Introduction to CORBA IDL

Based on Ch 4 of Orbix Programmer's Guide

January 22, 1999

IDL Modules and Scoping

An IDL module defines a naming scope for a set of IDL definitions. Modules allow you to group interface and other IDL type definitions in logical name spaces. When writing IDL definitions, always use modules to avoid possible name clashes.

The following example illustrates the use of modules in IDL:

```
module Finance {
  interface Account {
  ...
  };
};
```

- The interface `Account` is scoped within the module `Finance`.
- IDL definitions are available directly within the scope in which you define them.
- In other naming scopes, you must use the scoping operator (`::`) to access these definitions.

  For example, the fully scoped name of interface `Account` is `Finance::Account`
- IDL modules can be *reopened*.

  For example, a module declaration can appear several times in a single IDL specification if each declaration contains different data types.

  In most IDL specifications, this feature of modules is not required.

Defining IDL Interfaces

An IDL interface describes the functions that an object supports in a distributed application. Interface definitions provide all of the information that clients need to access the object across a network.

Consider the example of an interface that describes objects which implement bank accounts in a distributed application.

```
module Finance {
  interface Account {
    // The account owner and balance.
       readonly attribute string owner;
       readonly attribute float balance;
    // Operations available on the account.
```

```
        void makeDeposit(in float amount,
                         out float newBalance);
        void makeWithdrawal(in float amount,
        out float newBalance);
    };
  };
```

The definition of interface `Account` includes both attributes and operations. These are the main elements of any IDL interface definition.

Attributes in IDL Interface Definitions

- Conceptually, attributes correspond to variables that an object implements.
- Attributes indicate that these variables are available in an object and that clients can read or write their values.
- In general, attributes map to a pair of functions in the programming language used to implement the object. These functions allow client applications to read or write the attribute values.
- However, if an attribute is preceded by the keyword `readonly`, then clients can only read the attribute value.

  For example, the `Account` interface defines the attributes `balance` and `owner`.

  These attributes represent information about the account which the object implementation can set, but which client applications can only read.

Operations in IDL Interface Definitions

- IDL operations define the format of functions, methods, or operations that clients use to access the functionality of an object.
- An IDL operation can take parameters and return a value, using any of the available IDL data types.
- For example, the `Account` interface defines the operations `makeDeposit` and `makeWithdrawal` as follows:

```
module Finance {
  interface Account {
    // Operations available on the account.
    void makeDeposit(in float amount,
                     out float newBalance);
    void makeWithdrawal(in float amount,
    out float newBalance);
    ...
  };
};
```

- Each parameter definition must specify the direction in which the parameter value is passed.

  The possible parameter passing modes are as follows:

`in`	The parameter is passed from the caller of the operation to the object.
`out`	The parameter is passed from the object to the caller.
`inout`	The parameter is passed in both directions.

Raising Exceptions in IDL Operations

IDL operations can raise exceptions to indicate the occurrence of an error.

- CORBA defines two types of exceptions:
  - *System exceptions* are a set of standard exceptions defined by CORBA.
  - *Userdefined exceptions* are exceptions that you define in your IDL specification.
- Implicitly, all IDL operations can raise any of the CORBA system exceptions.
- No reference to system exceptions appears in an IDL specification.
- To specify that an operation can raise a user-defined exception, first define the exception structure and then add an IDL `raises` clause to the operation definition.

  For example, the operation `makeWithdrawal` in interface `Account` could raise an exception to indicate that the withdrawal has failed, as follows:

```
module Finance {
  interface Account {
  exception WithdrawalFailure {
    string reason;
  };
  void makeWithdrawal(in float amount,
                      out float newBalance)
  raises(WithdrawalFailure);
  ...
  };
};
```

- An IDL exception is a data structure that contains member fields.

  In the above example, the exception `WithdrawalFailure` includes a single member of type `string`.
- The `raises` clause follows the definition of operation `makeWithdrawal` to indicate that this operation can raise exception `WithdrawalFailure`.
- If an operation can raise more then one type of user-defined exception, then include each exception identifier in the `raises` clause and separate the identifiers using commas.

Invocation Semantics for IDL Operations

- By default, IDL operations calls are *synchronous*; that is, a client calls an operation and blocks until the object has processed the operation call and returned a value.

- The IDL keyword `oneway` allows you to modify these invocation semantics.
- If you precede an operation definition with the keyword `oneway`, a client that calls the operation will not block while the object processes the call.[1]

  For example, you could add a oneway operation to interface `Account` that sends a notice to an `Account` object, as follows:

```
module Finance {
  interface Account {
    oneway void notice(in string text);
    ...
  };
};
```

- Orbix does not guarantee that a oneway operation call will succeed; so if a oneway operation fails, a client may never know.
- There is only one circumstance in which Orbix indicates failure of a oneway operation. If a oneway operation call fails before Orbix transmits the call from the client address space, then Orbix raises a system exception.
- A oneway operation cannot have any `out` or `inout` parameters and cannot return a value.
- Nor can a oneway operation have an associated `raises` clause.

Inheritance of IDL Interfaces

IDL supports inheritance of interfaces. An IDL interface can inherit all the elements of one or more other interfaces.

For example, the following IDL definition illustrates two interfaces, called `CheckingAccount` and `SavingsAccount`, that inherit from interface `Account`:

```
module Finance {
  interface Account {
    ...
  };
  interface CheckingAccount : Account {
    readonly attribute overdraftLimit;
    boolean orderChequeBook ();
  };
  interface SavingsAccount : Account {
    float calculateInterest ();
  };
};
```

- Interfaces `CheckingAccount` and `SavingsAccount` implicitly include all elements of interface `Account`.
- An object that implements `CheckingAccount` can accept invocations on any of the attributes and operations of this interface, and on any of the elements of interface `Account`.

[1] Actually, the CORBA standard doesn't guarantee that it won't block.

- However, a `CheckingAccount` object may provide different implementations of the elements of interface Account to an object that implements `Account` only.
- The following IDL definition shows how to define an interface that inherits both `CheckingAccount` and `SavingsAccount`:

```
module Finance {
  interface Account {
    ...
  };
  interface CheckingAccount : Account {
    ...
  };
  interface SavingsAccount : Account {
    ...
  };
  interface PremiumAccount : CheckingAccount, SavingsAccount {
  };
};
```

  Interface `PremiumAccount` is an example of *multiple inheritance* in IDL.

- An interface *can* inherit from two interfaces that include a *constant*, *type*, or *exception* definition of the same name. But then you must fully scope that name when using that constant, type, or exception.
- An interface *cannot* inherit from two interfaces that include *operations* or *attributes* that have the same name.

The Object Interface Type

- IDL includes the pre-defined interface `Object`, which all user-defined interfaces inherit implicitly.
- The operations defined in this interface are described in the Orbix Reference Guide.
- While interface `Object` is never defined explicitly in your IDL specification, the operations of this interface are available through all your interface types.
- In addition, you can use `Object` as an attribute or operation parameter type to indicate that the attribute or operation accepts any interface type:

```
interface ObjectLocator {
  void getAnyObject(out Object obj);
};
```

- It is not legal IDL syntax to inherit interface `Object` explicitly.

Forward Declaration of IDL Interfaces

- In an IDL definition, you must declare an IDL interface before you reference it.
- A `forward` declaration declares the name of an interface without defining it.
- This feature of IDL allows you to define interfaces that mutually reference each other.

  For example, IDL interface `Account` could include an attribute of IDL interface type `Bank`, to indicate that `Account` stores a reference to a `Bank` object.

  If the definition of interface `Bank` follows the definition of interface `Account`, you would forward declare `Bank` as follows:

```
module Finance {
  // Forward declaration of Bank.
  interface Bank;
  interface Account {
    readonly attribute Bank branch;
    ...
  };
  // Full definition of Bank.
  interface Bank {
    ...
  };
};
```

- The syntax for a `forward` declaration is the keyword `interface` followed by the interface identifier.

Overview of the IDL Data Types

- In addition to IDL module, interface, and exception types, there three general categories of data type in IDL:
  - Basic types
  - Complex types
  - Pseudo object types
- We examine each category of IDL types in turn and also describe how you can define new data type name constants in IDL.

IDL Basic Types

The basic types supported in IDL are

IDL Type	Range of Values
short	$2^{15} \ldots\ 2^{15}-1$ (16bit)
unsigned short	$0 \ldots\ 2^{16}-1$ (16bit)
long	$-2^{31} \ldots\ 2^{31}-1$ (32bit)
unsigned long	$0 \ldots\ 2^{32}-1$ (32bit)
float	IEEE single-precision floating point numbers.
double	IEEE double-precision floating point numbers.
char	An 8-bit value.
boolean	TRUE or FALSE.
octet	An 8-bit value that is guaranteed not to undergo any conversion during transmission.
any	The any type allows the specification of values that can express an arbitrary IDL type.

The `any` data type allows you to specify that an attribute value, an operation parameter, or an operation return value can contain an arbitrary type of value to be determined at runtime.

IDL Complex Types

The IDL *complex types* are data types enum, struct, union, string, sequence, and array.

Enum An *enumerated* type allows you to assign identifiers to the members of a set of values, for example:

```
module Finance {
  enum Currency {pound, dollar, yen, franc};
  interface Account {
    readonly attribute float balance;
    readonly attribute Currency balanceCurrency;
    ...
  };
};
```

In this example, attribute `balanceCurrency` in interface `Account` can take any one of the values `pound`, `dollar`, `yen`, or `franc`.

Struct A *struct* data type allows you to package a set of named members of various types, for example:

```
  module Finance {
    struct CustomerDetails {
       string name;
       short age;
    };

    interface Bank {
       CustomerDetails getCustomerDetails(in string name);
       ...
    };
};
```

In this example, the struct `CustomerDetails` has two members. The operation `getCustomerDetails` returns a struct of type `CustomerDetails` that includes values for the customer name and age.

Union A *union* data type allows you to define a structure that can contain only one of several alternative members at any given time.

A union saves space in memory, as the amount of storage required for a union is the amount necessary to store its largest member.

All IDL unions are discriminated. A discriminated union associates a label value with each member. The value of the label indicates which member of the union currently stores a value.

For example, consider the following IDL union definition:

```
struct DateStructure {
   short Day;
   short Month;
   short Year;
};

union Date switch (short) {
   case 1: string stringFormat;;
   case 2: long digitalFormat;
   default: DateStructure structFormat;
};
```

The union type `Date` is discriminated by a short value.

For example, if this short value is 1, then the union member `stringFormat` stores a date value as an IDL `string`.

The default label associated with the member `structFormat` indicates that if the short value is not 1 or 2, then the `structFormat` member stores a date value as an IDL struct.

Note that the type specified in parentheses after the `switch` keyword must be an integer, char, boolean or enum type and the value of each case label must be compatible with this type.

String An IDL `string` represents a character string, where each character can take any value of the char basic type.

If the maximum length of an IDL string is specified in the string declaration, then the string is bounded. Otherwise the string is unbounded.

The following example shows how to declare bounded and unbounded strings:

```
module Finance {
  interface Bank {
    // A bounded string with maximum length 10.
    attribute string sortCode<10>;
    // An unbounded string.
    attribute string address;
    ...
  };
};
```

Sequence

In IDL, you can declare a *sequence* of any IDL data type. An IDL sequence is similar to a one-dimensional array of elements, but it does not have a fixed length. If the sequence has a fixed maximum length, then the sequence is bounded. Otherwise, the sequence is unbounded. For example, the following code shows how to declare bounded and unbounded sequences as members of an IDL struct:

```
module Finance {
  interface Account {
  ...
  };

  struct LimitedAccounts {
    string bankSortCode<10>;
    // Maximum length of sequence is 50.
    sequence<Account, 50> accounts;
  };

  struct UnlimitedAccounts {
    string bankSortCode<10>;
    // No maximum length of sequence.

  sequence<Account> accounts;
  };
};
```

A sequence must be named by an IDL `typedef` declaration (to be described) before it can be used as the type of an IDL attribute or operation parameter. The following code illustrates this:

```
module Finance {
  typedef sequence<string> CustomerSeq;
  interface Bank {
    void getCustomerList(out CustomerSeq names);
    ...
  };
};
```

Arrays

In IDL, you can declare an array of any IDL data type. IDL arrays can be multi-dimensional and always have a fixed size. For example, you can define an IDL struct with an array member as follows:

```
module Finance {
  interface Account {
  ...
  };

  struct CustomerAccountInfo {
    string name;
```

```
    Account accounts[3];
  };

  interface Bank {
    getCustomerAccountInfo (in string name,
    out CustomerAccountInfo accounts);
   ...
  };
};
```

In this example, struct `CustomerAccountInfo` provides access to an array of `Account` objects for a bank customer, where each customer can have a maximum of three accounts.

An array must be named by an IDL `typedef` declaration before it can be used as the type of an IDL attribute or operation parameter. The IDL typedef declaration allows you define an alias for a data type, as described in "Defining Data Type Names and Constants" on page 61. The following code illustrates this:

```
module Finance {
  interface Account {
    ...
  };

  typedef Account AccountArray[100];

  interface Bank {
    readonly attribute AccountArray accounts;
    ...
  };
};
```

Note that an array is a less flexible data type than an IDL sequence, because an array always has a fixed length. An IDL sequence always has a variable length, although it may have an associated maximum length value.

IDL Pseudo Object Types

CORBA defines a set of pseudo object types that ORB implementations use when mapping IDL to some programming languages. These object types have interfaces defined in IDL but do not have to follow the normal IDL mapping for interfaces and are not generally available in your IDL specifications.

You can use only the following pseudo object types as attribute or operation parameter types in an IDL specification:

```
NamedValue
Principal
TypeCode
```

To use any of these three types in an IDL specification, include the file `orb.idl` in the IDL file as follows:

```
#include <orb.idl>
...
```

This statement indicates to the IDL compiler that types `NamedValue`, `Principal`, and `TypeCode` may be used. The file `orb.idl` should not actually exist in your system. Do not name any of your IDL files orb.idl.

Defining Data Type Names and Constants

IDL allows you to define new data type names and constants. This section describes how to use each of these features of IDL.

Data Type Names

The typedef keyword allows you define a meaningful or simpler name for an IDL type.

```
module Finance {
  interface Account {
  ...
  };
  typedef Account StandardAccount;
};
```

The identifier `StandardAccount` can act as an alias for type `Account` in subsequent IDL definitions. Note that CORBA does not specify whether the identifiers `Account` and `StandardAccount` represent distinct IDL data types in this example.

Constants

IDL allows you to specify *constant* data values using one of several basic data types. To declare a constant, use the IDL keyword `const`.

```
module Finance {
  interface Bank {
    const long MaxAccounts = 10000;
    const float Factor = (10.0  6.5) * 3.91;
    ...
  };
};
```

The value of an IDL constant cannot change. You can define a constant at any level of scope in your IDL specification.