

OMNIBROKER

Version 2.0.1

Marc Laukien (ml@ooc.com)
Object-Oriented Concepts, Inc.
44 Manning Road
Billerica, MA 01821, USA

Dr. Uwe Seimet (us@ooc.de)
Object-Oriented Concepts GmbH
Rudolf-Plank-Straße 23
D-76275 Ettlingen, Germany

November 21, 1997

Copyright © 1997 Object-Oriented Concepts, Inc.

Copyright © 1997 Object-Oriented Concepts GmbH

Object-Oriented Concepts is a trademark of Object-Oriented Concepts, Inc.

OmniBroker is a trademark of Object-Oriented Concepts, Inc.

CORBA and Object Request Broker are trademarks of Object Management Group.

OMG is a trademark of Object Management Group.

Java is a trademark of Sun Microsystems, Inc.

Other names, products, and services may be the trademarks or registered trademarks of their respective holders.

Contents

1	Introduction	1
1.1	What is OMNIBROKER?	1
1.2	How is it licensed?	2
1.3	About this document	2
1.4	Getting help	3
2	Getting started	5
2.1	The “Hello World” Application	5
2.2	The IDL code	6
2.3	Implementing the example in C++	6
2.3.1	Implementing the server	7
2.3.2	Implementing the client	9
2.3.3	Compiling and linking	11
2.3.4	Running the application	11
2.4	Implementing the example in Java	11
2.4.1	Implementing the server	11
2.4.2	Implementing the client	14
2.4.3	Compiling	16
2.4.4	Running the application	17
2.5	First conclusions	17
2.6	More examples	18
2.7	Where to go from here	18
3	ORB and BOA initialization	19
3.1	ORB initialization	19
3.1.1	Initializing the C++ ORB	19
3.1.2	Initializing the Java ORB	19
3.1.3	General ORB options	20
3.1.4	C++ specific ORB options	21

3.1.5	Java specific ORB options	21
3.2	BOA initialization	21
3.2.1	Initializing the C++ BOA	21
3.2.2	Initializing the Java BOA	22
3.2.3	General BOA options	22
3.2.4	Java specific BOA options	23
4	The OMNIBROKER C++ library	25
4.1	Extensions to the ORB	25
4.1.1	Finding out which objects are active	25
4.1.2	Setting the Interface Repository	27
4.1.3	Setting the Naming Service	27
4.1.4	Enabling nested method invocations	28
4.1.5	Getting an object by host name, port number and object name	28
4.2	Extensions to the BOA	28
4.2.1	Getting the host name and port number	28
4.3	The Dynamic Invocation Interface	29
4.3.1	Definition of <code>Status</code>	29
4.3.2	Run-time checks	29
4.4	The Interface Repository	29
4.5	The Naming Service	30
4.6	Creating objects	30
4.6.1	Creating implementation objects on the server side	30
4.6.2	Creating proxy objects on the client side	31
4.7	Error messages and exceptions	34
4.7.1	CORBA exceptions used by OMNIBROKER	34
4.7.2	Non-compliant application asserts	38
5	The OMNIBROKER Java classes	43
5.1	Extensions to the ORB	43
5.1.1	Setting the Interface Repository	43
5.1.2	Setting the Naming Service	44
5.1.3	Creating named implementation objects	44
5.2	Extensions to the BOA	45
5.2.1	Getting the host name and port number	45
5.2.2	Setting the thread model	45
6	The OMNIBROKER code generators	47
6.1	Synopsis	47
6.2	Description	48

6.3	Options for <code>idl</code>	49
6.4	Options for <code>jidl</code>	50
6.5	Options for <code>hidl</code>	51
6.6	Options for <code>irserv</code>	52
6.7	Options for <code>irfeed</code>	52
6.8	Options for <code>irdel</code>	53
6.9	Options for <code>irgen</code>	53
6.10	The IDL-to-C++ translator and the Interface Repository	53
6.11	<code>#include</code> statements	54
6.12	Documenting IDL files	55
6.13	Using <code>javadoc</code>	55
7	The IDL-to-C++ mapping	59
7.1	Reserved names	59
7.2	Mapping of modules	59
7.3	Implementing interfaces	60
7.3.1	Inheritance-based implementation	60
7.3.2	Delegation-based implementation	62
7.4	Extensions	63
7.4.1	Extensions to the string type	63
7.4.2	Extensions to the <code>_var</code> types	63
7.4.3	Extensions to the sequence types	64
8	C++ Tips & Tricks	67
8.1	Strings	67
8.1.1	CORBA-specific string functions	67
8.1.2	Initialization and assignment from <code>char*</code> and <code>const char*</code>	68
8.1.3	Initialization and assignment from <code>CORBA_String_var</code>	69
8.1.4	Strings as parameters and return values	69
8.2	Object references	71
8.2.1	Object reference memory management	72
8.2.2	Object references as parameters and return values	74
8.2.3	Implementing a “destroy” function	76
8.2.4	Getting an implementation object from a reference	78
8.2.5	Cyclic object dependencies	80
8.2.6	String <code>_var</code> types and object reference <code>_var</code> types differences	84
8.2.7	Global <code>_var</code> type object references	85

9	OMNIBROKER communication	87
9.1	Method invocation semantics	87
9.1.1	Invocation by an interface stub	87
9.1.2	Invocation by the DII	88
9.1.3	Using timeouts	89
9.1.4	Nested method invocations	90
9.2	The Reactor	90
9.2.1	Available reactors	91
9.2.2	Writing a custom event handler	92
A	Exceptions	95
A.1	Standard exceptions	95
A.2	Minor exception codes	96
A.2.1	Minor exception codes for CORBA_COMM_FAILURE	96
A.2.2	Minor exception codes for CORBA_INTF_REPOS	96
B	OMNIBROKER License	99

Chapter 1

Introduction

1.1 What is OMNIBROKER?

OMNIBROKER is an Object Request Broker (ORB) that is compliant to the Common Object Request Broker Architecture (CORBA) specification, revision 2.0, as defined in [1] and [2] by the Object Management Group (OMG).

Some highlights of OMNIBROKER are:

- Full CORBA IDL support
- Complete CORBA IDL-to-C++ mapping
- Complete CORBA IDL-to-Java mapping
- Uses IIOP as native protocol
- Dynamic Invocation Interface
- Dynamic Skeleton Interface
- Interface Repository
- Peer-to-Peer communication with nested method invocations
- Support for non-blocking method invocations
- Support for timeouts
- Seamless integration with X11 and Windows
- A COS compliant Naming Service

- IDL-to-HTML translator for generating “javadoc”-like documentation
- DynAny API for dynamic Any type handling

The current beta version has the following limitations:

- Only persistent (i.e. manually launched) servers are currently supported
- No multi-threaded C++ applications (OmniBroker for Java supports tread-per-request and thread-per-client)

1.2 How is it licensed?

OMNIBROKER is licensed as “free for non-commercial use”. See the license agreement in chapter B on page 99 for details. For commercial licenses, please send an e-mail to m1@ooc.com.

1.3 About this document

This manual is — apart from the “Getting started” chapter — *not* a replacement for a good CORBA book. There are many excellent introductory books on CORBA fundamentals, for example [4] or [5].

This manual does also *not* contain the precise specifications of the CORBA standard. This would definitely be out of its scope. However, for the understanding of this manual, a good knowledge of the CORBA specification in [1] is *absolutely necessary*. Especially the chapters covering CORBA IDL and the IDL-to-C++ mapping should be studied thoroughly. Do not expect any of the CORBA teaching books to be a reference for the IDL-to-C++ mapping. The books currently available only give an overview and are neither complete nor up-to-date. *There is no substitute for the official CORBA specification as defined in [1]*.

What this manual does contain, however, is information on *how* OMNIBROKER implements the CORBA standard. The problem with the current CORBA specification is that it leaves a high degree of freedom to the CORBA implementation. For example, the precise semantics of a oneway call are not specified by the standard (see 9.1.1 on page 88).

To make it easier to get started with OMNIBROKER, this manual contains a “Getting started” chapter, explaining some OMNIBROKER basics with a very simple example.

1.4 Getting help

Should you have any problems with OMNIBROKER, do not hesitate to ask us for assistance. Just send an e-mail to support@ooc.com.

You will always find the latest information on OMNIBROKER on OOC's Web Pages <http://www.ooc.com/ob.html> and <http://www.ooc.de/ob.html>. Additionally there is an OMNIBROKER mailing list. To subscribe, just send an e-mail to majordomo@ooc.com with the command

```
subscribe ob
```

in the body of your message. To post a message to the mailing list, send it as an e-mail to ob@ooc.com. There is an archive for this mailing list at:

<http://www.ooc.com/ob-mailing-list/maillist.html>

Chapter 2

Getting started

2.1 The “Hello World” Application

This Quick-Start Guide’s example software is founded on a well-known application: A “Hello World!” program presented here in a special client–server version.

Many books on programming start with this tiny demo program. In introductory C++ books you’ll probably find the following piece of code in the very first chapter:

```
// C++

#include <iostream.h>

int
main(int, char*[], char*[])
{
    cout << "Hello World!" << endl;
    return 0;
}
```

Or in introductory Java books:

```
// Java

public class Greeter
{
    public static void main(String args[])
    {
        System.out.println("Hello, World!");
    }
}
```

```
    }  
}
```

These applications simply print “Hello World!” to standard output and that is exactly what this chapter is about: Printing “Hello World!” with a CORBA based client–server application. Basically this means that a client program invokes a `hello()` operation on an object in a server application. The server responds by printing “Hello World!” on standard output.

2.2 The IDL code

How do we write a CORBA based “Hello World!” application? The first thing to do is to create a file with IDL code, that has to be translated (“mapped”) to a particular programming language. In the case of OMNIBROKER this language is either C++ or Java.

Since our sample application isn’t a complicated one, the IDL code needed for this example is simple:

```
// IDL  
  
interface Hello  
{  
    void hello();  
};
```

This piece of IDL code defines an interface `Hello` consisting of a single operation `hello`. `hello` neither expects nor returns any parameters.

2.3 Implementing the example in C++

The first step is to translate the IDL code with the IDL–to–C++ translator. Save the IDL code shown above to a file called `Hello.idl`. Now translate the code to C++ using the following command:

```
idl Hello.idl
```

This command will create the files `Hello.h`, `Hello.cpp`, `Hello_skel.h` and `Hello_skel.cpp`. The only file that is of importance for you as an OMNIBROKER user is `Hello_skel.h`. This file has to be included when writing the implementation classes.

2.3.1 Implementing the server

For the server, we need to define an implementation class for the `Hello` interface defined in the previous paragraph. To do this, we create a class `Hello_impl` that is derived from the so-called skeleton class `Hello_skel`, defined in the file `Hello_skel.h`. The definition for `Hello_impl` looks like this:

```
// C++

#include <Hello_skel.h>

class Hello_impl : public Hello_skel
{
public:

    Hello_impl();

    virtual void hello();
};
```

The implementation for `Hello_impl` is:

```
// C++

#include <CORBA.h>
#include <Hello_impl.h>

Hello_impl::Hello_impl()
{
}

void
Hello_impl::hello()
{
    cout << "Hello World!" << endl;
}
```

As you can see, the constructor doesn't do anything¹ and the `hello()` function simply prints "Hello World!" on standard output.

¹Although the default constructor is empty, it must be defined since `Hello_skel` hides the default constructor by making it protected.

Save the class definition of `Hello_impl` in a file `Hello_impl.h` and the implementation of `Hello_impl` in a file `Hello_impl.cpp`.

Now we need to write the server's `main()` program, which looks like this:

```
// C++

#include <CORBA.h>
#include <Hello_impl.h>

#include <fstream.h>

int
main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    Hello_var p = new Hello_impl;

    CORBA_String_var s = orb -> object_to_string(p);
    const char* refFile = "Hello.ref";
    ofstream out(refFile);
    out << s << endl;
    out.close();

    boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}
```

Save this to a file `Server.cpp`.

The contents of the `main()` function need to be explained. The first thing a CORBA program has to do is to initialize the ORB² and the BOA³. This is done by the following source fragment:

```
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
```

`CORBA_ORB_init()` and `BOA_init()` both expect the parameters which the program was started with. These parameters may or may not be used by the ORB

²Object Request Broker

³Basic Object Adapter.

and BOA, depending on the CORBA implementation. OMNIBROKER recognizes certain options that will be explained later. Now an instance of `Hello_impl` can be created like this:

```
Hello_var p = new Hello_impl;
```

`Hello_var`, like all `_var`-types, is a “smart” pointer, i.e. `p` will release the object created by `new Hello_impl` automatically when `p` goes out of scope.

The client must be able to access the implementation object. This can be done by saving a “stringified” object reference to a file which can be read by the client and converted back to the actual object reference.⁴ The operation `object_to_string()` converts a CORBA object reference into its string representation:

```
CORBA_String_var s = orb -> object_to_string(p);
const char* refFile = "Hello.ref";
ofstream out(refFile);
out << s << endl;
out.close();
```

Finally, in order to react on incoming requests, the server must enter its event loop. This is done by:

```
boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
```

Since OMNIBROKER does not use the `CORBA_ImplementationDef` argument, `CORBA_ImplementationDef::_nil()` can be used as a dummy argument.

2.3.2 Implementing the client

Writing the client requires less work than writing the server, since the client, in this example, only consists of the `main()` function. In several respects the client’s `main()` is similar to the server’s `main()` function:

```
// C++
```

```
#include <CORBA.h>
```

⁴If your application contains more than one object, you do not need to save object references for all objects. Usually you save the reference of one object which provides calls that can subsequently return references to other objects.

```

#include <Hello.h>

#include <fstream.h>

int
main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    const char* refFile = "Hello.ref";
    ifstream in(refFile);
    char s[1000];
    in >> s;
    CORBA_Object_var obj = orb -> string_to_object(s);
    assert(!is_nil(obj));

    Hello_var hello = Hello::_narrow(obj);
    assert(!is_nil(hello));

    hello -> hello();
}

```

Save this into a file `Client.cpp`.

Like the server's implementation of `main()` the client's `main()` starts with the initialization of the ORB. In the next step the "stringified" object reference written by the server is read and converted to a `CORBA::Object` object reference. The `_narrow()` operation then generates a `Hello` object reference from the `CORBA::Object` object reference.⁵

The `assert()` macros ensure that the program does not continue unless the return values of `string_to_object()` and `_narrow()` are valid object references.

Finally, the `hello()` operation on the `hello` object reference is invoked, causing the server to print "Hello World!".

⁵Although CORBA's `T::_narrow` for an interface `T` works similar to `dynamic_cast<T>()` for a plain C++ class `T`, `dynamic_cast<T>()` must not be used for CORBA object references.

2.3.3 Compiling and linking

Both the client and the server must be linked with the compiled `Hello.cpp` (which usually has the name `Hello.o` under Unix and `Hello.obj` under Windows). The compiled `Hello_skel.cpp` (and `Hello_impl.cpp` of course) is only needed for the server.

Compiling and linking is highly compiler dependent. Many compilers require unique options to generate correct code. To link OMNIBROKER programs, you must at least link with the OMNIBROKER library `libOB.a` (Unix) or `ob.lib` (Windows). On some systems additional libraries are needed, for example `libsocket.a` and `libnsl.a` for Solaris or `wsock32.lib` for Windows.

The OMNIBROKER distribution comes with various README files for different platforms which give hints on the options needed for compiling and the libraries needed for linking. Please consult these README files for details.

2.3.4 Running the application

Our “Hello World!” application consists of two parts, namely the client and the server program. The first program to be started is the server because it creates the file `Hello.ref` that is needed by the client in order to connect to the server. As soon as the server is running, you can start the client. If all goes to plan, the “Hello World!” message will appear on the screen.

2.4 Implementing the example in Java

In order to code this demo application in Java, the interface specified in IDL is translated to Java classes similar to the way the C++ code was created. The OMNIBROKER IDL-to-Java translator `jidl` is called like this:

```
jidl --package hello Hello.idl
```

This command results in several Java source files on which the actual implementation will be based. The generated files are `Hello.java`, `HelloHelper.java`, `HelloHolder.java`, `StubForHello.java` and `_HelloImplBase.java`, all generated in a directory with the name `hello`.

2.4.1 Implementing the server

The server’s `Hello` implementation class has to be based on `_HelloImplBase`:

```
// Java

package hello;

import org.omg.CORBA.*;

public class Hello_impl extends _HelloImplBase
{
    public void hello()
    {
        System.out.println("Hello World!");
    }
}
```

As with the C++ implementation, the `hello()` function simply prints “Hello World!” on standard output. Save this class to a file `Hello_impl.java`.

We also have to write a class which holds the server’s `main()` function. We call this class `Server`:

```
// Java

package hello;

import org.omg.CORBA.*;
import java.io.*;

public class Server
{
    public static void main(String args[])
    {
        //
        // Create ORB and BOA
        //
        ORB orb = ORB.init(args, new java.util.Properties());
        BOA boa = orb.BOA_init(args, new java.util.Properties());

        //
        // Create implementation object
        //
        Hello_impl p = new Hello_impl();
    }
}
```

```

        //
        // Save reference
        //
        try
        {
            String ref = orb.object_to_string(p);
            String refFile = "Hello.ref";
            PrintWriter out =
                new PrintWriter(new FileOutputStream(refFile));
            out.println(ref);
            out.flush();
        }
        catch(IOException ex)
        {
            System.err.println("Can't write to `" +
                               ex.getMessage() + "`");
            System.exit(1);
        }

        //
        // Run implementation
        //
        boa.impl_is_ready(null);

        System.exit(0);
    }
}

```

Save this to a file `Server.java`.

As you can see, in Java the ORB and the BOA are initialized with the following commands:

```

ORB orb = ORB.init(args, new java.util.Properties());
BOA boa = orb.BOA_init(args, new java.util.Properties());

```

The instance of `Hello_impl` is created with:

```

Hello_impl p = new Hello_impl();

```

If you code an application in Java, there are no `_var`-types that automatically release their contents (like in C++). This is not necessary in Java, since Java

provides for automatic garbage collection. That is, `p` is released automatically when it is not used anymore.

The object reference is then “stringified” and written to a file, just like in the C++ implementation:

```
try
{
    String ref = orb.object_to_string(p);
    String refFile = "Hello.ref";
    PrintWriter out =
        new PrintWriter(new FileOutputStream(refFile));
    out.println(ref);
    out.flush();
}
catch(IOException ex)
{
    System.err.println("Can't write to `" +
        ex.getMessage() + "`");
    System.exit(1);
}
```

Finally, the server enters its event loop to receive incoming requests:

```
boa.impl_is_ready(null);
```

2.4.2 Implementing the client

Save this to a file `Client.java`:

```
// Java

package hello;

import org.omg.CORBA.*;
import java.io.*;

public class Client
{
    public static void main(String args[])
    {
        //
```

```
// Create ORB
//
ORB orb = ORB.init(args, new java.util.Properties());

//
// Get "hello" object
//
String ref = null;
try
{
    String refFile = "Hello.ref";
    BufferedReader in =
        new BufferedReader(new FileReader(refFile));
    ref = in.readLine();
}
catch(IOException ex)
{
    System.err.println("Can't read from '" +
        ex.getMessage() + "'");
    System.exit(1);
}

org.omg.CORBA.Object obj = orb.string_to_object(ref);
if(obj == null)
    throw new RuntimeException();

Hello p = HelloHelper.narrow(obj);

//
// Main loop
//
System.out.println("Enter 'h' for hello or 'x' for exit:");

int c;

try
{
    do
    {
        System.in.skip(System.in.available());
```

```

        System.out.print("> ");
        c = System.in.read();
        if(c == 'h')
            p.hello();
    }
    while(c != 'x');
}
catch(IOException ex)
{
    System.out.println("Can't read from standard input");
    System.exit(1);
}

System.exit(0);
}
}

```

The client's implementation is straightforward: The ORB is initialized, the "stringified" reference is read and converted to an object which is "narrowed" to a reference to a `Hello` object. The client then enters its main loop. Here the function `hello()` is called whenever the user enters a `h`.

2.4.3 Compiling

To compile the implementation classes and the classes generated by the OMNIBROKER IDL-to-Java translator `javac` is called:

```
javac *.java hello/*.java
```

Ensure that your `CLASSPATH` environment variable includes the OMNIBROKER Java library, i.e. the `OMNIBROKER_jlib` directory. If you are using a Unix bourne shell or compatible, you can do this with the following commands:

```
CLASSPATH=your_omnibroker_directory/jlib:$CLASSPATH
export CLASSPATH
```

Of course you must replace `your_omnibroker_directory` by the name of the directory where you have put your OMNIBROKER distribution.

If you are running OMNIBROKER on a Windows-based system, you can use the following commands within the Windows command interpreter:

```
set CLASSPATH=your_omnibroker_directory\jlib;%CLASSPATH%
```

Note that for Windows you must use ";" and not ":" as delimiter.

2.4.4 Running the application

The “Hello World” Java server is started with:

```
java hello.Server
```

And the client with:

```
java hello.Client
```

Again, make sure that your CLASSPATH environment variable includes the `jlib` directory.

You might also want to use a C++ server together with a Java client (or vice versa). This is the nice thing about CORBA: If something is defined in CORBA IDL, you don’t have to care about the programming language used for the implementation. CORBA applications can talk with each other, regardless of the language they are written in.

2.5 First conclusions

At this point you might be inclined to think that this method of getting a simple string displayed is the most complicated way you have ever encountered in your life as a programmer. At first glance a CORBA based approach may indeed seem complicated. On the other hand, think of the benefits this kind of approach has to offer. You can start the server and client application on different machines with exactly the same results. Concerning the communication between the client and the server you don’t have to worry about platform specific methods or protocols at all, provided there is a CORBA ORB available for the platform and a programming language of your choice. If possible, get some hands on experience and start the server on one machine, the client on another⁶. As you see, CORBA based applications run interchangeably in both local and network environments.

One last point to note, naturally enough you won’t use CORBA to write such a simple program as in our example. The more complex your applications get (and today’s applications *are* complex) the more you will learn to appreciate having a high-level abstraction of your applications’ key interfaces captured in CORBA IDL.

⁶Note that after the startup of the server program, you have to copy the stringified object reference, i.e. the file `Hello.ref`, to the machine where the client program is to be run.

2.6 More examples

In [4] you can find very useful information about CORBA as well as additional CORBA examples. The sources of this book can be compiled with OMNIBROKER after applying some patches. Please see the file `README.CORBA-BOOK` (which comes with the OMNIBROKER distribution) for details.

2.7 Where to go from here

To understand the remaining chapters of this manual, you *must* have read the CORBA specification in [1] and [2]. You will not be able to understand the following parts without a good knowledge of CORBA IDL, the IDL-to-C++ and IDL-to-Java mapping. CORBA in general and the IDL-to-C++ mapping are described in [1], the IDL-to-Java mapping is described in [2].

Chapter 3

ORB and BOA initialization

3.1 ORB initialization

3.1.1 Initializing the C++ ORB

In C++ the ORB is initialized with `CORBA_ORB_init()`. For example:

```
// C++

int
main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);

    // ...
}
```

The `CORBA_ORB_init()` call interprets arguments starting with `-ORB`. All these arguments, passed through the `argc` and `argv` parameters, are automatically removed from the argument list.

3.1.2 Initializing the Java ORB

In Java the ORB can be initialized this way:

```
// Java

import org.omg.CORBA.*;
```

```

public static void
main(String args[])
{
    ORB orb = ORB.init(args, new java.util.Properties());

    // ...
}

```

The `ORB.init()` call interprets arguments starting with `-ORB`.

3.1.3 General ORB options

In OMNIBROKER, the following options can be used with the C++ and Java ORB:

`-ORBid` **id**

This option may be used to select an ORB with a specific **id**. Currently only the **ids** `OB_IIOP_ORB` (for C++) and `OB_IIOP_MT_ORB` (for Java) are valid.

`-ORBrepository` **repository**`-IOR`

With this option the Interface Repository used by the ORB can be set. The argument must be a stringified IOR, created with `object_to_string()`.

`-ORBnaming` **naming**`-service``-IOR`

With this option the Naming Service used by the ORB can be set. The argument must be a stringified IOR, created with `object_to_string()`.

`-ORBconfig`

This option specifies the name of an OMNIBROKER configuration file. Such a file may consist of several sections configuring certain OMNIBROKER features. So far only one section called `[services]` is supported. The `[services]` section tells the ORB about IORs of initial services that are to be included in the ORB's initial services list. The ORB call `list_initial_services()` will return the OMNIBROKER standard initial services `NameService` and `InterfaceRepository` as well as the names of the services added via the ORB configuration file. An OMNIBROKER configuration file looks like this:

```

#
# Sample OmniBroker configuration file

```

```
#  
  
#Start of section with IORs of initial services  
[services]  
  
#Trading Service  
TradingService iiop://www.xxx.com:5000/Trader  
  
#Event Service  
EventService iiop://www.xxx.com:5001/Event
```

Lines starting with # are treated as comments. Note that IORs starting with `iiop://` are an OMNIBROKER specific extension.

`-ORBversion`

Shows the OMNIBROKER version.

`-ORBlicense`

Shows OMNIBROKER license type and number.

3.1.4 C++ specific ORB options

The C++ ORB supports an additional option:

`-ORBnested`

With this option the ORB supports nested method invocations, implying that incoming requests are dispatched during the execution of outgoing requests. (See 9.1.4 on page 90.)

3.1.5 Java specific ORB options

For the Java ORB the `-ORBconfig` option accepts a URL specification as filename.

3.2 BOA initialization

3.2.1 Initializing the C++ BOA

In C++ the BOA is initialized with `CORBA_ORB::BOA_init()`. For example:

```
// C++

int
main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    // ...
}
```

BOA_init() removes all arguments starting with -OA passed through the argc and argv parameters.

3.2.2 Initializing the Java BOA

The Java the BOA initialization looks like this:

```
// Java

import org.omg.CORBA.*;

public static void
main(String args[])
{
    ORB orb = ORB.init(args, new java.util.Properties());
    BOA boa = orb.BOA_init(args, new java.util.Properties());

    // ...
}
```

3.2.3 General BOA options

OMNIBROKER defines the following options for both the C++ and the Java BOA:

-OAid id

This option may be used to select a BOA with a specific **id**. Currently only the **ids** OB_IIOP_BOA (for C++) and OB_IIOP_MT_BOA (for Java) are valid.

-OAnumeric

This options instructs the BOA to generate object references that contain the internet address in dotted decimal notation instead of the canonical host name.

`-OAhost` **host**

This option can be used to set the host name explicitly. This is especially useful if a host has more than one name. The default is to use the canonical host name. Note that `-OAhost` is ignored if the option `-OAnumeric` is used.

`-OApport` **port**

This option can be used to choose the port number. The default is to let the BOA choose the port number.

3.2.4 Java specific BOA options

The JAVA BOA supports additional options:

`-OAsingle_threaded`

Use this option if you want the BOA to use one single thread for all requests from all clients.

`-OAtthread_per_client`

With this option the BOA uses a single thread for each client which handles all requests from that client.

`-OAtthread_per_request`

This option instructs the BOA to create a new thread for each request, independent from which client the request was initiated.

Chapter 4

The OMNIBROKER C++ library

4.1 Extensions to the ORB

The OMNIBROKER ORB has some very useful extensions. However, don't use these extensions if you want your program to be portable to other CORBA implementations.

4.1.1 Finding out which objects are active

The OMNIBROKER ORB provides a function `print_active_objects()`, that prints out a listing of all active objects whose class is derived from `CORBA_Object`. This option is especially useful to check if all objects have been released.

The following idiom which uses `print_active_objects()` is used by many of the OMNIBROKER demo applications. The idea is to have a `main()` function, that does nothing but creating the ORB and passing it together with command line arguments to a pseudo-`main()` function (here the name `run()` is used). After its return it checks whether all objects have been released using the `print_active_objects()` function (which, of course, if everything goes well, must not print anything). The separate `run()` function is used because this way it is possible to use `_var` types which provide for automatic release of the objects they hold after the return of `run()`. Here an example:

```
int
run(CORBA_ORB_ptr orb, int argc, char* argv[])
{
    //
    // Create BOA
```

```
//
CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

//
// Create other objects
//
CORBA_Object_var obj = ...

//
// Run implementation
//
boa -> impl_is_ready(CORBA_ImplementationDef::_nil());

//
// Bye-bye
//
return 0;
}

int
main(int argc, char* argv[], char*[])
{
    int status;

    try
    {
        //
        // Create ORB
        //
        CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);

        //
        // Run
        //
        status = run(orb, argc, argv);

        //
        // Nothing should be printed here, since all objects should
        // have been released after the return of run()
        //
    }
}
```

```

        orb -> print_active_objects(cout);
    }
    catch(CORBA_SystemException& ex)
    {
        OBPrintException(ex);
        status = 1;
    }

    return status;
}

```

4.1.2 Setting the Interface Repository

To set the Interface Repository used by OMNIBROKER the function `repository()` is available. The argument to `repository()` must be an object reference for an Interface Repository object. See also 4.4 on page 29.

To query the Interface Repository, `resolve_initial_references()` can be used. If the Interface Repository is not available, the exception `CORBA_ORB::InvalidName` is thrown. For example:

```

CORBA_ORB_var orb = ...

CORBA_Object_var obj;
obj = orb -> resolve_initial_references("InterfaceRepository");

CORBA_Repository_var repository;
repository = CORBA_Repository::_narrow(obj);

```

Note that `resolve_initial_references()` is defined by the CORBA-2 standard and therefore is *not* an OMNIBROKER-specific extension.

4.1.3 Setting the Naming Service

Analogous to the Interface Repository, the Naming Service used by OMNIBROKER can be set with function `naming()`. The argument to `naming()` must be an object reference for a COS Naming Context object. For more information on the Naming Service, see 4.5 on page 30.

`resolve_initial_references()` can also be used to query the Naming Service. For example:

```

#include <CosNaming.h>

```

```

CORBA_ORB_var orb = ...

CORBA_Object_var obj;
obj = orb -> resolve_initial_references("NameService");

CosNaming_NamingContext_var nc;
nc = CosNaming_NamingContext::_narrow(obj);

```

4.1.4 Enabling nested method invocations

To enable nested method invocations or to query whether this feature is currently enabled, the functions `nested(bool)` or `bool nested()` can be used. See 9.1.4 on page 90 for more information on nested method invocation.

4.1.5 Getting an object by host name, port number and object name

With the function `get_inet_object()` objects can be created by supplying the object's name, its server host name and port number. On how to create named objects, see 4.6 on page 30.

4.2 Extensions to the BOA

The OMNIBROKER BOA provides some additional functions that can be useful to some applications. Please note that if you use these extensions, your program is not portable.

4.2.1 Getting the host name and port number

If you don't use the `-OApport` option for BOA initialization, OMNIBROKER chooses a port number for you. To find out which one, you can use the `port()` function. Similarly, you can inquire the host name used by OMNIBROKER with the `host()` function. For example:

```

int
main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

```

```
    cout << "Running on host `" << boa -> host()
        << "', port number " << boa -> port() << endl;
}
```

The return value of `host()` is of type `const char*` and must not be released by `CORBA_string_free()`.

4.3 The Dynamic Invocation Interface

4.3.1 Definition of Status

The CORBA specification [1] allows two alternatives for the definition of the type `Status`, used as return status by the DII. It may be defined as either `unsigned long` or `void`. OMNIBROKER defines `Status` as `void`.

4.3.2 Run-time checks

In the current version of OMNIBROKER, parameters supplied to a DII request are not run-time type checked. This is true for all `in`, `inout` and `out` arguments as well as for the return value. *Using wrong arguments will most likely make your application crash.* For later versions of OMNIBROKER, run-time type checking of arguments using the Interface Repository is planned. However, this check will always be optional, since run-time type checking can cause an unacceptable overhead.

4.4 The Interface Repository

OMNIBROKER supports a full featured Interface Repository. However, since the current OMNIBROKER version has no support for automatic object or server activation, the Interface Repository server must be invoked manually and its IOR must be passed to the OMNIBROKER ORB by either using the option `-ORBrepository` or the function `CORBA_ORB::repository()`.¹ For example:

```
irserv --ior file1.idl file2.idl file3.idl > /tmp/IntRep.ref
myapplication -ORBrepository `cat /tmp/IntRep.ref`
```

Please note that it is only necessary to set the Interface Repository for the server side. Clients will query all information they need from the server. If your application does not make use of the Interface Repository, it is of course not necessary to use the `-ORBrepository` option.

¹This approach is cumbersome. Future versions of OMNIBROKER will provide for automatic start-up of the Interface Repository server.

4.5 The Naming Service

OMNIBROKER comes with a COS compliant Naming Service. Like the Interface Repository server, the Naming Service server must be invoked manually and its IOR must be passed to the OMNIBROKER ORB by either using the option `-ORBnaming` or the function `CORBA_ORB::naming()`. For example:

```
nsserv --ior > /tmp/Naming.ref
myapplication -ORBnaming `cat /tmp/Naming.ref`
```

4.6 Creating objects

4.6.1 Creating implementation objects on the server side

On the server side, implementation objects (i.e. objects implementing an interface skeleton) are simply created with `new`. Never allocate implementation objects on the stack. See 8.2.1 on page 72 for a discussion why.

Creating regular implementation objects

To create a regular implementation object, you first need to write an implementation class that is derived from the specific interface skeleton class. The skeleton class provides a default constructor, i.e. a constructor that does not take any arguments. However, this constructor is protected, so the derived implementation class must at least define an empty public constructor. For example:

```
// Implementation class for interface I
class I_impl : virtual public I_skel
{
public:

    void I_impl() { }

    // Other members ...
};
```

See 7.3 on page 60 for more information on how to write implementation classes.

Such implementation classes can easily be instantiated with the `new` operator to create implementation objects:

```
I_var impl = new I; // An I implementation
I_var anotherImpl = new I; // Another I implementation
```

Creating named implementation objects

Note: Named implementation objects are an OMNIBROKER-specific extension.

If you want to use `get_inet_object()` (see below) on the client side, you must create implementation objects with a name, since this name is needed as an argument for `get_inet_object()` to resolve a particular implementation object within a specific server.

For named implementation objects, the skeleton class provides a constructor that takes a string argument as the name. For example:

```
// Implementation class for interface I
class I_impl : virtual public I_skel
{
public:

    void I_impl(const char* n) : I_impl(n) { }

    // Other members ...
};
```

As with the default constructor of the skeleton class, the constructor with the string argument is also `protected`, i.e. it is always necessary to explicitly define a `public` constructor in the implementation class, even if this constructor just handles an argument through to the `protected` constructor of the skeleton class (like in the example above).

With such implementation classes, named implementation objects can now be instantiated with `new` as follows:

```
I_var impl = new I("aName"); // An I implementation
I_var anotherImpl = new I("anotherName"); // Another I implementation
```

A `CORBA_INV_IDENT` exception will be thrown if the name used is not unique.

4.6.2 Creating proxy objects on the client side

Usually proxy objects (i.e. objects for interface stubs) are created by returning an object reference to that proxy object from some other proxy object. For example:

```
// IDL

interface A;
interface B;
interface C;

interface I
{
    A getA();
    B getB();
    C getC();
};
```

Here the `getA()`, `getB()` and `getC()` operations of `I` return object references to an `A`, `B` and `C` respectively. On the server side, this can be implemented as follows:

```
// C++

class I_impl : virtual public I_skel
{
    A_var a_;
    B_var b_;
    C_var c_;

public:

    void I_impl()
    {
        a_ = new A_impl;
        b_ = new B_impl;
        c_ = new C_impl;
    }

    virtual A_ptr getA() { return A::_duplicate(a_); }
    virtual B_ptr getB() { return B::_duplicate(b_); }
    virtual C_ptr getC() { return C::_duplicate(c_); }
};
```

On the client side, the code could look like this:

```
// C++

I_var i = ... // Get an I object reference somehow
A_var a = i -> getA(); // Get an A object reference
B_var b = i -> getB(); // Get an B object reference
C_var c = i -> getC(); // Get an C object reference
```

But how do you get an object reference for the very first proxy object? In our example this is an object reference to the I proxy object. We have a chicken-and-egg problem here.

To solve this problem, two other methods are available on the client side for creating proxy objects. The first one is to use a “stringified” object reference and the ORB function `string_to_object()`. The second method uses a host name, a port number and an object name in combination with the function `get_inet_object()`.

Using `string_to_object()`

To create a proxy object with the ORB function `string_to_object()` you must somehow get a stringified object reference. For example, the server can create such a reference with the reverse ORB function `object_to_string()` and then write this stringified object reference to a file. Subsequently the client can read this reference from that file and use it as the argument to `string_to_object()`. This method is shown in the following example. First the relevant server code:

```
// Create an implementation object for an interface I
I_var impl = new I;

// Get "stringified" object reference
CORBA_String_var s = orb -> object_to_string(impl);

// Save reference to a file with the name "ref"
const char* file = "ref";
ofstream out(file)
out << s << endl;
out.close();
```

And the corresponding client code:

```
const char* file = "ref";
ifstream in(file);
```

```
char s[1000];
in >> s;
CORBA_Object_var obj = orb -> string_to_object(s);
I_var proxy = I::_narrow(obj);
```

Note that you can use `string_to_object()` or `object_to_string()` on any object. It doesn't matter if this object is a named object or a regular object.

Using `get_inet_object()`

Note: `get_inet_object()` is an OMNIBROKER-specific extension.

Sometimes it is convenient to directly specify an object by the name of the host where the server resides, the server's port number and the object's name which must be unique within this server. This can be done with the ORB function `get_inet_object()` on the client side, provided that the implementation object on the server side has been created as a named object (see above). For example:

```
char host[1024];
CORBA_ULong port;
cout << "Host name ? " << flush;
cin >> host;
cout << "Port number ? " << flush;
cin >> port;
CORBA_Object_var obj = orb -> get_inet_object(host, port, "aImpl");
I_impl proxy = I::_narrow(obj);
```

To gather information about the host name and the port number within a server, use the BOA functions `host()` and `port()` (see 4.2.1 on page 28). To explicitly set the host name and port number to a specific value, the BOA initialization options `-OAhost` and `-OApport` can be used (see 3.2 on page 21).

4.7 Error messages and exceptions

To write robust programs, a good exception and error handling mechanism is a must. OMNIBROKER provides for two kinds of error notification: CORBA exceptions and "asserts" for non-compliant applications. While the first are mainly used for detecting run-time errors, the latter are used for detecting programming mistakes.

4.7.1 CORBA exceptions used by OMNIBROKER

The CORBA specification defines standard system exceptions but is silent about exactly which exceptions are raised under specific conditions.

The following exceptions are raised by OMNIBROKER:

CORBA_UNKNOWN

This exception is raised if an object that implements an interface raises an exception which was not declared with `raises` in the IDL code for that interface.

CORBA_BAD_PARAM

This exception is currently not used by OMNIBROKER.

CORBA_NO_MEMORY

This exception is currently not used by OMNIBROKER.

CORBA_IMP_LIMIT

This exception is currently not used by OMNIBROKER.

CORBA_COMM_FAILURE

This exception describes general communication problems. It is used extensively by OMNIBROKER. For a list of possible minor codes that indicate the cause of the exception, see A.2 on page 96. To provide for robust behavior, applications should be prepared to catch and handle this kind of exception in any circumstance.

CORBA_INV_OBJREF

This exception is raised if the OMNIBROKER ORB cannot interpret an object reference, for example because the argument to `CORBA_ORB::string_to_object()` is corrupted.

CORBA_NO_PERMISSION

This exception is currently not used by OMNIBROKER.

CORBA_INTERNAL

This exception is currently not used by OMNIBROKER.

CORBA_MARSHAL

This exception is raised if something goes wrong in OMNIBROKER's "marshal" and "unmarshal" routines.² This indicates that either OMNIBROKER isn't working properly or, in cases where OMNIBROKER is operating with another ORB, that the other ORB is broken.

CORBA_INITIALIZE

This exception is raised if `CORBA_ORB_init()` or `CORBA_ORB::OA_init()` malfunctions, for example if wrong arguments were passed.

CORBA_NO_IMPLEMENT

This exception indicates that an implementation object (i.e. an object whose class is derived from a skeleton class) does not implement the called method.

CORBA_BAD_TYPECODE

This exception is raised if a null pointer instead of a valid type code is passed as argument or as return value. This exception is also raised if the type code of the Any argument in `CORBA_Context::set_one_value()` or the Anys in the `NVList` argument of `CORBA_Context::set_values()` are not of type string.

CORBA_BAD_OPERATION

This exception indicates that an application tried to invoke a method that does not exist. If an interface stub was used for the method invocation, this usually indicates that the stub and the skeleton were not generated from the same IDL code. If the Dynamic Invocation Interface was used, this is an indication of an incorrect operation argument.

CORBA_NO_RESOURCES

This exception is currently not used by OMNIBROKER.

²This is the code that is responsible for putting data of different types into or out of an octet stream.

CORBA_NO_RESPONSE

This exception is raised if there is a timeout. For timeouts in general, see 9.1.3 on page 89.

CORBA_PERSIST_STORE

This exception is currently not used by OMNIBROKER.

CORBA_BAD_INV_ORDER

This exception is currently not used by OMNIBROKER.

CORBA_TRANSIENT

This exception is currently not used by OMNIBROKER.

CORBA_FREE_MEM

This exception is currently not used by OMNIBROKER.

CORBA_INV_IDENT

The BOA raises this exception if a named object tries to use a name already in use. For named objects see 4.6 on page 30.

CORBA_INV_FLAG

Context objects raise this exception if an invalid `CORBA_Flags` value was specified.

CORBA_INTF_REPOS

This exception is raised by the Interface Repository. Please see A.2 on page 96 for a description of the defined minor codes.

CORBA_BAD_CONTEXT

This exception is raised by context objects if an invalid `start_scope` value was passed to `CORBA_Context::get_values()`.

CORBA_OBJ_ADAPTER

This exception is currently not used by OMNIBROKER.

CORBA_DATA_CONVERSION

This exception is currently not used by OMNIBROKER.

CORBA_OBJECT_NOT_EXIST

This exception indicates that an object reference denotes an object that does not exist. One possible reason is that the object has been destroyed on the server side.

4.7.2 Non-compliant application asserts

If the OMNIBROKER library was compiled without NDEBUB defined OMNIBROKER tries to detect common programming mistakes that lead to non-compliant CORBA applications. If such a mistake is found an error messages like this will appear:

```
Non-compliant application error detected:
Application used wrong memory allocation function
```

After detecting such an error, the OMNIBROKER library dumps a core (Unix only) and prints the file and line number where the error was detected. You can use the core in order to track down the problem with a debugger.

The following error messages can appear:

```
Application requested a feature that has not yet been
implemented
```

This is no application error. If this error message appears, a feature that has not yet been implemented in OMNIBROKER was used. In this case the only thing that can be done is to wait for the next OMNIBROKER version that has this particular feature implemented.

```
Application used wrong memory allocation function
```

If this message appears, an incorrect memory allocation function has been used. A common mistake that leads to this error is to use `malloc()`, `strdup()` and `free` (or the `new` and `delete` operator) instead of `CORBA_string_alloc()` and `CORBA_string_dup()` and `CORBA_string_free()` for string memory management.

Memory that was already deallocated was deallocated again

This message indicates multiple memory deallocations. For example, if `CORBA_string_free()` is called twice on the same string, this message will be displayed.

Object was deleted without an object reference count of zero

This message appears if an object was deleted by calling `delete` on its object reference. Never use the `delete` operator for that. Use `CORBA_release()` instead.

Object was already deleted (object reference count was already zero)

This message appears if the number of `release()` operations on an object reference is higher than the number of `duplicate()` operations.

Sequence length was greater than maximum sequence length

This message indicates that the application tried to set the length of a bounded sequence to a value greater than its maximum length.

Index for sequence operator[]() or `remove()` function was out of range

This message appears if the argument to the sequence member functions `operator[]()` or `remove()` exceeds the sequence length.

Null pointer was used to initialize `T_var` type

This message indicates an attempt to initialize a `_var` type with a null pointer.

`operator->()` was used on null pointer or nil object reference

This message indicates an attempt to use `operator->()` on an uninitialized `_var` type.

Application tried to dereference a null pointer

Some CORBA `_var` types have built-in conversion operators to a C++ reference type, i.e. some `_var` types for type `T` have a conversion operator to `T&`. This message appears if an application uses this conversion operator on an uninitialized `_var` type.

Null pointer was passed as string parameter or return value

According to the IDL-to-C++ mapping specification, no null pointers may be passed as string parameters or return values. This message appears if an application tries to do so.

Self assignment caused a dangling pointer

This message appears if the content of a `_var` type is assigned to itself. For example, the following code will lead to this error message:

```
// Somehow get a pointer to a variable struct
AVariableStruct_var var = ...

// This will result in a dangling pointer, because
// var will free its own content on assignment
AVariableStruct* ptr = var;
var = ptr;
```

Replacement of Any content by its own value caused a dangling pointer

This message appears if there is an attempt to replace the content of an `Any` by its own value. For example:

```
// Fill an Any with a string
char* s = CORBA_string_dup("Hello, world!");
CORBA_Any any;
any <<= s;

// This will result in a dangling pointer, because
// any will free its own value (which is s) on assignment
any <<= s;
```

Invalid union discriminator type used

This message appears if the discriminator type argument to `CORBA_ORB::create_union_tc()` denotes a type invalid for union discriminators. Valid types have a `CORBA_TCKind` that is one of `CORBA_tk_short`, `CORBA_tk_ushort`, `CORBA_tk_long`, `CORBA_tk_ulong`, `CORBA_tk_char`, `CORBA_tk_boolean` or `CORBA_tk_enum`.

Union discriminator mismatch

This message either indicates an attempt to set a union discriminator to an invalid value with the `_d()` modifier function or the use of a wrong accessor function, i.e. an accessor function that does not correspond to the type of the union's actual value.

Uninitialized union used

If this message appears, an uninitialized union (i.e. a union that was created with the default constructor and that was not set to any legal value) was used.

Dynamic implementation object cannot be used as static implementation object

This message appears if an attempt is made to use an DSI object implementation as a regular (i.e. static) implementation object.

Chapter 5

The OMNIBROKER Java classes

5.1 Extensions to the ORB

Like the OMNIBROKER C++ ORB the OMNIBROKER Java ORB provides for several extensions. Keep in mind that all these additional features are OMNIBROKER specific. Don't use them if you want to ensure compatibility with other ORBs.

5.1.1 Setting the Interface Repository

To set the Interface Repository used by OMNIBROKER the function `repository()` is available. The argument to `repository()` must be an object reference for an Interface Repository object. See also 4.4 on page 29.

To query the Interface Repository, `resolve_initial_references()` can be used. If the Interface Repository is not available, the exception `ORBPackage.InvalidName` is thrown. For example:

```
ORB orb = ...

org.omg.CORBA.Object obj;
obj = orb.resolve_initial_references("InterfaceRepository");

Repository repository;
repository = RepositoryHelper.narrow(obj);
```

Note that `resolve_initial_references()` is defined by the CORBA-2 standard and therefore is *not* an OMNIBROKER-specific extension.

5.1.2 Setting the Naming Service

Analogous to the Interface Repository, the Naming Service used by OMNIBROKER can be set with function `naming()`. The argument to `naming()` must be an object reference for a COS Naming Context object. For more information on the Naming Service, see 4.5 on page 30.

`resolve_initial_references()` can also be used to query the Naming Service. For example:

```
import CosNaming.*;
import CosNaming.NamingContextPackage.*;

ORB orb = ...

org.omg.CORBA.Object obj;
obj = orb.resolve_initial_references("NameService");

NamingContext nc;
nc = NamingContextHelper.narrow(obj);
```

5.1.3 Creating named implementation objects

Note: Named implementation objects are an OMNIBROKER-specific extension

By calling a special version of `connect()` on the ORB, it is possible to create named Java implementation objects, just like in C++. (See 4.6 on page 30.).

The following examples demonstrates how to use `connect()` in order to create an instance of `MyInterface_impl` with the name `my_name`:

```
public static void
main(String args[])
{
    //
    // Create ORB and BOA
    //
    ORB orb = ORB.init(args, new java.util.Properties());
    BOA boa = orb.BOA_init(args, new java.util.Properties());

    //
    // Special OmniBroker ORB and BOA functions are needed
    //
    com.ooc.CORBA.BOA OBBOA = (com.ooc.CORBA.BOA)boa;
```

```

com.ooc.CORBA.ORB OBORB = (com.ooc.CORBA.ORB)orb;

//
// Create implementation object
//
MyInterface_impl p = new MyInterface_impl();
OBORB.connect(p, "my_name");
}

```

5.2 Extensions to the BOA

The OMNIBROKER BOA provides some additional functions that might be useful to some applications. Please note that if you use these extensions, your program is not portable.

5.2.1 Getting the host name and port number

If you don't use the `-OApport` option for BOA initialization, OMNIBROKER chooses a port number for you. To find out which one, you can use the `port()` function. Similarly, you can inquire the host name used by OMNIBROKER with the `host()` function. For example:

```

import org.omg.CORBA.*;

public static void
main(String args[])
{
    ORB orb = ORB.init(args, new java.util.Properties());
    BOA boa = orb.BOA_init(args, new java.util.Properties());
    com.ooc.CORBA.BOA OBBOA = (com.ooc.CORBA.BOA)boa;

    System.out.println("Running on host \" " + OBBOA.host() +
                       "\", port number " + OBBOA.port());
}

```

5.2.2 Setting the thread model

OMNIBROKER for Java supports three different thread models, namely *single threaded*, *thread per client* and *thread per request*. The thread model can be se-

lected either when initializing the BOA with `BOA_init()` or, at a later stage, with the OMNIBROKER specific `set_thread_model()` function.

The current thread model can be inquired with `get_thread_model()`:

```
import org.omg.CORBA.*;

public static void
main(String args[])
{
    ORB orb = ORB.init(args, new java.util.Properties());
    BOA boa = orb.BOA_init(args, new java.util.Properties());
    com.ooc.CORBA.BOA OBBOA = (com.ooc.CORBA.BOA)boa;

    System.out.print("Current thread model: ");
    com.ooc.CORBA.ThreadModel threadModel = OBBOA.get_thread_model()
    switch(threadModel.value())
    {
        case com.ooc.CORBA.ThreadModel._SingleThreaded:
            System.out.println("single threaded");
            break;

        case com.ooc.CORBA.ThreadModel._ThreadPerClient:
            System.out.println("thread per client");
            break;

        case com.ooc.CORBA.ThreadModel._ThreadPerRequest:
            System.out.println("thread per request");
            break;
    }
}
```

With `set_thread_model()` the thread model can be changed anytime, for example:

```
OBBOA.set_thread_model(com.ooc.CORBA.ThreadModel.ThreadPerRequest);
```

Chapter 6

The OMNIBROKER code generators

OMNIBROKER includes the following IDL-to-C++ and IDL-to-Java translator, the IDL and HTML code generator and Interface Repository tools:

`idl`

The OMNIBROKER IDL-to-C++ Translator

`jidl`

The OMNIBROKER IDL-to-Java Translator

`hidl`

The OMNIBROKER IDL-to-HTML Translator

`irserv`

The OMNIBROKER Interface Repository Server

`irfeed`

The OMNIBROKER Interface Repository Feeder

`irdel`

The OMNIBROKER Interface Repository Deleter

`irgen`

The OMNIBROKER Interface Repository C++ Code Generator

6.1 Synopsis

`idl [options] idl-files ...`

```
jidl [ options ] idl-files ...  
hidl [ options ] idl-files ...  
irserv [ options ] idl-files ...  
irfeed [ options ] repository-IOR idl-files ...  
irdel [ options ] repository-IOR scoped-name ...  
irgen repository-IOR name-base
```

6.2 Description

`idl` is the OMNIBROKER IDL-to-C++ translator. It translates IDL files into C++ files. For each IDL file, four C++ files are generated. For example

```
idl MyFile.idl
```

produces the files:

`MyFile.h`

This is the header file containing `MyFile.idl`'s translated data types and interface stubs.

`MyFile.cpp`

This is the source file containing `MyFile.idl`'s translated data types and interface stubs.

`MyFile_skel.h`

This is the header file containing skeletons for `MyFile.idl`'s interfaces.

`MyFile_skel.cpp`

This is the source file containing skeletons for `MyFile.idl`'s interfaces.

`jidl` translates IDL files into Java files. For every construct in the IDL file that maps to a Java class or interface, a separate class file is generated. Directories are automatically created for those IDL constructs that map to a Java package (e.g. a module).

`jidl` can also add comments from the IDL file starting with `/**` to the generated Java files. This allows to use the `javadoc` tool to document the generated Java files. See 6.13 on page 55 for additional information.)

`hidl` creates HTML files from IDL files. For each module and interface defined in an IDL file an HTML file is generated. Comments included in the IDL file are handled as well as `javadoc` style keywords. 6.12 on page 55 provides more information.

`irserv` is the Interface Repository Server. Together with `irfeed`, a program that feeds the Interface Repository with IDL code, and `irgen`, the Interface Repository C++ Code Generator, it is also possible to generate C++ code directly from the content of an Interface Repository. See 6.10 on page 53 for an example.

6.3 Options for `idl`

- `-h, --help`
Show a short help message.

- `-v, --version`
Show the OMNIBROKER version number.

- `-e, --cpp NAME`
Use NAME as preprocessor program.

- `-d, --debug`
Print diagnostic messages. This option is for OMNIBROKER internal debugging purposes only.

- `-DNAME`
Defines NAME as 1. This option is directly passed to the preprocessor.

- `-DNAME=DEF`
Defines NAME as DEF. This option is directly passed to the preprocessor.

- `-UNAME`
Removes any definition for NAME. This option is directly passed to the preprocessor.

- `-IDIR`
Adds DIR to the include file search path. This option is directly passed to the preprocessor.

- `--no-skeletons`
Don't generate skeletons classes.

- `--no-type-codes`
Don't generate type codes and insertion and extraction functions for the any type. If you use this option the code generated gets more compact.
- `--tie`
Generate tie classes for delegate-based interface implementations. Tie classes depend on the corresponding skeleton classes, i.e. you must not use `--no-skeletons` in combination with `--tie`.
- `--c-suffix SUFFIX`
Use `SUFFIX` as suffix for source files. The default value is `.cpp`.
- `--h-suffix SUFFIX`
Use `SUFFIX` as suffix for header files. The default value is `.h`.
- `--all`
Generate code for included files instead of inserting `#include` statements. See 6.11 on page 54.
- `--no-relative`
When creating code, `idl` assumes that the same `-I` options that are used with `idl` are also going to be used with the C++ compiler. Therefore `idl` will try to make all `#include` statements relative to the directories specified with `-I`. The option `--no-relative` supresses this behavior, i.e. this option tells `idl` not to make `#include` statements for included files relative to the paths specified with the `-I` option.
- `--header-dir DIR`
This option can be used to make `#include` statements for header files relative to a specific directory.

6.4 Options for `jidl`

- `-h, --help`
- `-v, --version`
- `-e, --cpp NAME`
- `-d, --debug`
- `-DNAME`

-DNAME=DEF

-UNAME

-IDIR

These options are the same as for the `idl` command.

--no-skeletons

Don't generate skeletons classes.

--no-comments

The default behavior of `jidl` is to add any comments from the IDL file starting with `/**` to the generated Java files. Specify this option if you don't want these comments added to your Java files.

--all

Generate code for included files instead of inserting `#include` statements. See 6.11 on page 54.

--auto-package

Derives the package names for generated Java classes from the IDL prefixes. The prefix "ooc.com", for example, results in the package "com.ooc".

--package PKG

Specifies a package name for the generated Java classes. Each class will be generated relative to this package.

--prefix-package PRE PKG

Specifies a package name for a particular prefix. Each class with this prefix will be generated relative to the specified package.

--output-dir DIR

Specifies a directory where `jidl` will place the generated Java files. Without this option the current directory is used.

6.5 Options for hidl

-h, --help

-v, --version

-e, --cpp NAME

-d, --debug

-DNAME

-DNAME=DEF

-UNAME

-IDIR

These options are the same as for the `idl` command.

--no-sort

Don't sort symbols alphabetically.

6.6 Options for `irserv`

-h, --help

-v, --version

-e, --cpp NAME

-d, --debug

-DNAME

-DNAME=DEF

-UNAME

-IDIR

These options are the same as for the `idl` command.

--ior

Print a stringified IOR of the Interface Repository on standard output.

6.7 Options for `irfeed`

-h, --help

-v, --version

-e, --cpp NAME

-d, --debug

-DNAME

-DNAME=DEF

-UNAME

-IDIR

These options are the same as for the `idl` command.

6.8 Options for `irdel`

-h, --help

-v, --version These options are the same as for the `idl` command.

6.9 Options for `irgen`

-h, --help

-v, --version

--no-skeletons

--no-type-codes

--tie

--c-suffix SUFFIX

--h-suffix SUFFIX

--header-dir DIR

These options are the same as for the `idl` command.

6.10 The IDL-to-C++ translator and the Interface Repository

The OMNIBROKER IDL-to-C++ and IDL-to-Java translators internally use the Interface Repository for generating code. That is, these programs have their own private Interface Repository, that is fed with the specified IDL files. All code is generated from that private Interface Repository.

It is also possible to generate code from a global Interface Repository. First the command `irserv` must be used to start the Interface Repository. Then the Interface Repository must be fed with the IDL code, using the command `irfeed`. With `irgen` the actual C++ code is generated. For example:

```
irserv --ior > IntRep.ref &  
irfeed 'cat IntRep.ref' file.idl  
irgen 'cat IntRep.ref' file
```

The IDL-to-C++ translator `idl` performs all these steps at once, in a single process with a private Interface Repository. Thus, you just have to issue:

```
idl file.idl
```

6.11 #include statements

If you use the `#include` statement in your IDL code, the OMNIBROKER IDL-to-C++ translator `idl` will not create code for included IDL files. The translator will insert the appropriate `#include` statements in the generated header files instead. Please note that there are several restrictions on where to place the `#include` statements in your IDL files for this feature to work properly:

- `#include` may only appear at the beginning of your IDL files. All `#include` statements must be placed before the rest of your IDL code.¹
- Type definitions, e.g. interface definitions or struct definitions, may not be split among several IDL files, i.e. no `#include` statement may appear within such definitions.

If you don't want these restrictions to be applied, you can use the translator option `--all` with `idl`. With this option the IDL-to-C++ compiler treats code from included files as if this code appeared in your IDL file at the position where it is included. This means that the compiler will not place `#include` statements in the automatically generated output header files since it will always write all code to single output files. This is regardless of whether the code comes directly from your IDL file or from files included by your IDL file.

The above only works for the command `idl`. It does not work if code is generated from an Interface Repository with `irgen`. That is, `irgen` behaves like `idl` with an implicit `--all` option.

¹Preprocessor statements like `#define` or `#ifdef` may be placed before your `#include` statements.

6.12 Documenting IDL files

With the OMNIBROKER IDL-to-HTML translator `hidl` you can easily generate HTML files containing IDL interface descriptions. For each module and interface `hidl` generates a neatly formatted HTML file.

Basically `hidl` supports a `javadoc` compatible formatting style. The following keywords are recognized:

`@exception exception-name description`
Adds an exception description to the exception list.

`@param parameter-name description`
Adds a parameter description to the parameter list.

`@return description`
Adds descriptive text for the return value.

`@see reference`
Adds a “See also” note.

`@since since-text`
Comment related to the availability of new features.

In addition, `hidl` supports a `@member` keyword, which can be used to document struct, union, enum or exception members:

`@member member-name description`
Adds a member description to the member list.

Like `javadoc`, `hidl` uses the first sentence in the documentation comment as summary sentence. This sentence ends at the first period that is followed by a blank, tab or line terminator, or at the first `@`.

6.13 Using javadoc

If not explicitly suppressed with the `--no-comments` option, the OMNIBROKER IDL-to-Java translator `jidl` adds comments starting with `/**` from the IDL file to the generated Java files, so that `javadoc` can be used to generate documentation (as long as the comments are in a format compatible `javadoc`).

Here is an example which shows how to include a documentation in an IDL interface description file. Let’s assume we have an interface `I` in a module `M`:

```
// IDL

module M
{

/**
 *
 * This is a comment related to interface I.
 *
 * @author Uwe Seimet
 *
 * @version 1.0
 */
interface I
{

    /**
     *
     * This comment describes exception E.
     *
     */
    exception E { };

    /**
     *
     * The description for operation S.
     *
     * @param arg A dummy argument.
     *
     * @return A dummy string.
     *
     * @exception E Raised under certain circumstances.
     *
     */
    string S(in long arg)
        raises(E);
};

};
```

When running `jid1` on this file the comments will automatically be added to the generated Java files `M/I.java` and `M/IPackage/E.java`. For `I.java` the generated code looks as follows:

```
// Java

package M;

//
// IDL:M/I:1.0
//
/**
 * This is a comment related to interface I.
 *
 * @author Uwe Seimet
 *
 * @version 1.0
 *
 **/
public interface I extends org.omg.CORBA.Object
{
    //
    // IDL:M/I/S:1.0
    //
    /**
     *
     * The description for operation S.
     *
     * @param arg A dummy argument.
     *
     * @return A dummy string.
     *
     * @exception M.IPackage.E Raised under certain circumstances.
     *
     **/
    public String
    S(int arg)
        throws M.IPackage.E;
}
```

Note that `jid1` automatically inserts the fully qualified Java name for the ex-

ception E, in this case M.IPackage.E.

These are the contents of IPackage/E.java:

```
// Java

package M.IPackage;

//
// IDL:M/I/E:1.0
//
/**
 *
 * This comment describes exception E.
 *
 */
final public class E extends org.omg.CORBA.UserException
{
    public
    E()
    {
    }
}
```

Now you can use javadoc to extract the comments from the generated Java files and convert them to a neatly formatted HTML documentation.

For additional information please refer to the javadoc documentation.

Chapter 7

The IDL-to-C++ mapping

OMNIBROKER implements the IDL-to-C++ mapping as described in [1]. The standard IDL-to-C++ mapping is not a topic of this manual. Please refer to [1] for the exact specifications.

7.1 Reserved names

All names starting with with `OB`, `_OB_` or `_ob_` are reserved by OMNIBROKER for internal use and must not be used as identifiers. (Who wants to use such ugly names anyway?)

7.2 Mapping of modules

Generally IDL modules are mapped to C++ namespaces. However, since most C++ compilers currently do not support namespaces, the IDL-to-C++ mapping defines two alternatives. The first one maps modules to C++ classes, implying that nested classes are needed for interfaces or other modules defined within a module. The second alternative is to map modules to name prefixes, e.g. the name of an interface `I` in a module `M` is mapped to `M_I`.

OMNIBROKER uses the name prefix mapping alternative for the following reasons:

- As mentioned earlier, C++ namespaces are not widely available yet. OMNIBROKER was designed to be portable among a variety of C++ compilers. Therefore using namespaces was not possible.
- Although nested classes are available with most C++ compilers, this mapping alternative has the disadvantage that modules cannot be “reopened” (since

classes cannot be reopened). That is, it is not possible to define in one IDL file one part of a module and in another IDL file another part of the same module.

7.3 Implementing interfaces

7.3.1 Inheritance-based implementation

OMNIBROKER can use C++ inheritance for interface implementation. To implement an interface `I`, an implementation class must be written. By convention, the name of this class is the name of the interface with the suffix `_impl`, i.e. for an interface `I`, the implementation class is named `I_impl`.

`I_impl` must inherit from the skeleton class `I_skel`, which is automatically generated by the IDL-to-C++ translator. If `I` inherits from other interfaces, for example from the interfaces `A` and `B`, `I_impl` must also inherit from the corresponding implementation classes `A_impl` and `B_impl`.

Note that `virtual public` C++ inheritance must be used. The only case when the keyword `virtual` is not necessary is for an interface `I` which does not inherit from any other interface and from which no other interface inherits. This means that the implementation class `I_impl` only inherits from the skeleton class `I_skel` and no implementation class inherits from `I_impl`.

Here is an example:

```
// IDL

interface A
{
    void op_a();
};

interface B
{
    void op_b();
};

interface I : A, B
{
    void op_i();
};
```

The corresponding C++ code:

```
// C++

class A_impl : virtual public A_skel
{
public:

    A_impl();
    virtual void op_a();
};

class B_impl : virtual public B_skel
{
public:

    B_impl();
    virtual void op_b();
};

class I_impl : virtual public I_skel,
                virtual public A_impl,
                virtual public B_impl
{
public:

    I_impl();
    virtual void op_i();
};
```

As you can see from this example, an interface operation is implemented by a corresponding C++ member function (which must be declared `virtual`).

It is not strictly necessary to have an implementation class for every interface. For example, it is sufficient to only have the class `I_impl` as long as `I_impl` implements all interface operations, including the operations of the base interfaces:

```
// C++

class I_impl : virtual public I_skel
{
public:
```

```

    I_impl();
    virtual void op_a();
    virtual void op_b();
    virtual void op_i();
};

```

Note that since the constructor of a skeleton class is protected, it is necessary to define a public constructor in the implementation class, even if this constructor is empty.

See 4.6 on page 30 for examples on how to create implementation objects from your implementation classes.

7.3.2 Delegation-based implementation

Sometimes it is not adequate to take an inheritance-based approach for implementing an interface. This especially applies if inheritance results in an implementation being incompatible with existing legacy code.

OMNIBROKER also provides for interface implementations using a delegation-based mechanism. The key to this approach are special C++ template classes that form so called tie classes. These classes are derived from the corresponding skeleton classes and have the same names as these classes, with the prefix `_tie` appended. For the interface `I` from the example above, the template `I_skel_tie` is instantiated with a class type `I_impl_tie` that must all operations of `I`.

In contrast to the inheritance-based approach, it is not necessary for `I_impl_tie` to be derived from any skeleton class. Instead, an instance of `I_skel_tie` delegates all operation calls to `I_impl_tie`.

```

// C++

class I_impl_tie
{
public:

    virtual void op_a();
    virtual void op_b();
    virtual void op_i();
};

```

The `I_skel_tie` template is then instantiated like this:

```

// C++

```

```

//
// Create implementation object
//
I_var p1 = new I_skel_tie< I_impl_tie >(new I_impl_tie);

//
// Create named implementation object
//
I_var p2 = new I_skel_tie< I_impl_tie >("name", new I_impl_tie);

```

For more information on delegation-based object implementations refer to [3], section 18.1.4.

7.4 Extensions

OMNIBROKER provides several extensions to the standard IDL-to-C++ mapping. If you are concerned about source code compatibility with CORBA-compliant ORBs from other vendors, you should not use these extensions. If you plan, however, to use your source code exclusively with OMNIBROKER these extensions will reduce programming overhead.

7.4.1 Extensions to the string type

The OMNIBROKER `CORBA_String_var` type provides the `operator+=()` for appending to the string. The argument to `operator+=()` can be of type `const char*`, `char` and unsigned `char` as well as short, unsigned short, `int`, unsigned `int`, `long` and unsigned `long`. For example:

```

CORBA_String_var s;      // s is empty
s += "abc";              // s is "abc"
s += 'x';                // s is "abcx"
s += 'y';                // s is "abcxy"
s += 'z';                // s is "abcxyz"
s += 12345;              // s is "abcxyz12345"

```

7.4.2 Extensions to the `_var` types

All `_var` types have the following member functions:

`in()`: This function converts the `_var` type to a type suitable for `in` parameters.

`inout()`: This function converts the `_var` type to a type suitable for `inout` parameters.

`out()`: This function converts the `_var` type to a type suitable for `out` parameters. As a side effect, this function ensures that the value held by the `_var` is released or freed, by either calling `CORBA_string_free()` (in case of a string), `CORBA_release()` (in case of an object reference) or `delete` (in case of types like sequences, variable-length structs etc.).

`_retn()`: This function converts the `_var` type to a type suitable for function return values. The `_retn()` function also removes the value that is held by the `_var` type without destroying it, i.e. without calling `delete`, `CORBA_string_free()` or `CORBA_release()` on its value. For example consider a function `f()` that returns its three `in` string arguments as a single string:

```
char*
f(const char* s1, const char* s2, const char* s3)
{
    CORBA_String_var s = s1;
    s += s2;
    s += s3;
    return s._retn();
}
```

Please note that these functions are not covered by the current CORBA IDL-to-C++ mapping, but it is likely that they will become a part of the standard for the next major mapping revision.

7.4.3 Extensions to the sequence types

All unbounded non-array sequences (for example unbounded string, struct and object reference sequences) have an additional `insert()`, `append()` and `remove()` member function. For a sequence `s` and a value `v`, the `s.insert(v)` and `s.append(v)` behave as follows:

```
s.length(s.length() + 1);
... // Somehow shift sequence contents one to the right
s[0] = v;
```

and

```
s.length(s.length() + 1)
s[s.length() - 1] = v;
```

respectively.

Please note that OMNIBROKER's sequence implementation does not really shift the contents of the sequence. It is rather implemented as a "double ended queue" (like the STL¹ "deque"), and therefore needs no value shifting. That is, the `insert()` function is as efficient as the `append()` function.

¹Standard Template Library.

Chapter 8

C++ Tips & Tricks

Unfortunately there are some traps & pitfalls in the memory management handling of the IDL-to-C++ mapping. This chapter gives tips on how to avoid common problems.

8.1 Strings

When using CORBA strings, always remember the following rules:

8.1.1 CORBA-specific string functions

Use the CORBA-specific string functions `CORBA_string_alloc()`, `CORBA_string_free()` and `CORBA_string_dup()` if you're dealing with CORBA strings. Never use `new`, `delete`, `malloc()`, `free()`, `strdup()` or similar functions.

For example, the following code is incorrect:

```
// Error, CORBA_string_dup() must be used instead of strdup()
char* s1 = strdup("Hello!");

// Allocate a string for 10 characters + trailing '\0' ...

// No! CORBA_string_alloc() must be used!
String_var s2 = malloc(11);
```

This is the correct version:

```
// OK, CORBA_string_dup() is fine
```

```
char* s1 = CORBA_string_dup("Hello!");

// Allocate a string for 10 characters + trailing '\0' ...

// OK. Note that CORBA_string_alloc (unlike malloc) adds an
// additional character for the trailing '\0' automatically
CORBA_String_var s2 = CORBA_string_alloc(10);
```

This code is wrong, too:

```
// No! Use CORBA_string_free()!
free(s2);
```

And again, the corrected version:

```
// OK, this is correct
CORBA_string_free(s1);

// There is no need to free s2 explicitly since CORBA_String_var
// types release the string they manage automatically when the
// CORBA_String_var type is destroyed.
```

8.1.2 Initialization and assignment from `char*` and `const char*`

Initialization of a `CORBA_String_var` type or assignment to a `CORBA_String_var` type from a `char*` type value *consumes* that value. That means that if the `CORBA_String_var` is destroyed, the value from which the `CORBA_String_var` was initialized or that was assigned to the `CORBA_String_var` will *also be destroyed*.

Initialization of a `CORBA_String_var` type or assignment to a `CORBA_String_var` type from a `const char*` type value *duplicates* that value. This means that if the `CORBA_String_var` is destroyed, the value from which the `CORBA_String_var` was initialized or that was assigned to the `CORBA_String_var` is *not destroyed*.

Note that for compatibility reasons with C the type of strings in C++ is `char*`, *not* `const char*`. So the following code is wrong:

```
// Error, since "Hello!" is char*, not const char*
CORBA_String_var s = "Hello!";
```

The following code is OK:

```
// OK, s1 consumes value returned by CORBA_string_dup()
CORBA_String_var s1 = CORBA_string_dup("Hello!");

// OK, s2 will implicitly duplicate "Hello!"
CORBA_String_var s2 = (const char*)"Hello!";
```

8.1.3 Initialization and assignment from CORBA_String_var

Initialization of a CORBA_String_var type or assignment to a CORBA_String_var type from another CORBA_String_var type value automatically duplicates that value. This means that it is not necessary to use explicit calls to CORBA_string_dup(). The following examples are correct:

```
CORBA_String_var s1 = CORBA_string_dup("ABC");

// OK, s2 will implicitly duplicate "ABC"
CORBA_String_var s2 = s1;

// Also OK, explicit duplication
CORBA_String_var s3 = CORBA_string_dup(s1);
```

Note that string elements of a structure, elements of a string array and elements of a string sequence behave exactly like the CORBA_String_var type¹, i.e. you can deliberately assign between these types or use one of these types to initialize any other of these types. There is no need to call CORBA_string_dup() explicitly.

8.1.4 Strings as parameters and return values

If a function returning a string value via an out or inout parameter or as a return value is called, the callee must *duplicate* and the caller must *release* this value. The duplication can be done by using CORBA_string_dup() and the release by either explicitly calling CORBA_string_free() or by assigning the value to a CORBA_String_var. For example:

```
// IDL
```

```
interface I
```

¹In code generated by the OMNIBROKER IDL-to-C++ translator, array and structure string elements *are* actually of type CORBA_String_var. String sequence elements are not of type CORBA_String_var (for technical reasons), but the type used for string sequence elements behaves exactly like the CORBA_String_var type.

```
{
    string op(out string os, inout string ios);
};
```

The following implementation of I's op() operation is wrong:

```
// C++

class I_impl : virtual public I_skel
{
public:

    I_impl() { }
    virtual char* op(char*& os, char*& ios)
    {
        // Forgot to release the inout string parameter ios
        ios = "abc"; // Wrong. String must be duplicated.
        os = "def"; // Ditto
        return "ghi"; // Ditto
    }
};
```

Here is the correct version:

```
// C++

class I_impl : virtual public I_skel
{
public:

    I_impl() { }
    virtual char* op(char*& os, char*& ios)
    {
        CORBA_string_free(ios);
        ios = CORBA_string_dup("abc");
        os = CORBA_string_dup("def");
        return CORBA_string_dup("ghi");
    }
};
```

Here is an example on how to use string out, inout or return values on the caller side if CORBA_string_free() is used:

```
// C++

I_ptr i = ... // Get a reference to an I somehow

char* out;
char* inOut = CORBA_string_dup("This is my inout arg");
char* result;

result = i -> op(out, inOut);

CORBA_string_free(out);
CORBA_string_free(inOut);
CORBA_string_free(result);
```

The same example, but with self-managed `CORBA_String_var()` types instead of explicitly calling `CORBA_string_free()`:

```
// C++

I_ptr i = ... // Get a reference to an I somehow

CORBA_String_var out;
CORBA_String_var inOut = CORBA_string_dup("This is my inout arg");
CORBA_String_var result;

result = i -> op(out, ios);

// No explicit calls to CORBA_string_free are necessary, since the
// CORBA_String_var type destroys its contents automatically
```

Since method two in this example is much less error prone, you should always use the self-managed type `CORBA_String_var` in such a case.

8.2 Object references

If you use CORBA object references, i.e. `_ptr` and `_var` types for specific interfaces, keep the following in mind.

8.2.1 Object reference memory management

Object references are implemented as counted pointers. That is, if a new object reference is created, either with functions like `CORBA_string_to_object()` (for a proxy object on the client side, see 4.6 on page 30), or with `new` (for an implementation object on the server side), the initial reference count is set to one. The proxy (on the client side) or the implementation (on the server side) will be destroyed automatically if the reference count becomes zero. The reference count is decremented by one with `CORBA_release()` and incremented by one with `_duplicate()`. Never use the `delete` operator to destroy objects. Only use `CORBA_release()`. For example, the following code is wrong:

```
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
const char* s = ... // Obtain a stringified X reference somehow
X_ptr = orb -> string_to_object(s); // Convert string to object
delete p; // Never, ever try to do this
```

This is the correct version:

```
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
const char* s = ... // Obtain a stringified X reference somehow
X_ptr = orb -> string_to_object(s); // Convert string to object
release(p); // This is fine
```

You should use self-managed types whenever possible:

```
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
const char* s = ... // Obtain a stringified X reference somehow
X_var = orb -> string_to_object(s); // Convert string to object
// No release necessary, _var types provide for automatic release
```

It is also important that implementation objects on the server side are always allocated on the heap with the `new` operator. Never allocate implementation objects on the stack. If you do so, the object will be destroyed if the stack unwinds, without any calls to `CORBA_release()`. The following code demonstrates the problem:

```
#include <OB/CORBA.h>
#include <MyInterface_impl.h>

int
main(int argc, char* argv[], char*[])
```

```

{
    //
    // Create ORB and BOA
    //
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    //
    // Implementation object on the stack! Disaster!
    //
    MyInterface_impl i;
    MyInterface_ptr p = &i;

    //
    // Run implementation
    //
    boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}

```

The following code shows a working version:

```

#include <OB/CORBA.h>
#include <MyInterface_impl.h>

int
main(int argc, char* argv[], char*[])
{
    //
    // Create ORB and BOA
    //
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

    //
    // Implementation object created with new, automatically
    // released by MyInterface_var. That's fine.
    //
    MyInterface_var p = new MyInterface_impl();

    //

```

```

    // Run implementation
    //
    boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}

```

8.2.2 Object references as parameters and return values

If a function returning an object reference via an out or inout parameter or as a return value is called, the callee must *duplicate* and the caller must *release* the reference. As described above, an object reference to an object of type I (i.e. an object with the interface I) is duplicated with `I::_duplicate()` and released with `CORBA_release()`. This is quite similar to strings as parameters and return values. For example:

```

// IDL

interface I {... };

interface A
{
    I op(out I oref, inout I ioref);
};

```

This implementation of the `op()` operation is wrong:

```

// C++

class A_impl : virtual public A_skel
{
    I_var myref;

public:

    A_impl()
    {
        myref = ... // Initialize myref somehow
    }

    virtual I_ptr op(I_ptr& oref, I_ptr& ioref)
    {
        // Forgot to release the inout object reference parameter ic

```

```

        ioref = myref; // Wrong. Reference must be duplicated.
        oref = myref; // Ditto
        return myref; // Ditto
    }
};

```

This version is correct:

```

// C++

class A_impl : virtual public A_skel
{
    I_var myref;

public:

    A_impl()
    {
        myref = ... // Initialize myref somehow
    }

    virtual I_ptr op(I_ptr& oref, I_ptr& ioref)
    {
        CORBA_release(ioref);
        ioref = I::_duplicate(myref);
        oref = I::_duplicate(myref);
        return I::_duplicate(myref);
    }
};

```

The first example on how to use object reference out, inout or return values on the caller side uses explicit calls to `CORBA_release()`:

```

// C++

A_ptr a = ... // Get a reference to an A somehow

I_ptr out;
I_ptr inOut = ... // Get a reference to an I somehow
I_ptr result;

```

```

result = a -> op(out, inOut);

CORBA_release(out);
CORBA_release(inOut);
CORBA_release(result);

```

The second example uses self-managed I_var types:

```

// C++

A_ptr a = ... // Get a reference to an A somehow

I_var out;
I_var inOut = ... // Get a reference to an I somehow
I_var result;

result = i -> op(out, ios);

// No explicit calls to CORBA_release() are necessary, since the
// I_var type releases the reference it holds automatically.

```

We recommend to use method two with the self-managed types, since this method is much less error prone.

8.2.3 Implementing a “destroy” function

In many cases it is necessary that a client can destroy an implementation object on the server by calling one of its operations. For example, consider the following two interfaces:

```

// IDL

interface X;

interface XFactory
{
    X create(); // Creates a new X object
};

interface X
{

```

```
... // Other functions

void destroy(); // Destroys X object
};
```

Here a factory object is used to create X objects, which subsequently can be destroyed by calling the `destroy()` operation on the created X objects. The `create()` and `destroy()` function can be implemented as follows:

```
// C++

class XFactory_impl : virtual public XFactory_skel
{
public:

    XFactory_impl() { }

    create()
    {
        X_ptr p = new X_impl();
        return X::_duplicate(p);
    }
}

class X_impl : virtual public X_skel
{
public:

    X_impl() { }

    destroy()
    {
        X_var self = _this();
        CORBA_release(self);
    }
}
```

In this example, the factory object doesn't do any "bookkeeping" about the implementation objects it generates. The `create()` function simply creates a new implementation object and returns an object reference to this object to the client. After the return of `create()` the reference count of the new X is 1:

- The `new` sets the initial reference count to 1.
- The `_duplicate()` increases the reference count by 1.
- The caller of `create()` (which can be a user supplied function or the ORB in case of an up-call) decreases the reference count by 1 by calling `CORBA_release()`.

That means that the next call to `CORBA_release()` destroys the implementation object. This is how the function `destroy()` is implemented.

8.2.4 Getting an implementation object from a reference

Since in OMNIBROKER implementation classes are derived from skeleton classes, which are derived from stub classes, you can use the C++ `dynamic_cast<>()` to cast an object reference to a pointer to the implementation object. In case your compiler does not support RTTI², you can use the OMNIBROKER header file `Narrow_impl.h` to `_narrow()` to an implementation class. For example:

```
// IDL
```

```
interface I { ... };
```

Here an example that uses `dynamic_cast<>()`:

```
// C++
```

```
class I_impl : virtual public I_skel { ... };
```

```
void foo(I_ptr ref)
```

```
{
```

```
    I_impl* p = dynamic_cast<I_impl*>(ref);
```

```
    if(p)
```

```
    {
```

```
        // The implementation for ref is in the same process
```

```
    }
```

```
    else
```

```
    {
```

```
        // The implementation for ref is not in the same process
```

²Runtime Type Identification

```

    }
}

```

Here an example with the `OB_MAKE_NARROW_IMPL` macros from the header file `OB/Narrow_impl.h`:

```

// C++

#include <OB/Narrow_impl.h>

class I_impl : virtual public I_skel
{
    ...

    OB_MAKE_NARROW_IMPL(I_impl)
};

OB_MAKE_NARROW_IMPL_1(I_impl, I_skel)

void foo(I_ptr ref)
{
    I_impl* p = I_impl::_narrow_impl(ref);

    if(p)
    {
        // The implementation for ref is in the same process
    }
    else
    {
        // The implementation for ref is not in the same process
    }
}

```

The macro `OB_MAKE_NARROW_IMPL_1()` can only be used if your implementation class has exactly one super class (the skeleton class). If your implementation class has two or more super classes, use the macro `OB_MAKE_NARROW_IMPL_n()`, where **n** is the number of super classes. For example:

```

// IDL

```

```

interface A { ... };
interface B { ... };
interface C : A, B { ... };

// C++

class C_impl : virtual public C_skel,
               virtual public A_impl,
               virtual public B_impl
{
    ...

    OB_MAKE_NARROW_IMPL(C_impl)
};

OB_MAKE_NARROW_IMPL_3(C_impl, C_skel, A_impl, B_impl)

```

If you are using OMNIBROKER on multiple platforms, where some support RTTI and others don't, it might be best to always use `OB/Narrow_impl.h`, since `_narrow_impl()` will automatically use `dynamic_cast<>()` on those platforms where it is available. However, if you also want to compile your code with ORBs from other vendors, you should not use `_narrow_impl()`, since this is an OMNIBROKER specific extension.

8.2.5 Cyclic object dependencies

Consider the following code:

```

class X_impl : public X_skel // Implements interface X
{
    Y_var y_;

public:
    X_impl() { }

    void setY(Y_ptr y) { y_ = Y::_duplicate(y); }
};

class Y_impl : public Y_skel // Implements interface Y
{

```

```

    X_var x_;

public:

    Y_impl() { }

    void setX(X_ptr x) { x_ = X::_duplicate(x); }
};

void f()
{
    //
    // Create a X_impl and a Y_impl
    //
    X_var x = new X_impl;
    Y_var y = new Y_impl;
    x -> setY(y);
    y -> setX(x);

    //
    // Do something with x and y
    //
    ...
}

```

Here the `X_impl` has a reference to the `Y_impl` and the `Y_impl` has a reference to the `X_impl`, what is known as a “cyclic object dependency”. This means that when `f()` returns, even though `x` and `y` get destroyed, the objects they are referring to are *not* destroyed since the reference count never becomes zero. Why? Let’s have a deeper look into what happens in the example program:

```
X_var x = new X_impl
```

The initial reference count of the `X_impl` after the `new` is one.

```
Y_var y = new Y_impl
```

Same as above, the initial reference count of the `Y_impl` is one.

```
x -> setY(y)
```

After the `setY()`, the reference count of the `Y_impl` is two.

```
y -> setX(x)
```

After the `setX()`, the reference count of the `X_impl` is two.

```
return
    x and y get destroyed and therefore call CORBA_release() on their con-
    tents. So that the reference count of the X_impl and the Y_impl is one.
```

This means that after the return of `f()` the `X_impl` and the `Y_impl` will live forever.

This problem can be solved by adding a `releaseInternal()` function³ to at least one of the two interface implementations. For example:

```
class X_impl : public X_skel // Implements interface X
{
    Y_var y_;

public:

    X_impl() { }

    void setY(Y_ptr y) { y_ = Y::_duplicate(y); }
};

class Y_impl : public Y_skel // Implements interface Y
{
    X_var x_;

public:

    Y_impl() { }

    void setX(X_ptr x) { x_ = X::_duplicate(x); }
    void releaseInternal() { x_ = X::_nil(); }
};

void f()
{
    //
    // Create a X_impl and a Y_impl
    //
    X_var x = new X_impl;
    Y_var y = new Y_impl;
```

³Of course you are free to choose whatever name you like.

```

x -> setY(y);
y -> setX(x);

//
// Do something with x and y
//
...

//
// Call releaseInternal
//
y -> releaseInternal();
}

```

Now both the `X_impl` and the `Y_impl` get destroyed at the return of `f()`:

```
X_var x = new X_impl
```

The initial reference count of the `X_impl` after the `new` is one.

```
Y_var y = new Y_impl
```

Same as above, the initial reference count of the `Y_impl` is one.

```
x -> setY(y)
```

After the `setY()`, the reference count of the `Y_impl` is two.

```
y -> setX(x)
```

After the `setX()`, the reference count of the `X_impl` is two.

```
y -> releaseInternal()
```

The `releaseInternal()` function sets `x_` value of the `Y_impl` to `X::_nil()`. Assignment to a `_var` object reference causes `CORBA_release()` to be called on its contents. So now the reference count of the `X_impl` is one.

```
return
```

`x` and `y` are destroyed and therefore call `CORBA_release()` on their contents. That means that the reference count of the `X_impl` becomes zero, resulting in `X_impl` being destroyed. This of course also eliminates `X_impl`'s `y_` data member, causing `CORBA_release()` to be called on the `Y_impl`. So the `Y_impl`'s reference count also becomes zero and the `Y_impl` is also destroyed.

8.2.6 String _var types and object reference _var types differences

There is a slight but important difference between string _var types and object reference _var types regarding their initialization or assignment from in parameters. Consider the following IDL code:

```
// IDL

interface Y;

interface X
{
    void init(in string s1, in string s2, in Y y1, in Y y2);
};
```

Here the `init()` function is used to initialize an `X` with two strings and two `Y` object references. The following code shows the difference between _var type assignments from strings and from object references:

```
// C++

// Implementation for interface X
class X_impl : virtual public X_skel
{
    CORBA_String_var s1_;
    CORBA_String_var s2_;
    Y_var y1_;
    Y_var y2_;

public:

    X_impl() { }

    void init(const char* s1, const char* s2, Y_ptr y1, Y_ptr y2)
    {
        s1_ = s1; // OK, CORBA_String_var automatically
                // duplicates const char*
        y1_ = y1; // Error, y1_ consumes y1!

        s2_ = CORBA_string_dup(s2); // This is OK
        y2_ = Y::_duplicate(y2); // Also OK
    }
};
```

```

    }
}

```

The reason for this behavior is that there is no such thing as a constant object reference for `in` parameters. Therefore it is not possible for the object reference `_var` type to distinguish between assignments from regular object references and `in` object references.

8.2.7 Global `_var` type object references

Beware! You should never have global `_var` type object references, because you can never tell exactly when and in which order they will be destroyed. For example, it is possible that a reference is destroyed *after* the ORB was destroyed:

```

#include <OB/CORBA.h>
#include <MyInterface.h>

MyInterface_var i; // Global _var type object reference

int
main(int argc, char* argv[], char*[])
{
    //
    // Create ORB
    //
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);

    //
    // Get MyInterface object reference somehow
    //
    i = ...

    //
    // Do something with i
    //
    i -> ...

    //
    // ORB is destroyed on return of main. i will be destroyed
    // after the ORB. Total Disaster!

```

```
    //  
    return 0;  
}
```

The ORB must be the last object to be destroyed! Besides this technical problem, it is generally a bad programming style to have global object references.

Chapter 9

OMNIBROKER communication

The following information only refers to the communication semantics of OMNIBROKER for C++. OMNIBROKER for Java, which uses a thread model instead of a “Reactor” (see 9.2 on page 90), has different communication semantics.

9.1 Method invocation semantics

The following outlines OMNIBROKER’s invocation semantics for the various CORBA method invocation types. Please note that the CORBA specification [1]) is silent about the exact invocation semantics.¹ This implies that the following information is OMNIBROKER-specific.

9.1.1 Invocation by an interface stub

Regular method invocation

A regular method invocation (i.e. a non-oneway method invocation) will block and return only if:

1. A reply is received from the server. (This is also true for method invocations that do not return any value or exception.)
2. A communication failure is detected. In this case an exception, usually of type `CORBA_COMM_FAILURE`, is raised.

¹For example, all the specification says about oneway method invocations is that oneway calls don’t return values and can’t throw user-defined exceptions. The precise semantic is implementation-specific.

3. A timeout was specified (see 9.1.3 on page 89) and the blocking time exceeded this timeout.

Note that OMNIBROKER allows you to specify that the client must listen to incoming requests while blocking for completion of a method invocation. To enable this behavior, nested method invocations must be enabled (see 9.1.4 on page 90).

Oneway method invocation

When a oneway method is invoked, OMNIBROKER tries to transmit the request from the client to the server immediately. However, the oneway call will not block if it cannot be carried out without a delay. Such a delay can be caused by network flow control or a busy server. In case the oneway method cannot be transmitted immediately, OMNIBROKER doesn't simply throw away that invocation but puts it into a buffer. OMNIBROKER tries to clear this buffer (i.e. to transmit the oneway invocation), if:

1. The main event loop is entered, i.e. `impl_is_ready()` is called.
2. Another method invocation is carried out on the same server.
3. Nested method invocations are enabled (see 9.1.4 on page 90) and another method invocation is carried out on any server.

Note that in case a oneway invocation would block and is therefore put into the buffer for later transmission, exceptions due to communication failures are also buffered. If, for example, `impl_is_ready()` detects an communication failure while trying to transmit a buffered oneway method invocation, the corresponding exception is not raised immediately. It is buffered instead and will be raised when the client tries to carry out another method invocation on the same server. This provides for synchronous exceptions even though the operation that caused the exception was transmitted asynchronously.

9.1.2 Invocation by the DII

Invocation by `invoke()`

Using the DII method `invoke()` has the same semantics as regular (i.e. non-oneway) method invocation by an interface stub (see 9.1.1 on page 87). This means that `invoke()` blocks until a return value is available or an exception is raised (including possible timeout exceptions, see 9.1.3 on page 89).

Invocation by `send_oneway()`

The DII method `send_oneway()` behaves exactly like a oneway method invocation by an interface stub (see 9.1.1 on page 88). This means that `send_oneway()` never blocks.

Invocation by `send_deferred()`

`send_deferred()` is like a split-up `invoke()`, similar in behavior to `oneway()`. To be more precise, `send_deferred()` initiates a request that will never block. However, methods invoked with `send_deferred()` will be acknowledged by a reply from the server, just like `invoke()`, and therefore can have a return value (which is not the case for `oneway()`). The difference is that you have to explicitly call either `get_response()` or `poll_response()` to get the reply, which can hold return values or an exception.

So what's the difference between `get_response()` and `poll_response()`? `get_response()` blocks until a response is received (or until a communication failure is detected, in which case an exception, usually of type `CORBA_COMM_FAILURE`, is raised). `poll_response()` is like a reversed `oneway()`. That means that `poll_response()` tries to get the outstanding response and returns `true` if successful. Otherwise it returns `false` immediately.

9.1.3 Using timeouts

Note: Timeouts are an OMNIBROKER-specific extension.

Timeouts can be set by using the `_timeout()` method of `Object` types. A value of `-1` means to use no timeout. a value `>= 0` specifies a timeout in units of *msec*. Explicit timeout values can be used with regular method invocation by an interface stub and the DII methods `invoke()` and `get_response()`.

With a timeout set, these methods raise an exception of type `CORBA_NO_RESPONSE` if there is any blocking period that exceed the specified timeout value. This implies that no *total* timeout for these methods can be specified. For example, if the transmission of a method invocation blocks three times for a period of *1sec* (for example because the data to be transmitted is too much to be sent at once), no exception is raised if a timeout of *1.5sec* is set, even though the total blocking time is *3sec*. Only the partial blocking times are taken into account. So in this example the timeout needs to be set to a value smaller than *1sec* for a timeout exception to be raised.

If a timeout exception is raised all buffered messages are lost. For example, if a timeout was set, and a method call exceeds that timeout and raises a timeout

exception, all oneway messages, that have not yet been transmitted and therefore have been buffered, are lost.

Note that oneway method invocations by an interface stub and the DII methods `send_oneway()`, `send_deferred` and `poll_response()` have an implicit zero timeout. That is, these functions will never block. However, because of the different semantics of these functions, no timeout exception is raised. Instead, oneway method invocations by an interface stub and the DII methods `send_oneway()` and `send_deferred()` buffer the requests if they cannot be sent without a delay. The DII method `poll_response()` returns `false` if the reply cannot be received without a delay.

9.1.4 Nested method invocations

Note: Nested method invocation semantics are an OMNIBROKER-specific extension.

By enabling nested method invocations, you can instruct OMNIBROKER to dispatch (i.e. to carry out) incoming requests during any blocking period caused by outgoing requests. This is especially useful for implementing an object-oriented design that involves “callbacks”, e.g. for implementing a “Model-View-Observer” pattern. Here “Model” calls an “update” operation on the “View”, which in return calls back the “get data” operation of the “Model” (for more information on the “Model-View-Observer” pattern and patterns in general, see [8]).

As a side effect, enabling nested method invocations will also allow OMNIBROKER to use any blocking time to send out buffered oneway or deferred invocations or to receive replies for buffered deferred invocations. If nested method invocations are not enabled, OMNIBROKER can only handle invocations (or receive replies) if these go to the same server (or come from the same server) that caused the blocking.

There are two methods for enabling nested method invocations. The first one is to use the option `-ORBnested` with the function `CORBA_ORB_init()`. The second one is to call the method `CORBA_ORB::_nested()` with `true` as its argument.

9.2 The Reactor

OMNIBROKER uses a “reactor” for event dispatching that is quite similar to the reactor patterns as described in [7]. Simply speaking, the reactor is an instance in OMNIBROKER where special objects — so-called event handlers — can register if they are interested in specific events. Such events can be network events, like an

event signaling that data are ready to be read from a network connection.

9.2.1 Available reactors

Currently there are three reactors supported by OMNIBROKER:

1. The standard “select” reactor which relies on the Berkeley Sockets `select()` function.
2. A special reactor for use with the X11 Window System. This reactor handles X11 events (which for example can trigger X11 callbacks) and CORBA network events simultaneously.
3. A special reactor for use with Microsoft Windows 95 or Windows NT. This reactor handles Windows messages and CORBA network events simultaneously.

The “default” reactor is the “select” reactor. If one of the other reactors is to be used, it must be initialized explicitly.

The X11 reactor

An application that wants to use the X11 reactor simply has to call the function `OBX11Init()` *before* the ORB is initialized with `CORBA_ORB_init()`. For example:

```
#include <X11/Intrinsic.h>

#include <OB/CORBA.h>
#include <OB/X11.h>

int
main(int argc, char* argv[], char*[])
{
    XtAppContext appContext;
    Widget topLevel = XtAppInitialize(&appContext,
                                     "MyApplication",
                                     0, 0,
                                     &argc, argv,
                                     0, 0, 0);

    OBX11Init(appContext);
}
```

```

CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

...
}

```

The Windows reactor

For the Windows reactor, the function `OBWindowsInit()` must be called, also *before* the ORB is initialized. For example:

```

#include <Windows.h>

#include <OB/CORBA.h>
#include <OB/Windows.h>

int WINAPI
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpszArgs, int nWinMode)
{
    OBWindowsInit(hInstance);

    int dummy = 0;
    CORBA_ORB_var orb = CORBA_ORB_init(dummy, 0);
    CORBA_BOA_var boa = orb -> BOA_init(dummy, 0);

    ...
}

```

9.2.2 Writing a custom event handler

OMNIBROKER comes with support for customized event handlers. This means that while your application is running, it can react to events like keyboard events. In order to implement your own OMNIBROKER event handler you must derive a class from `OBEventHandler` and overload the `handleEvent()` and `handleStop()` member functions. The constructor of the derived class must ensure that objects of this class are registered with the reactor. This is an example for an event handler that listens to keyboard events:

```

#include <OB/Reactor.h>

```

```
class MyEventHandler : public OBEventHandler
{
public:

    MyEventHandler();
    virtual ~MyEventHandler();

    //
    // Event handling
    //
    virtual void handleEvent(CORBA_ULong);
    virtual void handleStop();
};

MyEventHandler::MyEventHandler()
{
    //
    // Register with the reactor
    //
    OBReactor* reactor = OBReactor::instance();
    reactor -> registerHandler(this, OBEventRead, 0);
}

MyEventHandler::~~MyEventHandler()
{
    //
    // Unregister with the reactor
    //
    OBReactor* reactor = OBReactor::instance();
    reactor -> unregisterHandler(this);
}

void
MyEventHandler::handleEvent(OBMask mask)
{
    assert(mask == OBEventRead);

    char c;
    cin.read(&c, 1);
}
```

```
        //
        // Handle character input here ...
        //
    }

void
MyEventHandler::handleStop()
{
    // Do nothing here
}
```

OMNIBROKER calls the `handleEvent()` function each time a read event from standard input is pending.

Appendix A

Exceptions

A.1 Standard exceptions

These are the standard CORBA exceptions listed in [1]. For a more detailed description, see 4.7.1 on page 34.

CORBA_UNKNOWN	Unknown exception type
CORBA_BAD_PARAM	An invalid parameter was passed
CORBA_NO_MEMORY	Failure to allocate dynamic memory
CORBA_IMP_LIMIT	Implementation limit was violated
CORBA_COMM_FAILURE	Communication failure
CORBA_INV_OBJREF	Invalid object reference
CORBA_NO_PERMISSION	The attempted operation was not permitted
CORBA_INTERNAL	Internal error in ORB
CORBA_MARSHAL	Error marshalling a parameter or result
CORBA_INITIALIZE	Failure when initializing ORB
CORBA_NO_IMPLEMENT	Operation implementation unavailable
CORBA_BAD_TYPECODE	Bad typecode
CORBA_BAD_OPERATION	Invalid operation
CORBA_NO_RESOURCES	Insufficient resources for a request
CORBA_NO_RESPONSE	Response to a request is not yet available
CORBA_PERSIST_STORE	Persistent storage failure
CORBA_BAD_INV_ORDER	Routine invocation out of order
CORBA_TRANSIENT	Transient failure, request can be reissued
CORBA_FREE_MEM	Cannot free memory
CORBA_INV_IDENT	Invalid identifier syntax
CORBA_INV_FLAG	Invalid flag was specified
CORBA_INTF_REPOS	Error accessing interface repository

CORBA_BAD_CONTEXT	Error processing context object
CORBA_OBJ_ADAPTER	Failure detected by object adapter
CORBA_DATA_CONVERSION	Error in data conversion
CORBA_OBJECT_NOT_EXIST	Non-existent object, references should be deleted

A.2 Minor exception codes

According to the CORBA specification the standard exceptions can be categorized by minor exception codes. These codes are helpful if you need specific information about a particular exception. Note that minor exception codes may have different meanings for ORBs from different vendors.

A.2.1 Minor exception codes for CORBA_COMM_FAILURE

OBMinorRecv	recv() failed
OBMinorSend	send() failed
OBMinorRecvZero	recv() returned zero
OBMinorSendZero	send() returned zero
OBMinorSocket	socket() failed
OBMinorSetsockopt	setsockopt() failed
OBMinorGetsockopt	getsockopt() failed
OBMinorBind	bind() failed
OBMinorListen	bind() failed
OBMinorConnect	connect() failed
OBMinorAccept	accept() failed
OBMinorSelect	select() failed
OBMinorGethostname	gethostname() failed
OBMinorGethostbyname	gethostbyname()
OBMinorWSAStartup	WSAStartup() failed
OBMinorWSACleanup	WSACleanup() failed
OBMinorNoGIOP	Not a GIOP message
OBMinorUnknownMessage	Unknown GIOP message
OBMinorWrongMessage	Wrong GIOP message
OBMinorCloseConnection	Got a close connection message
OBMinorMessageError	Got a message error message

A.2.2 Minor exception codes for CORBA_INTF_REPOS

OBMinorNoIntfRepos	Interface repository is not available
--------------------	---------------------------------------

<code>OBMinorIdExists</code>	Repository id already exists
<code>OBMinorNameExists</code>	Name already exists
<code>OBMinorRepositoryDestroy</code>	<code>destroy()</code> invoked on <code>Repository</code> object
<code>OBMinorPrimitiveDefDestroy</code>	<code>destroy()</code> invoked on <code>PrimitiveDef</code> object
<code>OBMinorAttrExists</code>	Attribute is already defined in a base interface
<code>OBMinorOperExists</code>	Operation is already defined in a base interface
<code>OBMinorLookupAmbiguous</code>	Search name for <code>lookup()</code> is ambiguous
<code>OBMinorAttrAmbiguous</code>	Attribute name collisions in base interfaces
<code>OBMinorOperAmbiguous</code>	Operation name collisions in base interfaces

Appendix B

OMNIBROKER License

ROYALTY-FREE PUBLIC LICENSE AGREEMENT FOR OMNIBROKER SOFTWARE

IMPORTANT-READ CAREFULLY: This Object-Oriented Concepts, Inc. Royalty-Free Public License Agreement for OmniBroker Software ("License") is a legal agreement between you, the Licensee, (either an individual or a single entity) and Object-Oriented Concepts, Inc. for non-commercially using, copying distributing and modifying the Software and any work derived from the Software, as defined hereinbelow. Any commercial use is subject to a different license.

By modifying or distributing the Software or any work derived from the Software, Licensee indicates acceptance of this License, and agrees to be bound by all its terms and conditions for using, copying, distributing or modifying the Software and works derived from the Software.

No rights are granted to the Software except as expressly set forth herein. Nothing other than this License grants Licensee permission to use, copy, distribute or modify the Software or any work derived from the Software. Licensee may not use, copy, distribute or modify the Software or any work derived from the Software except as expressly provided under this License. If Licensee does not accept the terms and conditions of this License, do not use, copy, distribute or modify the Software.

In consideration for Licensee's forbearance of commercial use of the Software, Object-Oriented Concepts, Inc. grants Licensee non-exclusive, royalty-free rights as expressly provided herein.

DEFINITIONS. The "Software" is the OmniBroker software, including, but not limited to, the OmniBroker Libraries and Class Files, the OmniBroker IDL-to-C++ and IDL-to-Java translators, associated media and printed materials, and any included "on-line" documentation.

A "work derived from the Software" is any derivative work, as defined in 17 U.S.C. paragraph 101, which is derived from the Software, for example, code generated by the OmniBroker IDL-to-C++ or IDL-to-Java translators, a program which is linked with or otherwise incorporates the OmniBroker Libraries or Class Files, or a translation, improvement, enhancement, extension or other modification of the Software.

To "use" means to execute (i.e. run) the Software.

To "copy" means to create one or more copies as defined in 17 U.S.C. §101.

To "distribute" means to broadcast, publish, transfer, post, upload, download or otherwise disseminate in any medium to any third party.

To "modify" means to create a work derived from the Software.

A "commercial use" is any copying, distribution or modification of the Software or any work derived from the Software to any party where payment or other consideration is made in connection with such copying, distribution or modification, whether directly (as in payment for a copy of the Software) or indirectly (including but not limited to payment for some good or service related to the Software, or payment for some product or service that includes a copy of the Software "without charge"). However, the following actions which involve payment do not in and of themselves constitute a commercial use:

- (a) posting the Software on a public access information storage and retrieval service for which a fee is received for retrieving information (such as an on-line service), provided that the fee is not content-dependent. Such fees which are not content dependent include, but are not limited to, fees which are based solely on the storage capacity required to store the information, and fees which are based solely on the time required to transfer the information from/to the public access information storage and retrieval service; and
- (b) distributing the Software on a CD-ROM, provided that the Software is reproduced entirely and verbatim on such CD-ROM, and provided further that all information on such CD-ROM may be distributed in a manner which does not constitute a commercial use.

GRANT OF LICENSE.

LICENSE TO USE. Licensee may use the Software.

LICENSE TO COPY AND DISTRIBUTE. Licensee may copy and distribute literal (i.e., verbatim) copies of the Software as Licensee receives it throughout the world, in any medium, provided that Licensee distributes an unmodified, easily-readable copy of this License with the Software, and provided further that such distribution does not constitute a commercial use.

LICENSE TO CREATE WORKS DERIVED FROM THE SOFTWARE. Licensee may create works derived from the Software, provided that any such work derived from the Software carries prominent notices stating both the manner in which Licensee has created a work derived from the Software (for example, notices stating that the work derived from the Software is linked with or otherwise incorporates the OmniBroker Libraries or Class Files or code generated by the OmniBroker IDL-to-C++ or IDL-to-Java translators, or notices stating that the work derived from the Software is an enhancement to the Software which Licensee has created) and the date any such work derived from the Software was created.

LICENSE TO COPY AND DISTRIBUTE WORKS DERIVED FROM THE SOFTWARE. Licensee may copy and distribute works derived from the Software throughout the world, provided that Licensee distributes an unmodified, easily-readable copy of this License with such works derived from the Software, and provided further that such distribution does not constitute a commercial use. Licensee must cause any work derived from the Software that Licensee distributes to be licensed as a whole and at no charge to all third parties under the terms of this License.

Any work derived from the Software must be accompanied by the complete corresponding machine-readable source code of such work derived from the Software, delivered on a medium customarily used for software interchange. The source code for the work derived from the Software means the preferred form of the work derived from the Software for making modifications to it. For an executable work derived from the Software, complete source code means all of the source code for all modules of the work derived from the Software, all associated interface definition files and all scripts used to control compilation and installation of all or any part of the work derived from the Software. However, the source code delivered need not include anything that is normally distributed, in either source code or binary (object-code) form, with major components (including but not limited to compilers, linkers and kernels) of the operating system on which the executable work derived from the Software runs, unless that component itself accompanies the executable code of the work derived from the Software;

Furthermore, if the executable code or object code of the work derived from the Software may be copied from a designated place, and if the source code of the work derived from the Software may be copied from the same place, then the work derived from the Software shall be construed as accompanied by the complete corresponding machine-readable source code of such work derived from the Software, even though third parties are not compelled to copy the source code along with the executable code or object code.

If the work derived from the Software normally reads commands interactively when run, Licensee must cause the work derived from the Software, at each time it commences operation, to print or display an announcement including an appropriate copyright notice and either a notice consisting of the verbatim warranty and liability provisions of this License, or a notice that Licensee, and not Object-Oriented Concepts, Inc., provides a warranty. Such notice must also state that users may distribute the Software and/or the work derived from the Software only under the conditions of this License, and must further state how to view

the copy of this License included with the work derived from the Software.

Licensee may not impose any further restrictions on the exercise of the rights granted herein by any recipient of any work derived from the Software.

RESTRICTIONS. Licensee acknowledges that the Software is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The Software is licensed, not sold. All title and copyrights in and to the Software, including but not limited to any images, photographs, databases, animations, video, text and "applets" incorporated into the Software, the accompanying printed materials, and any copies of the Software, are owned exclusively by Object-Oriented Concepts, Inc.

Licensee may not sublicense, assign or transfer this License, the Software or any work derived from the Software except as permitted by this License.

If Licensee distributes any written or printed material at all with the Software or any work derived from the Software, such material must include either (a) a written copy of this License, or (b) a prominent written indication that the Software or work derived from the Software is covered by this License, and also written instructions for printing and/or displaying the copy of this License which is provided on the distribution medium.

If using, copying, distributing and/or modifying the Software is restricted in certain countries for any reason, Object-Oriented Concepts, Inc. may in the future add an explicit geographical distribution limitation excluding those countries, so that using, copying, distributing and/or modifying is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

LICENSE TO WORKS DERIVED FROM THE SOFTWARE. Licensee hereby grants to Object-Oriented Concepts, Inc. a non-exclusive, non-transferable, royalty-free right to use, copy, distribute and modify, with the right to sublicense at any tier, any and all works derived from the Software that Licensee creates, provided such works derived from the Software are distributed to Object-Oriented Concepts, Inc. by Licensee, and further provided that, if such works derived from the Software comprise either code generated by the OmniBroker IDL-to-C++ or IDL-to-Java translators or a program which is linked with or otherwise incorporates the OmniBroker Libraries or Class Files, such works derived from the Software would constitute works derived from the Software independent of comprising code generated by the OmniBroker IDL-to-C++ or IDL-to-Java translators or a program which is linked with or otherwise incorporates the OmniBroker Libraries or Class Files, for example, a "bug fix" of the Software.

LIMITED WARRANTY. NO WARRANTIES.

OBJECT-ORIENTED CONCEPTS, INC. EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFTWARE. THE SOFTWARE IS PROVIDED TO LICENSEE "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES

OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. THE ENTIRE RISK AS TO THE USE, QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH LICENSEE. SHOULD THE SOFTWARE PROVE DEFECTIVE, LICENSEE ASSUMES THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

NO LIABILITY FOR GENERAL, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL OBJECT-ORIENTED CONCEPTS, INC., OR ANY OTHER PARTY WHO MAY COPY, DISTRIBUTE OR MODIFY THE SOFTWARE AS PERMITTED HEREIN, BE LIABLE FOR ANY GENERAL, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, INACCURATE INFORMATION, LOSS OF INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE, EVEN IF OBJECT-ORIENTED CONCEPTS, INC. OR SUCH OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT RESTRICTED RIGHTS. The Software is provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights 48 C.F.R. paragraph 52.227-19, as applicable. Manufacturer is Object-Oriented Concepts, Inc./44 Manning Road/Billerica, MA 01821

TERMINATION. Any violation or any attempt to violate any of the terms and conditions of this License will automatically terminate Licensee's rights under this License. Licensee further agrees upon such termination to cease any and all using, copying, distributing and modifying of the Software and any work derived from the Software, and further to destroy any and all of Licensee's copies of the Software and any work derived from the Software.

However, parties who have received copies of the Software or copies of any work derived from the Software, or rights, from Licensee under this License will not have their licenses terminated so long as such parties remain in full compliance with this License.

LICENSE SCOPE AND MODIFICATION. This License sets forth the entire agreement between Licensee and Object-Oriented Concepts, Inc., and supersedes all prior agreements and understandings between the parties relating to the subject matter hereof. None of the terms of this License may be waived or modified except as expressly agreed in writing by both Licensee and Object-Oriented Concepts, Inc.

SEVERABILITY. Should any provision of this License be declared void or unenforceable, the validity of the remaining provisions shall not be affected thereby.

GOVERNING LAWS. This License is governed by the laws of the State of Massachusetts, U.S.A., and shall be interpreted in accordance with and governed by the laws thereof. Licensee hereby waives any and all right to assert a defense based on jurisdiction and venue for any action stemming from this License brought in U.S. District Court for the District of Massachusetts.

Should Licensee have any questions concerning this License, or if Licensee desires to contact Object-Oriented Concepts, Inc. for any reason, please contact Object-Oriented Concepts, Inc. at:

Object-Oriented Concepts, Inc.
44 Manning Road
Billerica, MA 01821

Bibliography

- [1] *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, OMG Document 97-02-25.¹
- [2] *IDL/Java Language Mapping*, OMG Document 97-03-01.²
- [3] *ORB Portability Joint Submission*, OMG Document 97-04-14.³
- [4] Jon Siegel, *CORBA Fundamentals and Programming*, John Wiley & Sons, Inc.
- [5] Thomas J. Mowbray, Ron Zahavi, *The Essential CORBA*, John Wiley & Sons, Inc.
- [6] Robert Orfali, Dan Harkey, Jeri Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc.
- [7] Douglas C. Schmidt, *Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching in Pattern Languages of Program Design*, Addison-Wesley.
- [8] Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *A System of Patterns*, John Wiley & Sons, Inc.

¹ You can get [1] from <ftp://ftp.omg.org/pub/docs/formal/97-02-25.pdf>.

² You can get [2] from <ftp://ftp.omg.org/pub/docs/orbos/97-03-01.pdf>.

³ You can get [3] from <ftp://ftp.omg.org/pub/docs/orbos/97-04-14.pdf>.