

E CORBA Programming

E.1 Introduction

- Provide information on how to write CORBA applications
- Primary implementation language is Java
- C++ as an additional language – after all CORBA is cross-language
- No repetition of general CORBA concepts – see Lecture
- Focus on practical problems
- Some advanced topics to understand special features

1 CORBA

- Common Object Request Broker Architecture and Specification (CORBA)
 - The Core Spec

- Other specs based on CORBA with independent revisions
 - ◆ CORBAservices
 - ◆ (CORBAfacilities)
 - ◆ Domain Interfaces
 - ◆ CORBA Component Model
 - ◆ (Unified Modelling Language & Meta Object Facility)

- Numerous Task Forces and Special Interest Groups within the OMG
 - ◆ Addition of new concepts and extensions
 - ◆ Revision of existing standards

- All OMG specs are "moving targets"

2 CORBA Versions

- CORBA 1.x
 - ◆ CORBA object model and architecture
 - ◆ Interface Definition Language (IDL)
 - ◆ Language mappings for C, C++, and Smalltalk

- CORBA 2.0 (July 1996)
 - ◆ Interoperability through IIOP as a required protocol

- CORBA 2.1 (August 1997)
 - ◆ IDL extensions
 - ◆ New language mappings (Cobol, Ada)

- CORBA 2.2 (February 1998)
 - ◆ Portable Object Adaptor (POA) replaces Basic Object Adaptor (BOA)
 - ◆ New language mapping (Java)

2 CORBA Versions (2)

- CORBA 2.3/2.3.1 (June/October 1999)
 - ◆ Revised language mappings to adapt to POA spec
 - ◆ Valuetypes, object-by-value parameters
 - ◆ Separate documents for language mappings

- CORBA 2.4/2.4.1 (October/November 2000) – the current version
 - ◆ CORBA Messaging
 - ◆ Minimum CORBA
 - ◆ Real-time CORBA

- CORBA 3 (???) – probably the next official release
 - ◆ Huge hype
 - ◆ CORBA Component Model

3 Information on CORBA

- If you really want to know what CORBA is all about, you will ultimately have to read the specs!
- Specs are publically available
- OMG Web site
 - ◆ <http://www.omg.org/>
- Local mirror of interesting OMG documents
 - ◆ <file:/proj/i4doc/CORBA/OMG/docs/index.html>
 - ◆ Also available via the OODS Tutorial Web pages
- Lots of books of varying quality
 - ◆ List of selected titles on the OODS Tutorial WWW pages
- Beware of CORBA product documentation!
 - ◆ Often describes proprietary extensions

4 CORBA Products versus the CORBA Standard

- No established CORBA branding yet
 - ◆ Anyone can claim to be CORBA version x.y compliant
 - ◆ Open Group recently started certification

- CORBA vendors introduce(d) proprietary extensions
 - ◆ Fine as long as you don't rely on them
 - ◆ In former times there was no way round, e.g. BOA
 - ◆ Nowadays there is a standard-compliant way of achieving almost anything

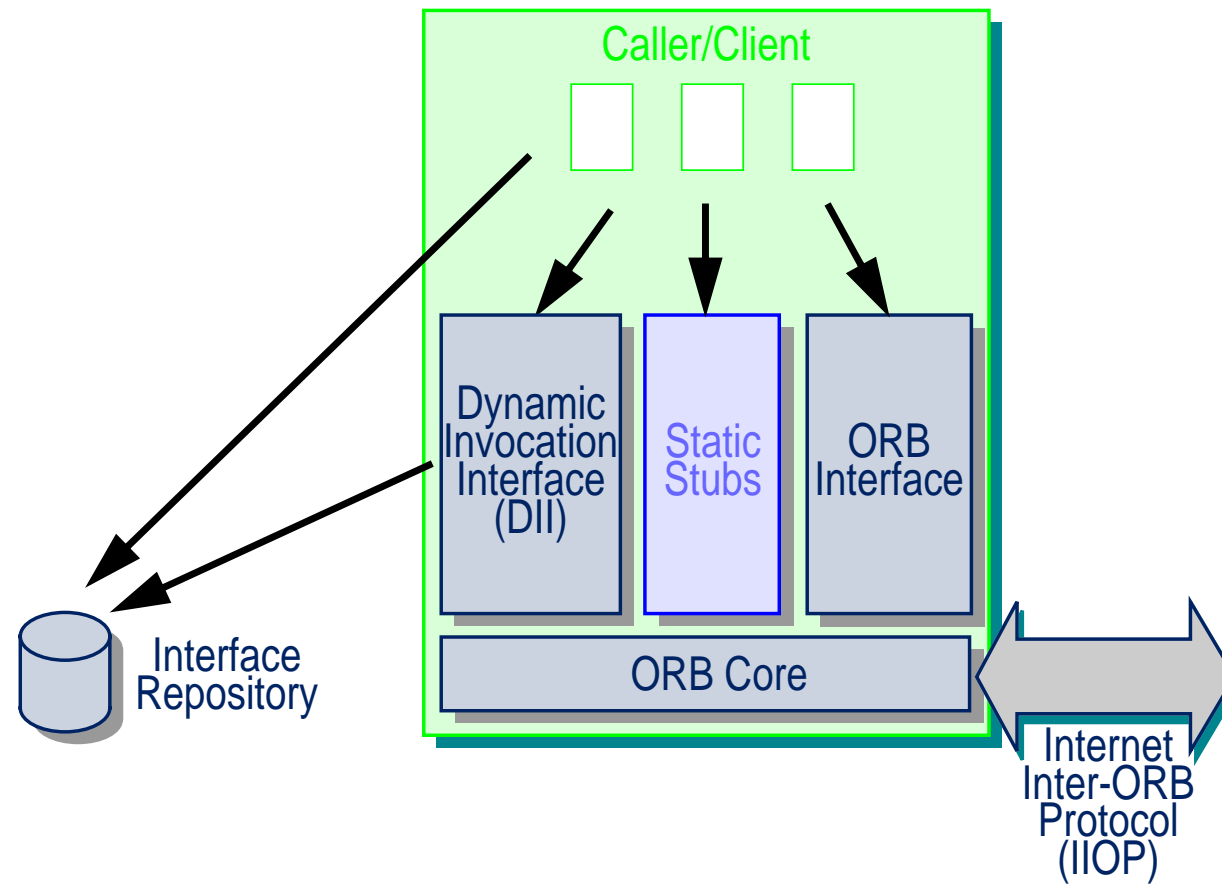
- Some features in products are not 100%-compliant with the specs
 - ◆ E.g. language mappings
 - ◆ Specs change, products change a little later

E.2 Using CORBA Objects

1 Caller's View of CORBA Objects

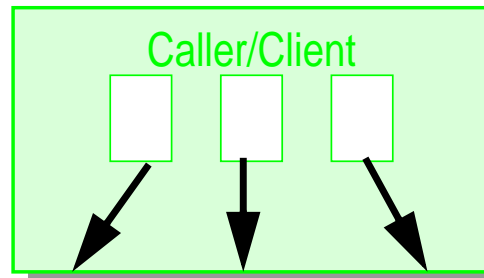
- CORBA objects have (exactly one) interface
 - ◆ Description of interface in the CORBA Interface Definition Language (IDL)
 - ◆ IDL interfaces are a contract between the CORBA object and its callers
- Callers of a CORBA object only have an opaque object reference
 - ◆ Invoke operations on the objects via reference
 - ◆ No need to know whether object is local or remote
 - ◆ Query objects interface
 - ◆ Create invocations at run-time (Dynamic Invocation)
- Object Request Broker (ORB) transmits invocations and replies
 - ◆ Only the ORB can interpret object references
- **Caller/Client is a *role* for one invocation only, e.g. callbacks**

2 Caller/Client Architecture



3 The Client

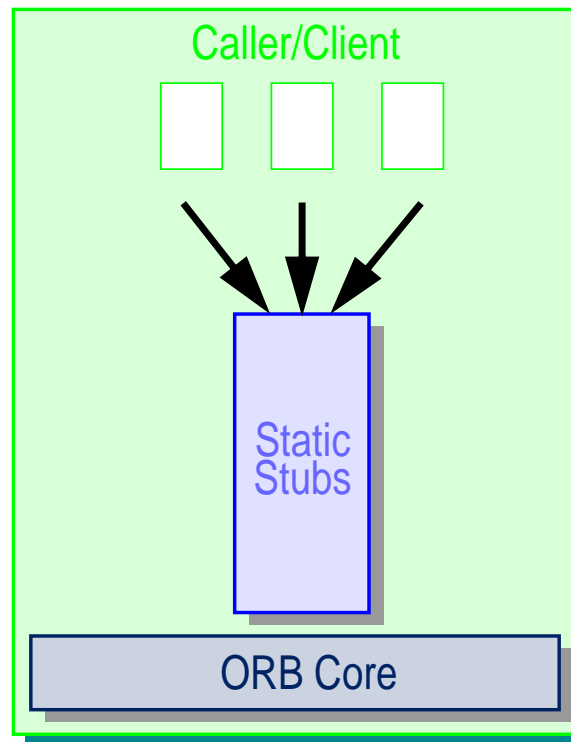
- Invokes operations on CORBA objects
- Doesn't have to be a (CORBA-) object itself



- ◆ Your first CORBA programs will only have a **static main()** method

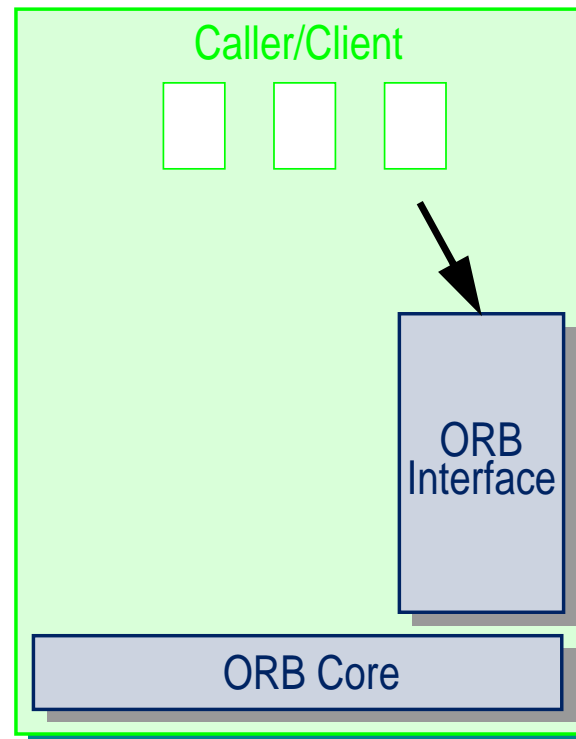
4 Static Stubs

- Can be automatically created from the IDL interface
- Marshalling of invocation parameters
- Demarshalling of return values or exceptions from the invocation



5 ORB Interface

- Export of initial object references (ORB, POA, Services, ...)
- Manipulation of object references (conversion into strings and back)

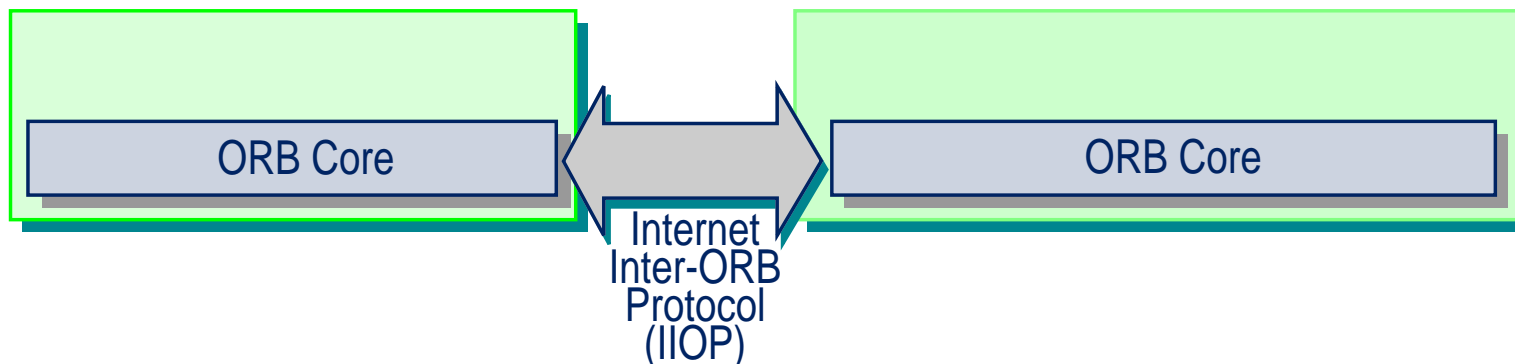


6 ORB Core

- Transmission of invocations using information in object references

7 General Inter-ORB Protocol (GIOP)

- Standard transmission protocol between ORBs
- Basis of interoperability
- GIOP over TCP connections is the Internet Inter-ORB Protocol (IIOP)
- Every CORBA 2.x ORB must implement IIOP



8 Caller/Client Summary

- Don't have to be CORBA objects themselves
- May invoke operations on CORBA objects
- Opaque object references
- ORB transmits invocation data

E.3 Interface Definition Language

- Identifiers
- Primitive types
- Constructed types
- Interfaces of CORBA objects
- Valuetypes
- Design issues

1 Basics

- IDL is for the description of data types and interfaces
- Independent of the implementation language(s)
- Syntax is strongly based on C++
 - ◆ Only description of data and interfaces (types, attributes, methods, ...)
 - ◆ No control statements (if, while, for, ...)
- Pre-processor like in C++
 - ◆ `#include` to include other IDL files
 - ◆ `#define` for macros
- Comments like in C++ and Java:

```
// This is a single-line comment
/*
 * This is a multi-line comment
 */
```

2 Identifiers

- Various reserved words
 - ◆ `module`, `interface`, `struct`, `void`, `long`, ...
- Any other combination of small and capital letters, numbers and underscores allowed
 - ◆ No numbers at the beginning of an identifier
- Once an identifier is used, any variation that has the same combination of letters but different capitalisation becomes illegal!

- Example:

```
module Example1 { ... };  
module eXample1 { ... }; // illegal in IDL
```

- Rationale:
 - ◆ Allow mapping of IDL to languages that are not case-sensitive
 - ◆ Preserve identifiers for case-sensitive languages

3 Modules

- Name space (scope) for IDL declarations

- Syntax:

```
module name {  
    Declarations  
};
```

- Access to other scopes via :: operator

- Example:

```
module Example1 {  
    typedef long IDNumber;  
};  
module Example2 {  
    typedef Example1::IDNumber MyID;    // typedef long MyID;  
};
```

4 Type Declarations

- Alias for an existing type

- Syntax:

```
typedef existing_type alias;
```

- Example:

```
typedef long IDNumber;
```

5 Primitive Types

■ Integer numbers

◆ short	-2^{15} to $2^{15}-1$
◆ unsigned short	0 to $2^{16}-1$
◆ long	-2^{31} to $2^{31}-1$
◆ unsigned long	0 to $2^{32}-1$
◆ long long	-2^{63} to $2^{63}-1$
◆ unsigned long long	0 to $2^{64}-1$

■ Floating point numbers (IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985)

◆ float	single precision
◆ double	double precision
◆ long double	extended precision (at least 15 bit exponent and 64 bit base)

5 Primitive Types (2)

■ Characters

- ◆ `char` ISO 8859-1 (Latin1) character
- ◆ `wchar` multi-byte character (e.g. Unicode)
- ◆ Length is dependent on implementations and programming languages

■ `boolean`

- ◆ The only values are TRUE and FALSE

■ `octet`

- ◆ 8 bit length
- ◆ No conversion at all during transmission

■ `any`

- ◆ Can encapsulate any CORBA-defined type

■ `void`

6 Structures

- Grouping of several types in a structure
- Syntax:

```
struct name {  
    Declaration of structure elements  
};
```

- Example:

```
struct AmountType {  
    float value;  
    char currency;  
};
```

- Usage:

```
AmountType amount;
```

6 Nested Structures

- Structures can be defined within other structures
- Example:

```
struct AmountType {  
    struct ValueType {  
        long integerPart;  
        short fractionPart;  
    } amount;  
    char currency;  
};
```

- Structures create a name space (scope) of their own!
- Complete name of the above type:

```
AmountType::ValueType
```

7 Unions

- Union of different types that are distinguished by the value of a switch type

- Syntax:

```
union name switch( switch_type ) {  
    case switch_constant: Declaration  
    ...  
    default: Declaration  
};
```

- Possible switch types: integers, characters, `boolean`, enumerations

- Declarations have to be unique

- Example:

```
union Example switch( long ) {  
    case 1:          long l;  
    case 2:          float f;  
};
```

8 Enumerations

- Enumerations with declared set of possible values

- Syntax:

```
enum name {  
    value1, value2, ...  
};
```

- Example:

```
enum Color {  
    GREEN, RED, BLUE  
};
```

- Caution: Enumerations do not create a scope of their own!

- Access to enumeration values:

```
GREEN  
not Color::GREEN
```

9 Arrays

- Single or multi-dimensional arrays
 - ◆ Fixed size in each dimension

- Syntax:

```
typedef element_type name[positive_constant][positive_constant]...;
```

- Example:

```
typedef long Matrix[3][3];
```

- Caution:

Array types have to be declared with `typedef` before they can be used!

10 Sequences

■ Single dimension array

- ◆ Variable size
- ◆ Optional maximum size (bounded sequence)

■ Syntax:

```
typedef sequence<element_type> name;           // unbounded
typedef sequence<element_type, positive_constant> Name; // bounded
```

■ Example:

```
typedef sequence<long> Longs;
typedef sequence< sequence<char> > Strings;
```

■ Caution:

Sequence types also have to be declared with `typedef` before they can be used!

11 Strings

■ Character strings

- ◆ Similar to `sequence<char>` and `sequence <wchar>`
- ◆ Are special types for performance reasons
- ◆ No need to declare strings with `typedef`

■ Syntax:

```
typedef string name; // unbounded
typedef string<positive_constant> name; // bounded
typedef wstring name; // unbounded
typedef wstring<positive_constant> name; // bounded
```

■ Example:

```
typedef string<80> Name;
```

12 Fixed-Point Numbers

- Similar to integer numbers
 - ◆ At most 31 digits
 - ◆ Scaling factor for decimal point

- Syntax:

```
typedef fixed<positive_constant, scaling_constant> name;
```

- Example:

```
typedef fixed<10,2> Amount;
```

- Caution:
Not yet implemented in most ORBs!

13 Constants

- Symbolic name for special values

- Syntax:

```
const type name = constant_expression;
```

- Constant expressions

- ◆ Constant values
- ◆ Operations with all constant values
- ◆ Arithmetic operations
- ◆ Logic operations

- Example:

```
const Color WARNING = RED;
```

14 Interfaces

- Visible interface of CORBA objects
- Contain:
 - ◆ Attributes
 - ◆ Operations
 - ◆ Local types, constants, and exceptions

- Syntax:

```
interface name {  
    Declaration of attributes and operations (as well as types and exceptions)  
};
```

- Interface also defines a scope of its own
- Names of attributes and operations must be unique
 - ◆ **No overloading!**

14 Interfaces – Attributes

- Public object variables
 - ◆ Write access can be prohibited (read-only attributes)
 - ◆ Not an instance variable

- Syntax:

```
attribute type name;           // read & write
readonly attribute type name; // read-only
```

- Example:

```
interface Account {
    readonly attribute float balance;
};
```

14 Interfaces – Operations

■ Methods of CORBA objects with:

- ◆ Method name
- ◆ Return value
- ◆ Parameters
- ◆ Exceptions
- ◆ (Invocation context)

■ Syntax:

```
return_type name( parameter_list ) raises( exception_list );
```

■ Only method name is significant

- ◆ No overloading by means of parameter types!

■ Method invocation with best-effort semantics (no return values and no exceptions allowed)

```
oneway void name( parameter_list );
```

14 Interfaces – Parameter Transmission

- For each parameter a copy direction is required:

- ◆ **in** from client to server only
- ◆ **out** from server to client only
- ◆ **inout** in both directions

- Syntax:

```
( copy_direction1 type1 name1, copy_direction2 type2 name2, ... )
```

- Example:

```
interface Account {  
    void makeDeposit( in float sum );  
    void makeWithdrawal( in float sum,  
                        out float newBalance );  
};
```

14 Interfaces – Inheritance

- Derivation of a new interface from existing ones
- Multiple inheritance possible
- Syntax:

```
interface name : inherited_interface1, inherited_interface2, ... {  
    Declaration of additional attributes and operations  
};
```

- Names of inherited attributes and operations must be unique
 - ◆ Exception: Identifiers that are inherited via different paths but originate from the same base interface

14 Interfaces – Inheritance (2)

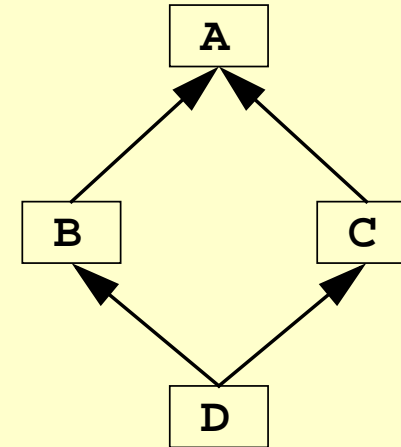
- Neither Overloading nor Overriding is allowed:

```
module Foo {  
    interface A {  
        void draw( in float num );  
    };  
  
    interface B {  
        void print( in float num);  
        void print( in string str); // Wrong: overloading  
    };  
  
    interface C: A, B {  
        void draw( in float num); // Wrong: Overriding  
    };  
};
```

14 Interfaces – Inheritance (3)

- Legal inheritance graph in CORBA:

```
module Foo {  
  interface A {  
    void draw( in float num );  
  };  
  
  interface B : A {  
  };  
  
  interface C : A {  
  };  
  
  interface D : B, C {  
  };  
};
```



15 Exceptions – User Exceptions

- User exceptions created in user code on the server side and propagated to the client

- Syntax:

```
exception name {  
    Declaration of data elements  
};
```

- Exceptions are special structs
 - ◆ Data elements only, no operations
 - ◆ **No inheritance for exceptions!**

- Example:

```
interface Account {  
    exception Overdraft { float howMuch; };  
    void makeWithdrawal( in float sum )  
        raises( Overdraft );  
};
```

15 Exceptions – System Exceptions

- System exceptions created by the ORB when invocation fails internally

```
module CORBA {
  enum completion_status { COMPLETED_YES, COMPLETED_NO,
                           COMPLETED_MAYBE};

  exception UNKNOWN {
    unsigned long      minor;
    completion_status completed;
  };
  exception BAD_PARAM {
    unsigned long      minor;
    completion_status completed;
  };
  exception NO_MEMORY {
    unsigned long      minor;
    completion_status completed;
  };
  exception COMM_FAILURE {
    unsigned long      minor;
    completion_status completed;
  };
  ...           // Many exceptions more
};
```

16 Forward Declarations

- Problem: Circular dependencies in declarations
 - ◆ Interface **A** has operation `op_b()` that returns object of type **B**
 - ◆ Interface **B** has operation `op_a()` that returns object of type **A**
- Solution: Forward declaration
 - ◆ Declare an identifier for a type but not the whole type
- Example:

```
interface B;           // Forward declaration
interface A {
    B op_b();
};
interface B {
    A op_a();
};
```

17 Value types

- Provide semantics that bridge between structs and interfaces
 - ◆ Support description of complex state (i.e., arbitrary graphs, with recursion and cycles)
 - ◆ Instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call)
 - ◆ Support both public and private (to the implementation) data members
- Value types support single **inheritance** (of valuetype) and can support an interface
- Example:

```
valuetype Person {
    public string name;    // A public state
    private long id;      // A private state

    void print();         // An operation
};
```

18 IDL Summary

- Description of data and interfaces of CORBA objects
- C++-like syntax
- Primitive types (`short`, `long`, `boolean`, `char`, ...)
- Constructed types (`struct`, `union`, `enum`)
- Arrays
- Template types (`sequence`, `string`, `fixed`)
- Object `interface` with attributes and operations
- Error reporting via exceptions
- Objects-by-value through `valuetypes`

19 Design Issues

- Problem: High-volume data objects
- Solution 1: Interface with attributes or access operations
 - + Clean OO abstractions
 - + All possibilities of distribution
 - High network traffic for data access
 - Scalability problems in some ORBs
- Solution 2: Struct with data members, local wrapping in objects
 - + Local data access
 - Broken OO abstractions
 - Multiple unsynchronised copies
- Solution 3: Value type
 - + Local data access and OO abstractions
 - Multiple unsynchronised copies