# CORBA Trader Service Guide, C++

Version 6.2, December 2004

Making Software Work Together™

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, Orbix Mainframe, Orbix Connect, Artix, Artix Mainframe, Artix Mainframe Developer, Mobile Orchestrator, Orbix/E, Orbacus, Enterprise Integrator, Adaptive Runtime Technology, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

---

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this book. This publication and features described herein are subject to change without notice.

Copyright © 2001–2004 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

Updated: 12-Oct-2004

M 3 2 2 0

# Contents

**List of Figures** vii

**List of Tables** ix

**Preface** xi

**Chapter 1 An Introduction to the CORBA Trading Service** 1

- **Introduction** 2
- **Service Types** 4
- **Service Offers** 8
- **The Trader Service's Components** 9

**Chapter 2 Configuring the Trader Service** 11

- **Configuring and Running the Trader Service** 12
  - Steps 1-2: Determine the hosts and ports to be used in the deployment 13
  - Step 3: Enter the host and port information in the configuration 14
  - Step 4: Configure the trader service to run in replicated or non-replicated mode 16
  - Step 5: Run the service in prepare mode to obtain initial references 17
  - Step 6: Adding the initial references to the configuration 18
  - Step 7: Running the trader service 19
- **Additional Configuration Information** 20

**Chapter 3 Getting Started with the Trader Service** 21

- **Starting the Trader Service** 22
- **The Printer Application** 23
- **Trader Service Programming** 25
  - Connecting to the Trader 26
  - Adding a New Service Offer Type 27
  - Exporting a Service Offer 29
  - Querying for a Service Offer 31

**Chapter 4 Querying for Service Offers** 33

How the Trader Service Processes a Query 34
A Basic Query for Service Offers 36
Selecting a Service from Query Results 38
Forming Constraints for Queries 40
Setting Preferences to Sort Service Offers 43
Refining the Properties a Query Returns 45

Chapter 5 Understanding Trader Service Policies 47
Overview 48
Policies that Affect Queries 49
Policies that Affect Trader Functionality 52
Using Policies in a Query 53
Setting a Trader's Global Policies 56

Chapter 6 Exporting and Managing Service Offers 59
Overview 60
Initializing Service Offer Properties 61
Exporting a Service Offer to Trader 63
Getting Service Offer Data from Trader 65
Modifying a Service Offer 66
Withdrawing a Service Offer from Trader 68

Chapter 7 Programming Topics 69
Managing the Service Type Repository 70
Using Dynamic Property Values 73
Managing Links Between Traders 76

Chapter 8 Trader Service Console 81
Starting the Trader Console 83
Main Window 84
The Trader Console Menus 86
Managing Service Types 89
Managing Offers 93
Managing Proxy Offers 98
Managing Links 100
Configuring the Trader Attributes 102
Support Attributes 103
Import Attributes 104

Link Attributes 106
Admin Attributes 107
Executing Queries 108
Connecting to a New Trader 111

Appendix A The OMG Constraint Language 113
Introduction 114
Language Basics 115
The Constraint Language BNF 118

Glossary 121

Index 127

# List of Figures

Figure 1: Typical trading service process 2
Figure 2: Property Mode Strengths 6
Figure 3: Typical Interactions with the trader 23
Figure 4: How Query Parameters Affect Offers Gathered 34
Figure 5: The Trader Console main window 84

# List of Tables

Table 1: Kinds of traders and their components 9
Table 2: Query policies and Trader Service policies 50
Table 3: Policies You Can Set for a Query 55
Table 4: Trader policies 56

# Preface

CORBA Trader Service is a Java implementation of the Object Management Group (OMG) Trading Service. The CORBA Trader Service provides facilities for object location and discovery. Unlike the CORBA Naming Service where an object is located by name, an object in the Trading Service does not have a name. Rather, a server advertises an object in the Trading Service based on the kind of service provided by the object. A client locates objects of interest by asking the Trading Service to find all objects that provide a particular service. The client can further restrict the search to select only those objects with particular characteristics.

The Trader Service is compliant with the OMG CORBAservices: Common Object Services Specification (ftp://www.omg.org/pub/docs/formal/98-12-09.pdf) and conforms to the specification's definition of a *full-service trader*, meaning that the service supports all of the functionality described in the specification.

**Audience**

This manual is aimed at users wanting to create a trader service for use by their applications.

**Related documentation**

The document set for Orbix includes the following:

- *CORBA Programmer's Guide*
- *Administrator's Guide*
- *CORBA Programmer's Reference*

The latest updates to the Orbix documentation can be found at http://www.iona.com/support/docs.

**Additional resources**

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to docs-support@iona.com .

**Typographical conventions**

This guide uses the following typographical conventions:

`Constant width`

Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

*Italic*

Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

`% cd /users/`*your_name*

**Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

**Keying conventions**

This guide may use the following keying conventions:

No prompt

When a command's format is the same for multiple platforms, a prompt is not used.

| | |
|---|---|
| `%` | A percent sign represents the UNIX command shell prompt for a command that does not require root privileges. |
| `#` | A number sign represents the UNIX command shell prompt for a command that requires root privileges. |
| `>` | The notation > represents the DOS or Windows command prompt. |
| `...`<br>`.`<br>`.`<br>`.` | Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion. |
| `[ ]` | Brackets enclose optional items in format and syntax descriptions. |
| `{ }` | Braces enclose a list from which you must choose an item in format and syntax descriptions. |
| `\|` | A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions. |

# An Introduction to the CORBA Trading Service

*The Trader Service is a full implementation of the CORBA Trading Object Service. With this service, servers can offer functionality by making a number of objects publicly available. Clients can then get references to objects that match a specified functionality.*

**In this chapter**

The following topics are discussed in this chapter:


| | |
|---|---|
| Introduction | page 2 |
| Service Types | page 4 |
| Service Offers | page 8 |
| The Trader Service's Components | page 9 |

# Introduction

**Overview**

The CORBA Trader Service is a Trading Object Service that allows an object to be registered with a description of its functionality. This service greatly increases the scalability of distributed systems by making services easier to locate. An example of a service that a client might search for is a printer.

**How clients and servers use a trader**

A trader contains a number of *service types* that describe a service. For example, a printer service type might have properties such as `pages_per_minute` (a long) and `location` (a string). Service types are stored in a *Service Type Repository*. *Service offers*, or *offers,* are instances of these service types.

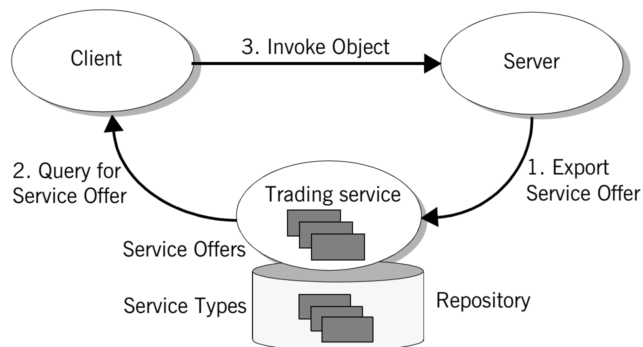



**Figure 1:** *Typical trading service process*

A server can export an offer to the trader, which includes an object reference for one of its objects and values for properties defined by the service type, for example, “50 pages per minute, located on the first floor”.

A client can then query the trading service based on these properties using a filter called a *constraint*. For example, a client could search for a printer where “pages_per_minute > 200". The trader then returns to the client an offer of a service. The client can then use the object reference in the offer to invoke on the server.

### Scalability

The trader can be a tool for constructing efficient distributed applications. The advantage of annotating a service offer with properties, and allowing offers to be filtered on the basis of those properties using a constraint, is that clients can select offers without having to incur the overhead of invoking operations on each object.

For example, suppose that `Printer2Interface`, which is a subclass of `PrinterInterface`, has an additional operation, `cost()`, which returned a value of type `float`:

```
//IDL
interface Printer2Interface : PrinterInterface {
  void page_counter();
  float cost();
};
```

In this situation, if the importer needed to select only those printers whose cost is within a certain range, the importer would need to iterate over each printer returned by the trading service to invoke the `cost()` operation. In a distributed environment, the overhead of this activity could be prohibitively expensive. It is the developer's responsibility to anticipate the types of queries that importers will need to perform and design their service types accordingly.

# Service Types

**Service type definition**

*Service types* are general descriptions of a kind of service. They consist of the following:

- A *type name* (for example, *printer)* uniquely identifies the service type.
- An *interface type* defines the IDL interface to which an advertised object of this type must conform (for example, `"IDL:MyAppModule/MyAppInterface:1.0"`*).*
- A collection of *property types* defines additional attributes of the service offer (for example, *"long page_per_min"*, *"string location"*).

**Service type names**

Each service type in the repository has a unique name. Orbix Trader supports two name formats:

- **Scoped names** - These names have formats such as `::One::Two`. Other supported variations include `Three::Four` and simply `Five`.
- **Interface repository identifiers** - These names adhere to the format of interface repository identifiers. The most common format is

```
IDL:[prefix/][Module/]Interface:X.Y
```

**Note:** Although both naming formats follow interface repository conventions, service type names are never used to look up information in the interface repository.

**Interface types**

An *interface type* describes the IDL signature of the advertised service. The interface type is a string whose format should be a scoped name or an interface repository identifier as described above for service type names. When a new service is exported, the trader may use the interface repository to confirm that the object being advertised *conforms* to the interface defined by the interface type. An object conforms to an interface if it implements that interface, or if it implements a subclass of that interface.

**Property types**

A service type can have zero or more *property types*, representing additional information that can be associated with an advertised service.

A property type definition consists of a name, a value type and a mode. The value type is a `CORBA::TypeCode`, and the mode indicates whether a property is mandatory and whether it is read-only.

The property modes have the following semantics:

- Mandatory—The exporter must provide a value for the property at the time the service is exported. Mandatory properties cannot be removed.
- Read-only—Once an exporter has supplied a value for the property, it cannot be modified. Read-only properties can be removed.
- Mandatory and Read-only—The property must have a value when the service is exported, and cannot subsequently be changed or removed.

A property that is neither mandatory nor read-only is considered optional, and can be changed and removed.

Orbix Trader accepts Java-style identifiers as property names, meaning a property name must start with a letter, and may consist of letters, numbers and underscores.

---

**Super types**

Service types can *inherit* from other service types, which enables the definition of *super types* that encapsulate behavior and characteristics common to many service types. When a new service type is created that has super types, the trader checks that several prerequisites are met:

1. All super types must already exist in the service type repository.
2. Any property type definitions in the new service type that have the same name as a definition in a super type must be compatible with the super type definition. For two property definitions to be compatible,

their value types must match, and the mode of the new definition must be the same as, or stronger than, the mode of the property in the super type according to the graph in Figure 2.

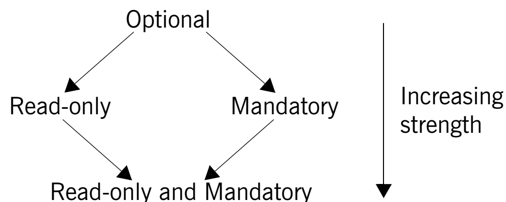



**Figure 2:** *Property Mode Strengths*

3. The interface type of the new service type must conform to the interface type of all super types. Orbix Trader may use the interface repository to verify that this is true.

For example, consider two IDL interfaces, `InterfaceA` and `InterfaceB`, defined below:

```
// IDL
interface InterfaceA {
  void do_something();
};

interface InterfaceB : InterfaceA {
  void do_something_else();
};
```

Here, `InterfaceB` inherits from `InterfaceA`. Now, let's define two service types:

```
service ServiceTypeA
{
  interface InterfaceA;
  property string name;
};

service ServiceTypeB : ServiceTypeA
{
  interface InterfaceB;
  mandatory property string name;
  readonly property float cost;
};
```

In the example above, `ServiceTypeB` inherits from `ServiceTypeA`. As such, it inherits all of the property types from `ServiceTypeA`, and declares an interface type of `InterfaceB`, which conforms to the interface type of its super type because `InterfaceB` is a subclass of `InterfaceA`.

Notice that `ServiceTypeB` redefines the mode of the "name" property. Whereas the definition in `ServiceTypeA` does not specify a mode (making the property optional), the definition in `ServiceTypeB` makes this property mandatory, therefore a value for the property must be supplied when the offer is exported. The reverse is not allowed; a subtype cannot redefine a mandatory property to be optional.

`ServiceTypeB` also adds a new property, "cost", which is defined to be read-only. Because the property is not mandatory, an exporter does not need to supply a value for it at the time a service offer is exported. However, once a value has been defined for this property, it cannot subsequently be changed.

# Service Offers

**Service offers**

A *service offer* is an instance of a service type and represents the advertisement of a service by a service provider.

A service offer has the following characteristics:

- A *service type name* associates the offer with a particular service type.
- An *object reference* provides the "pointer" (the object reference) to the advertised object that is necessary for clients to invoke the service being offered.
- A set of *properties* describe this service offer and must conform to the property types defined by the service type.

The trader uses the definition of the specified service type to perform several validation steps on a new offer:

1. The exporter must provide values for all mandatory properties (including all mandatory properties that the service type inherits from its super types, if any).
2. The object must conform to the interface type defined by the service type. Orbix Trader may use the interface repository to verify that this is true.
3. The value types of all properties must match the value types as defined by the service type. For example, a value of type `double` is not allowed for a property whose type is defined as `string` in the service type.

> **Note:** Orbix Trader allows an exporter to supply values for named properties that are not defined in the service type.

The value of a property in a service offer can be modified if the mode of the property is not read-only. A property can be removed from a service offer if the property is not mandatory. New properties can also be added to an existing service offer.

# The Trader Service's Components

**Trader components**

The Trader Service functionality is divided into components where each component has an associated interface as follows:

- Lookup
- Register
- Admin
- Link
- Proxy

The CORBA Trader Service is a full-service implementation of the OMG's Trading Object Service specification. The following table summarizes the different kinds of traders and the component functionality offered:

**Table 1:** *Kinds of traders and their components*

| Kind of Trader | Component Interfaces | | | | |
|---|---|---|---|---|---|
| | Lookup | Register | Admin | Link | Proxy |
| Full-Service | **CORBA Trader Service** | | | | |
| Linked | X | X | X | X | |
| Proxy | X | X | X | | X |
| Stand-alone | X | X | X | | |
| Simple | X | X | | | |
| Query | X | | | | |

The functionality of each kind of trader depends on the interfaces that it supports. The following is a list of the kinds of traders specified by the OMG:

- The simplest trader is the *Query trader*, which just supports the `Lookup` interface. This could be useful, for example, where a trader is pre-loaded and optimized for searching.
- The *simple trader* supports not only the `Lookup` interface but it also supports exporting of offers with the `Register` interface.

- The *stand-alone trader* supports the interfaces of a simple trader and additionally supports administration of the trader's configuration settings using the `Admin` interface.
- The *proxy trader* supports the interfaces of a stand-alone trader and additionally supports the `Proxy` interface. The proxy trader essentially exports a *lookup* interface for delayed evaluation of offers, and can be used for encapsulating legacy applications, or as a kind of service offer factory.
- The *linked trader* supports the interfaces of a stand-alone trader and additionally supports federation of traders using the `Link` interface.
- The *full-service trader* combines the functionality of all component interfaces. The Orbix CORBA Trader Service is a full-service trader.

CHAPTER 2

# Configuring the Trader Service

*This chapter provides a description of the steps necessary to configure the Trader Service.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Configuring and Running the Trader Service | page 12 |
| Additional Configuration Information | page 20 |

# Configuring and Running the Trader Service

**Introduction**

These instructions describe how to configure the Trader Service.

**Preparatory steps**

Several preparatory steps are necessary to configure and run the trader service. The specific actions taken at each step are somewhat different depending on whether you want to run the service replicated or non-replicated.

The general sequence of actions are as follows:

1. Determine on which hosts you want to run the master trader service and on which hosts any slaves will run.
2. Determine the port number on which the master, slaves, and Replicators will listen.
3. Enter the host and port number information into the configuration.
4. Configure the trader service to run in replicated or non-replicated mode.
5. Run the trader service in “prepare” mode to obtain initial references needed to enable clients to interact with the service.
6. Add each of the references obtained during step 4 to the configuration database.
7. Start the master trader service and any slaves.

**Explanation**

In the following explanation of the steps listed above, example settings are given assuming a deployment of one master trader service instance running on host “master”, and one slave trader service instance running on host “slave”. In addition, it will be pointed out where steps should be modified or bypassed in order to run a single non-replicated instance of the service.

# Steps 1-2: Determine the hosts and ports to be used in the deployment

These steps are completely deployment-specific. Depending on the number of trader service instances you want to deploy, you will need to select 1 or more distinct host/port pairs for each instance of the service to use as a communication end-point. In our example, we use a replicated service with one master and one slave. The master runs on host `master` and listens on port 15001; the slave runs on host `slave` and listens on port 15001. The master and slave need not listen on the same port number. Also, two or more replicas may run on the same host as long as they listen on different ports.

Furthermore, each trader service instance running in a replicated deployment scenario will also create a *Replicator* object. You must also select the ports on which each Replicator will listen. In the sample configuration, the Replicator always listens on port 15002.

# Step 3: Enter the host and port information in the configuration

The Trader Service configuration will contain variables set in a global scope (the outer scope not contained within a named block), and variables set in one or more named scopes. The global scope specifies configuration variable settings for all replicas in a replicated deployment, while the named scopes each specify configuration variable settings that apply to a specific trader service instance. The name of each scope corresponds to the ORB name that will be used when launching each instance of the service.

In the default `trader.cfg` included with the trader service package, there are two named scopes: one for ORB name `trading0`, and the other for ORB name `trading1`. All host/port information is set within a named configuration scope.

The host/port information within a given configuration scope is contained in the following variables:

```
trader:iiop:addr_list
replication:Replicator:iiop:addr_list
```

In the sample configuration, these variables are set as follows in the `trading0` scope:

```
trader:iiop:addr_list = ["master:15001", "+slave:15001"];
replication:Replicator:iiop:addr_list = ["master:15002"];
```

These settings indicate that the trader service instance using ORB name `trading0` will run on host `master` and listen on port 15001. The service will be replicated, with the one replica participating in the service running on host `slave` and listening on port 15001. The Replicator will listen on port 15002.

Note that if more replicas are being used in the deployment, an additional `"+<hostname>:<port>"` pair would be appended to the list for each replica. If running the service non-replicated, only a single `"<hostname>:<port>"` pair should be included in the `trader:iiop:addr_list`. Including additional pairs in the list will only increase the size of IORs used by the service, but this will result in unnecessary resource consumption when running non-replicated. In addition, in the non-replicated case, the second `addr_list` variable listed above need not be set.

In the sample configuration, these same variables are set as follows in the `trading1` configuration scope:

```
trader:iiop:addr_list = ["slave:15001", "+master:15001"];
replication:Replicator:iiop:addr_list = ["slave:15002"];
```

These settings indicate that the trader service instance run with ORB name `trading1` will run on host `slave` and listen on port 15001. The service will be replicated, and the one other replica will run on host `master` and also listen on port 15001. The Replicator used by this service instance will listen on port 15002.

# Step 4: Configure the trader service to run in replicated or non-replicated mode

Before running the trader service in "prepare" mode, you should decide if you want to run with replication enabled or disabled, and if replication is enabled how many replicas will be used.

Whether replication is enabled or disabled is controlled by the setting of the configuration variable `replication:enable`. This variable should be set to `"True"` to enable replication, and to `"False"` to disable replication.

If running with replication enabled, you must also indicate the number of replicas that will be used by setting the `replication:replica_count` to the appropriate value. This variable should be set to the total number of replicas including the master and any slaves. In the example scenario with one master trader service instance and one slave, this variable should be set to 2.

# Step 5: Run the service in prepare mode to obtain initial references

Now you are ready to run the service in prepare mode, and obtain the initial references necessary for clients to connect to the service. Note that when running a replicated service, each individual replica must be prepared. The command to run the trader service in prepare mode is:

```
asp/Version/bin/ittrader prepare [-publish_to_file <filename>]
```

If running with replication enabled, preparing each instance of the trader service will result in three IORs being sent to standard output:

- The IOR of the replicated trader service (which will be the same for all replicas)
- The non-replicated, per-instance trader service IOR
- The IOR of the per-trader service Replicator.

If running with replication disabled only the IOR of the prepared trader service instance will be output.

Save the values for use in step 5.

# Step 6: Adding the initial references to the configuration

The initial references of each trader service instance and each Replicator need to be added to the configuration.

If running one non-replicated instance of the service, the initial reference to the service returned by preparing the one instance should be set as the value of the following variable in the global configuration scope:

```
initial_references:TradingService:reference
```

If running with replication enabled, the IOR of the *replicated* trader service should be set as the value of the trader service initial reference in the global scope (the same variable as described above for the non-replicated case).

If replication is enabled, or if running multiple non-replicated instances of the service within the same domain, the trader service initial reference variable within each named scope must also be set. If replication is enabled, the value set for the following variable within each named scope should be the non-replicated, per-instance trader service IOR:

```
initial_references:TradingService:reference
```

In addition, the non-replicated IOR of each trader service instance, along with the Replicator IOR for each instance, should be added to the configuration as the values of the variables of the form:

```
replication:replica:<replica id>:TradingService:reference
replication:replica:<replica id>:Replicator:reference
```

In the current example, the IORs returned by preparing the master replica are set as the values of the following variables:

```
replication:replica:0:TradingService:reference
replication:replica:0:Replicator:reference
```

while the IORs returned by preparing the slave replica are set as the values of these variables:

```
replication:replica:1:TradingService:reference
replication:replica:1:TradingService:reference
```

## Step 7: Running the trader service

Run ittrader

# Additional Configuration Information

There are a couple of additional configuration settings to be aware of:

**trader:database:dir=”./traderdb0”;**

This variable should be modified to contain the pathname (absolute or relative to where the trader is launched from) of where the trader database will reside for each replica of the service.

**replication:replica_id = “0”;**

This is a numeric ID for the instance of the trader being configured in the current scope.

Each replica should have a unique `replica_id`. If a replica’s `replication:replica_id` is the same value as `replication:master` then it is the master replica.

**direct_persistence**

This variable specifies if the service runs using direct or indirect persistence. the default value is `FALSE`, meaning indirect persistence.

**iiop:port**

This variable specifies the port that the service listens on when running using direct persistence.

CHAPTER 3

# Getting Started with the Trader Service

*This chapter shows an example of a simple printer service to illustrate most of the common functionality in the Trader Service. A printer server makes a printer available for general use. Then, a client application asks the Trader Service for a suitable printer, and uses it to print a document.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Starting the Trader Service | page 22 |
| The Printer Application | page 23 |
| Trader Service Programming | page 25 |

# Starting the Trader Service

### Starting the Trader Service

To start the trader, enter the following command.

```
ittrader run
```

### Synopsis

```
ittrader [-launcher_help]
    [-ORBconfig_dir config_dir_value]
    [-ORBconfig_domains_dir config_domains_dir_value]
    [-ORBdomain_name domain_name_value]
    [-ORBproduct_dir product_dir_value]
    [-ORBlicense_file license_file]
    [-bg | -background]
    [-show_java_command]
    [-version]
    [run | prepare [publish_to_file = filename]
```

### Stopping the Trader Service

```
itadmin trd_admin stop
```

# The Printer Application

## Overview

The print server creates a Printer service type, and exports the descriptions of several printers to the trader. A client allows the user to execute queries and “print” files.

## Interaction with the trader

Figure 3 shows the typical interactions clients and servers have with the trader:

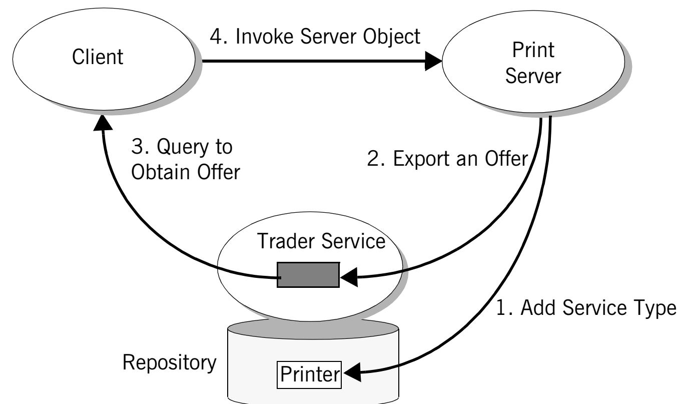



**Figure 3:** *Typical Interactions with the trader*

1. An offer server adds a service type (`Printer`) to the trader. The `Printer` type describes properties that office printers have, such as pages per minute. The service type names differ from the IDL interface names in this example, mainly to make their use clearer. For example, there could also be a `book_printer` service type that uses the `PrintServer` IDL interface, but it could have quite different properties such as options for hard or soft book binding.

2. The printer server creates a `printer_if` object. It exports and object reference to this object to the Trader Service as an *offer* of type `Printer`. It then waits for incoming requests, as normal.
3. The client process queries the Trader Service for a `Printer` offer.
4. The client process then uses the object reference in the offer obtained to invoke the printer server.

---

**The IDL specification**

The example application uses the following `PrintServer` IDL interface to describe the interface to a printer object:

```
// IDL
// This interface represents a print server that manages queues
// for several printers.
//
module TraderDemo
{
   interface PrintServer
   {
      typedef unsigned long JobID;

      // Add a file to a printer's queue.
      //
      JobID print (in string queue, in string file);
   };
};
```

# Trader Service Programming

This section outlines the three major programming steps used to interact with the trader. These steps are:

1. Add a service type using the offer server:
   i. Create a service offer type if a corresponding one doesn't already exist within the Trader Service. This example creates an `Printer` service offer type.
2. Register a service offer using the printer server:
   i. Create an object, for example, an instance of the IDL interface `PrintServer`.
   ii. Register the object reference with the Trader Service, within a service offer of type `Printer`. The server then accepts incoming object invocations as normal.
3. Get a service offer using the client:
   i. Query the Trader Service to get back a service offer.
   ii. Use the object reference specified in the service offer to invoke the object on the server.

Note that for simplicity, exception handling is omitted in the sample code.

# Connecting to the Trader

Servers need to connect to the trader to add a service offer type, for example, or to register a service offer. Clients need to connect to query the trader for service offers. The trader has a number of components represented by IDL interfaces including `Lookup`, `Register`, and others. The `"TradingService"` initial reference is a reference to the `CosTrading::Lookup` interface.

Do the following steps to get an object reference to the Trader Service:

```
1  CORBA::Object_var obj =
          orb->resolve_initial_references("TradingService");
2  CosTrading::Lookup_var trader;
   if (!CORBA::is_nil(obj))
     {
     trader = CosTrading::Lookup::_narrow(obj);
     }
```

1. Call `resolve_initial_references()` which returns a `CORBA::Object_ptr`.
2. Narrow the object reference.

# Adding a New Service Offer Type

An offer server inserts a service offer type called `printer` into the Trader Service. This is essentially a type declaration of an offer. Other servers may then use this type to register printer objects by creating instances of this type. Operations on service offer types are handled by the Service Offer Type Repository component of the Trader Service.

Do the following steps to add an offer type to the Offer Type Repository:

```
1  CosTrading::TypeRepository_var type_obj = trader->type_repos();
2  CosTradingRepos::ServiceTypeRepository_var trader_repos_obj =
      CosTradingRepos::ServiceTypeRepository::_narrow(type_obj);
3  CosTradingRepos::ServiceTypeRepository::PropStructSeq_var
     properties = new
      CosTradingRepos::ServiceTypeRepository::PropStructSeq;
   properties->length(3);

   properties[0].name = (const char *) "name";
   properties[0].value_type = CORBA::_tc_string;
   properties[0].mode =
      CosTradingRepos::ServiceTypeRepository::PROP_MANDATORY_READON
      LY;
   properties[1].name = (const char *) "location";
   properties[1].value_type = CORBA::_tc_string;
   properties[1].mode =
     CosTradingRepos::ServiceTypeRepository::PROP_MANDATORY;
   properties[2].name = (const char *)"page_per_min";
   properties[2].value_type = CORBA::_tc_long;
   properties[2].mode =
     CosTradingRepos::ServiceTypeRepository::PROP_NORMAL;

   // there are no super types for this service type
   CosTradingRepos::ServiceTypeRepository::ServiceTypeNameSeq_var
     default_supers = new
      CosTradingRepos::ServiceTypeRepository::ServiceTypeNameSeq;
4  trader_repos_obj->add_type (
       "Printer",                          // service type name
       "IDL:TraderDemo/PrintServer:1.0",  // idl type name
       properties,                         // property information
       default_supers                      // no super types
       );
```

The code is described as follows:

1. Get a reference to the Service Offer Type Repository.

2. The type `CosTrading::TypeRepository_var` is a `typedef` of `CORBA::Object`, and is essentially a forward reference. After obtaining a reference of this type, narrow it to `CosTradingRepos::ServiceTypeRepository`.
3. Construct the property information of a service offer type. In this example there are three properties: `name`, `location`, and `page_per_min`. The main parts of a service offer type include the following:
   - The name of the service type.
   - The IDL interface id for this service.
   - The properties which are a description of the offer. These are as follows:

```
enum PropertyMode {
      PROP_NORMAL, PROP_READONLY,
      PROP_MANDATORY, PROP_MANDATORY_READONLY
   };
   struct PropStruct {
      CosTrading::PropertyName name;
      TypeCode value_type;
      PropertyMode mode;
   };
   typedef sequence<PropStruct> PropStructSeq;
```

4. Invoke the `add_type()` function and pass it the relevant parameters.

# Exporting a Service Offer

When a server wants to make its service offers available, it registers with the Trader Service by exporting service offers. The code in Example 1 demonstrates the steps to export a service offer.

**Example 1:** *Exporting a service offer*

```
1   PrintServer_impl * impl = new PrintServer_impl();
    TraderDemo::PrintServer_var print_server = impl->_this();
    ...
2   CORBA::Object_var trader =
      orb->resolve_initial_references("TradingService");
    CosTrading::Lookup_var lookup =
      CosTrading::Lookup::_narrow(trader);
    CosTrading::Register_var register = lookup->register_if();

3   CosTrading::PropertySeq_var properties = new
        CosTrading::PropertySeq();
       properties->length(3);
       properties[0].name = (const char *) "name";
       properties[0].value <<= "laser4";
       properties[1].name = (const char *) "location";
       properties[1].value <<= "near coffee machine";
       properties[2].name = (const char *) "ppm";
       properties[2].value <<= (CORBA::Long) 50;

4   CosTrading::OfferId offer_id = register->export(
      print_server, // object reference to the CORBA object
      "Printer", // service type
      properties
    );
    // ... continued in next example
```

1. The printer server first creates an instance of the printer object.
2. The printer server connects to the Trader Service (as described in "Connecting to the Trader" on page 26) and gets a `trader_lookup_var`. It then uses this to access the Trader Service's register component, which handles exporting of service offers.

3. The server initializes the service offer properties with relevant values. The properties in service offers are name/any pairs, as follows:

```
typedef any PropertyValue;

struct Property {
   PropertyName name;
   PropertyValue value;
};
typedef sequence<Property> PropertySeq;
```

4. The server finally invokes the `export()` function to register the service offer.

# Querying for a Service Offer

Once offers have been exported to the trader service, clients can use the lookup interface to request services. Example 2 demonstrates a basic query that requests a printer that can print more than 5 pages per minute and uses the first offer returned by the trader.

**Example 2:** *Querying for a service offer*

```
// The Trader Service reference, trader, was aquired earlier

CosTrading::PolicySeq_var default_policies = new
  CosTrading::PolicySeq();

1   trader->query (
      "Printer",                    // service type name
      "ppm > 5",                    // constraint
      "random",                     // ordering of results
      default_policies,             // no special policies
      desired_properties,           // set to return all properties
      50,                           // max amount of offers wanted
      offers,                       // offers returned
      offer_itr,                    // remaining offers
      limits_applied                // applied internally by trader
   );
PrintServer_impl printer_obj;

2   if (offers.length() != 0)
  {
   printer_obj = PrintServer::_narrow (offers[0].reference);

3    printer_obj->print (doc, job_id);
  }
```

1. The client queries the Trader Service for a service offer matching certain criteria. In this example:
   - The constraint is that the offers returned have a `page_per_min` value that is greater than 5 pages per minute.
   - The results are returned in random order.
   - The default policies are used.
   - All properties are returned with the offer.

- A limit is set for the number of offers returned in the `offers` parameter. The trader will make find all of the possible matches, and return the remainder in the `iter` parameter.

2. The client selects a service offer from those returned in the query and invokes on the server. This simple example uses the first offer in the sequence.
3. The client uses the service offer to invoke on the object. In this case, the document is printed using the selected printer offer.
4. Any resources created by the trader for the iterator must be explicitly freed up.

# Querying for Service Offers

*In order for clients to find out about and use services offered by the Trader Service, the client code performs queries to obtain one or more service offers. A service offer contains, among other things, an object reference to a service. Clients then use the object reference to access a desired service.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| How the Trader Service Processes a Query | page 34 |
| A Basic Query for Service Offers | page 36 |
| Selecting a Service from Query Results | page 38 |
| Forming Constraints for Queries | page 40 |
| Setting Preferences to Sort Service Offers | page 43 |
| Refining the Properties a Query Returns | page 45 |

# How the Trader Service Processes a Query

**Overview**

It is easy to see how the set of offers that a trader contains can get quite large. In addition, traders can be linked together (federated) to search each other for service offers. This means that a query needs to have controls that complete a search in a reasonable amount of time. A query also needs controls that limit the amount of data returned.

**Format of a query**

A query starts with a service type name. A query then limits a search for appropriate offers by using a constraint on one or more properties of the service. You can also specify other limiting factors including the number of offers returned, a preference on the sort order, and the property values actually returned.

Figure 4 shows how the Trader Service uses these factors to process a query and generate a sequence of desired service offers.

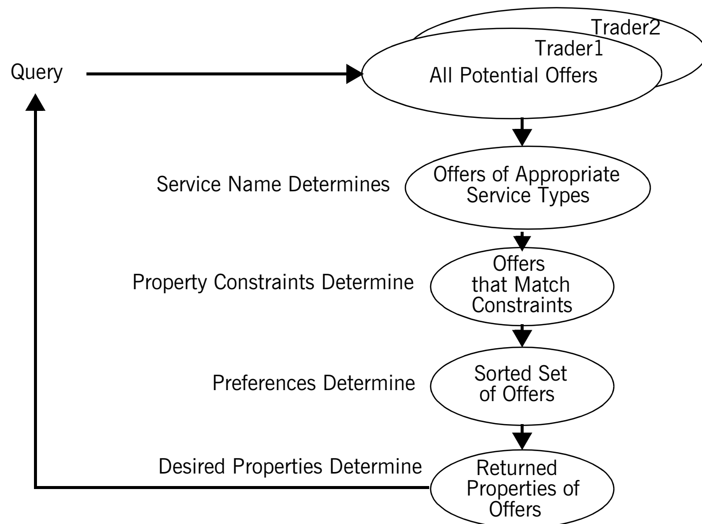



**Figure 4:** *How Query Parameters Affect Offers Gathered*

When the Trader Service processes a query, it gathers a sequence of offers together by narrowing down the set of all potential offers in all linked traders. The Trader Service uses query input to determine the following:

1. Uses the service name to determine if an offer is of an appropriate service type.
2. Uses the property constraints to determine if the offer matches the criteria specified by the client.
3. Uses preferences to determine the order in which to place the offer in the sequence of offers created.
4. Uses desired properties to determine which of the offer's property values (if any) are returned.

---

**Policies**

The Trader Service uses policies to control its behavior. For example, the maximum number of offers that can be searched for. You can also include one or more policies and values in a query to control the search behavior for a specific query.

# A Basic Query for Service Offers

**Connecting to the trader**

Clients need to connect to the Trader Service before they query for service offers. Do the following in your client to get an object reference to the trader:

```
// C++
CORBA::Object_var trader_obj trader_obj =
   orb->resolve_initial_references("TradingService");
CosTrading::Lookup_var lookup = CosTrading::Lookup::_narrow
   (trader_obj);
```

First, call `resolve_initial_references()` which returns a `CORBA::Object_ptr`. Then, narrow the object reference to a trader `Lookup` object.

**Querying the trader**

After connecting to the trader, clients can query the trader for service offers that match any desired criteria.

```
   // C++
   lookup -> query (
   // Query Input
1     "IDL:printer:1.0",                          // service type name
2     "(page_per_min > 5) and (page_type == A4)",  // constraint
3     "random",                                // preference sort order
4     default_policies,                           // policies desired
5     return_properties,                        // properties to return
6     10,                            // Initial number of offers wanted
   // Query Output
      offers,                                     // offers returned
      iterator,                                   // remaining offers
      limits_reached                     // Limits reached during query
   );
```

The input parameters to a query are explained in detail as follows:

1. The service type name parameter specifies the service type of the offers required. If the `exact_type_match` import policy is specified as true, only the service type is considered and no subtypes. If the `exact_type_match` policy is false or unspecified then subtypes are considered.

2. The constraint parameter specifies the constraint for restricting suitable offers. The constraint is a string that conforms to the OMG Constraint Language. Use an empty string if no constraints are required. See "Forming Constraints for Queries" on page 40 for more constraint examples.
3. A preference parameter specifies the order of the returned sequence of offers. You can sort offers by the following criteria:
    - In the order in which the Trader Service finds the offers. (This is the default.)
    - In descending order based on property values.
    - In ascending order based on property values.
    - All offers that meet a constraint first, followed by those offers that do not meet the constraint.
    - In random order.

    Use an empty string if no sort preference is required. See "Setting Preferences to Sort Service Offers" on page 43 for sort preference examples.
4. For now, default policies are used for the policies parameter. Policies are discussed in Chapter 5.
5. A return-properties parameter specifies the properties to return for the sequence of offers. You can choose to have none, some, or all properties returned. For example, if for your application it is adequate to use the first valid service offer, you can improve efficiency by returning no properties for the returned offers. See "Refining the Properties a Query Returns" on page 45 for an example of how to specify some properties to return.
6. The how-many-offers parameter specifies the number of offers to be initially returned via the `offers` out parameter. This example requests 10 initial offers.

The `offers` are returned as a sequence of offers. You can check for more offers and obtain them by using the `iterator` output parameter. If the Trader Service reached any policy limits during its search, the policy name is returned in the `limits_reached` output parameter. The `query()` output and how to use it is described in the next section.

# Selecting a Service from Query Results

**Overview**

The previous section described how the input parameters to the `query()` operation controls the offers you get. This section describes details of the output from `query()`.

**Output parameters**

The output parameters include a sequence of offers, an iterator object to obtain more offers, and a sequence of policy limits that the Trader Service may have encountered as it collected the offers.

```
//C++
lookup -> query (
// Query Input

// Query Output
   offers,              // offers returned
   iterator,            // remaining offers
   limits_reached       // Limits reached during query
);
```

**offers**

The `offers` parameter contains the returned sequence of offers. The client selects a service offer from those returned in the query and invokes on the desired server. The following example simply uses the first offer in the sequence:

```
// C++
PrintServer_impl printer_obj;
if (offers.length() != 0)
{
   printer_obj = printer_if::_narrow (offers[0].reference);
   printer_obj->print (doc, job_id);
}
```

**iterator**

The `iterator` parameter is an object reference to an `OfferIterator` interface. If all offers are returned in the `offers` parameter then the `iterator` parameter has a null reference value. However, recall that a query specifies the number of offers to be returned. If the number of offers

requested is lower than the number the Trader Service found, then an `OfferIterator` object reference is returned and the remaining offers can be retrieved via that object.

The following example shows how to peruse the remaining sequence of offers. In this example, the names of properties are printed:Once you are

```
//C++
char * value = 0; // for value of name property

// to tell when to stop
CORBA::Boolean more_offers = 1;
while (more_offers) {
  try {
    more_offers = iterator->next_n(2, offers);
    for (long i=0; i < offers->length(); i++) {
      (*offers)[i].properties[1].value >>= value;
      cout << value << endl;
    }
  }
  catch (...) {
    cerr << "printing names failed" << endl;
    cerr << "Unexpected Exception" << endl;
    exit(1);
  }
}
```

done with the iterator, you must use its `destroy()` function to release the resources it uses.

**limits_applied**

The `limits_applied` parameter is a sequence of policy names. If the Trader Service encounters any policy limits during a query, it returns the names of the policies in this sequence. For example, if a query generates more offers than the maximum number of offers the trader is allowed to search for, the name `max_search_card` is returned in the sequence. The values of the policies are not returned.

# Forming Constraints for Queries

**Using the constraint language**

This section describes how to use more features of the OMG constraint language to construct effective constraint expressions when querying for service offers. See "The OMG Constraint Language" on page 113 for a complete specification of the constraint language.

Although service properties can be defined using the great variety of IDL data types available, not all can be queried with the OMG constraint language. You can use the constraint language for properties defined with the following simple IDL data types:

```
boolean
short, unsigned short
long, unsigned long
float, double
char, Ichar
string, Istring
```

You can also use the constraint language for properties defined with sequences of the above data types.

**Evaluating property values**

A constraint contains a comparison of property values. The result of a comparison is a `boolean`. Thus, a potential offer is a match if the Trader Service evaluates the constraint as true.

### Comparison operators

Use the operators `==`, `!=`, `>`, `>=`, `<`, or `<=` to compare two of the same simple types. For example, the following constraint compares a float property with a float constant value:

```
float_property == 1.0
```

### Substring operator

Use the operator `~` to determine if the right operand is a substring of the left operand. The left operand is a property of type `string` or `Istring`, and the right operand is another string or string constant. For example:

```
string_property ~ 'String data'
```

String constants are delineated with apostrophes. To embed an apostrophe in a string, precede the apostrophe with a backslash (`\'`).

#### Sequence operators

Use the `in` operator to test if a value is in a sequence of values. The left operand must be a simple IDL type and the right operand must be a sequence of the same simple IDL type. For example:

```
'duplex' in output_options
```

### Combining expressions

Constraints can include combinations of expressions by using the keywords `and`, `or`, and `not`. For example, the following shows a constraint to obtain printers that produce output at a rate greater than 5 pages per minute and that support an A4 page type:

```
(page_per_min > 5) and (page_type == 'A4')
```

The following constraint is to obtain printers that do not produce output at a rate less than 5 pages per minute:

```
not (page_per_min < 5)
```

You can use parentheses to group expressions for clarity or to override the precedence relations of the constraint language.

### Testing for a property's existence

A constraint can test any service type property for its existence, even if the IDL data type used to define it is not a simple data type or sequence of a simple data type. Use the `exist` keyword to test whether a property exists for given offer:

```
exist page_per_min
```

Because properties with a mandatory mode must exist, it does not make sense to test for their existence. However, searching for the existence of optional properties can provide a powerful means of limiting the offers returned.

### Using arithmetic expressions

Constraints can include arithmetic expressions by using the standard operators `*/+-`. However, you can only use these operators between numbers and not between property names. For example:

```
page_per_min > 2 * 5
```

You can use float and double values where appropriate. Exponential notation is also valid.

# Setting Preferences to Sort Service Offers

**Creating a preference string**

When querying for service offers, you can set preferences to make the offers return in a particular order. Create a preference string using one of the following formats:

```
first
max numerical_expression
min numerical_expression
with constraint_expression
random
```

A preference string consists of a keyword and, in some cases, an expression. You cannot specify combinations of preferences by using more than one keyword in a single preference string.

**Constructing preference expressions**

Use the OMG constraint language to construct the preference expressions for `max`, `min`, and `with` formats. When you submit a query with one of these preference expressions, the Trader Service associates a sort value with each offer by evaluating the expression. The offers are then sorted with respect to the sort value and the type of preference as follows:

- A `max` preference sorts the offers in descending order from the maximum sort value evaluated.
- A `min` preference sorts the offers in ascending order from the minimum sort value evaluated.
- A `with` preference returns the offers that evaluate to true before the offers that evaluate to false.

If the Trader Service cannot evaluate the expression for a particular offer (for example, an expression that is based on an optional property may not evaluate), the offers are not discarded but are grouped after those offers that can be evaluated.

**Returning offers in the order of discovery**

The default behavior of the Trader Service is to return offers in the same order in which they were discovered. You can also specify this behavior by using the `first` preference.

**Returning offers in descending order**

Use the preference string format "`max` *numerical_expression*" to sort the returned service offers in descending order. For example:

```
max page_per_min
```

In this example, printers with the highest `page_per_min` value are returned first. The rest of the offers are returned in a descending order based on the sort value calculated in the numerical expression. Any offers that do not have a value for `page_per_min` are returned last.

**Returning offers in ascending order**

Use the preference string format "`min` *numerical_expression*" to sort the returned service offers in ascending order. For example:

```
min (jobs_in_queue)
```

In this example, printers with the lowest number of `jobs_in_queue` are returned first, followed in ascending order.

**Note:** The `max` and `min` preference formats do not constrain the offers returned to a maximum or minimum value. For example, the following is an incorrect expression that does not limit a sort to the offers with a minimum `page_per_min` value of 8:

`min (page_per_min == 8)` *This is an incorrect format*

**Returning offers by constraint**

Use the preference string format "`with` *constraint_expression*" to order the returned service offers based on a constraint expression. A constraint expression evaluates to either true or false. The offers with a constraint preference that evaluates to true precede those that evaluate to false. For example:

```
with (page_per_min > 10)
```

This example sorts the returned offers into two groups: the first group has pages per minute values greater than 10 and the second group has pages per minute values of less than or equal to 10.

**Returning offers in random order**

Use the `random` preference to make the Trader Service return offers in random order.

# Refining the Properties a Query Returns

**Specifying returned properties**

You can specify which properties you want returned in the sequence of offers. For example, if your application does not need to use all properties to determine which services to use, it can be more efficient for your memory and network traffic to return only those properties you need.

Example 3 shows how to specify the return of properties.

**Example 3:** *Specifying the return of properties*

```
// C++
1  CosTrading::Lookup::SpecifiedProps desired_properties;

2  CosTrading::PropertyNameSeq_var property_seq = new
       CosTrading::PropertyNameSeq();
   // two properties are specified
   property_seq->length(2);
   property_seq[0] = CORBA::string_dup("location");
   property_seq[1] = CORBA::string_dup("page_per_min");
3  desired_properties.prop_names(property_seq);

   CosTrading::PolicySeq_var default_policies = new
       CORBA::PolicySeq();

   lookup->query (
     "IDL:printer:1.0",
     "(page_per_min > 5) and (page_type == A4)",
     "random",
     default_policies,
4    desired_properties,                // properties to return
     10,
     offers,
     iterator,
     limits_reached
   );
```

The code is described as follows:

1. You first declare a `SpecifiedProps` union for properties.

To return all properties use the following code and go to step **4**:

```
// C++
CosTrading::Lookup::SpecifiedProps
   desired_properties(CosTrading::Lookup::all);
```

To return no properties use the following code and go to step **4**:

```
// C++
CosTrading::Lookup::SpecifiedProps
   desired_properties(CosTrading::Lookup::none);
```

2. If you want specific properties returned, create a property name sequence, `property_seq`. Make the sequence long enough to contain the names of all properties to be returned and fill it with the names of the desired properties.
3. Fill the `desired_properties` object with the list of properties to be returned.
4. Use the `desired_properties` object as a parameter in the `query()` function call.

CHAPTER 5

# Understanding Trader Service Policies

*Trader policies affect how the Trader Service works. Most policies control the scope of a search for offers. A few policies determine certain functionality that applies to the trader itself, including whether a trader supports modifiable properties, whether it supports dynamic properties, and whether it supports proxy offers.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Overview | page 48 |
| Policies that Affect Queries | page 49 |
| Policies that Affect Trader Functionality | page 52 |
| Using Policies in a Query | page 53 |
| Setting a Trader's Global Policies | page 56 |

# Overview

**What is a policy?**

A policy is a data structure containing a pre-defined policy name and a value for that policy. The value's data type depends on the particular policy. For example, the `supports_modifiable_properties` policy can have a `boolean` value. A value of 0 means that for the particular trader, properties of service offers cannot be changed after an offer is exported to the trader. A value of 1 means that the trader allows changes to its service offer properties.

# Policies that Affect Queries

**Query semantics**

Most policies that affect queries are scoping policies. These policies relate to the scope of a search when a query is submitted to the trader. Here are the high-level semantics when the trader processes a query:

1. Consider the number of initial offers to be searched.
2. Match the offers against the constraints specified in the query.
3. Consider the number of "hops" between linked traders during a search.
4. Order the results according to the preference supplied in the query. (No policies relate to this.)
5. Return these offers to the user.

**Search policies**

The following policies govern the scope of this search:

| | |
|---|---|
| `search_card` | Consider at most this number of offers for the search. |
| `match_card` | Match at most this number of offers before returning them to be ordered. |
| `hop_count` | Allow at most this number of hops from one trader to another linked trader. |
| `return_card` | Return at most this number of offers to the client. |

These policies can be optionally specified in a query. Each of these policies have "tuning" policies associated with them in the trader. The trader tuning policies are called `def_`*policy* and `max_`*policy* where *policy* is the name of one of the policies listed above. For example, the `search_card` policy may be specified in a query and the `def_search_card` and `max_search_card` policies have initial values in the Trader Service when it starts up. The trader policies may be changed using functions from the `CosTrading::Admin` interface.

If a query doesn't specify a value for a policy, then the appropriate `def_`*policy* value of the trader applies. If the query specifies a value for a policy, then it applies for the duration of that query, except where it exceeds the trader's `max_`*policy* value, in which case the value `max_`*policy* is used. For example, suppose that in the trader, `def_search_card` is 50, and `max_search_card` is 500:

- If the query doesn't specify a `search_card`, then at most 50 offers will be considered in the initial search.
- If the query specifies *"search_card = 100"*, then 100 offers will be initially considered.
- If the query specifies "*search_card = 600*", then, since this exceeds the trader's maximum, at most 500 offers will be initially considered.

**Return policy**

The policy `exact_type_match` may also be defined in a query. The value of this policy is a `boolean`. If it is specified as true, then only offers that exactly match the specified service type are considered; super-types are omitted. Otherwise, offers of any conforming service type are considered.

**List of query policies**

Table 2 shows the policies you can set in a query, along with the associated trader policies that affect the query policy. (A complete list and description of each policy is in the *CORBA Programmer's Reference*.)

**Table 2:** *Query policies and Trader Service policies*

| Policies You Can Set in a Query | Related Trader Service Policies |
|---|---|
| *Searching Service Offers* | |
| exact_type_match | |
| search_card | def_search_card<br>max_search_card |
| *Matching Query Constraints* | |
| match_card | def_match_card<br>max_match_card |
| *Number of Returned Offers* | |
| return_card | def_return_card<br>max_return_card |
| *Links Between Traders* | |
| hop_count | def_hop_count<br>max_hop_count |

**Table 2:** *Query policies and Trader Service policies*

| Policies You Can Set in a Query | Related Trader Service Policies |
|---|---|
| link_follow_rule | default_follow_rule[a]<br>def_follow_policy<br>limiting_follow_rule[a]<br>link_follow_rule<br>max_follow_policy |
| request_id_stem | request_id_stem |
| starting_trader | |
| *Optional Trader Capabilities* | |
| use_dynamic_properties | supports_dynamic_properties |
| use_modifiable_properties | supports_modifiable_properties |
| use_proxy_offers | supports_proxy_offers |

a. These are set in the links created between traders. See "Managing Links Between Traders" on page 76.

# Policies that Affect Trader Functionality

**Overview**

A trader may support some optional functionality. These include *modifiable properties*, *dynamic properties*, and *proxy offers*. In a particular query, a client may chose not to consider offers that require such functionality, even if the trader supports it (for example, to optimize the speed of a query).

**Evaluating policies**

To find out if a trader supports any of this functionality, the following attributes are provided in the `CosTrading::SupportAttributes` interface. The *get* function for these attributes return a `boolean` representing the value of the policy:

- `supports_modifiable_properties`(default: `true`)
- `supports_dynamic_properties`(default: `true`)
- `supports_proxy_offers`(default: `true`)

**Query policies**

In a query, these policies are specified using the following `boolean` variables. If the trader does not support the functionality, then the corresponding query policy is ignored.

- `use_modifiable_properties`
- `use_dynamic_properties`
- `use_proxy_offers`

# Using Policies in a Query

**Policies parameter**

You can use the policies parameter in a query to control the query and override some of the default Trader Service policies. A `Policy` data structure contains two members:

- A `PolicyName` is a pre-defined `string` identifier used by a trader to identify a policy.
- A `PolicyValue` is the value specified for a policy. The `PolicyValue` is of type `any`. The type of the `any` value should match the type of the corresponding policy.

An application may manipulate policies by using a `PolicySeq`, which is a sequence of `Policy` data structures.

### Creating a policy list

Example 4 shows how to create a policy sequence and pass it to a `query()`:

**Example 4:** *Creating a policy sequence*

```
//C++
1 CosTrading::PolicySeq_var policies;
  policies = new PolicySeq(2);
2 policies[0].name = (const char *) "search_card";
  policies[0].value <<= (CORBA::ULong) 50;
3 policies[1].name = (const char *)"use_dynamic_properties";
  policies[1].value <<= CORBA::Any::from_boolean (0);

  lookup->query("IDL:printer:1.0", "page_per_min > 5", "first",
4 policies,
  desired_properties,
  how_many_offers,
  offers,
  offer_itr,
5 limits_applied
  );
```

1. First create a sequence of `Policy` structures, and set the number of policies you want to specify for the query. This example uses just two policies.
2. Setting the `search_card` policy to 50 means that the query should look at a maximum of 50 offers initially before matching this query.
3. Setting the `use_dynamic_properties` policy to `false` means that the query should not consider offers with dynamic properties.
4. Use the policy sequence as a parameter in the `query()` function.
5. If the Trader Service encounters any policy limits during a query, it returns the names of the policies in this parameter as a sequence of policy names. For example, if a query generates more offers than the maximum number of offers the trader is allowed to search for, the name `max_search_card` is returned in the sequence. The values of the policies are not returned.

### Policy types

Table 3 shows the policies you can set and the associated IDL data type for each policy:

**Table 3:** *Policies You Can Set for a Query*

| Policy Name | IDL Type |
| --- | --- |
| exact_type_match | boolean |
| hop_count | unsigned long |
| link_follow_rule | CosTrading::FollowOption |
| match_card | unsigned long |
| request_id_stem | CosTrading::OctetSeq |
| return_card | unsigned long |
| search_card | unsigned long |
| starting_trader | CosTrading::TraderName |
| use_dynamic_properties | boolean |
| use_modifiable_properties | boolean |
| use_proxy_offers | boolean |

# Setting a Trader's Global Policies

**Setting global policies**

For each policy in the trader, the `CosTrading::Admin` interface has an associated set function that you can use to set the policy value. The set functions take the form `set_policy_name(value)`, where *policy_name* is the policy you wish to set. For example, you can use the `Admin::set_max_match_card()` function to set the `max_match_card` attribute of the `ImportAttributes` interface.

Also see "Setting policies for linked traders" on page 76 for another example of how to set policies.

**Global policies**

Table 4 summarizes the global trader policies an administration application can set:

**Table 4:** *Trader policies*

| Policy Name | Type |
|---|---|
| default_follow_rule | CosTrading::FollowOption |
| def_follow_policy | CosTrading::FollowOption |
| def_hop_count | unsigned long |
| def_search_card | unsigned long |
| def_match_card | unsigned long |
| def_return_card | unsigned long |
| max_follow_policy | CosTrading::FollowOption |
| max_hop_count | unsigned long |
| max_search_card | unsigned long |
| max_link_follow_policy | CosTrading::FollowOption |
| max_list | unsigned long |
| max_match_card | unsigned long |

**Table 4:** *Trader policies*

| Policy Name | Type |
| --- | --- |
| max_return_card | unsigned long |
| request_id_stem | CosTrading::OctetSeq |
| supports_dynamic_properties | boolean |
| supports_modifiable_properties | boolean |
| supports_proxy_offers | boolean |

CHAPTER 6

# Exporting and Managing Service Offers

*Application servers can advertise their services by exporting service offers to the Trader Service. Servers can also manage their service offers in the Trader Service by getting offer information, modifying an offer's properties, and withdrawing an offer.*

---

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Overview | page 60 |
| Initializing Service Offer Properties | page 61 |
| Exporting a Service Offer to Trader | page 63 |
| Getting Service Offer Data from Trader | page 65 |
| Modifying a Service Offer | page 66 |
| Withdrawing a Service Offer from Trader | page 68 |

# Overview

**Server tasks**

This chapter describes the following service offer tasks for servers:

- How to initialize service offer properties prior to exporting to the Trader Service.
- How to export service offers to the Trader Service.
- How to get service offer data from the Trader Service.
- How to modify a service offer already in the Trader Service.
- How to withdraw a service offer from the Trader Service.

**Environment**

All of this chapter's discussion and associated programming examples can be done within a specific application server, such as the printer server shown in the examples here. However, you might just as likely do this programming with a management server that is separate from the servers supplying specific resources. Whether it is better for your server to export and manage its own offer or for a separate management program to do it depends on your programming style and application design.

# Initializing Service Offer Properties

**Property structure**

Offer properties are stored as a sequence of property structures, where each property is a name-value pair, as follows:

```
// IDL
typedef Istring PropertyName;
typedef any PropertyValue;
struct Property {
PropertyName name;
PropertyValue value;
};
typedef sequence<Property> PropertySeq;
```

**Getting an offer type's property list**

Information about all services that the Trader Service can support is stored as service types in the Trader Service repository. When you develop a server for a specific service, you will need to have the information about the service type's properties. There may be documentation describing these properties or you may need to extract the information from the Trader Service. The section, "Managing the Service Type Repository" on page 70 explains how to add service types to a trader and how to list a trader's service type property information.

Before a server can export an offer to a trader, it must initialize the offer's properties. A server initializes the service offer properties with relevant values. For example:

```
// C++
CosTrading::PropertySeq_var properties = new
   CosTrading::PropertySeq ();
properties->length(3);
properties[0].name = (const char *) "name";
properties[0].value <<= "laser4";
properties[1].name = (const char *) "location";
properties[1].value <<= "near coffee machine";
properties[2].name = (const char *) "page_per_min";
properties[2].value <<= (CORBA::Long) 50;
```

## Read-only and mandatory properties

Before you initialize an offer's properties, check the service type information for any *mandatory* properties and any *readonly* properties. You must set a value for mandatory properties in order to successfully export an offer. Readonly properties cannot be modified once the offer is exported. Each property has assigned to it one of the following modes:

| | |
|---|---|
| normal | A service offer need not supply a value for this property. |
| readonly | A service offer need not supply a value for this property. However if it does, the value cannot be modified after its offer is exported. |
| mandatory | A value for this property must always be provided when a service offer is exported. |
| mandatory and readonly | A value for this property must be supplied and it cannot be modified after it is exported. |

# Exporting a Service Offer to Trader

**Overview**

Servers use the `export()` operation to register a service offer with the Trader Service.

**Synopsis**

The `export()` operation takes the following form:

```
// C++
CosTrading::OfferId CosTrading::Register::export(
  Any service_object,
  String service_type,
  PopertySeq properties)
```

It takes the following parameters:

| | |
|---|---|
| `service_object` | A reference to the object providing the service |
| `service_type` | The name of the service type that represents the service being offered |
| `properties` | A sequence of `Property` which describes the offer's properties. See "Initializing Service Offer Properties" on page 61 for more information. |

`export()` returns a `CosTrading::OfferId` which uniquely identifies the service offer within the trader. This value is needed to modify or withdraw the service offer.

**Example**

Example 5 shows how to export a service offer:

**Example 5:** *Exporting a service offer*

```
// C++
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
...
1 PrintServer_impl * impl = new PrintServer_impl();
TraderDemo::PrintServer_var print_server = impl->_this();
...
```

**Example 5:** *Exporting a service offer*

```
2  CORBA::Object_var trader =
      orb->resolve_initial_references("TradingService");

   CosTrading::Lookup_var lookup =
      CosTrading::Lookup::_narrow(trader);

3  CosTrading::Register_var register = lookup->register_if();

4  CosTrading::OfferId offer_id = register->export(
     print_server, // the object itself
     "Printer", // service type
     properties  // initialized in previous example
   );
```

Do the following programming steps to export the service offer:

1. Create an instance of the service object. For example, this application uses a printer service.
2. Connect to the Trader Service. Call the function `resolve_initial_references()` to get an object reference to the trader and narrow the returned value to get the trader's `Lookup` object.
3. Narrow the `Lookup` object to get the trader's `Register` component, which handles service offer exporting.
4. Invoke the `export()` function to export the service offer.

# Getting Service Offer Data from Trader

**Reviewing offer information**

After a server exports an offer, you can review the information contained in the offer by using the `describe()` operation of the `Register` interface. This function takes an `OfferId` as input and returns an `OfferInfo` structure representing offer information. The following example continues from the previous one:

```
// C++
CosTrading::Register::OfferInfo_var offer_info =
    register->describe(offer_id);
```

The `CosTrading::Admin` interface includes an `list_offers()` function you can use to obtain a list of all offers held by the Trader Service.

# Modifying a Service Offer

### Using modify()

You can modify any properties of an offer, other than those declared read-only, by using the `modify()` operation of the trader's `Register` object to delete, add, or change its properties.

`modify()` takes the following parameters:

| | |
|---|---|
| `offer_id` | The `OfferId` returned by the trader when the offer was exported. |
| `delete_list` | A sequence of `PropertName` specifying which properties to delete from the offer. |
| `change_list` | A sequence of `Property` specifying the properties to modify, along with their new values. New properties can included in this list and they will be added to the offer. |

### Example

Example 6 shows how to delete a property and how to change the value of a property.

**Example 6:** *Deleting a property*

```
// C++
1 CosTrading::PropertyNameSeq_var delete_list = new
     CosTrading::PropertyNameSeq();
  delete_list->length(1);
  delete_list[0]= (const char *) "page_per_min";

2 CosTrading::PropertySeq_var change_list = new
     CosTrading::PropertySeq ();
  change_list->length(1);
  change_list[0].name = (const char *) "location";
  change_list[0].value <<= "A-wing, first floor";

3 register->modify(offer_id, delete_list, change_list);
```

1. Create a sequence of the property names to be deleted.
2. Create the sequence of the properties to be added or modified.
3. Finally, call the `modify()` function

**Readonly properties**

For situations in which you need to change readonly properties, you can withdraw an offer and then export a new offer, but this changes the offer ID which may affect applications that already hold the current offer.

**Policy for supporting modifiable properties**

The `supports_modifiable_properties` policy is a `boolean` attribute that indicates whether or not a specific trader supports modifiable properties. Servers and administration applications can turn support for modifiable properties on or off by using `CosTrading::Admin::set_supports_modifiable_properties()`. To obtain the current value of this policy, query the `CosTrading::SupportAttributes::supports_modifiable_properties` attribute.

# Withdrawing a Service Offer from Trader

When it is necessary to withdraw an offer from the Trader Service, use the `CosTrading::Register::withdraw()` function. The function requires as input the `offer_id`, which is obtained as a result of the `export()` function:

```
register->withdraw(offer_id);
```

Administration applications can use the following function to withdraw multiple offers satisfying a specified service type and constraint:

```
register->withdraw_using_constraint(type_name, my_constraint);
```

# Programming Topics

*This chapter is a brief introduction to some advanced programming topics and features of the Trader Service. These topics include adding service types, using dynamic properties, and managing links between traders.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Managing the Service Type Repository | page 70 |
| Using Dynamic Property Values | page 73 |
| Managing Links Between Traders | page 76 |

# Managing the Service Type Repository

**Overview**

Servers cannot export service offers to the Trader Service unless the appropriate service types are stored in the Service Type Repository. You use the operations of the `ServiceTypeRepository` interface of the `CosTradingRepos` module to manage service types in a trader. Service types are added to a trader only occasionally, and usually by a management type of server. Service types are rarely removed from a trader.

**Creating service type properties**

You create service type properties and then you add these properties along with other information to the Trader Service. Example 7 shows how to create service type properties.

**Example 7:** *Creating service type properties*

```
   // C++
1  CosTradingRepos::ServiceTypeRepository::PropStructSeq_var
      properties = new
      CosTradingRepos::ServiceTypeRepository::PropStructSeq;
   propertyies->length(3);

2  properties[(CORBA::Ulong) 0].name = (const char *) "name";
3  properties[(CORBA::Ulong) 0].value_type = CORBA::_tc_string;
4  properties[(CORBA::Ulong) 0].mode =
      CosTradingRepos::ServiceTypeRepository::PROP_MANDATORY_READON
      LY;
   properties[(CORBA::Ulong) 1].name = (const char *) "location";
   properties[(CORBA::Ulong) 1].value_type = CORBA::_tc_string;
   properties[(CORBA::Ulong) 1].mode =
   CosTradingRepos::ServiceTypeRepository::PROP_MANDATORY;
   properties[(CORBA::Ulong) 2].name = (const char *)
      "page_per_min";
   properties[(CORBA::Ulong) 2].value_type = CORBA::_tc_long;
   properties[(CORBA::Ulong) 2].mode =
   CosTradingRepos::ServiceTypeRepository::PROP_NORMAL;
```

The code is described as follows:

1. Create a new sequence of property structures and set the length of the sequence to equal the number of properties for the service type.

2. For each property, assign a character string to represent the name of the property. This printer example has three properties named `name`, `location`, and `page_per_min`.
3. Set the data type for the value of each property. This is standard Orbix programming for setting values for typecodes.
4. Set the mode for each property.
   - A `PROP_MANDATORY_READONLY` mode means the property *must* be set when exporting a service offer of this service type, but once it is exported, it cannot be changed.
   - A `PROP_MANDATORY` mode means the property *must* be set when exporting a service offer, but its value may be changed later if needed, after the service offer is exported.
   - A `PROP_READONLY` mode (not shown) means the property may be set when exporting a service offer, but once it is exported, it cannot be changed.
   - A `PROP_NORMAL` mode means the property may be set in a service offer but it is not required. It can be changed later.

---

### Adding a service type

Example 8 adds a service type named `IDL:printer:1.0` to a trader. The interface type name from the IDL file is `IDL:TraderDemo/PrintServer:1.0`. The properties, created in Example 7 are the next parameter. The `super_types` parameter is a list of types from which this service type is derived. A subtype must support properties of its supertypes. In this example, there are no supertypes.

**Example 8:** *Adding a service type*

1

```
// C++
CORBA::Object_var trader =
    orb->resolve_initial_references("TradingService");
CosTrading::Lookup_var lookup =
    CosTrading::Lookup::_narrow(trader);
CosTradingRepos::ServiceTypeRepository_var type_repos_obj =
    CosTradingRepos::ServiceTypeRepository::_narrow(lookup->type_
    repos());
```

**Example 8:** *Adding a service type*

```
2  CosTradingRepos::ServiceTypeRepository::ServiceTypeNameSeq_var
      super_types =
      CosTradingRepos::ServiceTypeRepository::ServiceTypeName_Seq;

3  type_repos_obj->add_type (
     "IDL:printer:1.0",                  // service type name
     "IDL:printer_if:1.0",               // idl type name
     properties,                         // property information
     super_types                         // no super types
   );
```

The code is described as follows:

1. Connect to the trader using `resolve_initial_references`, narrow the returned object to a `Lookup` object, and use that to get a reference to the Trader Service's Service Type Repository.
2. Create a list of the supertypes which define the service type being created. For this example, there are no supertypes.
3. Call `add_type` to add the service type to the Service Type Repository.

---

### Managing service types

After a service type is added to the Trader Service, applications can use other operations of the `CosTradingRepos::ServiceTypeRepository` interface to manage service types. These include `remove_type()` and `list_types()`. The `describe_type()` operation returns information that describes the type, and the `fully_describe_type()` operation searches recursively to return information on all the supertypes for this type.

You can also hide service types from outside the service type repository by using the `mask_type()` operation. This may be used, for example, where a type is no longer needed, but it is the supertype of other types in the type repository. Use the `unmask_type()` operation if you need to make a masked service type visible again.

See also the `SupportAttributes::type_repos` attribute and the `Admin::set_type_repos()` operation. These get a reference to the type repository interface and set the type repository interface in a trader.

# Using Dynamic Property Values

**Dynamic property values**

Exported offers can contain dynamic property values. These are values that can change at runtime. For example, the number of print jobs in a printer queue. This is done by exporting an object reference that can be invoked to retrieve the current value from the server. Clients and the trader can then dynamically determine the length of the printer queue at the time of a query. This is not as fast as using static values, but it can greatly increase the flexibility involved.

The fact that a property has a dynamic value is only relevant at export time. There is no difference when defining the property in the service type.

**Exporting a dynamic property value**

When a dynamic property value is being exported, then the type `CosTradingDynamic::DynamicProp` is used rather than the expected type. The trader recognizes this as a special type, and treats it accordingly.

The following code shows how to set properties, including a dynamic property, and then export the offer that contains the dynamic property.

**Example 9:** *Setting dynamic properties*

```
// C++
CosTrading::PropertySeq_var  properties = new
   CosTrading::PropertySeq();
properties->length(4);
properties[(CORBA::Ulong) 0].name = (const char *) "name";
properties[(CORBA::Ulong) 0].value <<= "laser4";
properties[(CORBA::Ulong) 1].name = (const char *) "location";
properties[(CORBA::Ulong) 1].value <<= "near coffee machine";
properties[(CORBA::Ulong) 2].name = (const char *)
   "page_per_min";
properties[(CORBA::Ulong) 2].value <<= (CORBA::Long) 50;

CosTradingDynamic::DynamicProp dynam_prop_val;
1 dynam_prop_val.eval_if =
   CosTradingDynamic::DynamicPropEval::_duplicate(dynam_obj);
```

**Example 9:** *Setting dynamic properties*

```
2  dynam_prop_val.returned_type = CORBA::_tc_ushort;
   dynam_prop_val.extra_info <<= (CORBA::UShort) 0;
   properties[(CORBA::Ulong) 3].name = (const char *)
      "queue_length";
3  properties[(CORBA::Ulong) 3].value <<= dynam_prop_val;

   register_obj->export(
     obj,                          // object reference
     "IDL:printer:1.0",            // service type name
     properties                    // seq of properties
   );
```

The code is described as follows:

1. `eval_if` is essentially a callback object in the server. It implements the IDL interface `CosTradingDynamic::DynamicPropEval`, which contains one operation `evalDP`, which returns the current value of the property in the server when invoked.
2. The `returned_type` must be the same type as the corresponding property type defined in the service type. `extra_info` is essentially ignored by the trader, but may be used by users to carry additional information.
3. The dynamic value is assigned to the property value.

## Using a dynamic property value

Clients may need to check if the value in a property is dynamic or not, if it is possible that the value may be either a static or dynamic value.

**Example 10:** *Using a dynamic property value*

```
// C++
CosTrading::Property prop = properties[2];

CORBA::UShort length;
if (prop.value.containsType(CORBA::_tc_ushort)) {
          prop.value >>= length;
          cout << "static queue_length " << length << endl;
}
else if
    (prop.value.containsType(CosTradingDynamic::_tc_DynamicProp))
    {
     CosTradingDynamic::DynamicProp * dynam_prop;
     prop.value >>= dynam_prop;

     CosTradingDynamic::DynamicPropEval_ptr dynam_eval =
    dynam_prop->eval_if;
     CORBA::Any* length_any;

     length_any = dynam_eval->evalDP(
     prop.name,
     dynam_prop->returned_type,
     dynam_prop->extra_info);
     *length_any >>= length;
     cout << "dynamic queue_length " << length << endl;
}
```

Note that if the trader itself evaluates the dynamic property value, because it is used in a constraint expression (for example, “queue_length < 10"), then it will return the static value at the time of evaluation in the offer’s properties. This is to minimize the evaluation times on dynamic properties.

## Allowing dynamic properties

While the Trader Service allows dynamic properties by default, a specific trader may be set to not allow dynamic properties. The `CosTrading::SupportAttributes::supports_dynamic_properties` policy is a boolean attribute that indicates whether or not the trader allows dynamic properties. Servers and administration applications can set this policy value by using the operation `CosTrading::Admin::set_supports_dynamic_properties()`.

# Managing Links Between Traders

**Linked traders**

A linked trader shares information about its service offers with one or more other traders. Linked traders allow administrators to organize service types and service offers in logical and more efficient ways for specific environments.

This section describes the following Link function tasks:

- Setting link trader policies.
- Adding and removing links.
- Listing links to other traders.

**Setting policies for linked traders**

Chapter 5 introduced several policies that relate to linked traders including `hop_count`, `link_follow_rule`, and `default_follow_rule`. A client query can set some of these policies to control the search for offers, but other policies relating to linked traders control the links and may override the query policies.

Example 11 shows the use of `request_id_stem`. This should be a unique value per trader. It will be used in queries send to other traders, to prevent infinite looping. When a trader sees an incoming query with its own request id stem, it does not process the query, and returns a result of zero offers to the calling trader.

**Example 11:** *Using request_id_stem*

```
// C++
// set request_id_stem to "1"
const char * word = "1";
OctetSeq stem(1);
stem.length(1);
stem[0] = word[0];

// admin_obj is a pointer to the Admin Interface of Orbix Trader
admin_obj->set_request_id_stem(stem);
```

The following code shows how to set other trader policies relating to links:

```
// C++
// set all the follow options
CosTrading::FollowOption max_follow = CosTrading::always;
CosTrading::FollowOption max_link_follow = CosTrading::always;
CosTrading::FollowOption def_follow = CosTrading::always;

// set options for hops between traders
int max_hop = 10;
int def_hop = 10;

try {
admin_obj->set_max_follow_policy(max_follow);
admin_obj->set_def_follow_policy(def_follow);
admin_obj->set_max_link_follow_policy(max_link_follow);
admin_obj->set_def_hop_count(def_hop);
admin_obj->set_max_hop_count(max_hop);
} catch (...) {
cerr << "Call to set policies failed" << endl;
exit(1);
}
```

## Adding links

Example 12 shows how to add a link from one trader to another. The trader1 establishes a link to trader2. The link is called link_to_trader2.

**Example 12:** *Linking from one trader to another*

```
// C++
CosTrading::LinkName name =
   CORBA::string_dup("link_to_trader2");
// set CosTrading::FollowOptions for add_link
CosTrading::FollowOption def_pass_on_follow_rule =
   CosTrading::always;
CosTrading::FollowOption limiting_follow_rule =
   CosTrading::always;

// link_var is a pointer to the Link Interface of trader1
// target is a pointer to trader2 Lookup Interface
try {
   link_var->add_link(
      name,
      target,
      def_pass_on_follow_rule,
      limiting_follow_rule
   );
} catch (...) {
   cerr << "Call to add_link failed" << endl;
   exit(1);
}
```

## Removing links

Example 13 shows how to remove a link. The link removed is the one created in Example 12:

**Example 13:** *Removing a link*

```
// C++
// set LinkName
CosTrading::LinkName name =
   CORBA::string_dup("link_to_trader2");
// link_var is a pointer to the Link Interface of trader1
try {
   link_var->remove_link(name);
} catch (...) {
   cerr << "Call to remove_link failed" << endl;
   cerr << "Unexpected exception" << endl;
   exit(1);
}
```

## Creating lists of links

Example 14 shows how to create a listing of the links to other traders:

**Example 14:** *Creating a list of links to traders*

```
// C++
// lists links of trader1
CosTrading::LinkNameSeq_var link_names = 0;
// link_var is a pointer to the Link Interface of trader1
try {
   link_names = link_var->list_links();
} catch (...) {
   cerr << "Call to list_links failed" << endl;
   cerr << "Unexpected exception" << endl;
   exit(1);
}
// This prints the link names
unsigned long length = link_names->length();
for (CORBA::Ulong i = 0; i < length; i++)
   cout << (*link_names)[i] << endl;
```

CHAPTER 8

# Trader Service Console

*The Trader Service Console allows you to manage all aspects of the Trader Service, including service types, offers, proxy offers and links. It also lets you perform queries, and configure the trader attributes.*

**In this chapter**

This chapter contains the following sections:


| | |
|---|---|
| Starting the Trader Console | page 83 |
| Main Window | page 84 |
| Managing Service Types | page 89 |
| Managing Offers | page 93 |
| Managing Proxy Offers | page 98 |
| Managing Links | page 100 |
| Configuring the Trader Attributes | page 102 |
| Admin Attributes | page 107 |

| | |
|---|---|
| Executing Queries | page 108 |
| Connecting to a New Trader | page 111 |

# Starting the Trader Console

**How to start the console**

Run the following command in a command window:

```
ittrader_console
```

# Main Window

**GUI appearance**

The Trader Service Console main window appears as shown in Figure 5.

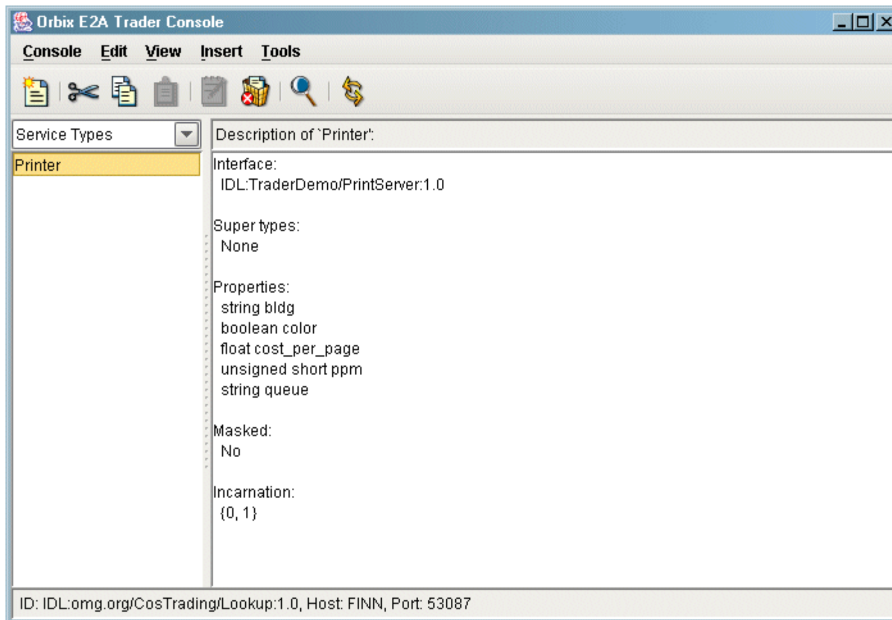



**Figure 5:** *The Trader Console main window*

**Window elements**

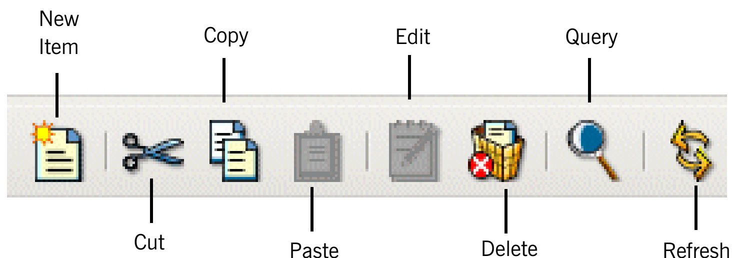
The main window includes the following elements:

| | |
|---|---|
| Menu bar | Provides access to all of the application's features |
| Toolbar | Shortcuts for the most common menu commands |
| Item type selector | Selects which type of item is shown in the item list |
| Item list | Displays the names or identifiers of all items contained in the trader |
| Item description | Provides a textual summary of the selected item |

| | |
|---|---|
| Status bar | Displays information about the trader to which the console is currently connected, including the host, port and IDL interface |

## The Toolbar

The toolbar contains buttons for the most common menu commands.




## Terms used in trader console

The Trader Service Console uses the term *item* to generically refer to the four types of data managed by a trader service:

- Service types
- Offers
- Proxy offers
- Links

The console window is used to browse these items. The window only shows one type of item at a time, which you can change with the **item type selector** drop-down list or by selecting a type from the **View** menu. When a new *item type* is selected, the current list of items is retrieved from the trader service and displayed in the **item list**.

# The Trader Console Menus

### The Console menu

You use the commands in the **Console** menu to manage the console windows.

| | |
|---|---|
| New Window | Creates a new console window, connected to the current trader. |
| Connect | Opens the **Connect** dialog box, allowing you to connect to a different trader service. |
| Close | Closes the current console window. If this window is the last console window present, the application exits. |
| Exit | Quits the application. |

### The Edit menu

The console supports the typical notion of a clipboard, which can be manipulated with cut, copy and paste commands. However, the console does not use the system clipboard, and therefore the application clipboard can only be accessed by windows from the same execution of the application. In other words, if you start two instances of the application, you cannot cut and paste between them. You can cut and paste if you start a single instance of the application, and create multiple windows with the **Console/New Window** command.

| | |
|---|---|
| Cut | Copies the selected items to the clipboard and then permanently removes the selected items |
| Copy | Copies the selected items to the clipboard |
| Paste | Pastes the items from the clipboard into the current trader |
| Select All | Selects all of the items in the item list |
| Clone | Creates a clone of the selected item. The appropriate dialog box is displayed to allow you to create a new item, but the fields of the dialog box are initialized with the values from the selected item. |
| Modify | Edits the currently selected item |
| Delete | Permanently removes the selected items |

### Using the cut, copy and paste commands

There are some issues to be aware of when using the cut, copy and paste operations:

- Service types and links must have unique names; therefore, you will not be able to paste one of these items if an item already exists in the trader with the same name.
- A certain amount of forethought is advised when you wish to cut and paste service types. Since service types can inherit from other service types, you cannot cut a service type that has subtypes. If you wish to cut or delete a number of service types, and if inheritance relationships exist between any of them, you must cut the types that don't have any subtypes first. The same principle applies to pasting service types, in that you cannot paste a type if its supertypes do not exist or have not yet been pasted. It is recommended that you only operate on one service type at a time when using the cut, copy, paste or delete commands.

### Using the Clone and Modify commands

Note the following when using the clone and modify operations:

- Some types of items, namely service types and proxy offers, cannot be modified. The **Modify** command (and its toolbar equivalent) are disabled while these types are displayed.
- The **Clone** and **Modify** commands operate on a single item at a time. If more than one item is currently selected, the application uses the first of the selected items.

### The View menu

You use the **View** menu to select the type of items you wish to be displayed in the item list. Selecting a new type of item from this menu is equivalent to changing the setting of the item type selector.

The **Refresh** command causes the application to retrieve an updated list of items from the trader and display them in the item list. This command can be useful if you know (or suspect) that the list of items has been changed by some other client of the trader service.

## The Insert menu

You use the **Insert** menu when you want to create a new item. It displays a dialog box in which you can supply the information about the new item. If you need to create a new item that is similar to an existing item, you can also use the **Edit/Clone** command.

## The Tools menu

The commands available in the **Tools** menu provide access to additional features of the trader service.

| | |
|---|---|
| Query | Perform query operations on the trader and review the matching offers |
| Attributes | Configure the trader attributes |
| Withdraw Offers | Removes offers using a constraint expression |
| Mask Type | Masks the selected service type |
| Unmask Type | Unmasks the selected service type |

# Managing Service Types

## IDL type support

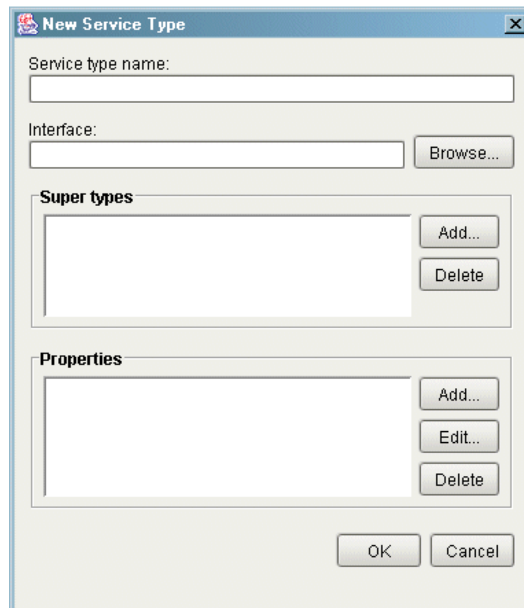
Although the Trader Service supports properties with user-defined IDL types, the console only supports simple IDL types and sequences of simple IDL types.

Refer to "Service Types" on page 4 for more information on service types.

## Adding a new service type

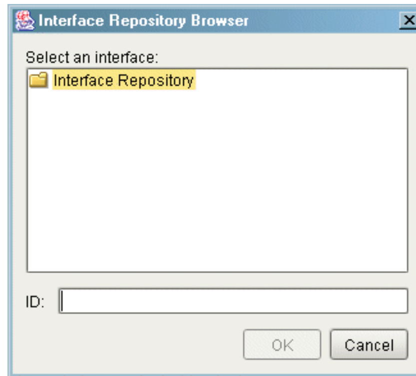
To add a new service type:

1. Select **Insert/Service Type**. The **New Service Type** dialog box appears as shown below.

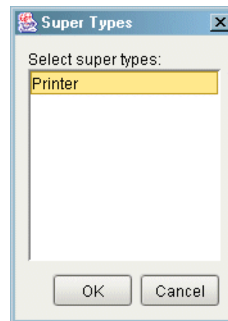



2. Enter a name for the service type in the **Service type name** text box. The name must be unique among all of the service types managed by the trader.

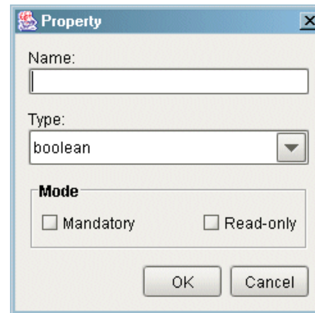
3. Enter an interface repository identifier in the **Interface** text box. If the interface repository service is available, clicking the **Browse...** button displays an interface repository browser as shown below.




4. The browser displays only modules and interfaces. When you select an interface, its identifier is displayed in the **ID** text box below. Click **OK** to accept the identifier you have selected.
5. Use the **Add...** and **Delete** buttons to add and remove super types. Clicking the **Add...** button displays the **Super Types** dialog box as shown below. Select any service types you wish to use as super types for the new type and click **OK**. The order in which you add super types is not important.



6. Use the **Add...**, **Edit...** and **Delete** buttons to manipulate the properties for this service type. Clicking the **Add...** or **Edit...** buttons displays the **Property** dialog box as shown below. Enter a name for the property, select a property type, and use the checkboxes to indicate the mode of this property. Click **OK** to add the new property.




7. Click **OK** on the **New Service Type** dialog box to add the new service type.

---

## Removing a service type

To remove a service type, do the following:

1. Select **View/Service Types** to display the service types in the item list.
2. Select the service type you wish to remove.
3. Select **Edit/Delete**. A confirmation dialog appears.
4. Click **Yes** to permanently remove the service type.

**Note:** If a service type has subtypes, you will not be able to remove the type until all of its subtypes have been removed.

---

## Masking a service type

To mask a service type, do the following:

1. Select **View/Service Types** to display the service types in the item list.
2. Select the service type you wish to mask.
3. Select **Tools/Mask Type**.

### Unmasking a service type

To unmask a service type, do the following:

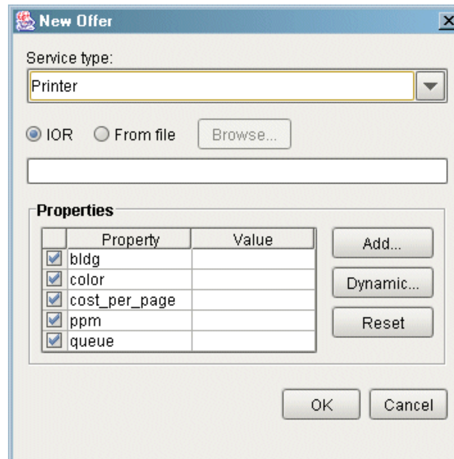
1. Select **View/Service Types** to display the service types in the item list.
2. Select the service type you wish to unmask.
3. Select **Tools/Unmask Type**.

# Managing Offers

## Adding a new offer

To add a new offer, do the following:

1. Select **Insert/Offer**. The **New Offer** dialog box appears as shown below.




2. Select a service type from the drop-down list. Each time you select a service type, the **Properties** table is updated to reflect the properties defined for that service type.
3. Select a method for specifying the object reference for this offer. Select the **IOR** toggle if you want to paste the stringified interoperable object reference into the text box. If you want the application to read the reference from a file, select **From file** and enter the filename in the text box, or click the **Browse...** button to display a file selection dialog box. If the trader service is configured to allow `nil` objects, and you do not wish to specify an object reference for this offer, you may leave the object reference blank.
4. Enter values for the properties in the **Properties** table. All properties have a checkbox to the left of the property name. For a mandatory property, the checkbox is disabled, meaning that a value must be provided for this property. For an optional property, you can use the

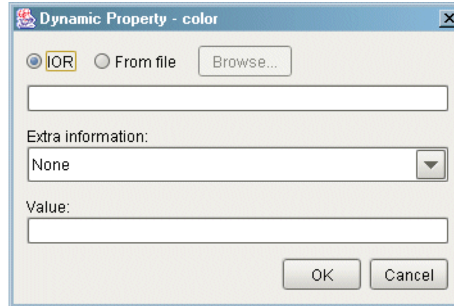
checkbox to indicate whether this property should be included with the offer. To enter a value for a property, double-click on the property value field. For properties with sequence types, you can enter multiple values by separating them with commas. Press **Return** when you are finished entering the value for a property.

5. Click the **Add...** button if you wish to add a property that is not defined by the service type. The **Add Property** dialog box appears as shown below. Enter a name for the property, select the property's type from the drop-down list, and enter a value in the text box. The name you use for the property must not be the same as any existing properties. Click **OK** to add the property to the **Properties** table.

**Note:** Once the property has been added to the **Properties** table, you can edit it directly, just as you can with any other property. If you later decide that you do not want to include the property with the offer, simply uncheck the property's checkbox.

6. If you wish to make a property dynamic, select the property and click the **Dynamic...** button. The **Dynamic Property** dialog box appears as shown below. Select a method for specifying an object reference as outlined in step 3 above. If you wish to include additional data, select

a type from the drop-down list and enter a value in the text box. Click **OK** to save the dynamic property. The property table displays `<dynamic>` as the value of a dynamic property.




7. To clear the value of a property, select the property and click **Reset**. You can use this command to convert a property from a dynamic property to a regular property.
8. Click **OK** to add the new offer. The application validates the information and reports any errors in a dialog box.

**Note:** For properties of type `string`, an empty value is accepted as a valid value, even for mandatory properties.

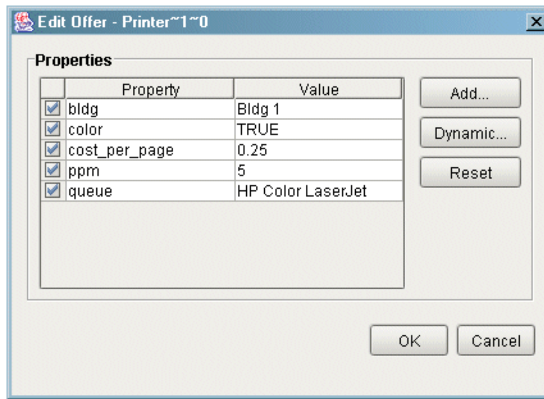
---

## Modifying an offer

To modify an offer, do the following:

1. Select **View/Offers** to display the offers in the item list.
2. Select the offer you wish to modify.

3. Select **Edit/Modify**. The **Edit Offer** dialog box appears as shown below.




4. You can modify a property by double-clicking on the property value. Press **Return** when you have finished editing a property value.
5. You can remove an existing property from the offer (if it is an optional property) by unchecking its checkbox. Similarly, you can add a property to the offer by checking its checkbox and entering a value for the property.
6. See the discussion of adding a new offer above for details on adding new properties and configuring dynamic properties.
7. Click **OK** to update the offer.

---

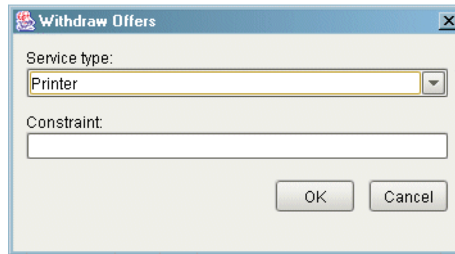
**Withdrawing offers**

There are two distinct ways to withdraw offers. The first way is by selecting individual offers, as outlined below:

1. Select **View/Offers** to display the offers in the item list.
2. Select the offer(s) you wish to withdraw.
3. Select **Edit/Delete**. A confirmation dialog appears.
4. Click **Yes** to withdraw the offers.

The above method is suitable for withdrawing a limited number of specific offers. A more efficient method for removing a large quantity of offers for a single type, or for removing offers without having to manually search for the right ones, is by withdrawing offers with a constraint expression:

1. Select **Tools/Withdraw Offers**. The **Withdraw Offers** dialog box appears as shown below.




2. Select the service type from the drop-down list. Offers with this service type or a subtype of this service type are considered for withdrawal.
3. Enter a constraint expression in the text box. See "Service Types" on page 4 for more information on constraint expressions.
4. Click **OK** to withdraw the offers. Only offers that match the constraint expression are withdrawn. An error message appears if no matching offers were found.

**Note:** A simple way to remove all of the offers for a service type is to use `TRUE` for the constraint expression.

# Managing Proxy Offers

**Adding a new proxy offer**

To add a new proxy offer, do the following:

1. Select **Insert/Proxy Offer**. The **New Proxy Offer** dialog box appears as shown below.

2. Select a service type from the drop-down list. Each time you select a service type, the property table is updated to reflect the properties defined for that service type.

3. Select a method for specifying the object reference of the target `CosTrading::Lookup` object for this proxy offer. Select the **IOR** toggle if you want to paste the stringified interoperable object reference into the

text box. If you want the application to read the reference from a file, select **From file** and enter the filename in the text box, or click the **Browse...** button to display a file selection dialog box.

4. Enter the constraint recipe in the text box.
5. Select **If match all** if a matching service type is all that is required for this proxy offer to be considered a match during a query.
6. Enter values for the properties in the **Properties** table. See "Adding a new offer" on page 93 for more information on entering offer properties.
7. Use the **Add...**, **Edit...** and **Delete** buttons to manipulate the policies to be passed on to the target object. Clicking the **Add...** or **Edit...** buttons displays the **Policy** dialog box as shown below.
8. Click **OK** to add the new proxy offer. The application validates the information and reports any errors in a dialog box.

## Withdrawing proxy offers

To withdraw a proxy offer, do the following:

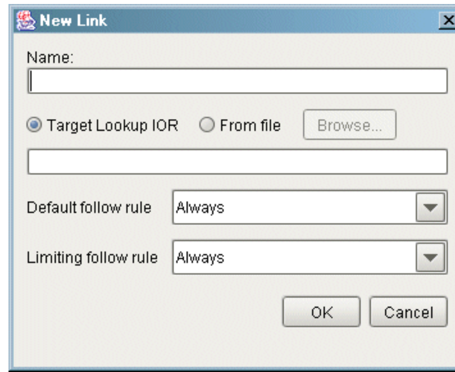
1. Select **View/Proxy Offers** to display the proxy offers in the item list.
2. Select the proxy offer you wish to withdraw.
3. Select **Edit/Delete**. A confirmation dialog appears.
4. Click **Yes** to withdraw the proxy offer.

# Managing Links

**Adding a new link**

To add a new link, do the following:

1. Select **Insert/Link**. The **New Link** dialog box appears as shown below.




2. Enter a name for this link in the text box.
3. Select a method for specifying the target trader's object reference for this link. Select the **IOR** toggle if you want to paste the stringified interoperable object reference into the text box. If you want the application to read the reference from a file, select **From file** and enter the filename in the text box, or click the **Browse...** button to display a file selection dialog box.
4. Select the appropriate link-follow rules from the drop-down lists.
5. Click **OK** to add the new link.

**Modifying a link**

To modify a link, do the following:

1. Select **View/Links** to display the links in the item list.
2. Select the link you wish to modify.
3. Select **Edit/Modify**. The **Edit Link** dialog box appears.
4. Update the settings for the link-follow rules.
5. Click **OK** to update the link.

### Removing a link

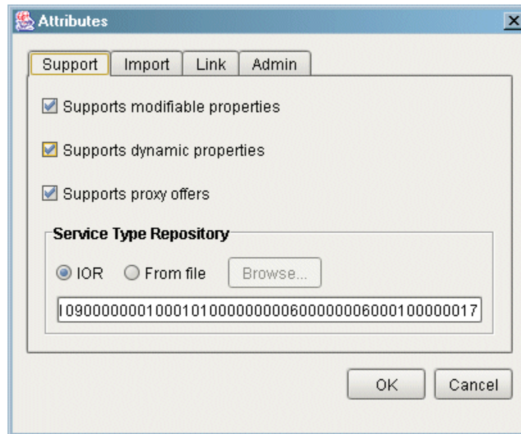
To remove a link, do the following:

1. Select **View/Links** to display the links in the item list.
2. Select the link you wish to remove.
3. Select **Edit/Delete**. A confirmation dialog appears.
4. Click **Yes** to remove the link.

# Configuring the Trader Attributes

**Configuring attributes**

To configure the trader attributes, select **Tools/Attributes**. The **Attributes** dialog box appears, containing a tabbed folder with four tabs.




The tabs provide access to the attributes from the following four IDL interfaces:

- `CosTrading::SupportAttributes`
- `CosTrading::ImportAttributes`
- `CosTrading::LinkAttributes`
- `CosTrading::Admin`

Each of the tabs is described below. Click **OK** when you have finished modifying the attributes.

# Support Attributes

**Supports modifiable properties**

If enabled, the trader considers offers with modifiable properties (that is, properties that are not read-only) during a query, unless the importer requests otherwise with the `use_modifiable_properties` policy. If disabled, the trader does not consider offers with modifiable properties, regardless of the importer's wishes. This setting also determines the behavior of the `modify` operation in the `CosTrading::Register` interface. If enabled, the server allows modification of offers. If disabled, the modify operation raises the `NotImplemented` exception.

**Supports dynamic properties**

If enabled, the trader considers offers with dynamic properties during a query, unless the importer requests otherwise with the `use_dynamic_properties` policy. If disabled, the trader does not consider offers with dynamic properties, regardless of the importer's wishes. The trading service specification does not define the behavior of a trader when this option is disabled and an offer is exported that contains dynamic properties; however, the Trader Service always accepts offers containing dynamic properties.

**Supports proxy offers**

If enabled, the trader considers proxy offers during a query, unless the importer requests otherwise with the `use_proxy_offers` policy. If disabled, the trader does not consider proxy offers, regardless of the importer's wishes. This setting also determines the behavior of the `proxy_if` attribute in the `CosTrading::SupportAttributes` interface. If enabled, `proxy_if` returns the reference of the server's `CosTrading::Proxy` object. If disabled, `proxy_if` returns `nil`.

**Service type repository**

The IOR of the service type repository currently in use by the trader is displayed in the text box. In order to change the service type repository, you first need to select a method for specifying its object reference. Select the **IOR** toggle if you want to paste the stringified IOR into the text box. If you want the application to read the reference from a file, select **From file** and enter the filename in the text box, or click the **Browse...** button to display a file selection dialog box.

# Import Attributes

**Import attributes**

Many of the Import attributes have default and maximum values. The default value is used if an importer does not supply a value for the corresponding importer policy. The maximum value is used as the allowable upper limit for the importer policy. If an importer supplies a policy value that is greater than the maximum value, the importer's policy value is overridden and the maximum value is used instead.

Attributes

Support | Import | Link | Admin

| | Default | Maximum |
|---|---|---|
| Search cardinality | 2147483647 | 2147483647 |
| Match cardinality | 2147483647 | 2147483647 |
| Return cardinality | 2147483647 | 2147483647 |
| Link hop count | 2147483647 | 2147483647 |
| Link follow policy | Always | Always |
| Maximum list count | 2147483647 | |

OK Cancel

**Search cardinality**

The number of offers to be searched during a query. The corresponding importer policy is `search_card`.

**Match cardinality**

The number of matched offers to be ordered during a query. The corresponding importer policy is `match_card`.

**Return cardinality**

The number of ordered offers to be returned by a query. The corresponding importer policy is `return_card`.

**Link hop count**

The depth of links to be traversed during a query. The corresponding importer policy is `hop_count`.

**Link follow policy**

The trader's behavior when considering whether to follow a link during a query. The default value is used if an importer does not specify a value for the `link_follow_rule` policy. The maximum value overrides the policy established for a link as well as the `link_follow_rule` policy proposed by an importer.

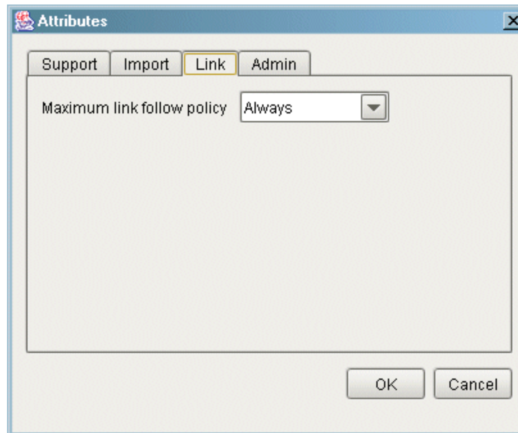
**Maximum list count**

The maximum number of items to be returned from any operation that returns a list, such as the `list_offers` operation in `CosTrading::Admin` or the `next_n` operation in `CosTrading::OfferIterator`. This attribute may override the number of items requested by a client.

# Link Attributes

**Link attributes**

The following is the Link Attributes pane.



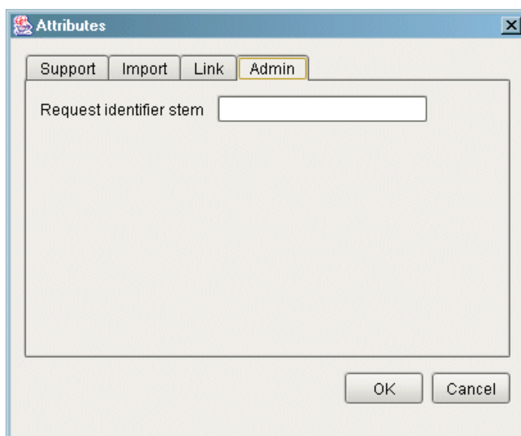

**Maximum link follow policy**

Determines the server's upper bound on the value of a link's limiting follow rule at the time of creation or modification of a link. The server raises the `LimitingFollowTooPermissive` exception if a link's limiting follow rule exceeds the value of this attribute.

# Admin Attributes

**Admin attributes**

The following is the Admin attributes pane.




**Request identifier stem**

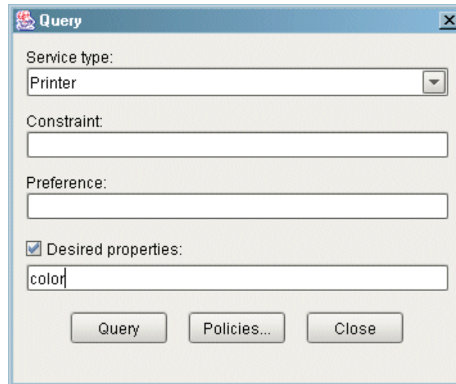
The request identifier stem is used as a prefix by the server to generate unique request identifiers during a federated query. Although the IDL attribute `request_id_stem` returns a sequence of octets, this property is defined in terms of a string, with the characters of the string comprising the octets of the stem. You need to provide a value for this property only if the server will have links to other traders and you want to ensure that circular links are handled correctly.

# Executing Queries

### Executing a query

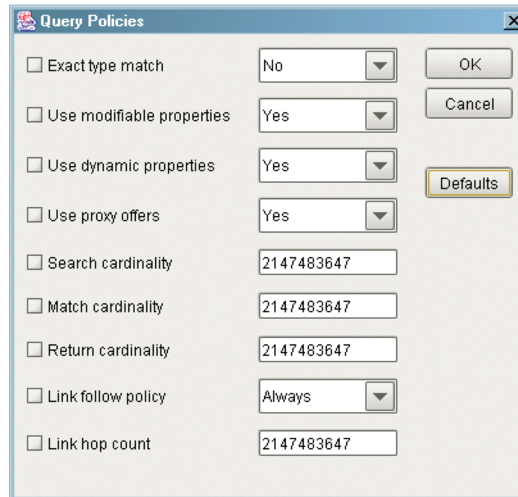
To execute a query, do the following:

1. Select **Tools/Query**. The **Query** dialog box appears as shown below.

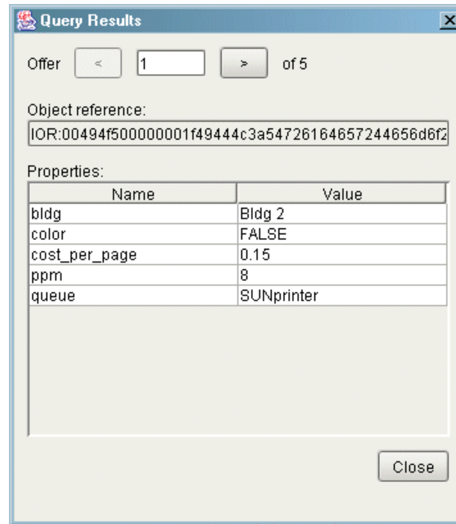



2. Select a service type from the drop-down list.
3. Enter a constraint expression in the **Constraint** text box.
4. (Optional) Enter a preference expression in the **Preference** text box. If this field is blank, the trader uses a default preference expression of `"first"`.
5. If you wish to specify which properties are returned in the matching offers, click **Desired properties** to activate the text box below and enter the names of the properties in the text box. Use commas to separate the property names.
6. To include importer policies, click the **Policies...** button. The **Policies** dialog box appears as shown below. Next to each field label is a checkbox. You must check the box for a policy for it to be included in

your query. Click the **Defaults** button to load the trader's default import attributes into the fields of the dialog box. Click **OK** to accept your changes.



7. Click **Query** to execute the query operation. If matching offers were found, the **Query Results** dialog box appears as shown below. You can scroll through the matching offers with the **<** and **>** buttons. Click **Close** when you have finished examining the results.

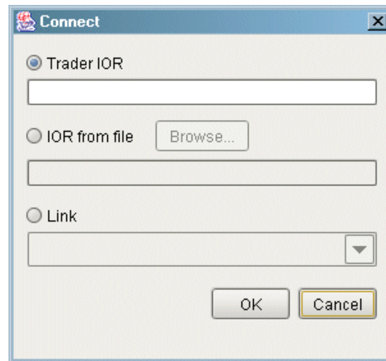



**Note:** The Query Results dialog box cannot be used to edit offers.

# Connecting to a New Trader

**Connecting to a new trader**

When the console is started, the first console window to appear is already connected to the trader you specified using the command line options. If you are managing multiple traders, you can connect to a different trader with the **Console/Connect** command. The **Connect** dialog appears as shown below.




There are three methods of connecting to the trader.

1. To provide the stringified object reference, select the **Trader IOR** option and paste the IOR into the text box.
2. To obtain the stringified object reference from a file, select **IOR from file** and enter the filename in the text box, or click the **Browse...** button to display a file selection dialog box.
3. To connect to a linked trader, select the **Link** option and choose the link from the drop-down list.

Click **OK** to connect to the trader. The contents of the current console window are updated to reflect the new trader.

**Note:** If you want to be connected to two or more traders at the same time, use the **Console/New Window** command to create a new console window, then select **Console/Connect** to connect the new window to another trader.

# The OMG Constraint Language

*This appendix provides the BNF specification of the CORBA standard constraint language (reproduced from Annex B in the OMG Trading Object Specification with the kind permission of the OMG). It is used for specifying both the constraint and preference expression parameters to various operations in the trader interfaces.*

**In this appendix**

This appendix contains the following sections:


| | |
|---|---|
| Introduction | page 114 |
| Language Basics | page 115 |
| The Constraint Language BNF | page 118 |

# Introduction

**Statement**

A statement in this language is an `Istring`. Other constraint languages may be supported by a particular trader implementation; the constraint language used by a client of the trader is indicated by embedding “`<<`*`Identifier major.minor`*`>>`” at the beginning of the string. If such an escape is not used, it is equivalent to embedding “`<<OMG 1.0>>`” at the beginning of the string.

# Language Basics

## Basic elements

Both the constraint and preference expressions in a query can be constructed from property names of conforming offers and literals. The constraint language in which these expressions are written consists of the following items (examples of these expressions are shown in square brackets below each bulleted item):

- Comparative functions:

  `==` (equality)

  `!=` (inequality)

  `>, >=, <, <=`

  `~` (substring match),

  `in` (element in sequence)

  The result of applying a comparative function is a `boolean` value. [“`Cost < 5`” implies only consider offers with a Cost property value less than 5; “`'Visa' in CreditCards`” implies only consider offers in which the CreditCards property, consisting of a set of strings, contains the string ’`Visa`’]

- Boolean connectives:

  `and`

  `or`

  `not`

  [“`Cost >= 2 and Cost <= 5`” implies only consider offers where the value of the Cost property is in the range 2 <= Cost <= 5]

- Property existence:

  `exist`

- Property names
- Numeric and string constants
- mathematical operators:

  `+, -, *, /`

[“`10 < 12.3 * MemSize + 4.6 * FileSize`” implies only consider offers for which the arithmetic function in terms of the value of the MemSize and FileSize properties exceeds 10]

- grouping operators:

  `( )`

Note that the keywords in the language are case sensitive.

---

**Precedence relations**

The following precedence relations hold in the absence of parentheses, in the order of highest to lowest:

```
() exist unary-minus
not
* /
+ -
~
in
== != < <= > >=
and
or
```

---

**Legal property value types**

While one can define properties of service types with arbitrarily complex IDL value types, only the following property value types can be manipulated using the constraint language:

- `boolean`, `short`, `unsigned short`, `long`, `unsigned long`, `float`, `double`, `char`, `Ichar`, `string`, `Istring`
- sequences of the above types

The `exist` operator can be applied to any property name, regardless of the property’s value type.

---

**Operator restrictions**

| | |
|---|---|
| `exist` | can be applied to any property |
| `~` | can only be applied if left operand and right operand are both strings or both Istrings |
| `in` | can only be applied if the left operand is one of the simple types described above and the right operand is a sequence of the same simple type |

| | |
|---|---|
| == | can only be applied if the left and right operands are of the same simple type |
| != | can only be applied if the left and right operands are of the same simple type |
| < | can only be applied if the left and right operands are of the same simple type |
| <= | can only be applied if the left and right operands are of the same simple type |
| > | can only be applied if the left and right operands are of the same simple type |
| >= | can only be applied if the left and right operands are of the same simple type |
| + | can only be applied to simple numeric operands |
| - | can only be applied to simple numeric operands |
| * | can only be applied to simple numeric operands |
| / | can only be applied to simple numeric operands |

The comparative functions <, <=, >, >= imply use of the appropriate collating sequence for characters and strings; TRUE is greater than FALSE for booleans.

## Representation of literals

| | |
|---|---|
| boolean | TRUE or FALSE |
| integers | sequences of digits, with a possible leading + or - |
| floats | digits with decimal point, with optional exponential notation |
| characters | char and Ichar are of the form '<char>', string and Istring are of the form '<char><char>+'; to embed an apostrophe in a string, place a backslash (\) in front of it; to embed a backslash in a string, use \\. |

# The Constraint Language BNF

**The constraint language proper in terms of lexical tokens**

```
<constraint>  :=  /* empty */
  |  <bool>

<preference>  :=  /* <empty> */
  |  min <bool>
  |  max <bool>
  |  with <bool>
  |  random
  |  first

<bool>  :=  <bool_or>

<bool_or>  :=  <bool_or> or <bool_and>
  |  <bool_and>

<bool_and>  :=  <bool_and> and <bool_compare>
  |  <bool_compare>

<bool_compare>  :=  <expr_in> == <expr_in>
  |  <expr_in> != <expr_in>
  |  <expr_in> <  <expr_in>
  |  <expr_in> <= <expr_in>
  |  <expr_in> >  <expr_in>
  |  <expr_in> >= <expr_in>
  |  <expr_in>

<expr_in>  :=  <expr_twiddle> in <Ident>
  |   <expr_twiddle>

<expr_twiddle>  :=  <expr> ~ <expr>
  |  <expr>

<expr>  :=  <expr> + <term>
  |  <expr> - <term>
  |  <term>

<term>  :=  <term> * <factor_not>
  |  <term> / <factor_not>
  |  <factor_not>

<factor_not>  :=  not <factor>
  |  <factor>
```

```
<factor>  :=  ( <bool_or> )
  |  exist <Ident>
  |  <Ident>
  |  <Number>
  |  - <Number>
  |  <String>
  |  TRUE
  |  FALSE
```

---

**"BNF" for lexical tokens up to character set issues**

```
<Ident>  :=  <Leader> <FollowSeq>

<FollowSeq>  :=  /* <empty> */
  |  <FollowSeq> <Follow>

<Number>  :=  <Mantissa>
  |  <Mantissa> <Exponent>

<Mantissa>  :=  <Digits>
  |  <Digits> .
  |  . <Digits>
  |  <Digits> . <Digits>

<Exponent>  :=  <Exp> <Sign> <Digits>

<Sign>  :=  +
  |  -

<Exp>  :=  E
  |  e

<Digits>  :=  <Digits> <Digit>
  |  <Digit>

<String>  :=  ' <TextChars> '

<TextChars>  :=  /* <empty> */
  |  <TextChars> <TextChar>

<TextChar>  :=  <Alpha>
  |  <Digit>
  |  <Other>
  |  <Special>

<Special>  :=  \\
  |  \'
```

## Character set issues

The previous BNF has been complete up to the non-terminals `<Leader>`, `<Follow>`, `<Alpha>`, `<Digit>`, and `<Other>`. For a particular character set, one must define the characters which make up these character classes.

Each character set which the trading service is to support must define these character classes. This annex defines these character classes for the ASCII character set.

```
<Leader>  :=  <Alpha>

<Follow>  :=  <Alpha>
  |  <Digit>
  |  _

<Alpha> is the set of alphabetic characters [A-Za-z]
<Digit> is the set of digits [0-9]
<Other> is the set of ASCII characters that are not <Alpha>,
   <Digit>, or <Special>
```

# Glossary

A

### ART

Adaptive Runtime Technology. IONA's modular, distributed object architecture, which supports dynamic deployment and configuration of services and application code. ART provides the foundation for IONA software products.

C

### CFR

See configuration repository.

### client

An application (process) that typically runs on a desktop and requests services from other applications that often run on different machines (known as server processes). In CORBA, a client is a program that requests services from CORBA objects.

### configuration

A specific arrangement of system elements and settings.

### configuration domain

Contains all the configuration information that Orbix ORBs, services and applications use. Defines a set of common configuration settings that specify available services and control ORB behavior. This information consists of configuration variables and their values. Configuration domain data can be implemented and maintained in a centralised Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organise ORBs into manageable groups, thereby bringing scalability and ease of use to the largest environments. See also configuration file and configuration repository.

### configuration file

A file that contains configuration information for Orbix components within a specific configuration domain. See also configuration domain.

### configuration repository

A centralised store of configuration information for all Orbix components within a specific configuration domain. See also configuration domain.

### configuration scope

Orbix configuration is divided into scopes. These are typically organized into a root scope and a hierarchy of nested scopes, the fully-qualified names of which map directly to ORB names. By organising configuration properties into various scopes, different settings can be provided for individual ORBs, or common settings for groups of ORB. Orbix services, such as the naming service, have their own configuration scopes.

### CORBA

Common Object Request Broker Architecture. An open standard that enables objects to communicate with one another regardless of what programming language they are written in, or what operating system they run on. The CORBA specification is produced and maintained by the OMG. See also OMG.

### CORBA objects

Self-contained software entities that consist of both data and the procedures to manipulate that data. Can be implemented in any programming language that CORBA supports, such as C++ and Java.

---

## D

### deployment

The process of distributing a configuration or system element into an environment.

---

## I

### IDL

Interface Definition Language. The CORBA standard declarative language that allows a programmer to define interfaces to CORBA objects. An IDL file defines the public API that CORBA objects expose in a server application. Clients use these interfaces to access server objects across a network. IDL interfaces are independent of operating systems and programming languages.

### IFR

See interface repository.

### IIOP

Internet Inter-ORB Protocol. The CORBA standard messaging protocol, defined by the OMG, for communications between ORBs and distributed applications. IIOP is defined as a protocol layer above the transport layer, TCP/IP.

### implementation repository

A database of available servers, it dynamically maps persistent objects to their server's actual address. Keeps track of the servers available in a system and the hosts they run on. Also provides a central forwarding point for client requests. See also location domain and locator daemon.

### IMR

See implementation repository.

### installation

The placement of software on a computer. Installation does not include configuration unless a default configuration is supplied.

### Interface Definition Language

See IDL.

### interface repository

Provides centralised persistent storage of IDL interfaces. An Orbix client can query this repository at runtime to determine information about an object's interface, and then use the Dynamic Invocation Interface (DII) to make calls to the object. Enables Orbix clients to call operations on IDL interfaces that are unknown at compile time.

### invocation

A request issued on an already active software component.

### IOR

Interoperable Object Reference. See object reference.

L

### location domain

A collection of servers under the control of a single locator daemon. Can span any number of hosts across a network, and can be dynamically extended with new hosts. See also locator daemon and node daemon.

### locator daemon

A server host facility that manages an implementation repository and acts as a control center for a location domain. Orbix clients use the locator daemon, often in conjunction with a naming service, to locate the objects they seek. Together with the implementation repository, it also stores server process data for activating servers and objects. When a client invokes on an object, the client ORB sends this invocation to the locator daemon, and the locator daemon searches the implementation repository for the address of the server object. In addition, enables servers to be moved from one host to another without disrupting client request processing. Redirects requests to the new location and transparently reconnects clients to the new server instance. See also location domain, node daemon, and implementation repository.

N

### naming service

An implementation of the OMG Naming Service Specification. Describes how applications can map object references to names. Servers can register object references by name with a naming service repository, and can advertise those names to clients. Clients, in turn, can resolve the desired objects in the naming service by supplying the appropriate name. The Orbix naming service is an example.

### node daemon

Starts, monitors, and manages servers on a host machine. Every machine that runs a server must run a node daemon.

O

### object reference

Uniquely identifies a local or remote object instance. Can be stored in a CORBA naming service, in a file or in a URL. The contact details that a client application uses to communicate with a CORBA object. Also known as interoperable object reference (IOR) or proxy.

**OMG**
Object Management Group. An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications, including CORBA. See www.omg.com.

**ORB**
Object Request Broker. Manages the interaction between clients and servers, using the Internet Inter-ORB Protocol (IIOP). Enables clients to make requests and receive replies from servers in a distributed computer environment. Key component in CORBA.

---

## P

**POA**
Portable Object Adapter. Maps object references to their concrete implementations in a server. Creates and manages object references to all objects used by an application, manages object state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. Can be transient or persistent.

**protocol**
Format for the layout of messages sent over a network.

---

## S

**server**
A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

---

## T

**trader service**
An implementation of the OMG CORBA Trading Object Service Specification. Facilitates the offering and the discovery of services. Other objects can use it to advertise their capabilities and to match their needs against those of advertised services.

# Index


**A**
add_link()
  usage 78
Admin interface
  in OMG stand-alone trader 10
administration
  setting Trader policies 56
advertising services 59

**B**
basic query 36
BNF specification, constraint language 113

**C**
connecting to Trader 64
constraint parameter, query() 37
constraints 40
  and IDL data types 40
  arithmetic expressions 42
  comparing property values 40
  connecting expressions together 41
  testing whether property exists 41
  when withdrawing offers 68
CORBA 113
CORBA Trading Object Service 1
creating service type properties 70

**D**
describe()
  usage 65
describe_type() 72
dynamic properties 73
  setting for a service offer 73, 75

**E**
exact_type_match policy
  usage 36
export()
  usage 64
  usage, with a dynamic property 73
export an offer 2
exporting service offers 59, 63

**F**
federated traders 34, 76
FollowOption type
  usage 77
full-service trader 10
fully_describe_type() 72

**H**
hop_count policy 49
how-many-offers parameter, query() 37

**I**
IDL data types and constaints 40
initialising service offer properties 61
iterator parameter, query() 38

**K**
kinds of traders 9

**L**
limits_applied parameter, query() 39
linked trader 10, 34
linked traders 76
  setting policies for 76
Link interface
  in OMG linked trader 10
LinkNameSeq sequence
  usage 79
list_links()
  usage 79
list_offers() 65
list_types() 72
Lookup interface
  in OMG query trader 9

**M**
mandatory properties 62
mask_type() 72
match_card policy 49

max_search_card policy 39
modes 62
  setting 71
modify()
  usage 66
modifying service offers 66
multiple service offers, withdrawing 68

## N

_narrow(), usage 64
narrowing object reference 64

## O

object reference, to Trader 36
object reference narrowing 64
OfferInfo structure
  usage 65
OfferIterator type
  usage 38
offers parameter, query() 38

## P

plugins:trader:direct_persitence 20
plugins:trader:iiop:port 20
policies 47
  set in a query 50
  setting for a trader 56
  setting for linked traders 76
  that affect queries 49
  that affect trader functionality 52
  using in a query 53
  you can set, with data types 54
preference parameter, query() 37
properties of service offers
  modifying 66
property modes, setting 71
PropertyNameSeq sequence
  usage 46, 66
PropertySeq sequence
  usage 61
PROP_MANDATORY mode 71
PROP_MANDATORY_READONLY mode 71
PROP_NORMAL mode 71
PROP_READONLY mode 71
PropStructSeq sequence
  usage 70
Proxy interface
  in OMG proxy trader 10
proxy trader 10

## Q

query
  basics 36
  forming constraints 40
  limiting returned properties 45
  results of 38
  setting policies 50
query()
  input parameters to 36
  output from 38
  usage 46
querying for service offers 33
query processing by Trader 34
query trader 9

## R

readonly properties 62
  changing 67
Register interface
  in OMG simple trader 9
remove_link()
  usage 79
remove_type() 72
resolve_initial_references()
  usage 64
return_card policy 49
return-properties parameter, query() 37

## S

search_card policy 49
searching for offers 47
selecting a service 38
sequence of offers 39
service offers 2
  exporting and managing 59
  exporting to Trader 63
  initialising properties 61
  modifying 66
  multiple withdraw 68
  querying for 33
  setting dynamic properties 73, 75
  sorting 43
  withdrawing from Trader 68
service type name parameter, query() 36
ServiceTypeNameSeq sequence
  usage 71

Service Type Repository 2, 72
ServiceTypeRepository interface
  usage 70
service types 2
  adding 71
  adding to Trader 70
  creating properties for 70
  definition 4
  hiding 72
  masking 72
  supertypes 71
set_def_follow_policy()
  usage 77
set_def_hop_count()
  usage 77
set_max_follow_policy()
  usage 77
set_max_hop_count()
  usage 77
set_max_link_follow_policy()
  usage 77
set_supports_modifiable_properties() 67
set_type_repos() 72
simple trader 9
sorting service offers 37, 43
  ascending 44
  by constraint 44
  decending 44
  in random order 44
SpecifiedProps union
  usage 45
stand-alone trader 10
supertypes, of service types 71
SupportAttributes
  supports_dynamic_properties policy 52
  supports_proxy_offers policy 52
  upports_modifiable_properties policy 52
SupportAttributes interface 52
supports_dynamic_properties policy 52
supports_modifiable_properties policy 52, 67
supports_proxy_offers policy 52

**T**
Trader
  as OMG linked trader 10
  connecting to 36, 64
  multiple traders 76
  policies to set 56
  processing a query 34
  usage by clients and servers 2
Trader Console 81
traders
  links between 76
traders, kinds of 9
typecodes 71
type_repos attribute 72
types of traders 9

**U**
unmask_type() 72
use_dynamic_properties policy 52
use_modifiable_properties policy 52
use_proxy_offers policy 52

**W**
withdraw()
  usage 68
withdrawing service offers 68
withdraw multiple offers 68
withdraw_using_constraint()
  usage 68