

# Introduction to CORBA

Alex Chaffee and Bruce Martin

1.0

JGuru Training by the Magelang Institute

(C)1999 jGuru.com

# Contents

<b>Chapter 1. Introduction to CORBA</b>	1
1.1. Introduction to CORBA	1
1.2. Distributed Applications	2
1.2.1. Distributed Applications	2
1.2.1.1. Data are Distributed	3
1.2.1.2. Computation is Distributed	3
1.2.1.3. Users are Distributed	3
1.2.1.4. Fundamental Realities of Distributed Systems	3
1.2.1.5. Distributed Object Systems	5
1.3. OMG CORBA Specification	5
1.3.1. What is CORBA?	5
1.3.1.1. The OMG	5
1.3.1.2. CORBA Architecture	5
1.3.1.3. The ORB	6
1.3.1.4. CORBA as a Standard for Distributed Objects	6
1.3.1.5. CORBA Services	7
1.3.1.6. CORBA Products	8
1.4. The Stock Application	9
1.4.1. The Stock Application	9
1.4.1.1. Some Objects in the Stock Application	11
1.5. Implementing a Client	11
1.5.1. Implementing a CORBA Client	12
1.5.1.1. CORBA Objects are Described by IDL Interfaces	12
1.5.1.2. Object References and Requests	14
1.5.1.3. IDL Type System	15
1.5.1.4. IDL to Java Binding	19
1.5.1.5. IDL to Java Compiler	20
1.5.1.6. Obtaining Object References	20
1.5.1.7. The Client's Model of Object Creation	21
1.5.1.8. Exceptions	21
1.6. Implementing a Simple Distributed Object	22
1.6.1. Object Implementations	22
1.6.1.1. Providing an Implementation	23

1.6.1.2. Interface versus Implementation Hierarchies	.. .. .	23
1.6.1.3. Implementation Type Checking	.. .. .	25
1.6.1.4. Implementing a Server Using the Java 2 ORB	.. .. .	25
1.6.1.5. Implementing a Server Using VisiBroker 3.x	.. .. .	27
1.6.1.6. Differences Between Server Implementations	.. .. .	28
1.6.1.7. Packaging Object Implementations	.. .. .	28
1.7. Object Adapters	.. .. .	29
1.7.1. Object Adapters	.. .. .	29
1.7.1.1. Activation on Demand by the Basic Object Adapter (BOA)	.. .. .	29
1.7.1.2. Portable Object Adapter (POA)	.. .. .	30
1.8. Resources	.. .. .	31
1.8.1. Resources	.. .. .	31
1.8.1.1. Web Sites	.. .. .	31
1.8.1.2. Documentation and Specs	.. .. .	31
1.8.1.3. Books	.. .. .	32
1.8.1.4. Miscellaneous	.. .. .	32
1.9. Appendix: Java 2 ORB Notes	.. .. .	32
1.9.1. About The Java 2 ORB	.. .. .	32
1.9.1.1. idltojava Notes	.. .. .	33
1.9.1.2. System Properties	.. .. .	33
1.10. Appendix: VisiBroker 3.x Notes	.. .. .	34
1.10.1. VisiBroker 3.x	.. .. .	34
1.10.1.1. VisiBroker Tools	.. .. .	35
1.10.1.2. Using VisiBroker with Java 2	.. .. .	35
1.10.1.3. Portable Stubs and Skeletons	.. .. .	36
1.10.1.4. Using the BOA with VisiBroker	.. .. .	36
1.10.1.5. Using the VisiBroker Smart Agent	.. .. .	37
<b>Chapter 2. Introduction to CORBA : Magercises</b>	.. .. .	39
2.1. Magercise: Run the Simple Stock Example	.. .. .	40
2.2. Magercise: Stock Example	.. .. .	42
2.3. Magercise: Dynamic Stock Example	.. .. .	46
2.4. Magercise: A Message Box	.. .. .	50
2.5. Magercise: Writing Callbacks	.. .. .	53
<b>Appendix A.</b>	.. .. .	57



# Chapter 1. Introduction to CORBA

## 1.1. Introduction to CORBA

The Java Developer Connection <sup>SM</sup> (JDC) presents a Short Course introducing the Common Object Request Broker Architecture written by Java <sup>TM</sup> Software licensee, the MageLang Institute. A leading provider of Java technology training, MageLang has contributed regularly to the JDC since 1996.

The MageLang Institute, since its founding in 1995, has been dedicated to promoting the growth of the Java technology community by providing excellent education and acting as an independent resource. To find out more about MageLang's Java technology training, visit the MageLang web site (<http://www.magelang.com/jdc/>) .

The Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) provides a platform-independent, language-independent architecture for writing distributed, object-oriented applications. CORBA objects can reside in the same process, on the same machine, down the hall, or across the planet. The Java language is an excellent language for writing CORBA programs. Some of the features that account for this popularity include the clear mapping from OMG IDL to the Java programming language, and the Java runtime environment's built-in garbage collection.

This introductory course will give you an overall view of how CORBA works, how to write CORBA applications in the Java programming language, and illustrates Sun's Java 2 ORB implementation of CORBA as well as Inprise's VisiBroker for Java. The approach to presenting CORBA taken in this course is through an example stock trading application.

## Objectives

After completing this module you will understand:

- The basic structure of a CORBA application
- Why the Java programming language is an ideal language for CORBA programming
- How to specify distributed object interfaces in IDL
- What areas of CORBA you should pursue for further study

By the end of this module will be able to:

- Write simple CORBA interfaces in IDL
- Use the Java classes generated by the IDL compiler
- Create CORBA client applications in the Java language
- Implement CORBA distributed objects in the Java language
- Find distributed objects at run time

## Prerequisites

A general familiarity with object-oriented programming concepts and the Java programming language. If you are not familiar with these capabilities, see the Java Tutorial (<http://java.sun.com/docs/books/tutorial/>) . The Magercises require the ability to modify and build simple Java programs.

## About the Authors

Alex Chaffee is a Software Guru with MageLang Institute. As the Director of Software Engineering for EarthWeb, Alex co-created Gamelan (<http://www.gamelan.com/>)

Bruce Martin is one of the pioneers of distributed object systems and programming. At Sun Microsystems he was one of Sun's CORBA architects and was the primary author of five of the OMG's CORBA Services specifications. At Inprise Corporation, Bruce contributed to Inprise's first CORBA-based Application Server. Now, Bruce is a Software Guru with MageLang Institute. Bruce has written many papers and given talks on distributed systems, advanced transaction models, object-oriented programming, and distributed object technologies at both academic conferences and industrial events.

### 1.2. Distributed Applications

#### 1.2.1. Distributed Applications

CORBA products provide a framework for the development and execution of *distributed applications*. But why would one want to develop a distributed application in the first place? As you will see later, distribution introduces a whole new set of difficult issues. However, sometimes there is no choice; some applications by their very nature are distributed across multiple computers because of one or more of the following reasons:

- The *data* used by the application are distributed
- The *computation* is distributed
- The *users* of the application are distributed

### 1.2.1.1. Data are Distributed

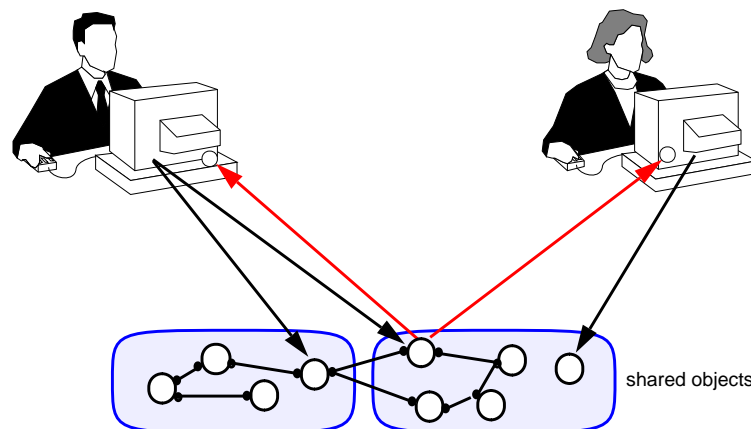
Some applications must execute on multiple computers because the data that the application must access exist on multiple computers for administrative and ownership reasons. The owner may permit the data to be accessed remotely but not stored locally. Or perhaps the data cannot be co-located and must exist on multiple heterogeneous systems for historical reasons.

### 1.2.1.2. Computation is Distributed

Some applications execute on multiple computers in order to take advantage of multiple processors computing in parallel to solve some problem. Other applications may execute on multiple computers in order to take advantage of some unique feature of a particular system. Distributed applications can take advantage of the scalability and heterogeneity of the distributed system.

### 1.2.1.3. Users are Distributed

Some applications execute on multiple computers because users of the application communicate and interact with each other via the application. Each user executes a piece of the distributed application on his or her computer, and shared objects, typically execute on one or more servers. A typical architecture for this kind of application is illustrated below.



Prior to designing a distributed application, it is essential to understand some of the fundamental realities of the distributed system on which it will execute.

### 1.2.1.4. Fundamental Realities of Distributed Systems

Distributed application developers must address a number of issues that can be taken for granted in a local program where all logic executes in the same operating system process. The following table summarizes some of the basic differences between objects that are co-located in the same process, and objects that interact across process or machine boundaries.

	<b>Co-located</b>	<b>Distributed</b>
<b>Communication</b>	Fast	Slow
<b>Failures</b>	Objects fail together	Objects fail separately Network can partition
<b>Concurrent access</b>	Only with multiple threads	Yes
<b>Secure</b>	Yes	No

The communication between objects in the same process is orders of magnitude faster than communication between objects on different machines. The implication of this is that you should avoid designing distributed applications in which two or more distributed objects have very tight interactions. If they do have tight interactions, they should be co-located.

When two objects are co-located, they fail together; if the process in which they execute fails, both objects fail. The designer of the objects need not be concerned with the behavior of the application if one of the objects is available and the other one is not. But if two objects are distributed across process boundaries, the objects can fail independently. In this case, the designer of the objects must be concerned with each of the object's behavior in the event the other object has failed. Similarly, in a distributed system the network can partition and both objects can execute independently assuming the other has failed.

The default mode for most local programs is to operate with a single thread of control. Single threaded programming is easy. Objects are accessed in a well-defined sequential order according to the program's algorithms, and you need not be concerned with concurrent access.

If you decide to introduce multiple threads of control within a local program, you must consider the possible orderings of access to objects and use synchronization mechanisms to control concurrent access to shared objects. But at least you have a choice of introducing multiple threads of control. In a distributed application, there are necessarily multiple threads of control. Each distributed object is operating in a different thread of control. A distributed object may have multiple concurrent clients. As the developer of the object and the developer of the clients, you must consider this concurrent access to objects and use the necessary synchronization mechanisms.

When two objects are co-located in the same process, you need not be concerned about security. When the objects are on different machines, you need to use security mechanisms to authenticate the identity of the other object.



### 1.2.1.5. Distributed Object Systems

*Distributed object systems* are distributed systems in which all entities are modeled as objects. Distributed object systems are a popular paradigm for object-oriented distributed applications. Since the application is modeled as a set of cooperating objects, it maps very naturally to the services of the distributed system.

In spite of the natural mapping from object-oriented modeling to distributed object systems, do not forget the realities of distributed systems described above. Process boundaries really do matter and they will impact your design.

That said, the next section of this course discusses the CORBA standard for distributed object systems.

## 1.3. OMG CORBA Specification

### 1.3.1. What is CORBA?

CORBA, or Common Object Request Broker Architecture, is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

#### 1.3.1.1. The OMG

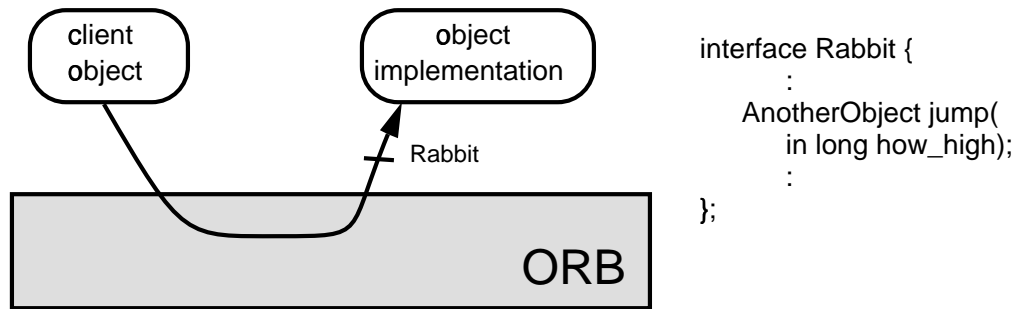
The Object Management Group (OMG) is responsible for defining CORBA. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

#### 1.3.1.2. CORBA Architecture

CORBA defines an architecture for distributed objects. The basic CORBA paradigm is that of a request for services of a distributed object. Everything else defined by the OMG is in terms of this basic paradigm.

The services that an object provides are given by its *interface*. Interfaces are defined in OMG's Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces.

The figure below graphically depicts a request. A client holds an object reference to a distributed object. The object reference is typed by an interface. In the figure below the object reference is typed by the **Rabbit** interface. The Object Request Broker, or ORB, delivers the request to the object and returns any results to the client. In the figure, a **jump** request returns an object reference typed by the **AnotherObject** interface.



### 1.3.1.3. The ORB

The ORB is the distributed service that implements the request to the remote object. It locates the remote object on the network, communicates the request to the object, waits for the results and when available communicates those results back to the client.

The ORB implements location transparency. Exactly the same request mechanism is used by the client and the CORBA object regardless of where the object is located. It might be in the same process with the client, down the hall or across the planet. The client cannot tell the difference.

The ORB implements programming language independence for the request. The client issuing the request can be written in a different programming language from the implementation of the CORBA object. The ORB does the necessary translation between programming languages. Language bindings are defined for all popular programming languages.

### 1.3.1.4. CORBA as a Standard for Distributed Objects

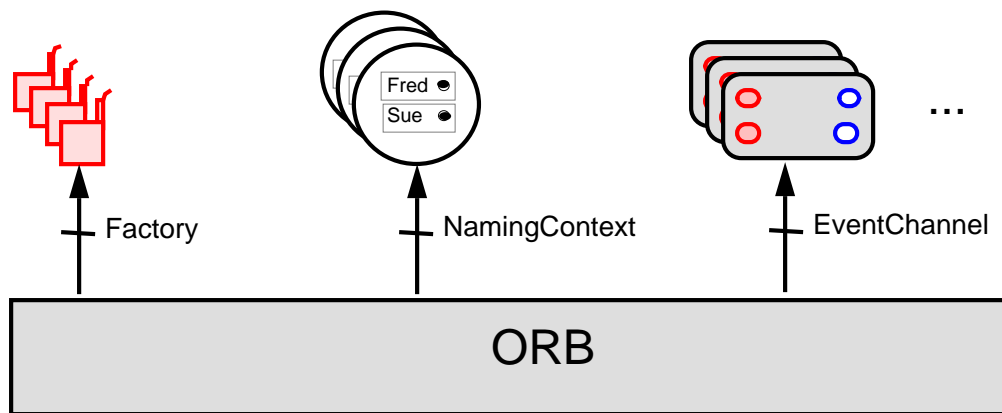
One of the goals of the CORBA specification is that clients and object implementations are portable. The CORBA specification defines an application programmer's interface (API) for clients of a distributed object as well as an API for the implementation of a CORBA object. This means that code written for one vendor's CORBA product could, with a minimum of effort, be rewritten to work with a different vendor's product. However, the reality of CORBA products on the market today is that CORBA clients are portable but object implementations need some rework to port from one CORBA product to another.

CORBA 2.0 added interoperability as a goal in the specification. In particular, CORBA 2.0 defines a network protocol, called *IIOP* (Internet Inter-ORB Protocol), that allows clients using a CORBA product from any vendor to communicate with objects using a CORBA product from any other vendor. IIOP works across the Internet, or more precisely, across any TCP/IP implementation.

Interoperability is more important in a distributed system than portability. IIOP is used in other systems that do not even attempt to provide the CORBA API. In particular, IIOP is used as the transport protocol for a version of Java™ RMI (so called "RMI over IIOP"). Since EJB is defined in terms of RMI, it too can use IIOP. Various application servers available on the market use IIOP but do not expose the entire CORBA API. Because they all use IIOP, programs written to these different API's can interoperate with each other and with programs written to the CORBA API.

#### 1.3.1.5. CORBA Services

Another important part of the CORBA standard is the definition of a set of distributed services to support the integration and interoperation of distributed objects. As depicted in the graphic below, the services, known as *CORBA Services* or COS, are defined on top of the ORB. That is, they are defined as standard CORBA objects with IDL interfaces, sometimes referred to as "Object Services."



There are several CORBA services. The popular ones are described in detail in another module of this course. Below is a brief description of each:

<i>Service</i>	<i>Description</i>
<b>Object life cycle</b>	Defines how CORBA objects are created, removed, moved, and copied
<b>Naming</b>	Defines how CORBA objects can have friendly symbolic names
<b>Events</b>	Decouples the communication between distributed objects
<b>Relationships</b>	Provides arbitrary typed n-ary relationships between CORBA objects
<b>Externalization</b>	Coordinates the transformation of CORBA objects to and from external media
<b>Transactions</b>	Coordinates atomic access to CORBA objects
<b>Concurrency Control</b>	Provides a locking service for CORBA objects in order to ensure serializable access
<b>Property</b>	Supports the association of name-value pairs with CORBA objects
<b>Trader</b>	Supports the finding of CORBA objects based on properties describing the service offered by the object
<b>Query</b>	Supports queries on objects

#### 1.3.1.6. CORBA Products

CORBA is a specification; it is a guide for implementing products. Several vendors provide CORBA products for various programming languages. The CORBA products that support the Java programming language include:

<i>ORB</i>	<i>Description</i>
<b>The Java 2 ORB</b>	The Java 2 ORB comes with Sun's Java 2 SDK. It is missing several features.
<b>VisiBroker for Java</b>	A popular Java ORB from Inprise Corporation. VisiBroker is also embedded in other products. For example, it is the ORB that is embedded in the Netscape Communicator browser.
<b>OrbixWeb</b>	A popular Java ORB from Iona Technologies.
<b>WebSphere</b>	A popular application server with an ORB from IBM.
<b>Netscape Communicator</b>	Netscape browsers have a version of VisiBroker embedded in them. Applets can issue request on CORBA objects without downloading ORB classes into the browser. They are already there.
<b>Various free or shareware ORBs</b>	CORBA implementations for various languages are available for download on the web from various sources.

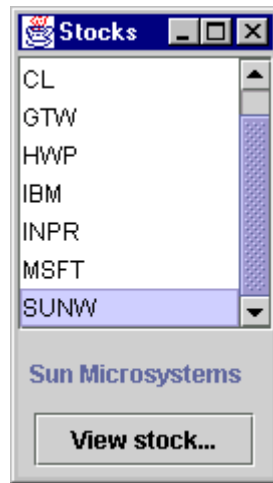
Providing detailed information about all of these products is beyond the scope of this introductory course. This course will just use examples from both Sun's Java 2 ORB and Inprise's VisiBroker 3.x for Java products.

## 1.4. The Stock Application

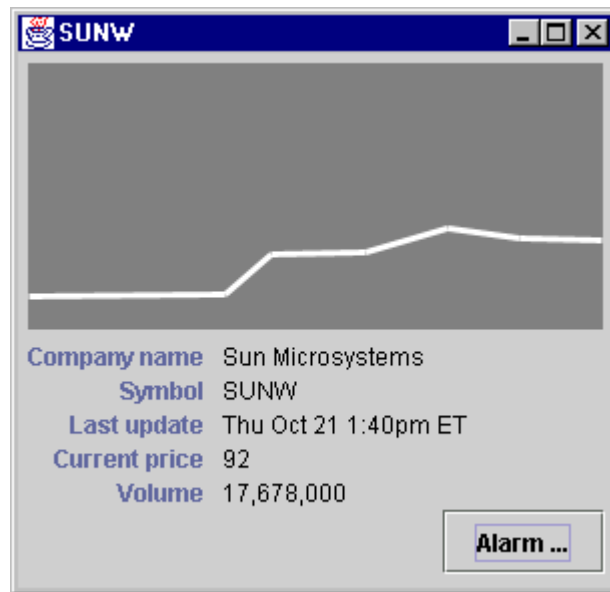
### 1.4.1. The Stock Application

The stock trading application is a distributed application that illustrates the Java™ programming language and CORBA. In this introductory module only a small simple subset of the application is used. Feel free to expand upon the application to enhance it once you are more comfortable with CORBA.

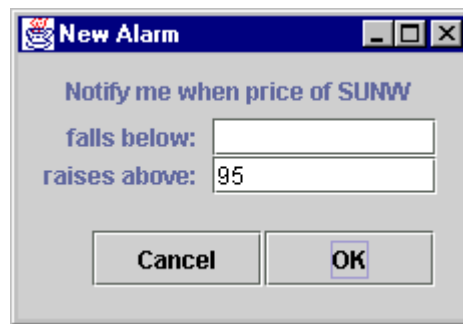
The stock application allows multiple users to watch the activity of stocks. The user is presented with a list of available stocks identified by their stock symbols. The user can select a stock and then press the "view" button.



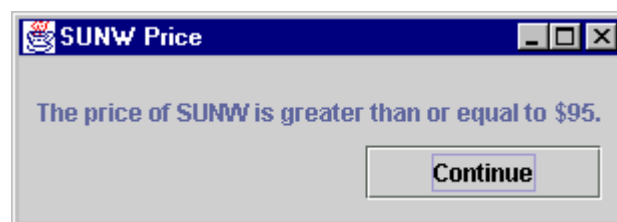
Selecting the "view" button results in a report about the stock, indicating the name of the company, the stock symbol, the current price, the last time it was updated, the trading volume, and a graph that shows the stock price over some interval. This report is automatically updated as new stock data becomes available.



The stock report also lets the user set an alarm by pressing the "Alarm" button. The alarm can be set to activate when the price of the stock falls below a certain price or when it exceeds a certain price.



When the price of the stock satisfies the alarm's condition, it activates and the user is notified.



Later the application could be extended to allow users to buy and sell stocks.

#### 1.4.1.1. Some Objects in the Stock Application

From the above description, you can easily identify the following distributed objects in the application.

<b>Stock</b>	A distributed object that represents a particular stock.
<b>StockPresentation</b>	A distributed object in the GUI that presents the stock data to the user for a particular stock.
<b>Alarm</b>	A distributed object that represents the alarm set by the user.
<b>AlarmPresentation</b>	A distributed object in the GUI that presents the alarm going off to the user.

The **Stock** object is now used to illustrate the CORBA distributed object model.

## 1.5. Implementing a Client

### 1.5.1. Implementing a CORBA Client

This section covers what you need to know to **use** CORBA objects from the Java™ programming language. It examines OMG IDL interfaces, the Java programming language binding for IDL interfaces, object references, and requests, how to obtain object references, and how, as a client, to create distributed objects. After reading this section and completing the exercises, you should be able to write a client using the Java programming language. Again, the stock example is used to illustrate the client's model of CORBA.

#### 1.5.1.1. CORBA Objects are Described by IDL Interfaces

The OMG Interface Definition Language IDL supports the specification of object interfaces. An object interface indicates the operations the object supports, but **not** how they are implemented. That is, in IDL there is no way to declare object state and algorithms. The implementation of a CORBA object is provided in a standard programming language, such as the Java programming language or C++. An interface specifies the contract between code using the object and the code implementing the object. Clients only depend on the interface.

IDL interfaces are programming language neutral. IDL defines *language bindings* for many different programming languages. This allows an object implementor to choose the appropriate programming language for the object. Similarly, it allows the developer of the client to choose the appropriate and possibly different programming language for the client. Currently, the OMG has standardized on language bindings for the C, C++, Java, Ada, COBOL, Smalltalk, Objective C, and Lisp programming languages.

So by using OMG IDL, the following can be described without regards to any particular programming language:

- Modularized object interfaces
- Operations and attributes that an object supports
- Exceptions raised by an operation
- Data types of an operation return value, its parameters, and an object's attributes

The IDL data types are:

- Basic data types ( **long**, **short**, **string**, **float**...)
- Constructed data types ( **struct**, **union**, **enum**, **sequence**)
- Typed object references
- The **any** type, a dynamically typed value



Again, IDL says nothing about object implementations. Here's the IDL interface for the example stock objects:

```
module StockObjects {

    struct Quote {
        string symbol;
        long at_time;
        double price;
        long volume;
    };

    exception Unknown{};

    interface Stock {

        // Returns the current stock quote.
        Quote get_quote() raises(Unknown);

        // Sets the current stock quote.
        void set_quote(in Quote stock_quote);

        // Provides the stock description,
        // e.g. company name.
        readonly attribute string description;
    };

    interface StockFactory {

        Stock create_stock(
            in string symbol,
            in string description
        );
    };
};
```

Note that the above example defines an IDL module named **StockObjects**, which contains the:

- Data structure **Quote**
- Exception **Unknown**
- Interface **Stock**
- Interface **StockFactory**

The module defines a scope for these names. Within the module, a data structure **Quote** and an exception **Unknown** are defined and then used in the **Stock** interface. The **Stock** interface is used in the definition of the **StockFactory** interface. Also note that the parameters to operations are tagged with the keywords **in**, **out**, or **inout**. The **in** keyword indicates the data are passed from the client to the object. The **out** keyword indicates that the data are returned from the object to the client, and **inout** indicates that the data are passed from the client to the object and then returned to the client.

IDL declarations are compiled with an IDL compiler and converted to their associated representations in the target programming languages according to the standard language binding. (This course uses the **Java** language binding in all of the examples. Later you will see the Java binding in more depth.)

#### 1.5.1.2. Object References and Requests

Clients issue a request on a CORBA object using an *object reference*. An object reference identifies the distributed object that will receive the request. Here's a Java programming language code fragment that obtains a **Stock** object reference and then it uses it to obtain the current price of the stock. Note that the code fragment does not directly use CORBA types; instead it uses the Java types that have been produced by the IDL to Java compiler.

```
Stock theStock = ...
try {
    Quote current_quote = theStock.get_quote();
} catch (Throwable e) {
}
```

Object references can be passed around the distributed object system, i.e. as parameters to operations and returned as results of requests. For example, notice that the **StockFactory** interface defines a **create()** operation that returns an instance of a **Stock**. Here's a Java client code fragment that issues a request on the factory object and receives the resulting stock object reference.

```
StockFactory factory = ...
Stock theStock = ...
try {
    theStock = factory.create(
        "GII",
        "Global Industries Inc.");
} catch (Throwable e) {
}
```

Note that issuing a request on a CORBA object is not all that different from issuing a request on a Java object in a local program. The main difference is that the CORBA objects can be anywhere. The CORBA system provides location transparency, which implies that the client cannot tell if the request is to an object in the same process, on the same machine, down the hall, or across the planet.

Another difference from a local Java object is that the life time of the CORBA object is not tied to the process in which the client executes, nor to the process in which the CORBA object executes. Object references persist; they can be saved as a string and recreated from a string.

The following Java code converts the **Stock** object reference to a string:

```
String stockString =  
    orb.object_to_string(theStock);
```

The string can be stored or communicated outside of the distributed object system. Any client can convert the string back to an object reference and issue a request on the distributed object.

This Java code converts the string back to a **Stock** object reference:

```
org.omg.CORBA.Object obj =  
    orb.string_to_object(stockString);  
Stock theStock = StockHelper.narrow(obj);
```

Note that the resulting type of the **string\_to\_object()** method is **Object**, not **Stock**. The second line narrows the type of the object reference from **Object** to **Stock**. IDL supports a hierarchy of interfaces; the **narrow()** method call is an operation on the hierarchy.

#### 1.5.1.3. IDL Type System

IDL interfaces can be defined in terms of other IDL interfaces. You previously saw a **Stock** interface that represents the basic behavior of a stock object.

Consider another IDL module:

```
module ReportingObjects {  
  
    exception EventChannelFailure{};  
  
    interface Reporting {
```

```
// Receive events in push mode
CosEventComm::PushSupplier push_events(
    in CosEventComm::PushConsumer consumer)
    raises(EventChannelFailure);

// Receive events in pull mode
CosEventComm::PullSupplier pull_events(
    in CosEventComm::PullConsumer consumer)
    raises(EventChannelFailure);

};

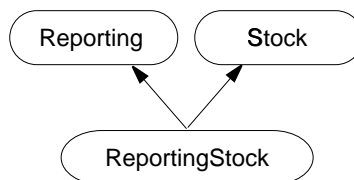
};
```

The **Reporting** interface supports the registration of interest in events. (Don't worry about the details of using the CORBA Event Service.)

Given the definition of the **Stock** interface and a **Reporting** interface, it is now possible to define a new **ReportingStock** interface in terms of **Reporting** and **Stock**.

```
interface ReportingStock: Reporting, Stock {
};
```

A **ReportingStock** supports all of the operations and attributes defined by the **Reporting** interface as well as all of those defined by the **Stock** interface. The **ReportingStock** interface inherits the **Stock** interface and the **Reporting** interface. Graphically this is represented as:



All CORBA interfaces implicitly inherit the **Object** interface. They all support the operations defined for **Object**. Inheritance of **Object** is implicit; there is no need to declare it.

Object references are typed by IDL interfaces. In a Java program you could type an object reference to be a **ReportingStock**.

```
ReportingStock theReportingStock;
```

Clients can pass this object reference to an operation expecting a supertype. For example assume there is an **EventManager** interface that has a **register** operation that takes an object reference typed by the **Reporting** interface.

```
interface EventManager {  
    :  
    void register(in Reporting event_supplier);  
    :  
};
```

The following is a legal request because a **ReportingStock** is a **Reporting**.

```
EventManager manager = ...  
ReportingStock theReportingStock = ...  
manager->register(theReportingStock); // ok
```

However, the following is not a legal request because a **Stock** is not a **Reporting**:

```
EventManager manager = ...  
Stock theStock = ...  
manager->register(theStock); // type error
```

## IDL Type Operations

Given that IDL interfaces can be arranged in a hierarchy, a small number of operations are defined on that hierarchy. The **narrow()** operation casts an object reference to a more specific type:

```
org.omg.CORBA.Object obj = ...  
Stock theStock = StockHelper.narrow(obj);
```

The `is_a()` operation, determines if an object reference supports a particular interface:

```
if (obj._is_a(StockHelper.id()) ...
```

The `id()` operation defined on the helper class returns a repository id for the interface. The repository id is a string representing the interface. For the stock example, the repository id is:

```
IDL:StockObjects/Stock:1.0
```

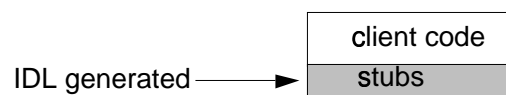
Finally, it is possible to widen an object reference, that is cast it to a less specific interface:

```
Stock theStock = theReportingStock;
```

There are no special operations to widen an object reference. It is accomplished exactly as in the Java programming language.

### Request Type Checking

The IDL compiler for Java programming language generates client-side stubs, which represent the CORBA object locally in the Java programming language. The generated code also represents in the Java programming language all of the IDL interfaces and data types used to issue requests. The client code thus depends on the generated Java code.



As you previously saw, passing an object reference typed by the **Stock** interface to the event manager would be illegal because the **Stock** interface does not inherit the **Reporting** interface. The Java compiler, not the IDL compiler, would catch this error at compile time.

#### 1.5.1.4. IDL to Java Binding

The Java binding for IDL maps the various IDL constructs to corresponding Java constructs. The following table shows how the IDL constructs are represented in the Java programming language. For comparison, the C++ binding is also shown.

<i>IDL</i>	<i>Java</i>	<i>C++</i>
module	package	namespace
interface	interface	abstract class
operation	method	member function
attribute	pair of methods	pair of functions
exception	exception	exception

Each of the IDL data types are represented in the Java programming language as follows:

<i>IDL Type</i>	<i>Java Type</i>
<b>boolean</b>	<b>boolean</b>
<b>char / wchar</b>	<b>char</b>
<b>octet</b>	<b>byte</b>
<b>short / unsigned short</b>	<b>short</b>
<b>long / unsigned long</b>	<b>int</b>
<b>long long / unsigned long long</b>	<b>long</b>
<b>float</b>	<b>float</b>
<b>double</b>	<b>double</b>
<b>string / wstring</b>	<b>String</b>

When discussing data type mapping, one term you run across frequently is *marshaling*. Marshaling is the conversion of a language-specific data structure into the CORBA IIOP streaming format. IIOP data can then be transmitted over a network to its destination, where it is then un-

marshaled from IIOP back into a language-dependent data structure.

#### 1.5.1.5. IDL to Java Compiler

CORBA products provide an IDL compiler that converts IDL into the Java programming language. The IDL compiler available for the Java 2 SDK is called `idltojava`. The IDL compiler that comes with VisiBroker for Java is called `idl2java`.

For the stock example, the command "`idltojava Stock.idl`" generates the files listed below. (The VisiBroker ORB generates the same files with the exception that the stub file is called `_st_Stock.java`, rather than `_StockStub.java`.)

<code>Stock.java</code>	The IDL interface represented as a Java interface
<code>StockHelper.java</code>	Implements the type operations for the interface
<code>StockHolder.java</code>	Used for <code>out</code> and <code>inout</code> parameters
<code>_StockStub.java</code>	Implements a local object representing the remote CORBA object. This object forwards all requests to the remote object. The client does not use this class directly.

The developer compiles the IDL using the IDL compiler and then compiles the generated code using the Java compiler. The compiled code must be on the classpath of the running Java program.

#### 1.5.1.6. Obtaining Object References

You may have noticed that there are three fundamental mechanisms in which a piece of code can obtain an object reference:

- It can be passed to it as a parameter
- It can be returned as the result of issuing a request
- It can be obtained by converting a string into an object reference

These fundamental mechanisms are supported by the ORB. Using these mechanisms, it is possible to define higher level services for locating objects in the distributed object system.



#### 1.5.1.7. The Client's Model of Object Creation

You may decide to export the ability to create an object to the distributed object system. You can accomplish this by defining a factory for the object. Factories are simply distributed objects that create other distributed objects.

There is nothing special about a factory. It is just another distributed object: It has an IDL interface, it is implemented in some programming language, and clients issue standard CORBA requests on factory objects.

There is no standard interface for a factory. Recall in the **stockObjects** example, the factory interface is:

```
interface StockFactory {
    Stock create_stock(
        in string stock_symbol,
        in string stock_description);
};
```

To create a stock object, a client simply issues a request on the factory.

Another object implementor could define an object factory differently.

#### 1.5.1.8. Exceptions

As you have seen in the stock example, CORBA has a concept of exceptions that is very similar to that of the Java programming language; naturally, CORBA exceptions are mapped to Java exceptions. When you issue a CORBA request, you must use the Java programming language's **try** and **catch** keywords.

There are two types of CORBA exceptions, *System Exceptions* and *User Exceptions*. System Exceptions are thrown when something goes wrong with the system—for instance, if you request a method that doesn't exist on the server, if there's a communication problem, or if the ORB hasn't been initialized correctly. The Java class **SystemException** extends **RuntimeException**, so the compiler won't complain if you forget to catch them. You need to explicitly wrap your CORBA calls in **try...catch** blocks in order to recover gracefully from System Exceptions.

CORBA System Exceptions can contain "minor codes" which may provide additional information about what went wrong. Unfortunately, these are vendor-specific, so you need to tailor your error recovery routines to the ORB you're using.

User Exceptions are generated if something goes wrong inside the execution of the remote method itself. These are declared inside the IDL definition for the object, and are automatically

generated by the `idltojava` compiler. In the stock example, `Unknown` is a user exception.

Since User Exceptions are subclasses of `java.lang.Exception`, the compiler will complain if you forget to trap them (and this is as it should be).

## 1.6. Implementing a Simple Distributed Object

### 1.6.1. Object Implementations

**NOTE:** the previous section discussed the *client's view* of CORBA, that is, how a Java™ client issues a request on a CORBA object. The client's view is standard across most CORBA products. Basically, the standard worked and there are only minor differences. Unfortunately, the same is not the case for the *implementation view* of CORBA. As such, some of the details given here might not match a particular CORBA product. Notes on different CORBA products appear as appendices.

This section describes what you need to know to implement a simple CORBA object in the Java programming language. It examines the Java server-side language binding for IDL, implementing objects and servers, implementation packaging issues, and CORBA object adapters. After completing this section, you should be able to write a simple CORBA object and server in the Java programming language. Again, the stock example is used to illustrate the implementation model of CORBA.

CORBA object implementations are completely invisible to their clients. A client can only depend on the IDL interface. In the Java programming language, or C++, this is not the case. The user of an object declares variables by a class name; doing so makes the code depend on much more than just the interface. The client depends on the object implementation programming language, the name of the class, the implementation class hierarchy, and, in C++, even the object layout.

The complete encapsulation for CORBA objects means the object implementor has much more freedom. Object implementations can be provided in a number of supported programming languages. This is not necessarily the same one the clients are written in. (Of course, here everything is in the Java programming language, but CORBA does not require this.)

The same interface can be implemented in multiple ways. There is no limit. In the stock example, the following are possible implementations of the `Stock` interface:

- A stock implementation class written in the Java programming language that obtains values from a commercial feed
- A stock implementation class written in C++ that accesses a database on the Internet

- A stock implementation written in Smalltalk that guesses stock prices

#### 1.6.1.1. Providing an Implementation

Recall that given an IDL file, the IDL compiler generates various files for a CORBA client. In addition to the files generated for a client, it also generates a *skeleton* class for the object implementation. A skeleton is the entry point into the distributed object. It unmarshals the incoming data, calls the method implementing the operation being requested, and returns the marshaled results. The object developer need only compile the skeleton and not be concerned with the insides of it. The object developer can focus on providing the implementation of the IDL interface.

To implement a CORBA object in the Java programming language, the developer simply implements a Java class that extends the generated skeleton class and provides a method for each operation in the interface. In the example, the IDL compiler generates the skeleton class `_StockImplBase` for the `Stock` interface. A possible implementation of the `Stock` interface is:

```
public class StockImpl extends StockObjects._StockImplBase {

    private Quote _quote=null;
    private String _description=null;

    public StockImpl(String name, String description) {
        super();
        _description = description;
    }

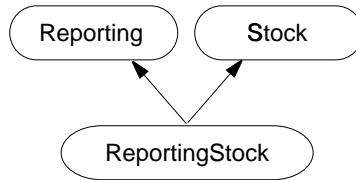
    public Quote get_quote() throws Unknown {
        if (_quote==null) throw new Unknown();
        return _quote;
    }

    public void set_quote(Quote quote) {
        _quote = quote;
    }

    public String description() {
        return _description;
    }
}
```

#### 1.6.1.2. Interface versus Implementation Hierarchies

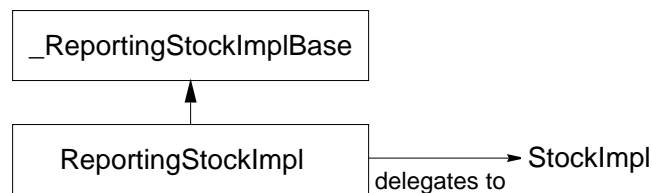
Notice that there are two separate hierarchies: an interface hierarchy and an implementation hierarchy. Recall that the interface hierarchy for the example of a `ReportingStock` is:



In IDL this is represented as:

```
interface ReportingStock: Reporting, Stock {
};
```

Now suppose there is an implementation of a **ReportingStock**, named **ReportingStockImpl**, that inherits the IDL generated skeletons **\_ReportingStockImplBase**, delegates some of its stock methods to **StockImpl**, and implements the **Reporting** operations directly. Graphically:



In the Java programming language, this class hierarchy is represented as:

```
class ReportingStockImpl implements ReportingStock
    extends _ReportingStockImplBase {
    ...
}
```

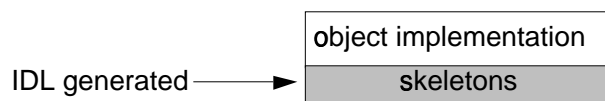
Since the Java programming language only supports single inheritance of implementation classes, implementations often create an instance of another class and delegate to it. In the above example, the **ReportingStockImpl** delegates to the **StockImpl** class for the implementation of some of its methods.

Other class hierarchies implementing the same interface hierarchy are possible. Furthermore, if you need to change the class hierarchy of the implementation in some way, the clients are not affected.

#### 1.6.1.3. Implementation Type Checking

Just as type checking is done at the client for the request to a distributed object, type checking is also done for the object implementation.

The IDL compiler for the Java programming language generates object skeletons and Java code to represent all of the IDL interfaces and data types used in the interface definition. The implementation code thus depends on the generated Java code.



If there are any type errors in the object implementation, the Java compiler, not the IDL compiler, catches the errors at compile time. Thus, in the example, suppose the developer erroneously implemented the `get_quote()` operation to return a double instead of the structure that is declared in the IDL:

```
Quote StockImpl.get_quote() {  
    double price = ...;  
    return price;  
}
```

The Java compiler would detect this error at compile time.

#### 1.6.1.4. Implementing a Server Using the Java 2 ORB

You previously saw how to provide an implementation of a CORBA object in the Java programming language. The remaining task is to define a server that when run makes the services of its objects available to clients. A server that will run with the Java 2 ORB needs to do the following:

- Define a main method
- Initialize the ORB
- Instantiate at least one object

- Connect each object to the orb
- Wait for requests

The server must instantiate at least one object since objects are the only way to offer services in CORBA systems.

Here's an implementation of the stock objects server. This code depends on the Java 2 ORB.

```
public class theServer {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init(args,null);

            // Create a stock object.
            StockImpl theStock =
                new StockImpl("GII","Global Industries Inc.");

            // Let the ORB know about the object
            orb.connect(theStock);

            // Write stringified object reference to a file
            PrintWriter out =
                new PrintWriter(new BufferedWriter(
                    new FileWriter(args[0])));
            out.println( orb.object_to_string(theStock) );
            out.close();

            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        } catch (Exception e) {
            System.err.println("Stock server error:  " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Notice that the server does a **new** on the **StockImpl** class implementing the **Stock** interface and then passes it to the ORB using the **connect()** call, indicating that the object is ready to accept requests. Finally, the server waits for requests.

### 1.6.1.5. Implementing a Server Using VisiBroker 3.x

You previously saw how to provide a server using the Java 2 ORB. If you are using Inprise's VisiBroker 3.x for Java ORB you need to do the following:

- Define a main method
- Initialize the ORB and the BOA (page 29) (basic object adapter)
- Instantiate at least one object
- Let the BOA know that the object is ready to provide service
- Let the BOA know that the server is ready

The server must instantiate at least one object since objects are the only way to offer services in CORBA systems.

Here's an implementation of the stock objects server. This code depends on VisiBroker 3.x:.

```
public class theServer {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb =
                org.omg.CORBA.ORB.init(args,null);

            // Initialize the BOA.
            org.omg.CORBA.BOA boa =
                ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();

            // Create a stock object.
            StockImpl theStock =
                new StockImpl("GII","Global Industries Inc.");

            // Write stringified object reference to a file
            PrintWriter out =
                new PrintWriter(new BufferedWriter(
                    new FileWriter(args[0])));
            out.println( orb.object_to_string(theStock) );
            out.close();

            // Tell the BOA that the object is ready to
            // receive requests.
            boa.obj_is_ready(theStock);

            // Tell the boa that the server is ready. This
            // call blocks.
            boa.impl_is_ready();
        }
    }
}
```

```

    } catch (Exception e) {
        System.err.println("Stock server error:  " + e);
        e.printStackTrace(System.out);
    }
}
}

```

Notice that the server does a **new** on the **StockImpl** class implementing the **Stock** interface and then passes it to the BOA, indicating that the object is ready to accept requests. Finally, the server calls the BOA to indicate that it is ready. At this point, the implementation will be called when requests arrive.

#### 1.6.1.6. Differences Between Server Implementations

The following summarizes the differences between implementing a transient CORBA server using the Java 2 ORB and implementing a transient server using Inprise's VisiBroker 3.x:

	<b>Java 2 ORB</b>	<b>VisiBroker 3.x for Java</b>
<b>Initialization</b>	Just initialize the ORB	Initialize both the ORB and the BOA
<b>Object export</b>	<code>orb.connect(theStock)</code>	<code>boa.obj_is_ready(theStock)</code>
<b>Indicate server ready for requests</b>	Suspend main thread doing a <code>wait()</code>	<code>boa.impl_is_ready()</code>

These are the only differences for transient object servers. There are further API differences of CORBA products due to persistence and automatic activation of servers.

#### 1.6.1.7. Packaging Object Implementations

As illustrated above, you should separate the implementations of your objects from the implementation of the server. This allows you to mix and match object implementations in a server. The object implementation does not depend on the server. The server, of course depends on the object implementations that it contains.

Another advantage of carefully isolating object implementation code from server code is portability. Most of the product-specific code exists in the server, not in the object implementation.

A good strategy is to package an object implementation with its generated stubs and skeletons as a JavaBean component. This allows the implementation to be manipulated by JavaBean



design tools.

## 1.7. Object Adapters

### 1.7.1. Object Adapters

The CORBA specification defines the concept of an *object adapter*. An object adapter is a framework for implementing CORBA objects. It provides an API that object implementations use for various low level services. According to the CORBA specification, an object adapter is responsible for the following functions:

- Generation and interpretation of object references
- Method invocation
- Security of interactions
- Object and implementation activation and deactivation
- Mapping object references to the corresponding object implementations
- Registration of implementations

The architecture supports the definition of many kinds of object adapters. The specification includes the definition of the *basic object adapter* (BOA). In the previous section (page 27), you saw some server code that uses the services of VisiBroker's implementation of the BOA. The BOA has been implemented in various CORBA products. Unfortunately, since the specification of the BOA was not complete, the various BOA implementations differ in some significant ways. This has compromised server portability.

To address this shortcoming, an entirely new object adapter was added, the *portable object adapter* (POA). Unfortunately, the POA is not yet supported in many products. In any event, the BOA and the POA are described here.

#### 1.7.1.1. Activation on Demand by the Basic Object Adapter (BOA)

One of the main tasks of the BOA is to support on-demand object activation. When a client issues a request, the BOA determines if the object is currently running and if so, it delivers the request to the object. If the object is not running, the BOA activates the object and then delivers the request.

The BOA defines four different models for object activation:

<b>Shared server</b>	Multiple active objects share the same server. The server services requests from multiple clients. The server remains active until it is deactivated or exits.
<b>Unshared server</b>	Only one object is active in the server. The server exits when the client that caused its activation exits.
<b>Server-per-method</b>	Each request results in the creation of a server. The server exits when the method completes.
<b>Persistent server</b>	The server is started by an entity other than the BOA (you, operating services, etc.). Multiple active objects share the server.

#### 1.7.1.2. Portable Object Adapter (POA)

According to the specification, "The intent of the POA, as its name suggests, is to provide an object adapter that can be used with multiple ORB implementations with a minimum of rewriting needed to deal with different vendors' implementations." However, most CORBA products do not yet support the POA.

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.

The POA allows the object implementor a lot more control. Previously, the implementation of the object was responsible only for the code that is executed in response to method requests. Now, additionally, the implementor has more control over the object's identity, state, storage, and lifecycle.

The POA has support for many other features, including the following:

- Transparent object activation
- Multiple simultaneous object identities
- Transient objects
- Object ID namespaces
- Policies including multithreading, security, and object management
- Multiple distinct POAs in a single server with different policies and namespaces

For more detail on the POA, please see the specification.

A word on multithreading. Each POA has a threading policy that determines how that particular

POA instance will deal with multiple simultaneous requests. In the single thread model, all requests are processed one at a time. The underlying object implementations can therefore be lazy and thread-unsafe. Of course, this can lead to performance problems. In the alternate ORB-controlled model, the ORB is responsible for creating and allocating threads and sending requests in to the object implementations efficiently. The programmer doesn't need to worry about thread management issues; however, the programmer definitely has to make sure the objects are all thread-safe.

## 1.8. Resources

### 1.8.1. Resources

#### 1.8.1.1. Web Sites

- The OMG web site: <http://www.omg.org>

#### 1.8.1.2. Documentation and Specs

- The CORBA/IIOP Specification can be obtained from the OMG web site ( <http://www.omg.org/library/c2indx.html> )
- *Objects By Value Proposal* can be obtained from the OMG FTP site: <ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf>
- *The VisiBroker For Java Reference* is available from Inprise: <http://www.inprise.com>
- *The VisiBroker For Java Programmers Guide* is available from Inprise: <http://www.inprise.com>
- JDK 1.2 CORBA documentation <http://java.sun.com/products/jdk/1.2/docs/guide/idl/>
- The Java 2 Documentation has extensive information on JavaIDL at the **org.omg.CORBA** package documentation and the JavaIDL Guide documentation.
- Java Transactions
  - ◆ <http://java.sun.com/products/jta/>
  - ◆ <http://java.sun.com/products/jts/>
  - ◆ <ftp://www.omg.org/pub/docs/formal/97-12-17.pdf>
- JavaIDL Naming Service documentation

### 1.8.1.3. Books

- Orfali & Harkey, *Client/Server Programming with Java and CORBA* (Wiley)
- Orfali, Harkey & Edwards, *Instant CORBA* (Wiley)
- Vogel, Andreas and Keith Duddy, *Java Programming with CORBA*(Wiley)
- Perdrick et al, *Programming with VisiBroker* (Wiley).

### 1.8.1.4. Miscellaneous

If you don't have the `idltojava` compiler, you can find it at Sun's JavaIDL web site.

RMI over IIOP compiler is available at <http://java.sun.com/products/rmi-iiop/>.

JacORB - Free Java ORB, including POA, DII, DSI, CosNaming

OrbixWeb from Iona

WebSphere Application Server from IBM

A nice list of ORBs on a nice CORBA information site

## 1.9. Appendix: Java 2 ORB Notes

### 1.9.1. About The Java 2 ORB

The Java IDL ORB that ships with the Java <sup>TM</sup> 2 platform allows applications to run either as stand-alone Java applications or as applets within Java-enabled browsers. It uses IIOP as its native protocol.

The Sun Java ORB is fairly generic. This is good, because there are few surprises; however, there are many advanced features of CORBA that are missing. There is no Interface Repository (though Java IDL clients can access an Interface Repository provided by another Java or C++ ORB), Transaction Service, or POA, for example. For a complete list of these unimplemented features, see the **CORBA Package JavaDoc Comments** – scroll down to find the section near the bottom of the page describing these shortcomings.

Java IDL is structured with a "pluggable ORB" architecture, which allows you to instantiate ORBs from other vendors from within the Java Virtual Machine. This is accomplished through setting environment variables, or system properties, or at run time through the use of a **Properties** or **String[]** object. See the **CORBA Package JavaDoc Comments** for more details (scroll down past the list of classes to find the appropriate sections).

### 1.9.1.1. idltojava Notes

If you don't have the **idltojava** compiler, you can find it at the Java IDL web site.

By default, **idltojava** tries to run a C preprocessor on the IDL files before compiling them. Unfortunately, if you do not have a C preprocessor installed on your system, or if **idltojava** cannot find it, you will see cryptic error message:

```
Bad command or file name
Couldn't open temporary file
idltojava: fatal error: cannot preprocess input;
No such file or directory
```

If you get this message, it means you must invoke **idltojava** with the **-fno-cpp** option, as follows:

```
idltojava -fno-cpp foo.idl
```

### 1.9.1.2. System Properties

The **ORB.init()** method can read in its configuration parameters from a number of different sources: from the application parameters (the first argument to **ORB.init()**), from an application-specific **Properties** object (the second argument to **ORB.init()**), or from the System Properties (defined on the command line by **-D** flags).

Quoted verbatim from the Java IDL guide :

Currently, the following configuration properties are defined for all ORB implementations:

**org.omg.CORBA.ORBClass**

The name of a Java class that implements the **org.omg.CORBA.ORB** interface. Applets and applications do not need to supply this property unless they must have a particular ORB implementation. The value for the Java IDL ORB is **com.sun.CORBA.iiop.ORB**.

**org.omg.CORBA.ORBSingletonClass**

The name of a Java class that implements the **org.omg.CORBA.ORB** interface. This is the object returned by a call to **orb.init()** with no arguments. It is used primarily to create typecode instances than can be shared across untrusted code (such as unsigned applets) in a secured environment. The value for the Java IDL ORB is **com.sun.CORBA.iiop.ORB**.

In addition to the standard properties listed above, Java IDL also supports the following properties:

**org.omg.CORBA.ORBInitialHost**

The host name of a machine running a server or daemon that provides initial bootstrap services, such as a name service. The default value for this property is **localhost** for applications. For applets it is the applet host, equivalent to **getCodeBase().getHost()**.

**org.omg.CORBA.ORBInitialPort**

The port the initial naming service listens to. The default value is **900**.

## **1.10. Appendix: VisiBroker 3.x Notes**

### **1.10.1. VisiBroker 3.x**

Here are some additional details for the VisiBroker 3.x implementation of CORBA. See the product documentation (<http://www.inprise.com/visibroker/>) for more details.

### 1.10.1.1. VisiBroker Tools

VisiBroker for Java ships with a number of tools. Some are replacements or wrappers for the standard Java™ compiler and interpreter. Others are specific to the VisiBroker product. The important ones are:

- **vbj** is a wrapper for **java** which sets some properties and adds to your classpath
- **vbjc** is a wrapper for **javac** which sets some properties and adds to your classpath
- **idl2java** is an IDL compiler that can produce proprietary or portable stubs and skeletons
- **osagent** launches the proprietary Smart Agent binding service

### 1.10.1.2. Using VisiBroker with Java 2

To make VisiBroker for Java 3.4 work with the Java 2 platform, a number of changes are necessary, involving both code and configuration.

The Java 2 platform ships with a standard implementation of CORBA classes in the **org.omg.CORBA.\*** package. These classes are somewhat different from the CORBA classes included with VisiBroker. The VisiBroker classes have several nonstandard extensions to CORBA; some of these nonstandard extensions are required for successful operation. To access these functions, you must change your source code to cast the JavaIDL ORB to a VisiBroker ORB. For example:

```
org.omg.CORBA.ORB orb =
    org.omg.CORBA.ORB.init(args, null);
org.omg.CORBA.BOA boa =
    ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
```

This example applies to the server code.

For this cast to work, you must also guarantee that the ORB returned by the **ORB.init()** call is indeed a VisiBroker ORB, and not the standard JavaIDL ORB from Sun. For that, you need to define two Java system properties before you launch the JVM. These properties are automatically set if you use **vbj** instead of **java** to launch your programs. But if you use **java**, you must set these properties as follows (where each lists the property name first and value second):

- **org.omg.CORBA.ORBClass** - **com.visigenic.vbroker.orb.ORB**
- **org.omg.CORBA.ORBSingletonClass** - **com.visigenic.vbroker.orb.ORB**

### 1.10.1.3. Portable Stubs and Skeletons

By default, VisiBroker for Java creates stub and skeleton code that is interoperable but not portable. This makes sense in the VisiBroker world, since the non-portable code is more efficient and slightly smaller. However, if your code needs to run on several different ORBs, you can use the command

```
idl2java -portable -no_bind Foo.idl
```

and the stubs and skeletons will be portable.

There are a few reasons for this. One is if you're writing an applet that will run inside a remote web browser environment, such as Netscape Communicator or the Java Plug-in. The latter uses JavaIDL; the former may be running an older version of VisiBroker.

What's the difference between portable and proprietary versions? A portable stub uses DII (Dynamic Invocation Interface) to marshal the object request; a portable skeleton uses DSI (Dynamic Skeleton Interface). The proprietary versions make direct calls (to the ORB or the implementation), and hence do not have to go through the overhead of creating and parsing the various DII and DSI objects.

Note that your code doesn't need to change—this all happens behind the scenes with `idl2java`. The only difference in the portable code is that `_FooImplBase` extends `org.omg.CORBA.DynamicImplementation` instead of `com.inprise.vbroker.CORBA.portable.Skeleton`, and that the stub class is named `_portable_stub_Foo.java` instead of `_st_Foo.java`. (Note also that if you really want to, you can switch stubs on the fly by using `FooHelper.setProxyClass(_portable_stub_Foo.class)` — though that would be kind of weird).

### 1.10.1.4. Using the BOA with VisiBroker

The VisiBroker BOA uses a slightly modified version of the standard BOA initialization sequence. For VisiBroker, follow the following boilerplate code.

```
// create and initialize the ORB
ORB orb = ORB.init(args, null);
```

The VisiBroker BOA is a customized, proprietary implementation of the `CORBA.BOA` interface. It has several methods that are not part of the standard interface. In order to use these



proprietary methods, you must **cast** the ORB to a VisiBroker class, as follows.

```
// Initialize the BOA.  
// Must cast to VBJ ORB for Java 2 compatibility  
org.omg.CORBA.BOA boa =  
((com.inprise.vbroker.CORBA.ORB)orb).BOA_init();
```

VisiBroker objects are usually **persistent**. In VisiBroker terms, this means that they are initialized with a name. This is not needed with a portable, non-VisiBroker object implementation.

```
// create object and register it with the ORB  
Stock theStock = new StockImpl(name);
```

The VisiBroker BOA skips the **boa.create()** phase and jumps straight to **obj\_is\_ready()**.

```
// Export the newly created object.  
boa.obj_is_ready(theStock);
```

The **impl\_is\_ready()** method waits indefinitely.

```
// Wait for incoming requests  
boa.impl_is_ready();
```

#### 1.10.1.5. Using the VisiBroker Smart Agent

VisiBroker for Java ships with its own location service, called the smart agent. The smart agent is a distributed location service. It collaborates with other smart agents running on the network to locate a suitable implementation of an object. If there is more than one implementation available; the smart agent selects one. This provides a degree of fault tolerance and load balancing. If a machine goes down, the smart agent will automatically find another implementation on another machine to service the request. The client is unaware of this.

If you create a persistent object, by passing in a name when you call its constructor, then the BOA

will automatically inform the smart agent. The name you passed in the constructor will become the name it is known by on the smart agent network.

On the client side, the proprietary Helper object defines a method **bind()** that fetches an object reference for you, bypassing the need to convert a string into an object reference. The **bind()** method is not part of the standard CORBA-Java mapping.

```
// "bind" (actually lookup) the object reference
Stock theStock = StockHelper.bind(orb, "GII");
```

# Chapter 2. Introduction to CORBA :

## Magercises

Welcome to the MageLang Institute Magercises (page 57) for the *Introduction to CORBA* short course.

These Magercises will give you some initial experience in writing and running simple CORBA applications using the Java <sup>TM</sup> 2 ORB and VisiBroker for Java from Inprise.

When you finish these Magercises, you will know the basic steps for designing, compiling, and running CORBA applications using the Java 2 ORB and VisiBroker.

A help (page 57) document provides a quick explanation of the concepts behind the Magercises.

## 2.1. Magercise: Run the Simple Stock Example

This simple Magercise takes you through the steps of running the simple stock server and a client. Basically it tests the installation of your ORB. This Magercise can be completed with either Sun's Java IDL ORB or Inprise's VisiBroker 3.x for Java.

Educational goal(s):

- Run a simple CORBA application.

### Prerequisites

None.

### Skeleton Files

- stockClient.jar
- stockServer.jar

**Solution** To run the server you need to download the **stockServer.jar** file. To run the client you need to download the **stockServer.jar** file.

- stockClient.jar
- stockServer.jar

### Introduction

In this Magercise, you will run a CORBA server and client that have been provided. Basically it tests the installation of your ORB.

This Magercise can be completed with either Sun's Java IDL ORB or Inprise's VisiBroker 3.x for Java.

**Perform the following tasks:**

1. Download the Jar files for the client and the for the server. (See "Skeleton Code" above.) To run the server you need to download the **stockServer.jar** file. To run the client you need to download the **stockServer.jar** file.
2. Run the programs in the order given below:

<b>run the server</b>	<b>java            StockServer.theServer gii.ior</b>
<b>run the client</b>	<b>java            StockClient.theClient gii.ior</b>

**Notes** Your classpath needs to include the jar file containing the StockServer or StockClient. For VisiBroker you also need to include the files **vbjorb.jar** and **vbjapp.jar** on the classpath. Alternatively, you can use the **vbj** program to run your programs instead of using **java** directly. The **vbj** command adds the **vbjorb.jar** and **vbjapp.jar** files to the classpath appropriately and then invokes Java. If you are using VisiBroker and Java 2, then you need to define the following system properties as well:

3. **org.omg.CORBA.ORBClass=com.visigenic.vbroker.orb.ORB**

## Conclusion

This Magercise has walked you through the steps of running a CORBA server and a CORBA client. In future excercises you will build your own CORBA programs and run them analogously.

## 2.2. Magercise: Stock Example

This Magercise reviews the order of tasks in developing a CORBA application using a minimal example.

Educational goal(s):

- Learn to modify, build and run a simple CORBA application.

### Prerequisites

Run the Simple Stock Example (page 40)

### Skeleton Files

- stock.idl
- StockClient.java
- StockServer.java
- StockImpl.java

### Solution

The following Java source files represent a solution to this Magercise:

- stock.idl
- StockClient.java
- StockServer.java
- StockImpl.java

### Introduction

In this exercise, you will add an operation to the stock IDL file, implement it, generate Java code from the IDL file and then compile and run the distributed stock application.

The order in which you will edit these files is the same order that you will follow for normal CORBA application development. These steps are indicated by notes within square brackets at the beginning of each task.

This exercise can be completed using either Sun's Java IDL ORB or Inprise's VisiBroker 3.x for Java. Notes on these products are available:

- **Sun Java IDL ORB Implementation (page 32)**

- 

**Inprise VisiBroker for Java (page 34)**

If you are using the Java 2 platform, see the notes on VisiBroker vs. Java 2 (page 35).

### Perform the following tasks:

1. [ **Modify the IDL file**] First edit the IDL file. There is already a definition for the module and interface. The interface, however, does not have any way of setting a price quote of a stock. Within the interface definition, add an operation named **setQuote()**. It should take an **in** parameter of type **Quote**. Remember that these are IDL types, not Java types.
2. [ **Run IDL Compiler**] At the command line, run the IDL compiler to generate Java code from the IDL file.

<b>Sun JavaIDL</b>	<b>idltojava -fno-cpp stock.idl</b>
<b>Inprise VisiBroker</b>	<b>idl2java stock.idl</b>

3. [ **Compile the generated Java code**] Now compile the generated Java code using the standard Java compiler. Compile it in the generated **StockObjects** directory.

<b>Sun JavaIDL</b>	<b>javac *.java</b>
<b>Inprise VisiBroker</b>	<b>vbjc *.java</b>

4. [ **Examine the generated Java files**] All the files will be generated into the **StockObjects** directory because the IDL file defines a module **StockObjects**. (The purpose of each file is described in more detail in the course notes (page 20).) Look carefully at the file **Stock.java** - - this is the interface that you will have to implement.

5. [ **Create "Impl" Classes**] The skeleton code file we have given you, **stockImpl.java** has most of the implementation you will need. To finish, add a body to the method **setQuote()**. Note: Be sure not to confuse the skeleton files that Magelang provides as part of our Magercises with CORBA skeleton files. The Magelang files appear as links at the beginning of the Magercise, and are files that you are to use as a starting point for coding. CORBA skeleton files are generated from IDL files, end with the name **ImplBase** and are used for creating classes that implement CORBA interfaces.
6. [ **Create Server Class**] Now look at the file **stockServer.java**. This class is a java application that starts up the orb, instantiates a **stockImpl** instance, connects it to the ORB, and prints out a stringified reference to it. There is nothing for you to do except look at the file. Look carefully at each line.
7. [ **Create Client Class**] In the file **stockClient.java**, you will find code to instantiate the orb, read in a stringified reference to the remote object, and call the **getQuote** operation on it. Again, there is nothing for you to do except look at the file. Look carefully at each line. Note: Most of the Magercises are applications that have separate "server" and "client" portions. The server portions instantiate distributed objects, the client portions use them. This is not a requirement of CORBA programs, any application that is part of a CORBA environment can both instantiate and use distributed objects. The client and server pattern is used by the Magercises for convenience and consistency. Some later Magercises dispense with this convention entirely.
8. [ **Compile the program**] Now compile the program using the standard Java compiler.

<b>Sun JavaIDL</b>	<b>javac -classpath . *.java</b>
<b>Inprise VisiBroker</b>	<b>vbjc *.java</b>

9. [ **Run the Server program**] The server needs to run as a separate process. In DOS/Windows, you will use the **start** command. In UNIX, you will use the **&** (ampersand). The Server outputs an IOR (or stringified reference) to a file and to the console for the Stock object it creates. The name of the file is passed as the first argument to the server. To run the server, use the normal **java** command. If you are using VisiBroker, you can use the **vbj** command instead. **vbj** is just like **java**, except it sets up the CLASSPATH variables automatically. Putting it all together:



<b>Sun JavaIDL</b>	<code>java -cp . StockServer gii.ior</code>
<b>Inprise VisiBroker</b>	<code>vbj StockServer gii.ior</code>

10. [ **Run the Client program**] The client reads the stringified object reference of the stock object created by your server. A file containing the stringified object reference is passed as an argument to the client.

<b>Sun JavaIDL</b>	<code>java -cp . StockClient gii.ior</code>
<b>Inprise VisiBroker</b>	<code>vbj StockClient gii.ior</code>

11. [ **Kill the Server**] Don't forget to kill the server.

## Conclusion

This exercise has walked you through the steps of writing, compiling, and running a portable CORBA application. Most other exercises will follow a similar pattern.

### 2.3. Magercise: Dynamic Stock Example

This Magercise adds dynamic creation of stock objects to the previous stock server.

Educational goal(s):

- Learn about factories in a simple CORBA application.

#### Prerequisites

Stock Example (page 42)

#### Skeleton Files

- stock.idl
- StockFactoryImpl.java
- StockImpl.java
- DynamicStockServer.java
- DynamicStockClient.java

#### Solution

The following Java source files represent a solution to this Magercise:

- stock.idl
- StockFactoryImpl.java
- StockImpl.java
- DynamicStockServer.java
- DynamicStockClient.java

#### Introduction

In this Magercise, you will add a factory interface with a single create

operation to the stock IDL file, implement it, generate Java code from the IDL file and then compile and run the distributed stock application.

This Magercise can be completed using either Sun's Java IDL ORB or Inprise's VisiBroker 3.x for Java. Notes on these products are available:

- **Sun JavaIDL ORB Implementation (page 32)**

- 

**Inprise VisiBroker for Java (page 34)**

If you are using the Java 2 platform, see the notes on Using VisiBroker with Java 2 (page 35).

### Perform the following tasks:

1. [ **Modify IDL file**] First edit the IDL file. The `StockObjects` module in the IDL file defines the interface to a stock object. But there is currently no way for a client to create stock objects. You need to add another interface, **`StockFactory`** that has a single operation to create a stock and return a reference to the created stock. The create operation should take two string parameters: one giving the stock symbol and the other giving a description of the company.
2. [ **Run IDL Compiler**] At the command line, run the IDL compiler to generate Java code from the IDL file.

<b>Sun JavaIDL</b>	<code>idltojava -fno-cpp stock.idl</code>
<b>Inprise VisiBroker</b>	<code>idl2java stock.idl</code>

3. [ **Compile the generated Java code**] Now compile the generated Java code using the standard Java compiler. Compile it in the generated **`StockObjects`** directory.

<b>Sun JavaIDL</b>	<code>javac *.java</code>
<b>Inprise VisiBroker</b>	<code>vbjc *.java</code>

4. [ **Examine the generated Java files**] All the files will be generated into the **`StockObjects`** directory because the IDL file defines a module `StockObjects`. (The purpose of each file is described in more detail in the course notes (page 20).) Look carefully at the file **`StockFactory.java`** -- this is the interface that you will have to implement.

5. [ **Implement the stock factory**] The skeleton code file we have given you, **StockFactoryImpl.java** has most of the implementation you will need. To finish, add a body to the method **create\_stock**.
6. [ **Create Server Class**] Now look at the file **DynamicStockServer.java**. This class is a java application that starts up the orb, instantiates a **StockFactoryImpl** instance, connects it to the ORB, and prints out a stringified reference to it. There is nothing for you to do except look at the file. Look carefully at each line.
7. [ **Create Client Class**] In the file **DynamicStockClient.java**, you will find code to instantiate the orb, read in a stringified reference to the remote factory, and call the **create\_stock** operation on it. Again, there is nothing for you to do except look at the file. Look carefully at each line.
8. [ **Compile the program**] Now compile the program using the standard Java compiler.

<b>Sun JavaIDL</b>	<b>javac -classpath . *.java</b>
<b>Inprise VisiBroker</b>	<b>vbjc *.java</b>

9. [ **Run the Server program**] The server needs to run as a separate process. In DOS/Windows, you will use the **start** command. In UNIX, you will use the **&** (ampersand). The Server outputs an IOR (or stringified reference) to a file and to the console for the Stock factory object it creates. The name of the file is passed as the first argument to the server. To run the server, use the normal **java** command. If you are using VisiBroker, you can use the **vbj** command instead. **vbj** is just like **java**, except it sets up the CLASSPATH variables automatically. Putting it all together:

<b>Sun JavaIDL</b>	<b>java -cp . DynamicStockServer stockfactory.ior</b>
<b>Inprise VisiBroker</b>	<b>vbj DynamicStockServer stockfactory.ior</b>

10. [ **Run the Client program**] The client reads the stringified object reference of the stock factory object created by your server. An file containing the stringified object reference is passed as the first argument to the client. The second and third arguments passed to the client is the stock symbol and company description of the stock to be created. Note the following commands are each on a single line.

<b>Sun JavaIDL</b>	<code>java -cp . DynamicStockClient stockfactory.ior gii "Global Industries Inc."</code>
<b>Inprise VisiBroker</b>	<code>vbj DynamicStockClient stock- factory.ior gii "Global Indus- tries Inc."</code>

11. **[Kill the Server]** Don't forget to kill the server.

## Conclusion

This Magercise has walked you through the steps of adding dynamic distributed object creation by means of a factory object. As you probably observed, a factory object is just like any other CORBA object.

## 2.4. Magercise: A Message Box

In this Magercise you will create the implementation class for an IDL interface.

Educational goal(s):

- Learn how to write interface implementation classes.
- Call methods on a distributed object.
- Throw and catch CORBA User Exceptions

This Magercise is meant for use with the *VisiBroker for Java* ORB from Inprise.

### Prerequisites

Stock Example (page 42)

### Skeleton Files

- messageBox.idl
- MBClient.java
- MBServer.java
- MessageBoxImpl.java

### Solution

The following Java source files represent a solution to this Magercise:

- solution/messageBox.idl
- solution/MBClient.java
- solution/MBServer.java
- solution/MessageBoxImpl.java

## Introduction

The application in this Magercise is a simple message box, like a telephone answering machine. The `MessageBox` interface has an operation for leaving messages, another for the owner to get the messages, and an attribute for setting the reply that is sent out when a message is left. Here is the IDL file:

```
module MessageModule {

    typedef sequence<string> MessageSeq;

    interface MessageBox {
        attribute string reply;

        string leaveMessage(in string msg)
            raises (boxFull);

        MessageSeq getMessages();
    };

};
```

Your task is to create a `MessageBoxImpl` class to implement this interface. You will also add methods to the `MBClient` class to test your `MessageBoxImpl`.

### Perform the following tasks:

1. First generate the IDL file.
2. Examine the generated Java interface in `MessageModule/MessageBox.java`. Using the given skeleton file, `MessageBoxImpl.java`, implement the necessary methods. Don't forget to add a constructor. It should take a string argument as a name, and call the superclass constructor to set the name of the object. Notice how the sequence is generated. No `MessageSeq` class is created; instead you use the Java type `String[]`. There are, however, helper and holder classes generated for `MessageSeq`.
3. Modify the `MBServer` class to instantiate a `MessageBoxImpl` with a name of your choosing. Remember, you want this name to be unique to avoid conflicts with other ORBs on your network.
4. Look at the `MBClient` source code provided for you. Make sure you understand how the command-line parameters are parsed. The usage of `MBClient` is:

```
vbj MClient servername command param
vbj MClient servername leave message
vbj MClient servername reply reply-message
vbj MClient servername get
```

5. Finish the skeleton version of **MClient** by adding code as specified by the comments. You should call the **reply** setter to set an appropriate reply, call **leaveMessage** to leave some messages, and retrieve your messages using **getMessages** so they can be printed out.
6. Now compile the program, using **vbjc**.
7. Run the server: **start vbj MServer**. Check that it is running ok.
8. Run the client: **vbj MClient servername command param**. Check the output of your program. See the *expected behavior* section for more detail on the correct output for your program.
- 9.

If you happen to have multiple machines, run the program on each and leave and retrieve messages on the other message boxes.



## 2.5. Magercise: Writing Callbacks

This Magercise shows how to create a pair of distributed objects that communicate with each other as peers.

Educational goal(s):

- Write code to instantiate ORB-aware objects.
- Create callbacks from one distributed object to another.

### Prerequisites

Stock Example (page 42)

### Skeleton Files

- pingpong.idl
- PingObjectImpl.java
- PongObjectImpl.java
- PingServer.java
- PongServer.java

### Solution

The following Java source files represent a solution to this Magercise:

- solution/pingpong.idl
- solution/PingObjectImpl.java
- solution/PongObjectImpl.java
- solution/PingServer.java
- solution/PongServer.java

### Introduction

Unlike a client/server system, the objects in CORBA application are peers: none is intrinsically more important than any other. You may decide to design your program using various kinds of Client/Server metaphors, but that decision is yours to make. The ORB just sees objects communicating with each other over a network.

In this Magercise there is a pair of peer objects, **PingObject** and **PongObject**, for which Java implementations have already been written: **PingObjectImpl** and **PongObjectImpl**. Their behavior is such that when the **PingObject** is pinged, it pongs the given **PongObject**. When **PongObject** is ponged, it returns the favor by pinging the given **PingObject**. To keep things from getting out of hand, the **PingObject** keeps track of the number of times it has been pinged, and stops after a maximum number of pings.

Your task is to create two Java classes, **PingServer** and **PongServer** whose **main** methods will instantiate either a **PingObject** or a **PongObject** respectively, and call the appropriate methods to start the pings and pongs going.

### Perform the following tasks:

1. First take a look at the IDL file, **pingpong.idl**, and generate it.
2. Examine the two implementation classes, **PingObjectImpl** and **PongObjectImpl** to see how they work and how to use them.
3. Using the skeleton code file as a starting point, modify the class **PingServer** to add all the code necessary to start up the ORB and BOA, create a **PingObject** instance, set the maximum number of pings, etc. There are comments to guide your way. Be sure to pick a unique name for your **PingObject** instance.
4. Now fill out the **PongServer** class. The code needed here will be very similar to that for **PingServer**, but you will also initiate the ping-pong exchange.
5. Compile your program and run it to see what happens.

### Conclusion

The timing information printed out when the solution is run gives you some idea of the latency involved when calling methods on a distributed object. For extra credit, you can modify the implementation classes to print out the time difference between pings and pongs.





# Appendix A

## About Magercises

A *Magercise* is a flexible exercise designed to provide help according to the needs of the student. For example, some students will simply complete the exercise given the information and the task list in the Magercise body; some students may want a few hints while others may want a step-by-step guide to successfully complete a particular Magercise. Students may use as much or as little help as they need per Magercise. Moreover, since complete solutions are also provided, students can skip a few Magercises and still be able to complete future Magercises requiring the skipped ones.

## The Anatomy of A Magercise

Each Magercise has a list of any prerequisite Magercises, a list of skeleton code for you to start with, links to necessary API pages, and a text description of the Magercise goal. In addition, the following information is available via five buttons:

- **Expected behavior:** Launches an applet illustrating the desired behavior from your applet.
- **Table of contents:** Brings up the table of contents for the course notes and the list of magercises.
- **Help:** Gives you help or hints on the current Magercise (an annotated solution).
- **Solution:** The `< applet >` tag and Java source resulting in the expected behavior.
- **API Documentation:** A link directly to the online API documentation.

## Magercise Design Goals

There are three fundamental magercise types:

### "Blank screen"

The programmer is confronted with a "blank screen"; i.e., the programmer creates the entire desired functionality.

### Extension

The programmer extends the functionality of an existing, correctly-working program.

**Repair**

The programmer repairs undesirable behavior in an existing program.

Where possible, **the programmer shall be relieved from chores that are irrelevant or unrelated to the technique or concept under examination.**

Where reasonable, **a common thread shall run through the magercises** for each lab section.

Given the constraints of the technique or concept under examination, the **magercises shall be made as interesting or useful as possible** without presenting an overly-complex programming problem to the student.

**Magercises shall execute via the web** unless a particular concept related to non-web execution is required or the browser does not support the capabilities yet. In addition, magercises that must access Java features or library elements causing web security violations are not executed on the web.