CORBA-IDL

1. Introduction
2. Basic IDL Constructs
3. IDL to Java Language Mapping

1. Introduction

Heterogeneity among Programming Languages

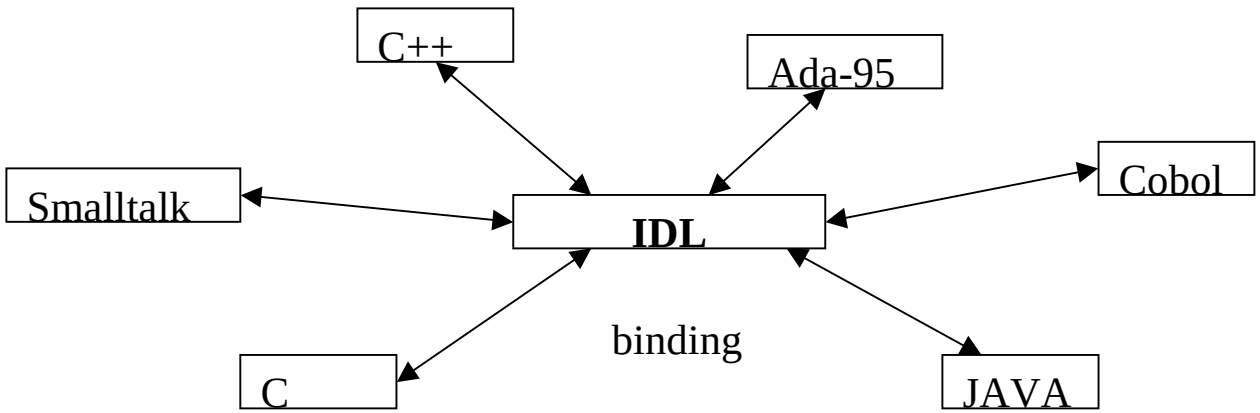
-May arise because, for instance, their constructs and features are different, or their machine code representations differ.

-Middleware systems support a common object model and an interface definition language (IDL) that represent a key for resolving programming language heterogeneity.

- That is achieved by 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.

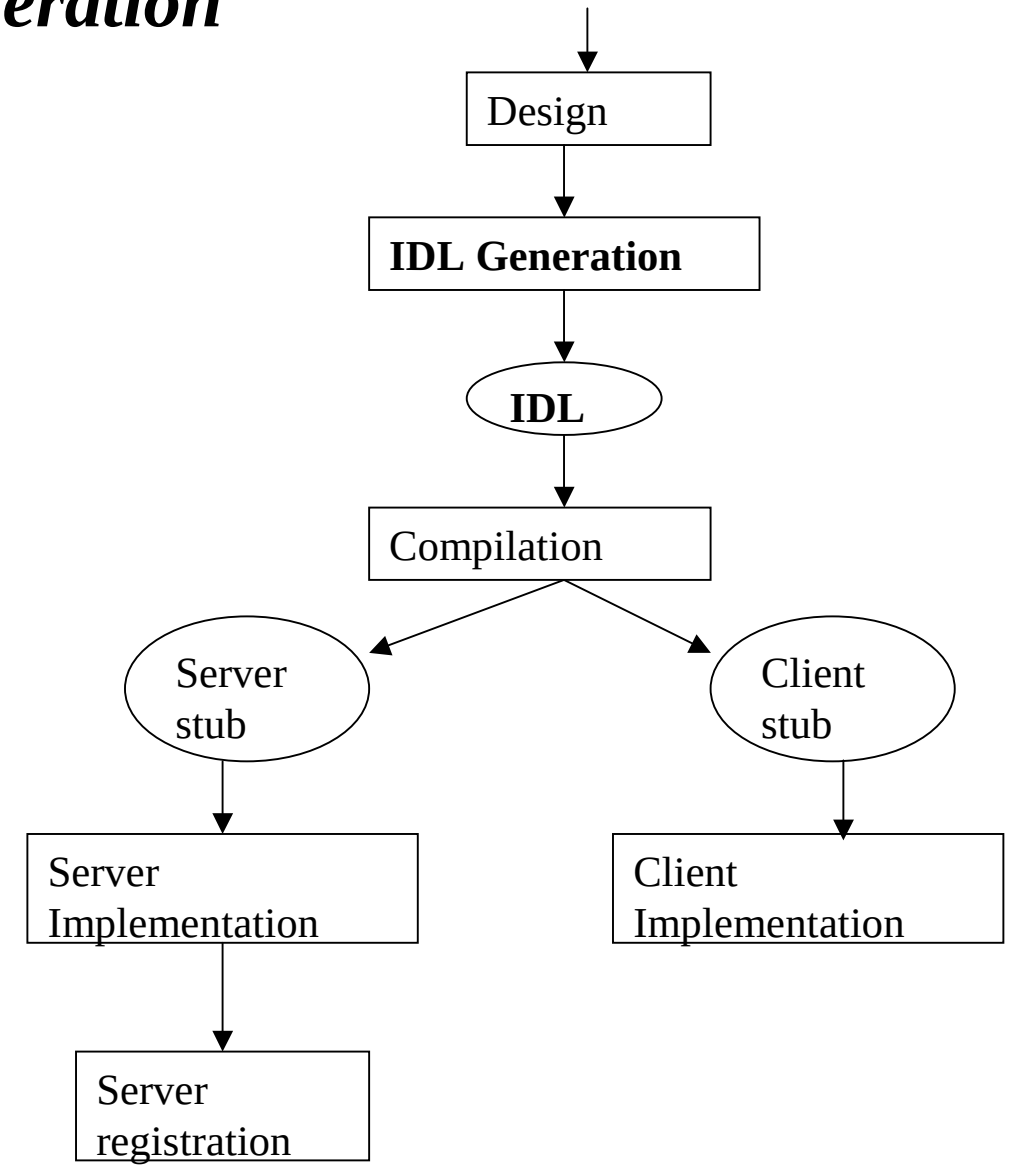
-A language binding defines a one-to-one direct mapping between its constructs and the IDL constructs:

- 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.

IDL Generation



2. Basic IDL

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:

```
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:

```
interface Account {
  //Account definitions
 };

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

```
interface Account {

    attribute string balance;
    readonly attribute long ssn;

 };
```

- For read-write attributes, there is a *set* and a *get* function generated for each attribute.
- For read-only attributes, a single *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:

```
interface Employee;    //forward declaration

interface Company {
  Employee supervisor;
  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

IDL Constants

- There is a restricted set of types including integer, character, boolean, floating point, string, and renamed types.

- 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

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.

typedef *EmployeeRecord Employees[100];*

Union type

```
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:

```
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 structures types.

- Example:

```
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:

```
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:***

- ***Abstract IDL model***

```
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:*

```
module CourseRegistration {
  // Forward Declarations
  interface Course;

  interface 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 SchoolSemestsers {FALL, SPRING, SUMMER};
       attribute SchoolSemesters semester;
       void register(in Student student);
       void cancel();
     };
};
```

3. IDL to Java Language Mapping

Programming Conventions

-Programming conventions for Java and IDL differs slightly:

- •IDL convention does not require capitalization for the names of modules, interfaces, or operations.
- •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
module BookStore {
    interface Account {
            …
    };
  };

//Java code generated by idltojava compiler would include:
package BookStore;
        …
```

IDL Types

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

IDL	**Java**
float	float
double	double
long, unsigned long	int
long long, unsigned long long	long
short, unsigned short	short
char, wchar	char
boolean	boolean
octet	byte
string, wstring	java.lang.String
enum, struct, union	class

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
typedef string CustomerName;
typedef sequence <long> CustomerOrderID;
```

IDL sequence

- 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 bounded sequence (but with different semantics).

Example:

```
//IDL
  const long length=20;
  typedef string custName[length];
```

IDL enum

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

```
//IDL
enum CityList {Boston, NewYork, Philadelphia, Baltimore};
```

//maps to a Java final class with the same name: *CityList.java*

IDL struct

- Maps to Java class with public data members.

- Example:

```
//an IDL struct
struct Book {
    string title;
    string author;
    string isbn_number;
    float price;
};
//maps to a Java class that is final: Book.java
```

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:

```
module bank {
        interface Account {
                void deposit();
        };
};
```

idlj –fall bank.idl command will generate the following files:

-*Account.java*: contains the java version of the IDL interface; used as signature type in method declarations.

-*_AccountImplBase.java*: a java class that contains the **skeleton** code

-*_AccountStub*: a java class that contains the **stub** code

-*_AccountHelper.java:* a **Helper** class that is used to narrow the object reference returned from a Naming Service to the stub required by the client.

-*_AccountHolder:* a **Holder** class that is used to contain a reference to the IDL interface object if the interface is passed as an argument.

-*AccountOperations:* all the operations defined in the IDL interface are put into this file, which is shared by both the stubs and the skeletons.

Attributes

- An attribute will generate an accessor and mutator 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 Example {
interface Modes {
        long operation(in long inArg,
                                out long outArg,
                                inout long inoutArg);
};
};


// Generated Java
package Example;
public interface ModesOperations {
        int operation(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() {...}
}
```

```
// Helper Class
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){...}
}
```

```
// user Java code
// select a target object
Example.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.operation(inArg, outHolder, inoutHolder);

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

// use the value of the inoutHolder
... inoutHolder.value ...
```

IDL Exception

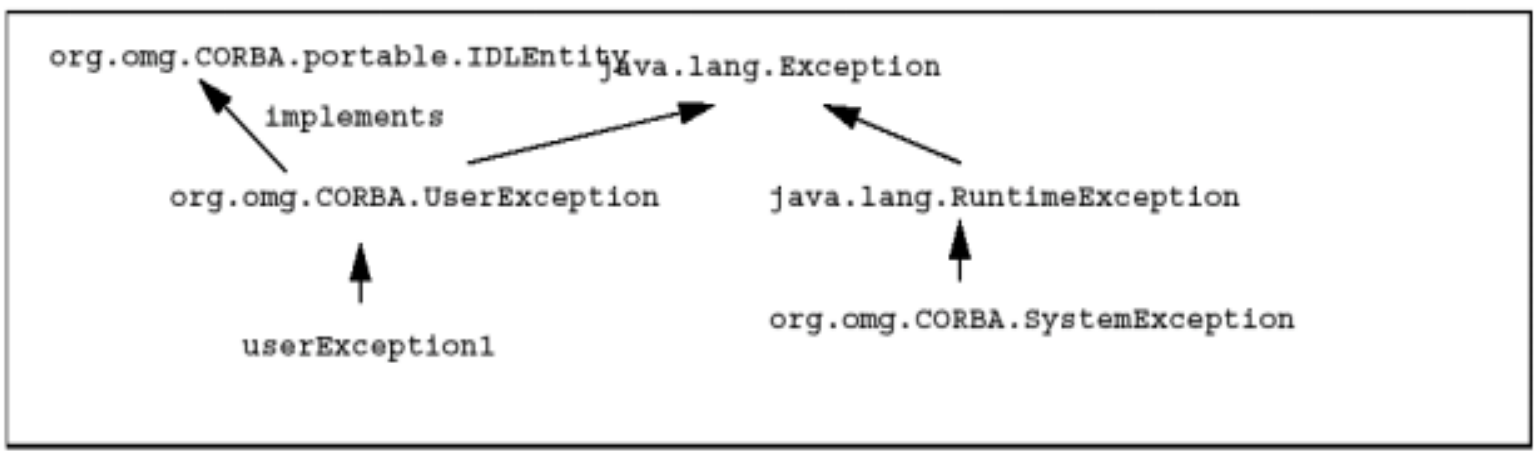
- Don't map directly onto the Java Exception API
- Example:

//an IDL exception

```
exception AccountException {
  string reason;
  float creditLine;
};
```

//gets compiled into a Java class definition (*AccountException.java*)

Inheritance of Java Exception Classes



Example

```
// IDL
module Example {
        exception ex1 {long reason_code;};
};

// Generated Java
package Example;
final public class ex1 extends org.omg.CORBA.UserException {
        public int reason_code; // instance
        public ex1() { // default constructor
                super(ex1Helper.id());
        }
        public ex1(int reason_code) { // constructor
                super(ex1Helper.id());
                this.reason_code = reason_code;
        }
        public ex1(String reason, int reason_code) {
        // full constructor
                super(ex1Helper.id()+" "+reason);
                this.reason_code = reason_code;
        }
}
```