Sistemas de Información

Tecnologías de objetos distribuidos:

CORBA: El lenguaje IDL (Invocación estática)

Agradecimientos: Jesus Villamor Lugo, Simon Pickin de IT/UCIIIM, Juan Pavón UCM, Fernándo Bellas UDC

CORBA IDL. Invocación estática
Índice

- ¿Qué es IDL?
- Características de IDL
- Jerarquía de tipos en IDL
- El lenguaje en breves trazos
  - `module`
  - `interface`
  - `Operations y parámetros`
  - `attributes`
  - `exception,raises`
  - `Tipos en IDL (primitivos, construidos)`
  - `typedef`
- Traducción de tipos
  - Tipos simples
  - Tipos complejos
  - `Const`
- Referencias

¿Qué es IDL?

- **IDL** *Interface Definition Language*
  - ¿Qué es?
    - Es un lenguaje de definición de interfaces
    - Especifica la sintaxis de los interfaces
    - Es neutral:
      - independiente del leng. de programación
      - Independiente del sistema operativo
  - ¿Para qué sirve?
    - Establecer un contrato entre cliente y servidor indicando que servicios van a estar accesibles para el cliente desde el servidor.

¿Qué es IDL?

- ¿Cómo se crea?
  - Es un fichero con extensión .idl que consta de:
    - Declaración de módulos
    - Declaración de interfaces
      - Soportan herencia
      - Contienen atributos y operaciones
    - Declaración de tipos de datos,constantes y excepciones necesarios para definir las operaciones y atributos
- ¿Cómo se usa?
  - Se utiliza un compilador de idl que genera código para el lenguaje destino para poder invocar operaciones e implementar las interfaces.
  - Existen mappings estandarizados para múltiples lenguajes (C, C++, Java, Cobol, Smalltalk, Ada, …)
  - Al compilar se generan varios ficheros (ej. en java)
    - El Stub de cliente y el Skeleton del servidor
    - Clases Holder y Helper
    - Una clase para la implementación base

Características de IDL

- Es un lenguaje neutro y de sólo definición
  - La manipulación de lo definido se hace en cualquiera de los lenguajes (generalmente OO) para los que se haya generado Stubs y Skeletons
  - Aunque su sintaxis está fuertemente basada en C (Java y C++)
- El fichero IDL crea su propio ámbito para cada uno de sus elementos
  - Los identificadores sólo pueden ser definidos una vez dentro de un ámbito
    - Los identificadores son sensitivos a las mayúsculas
  - Las operaciones dentro de un mismo ámbito no pueden sobrescribirse
    - No existe polimorfismo
- Permite herencia múltiple de interfaces
  - Siempre que las interfaces heredadas no definan la misma operación

El lenguaje en breves trazos

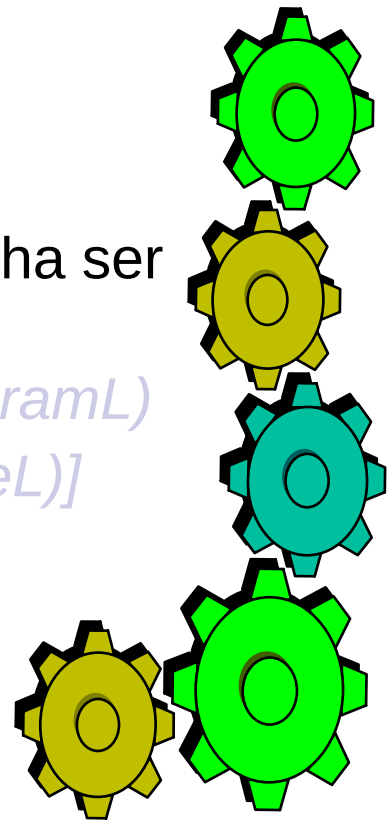
- Interfaces. Conjunto de operaciones que un cliente puede "*requerir*" sobre un objeto.

  ("*Target Object*" = Objeto de tipo interfaz **X**)

- Operaciones. Entidad que denota un servicio que va ha ser requerido.

  [oneway] <op_type_spec><identifier> (param1,...,paramL)
  [raises (except1,..,exceptL)] [context(name1,...,nameL)]

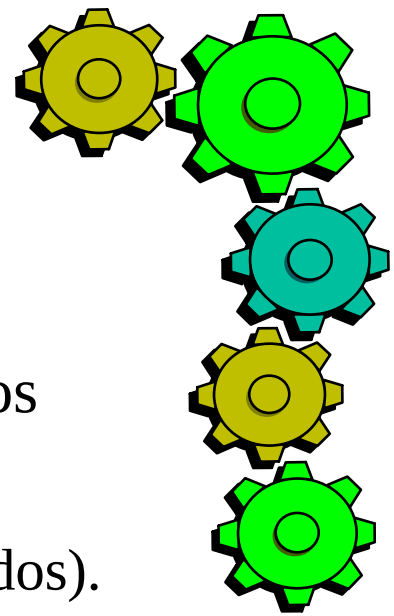
- Atributo. Define dos operaciones:
  - Get_atributoX
  - Set_atributoX
  - Puede ser de solo lectura (get_atributoX)



Implementación de los Objetos

La implementación de los objetos llevan consigo los conceptos relativos a su comportamiento en un sistema computacional

Existen dos modelos:



- ***Modelo de ejecución***: Describe "***como***" los servicios son "***ejecutados***"
- ***Modelo de construcción***: Describe "***como***" los servicios son "***definidos***" (define el estado de los objetos, métodos,infraestructura de selección de métodos).

Ejemplo IDL

```
module Banca {
  interface Cuenta {
    exception SaldoInsuficiente {  float balance;  };
    readonly attribute float balance;
    void ingreso(in float suma);
    void reintegro(in float suma) raises (SaldoInsuficiente);
   };

   interface CuentaCredito : Cuenta {
    readonly attribute float credito;
   };

   interface Banco {
    exception Rechazada {  string causa;  };
    exception CuentaInexistente { };
    Cuenta abrirCuenta (in string cliente) raises (Rechazada);
    CuentaCredito abrirCuentaCredito (in string cliente, in float credito)
    raises (Rechazada);
    void cerrarCuenta(in Cuenta c) raises (CuentaInexistente);
   };
};
```

Elementos del lenguaje IDL

- **Identificadores**
- **Comentarios**
- **module**
- **interface**
- **Operations y parámetros**
- **attributes**
- **exception, raises**
- **typedef**

Elementos del lenguaje **identificadores y comentarios**

- Identificadores
  - Deben comenzar con un carácter alfabético seguido por cualquier número de caracteres alfabéticos, dígitos o "_" (subrayado)
  - No distingue mayúsculas y minúsculas pero deben usarse consistentemente:
    - `miObjeto y MIOBJETO` son identificadores distintos pero no pueden usarse enla misma declaración de interfaz
    - El objetivo es facilitar el mapping a todo tipo de lenguajes:
      - Los que distinguen mayúsculas y minúsculas (C++ y Java)
      - Los que no distinguen mayúsculas y minúsculas (ADA)
  - Conviene evitar todas aquellas palabras que sean reservadas en algún lenguaje de destino (`package`,`self`, `etc`.)

- Comentarios
  - `// Esto es un comentario de una línea`
  - ```
    /* Esto es un comentario
      de varias líneas  */
    ```
    ```

Definiendo nuevos ámbitos **`module`**

- Permite agrupar varias definiciones IDL que comparten un mismo propósito
- El constructor **`module`** se usa para proveer a los identificadores IDL de un ámbito definido por el usuario evitando así conflictos de nombres

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

- Para hacer referencia a un identificador de otro módulo se usa **`::`**

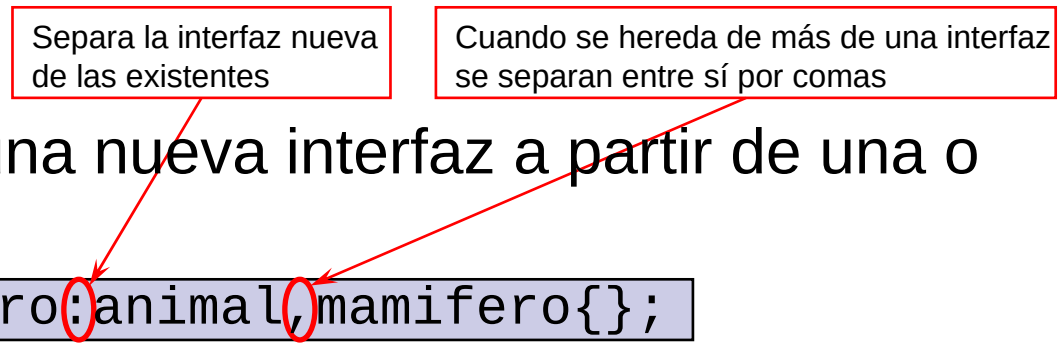
```
bank::Account
```

- Caso IDL→Java: el `module` se traduce a `package`
  - `Account` puede ser referenciado como `bank.Account`

Definiendo interfaces `interface`

- Las interfaces definen servicios IDL
  - Son una colección de atributos, excepciones y operaciones
- Caso IDL→Java: la precompilación produce
  - Una interfaz Java
    - Especifica los métodos que el cliente puede invocar
  - Unas clases Java
    - Una es un esbozo de la clase que implementa la interfaz Java
    - Otras provee funcionalidad para los proxies (stub and skeleton)
- El nombre de los ficheros generados varía dependiendo del compilador de IDL

Declaración de interfaces
Herencia



- Permite obtener una nueva interfaz a partir de una o varias existentes
  - `Interface perro:animal,mamifero{};`
  - Soporta herencia múltiple
  - Todas heredan de `Corba::Object`
  - Se pueden añadir nuevos elementos:
    - Constantes
    - Tipos
    - Atributos
    - Operaciones
  - Restricciones
    - No se permite **sobreescritura**: No se pueden redifinir operaciones heredadas
    - No se permite **sobrecarga**: No puede haber operaciones con el nombre de las ya existentes

Declaración de operaciones

- Una operación IDL
  - Requiere
    - Un tipo de retorno o **`void`**
    - Un identificador o nombre de operación
    - Cero o más parámetros direccionales (**`in`**,**`out`**, **`inout`**)
  - Opcional
    - Un modificador opcional **`oneway`**
    - Una cláusula opcional **`raise`**
- **`Tipo de retorno`** Permite que el cliente reciba copia de un valor inicializado
- **`Oneway`** y **`void`**
  - Comunicación asíncrona (no bloqueante)
    - tipo de retorno **`void`** con estilo **`oneway`**
    - cliente no espera respuesta
    - No puede tener parámetros **`out`** ni **`inout`**
    - No puede tener claúsula **`raises`**
  - Comunicación síncrona (bloqueante): Es el modo por defecto
    - tipo de retorno **`void`** sin estilo `oneway`
    - cliente espera respuesta sin datos
    - Se queda bloqueado hasta recibir confirmación de la terminación o no de la operación

Declaración de parámetros

- Una declaración de parámetros consiste en
  - el modo: **`in`**, **`out`**, **`inout`**
  - un tipo
  - un identificador o nombre de parámetro
- El modo es uno de
  - **`in`**: el parámetro se pasa del cliente al servidor
    - El cliente pasa una copia de un valor inicializado al servidor
    - El servidor no puede hacer ninguna modificación sobre ese valor
  - **`out`**: el parámetro se pasa del servidor al cliente
    - El cliente pasa una referencia de un valor sin inicializar al servidor
    - El servidor inicializa este valor modificando también el valor de la var cliente
  - **`inout`**: el parámetro se pasa en ambas direcciones
    - El cliente pasa una referencia de un valor al servidor
    - El servidor puede usar esta referencia para acceder al valor inicial y modificarlo
- Caso IDL→Java: parámetros pasados por referencia (**`out`**, **`inout`**) se traducen a clases Holder (envoltorios)
  - El ORB proporciona clases Holder para los tipos IDL de base

Declaración de operaciones
¿Cómo elegir modo de los parámetros?

- Si la operación acepta uno o más parámetros y devuelve un único resultado
  - Usar **`in`** para los parámetros
  - Usar valor de retorno para el resultado
- Si la operación tiene varios valores de retorno con igual importancia
  - Usar **`out`** para los valores de retorno
  - Usar **`void`** para el resultado
- Si la operación retorna varios valores pero uno es más importante que el resto
  - Usar el valor más importante como valor de retorno
  - Usar **`out`** para el resto de los valores
- Usar parámetros **`inout`** con precaución
  - Se asume que el que hace la invocación no quiere conservar valor
  - Útil cuando se pasan valores grandes
  - Ejemplo: `invertMatrix (inout Matrix aMatrix);`

Declaración de Atributos

- **No** representan el estado del objeto que implementa el interfaz
  - Sólo definen el acceso a un valor para leerlo o modificarlo
  - Si sólo se puede leer hay que indicarlo con **`readonly`**
  - Inconveniente: no pueden tener clausula **`raise`**
- Un atributo normal de IDL se traduce a dos operaciones de la clase servidor
  - Una operación de **lectura**
    - <tipo> **get**<nombre de la operación>();
  - Una operación de **escritura**
    - <tipo> void **set**<nombre de la operación>(param);
    - Ésta no aparecerá si el atributo es **`readonly`**
- Un atributo **`readonly`** de IDL se traduce a una sola operación de lectura de la clase servidor

Declaración de Atributos

Caso IDL→Java:

```
interface Account {
        readonly attribute float balance; // Sólo lectura
        attribute long accountNumber;     // Lectura y escritura
        void deposit(in float money);
};
```

Al precompilarse (con el compilador de IDL→Java) producirá:

```
public interface AccountOperations {
        float getbalance;                 // Lectura
        in getaccountNumber();            // Lectura
        void setaccountNumber(int arg);   // Escritura
        void deposit(float money);
};
```

Manejo de excepciones

Cláusulas **`raises`** y **`exceptions`**

- Pueden ser:
  - Excepciones estándar de CORBA
  - Excepciones definidas por el usuario (pueden tener campos)
- Definición de una excepción
  - Se define dentro de un ámbito `module` o `interface`

```
exception <identifier>  “{“ <member>* “}”
```

- Cláusula que indica que una operación levanta alguna excepción

```
raises ( MyExc1 [, MyExc2 ...] )
```

- Caso IDL→Java:
  - user exceptions → **`final`** Java classes que heredan de *org.omg.CORBA.UserException* que hereda de *java.lang.Exception*
  - CORBA system exceptions → **`final`** Java classes que heredan de *org.omg.CORBA.SystemException* que hereda de,*java.lang.RuntimeException*

Manejo de excepciones

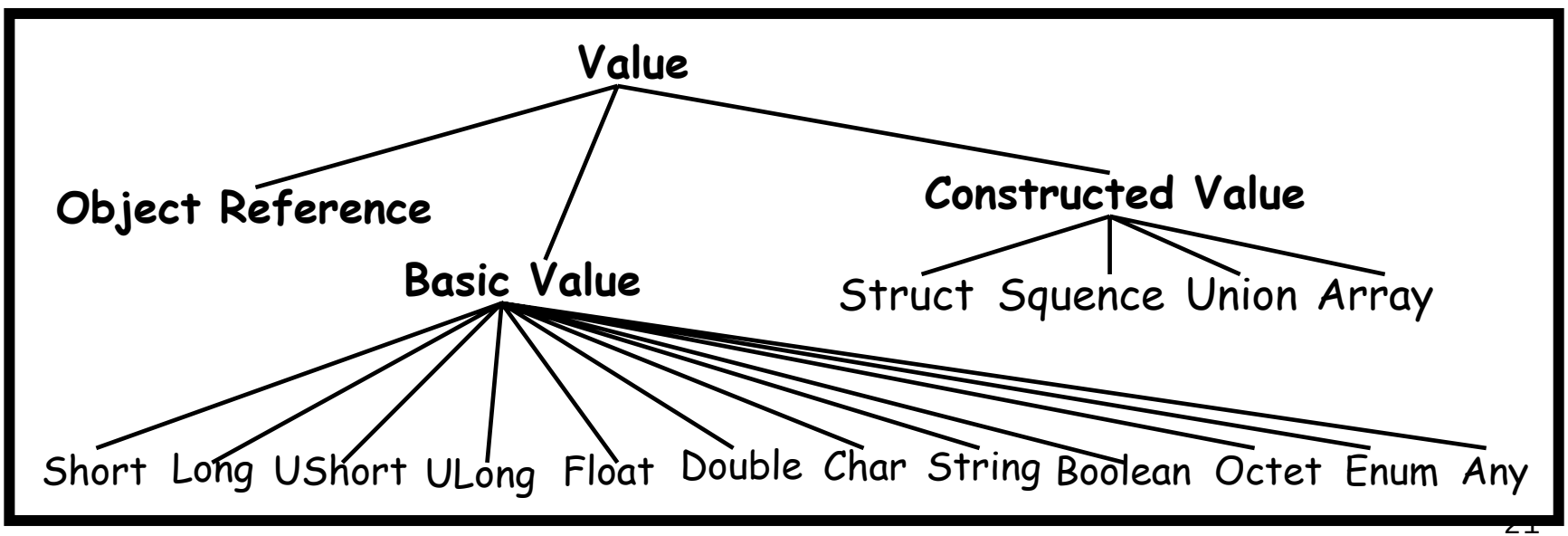
Aspectos a tener en cuenta

- Usarlas para reflejar situaciones en las que la operación no puede concluir normalmente.
  - Rompen flujo de control normal del programa
- Evitarlas cuando hay otras soluciones
  - Ejemplo si los parámetros no cumplen cierta condición devolver resultado vacío en vez de lanzar excepción
- Proporcionar información relevante para el usuario
  - Usar una excepción distinta para cada tipo de fallo en vez de excepciones genéricas
  - Añadir información útil en el mensaje por ejemplo en vez de devolver el valor incorrecto de un parámetro devolver el valor límite

Jerarquía de tipos en IDL

- Tipos. Entidad identificable con un conjunto de características.
  - Miembros de un tipo: Valor que satisface un tipo

(***tipo objeto → miembros objetos***)



Tipos de variables

- Primitivos
  - Entero:
    - short, long, long long
    - signed & unsigned
  - Coma flotante:
    - float, double, long double, fixed
  - char, wchar, boolean, octet
  - Any
  - Referencia a objeto CORBA
- Construidos
  - struct
  - union
  - enum
  - sequence
  - string, wstring
  - array
  - typedef

Tipos primitivos

`void`	para métodos que no devuelven ningún valor
`boolean`	TRUE o FALSE
`char` y **`wchar`** (*)	un carácter guardado en 8 o 16 bits
`short`	-215..215-1 (16 bits)
`unsigned short`	0..216-1 (16 bits)
`long`	-231..231-1 (32 bits)
`unsigned long`	0..232-1 (32 bits)
`long long` (*)	-263..263-1 (64 bits)
`unsigned long long` (*)	0.. 264-1 (64 bits)
`float`	núm de coma flotante precisión simple (IEEE)
`double`	núm de coma flotante precisión doble (IEEE)
`long double` (*)	núm coma flotante precisión extendida (IEEE)
`octet`	grupo de 8 bits que se transmiten tal cual
`any`	valor de cualquier tipo IDL

(*) Disponibles a partir de Corba 2.1 no soportados por ORBs antiguos

Any

- any
  - Sirve para especificar que un parámetro puede ser de cualquier tipo
  - El proceso que recibe un any debe determinar qué tipo de valor tiene y extraer ese valor
  - Hay que utilizar el tipo any con precaución porque hace las interfaces más difíciles de entender y no deja claro qué tipos se pasan
  - Se usa cuando no se conoce el tipo en tiempo de compilación (ejemplo: eventos)

```
inteface ejemploAny {
        void operacion (in any a);
};
```

Tipo string

- El tipo string en IDL representa una cadena de caracteres
  - De tipo char: string
  - De tipo wchar: **wstring** (*)
- Si se quiere limitar la longitud de un string, se puede indicándola entre < y >
  - `typedef string nombre;`
  - `typedef string<5> codigo_postal;`
  - `typedef wstring cadenaUnicode;`

(*) Disponibles a partir de Corba 2.1 no soportados por ORBs antiguos

Tipos construidos

- enum
  - Define un nuevo tipo con un conjunto de valores definidos por el usuario
  - Los valores deben ser únicos dentro de su ámbito
  - Estos valores no se pueden inicializar

```
enum Lenguaje {Java, C, Smalltalk, Cobol, Perl};
enum  Dia {lunes, martes, miercoles, jueves, viernes,
sabado, domingo};
enum Color {rojo, amarillo, verde, azul, blanco, negro,
gris};
```

Tipos construidos

- struct
  - Empaqueta un conjunto de miembros nombrados de varios tipos

```
struct sistema {
       string nombre;
       Lenguaje      programacion;
       short  nro_usuarios;
};
```

Tipos construidos

- union
  - Estructura que contiene un solo miembro de entre varias alternativas

```
union compilador switch (Lenguaje) {
      case Java:    string        versionstring;
      default:      long          versionlong;
};
```

Tipos contenedores

- sequence
  - Representa una secuencia de datos de un mismo tipo
  - Puede ser limitada a una longitud máxima o ilimitada (opción por defecto)
  - Una secuencia puede estar vacía
  - Tienen que definirse con typedef o dentro de un struct

```
typedef sequence<Banco> bancos;   // secuencia ilimitada

typedef sequence<long,3> numeros; // secuencia limitada de 3 long

struct  contribuyente {
        string                  nombre;
        sequence<string, 2>     apellidos;
}
```

Tipos contenedores

- array
  - Representa un array de una o más dimensiones de tamaño fijo
    - Para varias dimensiones se usa la sintaxis [m][n]
    - Siempre hay que especificar *todas* las dimensiones del array
    - Tiene que definirse en un typedef

```
typedef Banco  arrayDeBancos[10];

typedef long  matriz[6][8];
```

Tipos contenedores

- Comparación entre array y sequence
  - En un array se transmiten todos sus elementos en cada llamada a operación
  - Un array es siempre de longitud fija, una secuencia puede variar de tamaño (más o menos)
  - Un array no puede ser ilimitado, una secuencia sí
  - Una secuencia puede no tener ningún elemento, un array no
  - Un array puede ser multidimensional, las secuencias no (pero puede haber secuencias de secuencias)

Tipos contenedores

- Cuándo usar array o sequence
  - En general, si el número de los elementos puede variar, usar un sequence
  - Si se tiene una colección de elementos y su número siempre es el mismo, usar un array
  - Se pueden utilizar arrays de *char* para tratar strings de longitud fija
    - Ejemplo:

```
typedef char CodigoPostal[5];
```

  - Con las secuencias es más fácil definir estructuras de datos recursivas
    - Ejemplo:

```
struct NodoArbol {
       Any    contenido;
       sequence<NodoArbol>  hijos;
};
```

Definición de tipos **`typedef`**

- **`typedef`** asocia un nombre con un tipo de datos
  - Permiten realizar especificaciones más claras dando nombres significativos a tipos ya existentes
  - Deben construirse a partir de tipos ya construidos previamente
  - Ejemplo (a partir de tipos primitivos)

```
typedef long IDNumber;
typedef string SSNumber;
```

  - Evitar alias innecesarios

```
typedef IDNumber ID;
```

Traducción de constantes **`const`**

- Un **`const`** se utiliza para especificar valores inmutables
- Se pueden definir de cualquier tipo predefinido (excepto any) o de un enumerado
- IDL→Java:
  - Si se declara dentro de una interfaz se traduce en un campo **`public static final`** dentro de la interfaz Java correspondiente.
  - Si no, se traduce en una interfaz publica java con el mismo nombre y que contiene un campo **`public static final`**:

La constante

```
const float pi = 3.1415926;
```

Al precompilarse (con el compilador de IDL→Java) producirá:

```
public interface pi {
        public static final float value = (float) 3.1415926;
}
```

Traducción de tipos primitivos

IDL	Java
float	float
double	double
long	Int
short	short
unsigned long	int
unsigned short	short
char, wchar	char
boolean	boolean
octet	byte
string	String (class)
long long	long
unsigned long long	long
void	void
any	org.omg.CORBA.Any

Traducción de tipos complejos **`enum`**

- IDL→Java:
  - un **`enum`** se traduce en una clase java con
    - dos campos **`static final`** para cada miembro,
    - un constructor privado,
    - Dos métodos:
      - **`value()`**que devuelve el valor entero
      - **`from_int`** que devuele el enumerado a partir del valor entero especificado.
- El compilador también genera clase *Holder* y clase *Helper*

Traducción de tipos complejos **enum**

- Ejemplo:

  El enumerativo `enum Colores { rojo, verde, azul }`

  Al precompilarse (con el compilador de IDL→Java) producirá:

```
public class Colores {
  public static final int _rojo = 0;
  public static final Colores rojo = new Colores(_rojo);
  public static final int _verde = 1;
  public static final Colores verde = new Colores(_verde);
  public static final int _azul = 2;
  public static final Colores azul = new Colores(_azul);
};
private int Colores(int value){ /*…*/ };
public int value(){ /*…*/ };
public static Colores from_int(int value){ /*…*/ };
```

Traducción de tipos complejos **`struct`**

- IDL→Java: un **`struct`** se traduce en una clase Java del mismo nombre que contiene dos constructores
  - Sin parámetros
  - Con parámetros para inicializar todos los campos del struct
  - Ejemplo

```
module Bank {
        struct CustomerDetails {
                string Name;
                string Address;
        };
};
```

Al precompilarse (con el compilador de IDL→Java) producirá:

```
package Bank;
public final class CustomerDetails {
        public String Name;
        public String Address;
        public Bank();
        public Bank(String Name, String Address);
};
```

- El compilador también genera clase *Holder* y clase *Helper*

Traducción de tipos complejos **sequence**

- **sequence** es un array unidimensional con dos características
  - Una longitud máxima (en tiempo de compilación)
  - Una longitud (en tiempo de ejecución)
- Las secuencias pueden ser limitadas o ilimitadas
  - Limitadas (bounded) a un número fijo de elementos

    ej. `sequence< long > UnboundedLongSeq;`
  - Ilimitadas (unbounded) sin límite de elementos

    ej. `sequence< long, 10 > BoundedLongSeq;`
- IDL→Java: una secuencia se traduce a
  - Un array java (con el mismo nombre)
  - Compilador también genera clase *Holder* y clase *Helper*

Traducción de tipos complejos **sequence**

```
// IDL
typedef sequence<long,8> numeros; //
secuencia limitada de 3 long
interface Prueba {
      void suma(in numeros sumandos);
};
```

```
// Java
int[] nums = new int[5];
for (int i=0; i<5; i++)
      nums[i] = i;
refPrueba.suma(nums);
```

Traducción de tipos complejos **`array`**

- IDL→Java: un array se trata de la misma forma que una secuencia limitada (bounded)
- En concreto:
  - se traduce a un **`array`**
  - la longitud del **`array`**, se conoce por la constante **`length`**, presente en todos los **`arrays`** Java
- Compilador también genera clase *Holder* y clase *Helper*
- Si se trata de transmitir array con longitud distinta a la definida se produce excepción como ocurría con secuencias: org.omg.CORBA.MARSHALL

```
typedef Banco  arrayDeBancos[10];
typedef long  matriz[6][8];
```

```
Banco[] arrayDeBancos = new Banco[10];
long[][]  matriz = new long[6][8];
```

Traducción de tipos complejos **`union`**

- Una **`union`** es una estructura que en cualquier momento referencia sólo uno de los tipos posibles
- El tipo realmente contenido se referencia con el "discriminador"
- IDL→Java: un **`union`** se traduce en una clase Java con el mismo nombre y que incluye:
  - un constructor
  - un método para acceder al discriminador
  - un método para acceder a cada rama
  - un método para modificar a cada rama
  - un método para modificar a cada rama con más de una etiqueta

Traducción de tipos complejos **union**

```
union compilador switch (lenguaje) {
      case Java:   string       versionstring;
      default:            long  versionlong;
};
```

```
// código en el cliente:
compilador c = new compilador();
c.versionstring(“3.2”);

// código en el servidor:
switch (c.discriminator()) {
      case Java:
      System.out.println(c.versionstring());
                  break;
      default:
      System.out.println(c.versionlong());
};
```

Ejemplo IDL

```
module Banca {
  interface Cuenta {
    exception SaldoInsuficiente {  float balance;  };
    readonly attribute float balance;
    void ingreso(in float suma);
    void reintegro(in float suma) raises (SaldoInsuficiente);
   };

   interface CuentaCredito : Cuenta {
    readonly attribute float credito;
   };

   interface Banco {
    exception Rechazada {  string causa;  };
    exception CuentaInexistente { };
    Cuenta abrirCuenta (in string cliente) raises (Rechazada);
    CuentaCredito abrirCuentaCredito (in string cliente, in float credito)
    raises (Rechazada);
    void cerrarCuenta(in Cuenta c) raises (CuentaInexistente);
   };
};
```

Ejercicio de IDL
Examen 26 de enero de 2002

Defina un fichero IDL "Empresa.idl" con las siguientes características:

Consta de un empleado "**Empleado**" y una empresa "**Empresa**":

1. El "**Empleado**" tiene un "**numeroEmpleado**", un "**nombre**", una "**direccion**", un "**sueldo**" y un "**jefe**". Con dos operaciones: "**paga**" (que acepta el "**mes**" y devuelve "**mensualidad**") y otra "**cambioDireccion**" (que acepta la "**dirección**").
2. La "**Empresa**" tiene cuatro operaciones: "**nuevoEmpleado**", "**despido**", "**promocion**" y "**nuevoJefe**": ambas excepcionalmente pueden "**Rechazar**" hacer la operación.
   - "**nuevoEmpleado**" acepta todos los atributos del empleado, salvo el "**numeroEmpleado**" que lo genera:L también genera un Empleado.
   - "**despido**", se aplica sobre un "**Empleado**".
   - "**promocion**" se aplica sobre un "**Empleado**" al que se le sube el "**sueldo**";
   - "**nuevoJefe**" se aplica a un "**Empleado**" al que se le asocia un nuevo "**jefe**".

NOTAS:

1. El mes es un enumerativo que hay que construir con los 12 meses de un año.
2. El nombre y la dirección son cadenas de caracteres.
3. El sueldo es un número en coma flotante
4. El jefe es un empleado.
5. Respete los "**nombres**" que se ha dado a cada concepto.
6. Se considera equivalente el valor de retorno a un parámetro de salida.

Algunas referencias

– ***Dan Harkey, Robert Orfali***, ***Client/Server Programming with Java and CORBA, 2nd Edition*** *2nd. Edition (1998) John Wiley & Sons, Inc. ISBN: 0-471-24578-X Mirarse los capítulos 2, 3 y 21*

– ***Andreas Vogel and Keith Duddy (1998).*** *JAVA Programming with CORBA (Advanced Techniques for Building Distributed Applications). John Wiley & Sons, 1998. Inc. ISBN 0-471-24765-0*

–***Vinoski, Steve (1997).*** *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communications Magazine. Febrero 1997.*

–Consultar página web de la asignatura para referencias actualizadas y refs web