# CORBA-II

Prof. David Bakken

Cpt. S 464/564 Lecture

Sept 20, 2000

# Administrative Items

- Handouts today
  - These slides
- Notes on Project1
  - PROG and REF copies now in ETRL 301, two sets per language
  - You can ssh into the hosts in ETRL 301, but don't do until Project #2….
  - There will be other students running their programs at the same time, so always prefix any "naming string" with your username
    - E.g., "_bakken_BankManager", not "BankManager"
    - E.g., "/_bakken_bank_agent_poa", not "/bank_agent_poa"

    At least, do this if you don't want you (and others) to have very interesting debugging sessions…
  - You will be required to annotate a printout of IDL-generated code for a proxy, and turn it in, just like I handed out last week (Bank_c_chopped.*)
    - Oval around names of module or interface
    - Rectangle around names of methods
    - Underline parameter names and return values

    Keep it to one page for each of the 2 files (.cpp and .hh)
- Note: some info in some of these slides were borrowed from excellent slides by Doug Schmidt and the very good Hennig and Vinoski book "Advanced CORBA Programming with C++"

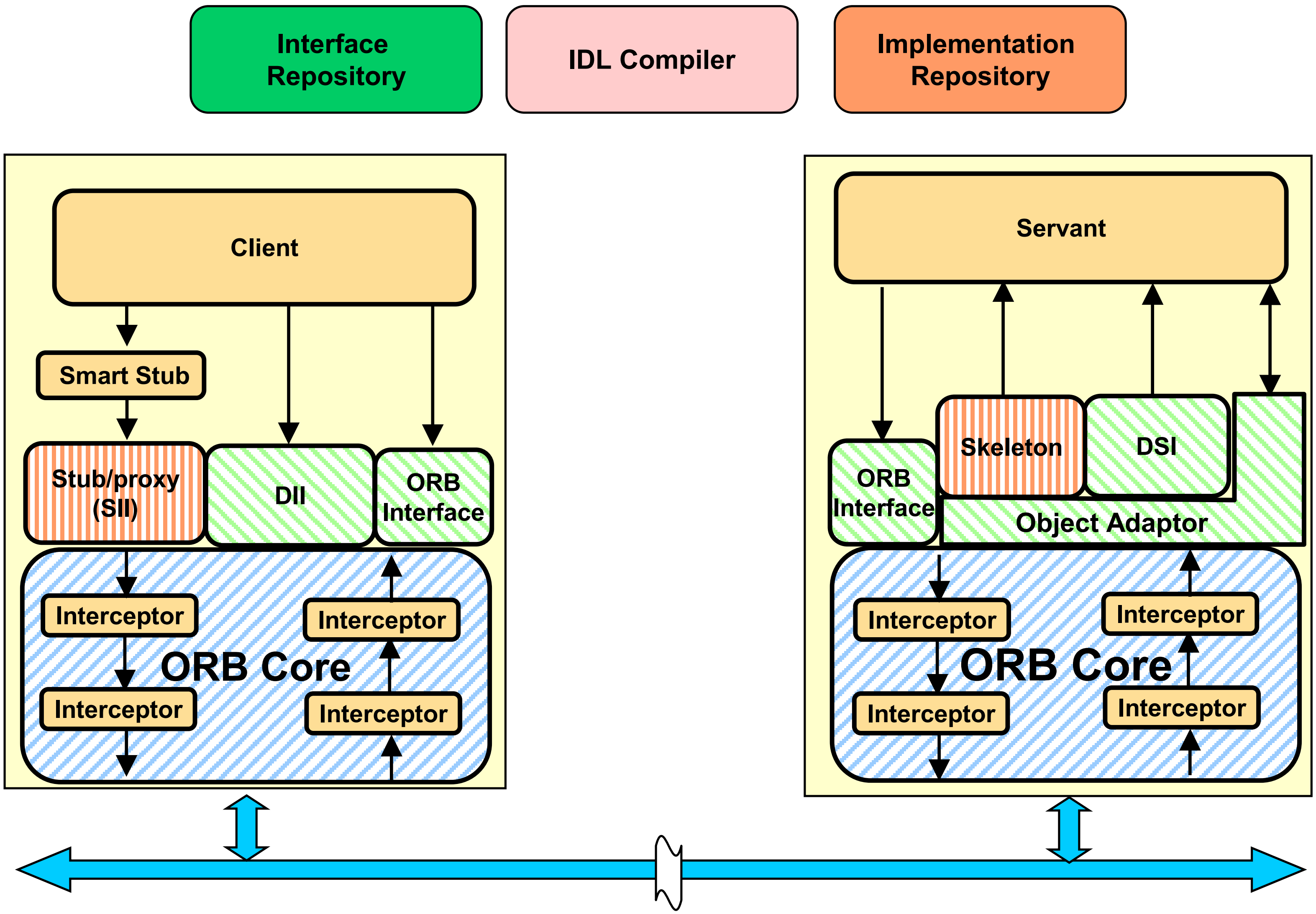
# Outline

- CORBA Features and Hooks
- Portable Object Adaptor (POA)

# Major CORBA Design Principles

- Separation of interface and implementation
  - Clients depend on interfaces, not implementations
- Location transparency
  - Service use is orthogonal to service location
- Access transparency
  - Invoke CORBA objects just like local ones
- Typed interfaces
  - Object references are typed by interfaces
- Support of multiple inheritance of interfaces
  - Inheritance extends, evolves, and specialized behavior
  - Note: not implementation of multiple implementations!
- Support of multiple interaction styles
  - Client/server
    - Some support for mobile code, too, with Objects by Value (OBV)
  - Peer processes
  - Publish/Subscribe (aka "push")

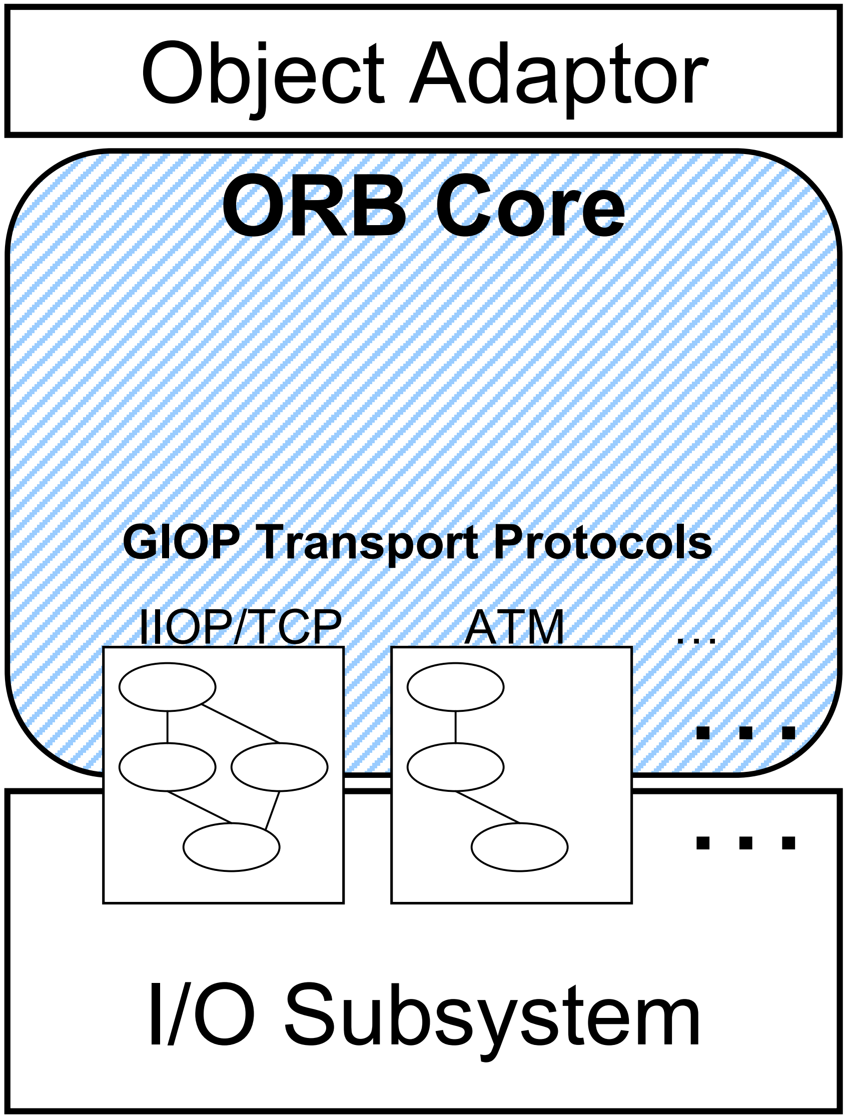
# CORBA Components and System Hooks



▨ Standard Interfaces ▥ IDL-generated ▧ ORB-Specific

(This slide adapted from FTCS-29 Tutorial
by Shalini Yajnik of Lucent Technologies)

# ORB Core Overview



Features (server-side)

- Connection management
- Memory management
- Request transfer
- Endpoint demuxing
- Concurrency control

Other Features

- object_to_string() and string_to_object()
- Etc.

# CORBA:Object class

- Base class for all proxies
- Useful utility methods:
  - `_is_a()`
  - `_is_equivalent()`
  - `_duplicate()`
  - `_release()`
  - `_is_local()`
  - `_is_remote()`
- Request methods for DII (more soon…)

# SII and DII

- Static Invocation Interface (SII)
  - Most common way to use IDL
  - All operations specified in advance and known to client (by proxies/stubs) and server (by skeletons)
  - Simple
  - Typesafe
  - Efficient
- Dynamic Invocation Interface (DII)
  - Less common way to use IDL
  - Lets clients invoke operations on objects whose IDL is not known to them at compile time (main advantage of DII)
    - Browsers of all sorts (interface browser, etc)
    - Debuggers
  - Also can use `send_deferred()` and `poll_response()`
  - Clients construct a CORBA::Request (local) object, “pushing” arguments and operation name etc. on it like a stack
    - Exactly what a proxy does: same API to ORB Core

# DII Example

- Notes (See PROG manual for more details (Chap 22 for VBCPP) ):
  - CORBA::Request object represents one invocation of one method of one CORBA object
  - CORBA::Any encapsulates any CORBA type
  - Example is from /local/dist_systems/cs564/vbcpp4_0/examples/basic/bank_dynamic :

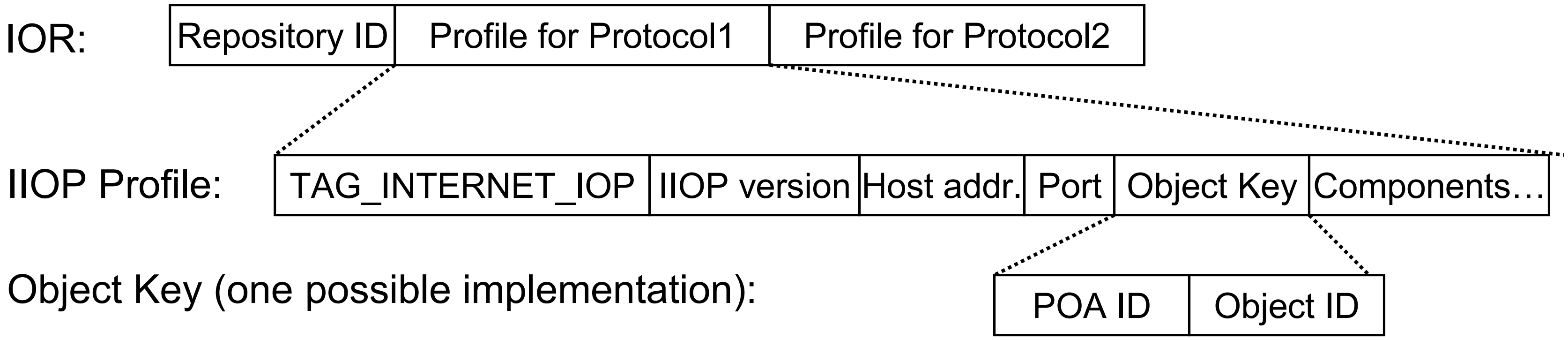
```
// Create request that will be sent to the manager object
CORBA::Request_var request = manager->_request("open");
// Create argument to request
CORBA::Any customer;
customer <<= (const char *) name;
CORBA::NVList_ptr arguments = request->arguments();
arguments->add_value( "name" , customer, CORBA::ARG_IN );
// Set result type
request->set_return_type(CORBA::_tc_Object);
// Invoke operation.  NOTE: VisiBroker example used send_deferred()
request->invoke();
// Get the return value
CORBA::Object_var account;
CORBA::Any& open_result = request->return_value();
open_result >>= CORBA::Any::to_object(account.out());
```

# Object References

- Object reference
  - Opaque handle for client to use
  - Identifies exactly one CORBA object
  - IOR == “Interoperable Object Reference”
- References may be passed among processes on different hosts
  - As parameters or “stringified”
  - ORB will convert into form suitable for transmission over network
  - ORB on receiver side will create a proxy and return a pointer to it

# Object References

- Object reference
  - Opaque handle for client to use
  - Identifies exactly one CORBA object
  - IOR == “Interoperable Object Reference”
- References may be passed among processes on different hosts
  - As parameters or “stringified”
  - ORB will convert into form suitable for transmission over network
  - ORB on receiver side will create a proxy and return a pointer to it



IOR: Repository ID | Profile for Protocol1 | Profile for Protocol2

IIOP Profile: TAG_INTERNET_IOP | IIOP version | Host addr. | Port | Object Key | Components…

Object Key (one possible implementation): POA ID | Object ID

- Object Key
  - Opaque to client
  - ORB-specific
- Object ID
  - Can be created by user or POA (more in POA slides…)

# Interface Repository

- Stores information on interfaces which can be looked up later by others at runtime.  Tells about
  - Interface names
  - Method signatures
  - …
  - Exactly the information in an IDL file.
- Allows for runtime discovery of interfaces.
  - Can be used by other useful hooks, such as the DII, DSI, and Interceptors.

# Implementation Repository (IR)

- Stores information on the implementations available for a given interface
  - Mainly bindings between interface names and executable files that implement them
- This allows the ORB to activate servants to process object invocations
  - Visibroker’s IR is called theObject Activation Daemon (OAD)
  - More details are in PROG (VBCPP Chapter 20)

# Outline

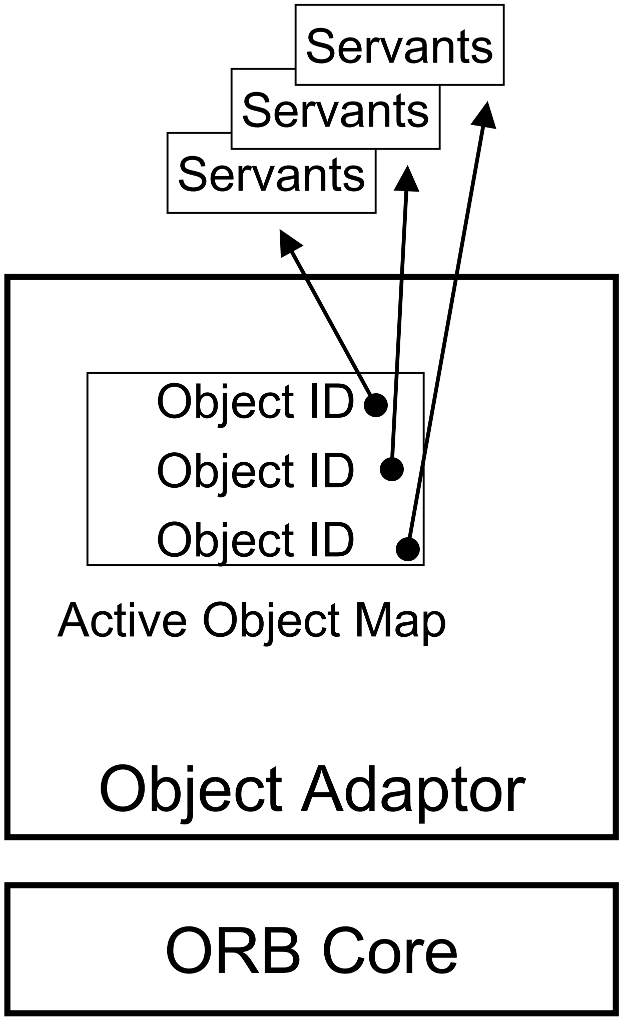
- CORBA Features and Hooks
- <u>Portable Object Adaptor (POA)</u>
  - <u>POA Terminology</u>
  - <u>POA Policies</u>
  - <u>POA example revisited</u>

Note: POA is <u>very</u> complicated and cannot be understood even halfway in this class – you will generally just cut-and-paste the POA code, only modifying it when you need to.  A basic understanding of the POA is very useful, however…

# POA Terminology I

- CORBA Object: a “virtual” entity capable of
  - Being located by an ORB
  - Having client requests delivered to it
- Servant: programming language construct that
  - Exists in the context of a server
  - Implements the functionality of a CORBA object
- Object Adaptor (OA): a component which connects a server-side ORB with a servant
- Note: CORBA is to a servant like virtual memory is to physical memory!
  - VM does not really exist….
  - VM can be read and written with help of a computer’s MMU and its mapping from VM to PM
  - ORB and OA cooperate to ensure that each CORBA object is mapped onto a servant

# POA Overview

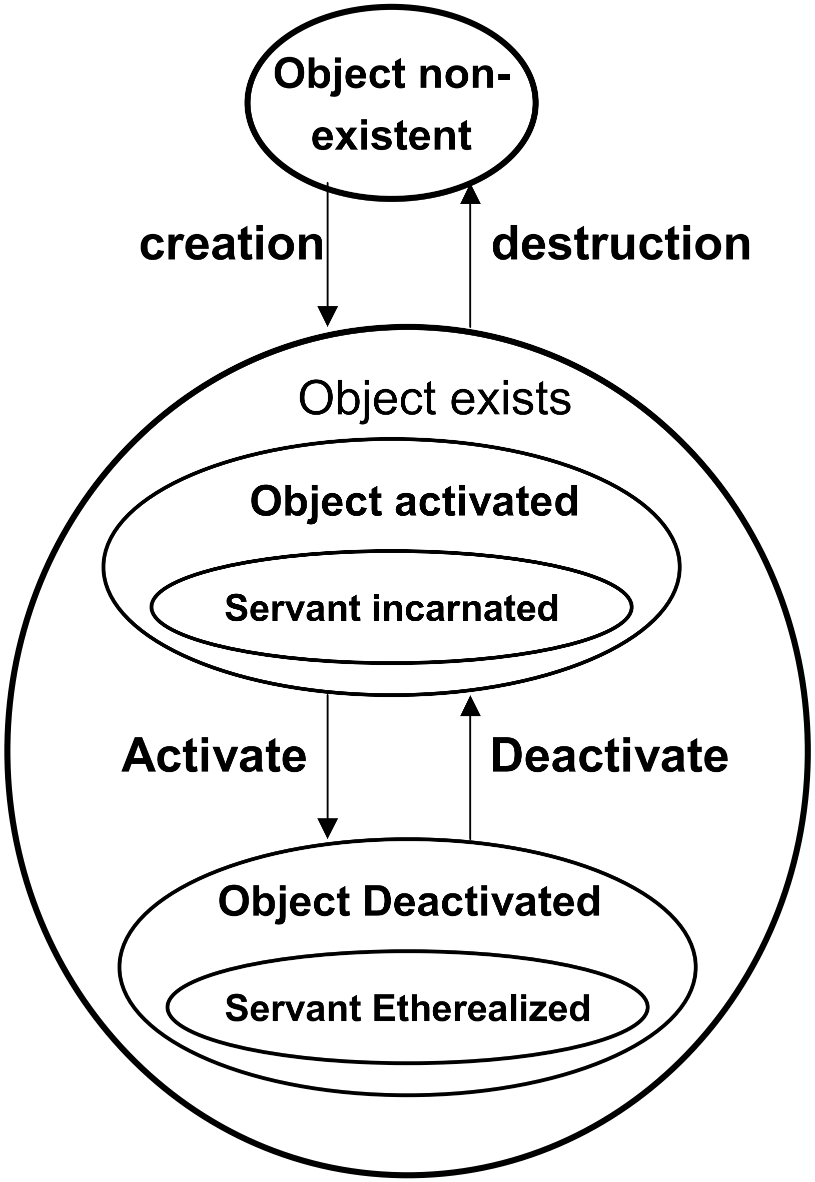


- POA Features
  - Activate and deactivate CORBA objects
  - Incarnate and etherealize servants
  - Create and manage object references
  - Map requests to servants
- POA designed to provide a great deal of flexibility that server writers can utilize for
  - Scalability
  - Memory and other resource usage
  - Flexibility in mapping requests onto servants

# POA Terminology II

- Skeleton: a programming language entity that connects a servant to an OA
  - This lets the OA to dispatch requests to the servant
  - In C++, a skeleton is a base class from which the servant derives
- Object ID: an identifier used to “name” an object within the scope of an OA
  - It is *not* guaranteed to be unique outside a given instance of an OA
  - May be defined by server programmer
  - May be automatically generated by POA

# POA Terminology III: Lifecycles



- Creation: the act of making a new CORBA object
  - Always results in an object reference
- Activation: starting an existing CORBA object to allow it to service requests
- Deactivation: the act of shutting down a CORBA object, including removing any associations with any servants
- Incarnation: the act of associating a servant with a CORBA object
- Etherealization: destroying the association between a servant and a CORBA object

# POA Lifecycle Notes

- Activation and deactivation happen to CORBA objects
- Incarnation/etherealization happen to servants
- But activation/incarnation and deactivation/etherealization sometimes happen at same time
- Not necessarily a 1:1 mapping between CORBA objects and servants
  - A CORBA object may be represented by one or more servants over its lifetime
  - A servant may represent  one or more CORBA objects simultaneously

# POA Policies

- A server application can have multiple POA instances in it
  - Each POA instance has the same set of characteristics
  - But different POA instances can have different policies
  - 7 different kinds of POA policies that define its characteristics
  - Each policy has an interface in module PortableServer, with 1 attribute
    - Below, "(D)" means the default policy for VisiBroker for C++
- CORBA Object Lifespan: LifespanPolicy
  - PERSISTENT: all CORBA objects can live beyond the lifetime of the particular process they were created by
  - TRANSIENT: (D) CORBA object does not live beyond its creating process
    - Require less bookkeeping from the ORB
    - Great for "temporary" objects such as callback objects
- Object Identifiers: IdAssignmentPolicy
  - An object identifier is just a stream of octets, opaque to applications
  - But can be useful to some applications to manage
    - Database key
    - Employee ID
  - SYSTEM_ID: (D) POA generates object identifiers
  - USER_ID: application provides its own object identifiers

# POA Policies (cont.)

- Mapping Objects to Servants: IdUniquenessPolicy
  - One extreme: app with only a few transient objects creates separate servants for each one; state of each object kept in its servant
  - Other extreme: app with many persistent CORBA objects may want only one servant to incarnate them all, e.g., for memory efficiency
    - May keep object state in a database or other external persistent store
  - UNIQUE_ID: (D) each object ID maps onto a different servant
  - MULTIPLE_ID: multiple Ids can map onto a single servant
- Implicit Activation: ImplicitActivationPolicy
  - Allow CORBA objects to be created and activated implicitly (sometimes through a language shortcut like `_this` in C++)?
  - IMPLICIT_ACTIVATION
  - NO_IMPLICIT_ACTIVATION (D)

# POA Policies (cont.)

- Matching Requests to Servants: RequestProcessingPolicy
  - Controlling the associations between CORBA objects and servants
  - USE_ACTIVIE_OBJECT_MAP_ONLY: (D) explicit activation and incarnation
  - USE_SERVANT_MANAGER: manager object is registered with POA and called if an invocation arrives for an object with no servant bound to it
    - Action can depend on IdUniquenessPolicy: create new servant or reuse existing one
  - USE_DEFAULT_SERVANT: incarnates all CORBA objects for a POA
- Object ID to Servant Associations: ServantRetentionPolicy
  - RETAIN: (D) keeps association across multiple invocations
  - NON_RETAIN: each arriving request invokes application to obtain the servant
    - Can control allocation of servants to CORBA objects
    - Useful for controlling memory usage
- Allocation of Requests to Threads: ThreadPolicy
  - SINGLE_THREAD_MODEL: all requests for all objects in a POA serealized
  - ORB_CTRL_MODEL: (D) ORB chooses an “appropriate” threading model
    - Really need more choices: thread pool model, thread-per-request, thread-per-object

# Sample: Setting up a POA with a Servant

- Generic steps
  - Obtaining Reference to the root POA
  - Defining the policies of the POA
  - Creating a POA as a child of the root POA
  - Creating a servant and activating it
  - Activating the POA through its manager
- Now go over code from Bank example, given in CORBA-I lecture…

# POA Steps on the Server

```
include "BankImpl.h"

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

    // get a reference to the root POA
    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

    CORBA::PolicyList policies;
    policies.length(1);
    policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
                              PortableServer::PERSISTENT);
```

# POA Steps on the Server (cont.)

```
// get the POA Manager
PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

// Create myPOA with the right policies
PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
                                        poa_manager, policies);
// Create the servant
AccountManagerImpl managerServant;

// Decide on the ID for the servant
PortableServer::ObjectId_var managerId =
                 PortableServer::string_to_ObjectId("BankManager");
// Activate the servant with the ID on myPOA
myPOA->activate_object_with_id(managerId, &managerServant);

// Activate the POA Manager
poa_manager->activate();
```

# POA Steps on the Server (cont.)

```
     CORBA::Object_var reference = myPOA->servant_to_reference(
                                                   &managerServant);
    cout << reference << " is ready" << endl;

    // Wait for incoming requests
    orb->run();
  }
  catch(const CORBA::Exception& e) {
    cerr << e << endl;
    return 1;
  }
  return 0;
}
```