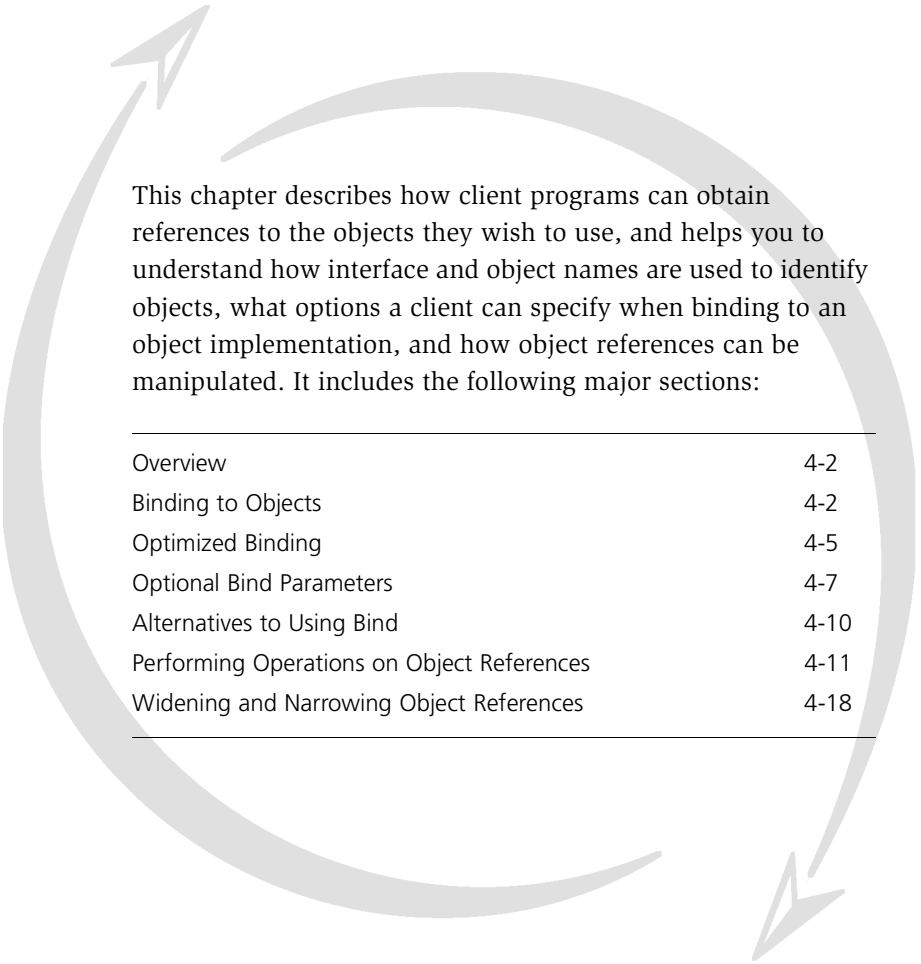
CHAPTER 4

ACCESSING DISTRIBUTED OBJECTS WITH OBJECT REFERENCES



This chapter describes how client programs can obtain references to the objects they wish to use, and helps you to understand how interface and object names are used to identify objects, what options a client can specify when binding to an object implementation, and how object references can be manipulated. It includes the following major sections:

Overview	4-2
Binding to Objects	4-2
Optimized Binding	4-5
Optional Bind Parameters	4-7
Alternatives to Using Bind	4-10
Performing Operations on Object References	4-11
Widening and Narrowing Object References	4-18

OVERVIEW

To invoke operations on an object, your client program must first obtain its object reference. This is usually accomplished by using a bind method and specifying the object's type. The ORB locates the host that offers an object with the requested type. If the server that implements the requested object is not currently executing, the ORB can ensure that the appropriate server is started. After the bind is completed, the client program can use the object reference to invoke operations on the object. The client's method invocations are translated into operation requests that are sent to the object.

An object reference may also be converted to a string, in a process known as *stringification*. This string may then be written to a file or passed to another client. At any time, a *stringified* object reference can be converted to an object reference. Then, a client can use it without binding again.

You can also use the VisiBroker Naming Service to locate objects using logical assigned names. The VisiBroker Naming Service is packaged separately.

BINDING TO OBJECTS

When you define an object interface in an IDL specification, you assign it an interface name. IDL Sample 4-1 shows how the `Account` object, introduced in Chapter 3, "Quick Start for Development with VisiBroker for C++," was given the name "Account" in the IDL specification. The `idl2cpp` compiler generates a C++ class with this interface name and provides a static `_bind()` method that your client programs can use to obtain a reference to the object.

IDL
```
interface Account {
        float balance();
};
```

IDL Sample 4-1 Assigning an interface name to an object in IDL.

Using Interface Names

The repository ID of the interface implemented by an object is registered with the VisiBroker Smart Agent when a server containing that object invokes the `BOA::obj_is_ready()` method for that object. The interface name is a less specific form of the repository ID by which an object is identified when your client program invokes the `_bind()` method.

For information on obtaining an interface name from an object reference, see "Obtaining Object and Interface Names" on page 4-15.

When to Use Object Names

A server must specify an object name when instantiating an object if the object is to be registered with the Smart Agent and made available to client programs through the `_bind()` method. The `idl2cpp` compiler generates a `NULL` object name as a default parameter for the object's constructor. If two or more servers plan to offer objects with the same interface name, each server can assign a unique object name to their object so that clients can distinguish between the various object instances. You must also assign an object name if you plan to register an object implementation with the Object Activation Daemon, described in Chapter 5, "Activating Objects and Implementations."

Specifying Object Names

Consider the account example from Chapter 3, "Quick Start for Development with VisiBroker for C++" and imagine that you need to have two `Account` objects available—one for Jack B. Quick and one for Jane Doe. You may even want to implement separate servers for each account, possibly executing on different hosts. Each server would instantiate a particular `Account` object, but each would specify a different object name. Code Sample 4-1 shows the server changes that you would need to make to create two `Account` objects with different object names.

CODE

```
...
int main(int argc, char* const* argv) {
        try {
          // Initialize the ORB and BOA.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
          CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);

          // Create an account object for Jack B. Quick
          AccountImpl quick("Jack B. Quick");
          // Export the newly created Quick account.
          boa->obj_is_ready(&quick);
          cout << quick << " is ready." << endl;

          // Create an account object for Jane Doe
          AccountImpl doe("Jane Doe");
          // Export the newly created Quick account.
          boa->obj_is_ready(&doe);
          cout << doe << " is ready." << endl;

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

Code Sample 4-1 Specifying an object name for each object's implementation.

Binding to Specific Object Implementations

Your client program is not required to specify an object name when binding to an object if the same service is available from multiple servers or if there is only one server that implements the object. The `idl2cpp` compiler generates a default, `NULL` parameter for the object name for the `_bind()` method.

Expanding the account example to represent two different accounts will require that you modify the client program's `_bind()` invocation to specify a particular `Account` object. Code Sample 4-2 shows the original client code used to bind to a default object that offers the `Account` interface.

CODE
```
...
int main(int argc, char* const* argv) {
      try {
          // Initialize the ORB.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

          // Bind to an account.
          Account_var account = Account::_bind();
          ...
}
```

Code Sample 4-2 The use of the `_bind()` method without an object name.

Code Sample 4-3 shows how you would modify the client program to specify an object name with the `_bind()` method.

CODE
```
...
int main(int argc, char* const* argv) {
      try {
          // Initialize the ORB.
          CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

          // Bind to an account.
          Account_var account = Account::_bind("Jack B. Quick");
          ...
}
```

Code Sample 4-3 The modified `_bind()` method, using an object name.

Actions Performed During the _bind() Process

When your client program invokes the `_bind()` method, the ORB performs several functions on behalf of your program.

- The ORB contacts the Smart Agent to locate an object implementation that offers the requested interface. If an object name was specified when `_bind()` was invoked, that name will be used to further qualify the directory service search. The Object Activation Daemon (OAD), described in Chapter 5, "Activating Objects and Implementations," may be involved in this process if the server object has been registered with the OAD.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client program.
- Once the connection is successfully established, the ORB will create a proxy object and return a reference to that object. The client will invoke methods on the proxy object which will, in turn, interact with the server object. If the object implementation is located in the same process as the client, a proxy object is not created. Instead, a pointer to the object implementation itself is returned.

NOTE *Your client program will never invoke a constructor for the server class. Instead, an object reference is obtained by invoking the static `_bind()` method.*

OPTIMIZED BINDING

VisiBroker automatically determines the most efficient way to establish a connection between a client program and an object implementation.

Binding to an Object on a Remote Host

If VisiBroker determines the object implementation you have requested resides on a remote host, a TCP/IP connection will be established between the client and server object. A *proxy object* will be created for your client to use. All methods invoked on the proxy object will be packaged as operation requests and sent to the server on the remote host. The server will unpack the request, invoke the desired method, and send the results back to your client. VisiBroker uses the Internet Inter-ORB Protocol (IIOP) to package the client requests and server responses, which allows interoperability with other ORB products that support IIOP.

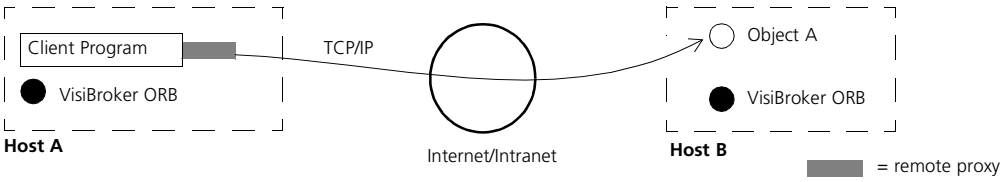


Figure 4-1 Binding when client and server processes are located on different hosts.

Binding to an Object on the Same Host

W If the ORB determines that the requested object implementation resides on the local Windows host, a connection will be established between the client and server object using shared memory—only if both the client and server are multi-threaded. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server using shared memory.

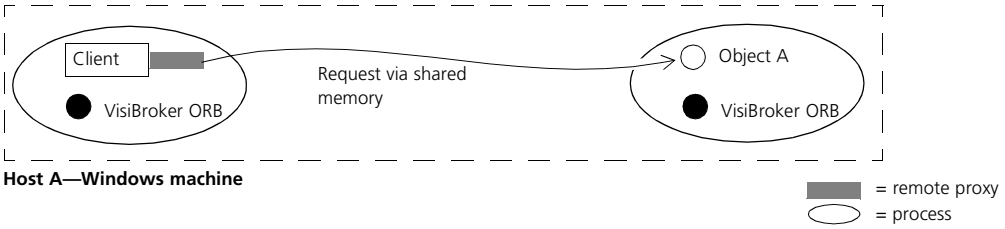


Figure 4-2 Binding when multithreaded client and server processes are located on the same Windows host.

U If the ORB determines that the requested object implementation resides on the local UNIX host, a connection will be established between the client and server object using UNIX Domain Protocol—only if both the client and server are multi-threaded. The ORB will instantiate a proxy object for your client to use. All methods invoked on the proxy object will be packaged as requests and sent to the server using UNIX Domain Protocol.

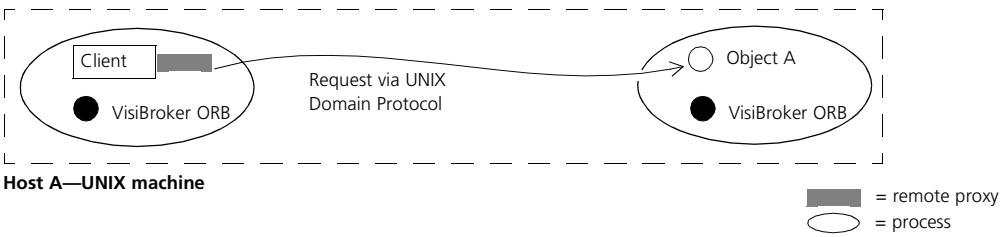


Figure 4-3 Binding when multithreaded client and server processes are located on the same UNIX host.

Binding to an Object in a Single Process

The previous discussions have assumed that object implementations have taken the form of a server process. Although this is often the case, a client program and an object implementation can also be packaged within a single process. When your client program 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 program. All methods invoked on your client's object will be invoked as C++ virtual functions on the object implementation. The ORB will not be involved after the bind process.

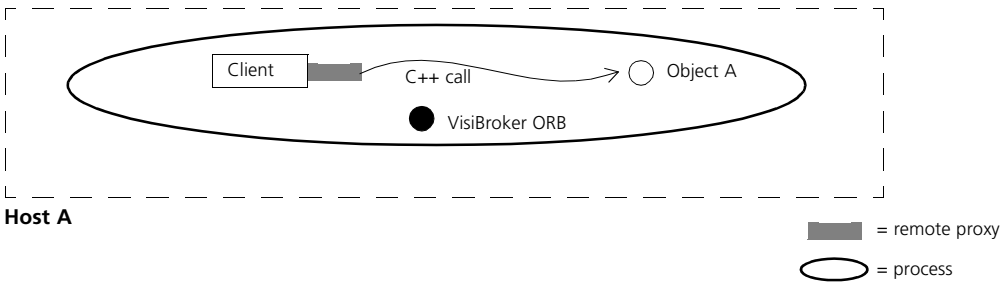


Figure 4-4 Binding when the client and object implementation are located in the same process.

OPTIONAL BIND PARAMETERS

This section describes options that you can use to control the behavior of the `_bind()` method. Code Sample 4-4 shows the `_bind()` method generated for the `Account` interface by the IDL compiler. The default value for all of the parameters is `NULL`.

CODE
```
class Account: public virtual CORBA::Object
{...
     static Account_ptr _bind(const char *object_name= NULL,
                              const char *host_name= NULL,
                              const CORBA::BindOptions* opt = NULL,
                              CORBA::ORB_ptr orb=NULL);
     ...
};
```

Code Sample 4-4 The `_bind` method generated for the `Account` class.

Binding to a Particular Host

In addition to the object name, your client program can specify a particular host on which a desired object implementation is executing. This can be useful if your program knows that a particular object implementation is located on a particular host. The `host_name`

parameter is a string containing either the host's fully qualified name or IP address. If you do not specify this parameter, the ORB will locate an implementation that meets all of the other bind parameters.

Controlling the Connection to the Object with BindOptions

Code Sample 4-5 shows the `BindOptions` structure, passed as the third argument to the `_bind()` method, which enables you to control various aspects of the connection between the client program and the object implementation. If the third parameter is `NULL`, the default bind options will be used. Each of the structure's members will be discussed in the following sections.

CODE
```
struct BindOptions {
    CORBA::Boolean defer_bind;
    CORBA::Boolean enable_rebind;
    CORBA::ULong   send_timeout;
    CORBA::ULong   receive_timeout;
    CORBA::ULong   connection_timeout;
};
```

Code Sample 4-5 The `BindOptions` structure.

DEFERRING BINDS

When you set the `defer_bind` member to `1`, the `_bind()` method creates a proxy object (if necessary) and returns an object reference to your client program. A connection will not be established with the object implementation until your client program actually invokes a method on the object. If you set `defer_bind` to `0`, then the connection will be established when `_bind()` is invoked.

If you do not specify the `BindOptions` parameter, the default behavior will be to establish the connection at the time the `_bind()` method is invoked.

ENABLING RE-BINDS

If the connection between your client program 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. This fault tolerant processing is described in Chapter 10, "The Smart Agent Architecture." The re-bind process is enabled when the `enable_rebind` member is set to 1. If you wish to prevent this re-binding process, set `enable_rebind` to 0.

If you do not specify the `BindOptions` parameter, the default `_bind()` behavior is to attempt to re-bind to the server if an error occurs.

SETTING A TIME-OUT FOR SENDING A REQUEST

You set the `send_timeout` member to specify the number of seconds your client program will wait for a request to be delivered to a server object. If the time-out period expires before the message is delivered to the server object, a ::`CORBA::NO_RESPONSE` exception is raised.

If you do not specify the `BindOptions` parameter, `send_timeout` is set to 0, indicating that your client program wishes to block indefinitely.

SETTING A TIME-OUT FOR RECEIVING A RESPONSE

You set `receive_timeout` to specify the number of seconds your client program will wait for a response to be received from a server object. If the time-out period expires before the message is received from the server object, a `CORBA::NO_RESPONSE` exception is raised.

By default, `receive_timeout` is set to 0, which indicates that your client program wishes to block indefinitely.

SETTING A CONNECTION TIME-OUT

You set the `connection_timeout` option to specify the number of seconds your client program will wait for a connection to be established with a server object. If the time-out period expires before a connection is established, a `CORBA::NO_IMPLEMENT` exception is raised.

By default, `connection_timeout` is set to 0 to indicate that your client program wishes to use your platform's default connection time-out. See your platform's documentation for information this value.

Specifying a Scope for the BindOptions

VisiBroker allows you to specify three distinct levels of `BindOptions`. You can specify the options for each invocation of the `_bind()` method, for a particular object reference, or for all object references used by your client program.

SETTING DEFAULT BINDOPTIONS FOR A PROCESS

VisiBroker provides a global `BindOptions` structure that contains default values for the `_bind()` method. These defaults are used if you do not explicitly specify a `BindOptions`

parameter when you invoke the `_bind()` method. Code Sample 4-6 shows the static methods you can use to query and set these default values.

```
class Object {
          static const CORBA::BindOptions *_default_bind_options();
          static void_default_bind_options(const CORBA::BindOptions&);
     ...
};
```

Code Sample 4-6 Static methods for getting and setting the default bind options for a process.

OVERRIDING THE DEFAULT BINDOPTIONS FOR A PROCESS

You can override the default, process-level bind options by passing a new `BindOptions` parameter when you invoke the `_bind()` method. These new options will remain in effect for the life of the object reference returned by `_bind()`, regardless of any changes to the process-level bind options.

SETTING OBJECT-LEVEL BINDOPTIONS

You can change bind options after you have invoked the `_bind()` method. Code Sample 4-7 shows a method you can use on the object reference returned by `_bind()`. This method allows you to change the send and receive time-out values for any valid object reference. If you change the connection time-out and a re-bind occurs, the new connection time-out value will apply. The bind options you set remain in effect for this object reference for as long as the reference is valid.

CODE

```
class Object {
      ...
      void _bind_options(const CORBA::BindOptions& opt);
      ...
};
```

Code Sample 4-7 The method for setting object-level bind options.

ALTERNATIVES TO USING BIND

VisiBroker provides you with an `object_to_string()` method which converts an object reference to a string. The CORBA specification refers to this process as *stringification*. Once an object reference has been stringified, it may be saved to a file, or passed as a command line argument when starting a client program.

VisiBroker provides the `string_to_object()` method for converting a stringified object reference back into an object reference that can be used to invoke methods on the object.

This method allows a client program to obtain and use an object reference without having to invoke the `_bind()` method. For more information on these methods, see "Converting a Reference to a String" on page 4-14.

VisiBroker also provides a Naming Service, packaged separately, that can be used to locate objects by their logically assigned names.

PERFORMING OPERATIONS ON OBJECT REFERENCES

The object reference returned to your client program by the `_bind()` method represents an ORB object. Your client program 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 `CORBA::Object` that you can use to manipulate the object.

Checking for Nil References

You can use the `CORBA` class method `is_nil()` shown below to determine if an object reference is `nil`. This method returns `1` if the object reference passed is `nil`. It returns `0` if the object reference is not `nil`.

CODE
```
class CORBA {
      ...
      static Boolean is_nil(CORBA::Object_ptr obj);
      ...
};
```

Code Sample 4-8 Method for checking for a `nil` object reference.

Obtaining a Nil Reference

You can obtain a `nil` object reference using the `CORBA::Object` class `_nil()` method. It returns a `NULL` value that is cast to an `Object_ptr`.

CODE
```
class Object {
      ...
      static CORBA::Object_ptr _nil();
      ...
};
```

Code Sample 4-9 Method for obtaining a `nil` reference.

Duplicating an Object Reference

When your client program invokes this method, the reference count for the object reference is incremented by one and the same object reference is returned. Your client program can use the `_duplicate()` method to increase the reference count for an object reference so that the reference can be stored in a data structure or passed as a parameter. Increasing the reference count ensures that the memory associated with the object reference will not be freed until the reference count has reached zero.

The IDL compiler generates a `_duplicate()` method for each object interface you specify. The `_duplicate()` method shown accepts and returns a generic `Object_ptr`.

CODE
```
class Object {
      ...
      static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
      ...
};
```

Code Sample 4-10 The method for duplicating an object reference.

NOTE *The `_duplicate()` method has no meaning for the BOA or ORB because these objects do not support reference counting.*

Releasing an Object Reference

You should release an object reference when it is no longer needed. One way of releasing an object reference is by invoking the `CORBA::Object` class `_release()` method.

CAUTION *Always use the `release()` method. Never invoke operator delete on an object reference.*

CODE
```
class CORBA {
      class Object {
          ...
          void      _release();
          ...
      };
};
```

Code Sample 4-11 The method for releasing an object reference.

You may also use the CORBA class `release()` method, which is provided for compatibility with the CORBA specification.

CODE
```
class CORBA {
      ...
      static void release();
      ...
};
```

Code Sample 4-12 The CORBA method for releasing an object reference.

Obtaining the Reference Count

Each object reference has a reference count that you can use to determine how many times the reference has been duplicated. When you first obtain an object reference by invoking `_bind()`, the reference count is set to one. Releasing an object reference will decrement the reference count by one. Once the reference count reaches 0, VisiBroker automatically deletes the object reference. Code Sample 4-13 shows the `_ref_count()` method for retrieving the reference count.

NOTE *When a remote client duplicates or releases an object reference, the server's object reference count is not affected.*

CODE
```
class Object {
      ...
      CORBA::Long _ref_count() const;
      ...
};
```

Code Sample 4-13 Method for obtaining the reference count.

Cloning Object References

The IDL compiler generates a `_clone()` method for each object interface that you specify. Unlike the `_duplicate()` method, `_clone()` will create an exact copy of the object's entire state and establish a new, separate connection to the object implementation. The object reference returned and the original object reference will represent two distinct connections to the object implementation. Code Sample 4-14 shows the `_clone()` method generated for the account interface introduced in Chapter 3, "Quick Start for Development with VisiBroker for C++."

CODE
```
class Account: public virtual CORBA::Object
{
    ...
    static Account_ptr _clone(Account_ptr obj) {
    ...
};
```

Code Sample 4-14 The `_clone()` method for the `Account` class.

Cloning an object reference overrides the ORB's default connection behavior. See "Opening a New Connection for a Thread Using _clone()" on page 7-8 for details.

Converting a Reference to a String

VisiBroker provides an ORB class with methods that allow you to convert an object reference to a string or convert a string back into an object reference. The CORBA specification refers to this process as stringification. A client program can convert an object reference to a string and pass it to another client program. The second client may then de-stringify the object reference and use it without having to explicitly bind to the object. Code Sample 4-15 shows the conversion methods. Note that the caller of `object_to_string()` is responsible for calling `CORBA::string_free()` on the returned string.

NOTE *Locally scoped object references that are stringified are not guaranteed to be valid beyond the life of the process that created the reference. See "Checking for Globally Scoped Object References" on page 5-4 for more information.*

CODE
```
class ORB {
      public:
          // Convert an object reference to a string
          char *object_to_string(CORBA::Object_ptr obj);
          // Convert a char * to an object reference
          CORBA::Object_ptr string_to_object(const char *);
        ...
};
```

Code Sample 4-15 The methods for converting an object reference to a string and vice versa.

Obtaining Object and Interface Names

Code Sample 4-16 shows the methods provided by the `Object` class that you can use to obtain the interface and object names as well as the repository id associated with an object reference. The interface repository is discussed in Chapter 12, "Using Interface Repositories."

NOTE *If you did not specify an object name when you invoked the `_bind()` method, invoking the `_object_name()` method with the resulting object reference will return `NULL`.*

CODE
```
class Object {...
    const char *_interface_name() const;
    const char *_object_name() const;
    const char *_repository_id() const;
...};
```

Code Sample 4-16 Methods for obtaining the interface name, object name, and repository id.

Determining the Type of an Object Reference

You can check whether an object reference is of a particular type by using the `_is_a()` method. You must first obtain the repository id of the type you wish to check using the `_repository_id()` method. This method returns `1` if the object is either an instance of the type represented by `repository_id()` or if it is a sub-type. The method returns `0` if the object is not of the type specified. Note that this may require remote invocation to determine the type.

CODE
```
class Object {
    ...
    CORBA::Boolean _is_a(const char *repository_id);
    ...
};
```

Code Sample 4-17 Method for determining the type of an object reference.

Code Sample 4-18 shows the `_is_equivalent()` method that you can use to check if two object references refer to the same object implementation. This method returns `1` if the object references are equivalent. This method returns 0 if the object references are distinct, but does not necessarily indicate that the object references are to distinct objects. This is a lightweight method and does not involve actual communication with the server object.

CODE
```
class Object {
    ...
    CORBA::Boolean _is_equivalent(CORBA::Object_ptr other_object);
    ...
};
```

Code Sample 4-18 Method for comparing object references.

You can use the `_hash()` method shown in Code Sample 4-19 to obtain a hash value for an object reference. While this value is not guaranteed to be unique, it will remain consistent through the lifetime of the object reference and can be stored in a hash table.

CODE
```
class Object {
    ...
    CORBA::ULong _hash(CORBA::ULong maximum);
    ...
};
```

Code Sample 4-19 The `_hash()` method.

Determining the Location and State of Bound Objects

Given a valid object reference, your client program can use the `_is_bound()` method, shown in Code Sample 4-20 to retrieve the current state of the bind for the object. The method returns `1` if the object is bound and `0` if the object is not bound.

CODE
```
class Object {
    public: ...
        CORBA::Boolean _is_bound() const;
        ...
};
```

Code Sample 4-20 The method for querying the state of the bind for an object reference.

Code Sample 4-21 shows two methods your client program can use after a successful `_bind()` invocation to determine the location of the object implementation. The `_is_local()` method returns 1 if the client program and the object implementation reside within the same process or address space.

The `_is_remote()` method returns 1 if the client program and the object implementation reside in different processes, which may or may not be located on the same host.

CODE
```
class Object {
      ...
      CORBA::Boolean _is_local()  const;
      CORBA::Boolean _is_remote() const;
      ...
};
```

Code Sample 4-21 Methods for determining the location of an object implementation.

NOTE *If the object is in the same process where the method is invoked, `_is_local ()` returns 1.*

Checking for Non-Existent Objects

You can use the `_non_existent()` method, shown in Code Sample 4-22, to determine if the object implementation associated with an object reference still exists. This method actually "pings" the object to determine if it still exists.

CODE
```
class Object {
      ...
      CORBA::Boolean _non_existent();
      ...
};
```

Code Sample 4-22 Method for determining the existence an object implementation.

Obtaining the Current BindOptions

Given a valid object reference, your client program can use the `_bind_options()` method in Code Sample 4-23 to retrieve the bind options currently in effect for that object.

CODE
```
class Object {
      ...
      CORBA::BindOptions* _bind_options();
      ...
};
```

Code Sample 4-23 The method for retrieving an object's bind options.

See "Specifying a Scope for the BindOptions" on page 4-9 for more information.

WIDENING AND NARROWING OBJECT REFERENCES

Converting an object reference's type to a super-type is called *widening*. Code Sample 4-24 shows an example of widening an `Account` pointer to an `Object` pointer. The pointer `acct` can be cast as an `Object` pointer because the `Account` class inherits from the `Object` class.

CODE

```
Account        *acct;
CORBA::Object  *obj;
acct  = Account::_bind();
obj   = (CORBA::Object *)acct;
```

Code Sample 4-24 Widening an object reference.

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*. VisiBroker maintains a typegraph for each object interface so that narrowing can be accomplished by 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 a `nil` reference.

CODE

```
Account  *acct;
Account  *acct2;
Object   *obj;

acct  = Account::_bind();
obj   = (CORBA::Object *)acct;
acct2 = Account::_narrow(obj);
```

Code Sample 4-25 Narrowing an object reference to a sub-type.

NOTE *The `_narrow()` method may construct a new C++ object and returns a pointer to that object. When you no longer need the object, you must release the object reference returned by `_narrow()`.*