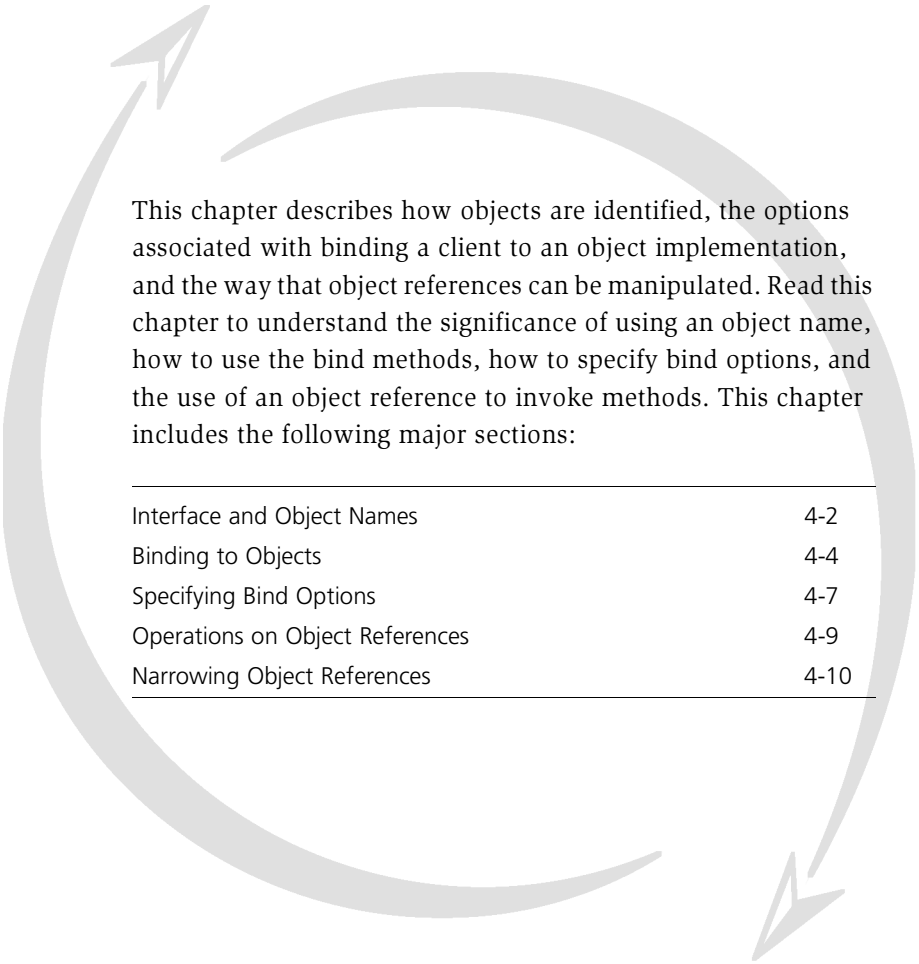
CHAPTER 4

NAMING AND BINDING TO OBJECTS



This chapter describes how objects are identified, the options associated with binding a client to an object implementation, and the way that object references can be manipulated. Read this chapter to understand the significance of using an object name, how to use the bind methods, how to specify bind options, and the use of an object reference to invoke methods. This chapter includes the following major sections:

Interface and Object Names	4-2
Binding to Objects	4-4
Specifying Bind Options	4-7
Operations on Object References	4-9
Narrowing Object References	4-10

INTERFACE AND OBJECT NAMES

When you define an object's interface in an IDL specification, you give it an *interface name*. For example, the `Account` interface introduced in Chapter 2 was given the name "Account" in the IDL specification.

IDL

```
module Bank {
    interface Account {
      float balance();
    };
    interface AccountManager {
      Account open(in string name);
    };
};
```

IDL Sample 4-1 Bank.idl

Object Names

When creating an object, a server must specify an *object name* if the object is to be made available to client applications through the osagent. When the server calls the `BOA.obj_is_ready` method, the object's interface name will only be registered with the VisiBroker `osagent` if the object is named. Objects that are given an object name when they are created return *persistent object references*. See Chapter 5 for a complete discussion of persistent and transient objects.

NOTE *If you want an application to bind to a particular server, specify an object name.*

The use of an object name by your client application is required if it plans to bind to more than one instance of an object at a time. The object name distinguishes between multiple instances of an interface. If an object name is not specified when `bind` is called, the `osagent` will return any suitable object with the specified interface.

For information about all of the available methods, see the *VisiBroker for Java Reference Manual*.

Using Qualified Object Names with Servers

Consider the `Bank` example from Chapter 2 and imagine that you need to have two `AccountManager` objects available; one for a bank in Chicago and one for another bank in Dallas. You may even want to implement two separate object servers, possibly on different hosts.

Each server could instantiate its own `AccountManager` object, but each would use the `AccountManager` constructor that accepts an object name. Code Sample 4-1 shows the server code for creating an `AccountManager` named "Dallas." Code Sample 4-2 shows how a client would then connect to the AccountManager for Dallas.

CODE

```
// Server.java

public class Server {
  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Initialize the BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Create the account manager object.
    Bank.AccountManager manager =
      new AccountManagerImpl("Dallas");
    // Export the newly created object.
    boa.obj_is_ready(manager);
    System.out.println(manager + " is ready.");
    // Wait for incoming requests
    boa.impl_is_ready();
  }
}
```

Code Sample 4-1 Creating an `AccountManager` *with the object name Dallas.*

CODE

```
// Client.java

public class Client {
  public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Locate an account manager.
    Bank.AccountManager manager =
      Bank.AccountManagerHelper.bind(orb, "Dallas");
    // use args[0] as the account name, or a default.
    String name = args.length > 0 ? args[0] : "Jack B. Quick";
    // Request the account manager to open a named account.
    Bank.Account account = manager.open(name);
    // Get the balance of the account.
    float balance = account.balance();
    // Print out the balance.
    System.out.println
      ("The balance in " + name + "'s account is $" + balance);
  }
}
```

Code Sample 4-2 A client binding to an `AccountManager` *with the object name Dallas.*

BINDING TO OBJECTS

Before your client application can invoke methods on an interface, it must first obtain an object reference using the `bind` method. Code Sample 4-3 shows the bind methods generated for the `AccountManagerHelper` class.

All forms of the `bind` method require that an org.omg.CORBA.ORB object be supplied as the first parameter. The first form has no additional parameters and will result in your client being connected to any suitable implementation of the `AccountManager` object. The second form allows your client to specify an object name. In this second form of`bind`, if the object name is a string for a URL, the ORB uses URL Naming to bind to the object. For more information about URL naming, see Chapter 12. The third form allows you to specify an object name, a host name, and bind options. Bind options are described on page 4–7.

CODE

```
abstract public class AccountManagerHelper {
  public static Bank.AccountManager bind(org.omg.CORBA.ORB orb) {
    ...
  }
  public static Bank.AccountManager bind(org.omg.CORBA.ORB orb,
                java.lang.String name) {
    ...
  }
  public static Bank.AccountManager bind(org.omg.CORBA.ORB orb,
                java.lang.String name, java.lang.String host,
                org.omg.CORBA.BindOptions options) {
    ...
  }
...
}
```

Code Sample 4-3 The `bind` *methods generated for* `AccountManagerHelper`.

NOTE *The static bind operations on the Helper class require that an instance of org.omg.CORBA.ORB be supplied.*

The Bind Process

When your client application invokes the `bind` method, the ORB performs several functions on behalf of your application.

- The ORB contacts the `osagent` to locate an object server that is offering the requested **interface**. If an **object name** and a host name (or IP address) are specified, they will be used to further qualify the directory service search.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client application.
- If the connection is successfully established, the ORB will create a proxy object, if necessary, and return a reference to that object.

NOTE *The ORB is not a separate process. It is a collection of classes and other resources that allow communication between clients and servers.*

Client and Server on Different Hosts

When a client requests an object that resides on a remote host, a TCP/IP connection will be established between the client and object server. The ORB will instantiate a stub (stubs are also called proxy objects) for your client to use. A method invoked on the proxy object will be written to a buffer, or marshalled, as a request and sent to the server on the remote host. The server on the remote host will un-marshal the request, invoke the desired method, and send the results back to the client.

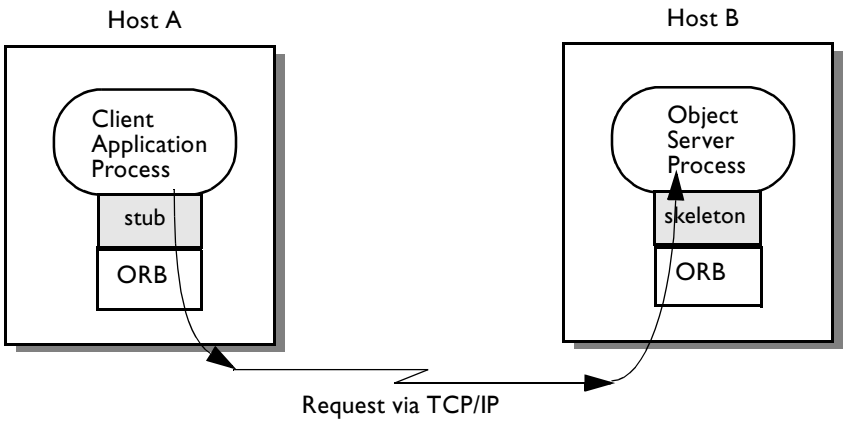


Figure 4-1 Client and Server processes on different hosts.

Client and Server in a Single Process

The previous discussions have assumed that object implementations have taken the form of a server process. While this is often the case, a client application and the object implementation can both be packaged inside a single process. When your client application invokes a `bind` in this scenario, the ORB will return a pointer to the object implementation itself. That pointer will be widened to the object type used by your client application. All methods invoked on your client's object will get called directly as **Java methods** on the object implementation. The ORB will be involved only during the bind process.

SPECIFYING BIND OPTIONS

This section describes options that you can use to control the behavior of the `bind` process. Code Sample 4-4 shows the `bind` method generated by the `idl2java` complier for the `AccountManagerHelper` class that allows you to specify bind options.

CODE

```
abstract public class AccountManagerHelper {
  ...
  public static Bank.AccountManager bind(org.omg.CORBA.ORB orb,
           java.lang.String name, java.lang.String host,
           org.omg.CORBA.BindOptions options) {
    ...
  }
...
}
```

Code Sample 4-4 The `BindOptions` *parameter of the* `bind` *method.*

BindOptions

The `BindOptions` class is used for the fourth parameter to the `bind` method shown in Code Sample 4-4. The `BindOptions` class enables you to control various aspects of the connection between the client application and the object implementation. This class has two boolean values, `defer_bind` and `enable_rebind`. See the following figure for an example.

CODE

```
...
    // Initialize bind options.
    org.omg.CORBA.BindOptions bindOptions = new org.omg.CORBA.BindOptions();
    bindOptions.defer_bind = false;     // do not defer bind
    bindOptions.enable_rebind = false;  // do not attempt to rebind
    // Locate an account manager.
    Bank.AccountManager manager =
      Bank.AccountManagerHelper.bind(orb, "BankManager",
                                     null, bindOptions);
 ...
```

Code Sample 4-5 Using the BindOptions class' `defer_bind` *and* `enable_bind`.

DEFERRING BINDS

When you set `defer_bind` to `true`, the `bind` method creates a proxy object (if necessary) and returns an object reference to your client application. A connection will not be established with the object implementation until your client application actually invokes a method on the object. If you set `defer_bind` to `false`, the connection will be established when `bind` is invoked. The default behavior is to establish the connection at the time the `bind` method is invoked.

ENABLING RE-BINDS

If the TCP connection between your client application and the object implementation fails because of a network error, VisiBroker will automatically attempt to re-bind to the server process or a replica of that server. Network errors include the following:

- a process fails
- a machine crashes
- a network crashes

This fault tolerant processing is described in Chapter 10. If you wish to enable this re-binding process, you must set `enable_rebind` to `true`. If you wish to prevent this re-binding process, set `enable_rebind` to `false`. The default `bind` behavior is to attempt to re-bind to the server if an error occurs.

OPERATIONS ON OBJECT REFERENCES

The object reference returned to your client application by the `bind` method represents an ORB object. Your client application can use the object reference to invoke methods on the object that have been defined in the object's IDL interface specification. In addition, there are methods that all ORB objects inherit from the class `org.omg.CORBA.Object` that you can use to manipulate the object. Some of these methods are listed below.

NOTE *You cannot use the* `instanceof` *keyword to determine the runtime type.*

`_clone`	Creates a copy of the object. It also creates another TCP/IP connection. Using this method is necessary only with thread-per-session or when you are working with other ORBs.
`_is_a`	Determines if an object implements a specified interface.
`_is_bound`	Returns `true` if a connection is currently active for this object.
`_is_equivalent`	Returns `true` if two objects refer to the same interface implementation.
`_is_local`	Returns `true` if this object is implemented in the local address space.
`_is_remote`	Returns `true` if this object's implementation does not reside in the local address space.
`_object_name`	Returns this object's name.
`_repository_id`	Returns the repository's type identifier.

Table 4-1 From `org.omg.CORBA.Object`*, some of the methods on object references*

NOTE *Table 4-1 shows a partial list of methods that you can use to manipulate an object. Check the VisiBroker for Java Reference Manual for a complete list with details.*

Converting an Object Reference to a String

Object references are opaque and can vary from one ORB to another, so VisiBroker provides methods that allow you to convert an object reference to a string as well as convert a

string back into an object reference. The CORBA specification refers to this process as "stringification".

`object_to_string`	Converts an object reference to a string.
`string_to_object`	Converts a string back to an object reference.

Table 4-2 Methods that convert an object reference to or from a string.

NARROWING OBJECT REFERENCES

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*.

NOTE *You cannot use the Java casting facilities for narrowing.*

VisiBroker maintains a typegraph for each object interface so that narrowing can be accomplished by using the object's `narrow` method. If the `narrow` method determines it is not possible to narrow an object to the type you request, it will return `NULL`.

CODE

```
abstract public class AccountManagerHelper {
  ...
  public static Bank.AccountManager narrow(org.omg.CORBA.Object object) {
    ...
  }
  ...
}
```

Code Sample 4-6 The `narrow` *method generated for the* `AccountManager`.