
Java IDL

Ein **Object Request Broker** (ORB) ist ein Kommunikationsmechanismus für die **synchrone Interaktion zwischen verteilten**, in verschiedenen Programmiersprachen implementierten **Objekten** über verschiedene Netzwerke und Betriebssystemplattformen hinweg.

In diesem Teil wird gezeigt, wie man mittels Java und vordefinierter Klassen, die eine ORB Kommunikation zur Verfügung stellen, interoperable Anwendungen entwickeln kann. Java wird Client- und Server-seitig verwendet; am Ende wird gezeigt, dass die entwickelten Programme mit den Programmen des vorigen Teils kommunizieren können.

Inhalt

1.Überblick.....	3
2.Interface Definition in IDL.....	5
3.Erzeugen des Servers.....	10
3.1.Grundlegende Einstellungen.....	16
3.1.1.Import der Packages der CORBA Bibliothek.....	18
3.2.Definition der Servant Klasse.....	19
3.3.Die Serverklasse.....	22

3.3.1. Aktivierung des POA Managers.....	23
3.3.2. Die Instanziierung des Servant Objektes.....	23
3.3.3. Arbeiten mit dem Naming Service.....	24
3.3.4. Registrierung des Servant beim Name Service.....	25
3.3.5. Start des ORB.....	26
3.4. Übersetzen und Starten des Servers.....	27
4. Realisieren der Clients.....	28
5. Glossary von Java IDL Begriffen.....	34

1. Überblick

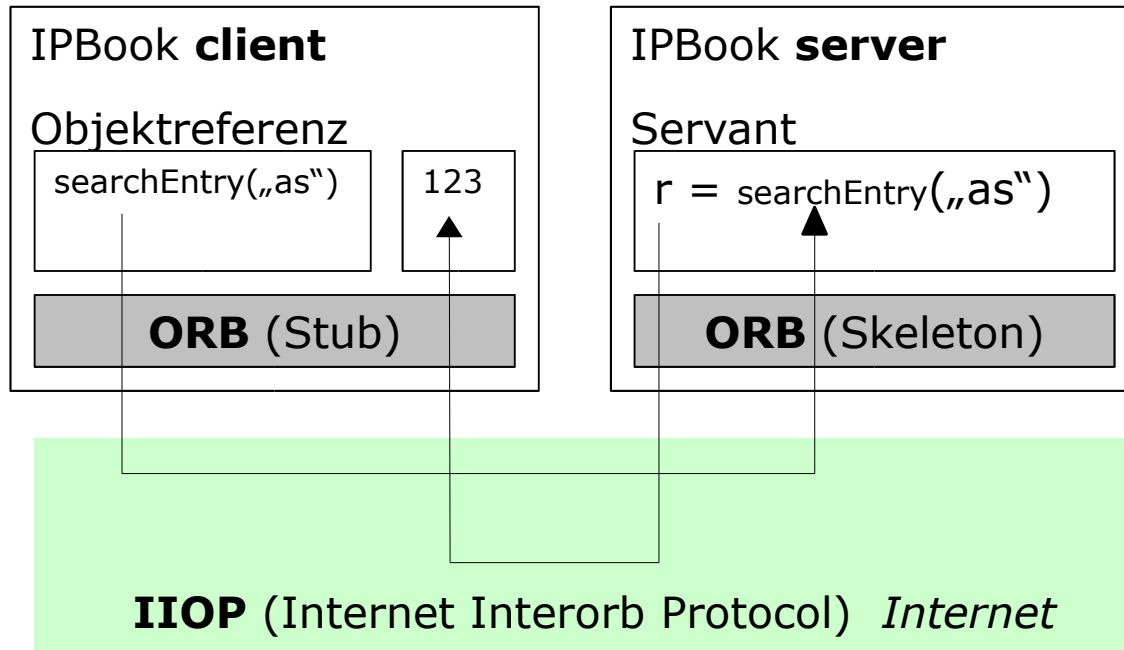
Java IDL ist eine Technologie zur Entwicklung von verteilten Anwendungen, bei denen Objekte interagieren, die auch auf unterschiedlichen Hard- und Betriebssystemplattformen ablaufen und auch mit verschiedenen Programmiersprachen entwickelt worden sein können.

Dies ist möglich, da Java IDL den **CORBA Standard** unterstützt. Ein Wesenszug von CORBA ist die Sprache IDL zur Beschreibung der Schnittstellen der Objekte. **Java IDL** ist ein **Mapping**, um IDL Beschreibungen auf **Java** abzubilden.

Die Beziehung zwischen verteilten Objekten hat zwei Seiten:

- ein Server stellt ein Remote-Interface zur Verfügung,
- ein Client verwendet das Remote-Interface des Servers.

Im Folgenden wird wieder das Beispiel des Telefon-Servers verwendet, um Java IDL zu verdeutlichen. In der nachfolgenden Abbildung ist die Methode `searchEntry` dargestellt.



2. Interface Definition in IDL

Zunächst wird eine Interfacebeschreibung erstellt, wir nehmen die **IDL-Datei** vom Beispiel des Telefonservers.

```
$ cat ipBook.idl
module ipBookApp {

    /* Interface for verry simple phone book */
    interface ipBook {
        /* Add an entry */
        void addEntry(in string name, in string number);

        /* Search for an entry */
        string searchEntry(in string name);
    };
};
$
```

Durch das Mapping des IDL Compilers wird ein **IDL Interface** auf ein **Java Interface** abgebildet.

Die **CORBA Operationen** (addEntry und searchEntry) werden durch das Mapping auf **Java Methoden** im erzeugten Interface abgebildet.

Der Java IDL Compiler wird mit der Option `-fall` aufgerufen, um zu erreichen, dass Client- und Servercode generiert werden soll (per Default wird nur Clientcode erzeugt).

Hier wird als Objektadapter POA ¹ verwendet, denn wir werden (später) mit dem Mico ORB kommunizieren.

```
$ ls -l
-rw-r--r--    1 as      Kein      233 Feb 21 09:22 ipBook.idl
$ idlj -fall ipBook.idl
$ ls -l
-rw-r--r--    1 as      Kein      235 Feb 21 09:29 ipBook.idl
drwxr-xr-x    2 as      Kein           0 Feb 21 09:29 ipBookApp
$
```

Der Aufruf des IDL Compilers `idlj` bewirkt, dass ein Verzeichnis mit dem Namen des Moduls, das in der IDL Datei angegeben ist, erzeugt wird. In diesem Verzeichnis befinden sich die generierten Java Dateien.

Die meisten der erzeugten Dateien kann man bei der Entwicklung der Anwendung ignorieren. Sie sind nur für das Deployment (Verteilen des Codes zum Einsatz der Anwendung) erforderlich.

- `ipBookPOA.java`

Hier ist der **Skeleton Kode** als abstrakte, streambasierte Klasse generiert, der die CORBA Serverfunktionalität enthält.

¹ Die Version J2SE v1.4 ist dazu erforderlich!

- `_ipBookStub.java`
Hier ist die Klasse für die **Client Stubs**.
- `ipBook.java`
Hier ist das **Java Interface** abgelegt, das zum IDL Interface „gemapped“ wurde.
- `ipBookHelper.java`
Hier sind Hilfsfunktionen abgelegt, die benötigt werden, um mit **Objektreferenzen** umzugehen.
- `ipBookHolder.java`
Hier sind ebenfalls Funktionalitäten zur CORBA Kommunikation abgelegt – hauptsächlich **Typumwandlungen** (CORBA-Java) für Parameter von Interfaces.
- **`ipBookOperations.java`**
Dieses Interface enthält die **Methoden `addEntry` und `searchEntry`**, die sowohl von Client als auch Server verwendet werden.

Wenn man sich `ipBookOperations.java`, `ipBook.java` und `ipBook.idl` ansieht, erkennt man, wie das Mapping erfolgt ist:

```

$ cat ipBook.idl
module ipBookApp {
    /* Interface for verry simple phone book */
    interface ipBook {
        /* Add an entry */
        void addEntry(in string name, in string number);

        /* Search for an entry */
        string searchEntry(in string name);
    };
};
$
$ cat ipBook.java
package ipBookApp;
/* Interface for verry simple phone book */
public interface ipBook extends ipBookOperations,
                                org.omg.CORBA.Object, org.omg.
CORBA.portable.IDLEntity
{
} // interface ipBook
$
$ cat ipBookOperations.java
package ipBookApp;

/* Interface for verry simple phone book */
public interface ipBookOperations
{

```

```

/* Add an entry */
void addEntry (String name, String number);

/* Search for an entry */
String searchEntry (String name);
} // interface ipBookOperations
$

```

IDL Anweisung	Java Anweisung
module ipBookApp	package ipBookApp
interface ipBook	public interface ipBook
void addEntry (in string name, in string number)	void addEntry (String name, String number);
string searchEntry(in string name);	String searchEntry (String name);

3. Erzeugen des Servers

Im Verzeichnis `ipBookApp (=Modul-Name)` wird nun der Code für den Server realisiert. Das Java-Programm wird schrittweise erklärt.

Das Serverprogramm besteht aus zwei Klassen

- Die **Klasse** für den **Servant** `ipBookImpl` ist die Implementierung des IDL Interface. Der Servant ist eine Subklasse von `ipBookPOA`, die durch den IDL Compiler generiert wurde. Der Servant enthält für jede IDL Operation eine Methode, also `addEntry()` und `searchEntry()`.
- Die **Serverklasse** beinhaltet die Methode `main()`, die für die **ORB Kommunikation** zuständig ist:
 1. Erzeugung und Initialisierung einer ORB Instanz
 2. Aktivieren des POA Managers
 3. Erzeugung der Servant Instanz und Anmelden beim ORB
 4. Erhalt einer CORBA Objekt-Referenz für einen Namenskontext
 5. Registrierung eines neuen Objektes
 6. Warten auf Aufrufe von einem Client

Der vollständige Server-Kode ist in der Datei `ipBookServer.java` enthalten.

```
$ cat ipBookServer.java
import ipBookApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.io.*;
import java.util.Properties;

class ipBookImpl extends ipBookPOA {
    private ORB orb;

    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    // implement searchEntry() method
    public String searchEntry(String name) {
        System.out.println("got searchEntry Book request");
        try {
            File inFile = new File("ipBook.dat");
            FileReader in = new FileReader(inFile);
            int c;
```

```

StringBuffer buf = new StringBuffer();
while ((c = in.read()) != -1) {
    // extraxt an entry
    if (c != '\n') {
        buf = buf.append((char)c);
    } else { // found an entry
        int pos;
        pos = buf.indexOf(name);
        if (pos != -1) { // found entry with name in it
            return buf.toString();
        } else {
            buf.setLength(0);
        }
    }
}

in.close();
return ("No entry for " + name);
}
catch (Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
    return e.toString();
}
}

```

```
// implement addEntry() method
public void addEntry(String name, String number) {
    System.out.println("got addEntry request\n");
    try {
        File outFile = new File("ipBook.dat");
        FileWriter out = new FileWriter(outFile, true);
        String buf = new String(name+": "+number+"\n");

        out.write(buf, 0, buf.length());
        out.close();
    }
    catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }
}
}
```

```

public class ipBookServer {
    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=
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();

            // create servant and register it with the ORB
            ipBookImpl ipBoImpl = new ipBookImpl();
            ipBoImpl.setORB(orb);

            // get object reference from the servant
            org.omg.CORBA.Object ref =
                rootpoa.servant_to_reference(ipBoImpl);
            ipBook href = ipBookHelper.narrow(ref);

            // get the root naming context
            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 = "ipBook";
    NameComponent path[] = ncRef.to_name( name );
    ncRef.rebind(path, href);

    System.out.println("ipBookServer ready and waiting ...");

    // wait for invocations from clients
    orb.run();
}
catch (Exception e) {
    System.err.println("ERROR: " + e);
    e.printStackTrace(System.out);
}

System.out.println("ipBookServer Exiting ...");

}
}
```

Betrachten wir nun die einzelnen Bestandteile.

3.1. Grundlegende Einstellungen

Die Struktur von **CORBA Server-Programmen** ist fast immer die gleiche:

- Import der Packages der CORBA Bibliothek
- Definition der Server Klassen
- Definition von `main` mit Ausnahmebehandlung

3.1.1.Import der Packages der CORBA Bibliothek

```
// The package containing our stubs
import ipBookApp.*;

// We use the naming servive
import org.omg.CosNaming.*;

// The package containing special exceptions
// thrown by the name service
import org.omg.CosNaming.NamingContextPackage.*;

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

// Classes needed for the Portable Server Inheritance Model
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

// Only for our implementation of addEntry and searchEntry
import java.io.*;

// Properties to initiate the ORB
import java.util.Properties;
```

3.2. Definition der Servant Klasse

Die Klasse `ipBookImpl` ist der Servant, die die Realisierung der Methoden `addEntry()` und `searchEntry()` beinhaltet.

```
class ipBookImpl extends ipBookPOA {  
    private ORB orb;  
  
    public void setORB(ORB orb_val) {  
        orb = orb_val;  
    }  
}
```

Innerhalb `searchEntry()` wird aus der Datei `ipBook.dat` zeichenweise ein Buffer gefüllt und nach dem Namen gesucht.

```

// implement searchEntry() method
public String searchEntry(String name) {
    System.out.println("got searchEntry Book request");
    try {
        File inFile = new File("ipBook.dat");
        FileReader in = new FileReader(inFile);
        int c;

        StringBuffer buf = new StringBuffer();
        while ((c = in.read()) != -1) {
            // extraxt an entry
            if (c != '\n') {
                buf = buf.append((char)c);
            } else { // found an entry
                int pos;
                pos = buf.indexOf(name);
                if (pos != -1) { // found entry with name in it
                    return buf.toString();
                } else {
                    buf.setLength(0);
                }
            }
        }
        in.close();
        return ("No entry for " + name);
    }
}

```

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

Die Realisierung von `addEntry()` ist einfach; an das Ende der Telefonbuchdatei wird ein Eintrag angefügt.

```

// implement addEntry() method
public void addEntry(String name, String number) {
    System.out.println("got addEntry request\n");
    try {
        File outFile = new File("ipBook.dat");
        FileWriter out = new FileWriter(outFile, true);
        String buf = new String(name+": "+number+"\n");

        out.write(buf, 0, buf.length());
        out.close();
    }
    catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }
}
}

```

3.3. Die Serverklasse

Hier wird der ORB initialisiert und die Ausnahmen definiert.

Dabei werden die Kommandozeilenargumente an den ORB weiter gereicht. Wir werden später sehen, dass es sich hier um den zu verwendenden Port und den Host handelt.

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

3.3.1. Aktivierung des POA Managers

Der Objektadapter muss aktiviert werden. Hier wird **POA** verwendet.

```
        // get reference to rootpoa & activate the POAManager  
        POA rootpoa=  
        POAHelper.narrow(orb.resolve_initial_references("RootPOA"));  
        rootpoa.the_POAManager().activate();
```

3.3.2. Die Instanziierung des Servant Objektes

Ein Server kann mehrere Servants instanziiieren. Der Servant erbt vom Interface, das vom idlj Compiler erzeugt wurde und ist für die Ausführung der Interface Operationen verantwortlich.

Zunächst die Instanziierung mit ipBookImpl(). Dann erhält man eine Objektreferenz, die später für den Naming Service benötigt wird wird.

```
// create servant and register it with the ORB
ipBookImpl ipBoImpl = new ipBookImpl();
ipBoImpl.setORB(orb);

// get object reference from the servant
org.omg.CORBA.Object ref =
    rootpoa.servant_to_reference(ipBoImpl);
ipBook href = ipBookHelper.narrow(ref);
```

3.3.3.Arbeiten mit dem Naming Service

Hier wird der **COS** (**C**ommon **O**bject **S**ervice) Naming Service verwendet, der es erlaubt, dass Clients die Servant Objekte adressieren können.

Der **Server** muss eine **Objektreferenz** an **Clients** geben können, damit die Servant Objekte ansprechbar sind. Dazu wird hier ein **Naming Server** **orbd** verwendet. Dies wird den Parameter „NamingService“ zum Ausdruck gebracht.

```
// get the root naming context
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);
```

3.3.4.Registrierung des Servant beim Name Service

Der String `ipBook` wird verwendet, um den Eintrag in der Datenbasis des Name Servers zu identifizieren.

```
// bind the Object Reference in Naming
String name = "ipBook";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);
```

3.3.5.Start des ORB

Nun wird der Server gestartet. Er wartet auf Client-Anfragen. Für jede Anfrage wird ein Thread erzeugt, die Anfrage darin befriedigt; dann wird erneut gewartet.

```
        System.out.println("ipBookServer ready and waiting ...");

        // wait for invocations from clients
        orb.run();
    }
    catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }

    System.out.println("ipBookServer Exiting ...");

}
}
```

3.4.Übersetzen und Starten des Servers

Der Server wird mit dem javac Compiler übersetzt.

```
$ javac ipBookServer.java ipBookApp/*.java
$
```

Damit der Server arbeiten kann, muss zuerst der Name Server gestartet werden.

Dabei sind durch Optionen Port und Host anzugeben. Der Port sagt aus, auf welchem Port der Name Server hören soll; Host gibt den Rechner an, auf dem er startet.

```
$ orbd -ORBInitialPort 1050 -ORBInitialHost localhost &
[2122]
$
```

Nun kann der Server gestartet werden. Hier ist der Port und der Host anzugeben, auf dem der **Name Server** läuft.

```
$ java ipBookServer -ORBInitialPort 1050 -ORBInitialHost localhost &
[2124]
$ ipBookServer ready and waiting ...
$
```

4. Realisieren der Clients

Als nächsten Schritt werden die Clients entwickelt. Die beiden Clients (Hinzufügen von und Suchen nach Einträgen sind nachfolgend angegeben).

Bei den Clients sind folgende Schritte notwendig:

1. Erzeugen des ORB
2. Bestimmung des Namenskontextes (hier Verwendung von orbd)
3. Ermittlung der Objektreferenz durch Anfrage an den Name Server
4. Verwendung der Remote Objekte

```
$ cat ipBookaddEntry.java
import ipBookApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class ipBookaddEntry {
    static ipBook ipBookImpl;
    public static void main(String args[]) {
        String eName;
        String eNumber;
        if (args.length != 6) {
            System.out.println(
                "usage: addEntry -ORBInitialPort <portt> " +
                "-ORBInitialHost <Host> <name> <number>");
            return;
        } else {
            eName = new String(args[4]);
            eNumber = new String(args[5]);
        }
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
```

```

// Use NamingContextExt instead of NamingContext. This is
// part of the Interoperable naming Service.
NamingContextExt ncRef =
    NamingContextExtHelper.narrow(objRef);

// resolve the Object Reference in Naming
String name = "ipBook";
ipBookImpl = ipBookHelper.narrow(ncRef.resolve_str(name));

// addEntry
ipBookImpl.addEntry(eName, eNumber);
} catch (Exception e) {
    System.out.println("ERROR : " + e) ;
    e.printStackTrace(System.out);
}
}
}
$

```

```

$ cat ipBooksearchEntry.java
import ipBookApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class ipBooksearchEntry {
    static ipBook ipBookImpl;
    public static void main(String args[]) {
        String eName;
        if (args.length != 5) {
            System.out.println(
                "usage: addEntry -ORBInitialPort <portt> "+
                "-ORB InitialHost <Host> <name>");
            return;
        } else {
            eName = new String(args[4]);
        }
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // get the root naming context
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            // Use NamingContextExt instead of NamingContext. This is
            // part of the Interoperable naming Service.

```

```
NamingContextExt ncRef =
    NamingContextExtHelper.narrow(objRef);

// resolve the Object Reference in Naming
String name = "ipBook";
ipBookImpl = ipBookHelper.narrow(ncRef.resolve_str(name));

// searchEntry
System.out.println(ipBookImpl.searchEntry(eName));

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

}
$
```

Die Clients werden wie folgt aufgerufen (**vorher** muss ordb und ipBookServer gestartet worden sein):

```
$ ps
  PID   PPID   PGID   WINPID  TTY  UID   STIME  COMMAND
  1924   1620   1924    1924   con  1000  17:05:24  orbd
  2384   1620   2384    2384   con  1000  17:13:38  ipBookServer
$
$ java ipBookaddEntry -ORBInitialPort 1050 \
      -ORBInitialHost localhost eve 1234
$
$ java ipBooksearchEntry -ORBInitialPort 1050 \
      -ORBInitialHost localhost eve
eve:1234
$
```

5. Glossary von Java IDL Begriffen

adapter activator

An adapter activator is an object that the application developer can associate with a POA. The ORB will invoke an operation on an adapter activator when a request is received for a child POA that does not currently exist. The adapter activator can then create the required POA on demand.

attribute (IDL)

An identifiable association between an object and a value. An attribute `A` is made visible to clients as a pair of operations: `get_A` and `set_A`. Readonly attributes only generate a `get` operation.

Also, that part of an IDL interface that is similar to a public class field or C++ data member. The idl compiler maps an OMG IDL attribute to accessor and modifier methods in the Java programming language. For example, an interface `ball` might include the attribute `color`. The `idlj` compiler would generate a Java programming language method to get the color, and unless the attribute is `readonly`, a method to set the color.

client

Any code which invokes an operation on a distributed object. A client might itself be a CORBA object, or it might be a non-object-oriented program, but while invoking a method on a CORBA object, it is said to be acting as client.

client stub

A Java programming language class generated by [idlj](#) and used transparently by the client [ORB](#) during object [invocation](#). The remote object reference held by the client points to the client stub. This stub is specific to the IDL interface from which it was generated, and it contains the information needed for the client to invoke a method on the CORBA object that was defined in the IDL interface.

client tier

The portion of a distributed application that requests services from the server tier. Typically, the client tier is characterized by a small local footprint, a graphical user interface, and simplified development and maintenance efforts.

Common Object Request Broker Architecture (CORBA)

An [OMG](#)-specified architecture that is the basis for the CORBA object model. The CORBA specification includes an interface definition language (IDL), which is a language-independent way of creating contracts between objects for implementation as distributed applications.

The Java 2 Platform, Standard Edition, v1.4, provides a CORBA Object Request Broker (ORB) and two CORBA programming models that utilize the Java CORBA ORB and Internet InterORB Protocol (IIOP). The two programming models are the RMI programming model, or [RMI-IIOP](#), and the IDL programming model, or [Java IDL](#). For more information on these programming models, read [CORBA Technology and the Java Platform](#).

See also: [client tier](#), [service tier](#), [data store tier](#)

CORBA object

An entity which (1) is defined by an OMG IDL interface, and (2) for which an object reference is available. `Object` is also the implicit common base type for object references of IDL interfaces.

data store tier

The portion of a distributed application that manages access to persistent data and its storage mechanisms, such as relational databases.

distributed application

A program designed to run on more than one computer, typically with functionality separated into tiers such as [client](#), [service](#), and [data store](#).

distributed environment

A network of one or more computers that use [CORBA objects](#). Objects are installed on the various machines and can communicate with each other.

Dynamic Invocation Interface (DII)

An API that allows a client to make dynamic invocations on remote CORBA objects. It is used when at compile time a client does not have knowledge about an object it wants to invoke. Once an object is discovered, the client program can obtain a definition of it, issue a parameterized call to it, and receive a reply from it, all without having a type-specific [client stub](#) for the remote object.

Dynamic Skeleton Interface (DSI)

An API that provides a way to deliver requests from an ORB to an object implementation when the type of the object implementation is not known at compile time. DSI, which is the

server side analog to the client side [DII](#), makes it possible for the application programmer to inspect the parameters of an incoming request to determine a target object and method.

exception (IDL)

An IDL construct that represents an exceptional condition that could be returned in response to an invocation. There are two categories of exceptions: (1) system exceptions, which inherit from `org.omg.CORBA.SystemException` (which is a `java.lang.RuntimeException`), and (2) user-defined exceptions, which inherit from `org.omg.CORBA.UserException` (which is a `java.lang.Exception`).

factory object

A CORBA object that is used to create new CORBA objects. Factory objects are themselves usually created at server installation time.

idlj compiler

A tool that takes an interface written in OMG IDL and produces Java programming language interfaces and classes that represent the mapping from the IDL interface to the Java programming language. The resulting files are `.java` files. Prior to J2SDK v1.3, the `idlj` compiler was known as the `idltojava` compiler. The `idlj` compiler supports new CORBA-standard features required for RMI-IIOP. The `idlj` compiler is placed in the SDK's `.bin` directory by the installer.

idltojava compiler

A tool that takes an interface written in OMG IDL and produces Java programming language interfaces and classes that represent the mapping from the IDL interface to the Java programming language. The resulting files are `.java` files. Beginning with J2SDK v1.3, the

IDL-to-Java language mapping is handled by the `idlj` compiler, which supports new CORBA-standard features required for RMI-IIOP. The `idltojava` compiler can be downloaded from the Java Developer Connection (JDC) web site.

implementation

A concrete class that defines the behavior for all of the operations and attributes of the IDL interface it supports. A servant object is an instance of an implementation. There may be many implementations of a single interface.

initial naming context

The `NamingContext` object returned by a call to the method `orb.resolve_initial_references("NameService")`. It is an [object reference](#) to the COS Naming Service registered with the ORB and can be used to create other `NamingContext` objects.

See also: [naming context](#)

Interface Definition Language (IDL)

The [OMG-standard](#) language for defining the interfaces for all CORBA objects. An IDL interface declares a set of [operations](#), [exceptions](#), and [attributes](#). Each operation has a signature, which defines its name, parameters, result and exceptions. OMG IDL does not include implementations for operations; rather, as its name indicates, it is simply a language for defining interfaces. The complete syntax and semantics for IDL are available in the OMG specification at the [OMG web site](#).

Interface Repository (IFR)

A service that contains all the registered component interfaces, the methods they support, and the parameters they require. The IFR stores, updates, and manages object interface definitions. Programs may use the IFR APIs to access and update this information. An IFR is not necessary for normal client/server interactions.

Internet InterORB Protocol (IIOP)

The OMG-specified network protocol for communicating between ORBs. Java IDL in J2SE v.1.4. conforms to CORBA/IIOP specification 2.3.1.

invocation

The process of performing a method call on a CORBA object, which can be done without knowledge of the object's location on the network. Static invocation, which uses a client stub for the invocation and a server skeleton for the service being invoked, is used when the interface of the object is known at compile time. If the interface is not known at compile time, [dynamic invocation](#) must be used.

Java IDL

The classes, libraries, and tools that make it possible to use CORBA objects from the Java programming language. The main components of Java IDL are an ORB, a naming service, and the `idlj` compiler. All are part of J2SE, v.1.4.

name binding

The association of a name with an [object reference](#). Name bindings are stored in a [naming context](#).

namespace

A collection of [naming contexts](#) that are grouped together.

naming context

A [CORBA object](#) that supports the `NamingContext` interface and functions as a sort of directory which contains (points to) other naming contexts and/or simple names. Similar to a directory structure, where the last item is a file and preceding items are directories, in a naming context, the last item is an object reference name, and the preceding items are naming contexts.

naming service

A CORBA service that allows [CORBA objects](#) to be named by means of binding a name to an object reference. The [name binding](#) may be stored in the naming service, and a client may supply the name to obtain the desired object reference. The two options for Naming Services shipped with J2SE v.1.4 are [orbd](#), which is a daemon process containing a Bootstrap Service, a Transient Naming Service, a Persistent Naming Service, and a Server Manager, and [tnameserv](#), a transient naming service.

object

A computational grouping of operations and data into a modular unit. An object is defined by the interface it presents to others, its behavior when operations on its interface are invoked, and its state.

object adapter

The ORB component which provides object reference, activation, and state related services to an object implementation. There may be different adapters provided for different kinds of implementations. The [POA](#) is a particular type of object adapter specified by OMG that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations.

object id

An object id is a value that is used by the [POA](#) and by the user-supplied [implementation](#) to identify a particular abstract CORBA object. Object Id values may be assigned and managed by the POA, or they may be assigned and managed by the implementation. Object Id values are hidden from clients, encapsulated by references. Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences.

object implementation

See [implementation](#).

Object Management Group (OMG)

An international organization with over 700 members that establishes industry guidelines and object management specifications in order to provide a common framework for object-oriented application development. Its members include platform vendors, object-oriented database vendors, software tool developers, corporate developers, and software application vendors. The OMG [Common Object Request Broker Architecture](#) specifies the CORBA object model. See www.omg.org for more information.

object reference

A construct containing the information needed to specify an object within an ORB. An object reference is used in method invocations to locate a [CORBA object](#). Object references are the CORBA object equivalent to programming language-specific object pointers. They may be obtained from a factory object or from the Naming Service. An object reference, which is opaque (its internal structure is irrelevant to application developers), identifies the same CORBA object each time it is used. It is possible, however, for multiple object references to refer to the same CORBA object.

Object Request Broker (ORB)

The libraries, processes, and other infrastructure in a [distributed environment](#) that enable CORBA objects to communicate with each other. The ORB connects objects requesting services to the objects providing them.

operation (IDL)

The construct in an [IDL interface](#) that maps to a Java programming language method. For example, an interface `ball` might support the operation `bounce`. Operations may take parameters, return a result, or raise exceptions. IDL operations can be `oneway`, in which case they cannot return results (return values or out arguments) or raise exceptions.

operations interface

A non abstract IDL interface is mapped to two public Java interfaces: a [signature interface](#) and an operations interface. The operations interface has the same name as the IDL interface with the suffix `Operations` appended to the end and is used in the server-side mapping and as a mechanism for providing optimized calls for collocated clients and servers.

ORBD (Object Request Broker Daemon)

The [orbd](#) tool is a daemon process containing a Bootstrap Service, a Transient [Naming Service](#), a Persistent Naming Service, and a Server Manager.

parameter (IDL)

One or more objects the client passes to an IDL operation when it invokes the operation. Parameters may be declared as "in" (passed from client to server), "out" (passed from server to client), or "inout" (passed from client to server and then back from server to client).

persistent object

An object that can survive the process or thread that created it. A persistent object exists until it is explicitly deleted.

PIDL (Pseudo-IDL)

The interface definition language for describing a CORBA [pseudo-object](#). Each language mapping, including the mapping from IDL to the Java programming language, describes how pseudo objects are mapped to language-specific constructs. PIDL mappings may or may not follow the rules that apply to mapping regular CORBA objects.

POA (Portable Object Adapter)

An object adapter is the mechanism that connects a request using an [object reference](#) with the proper code to service that request. The POA is a particular type of object adapter specified by OMG that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations.

The POA is also intended to allow persistent objects -- at least, from the client's perspective. That is, as far as the client is concerned, these objects are always alive, and maintain data values stored in them, even though physically, the server may have been restarted many times, or the implementation may be provided by many different [object implementations](#).

POA Manager

A POA manager is an object that encapsulates the processing state of one or more [POAs](#). Using operations on a POA manager, the developer can cause requests for the associated POAs to be queued or discarded. The developer can also use the POA manager to deactivate the POAs.

policy

A policy is an object associated with a [POA](#) by an application in order to specify a characteristic shared by the objects implemented in that POA. This specification defines policies controlling the POA's threading model as well as a variety of other options related to the management of objects. Other specifications may define other policies that affect how an [ORB](#) processes requests on objects implemented in the POA.

Portable Interceptors

Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB. For more information, see [PortableInterceptor Package](#).

pragma

In J2SE v.1.2, a directive to the `idltojava` compiler to perform certain operations while compiling an IDL file. For example, the pragma "javaPackage" directs the `idltojava` compiler to put the Java programming language interfaces and classes it generates from the IDL interface into the Java programming language package specified. In J2SE v.1.3 and v.1.4, this functionality is supported via the `-pkgPrefix` command line option to `idlj`.

pseudo-object

An object similar to a CORBA object in that it is described in IDL, but unlike a CORBA object, it cannot be passed around using its object reference, nor can it be narrowed or stringified. Examples of pseudo-objects include the Interface Repository and DII which, although implemented as libraries, are more clearly described in OMG specifications as pseudo-objects with IDL interfaces. The IDL for pseudo-objects is called "PIDL" to indicate that a pseudo-object is being defined.

RMI-IIOP

[Java RMI-IIOP](#) is an Object Request Broker and a compiler, `rmic -iiop`, that generates IIOP stub and tie classes. With RMI-IIOP, developers can write remote interfaces in the Java programming language and implement them simply using Java technology and the Java RMI APIs. These interfaces can be implemented in any other language that is supported by an OMG mapping and a vendor supplied ORB for that language. Similarly, clients can be written in other languages using IDL derived from the remote Java technology-based interfaces. Using RMI-IIOP, objects can be passed both by reference and by value over IIOP.

servant

A servant is a programming language object or entity that implements requests on one or more objects. Servants generally exist within the context of a server process. Requests made on an object's references are mediated by the [ORB](#) and transformed into [invocations](#) on a particular servant. In the course of an object's lifetime it may be associated with (that is, requests on its references will be targeted at) multiple servants.

servant manager

A servant manager is an object that the application developer can associate with a [POA](#). The [ORB](#) will invoke operations on servant managers to activate servants on demand, and to deactivate servants. Servant managers are responsible for managing the association of an object (as characterized by its Object Id value) with a particular servant, and for determining whether an object exists or not. There are two kinds of servant managers, called `ServantActivator` and `ServantLocator`; the type used in a particular situation depends on policies in the POA.

servant object

An instance of an [object implementation](#) for an [IDL interface](#). The servant object is registered with the ORB so that the ORB knows where to send invocations. It is the servant that performs the services requested when a CORBA object's method is invoked.

server

A program that contains the implementations of one or more [IDL interfaces](#). For example, a desktop publishing server might implement a `Document` object type, a `ParagraphTag` object type, and other related object types. The server is required to register each implementation ([servant object](#)) with the ORB so that the ORB knows about the servant. Servers are sometimes referred to as object servers.

server skeleton

A public abstract class generated by the `idlj` compiler that provides the ORB with information it needs in dispatching method invocations to the [servant object](#)(s). A server skeleton, like a [client stub](#), is specific to the IDL interface from which it is generated. A server skeleton is the server side analog to a client stub, and these two classes are used by ORBs in static invocation.

servertool

The Java IDL Server Tool, [servertool](#) provides the ease-of-use interface for the application programmers to register, unregister, startup, and shutdown a server.

service tier

The portion of a [distributed application](#) that contains the business logic and performs most of the computation. The service tier is typically located on a shared machine for optimum resource use.

signature interface

A non-abstract IDL interface is mapped to two public Java interfaces: a signature interface and an [operations interface](#). The signature interface, which extends `IDLEntity`, has the same name as the IDL interface name and is used as the signature type in method declarations when interfaces of the specified type are used in other interfaces.

skeleton

The object-interface-specific [ORB](#) component which assists an [object adapter](#) in passing requests to particular methods.

static invocation

See [invocation](#).

stringified object reference

An [object reference](#) that has been converted to a string so that it may be stored on disk in a text file (or stored in some other manner). Such strings should be treated as opaque because they are ORB-implementation independent. Standard `object_to_string` and `string_to_object` methods on `org.omg.CORBA.Object` make stringified references available to all CORBA Objects.

stub

A local procedure corresponding to a single operation that invokes that operation when called.

tnameserv

The CORBA COS (Common Object Services) Naming Service provides a tree-like directory for object references much like a filesystem provides a directory structure for files. The

Transient Naming Service provided with Java IDL, [tnameserv](#), is a simple implementation of the COS Naming Service specification. [Object references](#) are stored in the namespace by name and each object reference-name pair is called a name binding. Name bindings may be organized under naming contexts. Naming contexts are themselves name bindings and serve the same organizational function as a file system subdirectory. All bindings are stored under the initial naming context. The initial naming context is the only persistent binding in the namespace; the rest of the namespace is lost if the transient naming service process halts and restarts.

transient object

An object whose existence is limited by the lifetime of the process or thread that created it.