HR&Benefits" maintain an object model with details of the organization, employees with demographic information, benefits, payroll, and deductions. They are only different in the way they handle the details, based on the country and region they are operating for.

For each object type, similar to the HR&Benefits systems we discussed in the preceding section, we can define an interface using the **Interface Definition Language** (**OMG IDL**). The contract between these applications is defined in terms of an interface for the server objects that the clients can call. This IDL interface is used by each client to indicate when they should call any particular method to marshal (read and send the arguments). The target object is going to use the same interface definition when it receives the request from the client to unmarshal (read the arguments) in order to execute the method that was requested by the client operation. Again, during response handling, the interface definition is helpful to marshal (send from the server) and unmarshal (receive and read the response) arguments on the client side once received.

The IDL interface is a design concept that works with multiple programming languages that use OMG standards, including C, C++, Java, Ruby, Python, and IDLscript. This is close to writing a program to an interface, a concept we have been discussing that most recent programming languages and frameworks, such as Spring, adopt. This clear separation between the interface and implementation offered by CORBA standards is well empowered by OMG IDL and helps in easy integration of enterprise systems. However, the interface has to be defined clearly for each object. The systems encapsulate the actual implementation along with their respective data handling and processing, and only the methods are available to the rest of the world through the interface. Hence, the clients are forced to develop their invocation logic for the IDL interface exposed by the application they want to connect to with the method parameters (input and output) advised by the interface operation.

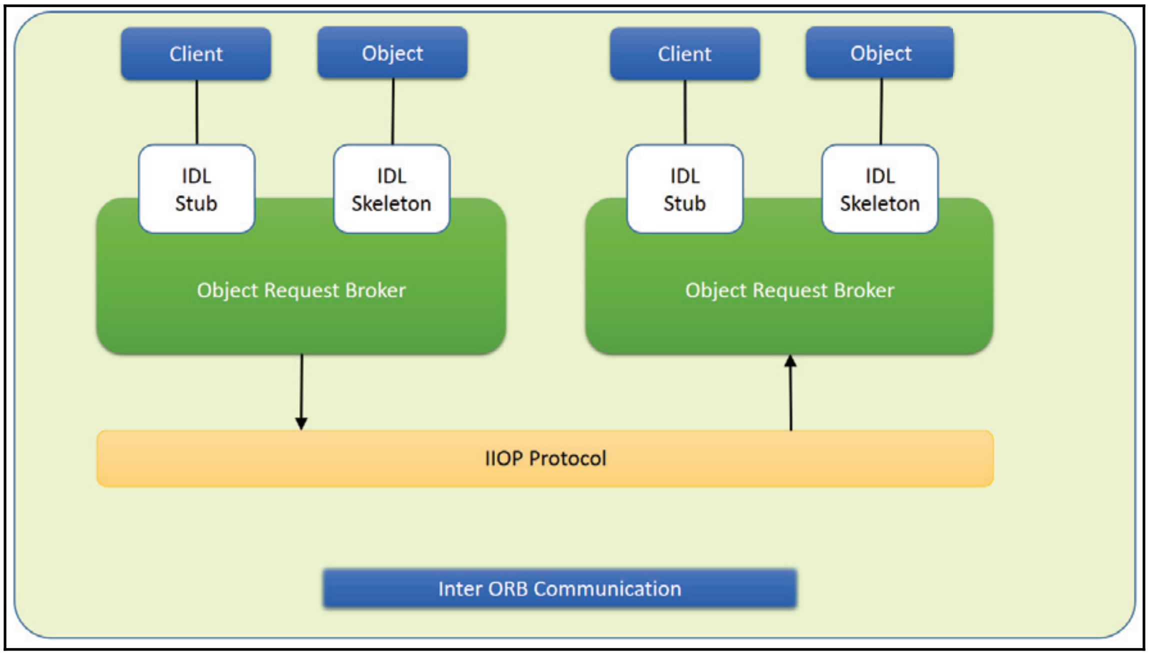
The following diagram shows a single-process ORB CORBA architecture with the IDL configured as client stubs with object skeletons, We have written our object (on the right) and a client for it (on the left), as represented in the diagram. The client and server use stubs and skeletons as proxies, respectively. As discussed earlier, the IDL interface follows a strict definition, and even though the client and server are implemented in different technologies, they should integrate smoothly with the interface definition strictly implemented.


Client
Server
Dynamic Invocation Interface
Static IDL Stubs
ORB Interface
Static IDL Skeletons
Dynamic Skeleton Interface
Object Adapter
Object Request Broker Core (ORB IIOP Protocol)
Interface Repository
Implementation Repository
CORBA ORB Architecture

In CORBA, each object instance acquires an object reference for itself with the electronic token identifier. Client invocations are going to use these object references that have the ability to figure out which ORB instance they are supposed to interact with. The stub and skeleton represent the client and server, respectively, to their counterparts. They help establish this communication through ORB and pass the arguments to the right method and its instance during the invocation.

Inter-ORB communication

The following diagram shows how remote invocation works for inter-ORB communication. It shows that the clients that interacted have created **IDL Stub** and **IDL Skeleton** based on **Object Request Broker** and communicated through **IIOP Protocol**.



To invoke the remote object instance, the client can get its object reference using a naming service. Replacing the object reference with the remote object reference, the client can make the invocation of the remote method with the same syntax as the local object method invocation. ORB keeps the responsibility of recognizing the remote object reference based on the client object invocation through a naming service and routes it accordingly. This configuration might have been more complex if you would have had cluster and load balancer configurations in place.

Java support for CORBA

Java has great support for using CORBA standards for several reasons, including the following:

- It is an object-oriented programming language
- It supports application portability across multiple platforms
- It provides seamless integration across web applications
- It has a strong component model with features such as enterprise Java beans.

CORBA's high-level object paradigm for distributed objects provided by Java with OMG IDL binding includes the following:

- Interfaces' definition with UML and no knowledge of their future implementations
- Easy integration with other applications developed in a different technology through IDL
- Distributed application integration across the globe
- Ability of the binding to generate code for remote communication
- Accessibility from binding to standard CORBA features

These features further extend the suitability of CORBA for the development of diverse distributed systems.

OMG IDL samples

Now let's review some of the sample programs for the OMG IDL language. We will review IDL interfaces, inheritance, types and constants, structures, exceptions, and other important components in this section.

Interfaces

The purpose of IDL is to define interfaces and their operations. To avoid name clashes when using several IDL declarations together, the module is used as a naming scope. Modules can contain nested modules. Interfaces open a new naming scope containing data type declarations, constants, attributes, and operations:

```
//EmployeeHiring.idl
Module EmployeeHiring {
  interface Employee();
}
```

The process of setting up a reference to one module from another can be defined with `outer::inner` as `::EmployeeHiring::Employee`:

```
Module outer{
  Module inner {
    interface inside{};
  };
  interface outside {
    inner::inside get_inside();
  };
};
```

The `get_inside()` operation is for returning the object reference for the `::outer::inner::inside` interface.

Inheritance

Interfaces can extend one or more interfaces if they want to carry out other interface operations, and in addition to that, declare their own operations as follows:

```
module EmployeeHierarchy {
  interface Person {
    typedef unsigned short ushort;
    ushort method1();
  };
  interface Employee : Person {
    Boolean method2(ushort num);
  };
};
```

In the preceding code, the `Employee` interface extends the `Person` interface and adds `method2()` in addition to `method1()` inherited from the `Person` interface.

The following is an example of one interface inheriting multiple interfaces:

```
interface Clerk: Person, Employee, Associate::Administrator {
};
```

Types and constants

The basic data types advised by the IDL and their corresponding descriptions are given in the following table:

Type Keyword	Description
[unsigned] short	Signed [unsigned] 16-bit 2's complement integer
[unsigned] long	Signed [unsigned] 32-bit 2's complement integer
[unsigned] long long	Signed [unsigned] 64-bit 2's complement integer
float	16-bit IEEE floating point number
double	32-bit IEEE floating point number
long double	64-bit IEEE floating point number
fixed	fixed-point decimal number of up to 31 digits
char	ISO Latin-1 character
wchar	character from other character sets to support internationalization. The size is implementation dependent
boolean	Boolean type taking values TRUE or FALSE
string	variable length string of characters whose length is available at run time
wstring	variable length string of wchar characters
octet	8 bit uninterpreted type
enum	enumerated type with named integer values
any	can represent a valu from any possible IDL type, basic or constructed, object or nonobject
native	Opaque type, representation specified by language mapping

The following is an example that uses data types:

```
interface EmployeeRegistry {
  typedef identification string <10>;
  typedef name string <100>;
  identification getId(in string name);
  name getName(in string id);
};
```

Structures

Structures are used to define reusable data types. Their syntaxes have a keyword struct followed by a variable, which acts as a valid type name.

The basic data types advised by the IDL and their corresponding descriptions are as follows:

```
interface EmployeeRegistry {
  struct address_spec {
    name string <100>;
    salary float;
  };
```

Discriminated unions

Like structures, discriminated unions help as a valid type name in further declarations. They are followed by the type name and then the switch keyword, and they can take parameters of the type int/char/Boolean or enum:

```
. employee_dept {admin, sales, it, business};
union  employee switch (employee_dept) {
  case admin, business : age_spec age;
  cae sales : sales_detail sales;
  default float salary;
};
```

Sequences

Like structures, sequences help you define an element type. A sequence can encapsulate multiple other types of elements. A sequence is an ordered element, and its size can increase during the execution:

```
// "employee” as defined in previous section
typedef sequence <employee> AssignmentOrderSeq;
typedef sequence <employee, 4> QuartelyAssignmentOrderSeq;
typedef sequence <sequence <<employee>, 12> AnnualAssignmentOrderSeq;
typedef sequence <sequence <employee> > CompleteAssignmentOrderSeq;
```

Additionally, arrays, constants, operations, attributes, value types, abstract interfaces, and exceptions can be defined with IDL as well as with this syntax that looks like a pseudocode.

CORBA services

Just like the API libraries in Java that provide some specialized function, CORBA provides services. These services are a set of distributed class libraries known as **Common Object Services**. They provide specific types of objects that are useful to programmers in a distributed environment during a transaction, event service, relationship definition, and even life cycle. Complex distributed applications require additional functionality than just the ability to invoke remote objects. OMG has recognized this requirement and provided a specification for this additional functionality with the basic services; it is called CORBA services. Some of the important CORBA services are as follows:

- **Naming service**: This is helpful in the network system to let the objects in one system point and identify a different set of objects. Its naming context is similar to that of packages in Java. In a given context, names should be unique. Contexts can be nested and can have one context inside other. CORBA naming resembles a lot with the RMI registry's `java.rmi.Naming` interface. The only difference is that RMI does not support the hierarchical naming context and compound object names. It is the most popular service in a CORBA implementation.
- **Trading service**: Trading service helps find a group of objects by referring to a distributed system network. Yellow Pages is the best example of this service. Objects from one system are able to search for another set of objects if they have an entry in the trader service along with the specs for such objects.
- **Notification service**: Notification service is an asynchronous, subscription-based event notification service. If any event occurs on a specific object, then that event information along with the reference are notified to the listening object. If any object wishes to receive such notifications for an event occurring on any other object, they have to register to get automatic event notifications. The best example of this notification service is the coordination between a system and printer. When we give a number of print requests for different documents, they get queued on the spool to get printed. If document printing is complete for one, the printer automatically gets notified about that event to let it start with the next printing task waiting in the print queue.
- **Transaction service**: Transaction service refers to transaction processing for distributed objects. Each object gets the instructions to add or update its state from any other objects. Say, you want to consider a group of operations together to update an object's state that would affect its behavior. During this state-changing operation, if any problem occurs in a part of the transaction of one of the operations, then the object's previous state will be rolled back along with the recovery from the error. This is taken care of by the transaction service.

- **Persistent object state service**: Persistent state service refers to persistent storage of an object state. It acts as an interface between the CORBA application and the object databases or object persistent frameworks.
- **Event service:** Event service enables asynchronous communications between cooperating remote objects. It's similar in nature to message-passing and event-based messaging.
- **Security service**: Security service is a service that provides authentication, authorization, encryption, and other security features. This is a complex service to implement. It consists of security models and interfaces for the implementation of security services, application development, and security administration. The interfaces it provides are as follows:
    - Authentication and credentials generation for principals and exchange of credentials between principals
    - Performing secured transactions
    - Generating secure transactions and performing audit log for tracking and evidence generation
- **Query service**: Query service can be used to query and modify an object collection. An example of this is an interface between CORBA and the databases (relational, hierarchical, object-based, and so on).

Other important services include life cycle, relationships, externalization, concurrency control, licensing, properties, time, and collections.

Sample CORBA program using JAVA IDL

Before we step into a sample Java program for CORBA, let's confirm the minimum environment setup requirements.

For working with CORBA using Java, we need to have the latest JDK installed along with VisiBroker for Java (stable release – 8.5) set up on the computer.

The steps involved while writing a typical CORBA program are as follows:

1. Define the IDL interface.
2. Generate stubs and skeletons by building IDL (done automatically).
3. Implement the interface.
4. Develop the server.
5. Develop the client.
6. Run the service (naming), the server, and the client.

IDL interface specification

In the following example, let's define an IDL interface to work with a client-server application. The client application invokes a method of the server program and gets the response from the server method.

The IDL interface has the interface name followed by the `.idl` extension:

```
//Employee.idl
module org {
  module admin {
    module hr {
      module employee {
        interface Registry {
          string register();
        };
      };
    };
  };
};
```

Compiling the IDL

The next step is to compile the IDL interface definition from the preceding step so that it can generate the stub and skeleton using the following command:

```
prompt> idl2java -strict -root_dir generated Employee.idl
```

This should generate the following Java classes:

```
Registry.java
RegistryHolder.java         RegistryHelper.java
RegistryStub.java           RegistryPOA.java
RegistryOperations.java     RegistryPOATie.java
```

Client application

A client application definition of the aforementioned communication can be established using the following steps:

- Initialize the CORBA environment and get the reference to ORB
- Get the object reference for the object to invoke the operation
- Invoke the operations and process the result

The following code snippet will generate the Java interface:

```
package org.admin.hr.employee;
public interface Registry extends RegistyOperations, org.omg.CORBA.object,
org.omg.CORBA.portable.IDLEntry
{
}
```

The `RegistryOperations` interface is defined as follows:

```
package org.admin.hr.employee;
public interface RegistryOperations {
  public java.lang.String register();
}
```

ORB can be initialized using the following code snippet:

```
package org.admin.hr.employee;
import java.io.*;
import org.omg.CORBA.*;
public class Client {
  public static void main(String args[]) {
    try {
      ORB orb = ORB.init (args. Null);
    }
    Catch(Exception e)  {
      System.out.println("Exception " + e);
    }
  }
}
```

An object reference can be obtained in this ORB setup as follows:

```
org.omg.CORBA.Object obj = orb.string_to_object (args[0]);
```

Get the `Registry` object reference by invoking the `narrow ()` method from the `helper` class, as follows:

```
Registry reg = RegistryHelper.narrow(obj);
If (reg == null) {
  System.err.println("object reference type fault");
  System.exit(-1);
}
```

Once the object reference is found from ORB as earlier, its method can be invoked as a local Java method itself:

```
System.out.println(reg.register());
```

Compile and execute the Java client program as follows:

```
prompt > javac Client.java
prompt > java org.admin.hr.employee.Client IOR:0002453..2
```

This should give the output as follows:

```
Employee Rohith, is now registered
```

Object implementation

A sample object implementation of the preceding client program is as follows:

```
package org.admin.hr.employee;
import org.omg.CORBA.*;
public class RegisterImpl extends RegisterPOA {
  private String name;
  RegisterImple(String name){
    this.name = name;
  }
  public String register() {
    return "Employee " + name + ", is now registered”;
  }
}
```

Compile the preceding `impl` class using the following command:

```
prompt> javac RegisterImpl.java
```

Defining the server

The next step is to implement the server program for ORB, as follows:

```
package org.admin.hr.employee;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class Server {
  public static void main(String[] args) {
    if(args.length < 1 ) {
      System.out.println("Usage : java org.admin.hr.employee <name> ");
      System.exit(-1);
    }
    try {
      ORB orb = ORB.init(args, null);
      //make java object
      RegisterImpl regImpl = new RegisterImpl (args[0]);
```

```
        //make CORBA object
        POA poa =
        POAHelper.narrow(orb.resolve_initial_references("RootPOA");
      };
      poa.the_POAManager().activate();
      //get the object reference
      Org.omg.CORBA.Object obj = poa.servant_to_reference (regImpl);
      //print the object reference
      System.out.println( orb.object_to_string(obj));
      );
      //wait for the requests to receive
      orb.run();
    }
    Catch(InvalidName e){
      System.out.println(e);
    }
    Catch(UserException e){
      System.err.println(e);
    }
    Catch(SystemException e) {
      System.err.println(e);
    }
    }
  }
```

Compiling and starting the server

Following commands helps in compiling and executing the Server component:

```
prompt> javac Server.java
prompt> java org.admin.hr.employee.Server Rohith
```

This should print stringified IOR, as follows:

```
IOR:000044589
```

This standard output can be saved in a file with the following command:

```
prompt> java org.admin.hr.employee.Server Rohith > shw.ior
```

Executing the client

Now that the server is ready, the client program can be executed as follows:

```
prompt> java org.admin.hr.employee.Client ‘Ram shw.ior’
```