Chap 6. CORBA-based Architecture

Part 6.2 CORBA-IDL

1. Rationale
2. IDL Overview
3. Basic IDL Constructs
4. IDL to Java Language Mapping
1. Rationale

-The main rationale for IDL is to solve the heterogeneity inherent in distributed architectures.

-Heterogeneity among programming languages may arise because, for instance, their constructs and features are different, or their machine code representations differ.

-CORBA standard supports a common object model and an interface definition language (IDL) that represents a key for resolving programming language heterogeneity.

▷ This is based on defining bindings from available programming languages to the IDL.

▷ Defining a programming language binding consists of specifying how IDL constructs can be used in a client or server implementation, and vice-versa, regardless of which language they are written in or which platform they are running on.
A language binding defines a one-to-one direct mapping between its constructs and the IDL constructs:
- Object types are mapped to client and server stubs.
- Server object references are encapsulated in client stub.
- Operations are mapped to procedures, operations or methods in the programming language.

Vendors implement bindings by providing APIs that can be used both by client and servers, and also compilers that generate client and server’s stubs.
2. IDL Overview

Example IDL Code

The example deals with a store with point-of-sale (POS) terminals. The interface the POS terminal object uses to communicate with its barcode-reader object, Keypad object, and receipt-printer object can be expressed in IDL:

```
//POS IDL Example
module POS {
    typedef string Barcode;

    interface InputMedia {
        typedef string OperatorCmd;
        void barcode_input(in media Barcode item) {
            void keypad_input(in OperatorCmd cmd);
        }
    }

    interface OutputMedia {
        boolean output_text(in string string_to_print);
    }

    interface POSTerminal {
        void end_of_sale();
        void print_POS_sales_summary();
    }
}
```

- This file declares the interfaces for all the objects in our POS terminal, in a programming-language independent way.

- In our “store”, other objects will invoke these objects on the POS Terminal computer through these interfaces.
**IDL Basics**

-Although IDL interfaces are programming language-independent, the IDL language itself has the appearance (but not the semantics) of ANSI C++.

÷Notable resemblances include:
  - C++ -like preprocessing
  - Lexical rules (with new keywords for distribution concepts)
  - Grammar, which is a subset of C++ and incorporates syntax for constant, type, and operation declarations; but lacks algorithmic structures and variables.

÷Differences include:
  - Function return type is required (but may be void)
  - Each formal parameter in an operation declaration must have a name
  - and others
Generating IDL
3. Basic IDL

-Basic IDL includes constructs such as module, interface, simple and complex data types, and exception.

**Modules**

- Create separate name spaces for IDL definitions by defining scopes
- Used to define the enclosing scope of a group of IDL interfaces.
- Can contain one or more interfaces and can nest other module constructs
- Modules do not inherit from other modules, but can be nested.
- Only IDL interfaces are capable of inheriting specifications

-Example:

```idl
module Assembly {
    typedef string Widget;
};
```
**Interfaces**

- Specify a software boundary between a service implementation and its clients.
- IDL interfaces can inherit from other interfaces
- IDL interfaces may involve *attributes*, *operations* and *types* definitions

**Example:**

```csharp
interface Account {
    //Account definitions
}
```

```csharp
interface Savings: Account {
    //Inherits all Account definitions
    //then adds Savings definitions
}
```
Attributes
• Attributes define general characteristics for an interface
• If an attribute or operation is private, it should not appear in a public IDL definition
• By default, all IDL definitions (known by the ORB) are public
• Attributes may be read-only or read-write

• Example

```inteface Account {

    attribute string balance;
    readonly attribute long ssn;

};
```

• For read-write attributes, the compiler generates a set and a get functions for each attribute.
• For read-only attributes, only a get function is generated.
**IDL Forward**

- Statement used to declare an interface before its complete definition appears in the IDL file.
- Can also be used to create recursive (or self-referential) definitions.

- Example:

```idl
interface Employee;    //forward declaration

interface Company {
    attribute Employee supervisor;
    attribute Employee secretary;
};

interface Employee {
    attribute string department;
    attribute string name;
}
```
**Data Types**
- IDL enables strong type checking of operation signatures, and includes renaming of intrinsic types in IDL, as well as the creation of user-defined types: enumeration, structures, arrays, sequences, unions

**Basic Types:** IDL supports most of the basic types supported by modern programming languages (e.g., C, C++, Java).

<table>
<thead>
<tr>
<th>IDL Basic Types</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>short, unsigned short</td>
<td>Integer-16 bits</td>
</tr>
<tr>
<td>long, unsigned long</td>
<td>Integer-32 bits</td>
</tr>
<tr>
<td>long long, unsigned long long</td>
<td>Integer-64 bits</td>
</tr>
<tr>
<td>float</td>
<td>IEEE single-precision floating point</td>
</tr>
<tr>
<td>double</td>
<td>IEEE double-precision floating point</td>
</tr>
<tr>
<td>long double</td>
<td>IEEE double-extended floating point</td>
</tr>
<tr>
<td>char</td>
<td>8-bit quantity</td>
</tr>
<tr>
<td>wchar</td>
<td>Encode wide characters from any character set (size undefined)</td>
</tr>
<tr>
<td>boolean</td>
<td>True or false</td>
</tr>
<tr>
<td>Octet</td>
<td>Byte- is guaranteed not to undergo any conversion when it passes over the wire.</td>
</tr>
<tr>
<td>string, wstring</td>
<td>Are basically array of chars and wchar</td>
</tr>
</tbody>
</table>
**IDL Constants**

- IDL constants are expressed using the keyword `const` followed by the type, name and value of the constant.

  • Example:

    ```
    const unsigned long km=2.2;
    const char cr='/';
    const boolean tautology=TRUE;
    const float pi=3.14;
    const double av=6.02e25;
    const string state="Virginia";
    ```

**Renamed Type**

- Construct for naming new IDL types from existing ones.

  ```
  typedef unsigned long PhoneNumber;
  ```

  **Example:** `typedef string LastName;

  ```
  const LastName my_lastname ="Smith";
  ```
**Enumeration type**  
- Used to represent an enumerated list.

    enum ChargeCard {MasterCard, Visa, Diners};

**Structure type**  
- Container class that may be used to pass a collection of data as a single object.

    struct GuestRecord {
      GuestName name;
      Address address;
      PhoneNumber number;
    };
**Sequence type**
- Single dimension arrays that may be bounded or unbounded
- Are essentially variable-length arrays
- A bounded sequence defines its maximum size in its declaration

```cpp
typedef sequence <GuestRecord> record; //unbounded sequence
typedef sequence <GuestList,10> list; //bounded sequence
```

**Array type**
- Used to create a single-dimension, bounded array of IDL type.

```cpp
typedef EmployeeRecord Employees[100];
```
**Union type**

```c
enum PersonKind {A_GUEST, AN_EMPLOYEE, OTHER}

union Person switch (PersonKind) {
    case A_GUEST:
        GuestRecord guest_record;
    case AN_EMPLOYEE:
        EmployeeRecord employee_record;
    default:
        string description;
}
```

**Dynamic IDL type Any**

- Allow definition of loosely typed data values
- Useful for defining reusable interfaces
- Example:

```c
typedef any DynamicallyTypedValue;
struct RunTimeValue {
    string description;
    any run_time_value;
};```
**IDL Exceptions**

- Define the values passed by the interface in case something goes wrong.
- Extend the `org.omg.CORBA.UserException` class
- May contain data that are accessed as public members of the named class and may be passed in the construction of the exception.
- Exception values are declared similar to IDL structure type.

- Example:

```java
exception CardExpired {string expiration_data;};
exception CardReportedStolen {
    string reporting_instructions;
    unsigned long hotline_phone_number;
};
```

- There are two general kinds of exceptions: user-defined and CORBA defined, also called Standard Exceptions and which extend the `org.omg.CORBA.SystemException` class.
**Operations**
- Define the acceptable way to access an object
- The IDL type of the target object is the declared name of the Interface.
- All operation definitions are declared within specific IDL interfaces.
- By default, IDL operations are synchronous
- An asynchronous option is provided using the *oneway* keyword, which indicates that an operation will be executed at most once.
- Operations that are oneway can only have input parameters.

**Operation Signatures**
- Operation signatures include:
  - the operation attribute (*oneway* or *none*),
  - the operation type specification,
  - the operation identifier,
  - the parameters declarations,
  - an optional *raises* expression,
- The operation type specification is the return value: may be any IDL type or the keyword `void`.
- Arguments to operations declare the call semantics of the argument: `in`, `out`, or `inout`
  - An `in` parameter is called by value
  - The `out` parameters use call-by-reference semantics.
  - The `inout` parameter semantics is call-by-value/return-by-reference
- Operations can declare that they raise an exception using the construct `raises (ExceptionName)` in their signature.
  - Exceptions in the raises clauses must be declared before they can be used.

**Example:**

```c
interface AirlineReservation {
    typedef unsigned long ConfirmationNumber;
    exception BadConfirmationNumber {};
    oneway void cancel_reservation (in ConfirmationNumber number)
        raises (BadConfirmationNumber);
};
```
Comments and Pre-compiler Directives

Comments
• Two forms:
  - single line comments that begin with a // symbol
  - multiple lines comments enclosed by /* and */ symbols.
• Example:
  // This is a single line comment
  /* This is a multiple-line comment. */

Pre-Compiler Directives
• IDL provides pre-compiler directives, as do C and C++
• Example: the include statement allows IDL files to reference each other’s definitions.
• By convention, IDL files are named after the module they contain.
• Example:
  //Enable access to CORBA Naming Service
  #include <CosNaming.idl>
Example: A Course Registration System

• **OO Model:**

<table>
<thead>
<tr>
<th>Student</th>
<th>Registered for</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>personalInfo</td>
<td>major</td>
<td>subject</td>
</tr>
<tr>
<td>enroll()</td>
<td>graduate()</td>
<td>semester</td>
</tr>
</tbody>
</table>

• **Abstract IDL model**

```idl
module CourseRegistration {
    interface Student {
        attribute any personalInfo;
        attribute any major;
        void enroll();
        void graduate();
    };

    interface Course {
        attribute any subject;
        attribute any semester;
        void register();
        void cancel();
    };
}
```
**Refined IDL model:**

```idl
module CourseRegistration {
  // Forward Declarations
  interface Course {
    attribute string subject;
    enum SchoolSemesters {FALL, SPRING, SUMMER};
    attribute SchoolSemesters semester;
    void register(in Student student);
    void cancel();
  };

  module Student {
    struct StudentRecord {
      String name;
      String address;
      unsigned long studentNo;
    };
    attribute StudentRecord personalInfo;
    attribute string major;
    exception ClassFull {};
    void enroll(in Course course) raises (ClassFull);
    exception HasNotCompletedReqts {};
    void graduate() raises (HasNotCompletedReqts);
  };

  interface Course {
    attribute string subject;
    enum SchoolSemesters {FALL, SPRING, SUMMER};
    attribute SchoolSemesters semester;
    void register(in Student student);
    void cancel();
  };
}
```
4. IDL to Java Language Mapping

-CORBA programming language mappings “map” one-to-one correspondences from OMG IDL constructs to programming language constructs.
-These constructs tell client and object-implementation writers what to write to invoke an operation on a CORBA object.
-In a non-OO language, invocations typically map to function calls. In OO languages, mappings make CORBA invocations look like language-object invocations.

Programming Conventions

-Programming conventions for Java and IDL differs slightly:
  • IDL convention does not require capitalization for the names of modules, etc.
  • IDL convention uses underscores instead of mixed case for long names.
  • An IDL file is composed of several elements that together create a naming scope.
  •Identifiers in IDL are case insensitive and may be used only once in the naming scope.
  • IDL does not support the overloading and overriding of operations, although inheritance (single and multiple) is supported.
**IDL Module**

- Each module construct compiles to a Java package name

- Example:

  ```idl
  //IDL
  module BookStore {
      interface Account {
          ...
      }
  }
  ```

  //Java code generated by `idltojava` compiler would include:
  ```java
  package BookStore;
  ...
  ```
**IDL Types**

- CORBA types can either be standard IDL types or another IDL interface

<table>
<thead>
<tr>
<th>IDL</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>long, unsigned long</td>
<td>int</td>
</tr>
<tr>
<td>long long,</td>
<td>long</td>
</tr>
<tr>
<td>unsigned long long</td>
<td></td>
</tr>
<tr>
<td>short, unsigned short</td>
<td>short</td>
</tr>
<tr>
<td>char, wchar</td>
<td>char</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>octet</td>
<td>byte</td>
</tr>
<tr>
<td>string, wstring</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>any</td>
<td>class Org.omg.CORBA.Any.java</td>
</tr>
<tr>
<td>enum, struct, union</td>
<td>class</td>
</tr>
</tbody>
</table>
**IDL typedef**

- Does not directly map onto Java, so the IDL compiler will substitute and replace any instance of the `typedef` name for the actual type in the IDL before compiling it.

- Example:

```idl
typedef string CustomerName;
typedef sequence <long> CustomerOrderID;
```
**IDL sequence**

- An IDL sequence is mapped to a Java array; bounds checking is always done on bounded sequences.
- A Java Helper and Holder class is generated for each sequence.
- Example:
  ```idl
  typedef sequence <long,10> openOrders;
  ```

**IDL Arrays**

- Mapped to Java the same way as a bounded IDL sequence; bounds checking is always done.
- A Holder class is also generated for the array.
- Example:
  ```idl
  const long length=20;
  typedef string custName[length];
  ```
**IDL Any**

- Maps to the `org.omg.CORBA.Any` java class.
- Nonprimitive types are inserted into an `any` via the static methods provided in the corresponding helper class.

**Example:**
To insert the CORBA object reference for an instance `obj` of `Account` interface into an `any` requires the following code:

```
AccountHelper.insert(any, obj);
```

To extract from the `any`:

```
obj = AccountHelper.extract(any);
```
• For each primitive type $XXX$, there is a pair of methods $insert_{XXX}$ and $extract_{XXX}$ to update or return a typed value contained in the Any.

Example:

To create new Any and then initialize it from a short:
```
org.omg.CORBA.Any x = orb.create_any();
x.insert_short((short) 4);
```

To retrieve the value in the Any:
```
short y = x.extract_short();
```
**IDL enum**

- Maps to a Java final class with the same name.

- Example:

```idl
enum Card {visas, master, amex};
```

//maps to a Java final class with the same name: `Card.java`

- Contains one `value` method, two static data members per label, an integer conversion method, and a private constructor.
- Holder and Helper classes are also generated for an enum.
Generated Java Code for enum

final public class Card {

    //static data members
    final public static int _visas = 0;
    final public static Card visas = new Card(_visas);
    final public static int _master = 1;
    final public static Card master = new Card(_master);
    final public static int _amex = 2;
    final public static Card amex = new Card(_amex);

    //Private constructor
    private int _value;

    private Card(int value) {
        this._value = value;
    }

    //Value accessor methods
    public int value() {
        return _value;
    }
}
public static Card from_int(int $value) {
    case _visas: return visas;
    case _master: return master;
    case _amex: return amex;
    default: throw new org.omg.CORBA.BAD_PARAM("Enum out of range: [0.." + (5 - 1) + "]: " + $value);
}

//string conversion
public java.lang.String toString () {
    org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
    CardHelper.insert(any,this);
    Return any.toString();
}

Code Snippet for enum

Card creditCard = Card.visas;
System.out.println("Credit card type = " + creditCard);
**IDL struct**

- Maps to Java final class with public data members.

- Example:

```java
//an IDL struct

Struct Address {
    string street;
    long number;
};
```

//maps to a Java class that is final: `Address.java`

- has public member variables with the same name and type as the struct fields.
- contains two constructors: the default constructor, and one to initialize each field to a non-default value.
- Holder and Helper classes are also generated for each IDL struct.
Generated Java code for struct

```java
final public class Address {
   //Data members
   public java.lang.String street;
   public int number;

   //Constructors
   public Address() {}  
   public Address (java.lang.String street, int number) {
      this.street = street;
      this.number = number;
   }

   //string conversion
   public java.lang.String toString () {
      org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
      AddressHelper.insert(any,this);
      return any.toString();
   }
}

Code Snippet for struct

Address addr = new Address(“Hills”,9567);
System.out.println(“My street is: “ + addr.street);
```
**IDL union**

- Maps to Java final class.
  - Has a default constructor, an accessor method for the discriminator, and an accessor and a modifier methods for each branch.
  - Helper and Holder classes are also generated for a union.

**Example:**

```java
enum DataType {BYTE_TYPE, SHORT_TYPE, LONG_TYPE}

union DataValue switch(DataType) {
    case BYTE_TYPE: octet byteValue;
    case SHORT_TYPE: short shortValue;
    case LONG_TYPE: long longValue;
}
```
Generated Java Code for Union

final public class DataValue {
    //constructor
    public DataValue () {} 

    //discriminator
    private java.lang.Object _object;
    private short _disc;
    public DataValue discriminator () {
        return _disc;
    };

    //Accessor and modifier methods for the branches
    public byte byteValue() {...}
    public void byteValue (byte value) {...}
    public short shortValue() {...}
    public shortValue (short value) {...}
    public int longValue() {...}
    public shortValue (int value) {...}

    //string conversion
    public java.lang.String toString() {...}
}
Code Snippet for Union

// create a union
DataValue data = new DataValue();

// initialize its value
data.longValue(1234);

// display a failure when the wrong method is called
System.out.println("This works: data value is " + data.longValue());
System.out.println("This fails: data value is " + data.shortValue());
**IDL Interface**

- Maps to a Java interface class
- Can contain *attributes, operations, and exceptions*.

- The **idlj** compiler generates a certain number of files according to the option used.

Example:

```java
module bank {
    interface Account {
        void deposit();
    }
};
```
**idlj --fall bank.idl** command will generate the following files:

<table>
<thead>
<tr>
<th>Generated Files</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountOperations.java</td>
<td>all the operations provided by the IDL interface are defined in this file, which is shared by both the stubs and the skeletons.</td>
</tr>
<tr>
<td>Account.java</td>
<td>contains the java version of the IDL interface; used as signature type in method declarations.</td>
</tr>
<tr>
<td>_AccountStub.java</td>
<td>a java class that contains the <strong>stub</strong> code</td>
</tr>
<tr>
<td>_AccountImplBase</td>
<td>a java class that contains the <strong>skeleton</strong> code</td>
</tr>
<tr>
<td>AccountHolder.java</td>
<td>a <strong>Holder</strong> class that is used to contain a reference to the IDL interface object if the interface is passed as an argument.</td>
</tr>
<tr>
<td>AccountHelper.java</td>
<td>a <strong>Helper</strong> class that is used to downcast a CORBA object to this type</td>
</tr>
</tbody>
</table>
Attributes

• An attribute will generate accessor and modifier methods for the type declared: the compiler does not generate a variable, but just the methods to access the variable.

- Example:

    attribute float price; //IDL
    
    float price();          //generated Java methods
    void price(float arg);

• The attribute may be declared **readonly**, in which case only an accessor is declared.

- Example: **readonly** attribute BookList theOrder;
**IDL Operations**

- Compiled to Java methods
- Each operation must declare a return type and may have zero or more arguments.
- Arguments to operations declare the call semantics of the argument: **in**, **out**, or **inout**
  - An **in** parameter is called by value and is mapped directly to the corresponding Java type.
  - The **out** parameters use call-by-reference semantics. Since Java does not support call-by-reference, **out** parameters are mapped onto a **JavatypeHolder** class, which encapsulates a data variable containing the parameter, and the value of the class reference is passed.
  - The **inout** parameter semantics is call-by-value/return-by-reference, and is also mapped onto a Java Holder class.
Example:

// IDL

module Monitor { 
    interface Modes { 
        long process(in long inArg, 
        out long outArg, 
        inout long inoutArg); 
    };
};

// Generated Java
package Monitor;
public interface ModesOperations { 
    int process(int inArg, 
        org.omg.CORBA.IntHolder outArg, 
        org.omg.CORBA.IntHolder inoutArg); 
}

public interface Modes extends ModesOperations, 
    org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity { 
}
// Holder Class
final public class ModesHolder
    implements org.omg.CORBA.portable.Streamable {
    public Modes value;
    public ModesHolder() {}
    public ModesHolder(Modes initial) {...}
    public void _read(org.omg.CORBA.portable.InputStream is) {...}
    public void _write(org.omg.CORBA.portable.OutputStream os){...}
    public org.omg.CORBA.TypeCode _type() {...}
}

- Holder classes are used for out and inout IDL operation parameters:
  - Holder classes exist for all primitive java types (in the org.omg.CORBA package) and for all user-defined types except those that are defined by typedef.
  - Each Holder class has two constructors and a public member variable always named value. You can get/set the value of the Holder variable using the value member variable.
abstract public class ModesHelper {
    public static void insert(org.omg.CORBA.Any a, Modes t) {...}
    public static Modes extract(Any a) {...}
    public static org.omg.CORBA.TypeCode type() {...}
    public static String id() {...}
    public static Modes read(
        org.omg.CORBA.portable.InputStream is) {...}
    public static void write(
        org.omg.CORBA.portable.OutputStream os, Modes val) {...}
    public static Modes narrow(java.lang.Object obj){...}
}

-Contains a number of static methods that are used to manipulate the type.
÷The most important methods are the any insert and extract methods for that type,
    getting the RepositoryID, getting the typecode, and reading and writing the type
    from and to a stream.
÷Another important static method called narrow performs a safe down-casting
    (i.e., from org.omg.CORBA.Object to interface type).
// user Java code
// select a target object
Monitor.Modes target = ...;

// get the in actual value
int inArg = 57;

// prepare to receive out
IntHolder outHolder = new IntHolder();

// set up the in side of the inout
IntHolder inoutHolder = new IntHolder(131);

// make the invocation
int result=target.process(inArg, outHolder, inoutHolder);

// use the value of the outHolder
... outHolder.value ...

// use the value of the inoutHolder
... inoutHolder.value ...
**IDL Exception**

- Map into a Java final class that has instance variables for the fields of the exception.
  - Has a default constructor, and a constructor allowing to initialize each field of the exception.

- CORBA system exceptions are direct subclasses of `java.lang.RuntimeException`. User defined exceptions subclasses of `java.lang.Exception`.

- Helper and holder classes are also generated.
Example

// IDL
module bank {

    exception account_exception {long reason_code;};

};

// Generated Java
package bank;

final public class AccountException extends org.omg.CORBA.UserException {

    public int reason_code; // instance

    public AccountException() { // default constructor
        super(AccountException Helper.id());
    }

    public AccountException(int reason_code) { // constructor
        super(AccountException Helper.id());
        this.reason_code = reason_code;
    }

    public AccountException(String reason, int reason_code) { // full constructor
        // full constructor
        super(AccountException.id()+" "+reason);
        this.reason_code = reason_code;
    }

}