The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

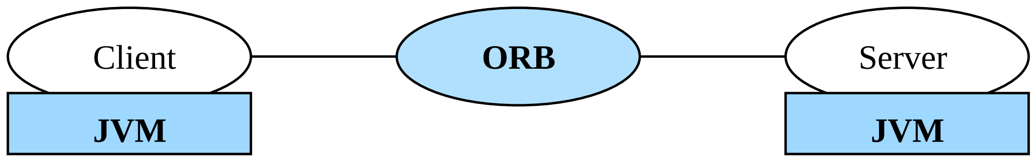
Room 1604

(Lecture 5: CORBA and CORBA IDL, 21st January 2002)

Common Object Request Broker Architecture (CORBA)

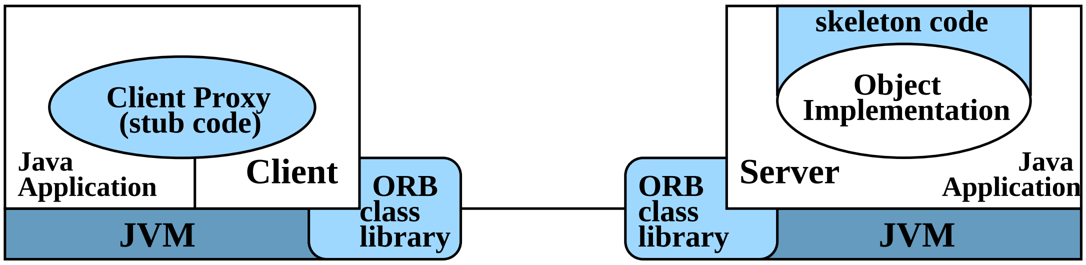
- CORBA is a distributed object *standard* developed by the OMG (Object Management Group).
- It is a collection of specifications for components and protocols.
- It aims to provide both **platform-** and **language-**independence.
- The system is **location transparent**—the locus of computation will move around the system transparently to the user and the application programmer does not need to know the location (or implementation details) of remote objects.
- CORBA 3.0 provides some support for object migration which means that objects may change location between invocations.

ORB: abstract view



- Central to CORBA is the **Object Request Broker (ORB)**.
- Sometimes called the **Software Bus**, it provides the means for distributed objects to communicate as well as infrastructure services such as a Naming Service or Security Services.
- It incorporates the communication modules, remote reference module and the dispatcher described generically in the previous lecture.
- Hides low-level details of platform-specific networking interfaces.
- May communicate with another ORB, using a General Inter-ORB Protocol(GIOP) such as Internet Inter-ORB Protocol (IIOP).

ORB: concrete view



- CORBA provides an **Interface Definition Language (IDL)** to specify object interfaces.
- A programming language specific IDL compiler then creates stub and skeleton code to facilitate the remote invocation (cf. RMI).
- JDK comes with the `idlj` compiler.
- An implementation of a CORBA object is called a **servant**.

CORBA Development Process

- Write IDL describing the interfaces to the objects that will be used or implemented.
- Compile the IDL file to produce the stub and skeleton code.
- Identify the IDL compiler-generated interfaces and classes that we need to use or specialise in order to invoke or implement operations.
- Write code to initialise the ORB and inform it of any CORBA objects we have created.
- Compile all generated and application code with a Java compiler.
- Run the distributed application.

Interface Definition Language (IDL)

- OMG IDL is a declarative language for defining interfaces to CORBA objects;
- It is intended to provide a language-independent way in which implementers and users can be assured of type-safe invocation of operations;
- No programming statements are included as it is only intended for defining interface signatures;
- IDL syntax is drawn from C++;
- ORB-specific IDL compilers generate stub and/or skeleton code that handles the marshalling and unmarshalling of arguments and results. Therefore all parameters of methods must be marked to indicate whether they are input, output or both.

IDL constructs

Constants — to assist with type declarations

Data type declarations — to use for parameter typing

Attributes — which get and set a value of a particular type

Operations — which take parameters and return values

Interfaces — which group data type, attribute and operation declarations

Modules — for name space separation.

N.B.: Identifiers are case sensitive but cannot co-exist with other identifiers that differ only in case.

Basic Types

`[unsigned] short`	signed[unsigned] 16-bit 2's complement integer
`[unsigned] long`	signed[unsigned] 32-bit 2's complement integer
`float`	16-bit IEEE floating point number
`double`	32-bit IEEE floating point number
`char`	ISO Latin-1 character
`boolean`	Boolean type (`TRUE/FALSE`)
`string`	variable length string of characters
`octet`	8-bit uninterpreted type
`enum`	enumerated type with named integer values
`any`	can represent values from any possible IDL type, basic or constructed, object or non-object

Sequences and Arrays

- A sequence is an ordered collection of items that can grow at run-time—this growth may be bounded or unbounded.
- Two run-time characteristics: maximum and current length.
- The advantage of sequences is that only the current number of elements is transmitted to a remote object when a sequence argument is passed.
- Sequence declarations must be given a `typedef` alias.
- Arrays are usually declared within a `typedef`, as they must be named before they are used as operation parameter or return types.
- Arrays at run-time will have a fixed length.
- The entire array will be marshalled and transmitted in a request if used as a parameter or a return type.
- Other C and C++ user-defined types (structures and discriminated unions) are also included.

Example:

```
module StructuredTypes {

   typedef sequence<float> TempSeq;
   typedef sequence<float,12> AnnualTempSeq;

   typedef short Scores[10];

   const short dim=200;
   typedef long My_matrix[dim][dim];
   .
   .
   .
};
```

User-defined Types

In addition to arrays and sequences, IDL offers a number of other user-defined types, i.e.

- `struct`,
- `union` and
- `enum`.

For example:

```
struct TestStruct{
   short a_short;
   long  a_long;
};
```

Modules and Interfaces

Modules

- The module is used as a naming scope which can be used to avoid name clashes when using several IDL declarations together.
- A module can contain any well-formed IDL, including nested modules.

Interfaces

- An interface can contain **constants**, **data type declarations**, **attributes** and **operations**.
- Interfaces also open a new naming scope.
- An interface name in the same scope can be used as a type name.
- An interface in another name scope can be referred to by giving a scoped name (in C++ `::` style).

Inheritance

The set of operations offered by an interface can be extended by declaring a new interface which inherits from the existing one: (`derived : base`)

```
module InheritanceExample {

   interface A {
      typedef unsigned short ushort;
      ushort op1();
   };
   interface B : A {
      boolean op2()(in ushort num);
   };
};
```

All interfaces implicitly inherit from `CORBA::Object`.

Operations

- Operation declarations are similar to C++ function prototypes.
- They consist of an operation name, a return type (or `void`) and a parameter list.
- They may additionally have a `raises` clause specifying which user-defined exceptions the operation may raise.
- They may have a `context` clause giving a list of names of string properties from the caller's environment that need to be supplied to the operation implementation.
- The list of parameters is surrounded by parentheses and separated by commas.
- Each parameter has a **directional indicator**: `in`, `out` or `inout`.
- The keyword `oneway` indicates that operation invocation will use *maybe semantics*: the caller gets an immediate return with no indication of success or results.

Exceptions

- A set of standard exceptions, known as **system exceptions**, is defined in the CORBA module.
- User-defined exceptions may be specified within an interface.

Example:

```
exception SomethingWrong {

    string reason;
    long   id;

};
```

Attributes

- An attribute is logically equivalent to a pair of accessor functions– `get` and `set`.
- Simpler to declare than operations.
- Consist of the keyword `attribute` followed by the type of the attribute(s) and the attribute name list.
- `readonly` attributes generate only the get function.
- No `raises` clause can be included so only standard exceptions may be raised by the accessor operations.

Example:

```
attribute string quote_of_the_day;

readonly attribute string corporate-motto;
```

Scoping example:

```
module outer {

   module inner {                  // nested module
      interface inside { };        // empty interface
   };

   interface outside {             // can refer to inner as local
      inner::inside get_inside();
   };

};
```

`get_inside()` returns an object reference of type `::outer::inner::inside`.

Forward Declarations

Interfaces may be mutually referential but forward declarations are necessary to avoid compilation errors.

```
module example {

   interface A;        // forward declaration

   interface B {       // B can use forward-declared
      A get_an_A();    // interfaces as type names
   };

   interface A {
      B get_a_B();
   };
};
```

Stock.idl

```
module StockObjects {
  struct Quote {
    string symbol;
    long   at_time;
    double price;
    long   volume;
  };
  exception Unknown {};
  interface Stock {
    Quote get_quote() raises (Unknown);    // returns quote
    void set_quote(in Quote stock_quote);  // sets current quote
    readonly attribute string description; // provides description
  };
  interface StockFactory {
    Stock create_stock(in string symbol, in string description);
  };
};
```

Compiling it...

with the command,

```
> idlj -fall Stock.idl
```

the following files will be generated in a StockObjects directory:

```
Quote.java                        StockHolder.java
QuoteHelper.java                  StockOperations.java
QuoteHolder.java                  Unknown.java
Stock.java                        UnknownHelper.java
StockFactory.java                 UnknownHolder.java
StockFactoryHelper.java           _StockFactoryImplBase.java
StockFactoryHolder.java           _StockFactoryStub.java
StockFactoryOperations.java       _StockImplBase.java
StockHelper.java                  _StockStub.java
```

Basic Data Type Mappings

The mapping for basic data types is straightforward due to the similarity between the IDL basic types and the Java primitive types.

IDL Type	Java
boolean	`boolean`
char	`char`
wchar	`char`
octet	`byte`
short/unsigned short	`short`
long/unsigned long	`int`
long long/unsigned long long	`long`
float	`float`
double	`double`

There is a potential problem mapping from IDL's unsigned integer types to Java's signed integer types.

Holder Classes for Basic Data Types

Holder classes for the basic IDL data types are defined in the package `org.omg.CORBA`:

```
package org.omg.CORBA;

final public class IntHolder {
      public int value;
      public IntHolder() { }
      public IntHolder( int initial ) {
            value = initial
      }
}
```

Parameter Management Responsibilities

	Invoking Client	*Object implementation*
Result	Declares variable of return type and assigns result.	Declares variable, creates and initialises instance, returns instance.
`in`	Declares variable, creates and initialises instance, passes to invocation.	Declares parameter, uses value passed.
`inout`	Declares variable, creates Holder object, initialises with a value, passes to invocation.	Declares Holder parameter, modifies `value` field of Holder parameter.
`out`	Declares variable, creates Holder object passes to invocation.	Declares Holder parameter, initialises `value` field of Holder parameter.

`idlj` compiler output (***Example*** `.idl`)

_*Example* `ImplBase.java` Abstract class forming the server skeleton. It provides basic CORBA functionality. The server implementer must write a "*Example*`Servant`" class, derived from this class.

_*Example* `Stub.java` The client stub. The client implementer must link this with the client class.

Example `Operations.java` This contains the Java operations interface which corresponds to the IDL interface *Example*.

Example `.java` This contains the Java signature interface. It extends the operations interface and `org.omg.CORBA.Object` thus providing standard CORBA object functionality.

Example `Helper.java` This `final` class provides auxiliary functionality; in particular the `narrow()` method providing safe downcasting of CORBA object references to their most derived type.

Example `Holder.java` This `final` class holds a public instance member of type *Example*. Instances of this are needed when an argument of type *Example* is an `out` or `inout` argument of an IDL operation.

Example Java interface: `StockOperations.java`

```
package StockObjects;
/**
* StockObjects/StockOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Stock.idl
* Friday, January 18, 2002 12:05:22 PM GMT
*/
public interface StockOperations
{
  StockObjects.Quote get_quote () throws StockObjects.Unknown;

  // returns quote
  void set_quote (StockObjects.Quote stock_quote);

  // sets current quote
  String description ();
} // interface StockOperations
```

Example Java interface: `Stock.java`

```
package StockObjects;

/**
* StockObjects/Stock.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Stock.idl
* Friday, January 18, 2002 12:05:22 PM GMT
*/

public interface Stock extends StockOperations,
           org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
} // interface Stock
```