인터페이스 정의언어(IDL)

이장에서 배울 내용은

- IDL
- IDL 파일작성하기
- 인터페이스
- 모듈
- 클래스와의 차이
- 객체형과 데이터형
- 데이터형

이번 장에서는 인터페이스 정의 언어인 IDL 에 대해서 알아보자.

■ IDL

CORBA 의 경우 분산된 네트워크상에 따로 위치한 객체 간의 투명한 접근을 제공하는 서로간의 약속이 필요하다. 이런 약속을 정의할 때 특정 언어(C, C++, Java 등)에 의존하지 않는 인터페이스 정의 언어가 필요하게 되었는데, 그것이 바로 IDL(Interface Definition Language)이다. 서버와 클라이언트가 서로 통신을 하기 위해서 서버는 클라이언트에게 제공하는 서비스 인터페이스를 IDL 로 정의하게 되며, 클라이언트는 이런 인터페이스 정보를 활용하여 서비스를 활용하게 되는 것이다. CORBA 프로그램을 개발하기 위해서는 가장 먼저 IDL 을 정의해야 하는데, IDL 은 구현에 대한 정보는 포함하고 있지 않아 정의된 IDL 을 원하는 언어로 매핑한 다음 실제 구현 작업을 하게 된다.
물론, 인터페이스를 정의하는 방법이 IDL 만 있는 것은 아니다. Visibroker 의 경우 Caffeine 이라는 것을 이용하면 IDL 을 사용하지 않아도 되며, Java 의 RMI 나 RMI-IIOP 를 이용해면 IDL 을 몰라도 인터페이스를 정의할 수 있다. 하지만, IDL 은 OMG 에서 규정하고 있는 인터페이스 정의 언어의 표준이고 개발자가 익히기에 어렵지 않은 만큼 CORBA 프로그램을 할 때는 꼭 IDL 을 사용하도록 하자.

- IDL 은 특정 언어에 독립적인 인터페이스 정의 언어이다. 따라서, 구현언어가 아닌 정의언어이며, 구현언어로의 매핑을 지원한다.

- IDL 은 객체지향 개념을 기초로 하고 있으며, 다중상속 및 동적 호출 메커니즘을 지원한다.

■ IDL 파일 작성하기

CORBA 프로그램에 대한 여러 가지 개념적 얘기를 접하기는 쉬워도 IDL 파일을 직접 정의해서 원하는 시스템을 구현해본 사용자는 많지 않을 것으로 생각된다. COM/DCOM 의 경우에도 IDL 파일을 정의하는데, 사용법이 복잡하여 각종 Wizard 기능을 제공하는 개발 툴을 활용하게 된다. 하지만, CORBA 의 경우는 텍스트 편집기만 있으면 힘들지 않게 IDL 파일을 작성할 수 있게 된다.
일단 사용하는 텍스트 편집기를 실행한 다음 새 파일을 생성하여 임의의 위치에 저장한다. 파일 이름은 사용자가 임의로 결정해도 되지만, 흔히 모듈이름이나 인터페이스 이름을 주는 것이 알아보기 쉬우며, 확장자는 '.idl '로 하는데 이것 역시 꼭 '.idl'일 필요는 없다.

이제 IDL 파일에 들어가게 될 내용을 기입할 차례인데, 이것은 이번 장을 공부하면서 배워보도록 하자. 처음 CORBA 프로그램을 하는 사람에게는 IDL 문법이 어렵게 느껴지겠지만 조금만 익숙해지면 IDL 문법보다는 객체라는 개념이 더 어렵게 느껴지게 된다. IDL 파일을 정의할 때 지켜야 할 규칙에 대해 잠시 살펴보자.

- 식별자(moduIe, interface, 데이터형 등의 이름)는 알파벳으로 시작되며, 알파벳과 숫자 또는 언더스코어(_)로써 이루어진다. 여기서 말하는 알파벳은 ISO Latin-1 문자이다.

- 대소문자를 구별하여 일관되게 적어주어야 한다. 만약 interface 이름을 Test 라고 했으면 test 와 혼용해서 사용해서는 안 되고, 미리 정의된 키워드의 경우에도 대소문자를 구별해야 한다.

- 주석(Comment)은 C, C++, Java 와 마찬가지로 /* */, //를 사용하게 된다.

■ 인터페이스(interface)

● 인터페이스

CORBA 의 경우 분산된 네트워크상에 따로 위치한 객체 간의 투명한 접근을 제공하는 서로간의 약속이 필요하다고 앞에서 언급하였다. 이러한 약속은 기본적으로 객체 단위로 이루어지기 때문에 인터페이스는 속성(attribute)과 동작(operation)의 묶음으로 이해하면 된다.

////////////////////////////////// 알고 넘어 갑시다!! //

[객체와 클래스]
객체 지향 프로그래밍에서 하나의 기본 단위를 객체(Object)라고 말하는데, 도대체 클래스와는 어떻게 다른 것일까? 쉽게 말해 객체라는 개념적 용어를 C++에서 실제 프로그램으로 구현하게 될 때 클래스(Class)로 표현하게 되는 것이며, IDL 에서는 인터페이스, Java 에서는 클래스와 인터페이스로 표현된다.

[객체와 인스턴스]
그럼, 객체와 인스턴스의 차이는 무엇일까? 실제로 객체를 프로그램하게 될 때 클

래스라는 용어로 표현된다고 말했다. 사람이라는 객체를 프로그램으로 구현하게 될 때 Person 이라는 클래스가 등장하게 되며, Person 이라는 데이터 형으로 실제 변수를 생성하게 된다. 이렇게 하여 생성된 변수를 인스턴스라고 한다. 예를 들면, '홍 길동' '김 갑동'과 같은 것을 Person 클래스의 인스턴스라고 할 수 있다.
하지만, 이처럼 정확한 의미로 사용하기 보다는 객체와 인스턴스 개념을 혼동하여 사용하곤 한다. 이 책에서도 '인스턴스 생성'이라는 말 대신 '객체 생성'이란 용어를 혼용해서 사용하고 있으니 착오가 없기를 바란다.

[서버객체와 서버프로그램]
기존에는 클라이언트와 서버 프로그램이 객체단위로 프로그램 되어 있지 않았다. 하지만, 분산 객체 환경에서는 클라이언트와 서버 프로그램 모두 객체단위로 이루어지기 때문에 서버 프로그램을 서버 객체 혹은 구현객체(Object Implementation)라는 용어로 표현하기도 한다. 좀더 정확한 언어로 표현을 하자면, CORBA 인터페이스를 구현하고 있는 클래스를 구현객체라고 하며, 구현객체를 BOA 에 등록하여 클라이언트의 요청을 기다리는 프로그램을 서버 프로그램, 서버 프로그램을 통해 구현 객체를 호출하는 프로그램을 클라이언트 프로그램이라고 한다. 보통 구현객체와 서버 프로그램을 하나로 묶어서 서버라고 얘기하기도 하고, 구현객체를 서버라고 표현하기도 한다. 이 책에서도 서버객체와 서버프로그램, 서버라는 용어를 혼용해서 사용하고 있기 때문에 용어상의 혼란이 없기 바란다.

[속성과 동작]
객체는 속성(attribute)과 동작(operation)을 가지게 되는데, 이것을 C++나 Java 로 프로그래밍하면 멤버 데이터(Member Data)와 멤버 함수(Member Function)로 표현된다. 물론, 용어의 혼용으로 인해 속성은 property, 멤버 변수, 멤버 데이터 등으로 표현되며, 동작은 behavior, method, 멤버 함수 등으로 표현하기도 하니 착오 없기 바란다.

//

사용방법이 똑같지는 않지만 C++나 Java 의 클래스를 정의하는 것과 비슷한 과정이 인터페이스 정의 과정이다. 그럼 자세한 사용방법을 알아보도록 하자.
인터페이스는 interface 키워드 후에 인터페이스 이름을 쓰고, 이어서 중괄호로 묶은 인터페이스 본체를 기술한다. 본체의 기술이 끝난 후에는 반드시 세미콜론(;)을 붙여야 한다. 언뜻 보기에 C++ 클래스 헤더 파일을 정의하는 것과 Java 의 인터페이스를 정의하는 것이 상당히 비슷하다는 것을 알 수 있다. 기존 객체지향 언어처럼 IDL 에서도 인터페이스 본체에는 정의선언, 형(型) 선언, 예외선언, 속성 선언 및 오퍼레이터 선언이 모두 포함될 수 있다.

다음에 나오는 Sample1.idl 은 가장 일반적인 인터페이스 정의의 예이다.

```
exception UsrException
{
};

interface Sample
{

        //long  age;  // attribute 가 빠지면 에러발생
          attribute long age;
```

```
        readonly attribute string name;
        string oper1(in string arg1, out string arg2, inout string arg3);
        oneway void oper2();
        void oper3() raises (UsrException)
};
```

IDL 에서는 다른 IDL 파일을 포함하거나 #ifndef, #define, #endif 등과 같은 Preprocess 를 추가할 수도 있다. 이 규정은 ANSI C++에서 사용하던 것과 동일하다. typedef 과 const 는 기존 언어에서 사용하던 것과 동일한 의미를 가진다.

```
#ifndef _KEYWORD_INCLUDED
#define _KEYWORD_INCLUDED

#include "SimpleType.idl"

interface Sample
{
        //구조체
        struct Point
        {
                double x;
                double y;
        };

        typedef sequence<Point> Points;
        const short MAXPOINTS= 100;
};

#endif
```

또한, 상위의 인터페이스로부터 상속을 받는 것이 가능하다(다중상속도 가능). 다시 말해 상위의 인터페이스를 상속하게 되면 상속 받은 하위 인터페이스는 상위 인터페이스의 attribute 와 operation 을 그대로 가지게 된다. 구현언어에서는 상속 받은 attribute 와 operation 을 재정의 하는 것이 가능하지만, IDL 은 정의언어이기 때문에 상속 받은 attribute 와 operation 을 다시 재정의 할 수 없다. 아래 예처럼 (2), (3) 은 상위 A 인터페이스에서 정의된 i 와 oper( ) 함수를 재정의하려 했기 때문에 에러가 발생하며, (1)은 C 인터페이스가 A 와 B 인터페이스를 다중상속 받는데 이름이 똑같아 발생하는 에러이다.
상속에 대해 더 자세한 내용은 다른 C++나 Java 교재를 참고하기 바란다.

```
interface A
{
        attribute long l;
        void oper( );
};
interface B
{
//      attribute long l; // 이름 충돌 에러 발생          -------(1)
};

interface C:A,B
{
//      attribute long l; // 재정의 시도 에러발생          -------(2)
```

```
//          void oper ( ); // 재정의 시도 에러발생          -------(3)

};
```

● attribute 선언

인터페이스에는 attribute 와 operation 이 포함된다. attribute 에는 속성이 포함되는데 디폴트는 read-write 이며, IDL 컴파일러는 자동으로 attribute 에 접근할 수 있는 set( )/get( ) 함수를 만들어준다. 문법적으로 attribute 을 선언할 때는 attribute 라는 키워드를 사용하며 이를 빠뜨리게 되면 에러가 발생한다. attribute 앞에 read-only 속성을 추가 할 수 있는데, 이 경우에는 get( )함수만 생성되게 된다.

```
interface Sample
{
        //long  age; //에러발생
          attribute long age;
          readonly attribute string name;
};
```

● operation 선언

인터페이스에 operation 을 추가 할 때는 method 이름, method 리턴형, parameter 데이터 형과 주고 받는 방향(in, out, inout), 에러가 발생했을 때에 반환되는 예외 등을 지정한다.

```
interface Samp le
{
        string oper(in string arg1, out string arg2, inout string arg3);
};
```

데이터를 주고 받는 방향을 의미하는 키워드를 제외하고는 기존 구현언어에서 사용하던 operation 선언 방식과 크게 다르지 않다. 이처럼 데이터를 주고 받는 방향을 통해 언어의 설정 자체가 클라이언트/서버 환경에 기초하고 있음을 알 수 있다. 키워드 중 in 은 클라이언트에서 서버로, out 은 서버에서 클라이언트로, inout 양방향을 의미한다. 이처럼 데이터의 전달 방향을 지정하는 키워드는 Java 에서는 큰 의미를 가지지 않겠지만(정확하게 얘기하면 Java 에서는 함수의 파라미터를 전달할 때 파라미터 값을 복사하는 것을 원칙으로 하고 있어서 out 이나 inout 의 경우는 Holder 라는 메커니즘을 제공하고 있다), C++의 경우는 메모리 관리 규칙을 결정짓는 중요한 요소가 된다.

이외에 oneway 라는 키워드가 사용이 가능하다. 경우에 따라서는 클라이언트로부터 서버 객체로 요청을 보낼 뿐 그 응답을 받을 필요가 없는 경우가 있는데, 이때 method 를 선언 할 때 oneway 라는 키워드를 붙여준다. 이렇게 하면 처리결과는 클라이언트로 반환되지 않는다. oneway 호출을 할 경우 일반 호출에 비해 속도가 빨라지나, 서버에게 보낸 요청이 실질적으로 받아들여졌는지는 알 수 없게 된다.
예외를 던지는 raises 라는 키워드가 있는데 이것은 예외선언에서 알아보도록 하자.

```
interface Sample
{
```

```
    oneway void oper();
};
```

● 리퀘스트 컨텍스트

CORBA 에서는 리퀘스트 컨텍스트를 이용하여 클라이언트에서 서버로 정보를 전달할 수 있으며 프로퍼티(오브젝트에 저장된 정보단위)와 값으로 이루어 진다. 각 어플리케이션은 임의의 개수의 리퀘스트 컨텍스트를 가질 수 있고, 각 컨텍스트는 임의의 개수의 프로퍼티를 저장한다. 현재까지는 프로퍼티 값으로 문자열만 사용가능하다. 아래 코드는 Name 과 Location 을 프로퍼티로 갖는 리퀘스트 컨텍스트를 사용하는 것이다.

```
interface Sample
{
        oneway void oper() context("Name", "Location");
};
```

프로퍼티명의 끝에 와일드 카드("*")을 사용할 수도 있는데, Name*라 지정했을 경우 Name 으로 시작하는 임의의 프로퍼티가 전달된다. 코드의 경우 클라이언트는 자신의 리퀘스트 컨텍스트 이름과 언어를 저장하고 오퍼레이션 호출 파라미터로 이 리퀘스트 컨텍스트의 오브젝트 레퍼런스를 건네준다. 그 결과, 리퀘스트와 함께 이 두 프로퍼티값이 서버로 전달되어 서버는 클라이언트가 보내온 컨텍스트의 값을 사용할 수 있게 된다. 이처럼 오퍼레이션의 파라미터로서 직접 값을 건네주는 방법은 적절하다고는 볼 수 없으나, 서버의 실행에 영향을 주는 환경정보를 전달할 경우 리퀘스트 컨텍스트가 사용된다.

● interface 와 struct

경우에 따라 interface 는 attribute 와 operation 을 둘 다 가지거나, 둘 중 하나만 가지거나, 둘 다 가지지 않을 수 있다. attribute 와 operation 을 둘 다 가지지 않는 interface 는 사용할 필요도 없겠지만 문법적으로 에러를 일으키지 않는다. 만약 operation 을 가지지 않는 인터페이스를 선언할 경우라면 이때는 interface 대신 struct 을 사용하도록 하자. 예를 들어 Java 프로그래밍 할 때 IDL 컴파일러는 interface 와 struct 을 모두 클래스로 변환한다. 하지만 중요한 차이가 있는데 interface 는 CORBA 객체(Object)에서 상속을 받는 클래스로 struct 는 CORBA 객체를 상속받지 않는 클래스로 변환된다. 단순히 데이터만 전달하기 위해 사용하는 것이라면 CORBA 객체를 상속 받음으로써 메모리의 낭비와 전달 속도의 감소를 가져올 수 있기 때문에 operation 을 가지지 않는다면 그냥 struct 으로 선언하도록 하자.

```
interface A
{
};

interface B
{
        attribute long l;
};

struct C
{
```

```
        long l;
};

interface D
{
        void oper( );
};

interface E
{
        attribute long l;
        void oper( );
};
```

● 예외선언

operation 호출이 실패했을 경우에는 클라이언트로 예외가 반환된다. 예외에는 표준 예외와 사용자 정의 예외 두 종류가 있다. 표준예외는 CORBA 의 사양서에서 그 예외정보 구조체가 규정되어 있는 시스템 예외이다. 그에 반해 사용자 정의 예외는 어플리케이션 개발자가 예외정보 구조체를 정의하게 된다. 사용자 예외는 사용하기에 앞서 미리 선언되어 있어야 하며, 예외를 일으킬만한 operation 은 raises 라는 키워드를 이용하여 발생할 가능성이 있는 예외를 명시해준다.

```
exception UsrException
{
};

interface Sample
{
        void oper() raises (UsrException)
};
```

예외에 대한 자세한 내용은 예외 다루기에서 참고하기 바란다.

■ 모듈(module)

IDL 에서는 모듈을 사용하여 인터페이스나 데이터의 범위(scope)를 계층화하여 분류할 수 있다. 아래의 예처럼 C 라는 인터페이스가 두개 이지만 하나는 A::B::C 라는 계층적 구조를 가지고 또 다른 하나는 D::C 라는 계층적 구조를 가진다. 선언되어 있는 인터페이스나 데이터를 사용할 경우에도 범위 연산자(::)를 사용하여 속해 있는 모듈 이름을 모두 적어 주어야 한다. 모듈은 하나의 IDL 파일 내에 복수개가 존재 할 수 있으며, 모듈 안에 모듈이 중복(nested)적으로 들어 올 수 있다. 모듈 안에 들어 들어올 수 있는 데이터 형은 모든 것이 가능한 데 주의 할 점이 있다. 인터페이스나 구조체, 공용체, 열거형등과 같은 사용자 정의형은 바로 사용이 가능하나 기본형이나 시퀀스, 문자열형은 typedef 키워드와 함께 사용해야 한다.

```
module A
{
   module  B
    {
        interface C
```

```
            {
            };
          typedef sequence<long> Number;
            typedef string<20> Name;
    };
};

module D
{
   interface C
    {
        void oper (in A::B::C arg);
    };
};
```

■ C++ 클래스와의 차이

IDL 의 인터페이스는 다음과 같은 차이점 때문에 C++의 클래스 보다는 Java 의 인터페이스에 더 가깝다. 그렇다면 C++의 클래스와 어떤 점에서 다를까?

- 생성자와 소멸자가 없다. 이것은 IDL 이 단순한 인터페이스 정의 언어의 역할만 할 뿐이고 실제 객체가 생성과는 무관하기 때문이다. 이것을 좀더 자세히 설명하면 CORBA 에서의 객체 생성이란 객체에 대한 레퍼런스만 생성하게 되는 것이고, 실제 초기화 과정은 구현객체(Object Implementation)의 실행을 통해서 이루어지기 때문에 결국 생성자와 소멸자가 필요 없게 된다.

- 함수의 다형성을 구현할 수 없다. 이것은 IDL 이 전적으로 클라이언트에게 노출하는 인터페이스만을 정의하는 언어이기 때문에 정의가 같은 operation 이 있다고 할 경우 내부 구현 코드가 다르더라도 클라이언트에서는 똑같은 인터페이스로 인식되기 때문이다.

- IDL 에서 예외는 다른 예외를 상속할 수 없다. 자세한 내용은 예외를 참고 하기 바란다.

- 연산자 오버로딩이 지원되지 않는다.

다음 예는 C++의 클래스와의 차이를 나타낸 것이다.

```
interface Difference
{
        //생성자와 소멸자를 가지지 않음
        //Difference( )
        //~Difference( )

        void oper ( );
        //void oper (long l);//함수의 다형성이 인정되지 않음
        //Difference & operator = (const Difference & a) //연산자 오버로딩이 인정되지 않음
};
```

■ 객체형과 데이터형

IDL 에서 객체형을 정의할 때 사용하는 키워드가 interface 라고 앞에서 얘기했다. 이와 달리 사용할 수 있는 데이터형이 있는데 기본 데이터형과 struct, union, enum, array, string, sequence 등이다. CORBA 에서는 객체형은 참조로 전달하며, 데이터형은 값으로 전달하는 것을 원칙으로 하고 있다. 객체형에 대한 값 전달을 CORBA3.0 에서는 정의되어 있지만, 아직 지원하지 않는다. 그 외에 서로의 차이점을 표로 정리해 놓았으니 참고하기 바란다.

항목	객체형	데이터형
키워드	interface	Simple types, struct, union, enum, array, string, sequence
인자전달	참조전달	값 전달
operation 유무	가진다.	가지지 못한다.

객체의 전달에 있어서 참조를 사용할 경우는 복사에 따른 시간과 메모리의 혜택을 볼 수 있는 반면, 호출 당시의 객체의 상태를 전달할 수 없는 단점을 가지게 된다.

■ 데이터형

● 기본 데이터 형

IDL 에서 사용하는 기본 데이터형은 C, C++, Java 에서 사용하던 기본 데이터형과 유사하다.

1)정수형 및 부동소수점형
정수형과 부동소수점형은 이식성을 보증하기 위해 다양한 데이터형이 규정되어 있으며, 정수형은 int 라는 키워드를 사용하지 않기 때문에 IDL 파일을 작성할 때 주의해야 한다.

```
//정수형
long l;
long long ll;
short s;
unsigned long ul;
unsigned long long ull;
unsigned short us;

//부동소수점형
float f;
double d;
long double ld;
```

2)문자형(char, wchar)
CORBA 2.0 에서는 멀티바이트 문자에 관한 고려가 아직 되어 있지 않기 때문에 멀티바이트 문자 전송을 위해서는 문자형의 배열 혹은 octet 형을 사용하면 된다.
wchar 은 유니코드(unicode)를 위한 문자형인데 특히 Java 로 구현된 객체와 C++로 구현된 객체간에 통신을 할 경우 유니코드와 Latin-1 코드 체계(대부분의 시스템에서 문자형 및 문자열형은 주로 ISO Latin-1 을 가정하고 있다)와의 변환 문제가 발생할 경우 사용하면 된다.

```
//문자형
```

```
char c;
wchar wc;
```

3) 부울리언형(boolean)
C 나 C++, Java 에서 사용하는 것과 똑같은 데이터 형이다.

```
//부울리언형
boolean b;
```

4)옥테트형(octet)
옥테트형은 파일을 전송할 경우나 문자열 전송과 같이 함수의 인자로 가공되지 않은 데이터를 그대로 주고 받을 때 사용하게 된다.

```
//옥테트형
octet o;
```

5)any 형
any 형은 C++에서 사용하던 void* 형 정도로 생각하면 되는데, any 형은 void* 형과 달리 실제로 전송되는 타입에 대한 정보를 포함하고 있다.

```
//any 형
any a;
```

Data Type	키워드	설명
정수형	long long long short unsigned long unsigned long long unsigned short	부호가 있는 정수. -2^{31}~2^{31}-1 부호가 있는 정수. -2^{63}~2^{63}-1 부호가 있는 정수. -2^{15}~2^{15}-1 부호가 없는 정수. 0~2^{32}-1 부호가 없는 정수. 0~2^{64}-1 부호가 없는 정수. 0~2^{16}-1
부동소수점형	float double long double	IEEE 단정도 부동소수점 IEEE 배정도 부동소수점 IEEE 확장배정도 부동소수점
문자형	char wchar	ISO Latin-1 문자 ISO Unicode 문자
부울리언형	boolean	TRUE 또는 FALSE
옥테트형	octet	전송 중에 변환되지 않는 8 비트 데이터
any 형	any	임의의 데이터형

6) 기본형 사용 예

다음은 기본형을 사용하고 있는 예이다.

```
interface SimpleType
{
        //정수형
        attribute long l;
        attribute long long ll;
        attribute short s;
```

```
        attribute unsigned long ul;
        attribute unsigned long long ull;
        attribute unsigned short us;

        //부동소수점형
        attribute float f;
        attribute double d;
        attribute long double ld;

        //문자형
        attribute char c;
        attribute wchar wc;

        //부울리언형
        attribute boolean b;

        //옥테트형
        attribute octet o;

        //any 형
        attribute any a;
};
```

● 복합형

Data Type	키워드	설명
구조체	struct	요소를 갖는 데이터형
공용체	union	변별자 값에 응한 데이터형이 선택된다
열거형	enum	순서가 부여된 식별 리스트
시퀀스	sequence	일차원 배열
문자열	string wstring	8 비트 문자로 이루어지는 문자열 unicode 지원 문자열

1) 구조체(struct)

구조체는 임의의 데이터형이 하나 이상의 데이터를 멤버로 갖는 데이터형이다. C나 C++의 구조체와 같은 구문(Syntax)을 가진다.

```
//구조체
struct Point
{
        double x;
        double y;
};
```

2) 공용체(union)

공용체는 다른 데이터형 데이터를 하나의 파라미터로 송수신할 경우에 사용되는 데이터 구조이다. IDL 에서 공용체를 사용하는 문법은 C 또는 C++의 공용체와 switch 문을 함께 갖고 있다. 실행시간에 switch(short)로 지정된 short 형 변별자(discriminator) 값에 맞는 데이터 형태로 해석이 되게 된다.

```
//공용체
```

```
union DataItem switch(char)
{
        case ‘c’: long count;
        case ‘a’: double amount;
        default: char initial;
};
```

3)**열거형**(enum)
열거형은 순서가 정해진 식별자 리스트이다. 이 형에 속하는 데이터는 실행 시에 리스트 중의 하나의 값을 취한다.

```
//열거형
enum Grade {A, B, C, D};
```

4)**시퀀스형**(sequence)
시퀀스는 지정된 데이터형의 일차원 가변길이 배열이다. 데이터의 최대 크기를 지정할 수도 있고 지정하지 않을 수도 있다. 최대 크기를 지정한 시퀀스형을 bounded sequence **라고 하고 크기를 지정하지 않은 시퀀스형을** unbounded sequence **라 부른다.** bounded **의 의미는** stub **과** skeleton **이 서로 통신할 때, 주고 받을 데이터의 최대값이 얼마이다라는 정보일 뿐이지 데이터의 크기가 고정되어 있다고 생각해서는 안 된다. 어느 경우라도 실제로 송수신된 데이터 크기는 실행 할 때 결정되므로 가변길이 데이터를 효과 있게 전송할 수 있다.**

```
//시퀀스형
sequence<long> Score1;
sequence<unsigned long, 10> Score2;
```

5)**문자열형**(string, wstring)
8 **비트 길이의 데이터로 이루어지는 문자열이다.** string **형은 주로 서유럽 언어의 문자를 수용하는** ISO Latin-1 **을 사용하며,** wstring **형은 유니코드를 사용한다. 시퀀스형과 마찬가지로 최대 크기를 지정할 수 있으며 최대 크기를 지정한 문자열을** bounded **문자열, 최대 크기를 지정하지 않은 문자열을** unbounded **문자열이라 한다. 그렇다면 앞에서 배운** sequence<char>**와** string **은 어떤 차이가 있을까? 한마디로 말해 기존 언어에서 사용하던 문자열**(null-byte **포함)과 비슷한 것이** string **이며** sequence<char>**은 사용자가 직접 처리를 해주어야 한다. 따라서,** sequence<char>**보다는** string **사용을 적극 추천한다.**

```
//문자열형
string Name1;
string<10> Name2;
wstring Name3;
wstring<10> Name4;
```

6)**배열**
IDL **에서는 일차원 가변길이 배열 시퀀스에 대해 다차원 고정길이의 배열도 사용할 수 있다. 문법적으로** C **나** C++**에서 사용하던 방식과 똑 같은데, 배열의 모든 데이터가 네트워크 상에서 전송되기 때문에 성능상 주의가 필요하다.**

```
//배열형
char Course1[10];
```

```
char Course2[10][10];
```

7) 복합형 사용 예

아래 복합형을 사용하고 있는 예를 들어 놓았다.

```
module ComplexType
{
        //구조체
        struct Point
        {
                double x;
                double y;
        };


        //공용체
        union DataItem switch(char)
        {
                case 'c': long count;
                case 'a': double amount;
                default: char initial;
        };


        //열거형
        enum Grade {A, B, C, D};


        //시퀀스형
        typedef sequence<long> Score1;
        typedef sequence<unsigned long, 10> Score2;

        //문자열형
        typedef string Name1;
        typedef string<10> Name2;
        typedef wstring Name3;
        typedef wstring<10> Name4;

        //배열형
        typedef char Course1[10];
        typedef char Course2[10][10];
};
```

■ 리포지터리 ID

인터페이스 리포지터리에 저장되는 정보를 식별하기 위해 리포지터리 ID 를 사용한다. OMG 에서 정의하고 있는 리포지터리 ID 는 다음의 세가지 컴포넌트를 콜론으로 구분한다.

포맷명(IDL):포맷에 따른 문자열:버전번호

문자열은 프리픽스(옵션) 및 스코프환된 이름의 각 요소를 슬래시(/)로 구분하여 표

기한다.

CORBA2.0 에서는 다음에 설명하는 3 개의 포맷이 정의되고 있다. 리포지터리 ID 는 어플리케이션 개발자가 IDL 중에 직접 기술할 수 있고, 툴이 자동적으로 생성하고 IDL 정의와 함께 리포지터리에 저장된 ORB 제품도 있다.

```
module Enterprise
{
        #pragma prefix “Corporate-A_IDL”
        module Bank
        {
                interface Account
                {
                        //생략
                };
                #pragma version Account 1.2
        };
};
```

위와 같은 IDL 정의가 있다고 할 때 리포지터리 ID 는 다음과 같다.

```
IDL:Corporate-A_IDL/Bank/Account:1.2
```