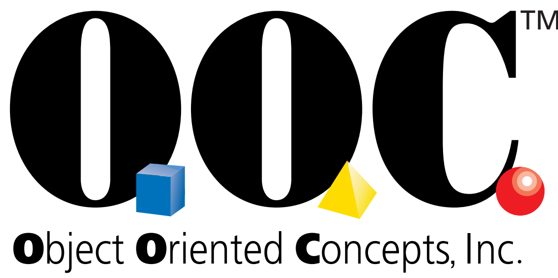
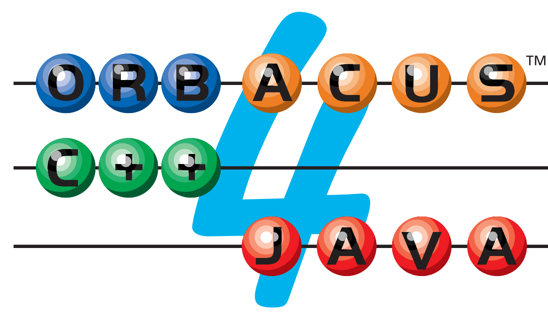


Advanced CORBA Programming with C++

Student Workbook

Version 1.1
Printed in USA
September 2000
Object Oriented Concepts, Inc.



Copyright © 2000 Object Oriented Concepts, Inc.

Parts of this material are adapted from M. Henning/S. Vinoski, *Advanced CORBA Programming with C++*. © 1999 Addison Wesley Longman, Inc. Reprinted by permission of Addison Wesley Longman.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in these notes and Object Oriented Concepts was aware of the trademark claim, the designations have been printed in initial caps or all caps.

Object Oriented Concepts, Inc. has taken care in the preparation of this material, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained therein.

Contents

Unit 1: The Portable Object Adapter (POA) 1-1

1.1 Interface Overview 1-2
1.2 Functions of a POA 1-4
1.3 Functions of a POA Manager 1-5
1.4 POA Manager State Transitions 1-6
1.5 Request Flow 1-8
1.6 Contents of an Object Reference 1-9
1.7 Policies 1-10
1.8 POA Policies 1-12
1.9 POA Creation 1-14
1.10 POA-to-POA Manager Relationship 1-17
1.11 The Life Span Policy 1-18
1.12 The ID Assignment Policy 1-19
1.13 The Active Object Map (AOM) 1-20
1.14 The ID Uniqueness Policy 1-21
1.15 The Servant Retention Policy 1-22
1.16 The Request Processing Policy 1-23
1.17 The Implicit Activation Policy 1-24
1.18 The Thread Policy 1-25
1.19 The Root POA Policies 1-26
1.20 Policy Creation 1-28
1.21 Creating Persistent Objects 1-30
1.22 Creating a Simple Persistent Server 1-32
1.23 Explicit Servant Activation 1-36
1.24 Object Creation 1-40
1.25 Destroying CORBA Objects 1-42
1.26 Deactivation and Servant Destruction 1-48

Unit 2: Advanced Uses of the POA 2-1

2.1 Pre-Loading of Objects 2-2

2.2 Servant Managers 2-3
2.3 Servant Activators 2-4
2.4 Implementing a Servant Activator 2-6
2.5 Use Cases for Servant Activators 2-8
2.6 Servant Manager Registration 2-9
2.7 Type Issues with Servant Managers 2-10
2.8 Servant Locators 2-11
2.9 Implementing Servant Locators 2-12
2.10 Use Cases for Servant Locators 2-14
2.11 Servant Managers and Collections 2-16
2.12 One Servant for Many Objects 2-18
2.13 The `Current` Object 2-20
2.14 Default Servants 2-22
2.15 Trade-Offs for Default Servants 2-24
2.16 POA Activators 2-25
2.17 Implementing POA Activators 2-26
2.18 Registering POA Activators 2-28
2.19 Finding POAs 2-30
2.20 Identity Mapping Operations 2-32

Unit 3: The Implementation Repository (IMR) 3-1

3.1 Purpose of an Implementation Repository 3-2
3.2 Binding 3-4
3.3 Indirect Binding 3-6
3.4 Automatic Server Start-Up 3-8
3.5 IMR Process Structure 3-9
3.6 Location Domains 3-10
3.7 The **`imradmin`** Tool 3-11
3.8 Server Execution Environment 3-12
3.9 Server Attributes 3-14
3.10 Getting IMR Status 3-16
3.11 IMR Configuration 3-18
3.12 IMR Properties 3-20
3.13 The **`mkref`** Tool 3-22

1. The Portable Object Adapter (POA)

Summary

This unit presents the Portable Object Adapter (POA) in detail and covers most of the functionality of the POA interfaces. It explains how to create persistent objects and how to link database state of objects to object references and servants. In addition, this unit covers how to support life cycle operations for CORBA objects.

Objectives

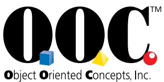
By the completion of this unit, you will be able to create servers that permit clients to create and destroy objects and that offer objects whose state is persistent. In addition, you will have a thorough understanding of the functionality of the POA, including how to control request flow, initialization, finalization, and memory management techniques.

Interface Overview

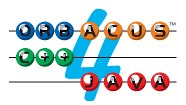
The Portable Object Adapter provides a number of core interfaces, all part of the **PortableServer** module:

- **POA**
- **POAManager**
- **Servant**
- POA Policies (seven interfaces)
- Servant Managers (three interfaces)
- **POACurrent**
- **AdapterActivator**

Of these, the first five are used regularly in almost every server; **POACurrent** and **AdapterActivator** support advanced or unusual implementation techniques.



© Copyright 2000 Object Oriented Concepts Inc.



1.1 Interface Overview

The interfaces to the POA are defined in IDL in the `PortableServer` module:

- `POA`

  The POA interface is the central server-side interface and contains quite a large number of operations. POAs are concerned with tasks such as keeping track of which servants are currently instantiated and their addresses in memory, the activation and deactivation of servants, the creation of object references, and various other life cycle issues (such as permitting a servant to be deleted at a time when no operation invocation is in progress in that servant).

- `POAManager`

  Conceptually a POA manager represents a transport endpoint that is used by one or more POAs. POA managers control the flow of requests into POAs.

- `Servant`

  The IDL `Servant` type is defined in the specification as follows:

```
module PortableServer {
    native Servant;
    // ...
};
```

  `native` is an IDL keyword that may be used only by OMG-defined specifications and ORB vendors. The `native` keyword indicates that the corresponding IDL construct is highly dependent on the target programming language and therefore does not have an IDL interface;

 © Copyright 2000 Object Oriented Concepts Inc.

instead, each language mapping must specify how the native type is represented as programming-language artifacts for a specific implementation language.

The `native` keyword was added to IDL after earlier attempts to specify the interface for servants were unsuccessful—as it turns out, to get elegant language mappings, servant implementations must use features that are specific to each programming language and cannot be expressed in IDL. (This is not surprising when you consider that servants straddle the boundary between language-independent IDL definitions and language-specific implementations.)

- POA Policies (seven interfaces)

  Each POA has seven policies that are associated with that POA when it is created (and remain in effect without change for the life time of each POA). The policies control aspects of the implementation techniques that are used by servants using that POA, such as the threading model and whether object references are persistent or transient.

- Servant Managers (three interfaces)

  Servant managers permit lazy instantiation of servants. Instead of requiring a server to keep a separate C++ object instantiated in memory for each CORBA object, servant managers allow servers to be written such that C++ instances are created on demand for only those servants that are actually used.

- `POACurrent`

  `POACurrent` is an object that provides information about a currently executing operation to the operation's implementation. This information is useful mainly for interceptors (which are used to implement functionality required by services such as the Transaction and Security Service).

- `AdapterActivator`

  An adapter activator is a callback object that permits you to create an object adapter on demand, when a request arrives for it, instead of forcing you keep all adapters active in memory at all times. Adapter activators are useful mainly to implement optimistic caching schemes, where entire groups of objects are instantiated in memory when a request for any one of the objects in the group is received.

© Copyright 2000 Object Oriented Concepts Inc.

Functions of a POA

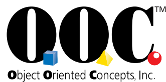
Each POA forms a namespace for servants.

All servants that use the same POA share common implementation characteristics, determined by the POA's policies. (The Root POA has a fixed set of policies.)

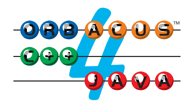
Each servant has exactly one POA, but many servants may share the same POA.

The POA tracks the relationship between object references, object IDs, and servants (and so is intimately involved in their life cycle).

POAs map between object references and the associated object ID and servants and map an incoming request onto the correct servant that incarnates the corresponding CORBA object.



2
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.2 Functions of a POA

The main purpose of a POA is to bridge the gap between the abstract notion of a CORBA object and the concrete representation of that object's behavior in form of a servant. In other words, POAs can be seen as a mapping mechanism that associates incoming requests with the correct C++ object in the server's memory.

A server can contain any number of POAs besides the Root POA (which is always present). Each POA, when it is created, is associated with a set of seven policies. These policies remain with the POA for its life time (that is, they become immutable once the POA is created). The policies determine the implementation characteristics of the servants associated with the POA, as well as aspects of object references (such as whether references are transient or persistent).

A POA can have any number of servants, but each servant belongs to exactly one POA.[1]

1. The specification (at least in theory) permits a single servant to be associated with more than POA at a time. However, this must be considered a defect because it creates a number of semantic conflicts; we *strongly* recommend that you never use the same servant with more than one POA.

 © Copyright 2000 Object Oriented Concepts Inc.

Functions of a POA Manager

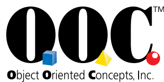
A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

A POA manager controls the flow of requests into one or more POAs.

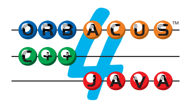
A POA manager is associated with a POA when the POA is created. Thereafter, the POA manager for a POA cannot be changed.

A POA manager is in one of four possible states:

- **Active**: Requests are processed normally
- **Holding**: Requests are queued
- **Discarding**: Requests are rejected with **`TRANSIENT`**
- **Inactive**: Requests are rejected; clients may be redirected to a different server instance to try again.



3
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.3 Functions of a POA Manager

A POA manager acts as a gate that controls the flow of requests to one or more associated POAs. Conceptually, a POA manager represents a transport endpoint (such as a host–port pair for TCP/IP). A POA is associated with its POA manager when the POA is created; thereafter, the POA manager for a POA cannot be changed.

A POA manager is in one of four possible states:

- `Active`

  This is the normal state in which the POA manager passes an incoming request to the target POA, which in turn passes the request to the correct servant.

- `Holding`

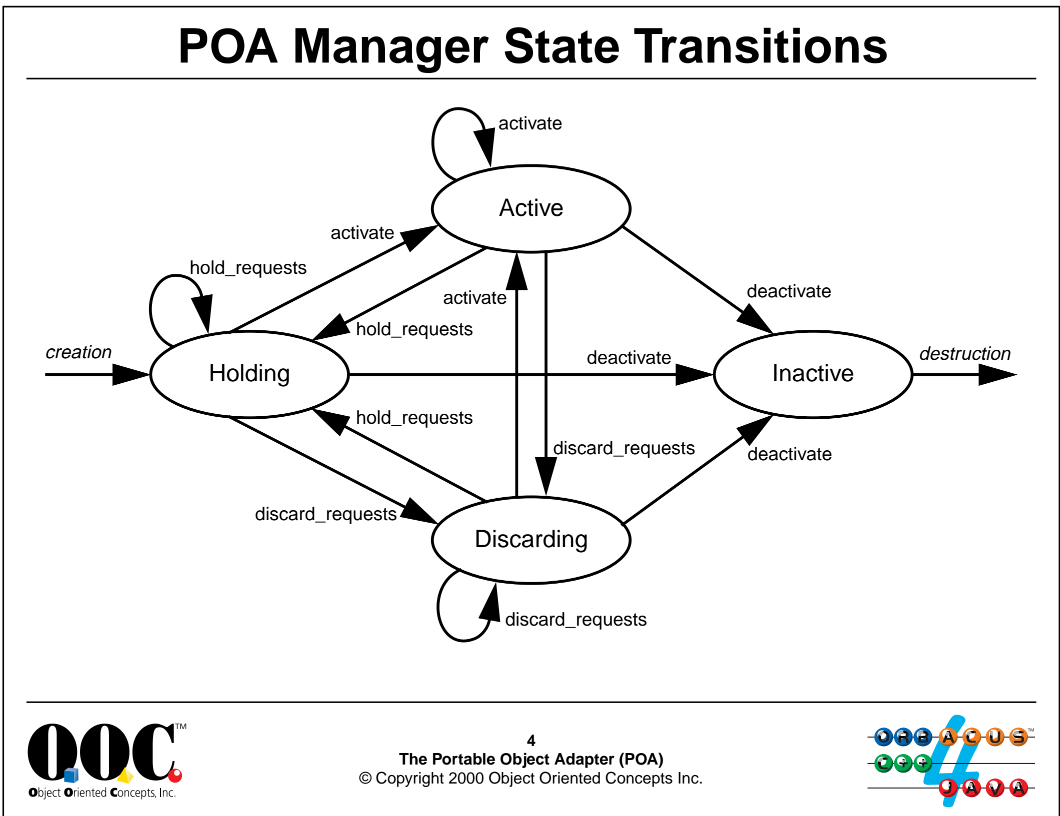
  In this state, the POA manager holds requests in a queue. Once the POA manager enters the active state, it passes the requests to their destination POAs.

- `Discarding`

  Incoming requests are rejected with a `TRANSIENT` exception. This exception indicates to the client that the request cannot be delivered right now, but that it may work if retransmitted again later.

- `Inactive`

  Requests are rejected; however, instead of raising an exception, the POA manager indicates to the client that the connection to the server is no longer usable. Depending on how the client is configured, this may result in an attempt by the client to locate a new instance of the server.



1.4 POA Manager State Transitions

The above diagram shows the possible state transitions. The arcs in the diagram are labeled with the corresponding IDL operation name. Initially, when it is first created, a POA manager starts out in the holding state. Before the ORB delivers requests to POAs associated with that POA manager, you must transition to the active state (see page 9-14).

Note that, once the POA manager enters the inactive state, it cannot be reactivated again and the only remaining transition is the destruction of the POA manager. POA managers are not destroyed explicitly; instead, a POA manager is destroyed once the last of its POAs is destroyed. You can freely transition among the remaining states by invoking the corresponding transition operation.

The IDL for the POAManager interface is as follows:

```
module PortableServer {
    // ...
    interface POAManager {]
        exception AdapterInactive {};

        enum State { HOLDING, ACTIVE, DISCARDING, INACTIVE };

        State   get_state();
        void    activate() raises(AdapterInactive);
        void    hold_requests(in boolean wait_for_completion)
                    raises(AdapterInactive);
        void    discard_requests(in boolean wait_for_completion)
                    raises(AdapterInactive);
        void    deactivate(
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
                        in boolean etherealize_objects,
                        in boolean wait_for_completion
                    ) raises(AdapterInactive);
    };
};
```

State get_state()

The get_state operation returns the current state of the of the POA manager as an enumerated value.

void activate() raises(AdapterInactive)

The activate operation transitions the POA manager into the active state. If the POA manager was previously in the holding state, the queued requests are dispatched in the order in which they were received. Attempts to activate an inactive POA manager raise AdapterInactive.

**void hold_requests(in boolean wait_for_completion)
raises(AdapterInactive)**

The hold_requests operation transitions the POA manager into the holding state. Incoming requests are queued up to some implementation-dependent limit.[2] If wait_for_completion is false, the operation returns immediately; otherwise, it queues incoming requests but waits until all currently executing requests have completed before returning control to the caller. If you call this operation with wait_for_completion set to true from within a servant that has a POA that is controlled by this POA manager, the operation raises BAD_INV_ORDER (because it would deadlock otherwise). Attempts to invoke this operation on an inactive POA manager raise AdapterInactive.

**void discard_requests(in boolean wait_for_completion)
raises(AdapterInactive)**

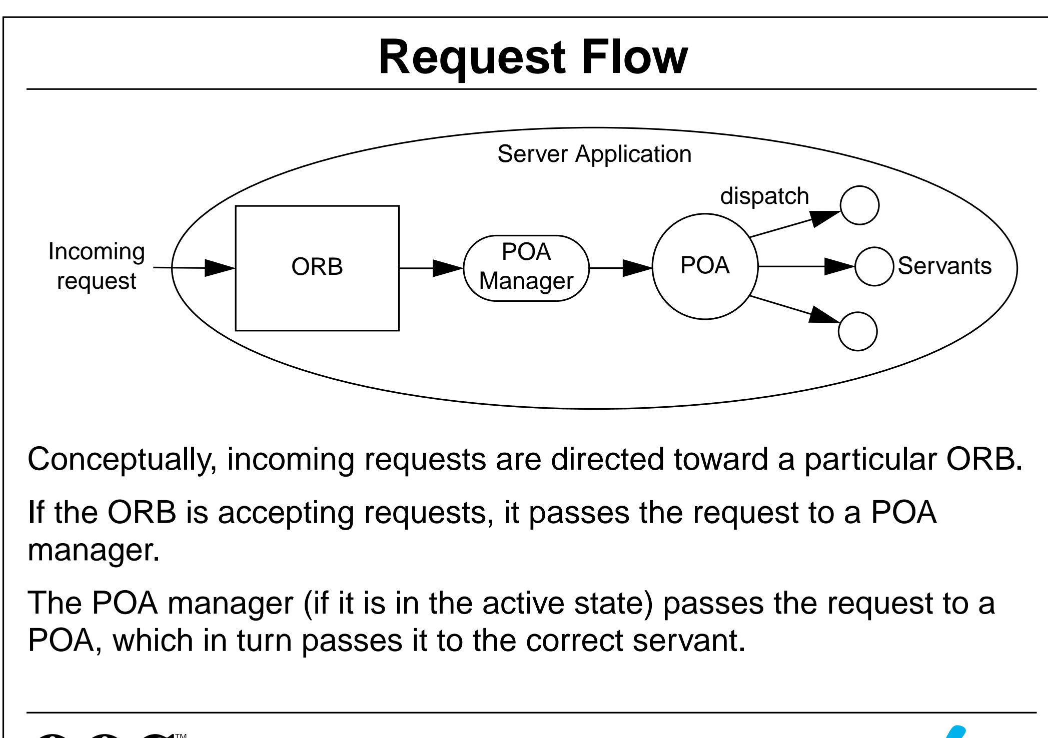
The discard_requests operation transitions the POA manager into the discarding state. Incoming requests are rejected with a TRANSIENT exception. The wait_for_completion parameter has the same semantics as for hold_requests. Attempts to invoke this operation on an inactive POA manager raise AdapterInactive.

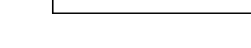
**void deactivate(
in boolean etherealize_objects,
in boolean wait_for_completion
) raises(AdapterInactive)**

The deactivate operation transitions the POA manager into the inactive state. Incoming requests are faced with a closed connection; the behavior that is visible to the client in this case depends on the type of object reference (transient or persistent) and the rebinding policy of the client. The wait_for_completion parameter has the same semantics as for discard_requests. The etherealize_objects parameter determines whether or not servant activators will be asked to destroy existing servants. (See page 2-4.) Attempts to invoke this operation on an inactive POA manager raise AdapterInactive.

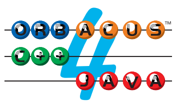
2. In ORBacus, the underlying transport is used as the queueing mechanism. This means that, due to TCP/IP flow control, leaving a POA manager in the holding state may cause flow control to affect the client (if transport buffers fill up completely) and cause the client to block in an operation until the POA manager transitions out of the holding state.

© Copyright 2000 Object Oriented Concepts Inc.





5
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.5 Request Flow

The general request flow into a server is shown above. Note that the diagram represents a conceptual view only. In the implementation, requests are not physically passed in this way for efficiency reasons.

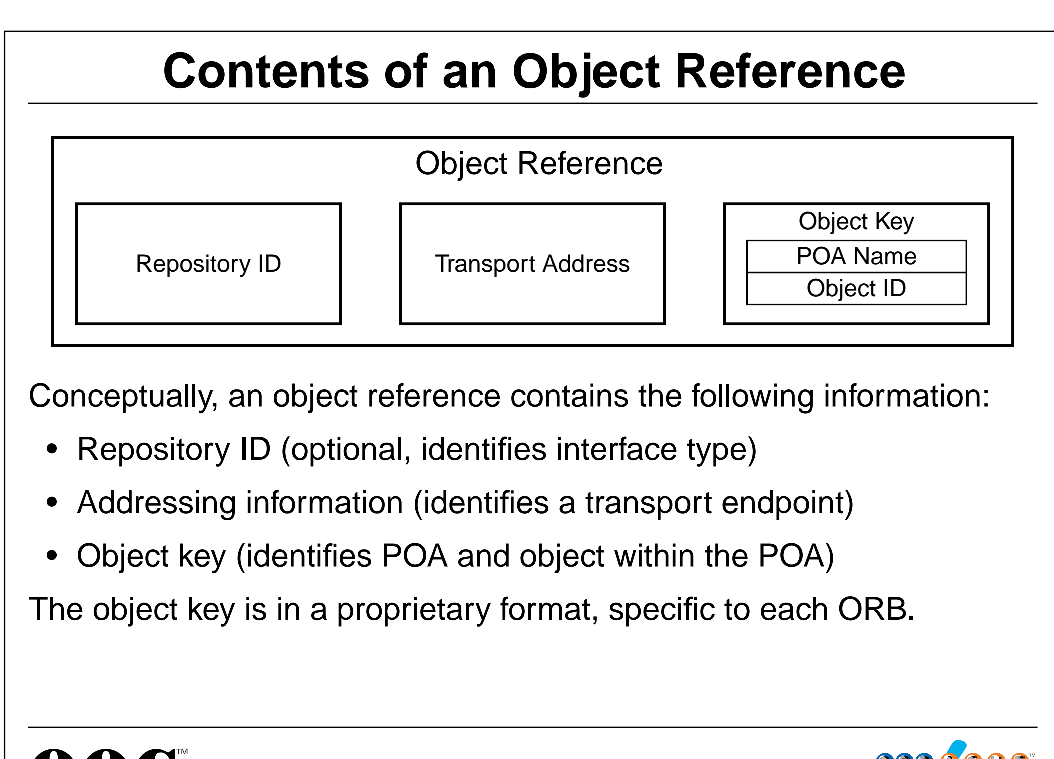
Conceptually, the request is directed toward a particular ORB within a server.[3] If the ORB is processing requests (that is, has created a dispatch loop by calling `ORB::run` or is dispatching requests explicitly via `ORB::work_pending` and `ORB::perform_work`), the request is passed to the POA manager.

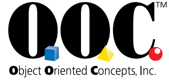
The POA manager determines whether the request is queued, discarded, or passed on. If the POA manager is in the active state, the request is passed to the correct POA.

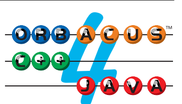
The POA determines the relationship between the CORBA reference that was used to make the call (and, therefore, the CORBA object represented by that reference) and the servant, and passes the request to the correct servant.

3. It is possible to instantiate multiple ORBs by calling `ORB_init` more than once with different ORB IDs. This is useful if you, for example, require different dispatch policies to be used for different objects.

 © Copyright 2000 Object Oriented Concepts Inc.







1.6 Contents of an Object Reference

For a server to correctly dispatch incoming requests to the correct servant, and for a client to correctly connect to the a server, an object reference must contain a minimum amount of information. In particular, it must contain an address (such as a host–port pair) that the client can use to contact the server, and it must contain information that, once a request is passed to the server, identifies the particular target object for an invocation.

As shown above, an object reference contains exactly that. The transport information, which (at least for IIOP) is standardized, enables the client to connect to the correct server. When a client sends an invocation to a particular server, it sends the object key with the request. The object key, internally, contains both a POA name and an object ID. The POA name enables the receiving ORB to identify the correct POA to pass the request to. In turn, the POA uses the object ID part of the object key to identify the specific servant that must handle the request.

Note that the object key is in a proprietary format, specific to each ORB vendor, and is never looked at except by the server that created it. Other clients and servers in a CORBA system treat the object key as an opaque blob of data.

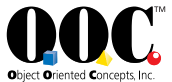
Policies

Each POA is associated with a set of seven policies when it is created.

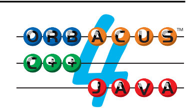
Policies control implementation characteristics of object references and servants.

The **CORBA** module provides a **Policy** abstract base interface:

```
module CORBA {
    typedef unsigned long PolicyType;

    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy  copy();
        void    destroy();
    };
    typedef sequence<Policy> PolicyList;
    // ...
};
```



7
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.7 Policies

Each POA is associated with a set of seven policies when the POA is created. The policies determine implementation characteristics of object references created by that POA, as well as of servants that are associated with that POA, such as the life time of references or the threading model to be used for request dispatch.

Policies are used in contexts other than the POA.[4] For that reason, the `CORBA` module provides an abstract base interface for policies from which all concrete policies are derived. The `Policy` interface provides only the basic features that are common to all policies. The `policy_type` attribute identifies the specific kind of policy. (Policy numbers are assigned by the OMG.)

The `copy` operation returns a (polymorphic) copy of a policy, and the `destroy` operation destroys a policy object. The specification requires you to call `destroy` on a policy object you have created before releasing its last reference.[5]

4. CORBA is using policies as a general abstraction for a quality-of-service (QoS) framework. For example, the real-time and messaging specifications both use policies to control various operational aspects of an ORB.
5. This is a rather useless requirement because policy objects are locality constrained (implemented as library objects) and the ORB can reclaim their resources automatically, when the last reference to a policy object is released. As a result, all ORBs we are aware of implement `destroy` as a no-op, so you don't suffer a resource leak if you do not call `destroy` before releasing a policy reference.

 © Copyright 2000 Object Oriented Concepts Inc.

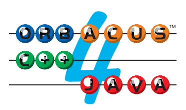
© Copyright 2000 Object Oriented Concepts Inc.

POA Policies

The **PortableServer** module contains seven interfaces that are derived from the **CORBA::Policy** interface:

- **LifespanPolicy**
- **IdAssignmentPolicy**
- **IdUniquenessPolicy**
- **ImplicitActivationPolicy**
- **RequestProcessingPolicy**
- **ServantRetentionPolicy**
- **ThreadPolicy**

8
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.8 POA Policies

The seven POA policies are all derived from the `CORBA::Policy` base interface. Each controls a different aspect of the implementation of an object. We briefly describe the purpose of each policy here and discuss it in more detail as we present the relevant implementation techniques.

- `LifespanPolicy`

  The life span policy controls whether a reference is transient or persistent. A transient reference works only for as long as its POA remains in existence and then becomes permanently non-functional. Therefore, transient references do not survive server shut-down. Persistent references continue to denote the same object even if the server is shut down and restarted.

- `IdAssignmentPolicy`

  The ID assignment policy controls whether the object ID that is part of the object key of every reference is created by the ORB or is provided by the application. Transient references usually use IDs that are created by the ORB, whereas persistent reference usually use IDs that are provided by the application.

- `IdUniquenessPolicy`

  The ID uniqueness policy determines how object references are mapped to C++ servants. You can choose to use one servant for each CORBA object that is provided by a server, or you can choose to incarnate multiple CORBA objects with the same C++ servant.

- `ImplicitActivationPolicy`

  The implicit activation policy determines whether a newly instantiated servant must be explicitly activated (registered with the ORB) or will be activated automatically when you first

 © Copyright 2000 Object Oriented Concepts Inc.

create a reference for the servant. Transient references usually use implicitly activated servants, whereas persistent references must use explicitly activated servants.

- `RequestProcessingPolicy`

  The request processing policy controls whether the POA maintains the object ID-to-servant associations for you (either to multiple servants or a single servant). You can also choose to maintain these associations yourself. Doing so is more work, but provides more powerful implementation choices.

- `ServantRetentionPolicy`

  The servant retention policy controls whether you keep your servants in memory at all times or instantiate them on demand, as requests arrive from clients.

- `ThreadPolicy`

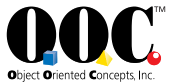
  The thread policy controls whether requests are dispatched on a single thread or multiple threads.

© Copyright 2000 Object Oriented Concepts Inc.

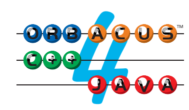
POA Creation

The POA interface allows you to create other POAs:

```
module PortableServer {
    interface POAManager;

    exception AdapterAlreadyExists {};
    exception InvalidPolicy { unsigned short index; };
    interface POA {
        POA create_POA(
                in string              adapter_name,
                in POAManager          manager,
                in CORBA::PolicyList   policies;
            ) raises(AdapterAlreadyExists, InvalidPolicy);
        readonly attribute the_POAManager;
        // ...
    };
    // ...
};
```



9
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.9 POA Creation

The POA interface provides an operation that creates POAs. (This is an example of the factory pattern, which we will examine in more detail in Section 1.24.) Initially, the only POA you have access to is the Root POA, returned by `resolve_initial_references`. In order to create other POAs, you call the `create_POA` operation on the Root POA or, once you have created other POAs, on a POA other than the Root POA.

The newly created POA becomes a child of the POA on which you invoke `create_POA`. In other words, if you have multiple POAs in a server, they are arranged into a hierarchy with the Root POA at the top. You control the shape of the hierarchy by choosing the POA on which you call `create_POA`.

Each POA has a name, controlled by setting the `adapter_name` parameter. You can choose any name you deem suitable, but you must ensure that no other sibling POA has the same name; otherwise, `create_POA` raises an `AdapterAlreadyExists` exception. As with a directory tree, the name of a POA must be unique only within the context of its parent, so you can have several POAs with the same name, as long as they have different parent POAs.

The `manager` parameter controls whether the new POA will use a separate POA manager or share a POA manger with other POAs: if you pass a nil reference, a new POA manager will be created for this POA; otherwise, you can pass a reference to an existing POA manager[6] and the new POA will be added to the list of POAs controlled by that manager.

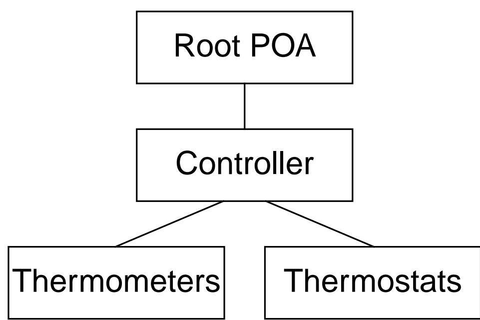
The `policies` parameter sets the policies to be applied to the new POA. The policy list can contain up to seven distinct POA policies. If you supply a value for the same policy more than

6. The `the_POAManager` read-only attribute on the POA interface returns the POA manager reference for a POA.

 © Copyright 2000 Object Oriented Concepts Inc.

once, or if one of the policies does not apply to the POA, the `create_POA` raises `InvalidPolicy`; the `index` member of the exception indicates the first policy that was found to be in error. You can create a POA with an empty policy sequence. If you do, each of the seven policies gets a default value.

For now, let us look at a simple example. The code that follows creates the following POA hierarchy:



For now, we will use the simplest way to create this hierarchy, using the default policies for all POAs, and using a separate POA manager for each POA.

```
// Initialize ORB and get Root POA
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
    PortableServer::POA::_narrow(obj);
assert(!CORBA::is_nil(root_poa));

// Create empty policy list
CORBA::PolicyList policy_list;

// Create Controller POA
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
                    "Controller",
                    PortableServer::POAManager::_nil(),
                    policy_list);

// Create Thermometer POA as a child of the Controller POA
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
                    "Thermometers",
                    PortableServer::POAManager::_nil(),
                    policy_list);

// Create Thermostat POA as a child of the Controller POA
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
                    "Thermostats",
                    PortableServer::POAManager::_nil(),
                    policy_list);
```

© Copyright 2000 Object Oriented Concepts Inc.

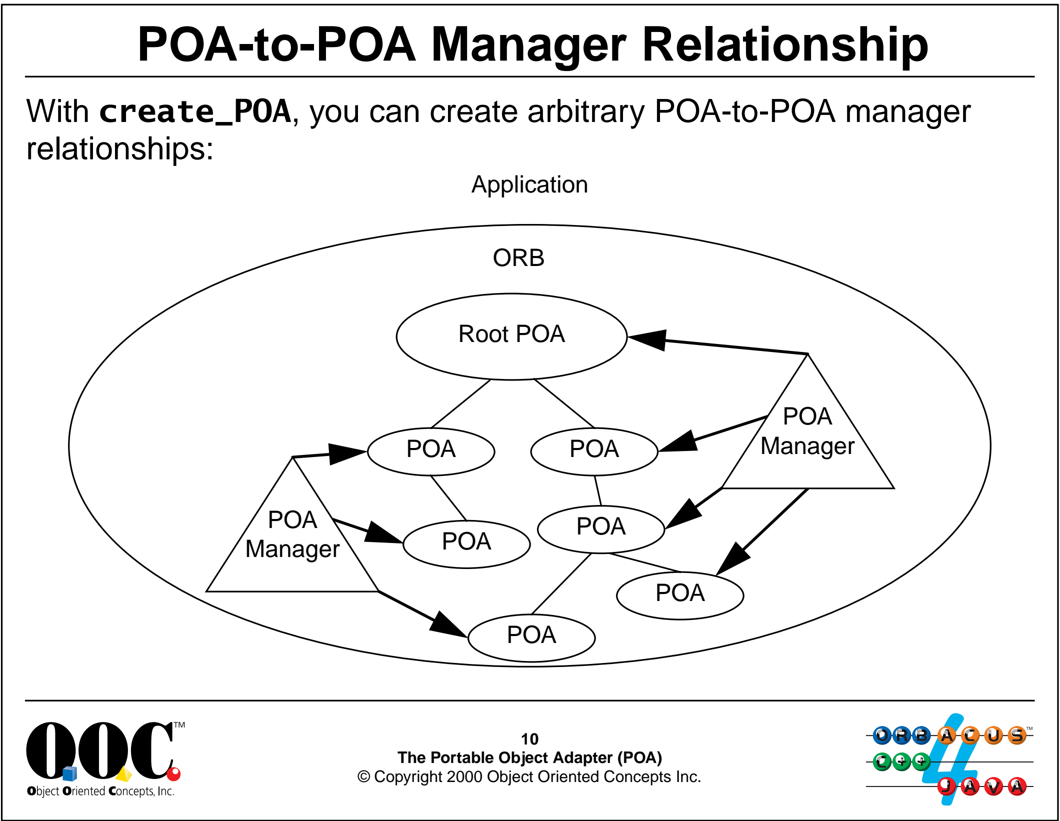
Because the code passes a nil reference as the `manager` parameter, each POA ends up with its own, separate POA manager; because the code passes an empty policy sequence as the `policies` parameter, each POA gets created with the default policies.

If we wanted to use the same POA manager for all four POAs, we could write the code as follows:

```
// Initialize ORB and get Root POA
PortableServer::POA_var root_poa = ...;

// Create empty policy list
CORBA::PolicyList policy_list;

// Get the Root POA manager
PortableServer::POAManager_var mgr = root_poa->the_POAManager();

// Create Controller POA, using the Root POA's manager
PortableServer::POA_var ctrl_poa = root_poa->create_POA(
                    "Controller",
                    mgr,
                    policy_list);

// Create Thermometer POA as a child of the Controller POA,
// using the Root POA's manager
PortableServer::POA_var thermometer_poa = ctrl_poa->create_POA(
                    "Thermometers",
                    mgr,
                    policy_list);

// Create Thermostat POA as a child of the Controller POA,
// using the Root POA's manager
PortableServer::POA_var thermostat_poa = ctrl_poa->create_POA(
                    "Thermostats",
                    mgr,
                    policy_list);
```

This code is almost identical to the preceding example. The only difference is that the code first gets a reference to the Root POA’s manager by reading the `the_POAManager` attribute of the Root POA, and then passes that manager’s reference to the three `create_POA` calls.

© Copyright 2000 Object Oriented Concepts Inc.



1.10 POA-to-POA Manager Relationship

`create_POA` is flexible enough to permit you to create arbitrary hierarchies of POAs, controlled by an arbitrary number of POA managers.

There are many reasons for using more than one POA. For example, it is common to use a separate POA for each interface that is provided by a server. (This technique is common in conjunction with servant managers.) The POA hierarchy also controls the order of destruction when you shut down the ORB: child POAs are destroyed before their parents. (This is useful to, for example, ensure that thermometers and thermostats are destroyed before the controller is destroyed.)

Similarly, you may want to use more than one POA manager in order to separately control the processing of requests for different groups of objects. (This is useful if you want to suspend request processing temporarily for one set of objects without affecting the processing of requests for another set of objects.)

The Life Span Policy

The life span policy controls whether references are transient or persistent. The default is **TRANSIENT**.

```
enum LifespanPolicyValue { TRANSIENT, PERSISTENT };

interface LifespanPolicy : CORBA::Policy {
    readonly attribute LifespanPolicyValue value;
};
```

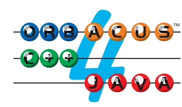
You should combine **PERSISTENT** with:

- **ImplicitActivationPolicy**: **NO_IMPLICIT_ACTIVATION**
- **IdAssignmentPolicy**: **USER_ID**

You should combine **TRANSIENT** with:

- **ImplicitActivationPolicy**: **IMPLICIT_ACTIVATION**
- **IDAssignmentPolicy**: **SYSTEM_ID**

11
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.11 The Life Span Policy

The life span policy controls whether references are transient or persistent. Transient references are usually created for objects that only exist temporarily, to support short-lived client–server interactions. On the other hand, persistent references are usually created if a server acts as a front end to some form of persistent store, such as a document retrieval service, which makes it desirable to pass out references to clients that can survive server shut-down.

Although the specification does not require it, you should combine the `PERSISTENT` life span policy with an implicit activation policy value of `NO_IMPLICIT_ACTIVATION`, and an ID assignment policy value of `USER_ID`.[7]

Note that, to create persistent objects, you must do a few things other than using the `PERSISTENT` life span policy. We discuss these details in Section 1.21.

7. While the other two combinations are legal, they do not have realistic use cases.

 © Copyright 2000 Object Oriented Concepts Inc.

The ID Assignment Policy

The ID assignment policy controls whether object IDs are created by the ORB or by the application. The default is **SYSTEM_ID**.

```
enum IdAssignmentPolicyValue { USER_ID, SYSTEM_ID };

interface IdAssignmentPolicy : CORBA::Policy {
    readonly attribute IdAssignmentPolicyValue value;
};
```

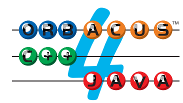
You should combine **USER_ID** with:

- **ImplicitActivationPolicy**: **NO_IMPLICIT_ACTIVATION**
- **LifespanPolicy**: **PERSISTENT**

You should combine **SYSTEM_ID** with:

- **ImplicitActivationPolicy**: **IMPLICIT_ACTIVATION**
- **LifespanPolicy**: **TRANSIENT**

12
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



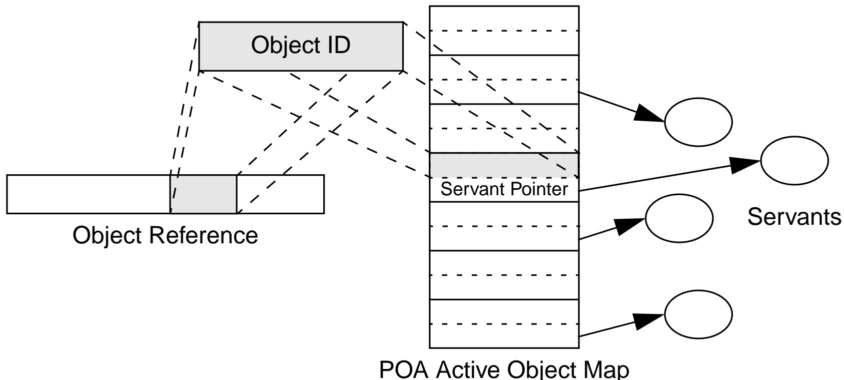
1.12 The ID Assignment Policy

The ID assignment policy controls whether object IDs (which end up being embedded into the object key inside references) are generated by the POA or supplied explicitly by the application. As we saw on page 1-9, the object ID ultimately identifies which servant is to handle an incoming request. This means that each ID denotes exactly one servant at a time.

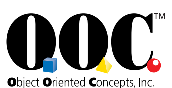
If the ID assignment policy is `SYSTEM_ID`, the POA automatically creates unique identifiers. If the policy is `USER_ID`, the POA rejects attempts to use the same ID a second time.

The Active Object Map (AOM)

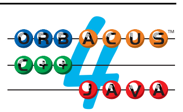
Each POA with a servant retention policy of **RETAIN** has an AOM. The AOM provides a mapping from object IDs to servant addresses:



The object ID is the index into the map and sent by clients with each incoming request as part of the object key.



13
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.13 The Active Object Map (AOM)

The POA maintains a lookup table known as the Active Object Map (AOM) that associates each object ID with the address of the corresponding servant in memory.[8] This means that each object ID must uniquely identify a servant; otherwise, the POA could end up with a single object ID designating two servants simultaneously and would not know which servant to give the request to.

8. You can change the setting of the servant retention policy to NON_RETAIN in order to provide your own AOM.

 © Copyright 2000 Object Oriented Concepts Inc.

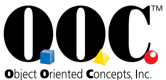
The ID Uniqueness Policy

The ID uniqueness policy controls whether a single servant can represent more than one CORBA object. The default is **UNIQUE_ID**.):

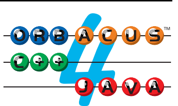
```
enum IdUniquenessPolicyValue { UNIQUE_ID, MULTIPLE_ID };

interface IdUniquenessPolicy : CORBA::Policy {
    readonly attribute IdUniquenessPolicyValue value;
};
```

- **UNIQUE_ID** enforces that no servant can appear in the AOM more than once.
- **MULTIPLE_ID** permits the same servant to be pointed at by more than one entry in the AOM.

For **MULTIPLE_ID**, an operation implementation can ask its POA for the object ID for the current invocation.



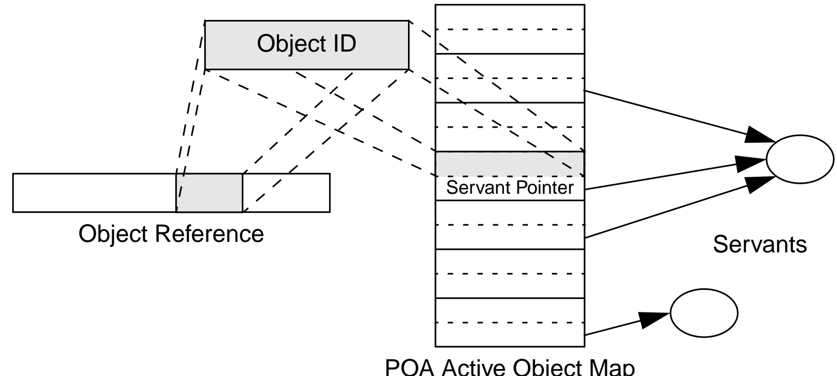
14
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.14 The ID Uniqueness Policy

You can chose to provide a separate C++ servant for each CORBA object by setting the ID uniqueness policy to `UNIQUE_ID`. This setting enforces that no servant can appear more than once in the AOM, so each CORBA object is incarnated by a separate servant. (This corresponds to the diagram shown on page 1-20.)

If you set the policy to `MULTIPLE_ID`, a single servant can incarnate more than one CORBA object simultaneously:



`MULTIPLE_ID` is useful if a server must provide access to a large number of CORBA objects with limited memory footprint. The cost of this increased scalability is that the identity of the CORBA object for a request is no longer implicit in the particular servant instance. Instead, the implementation of each operation must associate the object ID with the correct object state at run time.

The Servant Retention Policy

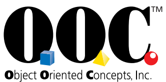
The servant retention policy controls whether a POA has an AOM. (The default is **RETAIN**).

```
enum ServantRetentionPolicyValue { RETAIN, NON_RETAIN };

interface ServantRetentionPolicy : CORBA::Policy {
    readonly attribute ServantRetentionPolicyValue value;
};
```

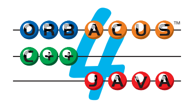
NON_RETAIN requires a request processing policy of **USE_DEFAULT_SERVANT** or **USE_SERVANT_MANAGER**.

With **NON_RETAIN** and **USE_DEFAULT_SERVANT**, the POA maps incoming requests to a nominated default servant.

With **NON_RETAIN** and **USE_SERVANT_MANAGER**, the POA calls back into the server application code for each incoming request to map the request to a particular servant.



15
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.15 The Servant Retention Policy

The servant retention policy controls whether an AOM is present (RETAIN) or absent (NON_RETAIN). Obviously, for the NON_RETAIN case, this deprives the POA of automatically mapping the object ID for an incoming request to the correct C++ servant. Depending on the setting of the request processing policy, the ORB either maps all requests to a nominated default servant (USER_DEFAULT_SERVANT) or it calls back into the application code to supply it with a servant for the request (USE_SERVANT_MANAGER).

Most servers that must provide access to a large number of CORBA objects simultaneously use the NON_RETAIN policy to limit the number of servants that must be in memory simultaneously.

 © Copyright 2000 Object Oriented Concepts Inc.

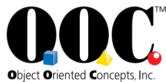
The Request Processing Policy

The request processing policy controls whether a POA uses static AOM, a default servant, or instantiates servants on demand. (The default is **USE_ACTIVE_OBJECT_MAP_ONLY**.)

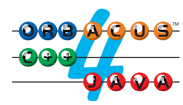
```
enum RequestProcessingPolicyValue {
            USE_ACTIVE_OBJECT_MAP_ONLY,
            USE_DEFAULT_SERVANT,
            USE_SERVANT_MANAGER
     };

interface RequestProcessingPolicy : CORBA::Policy {
    readonly attribute RequestProcessingPolicyValue value;
};
```

USE_DEFAULT_SERVANT requires **MULTIPLE_ID**.

USE_ACTIVE_OBJECT_MAP_ONLY requires **RETAIN**.



16
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.16 The Request Processing Policy

The most simple approach to implementing a server is to use a request processing policy of USE_ACTIVE_OBJECT_MAP_ONLY together with a servant retention policy of RETAIN. This combination uses a separate servant for each CORBA object (see page 1-20), and all servants are permanently in memory. If a request arrives for a an object ID that is not in the AOM, the request raises OBJECT_NOT_EXIST in the client.

The USE_DEFAULT_SERVANT policy can be combined with both RETAIN and NON_RETAIN policies:

- If used with NON_RETAIN, the POA passes all incoming requests to a nominated default servant (established by calling the set_servant operation on the POA).
- If used with RETAIN, the POA first looks for an instantiated servant with the given object ID. If one is found in the AOM, the request is passed to that servant; otherwise, the requests is passed to the default servant.

USE_SERVANT_MANAGER can be used with either RETAIN or NON_RETAIN:

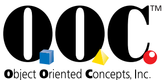
- If used with RETAIN and a request arrives for which no entry can be found in the AOM, the ORB makes a callback to an application-provided servant manager that is asked to instantiate a servant for the request. If the servant manager instantiates such a server, that servant is added to the AOM and the request is passed to the new servant; otherwise, the operation raises OBJECT_NOT_EXIST in the client.
- If used with NON_RETAIN, the ORB also calls back to an application-provided servant manager and dispatches the request if the servant manager returns a servant. However, the association between CORBA object and the servant is effective for only a single request.

The Implicit Activation Policy

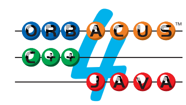
The implicit activation policy controls whether a servant can be activated implicitly or must be activated explicitly. (The default is **NO_IMPLICIT_ACTIVATION**).

```
enum ImplicitActivationPolicyValue {
        IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION
};

interface ImplicitActivationPolicy : CORBA::Policy {
    readonly attribute ImplicitActivationPolicyValue value;
};
```

- For **IMPLICIT_ACTIVATION** (which requires **RETAIN** and **SYSTEM_ID**), servants are added to AOM by calling **_this**.
- For **NO_IMPLICIT_ACTIVATION**, servants must be activated with a separate API call before you can obtain their object reference.



17
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.17 The Implicit Activation Policy

The code examples we have seen so far simply call `_this` on a newly instantiated servant in order to create a reference for the corresponding CORBA object. This technique works because the Root POA always uses the `IMPLICIT_ACTIVATION` policy. The first call to `_this` generates a new unique ID for the servant and adds the servant to the AOM. (The Root POA uses `SYSTEM_ID` and `RETAIN`).

However, as we will see in Section 1.22, `IMPLICIT_ACTIVATION` is useful only for transient objects. For persistent objects, you must use `NO_IMPLICIT_ACTIVATION` (because persistent objects almost always use `USER_ID`, for which `IMPLICIT_ACTIVATION` is illegal).

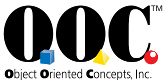
 © Copyright 2000 Object Oriented Concepts Inc.

The Thread Policy

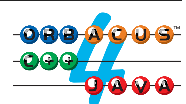
The thread policy controls whether requests are dispatched single-threaded (are serialized) or whether the ORB chooses a threading model for request dispatch. The default is **ORB_CTRL_MODEL**.

```
enum ThreadPolicyValue {
        ORB_CTRL_MODEL, SINGLE_THREAD_MODEL
};

interface ThreadPolicy : CORBA::Policy {
    readonly attribute ThreadPolicyValue value;
};
```

- **ORB_CTRL_MODEL** allows the ORB to chose a threading model. (Different ORBs will exhibit different behavior.)
- **SINGLE_THREAD_MODEL** serializes all requests on a per-POA basis.



18
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.18 The Thread Policy

The thread policy controls whether requests are dispatched on a single thread or multiple threads per POA. If you choose `SINGLE_THREAD_MODEL`, all invocations on that POA are serialized. If you choose `ORB_CTRL_MODEL`, the ORB is free to implement any threading strategy it prefers (including single-threaded dispatch).

Unfortunately, the specification is rather weak when it comes to controlling the threading model of a server, so ORBs from different vendors exhibit different behavior with respect to threading. For ORBacus, additional policies control a server's concurrency model with more precision. We cover these in Unit 25.

The Root POA Policies

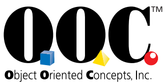
The Root POA has a fixed set of policies:

Life Span Policy	`TRANSIENT`
ID Assignment Policy	`SYSTEM_ID`
ID Uniqueness Policy	`UNIQUE_ID`
Servant Retention Policy	`RETAIN`
Request Processing Policy	`USE_ACTIVE_OBJECT_MAP_ONLY`
Implicit Activation Policy	`IMPLICIT_ACTIVATION`
Thread Policy	`ORB_CTRL_MODEL`

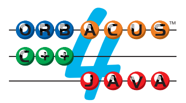
Note that the implicit activation policy does *not* have the default value.

The Root POA is useful for transient objects only.

If you want to create persistent objects or use more sophisticated implementation techniques, you must create your own POAs.



19
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.19 The Root POA Policies

The Root POA always has the policies shown above. The policies for the Root POA have the default values, except for the implicit activation policy (which has a default value of `NO_IMPLICIT_ACTIVATION`).

The Root POA uses `TRANSIENT` and `SYSTEM_ID`, so it is useful only for creation of transient references. You should therefore restrict use of the Root POA to short-lived temporary objects.

 © Copyright 2000 Object Oriented Concepts Inc.

Policy Creation

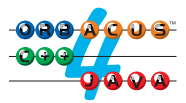
The POA interface provides a factory operation for each policy.

Each factory operation returns a policy with the requested value, for example:

```
module PortableServer {
    // ...
    interface POA {
        // ...
        LifespanPolicy create_lifespan_policy(
                                    in LifespanPolicyValue value
                              );
        // ...
};
```

You must call **destroy** on the returned object reference before you release it.

© Copyright 2000 Object Oriented Concepts Inc.



1.20 Policy Creation

The POA offers one factory operation for each of the seven policies:

```
module PortableServer {
    // ...
    interface POA {
        LifespanPolicy
            create_lifespan_policy(
                  in LifespanPolicyValue           value
            );

        IdAssignmentPolicy
            create_id_assignment_policy(
                  in IdAssignmentPolicyValue       value
            );

        IdUniquenessPolicy
            create_id_uniqueness_policy(
                  in IdUniquenessPolicyValue       value
            );

        ImplicitActivationPolicy
            create_implicit_activation_policy(
                  in ImplicitActivationPolicyValue value
            );
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
        RequestProcessingPolicy
            create_request_processing_policy(
                  in RequestProcessingPolicyValue  value
            );

        ServantRetentionPolicy
            create_servant_retention_policy(
                  in ServantRetentionPolicyValue   value
            );

        ThreadPolicy
            create_thread_policy(
                  in ThreadPolicyValue             value
            );
        // ...
    };
};
```

To create a new POA, you first create the required policies, add them to a policy list, and then call the `create_POA` operation with the policy list. Here is an example that creates a POA with the `PERSISTENT`, `USER_ID`, and `SINGLE_THREAD_MODEL` policy values, leaving the remaining policies at their defaults:

```
PortableServer::POA_var root_poa = ...;     // Get Root POA

CORBA::PolicyList pl;
pl.length(3);

pl[0] = root_poa->create_lifespan_policy(
            PortableServer::PERSISTENT
        );
pl[1] = root_poa->create_id_assignment_policy(
            PortableServer::USER_ID
        );
pl[2] = root_poa->create_thread_policy(
            PortableServer::SINGLE_THREAD_MODEL
        );

PortableServer::POA_var CCS_poa =
    root_poa->create_POA("CCS", nil_mgr, pl);

pl[0]->destroy();
pl[1]->destroy();
pl[2]->destroy();
```

Note that policies are copied when they are passed to `create_POA`, so destroying the policy objects after creating a POA does not affect the created POA. (Of course, if you need to create several POAs, you can keep the policy list around and reuse it for different calls to `create_POA`.)

NOTE: The above code is somewhat tedious if written in-line, so we suggest that you write a simple helper function that you can use to simplify your POA creation.

Creating Persistent Objects

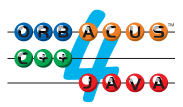
Persistent objects have references that continue to work across server shut-down and re-start.

To create persistent references, you must:

- use **PERSISTENT**, **USER_ID**, and **NO_IMPLICIT_ACTIVATION**
- use the same POA name and object ID for each persistent object
- link the object IDs to the objects' identity and persistent state
- explicitly activate each servant
- allow the server to be found by clients by
    - either specifying a port number (direct binding)
    - or using the implementation repository (IMR)

It sounds complicated, but it is easy!

21
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.21 Creating Persistent Objects

The server we developed in Unit 10 created transient references because we used the Root POA for all its objects. Obviously, this is useless if the server is to retain any state across shut-down and re-start. Frequently, we need to develop servers that offer objects whose state is stored in a database or a network, such that the server acts as a CORBA front end to the persistent state. (This is a very common scenario when adding legacy applications to a CORBA system.)

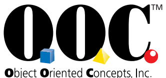
To create persistent references, you must take care of the steps shown above, which we will examine over the next few slides.

 © Copyright 2000 Object Oriented Concepts Inc.

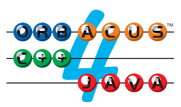
© Copyright 2000 Object Oriented Concepts Inc.

Creating a Simple Persistent Server

- Use a separate POA for each interface.
- Create your POAs with the **PERSISTENT**, **USER_ID**, and **NO_IMPLICIT_ACTIVATION** policies.
- Override the **_default_POA** method on your servant class. (*Always* do this for all POAs other than the Root POA. If you have multiple ORBs, do this even for the Root POA on non-default ORBs.)
- Explicitly activate each servant with **activate_object_with_id**.
- Ensure that servants have unique IDs per POA. Use some part of each servant's state as the unique ID (the identity).
- Use the identity of each servant as its database key.



22
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.22 Creating a Simple Persistent Server

The above list presents a "cook book" approach to implementing a simple server for persistent objects. Note that, for now, we restrict ourselves to a simple implementation model where each CORBA object has a separate C++ servant that is permanently in memory.

1.22.1 Creating Persistent POAs

The first step is to create persistent POAs. A simple approach is to use a separate POA for each interface that is supported by the server.[9] In the CCS system, we have three different interfaces and the controller object is a collection manager for thermometers and thermostats. We can reflect this relationship by creating three POAs, with the POA for the controller as the parent of the POAs for thermometers and thermostats:

```
PortableServer::POA_ptr
create_persistent_POA(
    const char *            name,
    PortableServer::POA_ptr parent)
{
    // Create policy list for simple persistence
    CORBA::PolicyList pl;
    pl.length(3);
    pl[0] = parent->create_lifespan_policy(
                PortableServer::PERSISTENT
```

9. For servers that keep all servants permanently in memory, a single persistent POA can be sufficient. However, once you use servant managers, having separate POAs simplifies servant instantiation and avoids object ID name clashes.

 © Copyright 2000 Object Oriented Concepts Inc.

```
                );
        pl[1] = parent->create_id_assignment_policy(
                    PortableServer::USER_ID
                );
        pl[2] = parent->create_thread_policy(
                    PortableServer::SINGLE_THREAD_MODEL
                );

        // Get parent POA's POA manager
        PortableServer::POAManager_var pmanager
            = parent->the_POAManager();

        // Create new POA
        PortableServer::POA_var poa =
            parent->create_POA(name, pmanager, pl);

        // Clean up
        for (CORBA::ULong i = 0; i < pl.length(); ++i)
            pl[i]->destroy();

        return poa._retn();
    }

    int
    main(int argc, char * argv[])
    {
        // ...

        PortableServer POA_var root_poa = ...;  // Get Root POA

        // Create POAs for controller, thermometers, and thermostats.
        // The controller POA becomes the parent of the thermometer
        // and thermostat POAs.
        PortableServer::POA_var ctrl_poa
            = create_persistent_POA("Controller", root_poa);
        PortableServer::POA_var thermo_poa
            = create_persistent_POA("Thermometers", ctrl_poa);
        PortableServer::POA_var tstat_poa
            = create_persistent_POA("Thermostats", ctrl_poa);

        // Create servants...

        // Activate POA manager
        PortableServer::POAManager_var mgr
            = root_poa->the_POAManager();
        mgr->activate();

        // ...
    }
```

© Copyright 2000 Object Oriented Concepts Inc.

Creating a Simple Persistent Server (cont.)

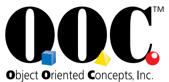
`PortableServer::ServantBase` (which is the ancestor of all servants) provides a default implementation of the **`_default_POA`** function.

The default implementation of **`_default_POA`** always returns the Root POA.

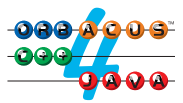
If you use POAs other than the Root POA, you *must* override **`_default_POA`** in the servant class to return the correct POA for the servant.

If you forget to override **`_default_POA`**, calls to **`_this`** and several other functions will return incorrect object references and implicitly register the servant with the Root POA as a transient object.

Always override **`_default_POA`** for servants that do not use the Root POA! If you use multiple ORBs, override it for *all* servants!



23
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.22.2 Overriding `_default_POA`

If you use POAs other than the Root POA, you must override the `_default_POA` operation inherited from `ServantBase`. An easy way to do this is to use a private static class member in your servant class, together with an modifier and an accessor. For example:

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(/* ... */)
    {
        if (CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa not set!"
        // ...
    }

    static void
    poa(PortableServer::POA_ptr poa)
    {
        if (!CORBA::is_nil(m_poa))
            throw "Thermometer_impl::m_poa set more than once!"
        m_poa = PortableServer::POA::_duplicate(poa);
    }

    static PortableServer::POA_ptr
    poa()
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
    {
        return m_poa;    // Note: no call to _duplicate() here!
    }

    virtual PortableServer::POA_ptr
    _default_POA()
    {
        return PortableServer::POA::_duplicate(m_poa);
    }
private:
    static PortableServer::POA_var m_poa;
    // ...
};

int
main(int argc, char * argv[])
{
    // ...
    PortableServer::POA_var thermo_poa
        = create_persistent_POA("Thermometers", ctrl_poa);
    Thermometer_impl::poa(thermo_poa);
    // ...
    PortableServer::POAManager_var mgr
        = root_poa->the_POAManager();
    mgr->activate();
    // ...
}
```

Note that this technique ensures that the POA for a servant class is set only once and that you cannot instantiate a servant before the POA has been set. The static `poa` accessor is useful if you want to get at the servant's POA before you have a servant instance, for example, when creating references with `create_reference_with_id` (see page 2-16). Note that this accessor does not call `_duplicate` on the returned reference, so you can make a call such as

```
Thermometer_impl::poa()->create_reference_with_id(...);
```

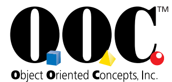
without having to release the returned reference.

© Copyright 2000 Object Oriented Concepts Inc.

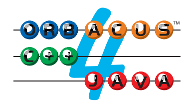
Creating a Simple Persistent Server (cont.)

To explicitly activate a servant and create an entry for the servant in the AOM, call **activate_object_with_id**:

```
typedef sequence<octet> ObjectId;
// ...
interface POA {
    exception ObjectAlreadyActive {};
    exception ServantAlreadyActive {};
    exception WrongPolicy {};
    void activate_object_with_id(
            in ObjectId id, in Servant p_servant
         ) raises(
             ObjectAlreadyActive,
             ServantAlreadyActive,
             WrongPolicy
         );
    // ...
};
```



24
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.23 Explicit Servant Activation

For POAs with the USER_ID policy, we must explicitly add servants to the POA's AOM by calling activate_object_with_id. The object ID is passed to the call as a sequence of octets, which allows us to use IDs of any length and data type. However, because dealing with octet sequences directly and because, in practice, IDs are frequently string values, the C++ mapping provides four helper functions to make it easier to convert object IDs to strings and vice versa:

```
namespace PortableServer {
    // ...
    char *           ObjectId_to_string(const ObjectId &);
    CORBA::WChar *   ObjectId_to_wstring(const Object Id &);

    ObjectId *       string_to_ObjectId(const char *);
    ObjectId *       wstring_to_ObjectId(const CORBA::WChar *);
}
```

Note that ObjectId_to_string and ObjectId_to_wstring will throw a BAD_PARAM exception if called with an object ID that is considered malformed by the ORB. (For example, for POAs with the SYSTEM_ID policy, object IDs usually must conform to an internal format.)

You can use any value as the object ID that uniquely (within its POA) identifies the target object. In addition, you must ensure that you use the same ID for the same logical CORBA object whenever you activate that object. For that reason, the best choice for an ID value is whatever bit of object state provides the object's identity. It might be a database row identifier, a social security

 © Copyright 2000 Object Oriented Concepts Inc.

number, or, in case of the CCS, an asset number. One convenient place to do this is a servant's constructor:

```
class Thermometer_impl : public virtual POA_CCS::Thermometer {
public:
    Thermometer_impl(CCS::AssetType anum /* , ... */);
    // ...
};

Thermometer_impl::
Thermometer_impl(CCS::AssetType anum /* , ... */)
{
    // ...
    ostrstream tmp;
    tmp << anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->activate_object_with_id(oid, this);
}
```

Merely instantiating the servant is then sufficient to ensure that it is activated correctly:

```
Thermometer_impl * t1 = new Thermometer_impl(22);
```

Of course, you can also activate the servant without doing this from a servant's constructor. (The advantage of this is that the POA policies and the way the servant is activated are unknown to the servant.)

NOTE: For multi-threaded servers, the POA dispatches requests as soon as you call `activate_object_with_id`. This means that, to avoid race conditions, you must call `activate_object_with_id` only once all other initialization for the servant is complete; otherwise, you end up with a race condition that can permit an incoming request to be dispatched before you have fully initialized the servant.

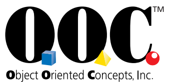
Creating a Simple Persistent Server (cont.)

A servant's object ID acts as a key that links an object reference to the persistent state for the object.

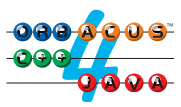
- For read access to the object, you can retrieve the state of the servant in the constructor, or use lazy retrieval (to spread out initialization cost).
- For write access, you can write the updated state back immediately, or when the servant is destroyed, or when the server shuts down.

When to retrieve and update persistent state is up to your application.

The persistence mechanism can be anything you like, such as a database, text file, mapped memory, etc.



25
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.23.1 Storing Persistent Servant State

Obviously, CORBA objects cannot be persistent unless you have some kind of database to hold their state. The object ID acts as the key that permits you to associate an object reference with a CORBA object and its servant, and the object ID acts as the database key to the object's state.

Exactly when and how to read and update the state for an object depends on your application. You can choose to hold all of a servant's state in memory (for example, as private data members of the servant) and to initialize all servants on construction, or you can use more sophisticated techniques, such as lazy initialization, depending on the space-time trade-offs for your application.

Similarly, for updates, you can choose to update the database immediately as soon as an update is made, or to cache updates until some timer expires, the servant is destroyed, or the server shuts down. The exact strategy as to how to update persistent servant state is largely determined by how much you are willing to sacrifice performance in order to reduce the risk of data loss in case of a crash.

As far as the ORB is concerned, the persistence mechanism you use is entirely up to you. The ORB merely enables you to correctly associate an incoming request with the persistent state of an object; the ORB does not provide persistence for the objects you create.

 © Copyright 2000 Object Oriented Concepts Inc.

Creating a Simple Persistent Server (cont.)

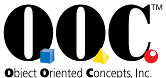
POAs using the **PERSISTENT** policy write addressing information into each object reference.

You must ensure that the same server keeps using the same address and port; otherwise, references created by a previous server instance dangle:

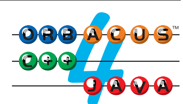
- The host name written into the each IOR is obtained automatically.
- You can assign a specific port number using the **-OAport** option.

If you do not assign a port number, the server determines a port number dynamically (which is likely to change every time the server starts up).

If you do not have a working DNS, use **-OAnumeric**. This forces dotted-decimal addresses to be written into IORs.



26
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.23.2 Fixing a Port Number

Because persistent references contain addressing information, you must ensure that a server for a set of persistent objects starts on the same machine every time, and uses a fixed port number; otherwise, references created by a previous run of the same server are no longer valid.

The **-OAport** option allows you to set a port number when you start a server. By keeping the port number constant for each server execution, you can ensure that references continue to denote the same objects across server start-up and shut-down.[10]

Occasionally, you may find that incorrectly configured DNS servers prevent clients from resolving a domain name that is embedded in an IOR. For such cases, you can use the **-OAnumeric** option to override the domain name with a dotted-decimal IP address.

NOTE: Options beginning with **-OA** are ORBacus-specific and processed by `ORB_init`.

If you use **-OAport**, it only affects the Root POA manager, so **-OAport** works correctly if all persistent POAs use the that POA manager. There is no portable way to assign a port to other POA managers because POA managers do not have a separate identity (such as a name) that you could use to associate the port number with. If you need to control the port number for multiple POA managers, you can use the (ORBacus-specific) `OB::POAManagerFactory` interface to create named POA managers. You can then use configuration properties to attach a different port number to each named POA manager. (See the ORBacus manual for details.)

10. For large installations with many servers, manual administration of port numbers becomes a burden. For such installations, you can use the Implementation Repository (IMR), which permits you (among other things) to have port numbers assigned dynamically without breaking existing references. (See Unit 3 for details.)

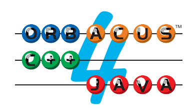
Object Creation

Clients create new objects by invoking operations on a factory. The factory operation returns the reference to the new object. For example:

```
exception DuplicateAssset {};

interface ThermometerFactory {
    Thermometer create(in AssetType n) throw(DuplicateAsset);
};

interface ThermostatFactory {
    Thermostat create(in AssetType n, in TempType t)
        throw(DuplicateAsset, Thermostat::BadTemp);
};
```

As far as the ORB is concerned, object creation is just another operation invocation.

27
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.24 Object Creation

If we want to permit clients to create new objects (instead of offering a fixed set of objects in the server), we can add one or more object factories to the system. An object factory contains one or more operations that return an object reference. As far as the ORB is concerned, nothing special is actually happening—the client simply invokes an operation that returns an object reference. However, the implementation of a factory operation creates a new CORBA object as a side effect (possibly creating a new record in a database for the object).

There are many ways to create factory interfaces. For example, you can have a single factory that offers a separate creation operation for each type of object, you can add factory operations to a collection manager interface (such as the `Controller` interface), or you can add creation operations that create more than one object at a time.

It is important that your creation operations completely initialize the new object. Designs that create an object with one operation and then initialize it with another are poor (because you permit objects to be created that are not fully initialized, and therefore run the risk of a client using an uninitialized object).

Implementing a create operation is almost trivial. You must return an object reference, so you must call `_this` on an instantiated servant.[11] That servant can be persistent or transient, as appropriate for your situation. Initialization of persistent object state (if any) is up to the factory implementation. For example:

11. See Section 2.2 for how to avoid the cost of instantiating a servant immediately.

 © Copyright 2000 Object Oriented Concepts Inc.

```
CCS::Thermometer_ptr
ThermometerFactory_impl::
create(AssetType n) throw(CORBA::SystemException)
{
    // Create database record for the new servant (if needed)
    // ...

    // Instantiate a new servant on the heap
    Thermometer_impl * thermo_p = new Thermometer_impl(n);

    // Activate the servant if it is persistent (and
    // activation is not done by the constructor)
    // ...

    return thermo_p->_this();
}
```

The main thing to note here is that you *must* instantiate the new servant on the heap by calling `new`. If you use a stack-based servant, you will leave a dangling pointer in the POA's AOM, with an eventual crash the most likely outcome.[12]

NOTE: The preceding code example drops the return value from `new`. This need not cause a memory leak, depending on how you write your code to shut down the server. (Recall that, if the POA uses the `RETAIN` policy, it uses an AOM, so the servant pointer is not lost, but stored in the AOM after registration of the servant.)

Of course, you can also choose to store the servant pointer explicitly in a data structure and explicitly delete the servant again later.

12. Note that there are several other reasons why servants should be instantiated on the heap. Rather than elaborate on these here, we suggest that you simply make it a habit to use heap-instantiated servants as a matter of principle.

© Copyright 2000 Object Oriented Concepts Inc.

Destroying CORBA Objects

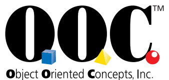
To permit clients to destroy a CORBA object, add a **destroy** operation to the interface:

```
interface Thermometer {
    // ...
    void destroy();
};
```

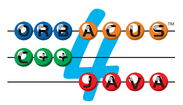
The implementation of destroy deactivates the servant and permanently removes its persistent state (if any).

Further invocations on the destroyed object raise **OBJECT_NOT_EXIST**.

As far as the ORB is concerned, **destroy** is an ordinary operation with no special significance.



28
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.25 Destroying CORBA Objects

To permit clients to destroy a CORBA object, simply add a `destroy` operation to the object's interface, as shown above.

Note that you could also add a `destroy` operation to the object's factory:

```
interface ThermometerFactory {
    Thermometer create(in AssetType n);
    void        destroy(in AssetType n);    // Not recommended!
};
```

There are two fundamental problems with this IDL:

- Placing the `destroy` operation on the object's factory means that you must specify the asset number of the object to be destroyed. However, this creates an additional error scenario because it allows a client to supply the asset number of a non-existent thermometer. You can deal with this by throwing a user exception, but it makes the design more complex than necessary. (Note that you could not use `OBJECT_NOT_EXIST` to indicate an invalid asset number because then the client would conclude that the *factory* does not exist instead of the object to be destroyed, so you would probably use `BAD_PARAM`.)
- Placing `destroy` on the factory places an additional burden on clients because, for each object, they have to remember which factory was used to create the object. For large systems with large numbers of factories and objects, this approach can rapidly become very unwieldy.

 © Copyright 2000 Object Oriented Concepts Inc.

Destroying CORBA Objects (cont.)

The POA holds a pointer to each servant in the AOM. You remove the AOM entry for a servant by calling **deactivate_object**:

```
interface POA {
    // ...
    void deactivate_object(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy);
};
```

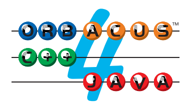
Once deactivated, further requests for the same object raise **OBJECT_NOT_EXIST** (because no entry can be found in the AOM).

Once the association between the reference and the servant is removed, you can delete the servant.

deactivate_object does not remove the AOM entry immediately, but waits until all operations on the servant have completed.

Never call **delete this;** from inside **destroy**!

29
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.25.1 Object Deactivation

In order to inform the ORB that an object is destroyed, you must remove its entry from the Active Object Map by calling `deactivate_object`. Once the entry for the servant is removed, further requests raise `OBJECT_NOT_EXIST` in the client because no entry with the corresponding object ID can be found in the AOM.

However, `deactivate_object` does *not* remove the entry for the specified object ID immediately. Instead, it marks the entry as to be removed once all operations (including `destroy` itself) have finished. This means that you *cannot* implement `destroy` like this:

```
void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostrstream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str()));
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);              // Fine
    delete this;                                // Disaster!!!
}
```

Calling `delete this;` is wrong because the POA invokes operations on `ServantBase` after the operation completes, which results in accessing deleted memory.

Destroying CORBA Objects (cont.)

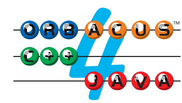
For multi-threaded servers, you must wait for all invocations to complete before you can physically destroy the servant.

To make this easier, the ORB provides a reference-counting mix-in class for servants:

```
class RefCountServantBase : public virtual ServantBase {
public:
                    ~RefCountServantBase();
    virtual void    _add_ref();
    virtual void    _remove_ref();
protected:
                    RefCountServantBase();
};
```

The ORB keeps a reference count for servants and calls **delete** on the servant once the reference count drops to zero.

30
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.25.2 Reference-Counted Servants

Because we cannot call `delete this;` from inside `destroy`, we end up with a problem: when and from where should we call `delete` on the servant after it is deactivated? (There is no convenient place where we could do this.) In addition, if a server is multi-threaded, we must worry about the fact that multiple operation invocations may be in the servant simultaneously and that we cannot delete the servant until after all these invocations have finished.

To get around this, the `PortableServer` namespace contains the `RefCountServantBase` reference-counting mix-in class. To use it, you simply inherit from it:

```
class Thermometer_impl :
        public virtual POA_CCS::Thermometer,
        public virtual PortableServer::RefCountServantBase {
// ...
};
```

Note that the constructor is protected, so you can only instantiate classes that are derived from `RefCountServantBase`. The constructor initializes a reference count for the servant to 1. The `_add_ref` and `_remove_ref` operations increment and decrement the reference count, respectively. In addition, `_remove_ref` calls `delete this;` once the reference count drops to zero.

By inheriting from `RefCountServantBase`, we can implement our `destroy` operation as follows, without having to artificially find a point of control at which it is safe to call `delete` (and without having to worry about still-executing operations in that servant):

 © Copyright 2000 Object Oriented Concepts Inc.

```
CCS::Thermometer_ptr
ThermometerFactory_impl::
create(AssetType n) throw(CORBA::SystemException)
{
    CCS::Thermometer_impl * thermo_p = new Thermometer_impl(...);
    // ...

    m_poa->activate_object_with_id(oid, thermo_p);
    thermo_p->_remove_ref();    // Drop ref count
    return thermo_p->_this();
}

void
Thermometer_impl::
destroy() throw(CORBA::SystemException)
{
    ostrstream tmp;
    tmp << m_anum << ends;
    PortableServer::ObjectId_var oid
        = PortableServer::string_to_ObjectId(tmp.str());
    tmp.rdbuf()->freeze(0);
    m_poa->deactivate_object(oid);              // Fine
}
```

The trick here is to realize that, in the factory `create` operation, we allocate the servant by calling `new`. This sets the reference count for the servant to 1. Calling `activate_object_with_id` increments the reference count a second time, so now it is 2. The call to `_remove_ref` decrements the reference count again, so now the only thing that keeps the servant alive in memory is its entry in the AOM. Every time the ORB invokes an operation, it calls `_add_ref` (which increments the reference count) and every time an operation completes, the ORB calls `_remove_ref` (which drops the reference count again). This means that, for as long as the CORBA object exists, the reference count on the servant will be at least 1, and greater than 1 during operation invocations.

When `destroy` is called by a client, the reference count for the servant is 2 on entry to the operation (assuming no other threads are executing inside the servant). `destroy` breaks the reference-to-servant association by calling `deactivate_object`. (Remember that this does not remove the servant's entry from AOM immediately, but only when all requests for the servant have completed.) `decativate_object` calls `_remove_ref`, which drops the reference count to 1. Once `destroy` returns control to the ORB, the ORB calls `_remove_ref` to balance the call to `_add_ref` it made when it invoked `destroy`. This drops the reference count to 0, removes the servant's entry from the AOM, and calls `delete` on the servant.

NOTE: We strongly encourage you to use `RefCountServantBase` for your servants if you support life cycle operations. This is of paramount importance for threaded servers, where you cannot otherwise be sure when it is safe to delete a servant. Reference-counted servants require you to use heap allocation, but you should be using heap-allocated servants anyway. The main reason for preferring heap allocation is that it is also required for more advanced implementation techniques that instantiate servants on demand. (See Section 2.2.)

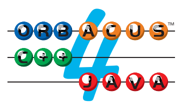
Destroying CORBA Objects (cont.)

Destroying a persistent object implies destroying its persistent state.

Generally, you cannot remove persistent state as part of **destroy** (because other operations executing in parallel may still need it):

- It is best to destroy the persistent state from the servant's destructor.
- The servant destructor also runs when the server is shut down, so take care to destroy the persistent state only after a previous call to **destroy**.
- Use a boolean member variable to remember a previous call to **destroy**.

31
The Portable Object Adapter (POA)
© Copyright 2000 Object Oriented Concepts Inc.



1.25.3 Destroying the Persistent State of an Object

For a single-threaded server, it is OK to destroy servant state immediately from within `destroy`. However, this technique does not generalize to multi-threaded servants because other requests may still be executing inside the servant in parallel and require the persistent state to remain. This means that it is best to destroy persistent servant state from the servant's destructor, when you can be sure that no other requests are still active in the servant.

However, take care not to destroy persistent state unconditionally. If you do, you will end up destroying the persistent state for every object when the server shuts down. This, of course, would be wrong because, for persistent objects, server shut-down does not imply that the server's CORBA objects are destroyed, only that their implementation is temporarily unavailable. An easy way to deal with this is to add a boolean member variable to each servant:

```
class Thermometer_impl :
        public virtual POA_CCS::Thermometer,
        public virtual PortableServer::RefCountServantBase {
public:
    Thermometer_impl(AssetType anum) :
        m_anum(anum), m_removed(false) { /* ... */ }
    ~Thermometer_impl();
    // ...
protected:
    AssetType m_anum;
    bool      m_removed;
};
```

 © Copyright 2000 Object Oriented Concepts Inc.

In the `destroy` member function, set the `m_removed` member to true and then check it in the destructor:

```
Thermometer_impl::
~Thermometer_impl()
{
    if (m_removed) {
        // Destroy persistent state for this object...
    }
    // Release whatever other resources (not related to
    // persistent state) were used...
}
```

NOTE: Using a member variable to record object destruction can also be valuable for multi-threaded servers. Because `deactivate_object` does not immediately remove the entry for the servant from the AOM, it is possible for hyperactive clients to keep a servant pinned in memory by continuously sending requests, thereby preventing the servant from ever becoming idle so it could be destroyed.

If this behavior is unacceptable in your application and you want to force servants to be destroyed as soon as all requests have completed and prevent entry of new requests into the servant, you can test the m_removed member on entry to every operation and throw `OBJECT_NOT_EXIST` if the object was previously destroyed:

```
class Thermometer_impl :
    public virtual POA_CCS::Thermometer,
    public virtual PortableServer::RefCountServantBase {
protected:
    void check_exists() throw(CORBA::OBJECT_NOT_EXIST) {
            if (m_removed)
                throw CORBA::OBJECT_NOT_EXIST();
        }
// ...
};

// ...

void
Thermometer_impl::
some_operation() throw(CORBA::SystemException)
{
    check_exists();
    // remainder of operation here...
}
```

This technique is particularly useful in combination with a servant locator (see Section 2.9) because servant locators permit you to automatically make the check on every operation invocation.

Deactivation and Servant Destruction

The POA offers a **destroy** operation:

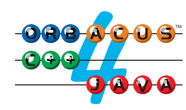
```
interface POA {
    // ...
    void destroy(
            in boolean etherealize_objects,
            in boolean wait_for_completion
         );
};
```

Destroying a POA recursively destroys its descendant POAs.

If **wait_for_completion** is true, the call returns after all current requests have completed and all POAs are destroyed. Otherwise, the POA is destroyed immediately.

Calling **ORB::shutdown** or **ORB::destroy** implicitly calls **POA::destroy** on the Root POA.

© Copyright 2000 Object Oriented Concepts Inc.



1.26 Deactivation and Servant Destruction

You can explicitly destroy a POA by calling its `destroy` operation. Doing this recursively destroys any descendent POAs before destroying the parent POA. In other words, `destroy` does a depth-first traversal of the POA hierarchy; the order in which siblings are destroyed is undefined.

The `wait_for_completion` flag determines whether the invocation waits until current requests have finished executing and destruction is complete. We strongly recommend that you do not use a `wait_for_completion` flag of false unless you are prepared to have currently executing operations fail in unpredictable ways (because all their POA-related calls will raise `OBJECT_NOT_EXIST` if you do this).

The `etherealize_objects` parameter is relevant to servant activators. (See Section 2.3.) It determines whether servants will be etherealized as part of the call or not.

If you call `ORB::shutdown` or `ORB::destroy`, the ORB makes an implicit call to `POA::destroy` on the Root POA. (`ORB::shutdown` passes the value of its `wait_for_completion` flag through to `POA::destroy`.)

Note that calling `destroy` on a POA implicitly invokes `_remove_ref` on every servant in the AOM. This means that each servant's destructor runs once operations have drained out of the servant.

 © Copyright 2000 Object Oriented Concepts Inc.

2. Advanced Uses of the POA

Summary

This unit covers advanced aspects of the POA that permit you to exercise tight control over the trade-offs in performance, scalability, and memory consumption of a server.

Objectives

By the completion of this unit, you will be able to create sophisticated servers that scale to unlimited numbers of objects. In addition, you will have an appreciation of advanced caching techniques, such as eviction of servants and optimistic caching.

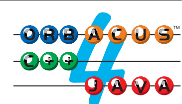
Pre-Loading of Objects

The **USE_ACTIVE_OBJECT_MAP_ONLY** requires one servant per CORBA object, and requires all servants to be in memory at all times.

This forces the server to pre-instantiate all servants prior to entering its dispatch loop:

```
int main(int argc, char * argv[])
{
    // Initialize ORB, create POAs, etc...
    // Instantiate one servant per CORBA object:
    while (database_records_remain) {
        // Fetch record for a servant...
        // Instantiate and activate servant...
    }
    // ...
    orb->run();     // Start dispatch loop
    // ...
}
```

1
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.1 Pre-Loading of Objects

So far, all the server code we have seen used the `USE_ACTIVE_OBJECT_MAP_ONLY` policy for its POAs. As a result, we end up with a design that requires a separate servant for each CORBA object and forces us to keep all servants in memory at all times. This is a perfectly acceptable and sensible implementation choice, provide that you can afford it. This will be the case if:

- The number of objects is small enough to fit into memory without increasing the working set unacceptably. Generally, this means that you can either have a fairly small number of large objects or a fairly large number of small objects.
- The time taken to iterate over the database and to instantiate all the servants is acceptable. Generally, this means that initialization of each servant has to be fast, otherwise the server may take minutes or hours to enter its dispatch loop, which is usually unacceptable (especially if servers are started automatically as requests arrive—see Unit 3).

To get around this problem, the POA offers servant managers.

 © Copyright 2000 Object Oriented Concepts Inc.

Servant Managers

Servant managers permit you to load servants into memory on demand, when they are needed.

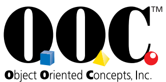
Servant managers come in two flavors:

- **ServantActivator** (requires the **RETAIN** policy)

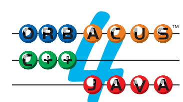
  The ORB makes a callback the first time a requests arrives for an object that is not in the AOM. The callback returns the servant to the ORB, and the ORB adds it to the AOM.

- **ServantLocator** (requires the **NON_RETAIN** policy)

  The ORB makes a callback every time a request arrives. The callback returns a servant for the request. Another callback is made once the request is complete. The association between request and servant is in effect only for the duration of single request.



2
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.2 Servant Managers

Servant managers allows you to bring a servant into memory when a request for it arrives, instead of having to keep all servants in memory at all times, just in case they are needed. Servant managers come in two flavors, both derived from a common base interface:

```
module PortableServer {
    // ...

    interface ServantManager {};

    interface ServantActivator : ServantManager {
        // ...
    };

    interface ServantLocator : ServantManager {
        // ...
    };
};
```

Servant activators require the `RETAIN` policy to bring servants into memory on demand (and leave them in the AOM thereafter). Servant locators require the `NON_RETAIN` policy and only provide the association between a request and its servant for the duration of a single request.

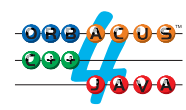
Servant locators require more implementation work from the application than servant activators, but are more powerful and offer more aggressive scalability options.

Both servant activators and servant locators also require the `USE_SERVANT_MANAGER` policy.

Servant Activators

```
exception ForwardRequest {
    Object forward_reference;
};
interface ServantManager {};
interface ServantActivator : ServantManager {
    Servant incarnate(
                in ObjectId oid;
                in POA      adapter
            ) raises(ForwardRequest);

    void    etherealize(
                in ObjectId oid,
                in POA      adapter,
                in Servant  serv,
                in boolean  cleanup_in_progress,
                in boolean  remaining_activations
            );
};
```

© Copyright 2000 Object Oriented Concepts Inc.



2.3 Servant Activators

Servant activators have the IDL interface shown above. As the server programmer, you create an implementation of this interface and register it with a POA. (See page 2-9.) The corresponding POA must use the RETAIN and USE_SERVANT_MANAGER policies.

Once an activator is registered, if a request comes in for which the ORB cannot locate an entry in the AOM, instead of raising OBJECT_NOT_EXIST in the client, it first calls the incarnate operation, passing the object ID and the POA for the request. Now incarnate is in control and can use the information passed to it to locate the state of a servant for the request and instantiate the servant:

- If incarnate can instantiate a servant, it returns the servant to the ORB, which then adds the object ID and servant to its AOM and dispatches the request as usual. (This is, of course, completely transparent to the client.)
- If incarnate cannot locate the state for a servant with the given object ID (probably because the corresponding object was destroyed previously), it simply raises OBJECT_NOT_EXIST, which is propagated back to the client.

The Request exception presents a third option to return control to the ORB. If incarnate raises this exception, it returns an object reference. This indicates to the ORB that the request could not be handled by this POA, but that retrying the request at the returned IOR may work. The ORB returns the IOR to the client, and the client-side ORB (transparently to the application) then tries to dispatch the request using the new IOR. This mechanism can be used to support object migration (albeit at the cost of slower request dispatch).

 © Copyright 2000 Object Oriented Concepts Inc.

The `etherealize` operation is invoked by the ORB to instruct you that it no longer requires the servant and gives you a chance to reclaim the servant.

The ORB invokes `etherealize` in the following circumstances:

- `deactivate_object` was called for an object represented by the servant
- `POAManager::deactivate` was called on a POA manager with active servants
- `POA::destroy` was called with `etherealize_objects` set to true
- `ORB::shutdown` or `ORB::destroy` were called

The `cleanup_in_progress` parameter is true if the call to `etherealize` resulted because of a call to `POAManager::deactivate`, `POA::destroy`, `ORB::shutdown`, or `ORB::destroy`; otherwise, the parameter is false. This allows you to distinguish between normal deactivation and application (or POA) shut-down.

The `remaining_activations` parameter is false if this servant is used to only represent a single CORBA object; otherwise, if true, the parameter indicates that this servant still represents other CORBA objects and therefore should not be physically deleted. (See page 2-18 for how to map multiple CORBA objects onto a single C++ servant.)

NOTE: The ORB removes the entry for the passed object ID from the AOM *before* it calls `etherealize`.

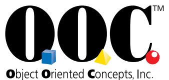
Implementing a Servant Activator

The implementation of **incarnate** is usually very similar to a factory operation:

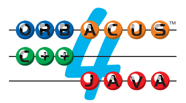
1. Use the object ID to locate the persistent state for the servant.
2. If the object ID does not exist, throw **OBJECT_NOT_EXIST**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **etherealize** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. If **remaining_activations** is false, call **_remove_ref** (or call **delete**, if the servant is not reference-counted).



© Copyright 2000 Object Oriented Concepts Inc.



2.4 Implementing a Servant Activator

To implement a servant activator, you must derive your implementation from `PortableServer::ServantActivator`. This means your servant activator is itself a servant for the `ServantActivator` interface:

```
class Activator_impl :
    public virtual POA_PortableServer::ServantActivator {
public:
    virtual PortableServer::Servant
    incarnate(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa
    ) throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

    virtual void
    etherealize(
        const PortableServer::ObjectId & oid,
        PortableServer::POA_ptr          poa,
        PortableServer::Servant          serv,
        CORBA::Boolean                   cleanup_in_progress,
        CORBA::Boolean                   remaining_activations
    ) throw(CORBA::SystemException);
};
```

 © Copyright 2000 Object Oriented Concepts Inc.

The implementation of `incarnate` can be outlined as follows:

```
PortableServer::Servant
Activator_impl::
incarnate(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}
```

The implementation of `etherealize` is usually very simple. If your servants are implemented to write updates directly to the database with each update operation, there is no persistent state to be finalized. Otherwise, if updates are held in memory and written to the database only when the servant is destroyed, `etherealize` must write the database state before destroying the servant:

```
void
Activator_impl::
etherealize(
    const PortableServer::ObjectId & oid,
    PortableServer::POA_ptr          poa,
    PortableServer::Servant          serv,
    CORBA::Boolean                   cleanup_in_progress,
    CORBA::Boolean                   remaining_activations
) throw(CORBA::SystemException)
{
    // Write updates (if any) for this object to database and
    // clean up any resources that may still be used by the
    // servant (or do this from the servant destructor)...
    if (!remaining_activations)
       serv->_remove_ref(); // Or delete serv, if not ref-counted
}
```

© Copyright 2000 Object Oriented Concepts Inc.

Use Cases for Servant Activators

Use servant activators if:

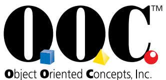
- you cannot afford to instantiate all servants up-front because it takes too long

  A servant activator distributes the cost of initialization over many calls, so the server can start up quicker.

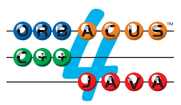
- clients tend to be interested in only a small number of servants over the period the server is up

  If all objects provided by the server are eventually touched by clients, all servants end up in memory, so there is no saving in that case.

Servant activators are of interest mainly for servers that are started on demand.



5
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.5 Use Cases for Servant Activators

Servant activators are useful mainly because they incur the cost of servant initialization only for those servants that are actually used. In addition, the initialization cost is distributed over many calls instead of incurred up-front, so the server starts up more quickly. This is important if you arrange for servers to be started by an implementation repository when a client sends a request (see Unit 3), which works well only if the server comes up quickly.

If only a subset of the server's objects are touched by clients, servant activators offer some saving in memory consumption because only those servants that are actually used need be instantiated. However, if a server runs for a long time and clients, over time, end up using every object offered by the server, all servants eventually end up in memory, so there is no saving.

NOTE: You can call `deactivate_object` to get rid of unused servants in order to limit the memory consumption of the server. This idea generalizes to that of a servant cache and is described in more detail in Henning & Vinoski as the Evictor Pattern. However, such servant caches are best implemented with servant locators instead of servant activators.

 © Copyright 2000 Object Oriented Concepts Inc.

Servant Manager Registration

You must register a servant manager with the POA before activating the POA's POA manager:

```
interface POA {
    // ...
    void            set_servant_manager(in ServantManager mgr)
                        raises(WrongPolicy);
    ServantManager  get_servant_manager() raises(WrongPolicy);
};
```

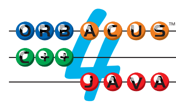
If you pass a servant activator, to **set_servant_manager**, the POA must use **USE_SERVANT_MANAGER** and **RETAIN**.

You can register the same servant manager with more than one POA.

You can set the servant manager only once; it remains attached to the POA until the POA is destroyed.

get_servant_manager returns the servant manager for a POA.

6
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.6 Servant Manager Registration

You must register a servant manager with a POA before you activate the POA's manager. This is necessary because, otherwise, clients will get OBJ_ADAPTER exceptions for their requests. (If you use the same servant manager for multiple POAs and later create another POA that will use the same servant manager, you must temporarily put the POA manager into the holding state and activate it again once you have created the new POA and set its servant manager.)

In C++, instantiation and registration of a servant manager is trivial:

```
Activator_impl activator;
PortableServer::ServantManager_var mgr_ref = activator._this();
some_poa->set_servant_manager(activator);
```

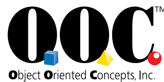
You cannot change the servant manager of a POA once you have assigned it. Calling set_servant_manager a second time raises OBJ_ADAPTER.

Type Issues with Servant Managers

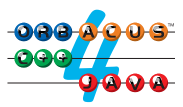
How does a servant manager know which type of interface is needed? Some options:

- Use a separate POA and separate servant manager class for each interface. The interface is implicit in the servant manager that is called.
- Use a separate POA for each interface but share the servant manager. Infer the interface from the POA name by reading the **`the_name`** attribute on the POA.
- Use a single POA and servant manager and add a type marker to the object ID. Use that type marker to infer which interface is needed.
- Store a type marker in the database with the persistent state.

The first option is usually easiest.



7
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.7 Type Issues with Servant Managers

When a POA calls `incarnate`, it passes the POA and object ID for the servant to be instantiated. `incarnate` is not told by the POA which type of interface (for example, a thermometer or a thermostat) is required. In fact, the POA itself does not know what type of interface a request is for until after it gets a servant for the request from the application. (The type of interface that eventually handles a request is not known at compile time and not transmitted over the wire because of late binding.) This means that a servant manager must know what type of interface to instantiate purely from the arguments it receives. The above slide presents the options for how you can deal with multiple interface types and servant managers.

The easiest thing is to simply use a separate POA and separate servant manager *class* for each interface type. That way, the decision which interface is needed is hard-coded into each servant manager class and made at compile time.

Another option is to share a servant manager among several POAs, one POA for each interface type. In that case, the servant manager can use the POA name (available by reading the `the_name` attribute on the POA). This approach allows you to use a single servant manager class for each interface type and decides what type to instantiate at run time. This approach is useful if instantiation of different servant types is substantially similar in nature (such as for thermometers and thermostats, where a lot of the code is identical).

You can use a type marker that becomes part of the object ID of each servant. For example, you could use a a “t” prefix on the object ID for thermostats, and an “m” for thermometers. This approach also works well, but is maintenance intensive if the interface types supported by a server grow over time (and it adds a small size penalty to object references). Alternatively, as a variation on this idea, you can keep the type marker in the database with the persistent state of each object.

 © Copyright 2000 Object Oriented Concepts Inc.

Servant Locators

```
native Cookie;
interface ServantLocator : ServantManager {
    Servant preinvoke(
                in ObjectId             oid,
                in POA                  adapter,
                in CORBA::Identifier    operation,
                out Cookie              the_cookie
            ) raises(ForwardRequest);

    void    postinvoke(
                in ObjectId             oid,
                in POA                  adapter,
                in CORBA::Identifier    operation,
                in Cookie               the_cookie,
                in Servant              serv
            );
};
```

OOC Object Oriented Concepts, Inc.

8
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.

ORBACUS C++ JAVA

2.8 Servant Locators

Servant locators provide a more powerful alternative to servant activators. They require the `NON_RETAIN` policy and `USE_SERVANT_MANAGER` policies on the POA. One consequence of `NON_RETAIN` is that the POA no longer maintains an Active Object Map. Instead, the association between a request and its servant remains in effect only for the duration of the request and is forgotten by the POA as soon as a request completes.

If you use servant locators, *every* incoming request causes the POA to first call `preinvoke`. Like `incarnate`, `preinvoke` can either throw an `OBJECT_NOT_EXIST` exception or return a servant. If `preinvoke` returns a servant, the request is given to the servant. As soon as the servant has carried out the request, the POA calls `postinvoke`, which permits you to clean up. Once `postinvoke` has finished, the POA forgets the association between the request and the servant. Another request for the same object will call `preinvoke` again (and may be carried out by the same servant or a different one).

Note that, in contrast to servant activators, servant locators are told which operation (or attribute) is being invoked via the `operation` parameter (`CORBA::Identifier` is an unbounded string). This permits you to select a servant based on the operation name as well as the POA and object ID.

`preinvoke` can return a value to the ORB in the `the_cookie` parameter. The ORB treats the cookie as an opaque value and never looks at it. However, it guarantees that the cookie that was returned by `preinvoke` will be passed to `postinvoke`. This allows you pass information from `preinvoke` to `postinvoke`, for example, by using the cookie as a key into a data structure.

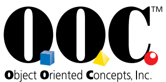
Implementing Servant Locators

The implementation of **preinvoke** is usually very similar to a factory operation (or **incarnate**):

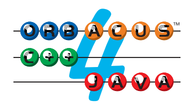
1. Use the POA, object ID, and operation name to locate the persistent state for the servant.
2. If the object does not exist, throw **OBJECT_NOT_EXIST**.
3. Instantiate a servant using the retrieved persistent state.
4. Return a pointer to the servant.

The implementation of **postinvoke** gets rid of the servant:

1. Write the persistent state of the servant to the DB (unless you are using write-through).
2. Call **_remove_ref** (or call **delete**, if the servant is not reference-counted).



9
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.9 Implementing Servant Locators

To implement a servant locator, you must derive your implementation from PortableServer::ServantLocator:

```
// In the PortableServer namespace:
// typedef void * Cookie;

class Locator_impl :
    public virtual POA_PortableServer::ServantLocator,
    public virtual PortableServer::RefCountServantBase {
public:
    virtual PortableServer::Servant
    preinvoke(
        const PortableServer::ObjectId &          oid,
        PortableServer::POA_ptr                   adapter,
        const char *                              operation,
        PortableServer::ServantLocator::Cookie & the_cookie
    ) throw(CORBA::SystemException,
            PortableServer::ForwardRequest);

    virtual void
    postinvoke(
        const PortableServer::ObjectId &         oid,
        PortableServer::POA_ptr                  adapter,
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
        const char *                                operation,
        PortableServer::ServantLocator::Cookie  the_cookie,
        PortableServer::Servant                     the_servant
    ) throw(CORBA::SystemException);
};
```

A simple implementation of `preinvoke` can be identical to an implementation of `incarnate`. Note that the IDL `Cookie` type is mapped to a `void *` in C++, so you can move arbitrary information between `preinvoke` and `postinvoke`.

```
PortableServer::Servant
Locator_impl::
preinvoke(
    const PortableServer::ObjectId &         oid,
    PortableServer::POA_ptr                  adapter,
    const char *                             operation,
    PortableServer::ServantLocator::Cookie & the_cookie
) throw(CORBA::SystemException, PortableServer::ForwardRequest)
{
    // Turn the OID into a string
    CORBA::String_var oid_str;
    try {
        oid_string = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST(); // Malformed OID
    }

    // Use OID to look in the DB for the persistent state...
    if (object_not_found)
        throw CORBA::OBJECT_NOT_EXIST();

    // Use the state retrieved from the database to
    // instantiate a servant. The type of the servant may be
    // implicit in the POA, the object ID, or the database state.
    AssetType anum = ...;
    return new Thermometer_impl(anum, /* ... */);
}
```

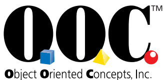
Note that this is not the most efficient implementation of `preinvoke`, mainly because it creates and destroys a servant for each request. However, with a bit of thought, we can come up with designs in which servants are not destroyed by `postinvoke` but are placed into a pool instead, to be reused if another request comes in for the same object. (See Henning & Vinoski for details.)

NOTE: The implementation of `postinvoke` is very similar to an implementation of `etherealize`, so we do not show it here.

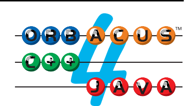
Use Cases for Servant Locators

Advantages of servant locators:

- They provide precise control over the memory use of the server, regardless of the number of objects supported.
- **`preinvoke`** and **`postinvoke`** bracket every operation call, so you can do work in these operations that must be performed for every operation, for example:
  - initialization and cleanup
  - creation and destruction of network connections or similar
  - acquisition and release of mutexes
- You can implement servant caches that bound the number of servants in memory to the *n* most recently used ones (Evictor Pattern.)



10
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.10 Use Cases for Servant Locators

The main use case for servant locators is that they limit memory consumption in the server to the number of objects actually in use at any one time. This means that you can build servers that scale to very large numbers of objects whose state is retrieved from persistent storage on demand. (Naturally, this scalability does not come for free because each invocation does more work, so throughput will be somewhat less.)

Another advantage of servant locators is that you can use the `preinvoke` and `postinvoke` calls to perform work that is common to all operations. For example, if you support life cycle operations and use an `m_removed` member as shown on page 1-46, you test this member in `preinvoke` and throw an `OBJECT_NOT_EXIST` exception if the object no longer exists. (Note that servant activators and servant locators can only throw system exceptions, not user exceptions. Also note that, if you want to access servant members during preinvoke or postinvoke, you must perform a down-cast from `ServantBase *` to a pointer to the actual type of your servant.)

`preinvoke` and `postinvoke` can also be useful to ensure exclusive access to a critical region by locking and unlocking a mutex before and after every operation (or only some operations—you can make a choice depending on the operation name). This works because the POA guarantees that, in multi-threaded servers, `preinvoke`, the operation body, and `postinvoke` all run in the same thread.

Finally, servant locators are useful to implement the Evictor Pattern, which creates a bounded pool of the most recently used servants. This technique is extremely effective because it exploits the fact that clients usually are interested in a small number of objects for some time before they shift interest to a different group of objects; this locality of reference means that most requests can be satisfied by an already instantiated servant. (See Henning and Vinoski for details.)

 © Copyright 2000 Object Oriented Concepts Inc.

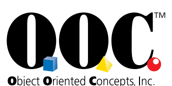
© Copyright 2000 Object Oriented Concepts Inc.

Servant Managers and Collections

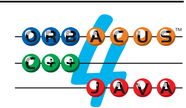
For operations such as **Controller::list**, you cannot iterate over a list of servants to return references to all objects because not all servants may be in memory.

Instead of instantiating a servant for each object just so you can call **_this** to create a reference, you can create a reference without instantiating a servant:

```
interface POA {
    // ...
    Object create_reference(in CORBA::RepositoryId intf)
            raises(WrongPolicy);

    Object create_reference_with_id(
                in ObjectId             oid,
                in CORBA::RepositoryId  intf
            ) raises(WrongPolicy);
};
```



11
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.11 Servant Managers and Collections

In the absence of servant managers, an operation like `list` on the `Controller` interface can be implemented by iterating over a list of servant pointers and calling `_this` on each servant to obtain its reference. However, if servant managers are used, this creates a problem: not all servants are in memory, so iterating over them will not return the complete list of objects. In addition, in order to create a reference, you would have to instantiate the servant first in order to call `_this` on the servant. Of course, this would defeat the purpose of using servant managers in the first place and would make operations such as `list` prohibitively expensive.

To get around this problem, the POA offers `create_reference` and `create_reference_with_id`. Both operations create an object reference in isolation, that is, without the need for an instantiated servant. `create_reference` requires `SYSTEM_ID` and generates a unique object ID for the reference, whereas `create_reference_with_id` requires `USER_ID` and you have supply an object ID explicitly. (This is analogous to the difference between `activate_object` and `activate_object_with_id`.)[1]

Both operations require you to supply the repository ID for interface the reference will denote (such as `"IDL:acme.com/CCS/Thermometer:1.0"`).

Once you have created a reference this way, you can return it to clients as usual. The only difference to calling `_this` is that the reference does not have a servant yet. When a client invokes an operation via the reference, a servant manager can take care of bringing the servant into memory as usual.

1. If you use `create_reference`, you do not know what the generated object ID is unless you also call `reference_to_id`. (See page 2-32.)

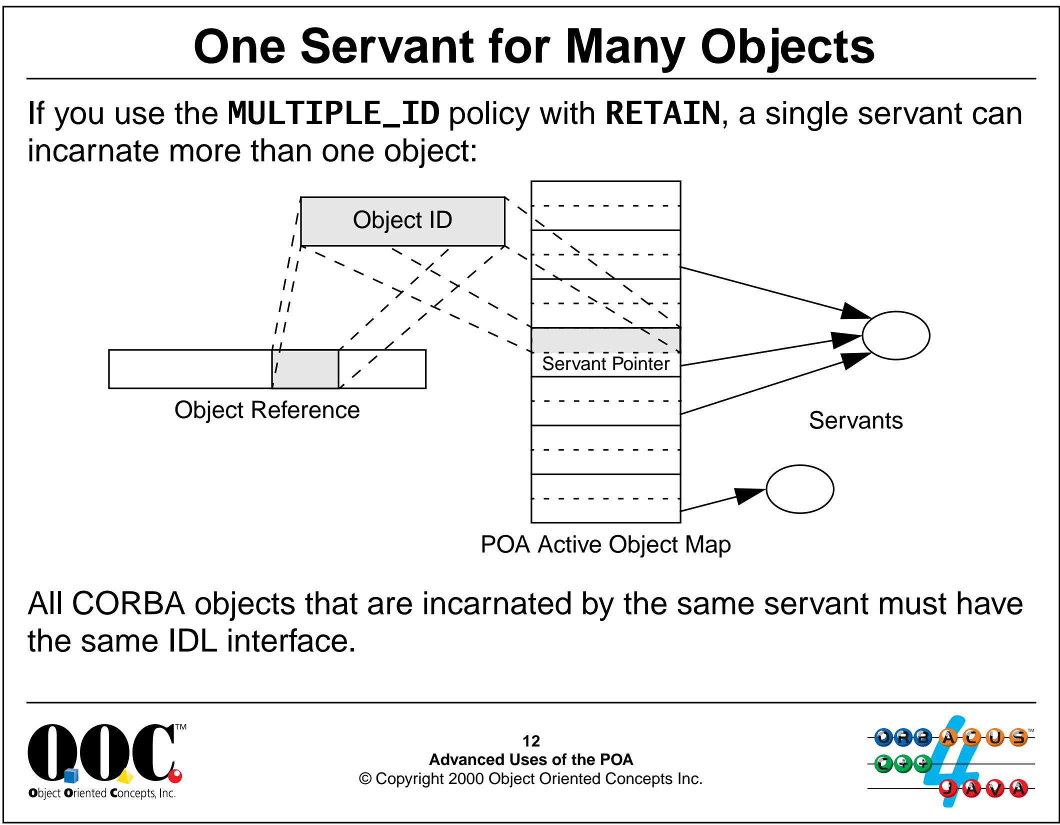
 © Copyright 2000 Object Oriented Concepts Inc.

Because operations such as `list` are strongly typed and return something other than type `Object`, you must call `_narrow` before you can return the reference:

```
CCS::Controller::ThermometerSeq *
Controller_impl::
list() throw(CORBA::SystemException)
{
    CCS::Controller::ThermometerSeq_var return_seq
        = new CCS::Controller::ThermometerSeq;
    CORBA::ULong index = 0;

    // Iterate over the database contents (or other
    // list of existing objects) and create a reference
    // for each one.
    while (more objects remain) {
        // Get asset number from database and convert to OID.
        CCS::AssetType anum = ...;
        ostrstream ostr;
        ostr << anum << ends;
        PortableServer::ObjectId_var oid =
            PortableServer::string_to_ObjectId(ostr.str());
        ostr.rdbuf()->freeze(0);

        // Use info from the object state to work out which
        // type of device we are dealing with and figure out
        // which POA to use.
        const char * rep_id;
        PortableServer::POA_var poa;
        if (device is a thermometer) {
            rep_id = "IDL:acme.com/CCS/Thermometer:1.0";
            poa = ...;  // Thermometer POA
        } else {
            rep_id = "IDL:acme.com/CCS/Thermostat:1.0";
            poa = ...;  // Thermostat POA
        }

        // Create reference
        CORBA::Object_var obj =
            poa->create_reference_with_id(oid, rep_id);

        // Narrow and store in our return sequence.
        return_seq->length(index + 1);
        return_seq[index++] = CCS::Thermometer::_narrow(obj);
    }
    return return_seq._retn();
}
```

Note that this code relies on an ORB-specific optimization, namely, that a successful call to `_narrow` will be short-cut by the ORB and not sent to the target servant. ORBacus provides this optimization (but other ORBs may not). Without the optimization, this technique is useless.

© Copyright 2000 Object Oriented Concepts Inc.



2.12 One Servant for Many Objects

If you use the `MULTIPLE_ID` policy with RETAIN,[2] you can add the same servant to the AOM several times with different object IDs. This means that there are as many CORBA objects as there are entries in the AOM, but that some of these objects happen to be represented by the same servant. This idea is attractive if we need a server that supports a large number of CORBA objects that are used by clients simultaneously, but must limit its memory footprint.

If you use `MULTIPLE_ID`, you must ensure that, if you register the same servant for multiple objects, all objects support the same interface (have the same repository ID), otherwise you will get unpredictable behavior at run time (such as clients receiving a `BAD_OPERATION` exception, or even silent and potentially fatal run time errors).

Using a single servant for multiple CORBA objects means that we can no longer maintain the object state in the servant, because the identity of the CORBA object for a particular request is not longer implicit in the C++ servant that handles the request. In other words, the same single servant must now pretend to be different CORBA objects for different requests, on a per-request basis. Servants use the `Current` interface to achieve this.

2. For `NON_RETAIN` POAs, neither `MULTIPLE_ID` nor `UNIQUE_ID` have any effect because the mapping from object IDs to servants is managed by the application.

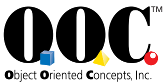
 © Copyright 2000 Object Oriented Concepts Inc.

The Current Object

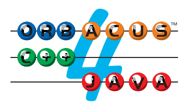
The **Current** object provides information about the request context to an operation implementation:

```
module PortableServer {
    // ...
    exception NoContext {};

    interface Current : CORBA::Current {
            POA         get_POA() raises(NoContext);
            ObjectId    get_object_id() raises(NoContext);
    };
};
```

The **get_POA** and **get_object_id** operations must be called from within an executing operation (or attribute access) in a servant.



© Copyright 2000 Object Oriented Concepts Inc.



2.13 The Current Object

The `Current` object delivers information about the currently executing request to a servant (and therefore must be invoked only from within the context of an operation on a servant). Calling `get_POA` or `get_object_id` from outside the context of an executing operation raises `NoContext`.

You obtain access to the `Current` object by calling `resolve_initial_references`:

```
CORBA::Object_var obj =
    orb->resolve_initial_references("POACurrent");

PortableServer::Current_var cur =
    PortableServer::Current::_narrow(obj);
```

You only need to call `resolve_initial_references` once to obtain the `Current` object and, thereafter, you can use that same object reference for all POAs and servants. Invocations on the `Current` object automatically return the correct information for the currently executing request, even in multi-threaded servers. In effect, `Current` is a singleton object that returns thread-specific data.

The implementation of our servants now must use the `Current` object on entry to each operation to find out what its identity should be for the current request, and use that information to act on the correct state. Assuming that we store a reference to the `Current` object in the global variable `poa_current`, we can write a helper function that retrieves the asset number for the current request. (You would usually make this function a private member function of the servant):

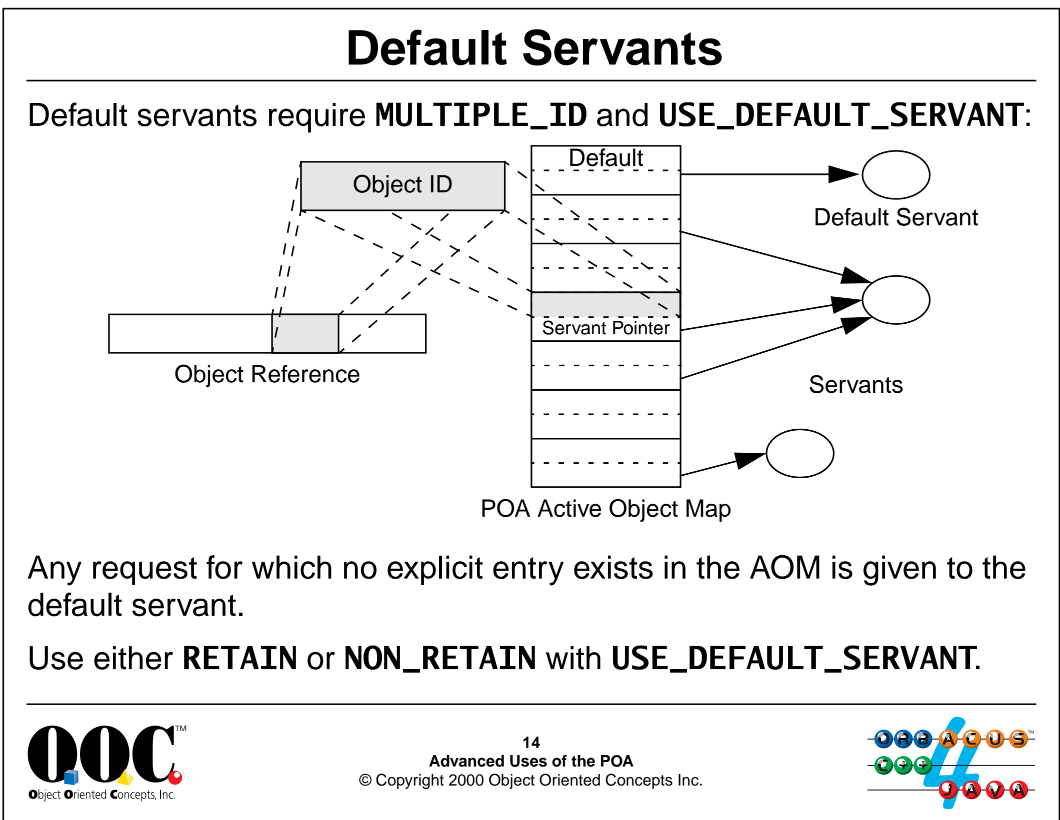
 © Copyright 2000 Object Oriented Concepts Inc.

```
CCS::AssetType
Thermometer_impl::
get_anum()
throw(CORBA::SystemException)
{
    // Get object ID from Current object
    PortableServer::ObjectId_var oid =
        poa_current->get_object_id();

    // Check that ID is valid
    CORBA::String_var tmp;
    try {
        tmp = PortableServer::ObjectId_to_string(oid);
    } catch (const CORBA::BAD_PARAM &) {
        throw CORBA::OBJECT_NOT_EXIST();
    }

    // Turn string into asset number
    istrstream istr(tmp.in());
    CCS::AssetType anum;
    istr >> anum;
    if (str.fail())
        throw CORBA::OBJECT_NOT_EXIST();
    return anum;
}
```

To implement the servant, we call this helper function on entry to every operation to get the asset number, and in turn use the asset number as a key to locate the data (typically, by looking it up in a database or interrogating a network):

```
CCS::LocType
Thermometer_impl::
location() throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();    // Who are we?

    // Get location string from the database
    CORBA::String_var loc = db_get_field(anum, "LocationField");
    return loc._retn();
}

void
Thermometer_impl::
location(const char * loc) throw(CORBA::SystemException)
{
    CCS::AssetType anum = get_anum();    // Who are we?

    // Set location string in the database
    db_set_field(anum, "LocationField", loc);
}
```

© Copyright 2000 Object Oriented Concepts Inc.



2.14 Default Servants

If you set the USE_DEFAULT_SERVANT policy, the POA allows you to register a default servant:

- If you combine USE_DEFAULT_SERVANT with RETAIN, the POA first attempts to locate a matching servant in the AOM. If no servant is explicitly for the object ID in the request, the POA passes the request to the default servant. (This situation is shown above.)
- If you combine USE_DEFAULT_SERVANT with NON_RETAIN, all requests go to the default servant.

The first combination, even though appealing at first, is probably overkill. For example, you can achieve the same effect by using two POAs, one with a default servant and one without. This is not only simple, but also makes call dispatch to the default servant faster because it avoids the two-step process of locating a servant by first looking for an explicitly registered servant and passing the request to the default servant only if no explicitly registered servant is found. For this reason, we recommend that, if you use default servants, you should use POAs with the NON_RETAIN policy.

For USE_DEFAULT_SERVANT and NON_RETAIN, you must use a separate POA for each interface that is implemented by a default servant (because a single servant cannot implement two interfaces simultaneously). For USE_DEFAULT_SERVANT and RETAIN, you can mix interface types on the same POA, but all requests that are directed to the default servant must be for objects with the same interface.

 © Copyright 2000 Object Oriented Concepts Inc.

As for servant managers, you must explicitly register the default servant for a POA:

```
interface POA {
    // ...
    Servant get_servant() raises(NoServant, WrongPolicy);
    void    set_servant(in Servant s) raises(WrongPolicy);
};
```

`get_servant` returns the current default servant and raises `NoServant` if none has been set.

`set_servant` sets the default servant. You can call `set_servant` more than once to change the default servant. (However, it is unlikely that you would ever do this unless you wanted to dynamically replace the code for the default servant at run time.)

The POA calls `_add_ref` on the default servant during the call to `set_servant`. This means that, if you use a reference-counted default servant, you can call `_remove_ref` immediately after calling `set_servant`. If you do this, the default servant will be automatically destroyed when its POA is destroyed. Otherwise, you must destroy the default servant explicitly once it is no longer needed.

Also note that the POA calls `_add_ref` when you call `get_servant`. This means that you must eventually call `_remove_ref` on the returned servant, otherwise the reference count will be left too high. An easy way to ensure this happens is to use a `ServantBase_var` when calling `get_servant`:

```
PortableServer::ServantBase_var servant
    = some_poa->get_servant();

// ServantBase_var destructor calls _remove_ref() eventually...

// If we want the actual type of the servant again, we must
// use a down-cast:
Thermometer_impl * dflt_serv =
    dynamic_cast<Thermometer_impl *>(servant);
```

© Copyright 2000 Object Oriented Concepts Inc.

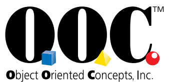
Trade-Offs for Default Servants

Default servants offer a number of advantages:

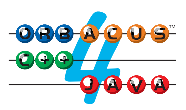
- simple implementation
- POA and object ID can be obtained from **`Current`**
- ideal as a front end to a back-end store
- servant is completely stateless
- infinite scalability!

The downside:

- possibly slow access to servant state



15
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.15 Trade-Offs for Default Servants

The main motivation for using default servants is scalability because a single C++ instance can be used to simultaneously represent an unlimited number of CORBA objects. The implementation becomes very simple: each operation first retrieves the object ID to identify the CORBA object and then uses the identity of the object to retrieve its state. As a result, the servant itself is completely stateless, which make this approach ideal if you want to use a CORBA server as a front end to a database that may be updated independently. Provided that the database provides atomic access to servant state, there are no cache coherency issues that could arise through independent updates.

Using default servants, the scalability of a CORBA server is no longer limited by its memory consumption and only depends on the bandwidth to the database and the number of parallel invocations you can afford to support.

The unlimited scalability of default servants comes at a price though: each access to servant state takes longer than if you would use multiple servants that hold their state in member variables and write to the database only occasionally when they are updated. However, if you have accesses with good locality of reference and a database with effective caching, the performance penalty of default servants can be surprisingly small, so the technique is well worth exploring.

 © Copyright 2000 Object Oriented Concepts Inc.

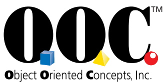
POA Activators

You can create POAs on demand, similar to activating servants on demand:

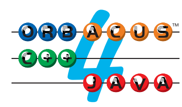
```
module PortableServer {
  // ...
  interface AdapterActivator {
    boolean unknown_adapter(in POA parent, in string name);
  };
};
```

This is a callback interface you provide to the ORB.

If a request for an unknown POA arrives, the ORB invokes the **unknown_adapter** operation to allow you to create the POA.



16
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.16 POA Activators

The POA offers a mechanism to activate POAs on demand. The idea is similar to those of servant managers: by providing a callback mechanism, it is possible to avoid having all POAs in memory at all times. The main motivation for POA activators is to permit a server to easily implement optimistic caching (also known as pre-fetching). This is important if, for example, a number of objects are always used as a group. By using a separate POA for each group, you can use a POA activator to activate all the servants for the objects in a group at once, so they are available immediately, without further initialization while they are in use. Once you no longer require a group, you can simply invoke `destroy` on the POA to reclaim the resources that were used by the group.

To implement a POA activator, you must derive a servant class from `PortableServer::AdapterActivator`:

```
class POA_Activator_impl :
    public virtual POA_PortableServer::AdapterActivator {
public:
                POA_Activator_impl() {}
    virtual     ~POA_Activator_impl() {}
    virtual CORBA::Boolean
                unknown_adapter(
                    PortableServer::POA_ptr parent,
                    const char *            name
                ) throw(CORBA::SystemException);
};
```

© Copyright 2000 Object Oriented Concepts Inc.

Implementing POA Activators

The **parent** parameter allows you to get details of the parent POA (particularly, the name of the parent POA).

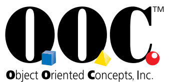
The **name** parameter provides the name for the new POA.

While **unknown_adapter** is running, requests for the new adapter are held pending until the activator returns.

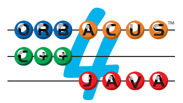
The implementation of the activator must decide on a set of policies for the new POA and instantiate it.

If optimistic caching is used, the activator must instantiate the servants for the POA. (If combined with **USE_SERVANT_MANAGER**, a subset of the servants can be instantiated instead.)

On success, the activator must return true to the ORB (which dispatches the request as usual.) A false return value raises **OBJECT_NOT_EXIST** in the client.



17
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.17 Implementing POA Activators

To implement a POA activator, you only need to create the `unknown_adapter` member function:

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *            name
) throw(CORBA::SystemException)
{
    // Check which adapter is being activated and
    // create appropriate policies. (Might use pre-built
    // policy list here...)
    CORBA::PolicyList policies;
    if (strcmp(name, "Some_adapter") == 0) {
        // Create policies for "Some_adapter"...
    } else if (strcmp(name, "Some_other_adapter") == 0) {
        // Create policies for "Some_other_adapter"...
    } else {
        // Unknown POA name
        return false;
    }
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
    // Select POA manager for new adapter (parent POA
    // manager in this example).
    PortableSerer::POAManager_var mgr = parent->the_POAManager();

    // Create new POA.
    try {
        PortableServer::POA_var child =
            parent->create_POA(name, mgr, policies);
    } catch (const PortableServer::POA:AdapterAlreadyExists &) {
        return false;
    } catch (...) {
        return false;
    }

    // For optimistic caching, activate servants here...

    return true;
}
```

Note that the code ensures that false is returned if the POA being activated already exists. This is necessary because (at least for multi-threaded servers), another thread may have already activated the adapter.

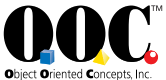
Registering POA Activators

An adapter activator must be registered by setting the POA's **`the_activator`** attribute:

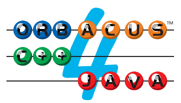
```
interface POA {
    // ...
    attribute AdapterActivator the_activator;
};
```

You can change the adapter activator of an existing POA, including the Root POA.

By attaching an activator to all POAs, a request for a POA that is low in the POA hierarchy will automatically activate all parent POAs that are needed.



18
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.18 Registering POA Activators

To register a POA activator, you must set the `the_activator` attribute of its parent POA. Note that, by attaching an activator to all POAs (including the Root POA), you can ensure that an incoming request automatically activates the necessary parent POAs as well. For example, the code of the preceding example can be modified such that the POA activator adds itself to the POAs it creates:

```
CORBA::Boolean
POA_Activator_impl::
unknown_adapter(
    PortableServer::POA_ptr parent,
    const char *            name
) throw(CORBA::SystemException)
{
    // ...

    // Create new POA.
    try {
        PortableServer::POA_var child =
            parent->create_POA(name, mgr, policies);
        PortableServer::AdapterActivator_var act = _this();
        child->the_activator(act);
    } catch (const PortableServer::POA:AdapterAlreadyExists &) {
        return false;
```

 © Copyright 2000 Object Oriented Concepts Inc.

```
    } catch (...) {
        return false;
    }

    // ...
}
```

In `main`, we use the same adapter activator as the Root POA's activator:

```
// ...

PortableServer::POA_var root_poa = ...;

// Create activator servant.
POA_Activator_impl act_servant;

// Register activator with Root POA.
PortableServer::AdapterActivator_var act = act_servant._this();
root_poa->the_activator(act);

// ...
```

Note that, for POA activators to work, the POA manager for the Root POA must be active if the server uses indirect binding via the implementation repository (see Unit 3); otherwise, the server has no transport endpoint for the incoming request that should result in activation of a new POA. However, it is not necessary for POAs that are dynamically activated to actually use the Root POA manager; you can use any POA manager you like but, for indirect binding, activation will work only when the Root POA manger is active.

If your server does not use the implementation repository, the POA manager of the to-be-activated POA must be in the active state.

Finding POAs

The **find_POA** operation locates a POA:

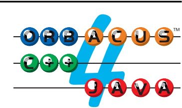
```
// In module PortableServer: typedef sequence<POA> POAList;
interface POA {
    // ...
    POA find_POA(in string name, in boolean activate_it)
            raises(AdapterNonExistent);
    readonly attribute POAList the_children;
    readonly attribute POA     the_parent;
};
```

You must invoke **find_POA** on the correct parent (because POA names are unique only within their parent POA).

If **activate_it** is true and the parent has an adapter activator, **unknown_adapter** will be called to create the child POA.

You can use this to instantiate all your POAs by simply calling **find_POA**.

19
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.19 Finding POAs

The `the_children` attribute retrieves a list of all the (current) children of a POA.

The `the_parent` attribute returns the parent of a POA. (The parent of the Root POA is a nil reference.)

The `find_POA` operation returns the child POA with the supplied name. If the `activate_it` parameter is true, calls to `find_POA` trigger the parent's POA activator. This is particularly useful if you use adapter activators anyway because you can create new POAs by simply calling `find_POA` from main. This has the advantage that all your POA creation code is kept in one place and that you can centralize the mapping from POA names to policies (for example, in a single lookup table that maps POA names to policy lists).

```
// ...

PortableServer::POA_var root_poa = ...;

// Create activator servant.
POA_Activator_impl act_servant;

// Register activator with Root POA.
PortableServer::AdapterActivator_var act = act_servant._this();
root_poa->the_activator(act);

// Use find_POA to create a POA hierarchy. The POAs will be
// created by the adapter activator.
```

 © Copyright 2000 Object Oriented Concepts Inc.

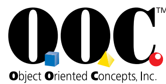
```
PortableServer::POA_var ctrl_poa
    = root_poa->find_POA("Controller", true);
PortableServer::POA_var thermometer_poa
    = ctrl_poa->find_POA("Thermometers", true);
PortableServer::POA_var thermostat_poa
    = ctrl_poa->find_POA("Thermostats", true);

// Activate POAs...
```

© Copyright 2000 Object Oriented Concepts Inc.

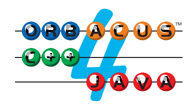
Identity Mapping Operations

The POA offers a number of operations to map among object references, object IDs, and servants:

```
interface POA {
    // ...
    ObjectId servant_to_id(in Servant s)
                raises(ServantNotActive, WrongPolicy);
    Object   servant_to_reference(in Servant s)
                raises(ServantNotActive, WrongPolicy);
    Servant  reference_to_servant(in Object o)
                raises(ObjectNotActive, WrongAdapter, WrongPolicy);
    ObjectId reference_to_id(in Object reference)
                raises(WrongAdapter, WrongPolicy);
    Servant  id_to_servant(in ObjectId oid)
                raises(ObjectNotActive, WrongPolicy);
    Object   id_to_reference(in ObjectId oid)
                raises(ObjectNotActive, WrongPolicy);
};
```



20
Advanced Uses of the POA
© Copyright 2000 Object Oriented Concepts Inc.



2.20 Identity Mapping Operations

The POA provides operations that allow you to map among references, object IDs, and servants.

```
ObjectId servant_to_id(in Servant s)
            raises(ServantNotActive, WrongPolicy)
```

This operation returns the object ID for a servant. The behavior depends on whether you invoke `servant_to_id` from inside an executing operation on the specified servant, or from the outside:

- If called from inside an executing operation on the specified servant, the operation returns the object ID for the current request (that is, the semantics are the same as calling `get_object_id` on the `Current` object).
- If called from outside an executing operation on the specified servant, the POA must
  - either use the `RETAIN` policy, together with `UNIQUE_ID` or `IMPLICIT_ACTIVATION`,
  - or the `USE_DEFAULT_SERVANT` policy.

  `servant_to_id` raises `WrongPolicy` if the policies on the POA do not match these criteria.

  The behavior when called from outside an executing operation on the specified servant is as follows:

  - If the specified servant is in the AOM and `UNIQUE_ID` is in effect, the operation returns that servant's ID.
  - If the POA uses `IMPLICIT_ACTIVATION` (which implies `SYSTEM_ID`) and the servant is not in the AOM, it implicitly activates the servant with a new object ID and returns that ID. This happens whether `UNIQUE_ID` or `MULTIPLE_ID` is in effect and whether you

 © Copyright 2000 Object Oriented Concepts Inc.

use USE_DEFAULT_SERVANT or USE_ACTIVE_OBJECT_MAP_ONLY and is almost certainly not what you want, so we suggest you avoid IMPLICIT_ACTIVATION.

- If neither of the preceding conditions holds, the operation raises ServantNotActive.

Object servant_to_reference(in Servant s)
raises(ServantNotActive, WrongPolicy)

This operation returns the object reference for a servant. The behavior depends on whether you invoke servant_to_reference from inside an executing operation on the specified servant, or from the outside:

- If called from inside an executing operation on the specified servant, the operation returns the reference for the current request (that is, the semantics are the same as calling get_object_id on the Current object and creating a reference with that object ID).
- If called from outside an executing operation on the specified servant, the POA must
  - either use the RETAIN policy, together with UNIQUE_ID or IMPLICIT_ACTIVATION,
  - or the USE_DEFAULT_SERVANT policy.

  servant_to_reference raises WrongPolicy if the policies on the POA do not match these criteria.

  The behavior when called from outside an executing operation on the specified servant is as follows:

  - If the specified servant is in the AOM and UNIQUE_ID is in effect, the operation returns that servant's reference.
  - If the POA uses IMPLICIT_ACTIVATION (which implies SYSTEM_ID) and the servant is not in the AOM, it implicitly activates the servant with a new object ID and returns a reference containing that ID. This happens whether UNIQUE_ID or MULTIPLE_ID is in effect and whether you use USE_DEFAULT_SERVANT or USE_ACTIVE_OBJECT_MAP_ONLY and is almost certainly not what you want, so we suggest you avoid IMPLICIT_ACTIVATION.
  - If neither of the preceding conditions holds, the operation raises ServantNotActive.

Servant reference_to_servant(in Object o)
raises(ObjectNotActive, WrongAdapter, WrongPolicy)

Calling reference_to_servant on a POA other than the one that created the reference raises WrongAdapter.

Calling reference_to_servant on a POA that does not use either RETAIN or USE_DEFAULT_SERVANT raises WrongPolicy.

Otherwise, the operation returns the servant for an object reference:

- If the POA uses RETAIN and the object denoted by the reference is in the AOM, the operation returns the corresponding servant.
- Otherwise, if the POA uses USE_DEFAULT_SERVANT, the operation returns the default servant.
- Otherwise, the operation raises ObjectNotActive.

© Copyright 2000 Object Oriented Concepts Inc.

```
ObjectId reference_to_id(in Object reference)
            raises(WrongAdapter, WrongPolicy)
```

Calling `reference_to_id` on a POA other than the one that created the reference raises `WrongAdapter`. Otherwise, `reference_to_id` returns the object ID encapsulated in the reference. (The `WrongPolicy` exception is currently not raised and reserved for future extensions.)

```
Servant id_to_servant(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy)
```

Calling `id_to_servant` on a POA that does not have either the `RETAIN` policy or the `USE_DEFAULT_SERVANT` policy raises `WrongPolicy`. Otherwise, the operation behaves as follows:

- If the specified ID is in the AOM, `id_to_servant` returns the corresponding servant.
- Otherwise, if the POA uses `USE_DEFAULT_SERVANT`, `id_to_servant` returns the default servant.
- Otherwise, the operation raises `ObjectNotActive`.

```
Object id_to_reference(in ObjectId oid)
            raises(ObjectNotActive, WrongPolicy)
```

Calling `id_to_reference` on a POA that does not use the `RETAIN` policy raises `WrongPolicy`. Otherwise, if an object with the specified ID is currently active, the operation returns an object reference that encapsulates the specified ID. If no object with the specified ID is active, the operation raises `ObjectNotActive`.

 © Copyright 2000 Object Oriented Concepts Inc.

3. The Implementation Repository (IMR)

Summary

This unit explains the difference between direct and indirect binding and shows how an implementation repository may be used to both enable server migration and to achieve automatic server activation on demand. The unit also covers performance and reliability trade-offs for implementation repositories and shows how to configure the IMR for various deployment scenarios.

Objectives

By the completion of this unit, you will know how to configure and use an IMR effectively. You will also understand the environment in which processes are started by the IMR and how to write your server to work within this environment.

Purpose of an Implementation Repository

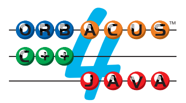
An implementation repository (IMR) has three functions:

- It maintains a registry of known servers.
- It records which server is currently running on what machine, together with the port numbers it uses for each POA.
- It starts servers on demand if they are registered for automatic activation.

The main advantage of an IMR is that servers that create persistent references

- need not run on a fixed machine and a fixed port number
- need not be running permanently

1
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.1 Purpose of an Implementation Repository

The main motivation for providing an IMR is to permit servers to move from port to port or machine to machine without breaking existing references that are held by clients. Fundamentally, an IMR has three functions:

- The IMR maintains a registry of known servers.

  A server that wants to take advantage of IMR functionality must be known to the IMR. This is achieved by explicitly registering the server with the IMR when the server is deployed.

- The IMR records which server is currently running on what machine, together with the port numbers it uses for each POA.

  This function of the IMR allows servers to change location (machine or port) without clients being aware of this happening.

- The IMR starts servers on demand if they are registered for automatic activation.

  This function of the IMR permits you to have servers that correctly respond to client requests without the need to run them permanently. This is useful particularly for large systems with many servers, some of which may not be needed all the time. (Idle servers still consume system resources and so incur a cost. In addition, the feature permits automatic recovery if a server crashes because the IMR will transparently restart it.)

© Copyright 2000 Object Oriented Concepts Inc.

Binding

There are two methods of binding object references:

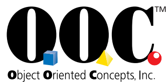
- Direct Binding (for persistent and transient references)

  References carry the host name and port number of the server. This works, but you cannot move the server around without breaking existing references.

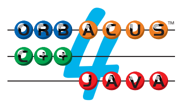
- Indirect Binding (for persistent references)

  References carry the host name and port number of an Implementation Repository (IMR). Clients connect to the IMR first and then get a reference to the actual object in the server. This allows servers to move around without breaking existing references.

IMRs are proprietary for servers (but interoperate with all clients).



2
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.2 Binding

When a client receives an object reference from somewhere, the client-side run time must establish a connection to the server that hosts the target object eventually. The process of associating an object reference with a transport end point is known as binding. Binding can either be direct or indirect:

- For direct binding, a server embeds its own host name (or IP address) and port number into each reference. When clients connect, they therefore connect directly to the server. This is simple and efficient, but has drawbacks:
  - You must assign a distinct port number to each server and set that port number consistently on every execution of the server, using the **`-OAport`** option (or using a server property—see Unit 18). For installations with a large number of servers, the administration of port numbers can get cumbersome.
  - Once a server has handed out persistent object references to its clients, you can no longer move the server to a different port or different machine, at least not if you want to avoid breaking the references that are held by clients. Because clients may store their references in files or a service such as the Naming Service (see Unit 19), you have no way of knowing whether it is safe to move a server.

  These problems can be avoided by indirect binding.
- Indirect binding requires a daemon process known as the Implementation Repository (IMR) to permanently run at a fixed location (host and port number). When a server creates a persistent reference, it embeds the host name and port number of the IMR in the reference, instead of its own addressing information. When clients bind a reference, they first connect to

 © Copyright 2000 Object Oriented Concepts Inc.

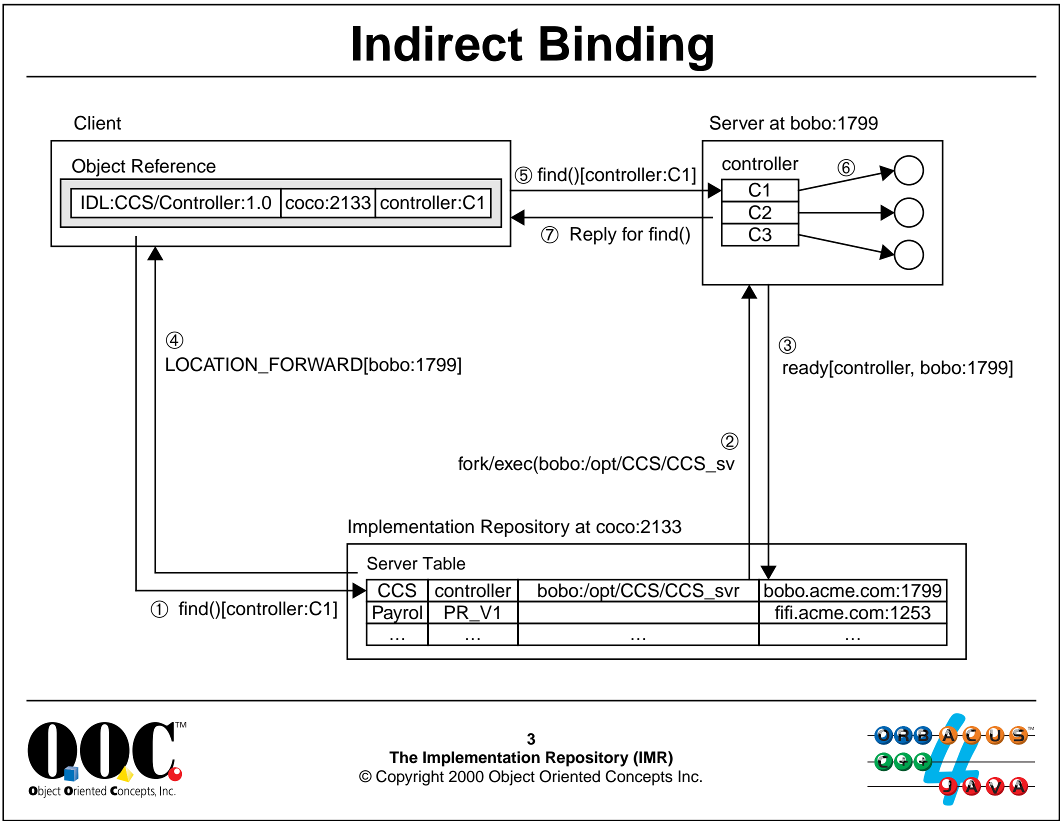
the IMR and send their first request. On receipt of the request, the IMR works out where the server is currently running and returns a new reference to the client-side run time in a special location-forward reply. The client ORB then opens another connection using the new reference returned by the IMR, which contains the actual address details of the server. Provided the server is actually running at the location that was forwarded by the IMR, this second attempt succeeds and the client request ends up at the correct destination.

The extra level of indirection provided by the IMR allows you to move servers around over time without breaking existing references. Servers, when they start up, inform the IMR of where they currently are, so the IMR always knows about the current location of each server. In effect, the IMR provides a fixed point for object addressing in order to permit servers to change their addresses.

Implementation repositories are not standardized by the OMG, so their implementation is completely proprietary. In particular, servers can only use an implementation repository from the same ORB vendor. For example, an ORBacus server requires an ORBacus IMR and will not work with any other vendor's IMR; similarly, no other vendor's server can use an ORBacus IMR.

However, the interactions between clients and IMRs are completely standardized and only rely on IIOP. This means that a client using any vendor's ORB can interact with any other vendor's IMR.

© Copyright 2000 Object Oriented Concepts Inc.



3.3 Indirect Binding

The above diagram illustrates the sequence of interactions for indirect binding of a reference to the controller object. The diagram assumes that the implementation repository runs on machine `coco` at port 2133 and that the CCS server is not running when the client invokes the request. The sequence of steps during binding is as follows.

1. The client invokes the `find` operation on the controller. This results in the client-side run time opening a connection to the address found in the controller IOR, which is the address of the repository. With the request, the client sends the object key (which contains the POA name and the object ID—`controller` and `C1` in this example).
2. The IMR uses the POA name (`Controller`) to index into its server table and finds that the server is not running. Because the server is registered for automatic start-up, the IMR executes the command to start the server.
3. The server sends messages that inform the repository of its machine name (`bobo`), the names of the POAs it has created and their port numbers (`Controller` at 1799), and the fact that it is ready to accept requests.
4. The implementation repository constructs a new object reference that contains host `bobo`, port number 1799, and the original object key and returns it in a `LOCATION_FORWARD` reply to the client.
5. The client opens a connection to `bobo` at port 1799 and sends the request a second time.
6. The server uses the POA name (`Controller`) to locate the POA that contains the servant for the request, and uses the object ID (`C1`) to identify the target servant. The request is given to the servant for processing.

 © Copyright 2000 Object Oriented Concepts Inc.

7. The servant completes the `find` operation and returns its results, which are marshaled back to the client in a `Reply` message.

As you can see, indirect binding uses the implementation repository as a location broker that returns a new IOR to the client that points at the current server location. The CORBA specification does not limit indirection to a single level. Instead, it requires a client to always respond to a `LOCATION_FORWARD` reply by attempting to send another request.

© Copyright 2000 Object Oriented Concepts Inc.

Automatic Server Start-Up

The IMR can optionally start server processes.

Two modes of operation are supported by the IMR:

- shared

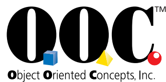
  All requests from all clients are directed to the same server. The server is started on demand.

- persistent

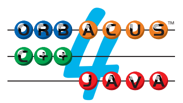
  Same as the shared mode, but the server is started whenever the IMR starts and kept running permanently.

Servers are started by an Object Activation Daemon (OAD).

A single repository can have multiple OADs. An OAD must be running on each machine on which you want to start servers.



1
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



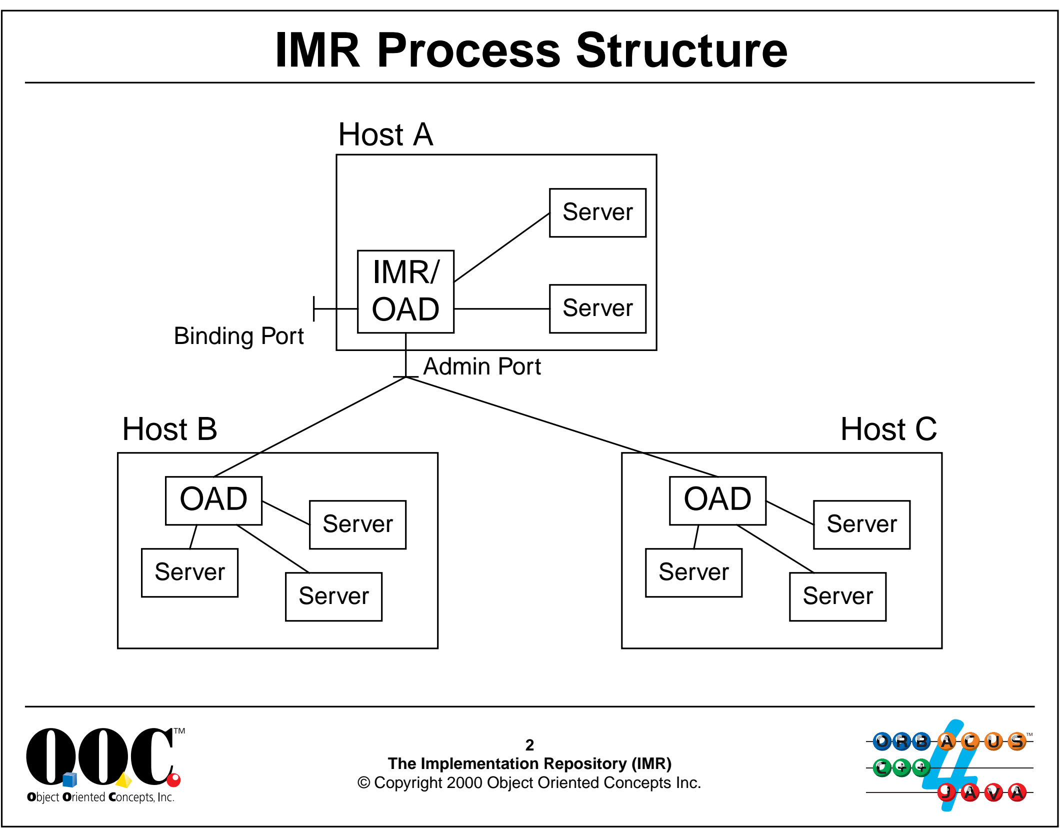
3.4 Automatic Server Start-Up

The IMR maintains knowledge of where each server is currently running in order to be able to forward that information to clients during indirect binding. This means that it is easy to extend the IMR to not only forward the location of a running server to clients, but to also start up servers on demand when a client request arrives for a server that is not currently running.

The ORBacus IMR offers two modes of automatic server activation:

- In shared mode, a single copy of the server is started when the first request for that server arrives. Thereafter, all requests from all clients are directed to that server instance.
- In persistent mode, a single server instance is started by the IMR as soon as the IMR itself is started. Thereafter, the IMR monitors the server process and restarts it when it goes down. All requests from all clients are sent to that same server instance.

 © Copyright 2000 Object Oriented Concepts Inc.



3.5 IMR Process Structure

Each server that uses indirect binding is configured to use a specific OAD. The OAD monitors the state of the server process and is informed by the server of state changes, such as POA creation and destruction. A server cannot use more than one OAD.

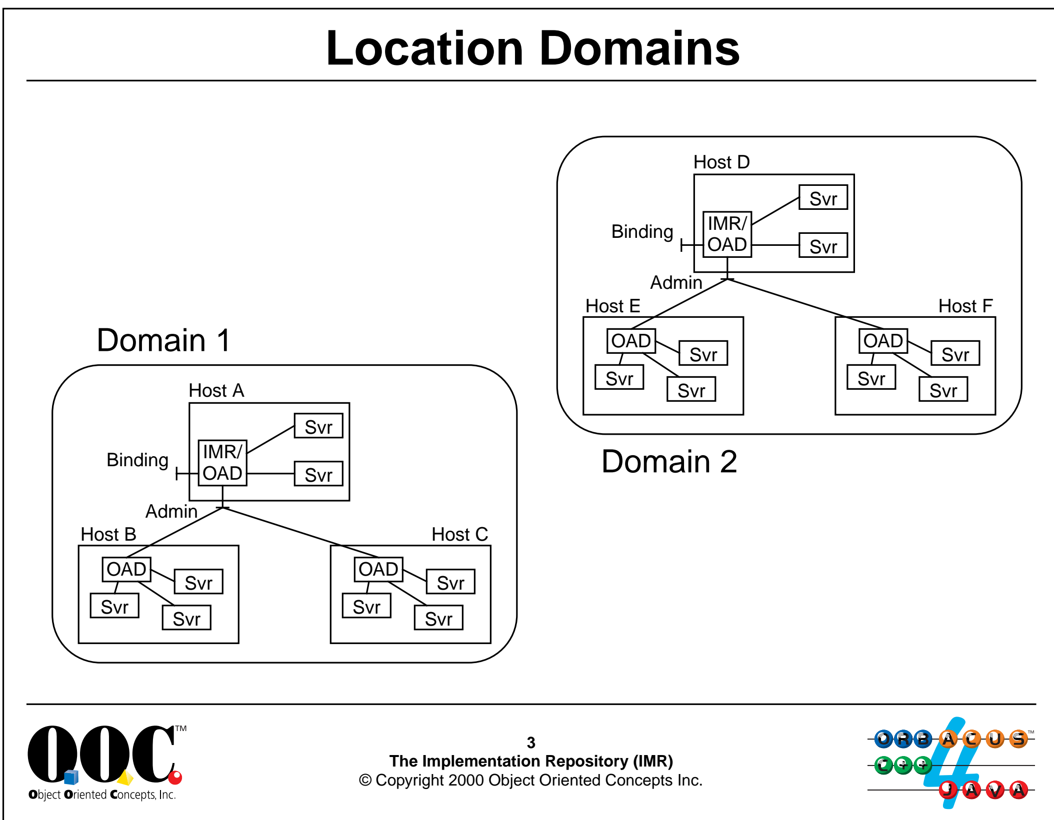
Each OAD, in turn, is configured to use a specific IMR. The IMR monitors the state its OADs; in turn, each OAD passes state changes to its IMR, such as start-up of a new server process.

The IMR uses the information it receives from its OADs (such as the host name and port number for each of the server's POAs) to resolve requests that arrive from clients. These requests arrive via the binding port of the IMR; the IMR replies with location forward replies which direct the clients to the correct server end point. The communication between the IMR and the OADs is carried out on a separate port, so administrative access to the IMR can be secured separately from the port on which clients bind requests.

Usually, the IMR on a host will also activate servers on that same host. In this case, the IMR can run in dual mode, in which a single process combines the functions of the IMR and OAD. You can also run the IMR in master mode (without the OAD funcationality) if you want to activate servers only on remote hosts. If you run the IMR in slave mode, it acts as an OAD only.

Server activation on each host is carried out by the OAD on that host: the OAD creates a new server process when instructed by the IMR.

IMRs and OADs maintain state information in a small database. This information is used mainly for error recovery. For example, if the IMR machine goes down, the IMR uses the state in its database to update its knowledge of which servers are running where.



3.6 Location Domains

Each IMR defines a location domain. All the servers that are configured to use the same IMR are part of the IMR's location domain. How you choose your location domains has influence on the reliability and performance of a system, as well as on server migration:

- You can choose to run one IMR and OAD on each machine so each machine forms its own location domain.

  The advantage of this approach is that the communication among the servers, the OAD, and the IMR is very fast because it happens via the back plane. In addition, you get high reliability with such a configuration because if one machine dies, only servers on that machine will be affected. All other servers continue to work normally.

  The down side of this approach is that you cannot move a server to another machine without breaking existing references that are held by clients.

- You can place several machines into a location domain.

  The advantage of this approach is that you can freely move servers among the machines in the location domain without invalidating references held by clients (because the IMR for the server remains the same when the server is moved).

  The down side of this approach is that the communication overhead is a little larger. This is rarely a concern because binding of a new reference via the IMR happens only once, when the reference is first used by a client. However, if the machine running the IMR dies, none of the servers in the location domain are reachable by newly connecting clients until the IMR machine is available again.

 © Copyright 2000 Object Oriented Concepts Inc.

The **`imradmin`** Tool

`imradmin` allows you to register servers with the IMR. General syntax:

`imradmin` ***`<command>`*** **`[`*`<arg>...`*`]`**

You must register each server under a server name with the IMR. The server name must be unique for that IMR:

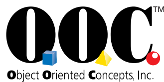
`imradmin --add-server CCS_server /bin/CCSserver`

When the IMR starts the server, it automatically passes the server name in the **`-ORBserver_name`** option. (So the server knows that it should contact the IMR.)

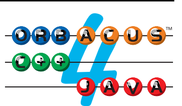
If you want to manually start a server that is registered with the IMR, you must add the **`-ORBserver_name`** option when you start the server:

`/bin/CCSserver -ORBserver_name CCS_server`

Servers automatically register their persistent POAs with the IMR.



4
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.7 The `imradmin` Tool

You must register a server with the IMR under a unique name with **`imradmin --add-server`**. This creates an entry in the IMR's server table with the supplied command line. The IMR instructs the appropriate OAD to start the server using the registered command line when client requests arrive at the IMR's binding port. You can register a server to be started on a host other than the host on which the IMR runs by supplying a host name:

```
imradmin --add-server CCS_server /bin/CCSserver HostB
```

Servers automatically register their persistent POAs with the IMR. You can disable this behavior by setting the activate-poas attribute to false (see page 3-15).

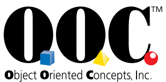
Once a server is registered, the first request from a client activates the server. Note that the OAD passes the **`-ORBserver_name`** option to the server. That option is processed by `ORB_init` and establishes communication between the server and its OAD.

Server Execution Environment

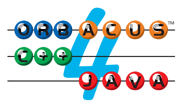
An NT server started by the IMR becomes a detached process.

A UNIX server started by the IMR has the execution environment of a daemon:

- File descriptors **`0`**, **`1`**, and **`2`** are connected to **`/dev/null`**
- One additional file descriptor is open to the OAD.
- The **`umask`** is set to **`027`**.
- The working directory is `/`.
- The server has no control terminal.
- The server is a session and group leader.
- The user and group ID are those of the OAD.
- Signals have the default behavior.



5
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.8 Server Execution Environment

Under NT, the server is created as a detached server. (The standard file descriptors are disconnected.)

Under UNIX, the server is turned into a proper daemon process. This mainly means that the working directory is the root directory and that the standard file descriptors are connected to `/dev/null`.

3.8.1 Security Issues

There are a few consequences of this environment that you need to keep in mind:

- *Never* run the OAD from the `root` user. Doing so means that all servers started by the OAD are run as `root`! The safest approach is to either run the OAD as the user **`nobody`** or to create a separate user for the OAD and use that user ID exclusively. You can ensure that the OAD starts with the correct privileges by making it set-uid and set-gid to that user.
- If you have followed the previous advice, you may find that your server has insufficient privileges to do its work. If so, the easy and reliable solution is to make the server executable set-uid and set-gid to a user and group ID with appropriate privileges. (There are no security issues with artificially raising the privilege of a process by setting s-bits —all the security holes stem from binaries that are set-uid to `root` and then do not correctly lower their privilege level when they should.)

 © Copyright 2000 Object Oriented Concepts Inc.

3.8.2 Setting the Server Environment

The environment in which a server is started by the OAD means that some things are inconvenient. For example, because the working directory is set to `/`,[1] you must use absolute pathnames for files. Similarly, you cannot simply write to the standard output for tracing because the server's file descriptors are connected to the null device.

You should consider the following points when you decide to run a server with automatic activation:

- Catch `SIGTERM`, `SIGHUP`, and `SIGINT` and ensure that you shut down cleanly and quickly on receipt of those signals.
- If your server creates child processes, pass received signals to the children; otherwise, you will leave the children abandoned. You can easily do this by using a `kill` system call with a process ID of 0 to send the signal to all processes in your process group.
- You must log to `syslog` or redirect your standard file descriptors to a terminal or file for tracing and debugging.
- You cannot rely on environment variables to be set up to anything meaningful because the environment is that of the OAD.
- Set your `umask` to something meaningful for your server. The default of `027` may not be correct for your needs.

You can do all of these things directly in your server executable. However, doing so is inconvenient and may require quite a complex configuration mechanism. A much better way to start your server with the correct file descriptors, environment variables, umask, or working directory is to not register the server with the IMR, but to register a shell script instead. The minimum skeleton for such a script is:

```
#!/bin/sh
exec "$@"
```

For example, if your script is called `/usr/local/bin/launch`, you can register the CCS server command as

```
/usr/local/bin/launch /bin/CCSserver
```

As shown, the script does nothing and effectively becomes a no-op. However, you can add whatever setup you require before the script execs the actual server. For example, you can easily change the `umask`, redirect file descriptors, or set environment variables this way:

```
#!/bin/sh
umask 077
PATH=/bin:/usr/bin:/usr/local/bin; export PATH
HOME=/tmp; export HOME
cd $HOME
exec 1>>/$HOME/CCSserver.stdout
exec 2>>/$HOME/CCSserver.stderr
exec "$@"
```

1. If you change the working directory of your server, you should change to somewhere in the root file system. If you do not, a system administrator cannot unmount the file system without killing your server.

Server Attributes

imradmin permits you to set attributes for a registered server with the **--set-server** command:

imradmin --set-server ***<server-name> <mode>***

Valid modes are:

- **exec**

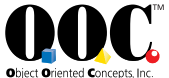
  Changes the executable path for the server.

- **args**

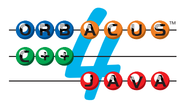
  Changes the arguments passed to the server.

- **mode**

  Changes the mode. The mode must be **shared** or **persistent**.



6
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.9 Server Attributes

The **--set-server** command allows you to change the registration details of a server.

- The **exec** attribute stores the pathname to the server executable, for example:

```
imradmin --set-server CCS_server exec /usr/local/bin/CCSserver
```

- The **args** attribute changes the additional arguments that are passed to a server, for example:

```
imradmin --set-server CCS_server args -dbfile /tmp/CCS_DB
```

  The server will now be invoked as:

```
/usr/local/bin/CCSserver -dbfile /tmp/CCS_DB
```

  Note that there are additional options beginning with **-ORB** that are used by the OAD to pass additional information to the server.

- The **mode** attribute is either persistent or shared and changes the activation mode of the server, for example:

```
imradmin --set-server CCS_server mode persistent
```

  This changes the server's mode to persistent activation.

 © Copyright 2000 Object Oriented Concepts Inc.

Server Attributes (cont.)

- **`activate_poas`**

  If **`true`**, persistent POAs are registered automatically. If **`false`**, each persistent POA must be registered explicitly.

- **`update_timeout`** (msec)

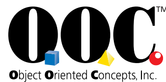
  The amount of time the IMR waits for updates to propagate.

- **`failure_timeout`** (sec)

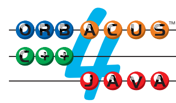
  How long to wait for the server to start up and report as ready.

- **`max_spawns`**

  The number of times to try and restart the server before giving up.
  `imradmin --reset-server` ***`<server-name>`***
  resets the failure count.



7
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



- The **`activate_poas`** attribute controls whether POAs are registered automatically with the IMR as they are created or must each be explicitly registered with the IMR. For example:

```
imradmin --set-server CCS_server activate_poas false
imradmin --add-poa CCS_server Controller
imradmin --add-poa CCS_server Controller/Thermometers
```

  With this registration, only requests for the controller and for thermometers cause server activation, but requests for thermostats do not.

- The **`update_timeout`** attribute controls how long the IMR waits for status changes to propagate to all the OADs. You should not have to change this value.

  Note that there are additional options beginning with -ORB that are used

- The **`failure_timeout`** controls the amount of time the OAD waits for a server to contact it and report as ready before the OAD concludes that the server is hanging or otherwise broken.

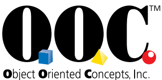
- The **`max_spawns`** attribute controls the number of times the OAD will attempt to restart a server that does not respond within **`failure_time`** before concluding that the server is permanently broken. Once in that state, the OAD will no longer activate the server until you explicitly reset the server state with the **`--reset-server`** command:

```
imradmin --reset-server CCS_server
```

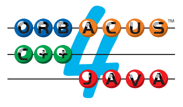
Getting IMR Status

A number of **imradmin** commands show you the status of the IMR and its OADs:

- **imradmin --get-server-info** ***<server-name>***
- **imradmin --get-oad-status [*<host>*]**
- **imradmin --get-poa-status** ***<server-name> <poa-name>***
- **imradmin --list-oads**
- **imradmin --list-servers**
- **imradmin --list-poas** ***<server-name>***
- **imradmin --tree**
- **imradmin --tree-oad [*<host>*]**
- **imradmin --tree-server** ***<server-name>***



8
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.10 Getting IMR Status

The IMR closely keeps track of the status of its OADs and servers. You can use various commands to check on the current status and the configuration of the system.

- **imradmin --get-server-info** ***<server-name>***

  This command displays the status of the specified server. You can see all the configuration attributes of the server, as well as its current status. The status of a server has one of the following values:

  - **forked**

    The OAD has created the server process, but the server has not contacted the OAD yet.

  - **starting**

    The server has initiated contact with the OAD.

  - **running**

    The server is running and ready to accept requests (that is, the server has activated the POA manager for the Root POA).

  - **stopping**

    The server has called `ORB::shutdown` but has not exited yet.

  - **stopped**

    The server is not running.

 © Copyright 2000 Object Oriented Concepts Inc.

- **imradmin --get-oad-status [*<host>*]**

  This command shows the status (**up** or **down**) of all OADs or the status for the OAD on the specified host.

- **imradmin --get-poa-status *<server-name> <poa-name>***

  This command shows the status of the specified POA. The status is that of the POA's POA manager, that is **active**, **inactive**, **holding**, or **discarding**. In addition, the **nonexistent** state indicates that the POA is not currently instantiated.

- **imradmin --list-oads**

  This command shows all OADs that are registered with the IMR.

- **imradmin --list-servers**

  This command lists all servers that are registered with the IMR.

- **imradmin --list-poas *<server-name>***

  This command lists all the POA names that are known to the IMR for the specified server.

- **imradmin --tree**

  This command lists the complete status of the IMR as a tree structure.

- **imradmin --tree-oad [*<host>*]**

  This command limits the OAD status display to the specified host.

- **imradmin --tree-server *<server-name>***

  This command limits the server status display to the specified server.

IMR Configuration

1. Set up a configuration file for each host in the location domain.
2. Run an IMR in master or dual mode on exactly one machine in your location domain.
3. Run an OAD on each of the other hosts in the location domain by running the IMR in slave mode.

Once you have configured the IMR, run the **`imr`** commands from a start-up script in **`/etc/rc`**.

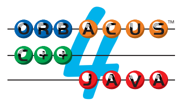
You can explicitly add an OAD (instead of having OADs add themselves implicitly) with:

```
imradmin --add-oad [<host>]
```

To remove an OAD from the configuration:

```
imradmin --remove-oad [<host>]
```

9
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.11 IMR Configuration

Configuration of the IMR is quite easy:

1. Create a configuration file for each machine on which you want to run an IMR or OAD.
2. Run an IMR in master mode or dual mode on exactly one of those machines.
3. Run an OAD on each of the other hosts by running the IMR in slave mode on those machines.

This completes the configuration. You can now register servers with **`imradmin`** as discussed in the previous sections.

`imradmin` also offers commands to remove items from the configuration:

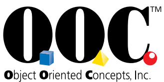
- **`imradmin --remove-oad [<host>]`**
- **`imradmin --remove-server <server-name>`**
- **`imradmin --remove-poa <server-name> <poa-name>`**

 © Copyright 2000 Object Oriented Concepts Inc.

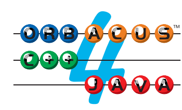
IMR Properties

The IMR and OAD use configuration properties:

- **ooc.imr.dbdir=<*dir*>**
- **ooc.imr.forward_port=<*port*>**
- **ooc.imr.admin_port=<*port*>**
- **ooc.imr.administrative=<true/false>**
- **ooc.imr.slave_port=<*port*>**
- **ooc.imr.mode=<dual/master/slave>**
- **ooc.orb.service.imr=<*corbaloc URL*>**
- **ooc.imr.trace.peer_status=<*level*>**
- **ooc.imr.trace.process_control=<*level*>**
- **ooc.imr.trace.server_status=<*level*>**



10
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.12 IMR Properties

The IMR reads configuration properties on start-up. These properties determine how the various processes connect to each other and where they store their persistent state. You can control these properties by setting the **ORBACUS_CONFIG** environment variable to the pathname of a configuration file, or you can point a server at a configuration file with the **-ORBconfig <pathname>** option. By controlling these properties, you can run more than one IMR and OAD on a single host and therefore have servers on the same machine use different location domains. This is particularly useful during development and for debugging purposes.

- **ooc.imr.dbdir=<*dir*>**

  This property controls where the IMR stores its persistent state. The current directory is the default.

- **ooc.imr.forward_port=<*port*>**

  This property determines the port number that is written into each persistent IOR. In other words, this is the port number on which client requests are forwarded. The default port is 9998.

- **ooc.imr.admin_port=<*port*>**

  This property determines the port number on which the IMR communicates with its OADs when it is in dual or master mode.

- **ooc.imr.administrative=**<**true**/**false**>

  This property controls whether the IMR runs in administrative mode. When in administrative mode, the IMR accepts modifications to its registration database. When this mode is disabled,

 © Copyright 2000 Object Oriented Concepts Inc.

the registration database cannot be modified, that is, it is impossible to change registration of OADs, servers, server attributes, and POAs. If you run an IMR in administrative mode, make sure that the administrative port is not accessible through your firewall!

- **ooc.imr.slave_port=*<port>***

  This property determines the port number on which the OAD communicates with its IMR. You must set this property on all hosts on which the IMR or an OAD runs, otherwise the IMR cannot find its OADs. (All OADs must use the same port.) The default port is 9997.

- **ooc.imr.mode=**<**dual**/**master**/**slave**>

  This property controls the mode in which the IMR runs. In dual mode, it acts as both the IMR and as an OAD for the local host. In master mode, the IMR acts as a pure IMR (that is, it cannot activate server processes on the local host). In slave mode, the IMR acts as a pure OAD (that is, expects an IMR to be running elsewhere).

- **ooc.orb.service.IMR=*<corbaloc URL>***

  This property contains the IOR to the implementation repository. The IOR denotes the IMR's binding port (**ooc.imr.forward_port**) and is required by tools such as **imradmin**. You can use a `corbaloc` IOR (see Section 19.20) with an object key of `IMR` to set this property:

  ```
  ooc.orb.service.IMR=corbaloc::janus.ooc.com.au:9999/IMR
  ```

  Note that the port number in the IOR must match the setting of **ooc.imr.forward_port**. You must set this property also on each machine that runs an OAD; otherwise, the OAD will not know where to find its IMR.

  Setting this property controls the reference that is returned by a call to `resolve_initial_references` with a token of `IMR`.

- **ooc.imr.trace.peer_status=*<level>***

  This property controls the tracing output for process control messages sent by the IMR to its OADs. Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

- **ooc.imr.trace.process_control=*<level>***

  This property controls the tracing output for server process (server start-up and shut-down). Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

- **ooc.imr.server_status=*<level>***

  This property controls the tracing output for diagnostic messages. Valid levels are 0, 1, and 2. The default level is 0, which produces no output.

© Copyright 2000 Object Oriented Concepts Inc.

The `mkref` Tool

You can create an object reference on the command line:

`mkref <server-name> <object-ID> <poa-name>`

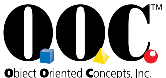
For example:

`mkref CCS_server the_controller Controller`

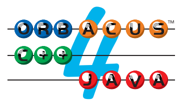
This writes a **`corbaloc`** reference to standard output that you can use to configure clients:

```
corbaloc::janus.ooc.com.au:9998/%AB%AC%AB0_RootPOA%00forward%00
%00%AB%AC%AB0CCS_server%00Controller%00%00the_controller
```

`mkref` is useful during installation, for example, if you want to produce an IOR for bootstrapping.



11
The Implementation Repository (IMR)
© Copyright 2000 Object Oriented Concepts Inc.



3.13 The `mkref` Tool

Especially during installation of your application, it is useful to be able to create references to bootstrapping objects without having to run the actual server that hosts the object. The **`mkref`** command allows you to do this. Note that, as for any URL-style IOR, you should not use references created with **`mkref`** as a general IOR replacement. They are purely a convenience.

 © Copyright 2000 Object Oriented Concepts Inc.