



AGETOR[®]

Interface Definition Language

Table of contents

1	IDL - Interface Definition Language	3
1.1	Overall	3
1.1.1	Language mapping.....	3
1.1.2	Pass by value semantics	3
1.2	Types in AIDL	4
1.2.1	Simple types	4
1.2.2	Structures	4
1.2.3	Sequences	5
1.2.4	Byte arrays	5
1.3	IDL specifications.....	5
1.3.1	Modules.....	6
1.3.2	Structures	6
1.3.3	Sequences	7
1.3.4	Interfaces.....	7
1.3.5	Methods.....	8
1.3.6	Including IDL	9
1.4	Code generation	10
1.5	Java mapping rules.....	10
1.5.1	Mapping of types to Java.....	11

1 IDL - Interface Definition Language

This section describes how to specify client/service interfaces in the IDL (see also [Interface Definition Language](#)).

1.1 Overall

The overall idea behind IDL is to use remote procedure calls (RPC)¹ as the abstraction for the actual message based communication between a client and a service. The output parameters from a procedure call makes up the reply to a specific request. Procedure calls are synchronous, so remote procedure calls will block the caller until the reply is available. However in contrast to local procedure calls, network errors or machine crashes may prevent a service from responding, so remote procedure calls will usually timeout after a pre-specified period.

By defining the interface in IDL the service and client side will not directly interact with the message protocol format. Instead they will use IDL generated skeleton and stub code that per definition match each other since they are both generated from the same IDL. Hence protocol mismatches should be impossible.

1.1.1 Language mapping

AGETOR is based on a subset of IDL called AGETOR IDL (AIDL). Some features were removed from standard IDL to ease the mapping to different host languages.

Clients and services can be written in many different host languages which all map to the same IDL specification so procedure calls can be transmitted between these different languages. The mapping from IDL to any host language is defined by mapping rules, specifying which language types corresponds to IDL types and how IDL structs, sequences, interfaces and methods are to be represented in the host language.

1.1.2 Pass by value semantics

When invoking AIDL based remote procedures all parameters are passed by value. IDL is non object oriented hence no objects are specified, but instead an interface containing a set of procedures/methods are defined. A service will implement one or more interfaces by declaring the code necessary to process the request.


All input and output parameters to a method are passed by value in contrast to passing parameters by reference. This means that once the invocation request gets to the service all input parameters are available locally. Likewise once the invocation response reaches the client all output parameters are also available local to the client.

¹ This document use both Remote Procedure Calls and Remote Method Invocation which are synonymous.

1.2 Types in AIDL

IDL contains a number of simple types and allows you to define structures (i.e. record types) of these simple types and other user defined types. Sequences may be defined from structures representing unbounded repetitions of a structure type.

Both simple types, structures and sequences are referred to as types and may be used anywhere a type is valid.

 Note that IDL is case-sensitive and hence all type, interface and method names must be written in the correct case.

1.2.1 Simple types

AIDL defines the following simple data types.

AIDL types	
boolean	Holds a true or false value.
char (8 bit)	Holds a single character.
octet (8 bit)	Holds an 8 bit integer (byte).
string<size>	Holds a string of characters. May be bounded with <size>.
short (16 bit)	Holds a 16 bit integer.
long (32 bit)	Holds a 32 bit integer.
float (32 bit)	Holds a single precision floating point number.
double (64 bit)	Holds a double precision floating point number.
Date	Holds a date and time. Note that this type is not standard IDL.

1.2.2 Structures

IDL supports definition of structs resembling C structs, declaring a data entity, consisting of several data types.

The syntax for declaring structures are:

```
struct struct-name {  
    type field-name;  
    type field-name;  
    ...  
};
```

The `struct-name` must be a unique type name within this IDL specification. We recommend that the `struct-name` starts with a capital letter to clearly separate simple types from user defined

types. Note also that the non standard Date type is named with a capital 'D' as if it were a structure. The `type` may be any simple type, structure or sequence. The `field-name` must be unique within this structure definition.

1.2.3 Sequences

IDL supports definition of sequences which are ordered collections of structures.

The syntax for defining sequences:

```
typedef sequence <struct-type> sequence-name;
```

The `struct-type` must be a structure type, so you can not directly define sequences of simple types or other sequences. However by wrapping a simple type or sequence type in a structure this is indirectly possible. The `struct-type` must refer to a defined structure in the IDL specification, even though the structure is defined later than the sequence. This allows you to define structures containing sequences of the structure itself.

⚠ Types may refer directly or indirectly to themselves, but instances of these types should never refer to themselves. Creating cyclic data structures and passing these as parameters to method invocations will give unpredictable results.

The `sequence-name` must be a unique type name within this IDL specification.

1.2.4 Byte arrays

To allow the transfer of raw data a byte array type is also supported. It uses the generic array construct in IDL to define the array type and size. Note however that only octet is allowed even though the compiler accepts arrays of other types.

The syntax for defining byte arrays are:

```
typedef octet type-name[size];
```

This is mapped to the Java construction `byte[]` and only the actual size of a Java input byte array are transferred.

1.3 IDL specifications

Each IDL file consists of a module, for the current project. The IDL file must have the following template :

```
module module-name {
    struct struct-name {
        type field-name;
        type field-name;
        ...
    };
    ...
    typedef sequence <struct-type> sequence-name;
    ...
    interface interface-name #env="..." #qno=... {
        return-type method-name (
            in type parameter-name,
            out type parameter-name,
            inout type parameter-name
            ...
        ) #qno=...;
        ...
    };
};
#include "idl-file";
```

1.3.1 Modules

Within the module, simple types, structures, sequences as well as interfaces are defined. The interface holds the method declarations, that will be invoked remotely. Comments are very much recommended (although not enforced by the compiler) since this IDL is the contract between the front-end and backend programmer.

We recommend the Javadoc style for these comments.

```
/**
 * Module test contains an IDL test.
 */
module test {
    ...
};
```

1.3.2 Structures

Structures are defined within a module, and will be available to other types in the same module or in other IDL files that include this module.

The following example defines a structure named `UserObject2` containing only the integer `number2`.

```
/**
 * UserObject2 is a userdefined object.
 */
struct UserObject2 {
    /** 16-bit integer value. */
    short number2;
};
```

Below a structure named `UserObject`, consisting of a string, an integer, a date, a structure of the previously defined type `UserObject2` and a sequence of type `UserObjects2`.

```
/**
 * UserObject holds other objects/sequences.
 */
struct UserObject {
    /** String value. */
    string str;
    /** Integer value. */
    short tal;
    /** Date. */
    Date dato;
    /** An object of type UserObject2. */
    UserObject2 obj;
    /** A sequence of UserObject2. */
    UserObjects2 objs;
};
```

1.3.3 Sequences

The following example defines a sequence consisting of structures of type `UserObject2`:

```
/**
 * Sequence of UserObject2.
 */
typedef sequence <UserObject2> UserObjects2;
```

The next example defines a sequence named `UserObjects` consisting of `UserObject` structures. This indirectly creates a sequence of sequences, since `UserObject` contains a sequence of type `UserObject2`.

```
/**
 * Sequence of UserObject.
 */
typedef sequence <UserObject> UserObjects;
```

1.3.4 Interfaces

Interfaces are collections of methods grouping some logical entity that a service will implement.

The following example defines an interface named `TestService`:

```
/**
 * TestService implements a test method.
```

```

*/
interface TestService #env="test" #qno=701 {
    ...
};

```

The interface name an environment (#env) must be defined, but the question number (#qno) is optional. The environment corresponds to the environment in which the service is configured in the Broker (see document IRE_guide). The question number corresponds to the question number defined in the Broker configuration. Defining the question number for the entire interface will map all methods in this interface to this question number, meaning that all methods will have the same permissions. If individual access control is required, the question number can be defined explicitly on some methods, and will then override the interface question number.

1.3.5 Methods

Methods are defined in interfaces and the syntax much resembles C and Java. Methods may be functions with a return value of any type (both simple, structure or sequence).

Parameters are passed by value and are associated with an attribute indicating which way they are transferred.

Direction attribute

in	Parameter is passed from the client to the service when the method is invoked.
out	Parameter is returned from the service to the client once the method was executed.
inout	Parameters are transferred both ways.

Methods may optionally be associated with a question number which overrides the question number given to the interface. The Broker enforces permission checks on question numbers and uses the environment and question number for service identification.

```

/**
 * method returns nothing of significance.
 */
short method(
    /** Input integer value a. */
    in short a,
    /** Input string value b. */
    in string b,
    /** Output date c. */
    out Date c,
    /** Input/Output UserObject d. */
    inout UserObject d,
    /** Input/Output sequence of UserObjects. */
    inout UserObjects e
) #qno=702;

```

1.3.6 Including IDL

It is possible to include definitions from an external IDL file, by using the `#include` compiler directive:

```
#include "idl-file";
```

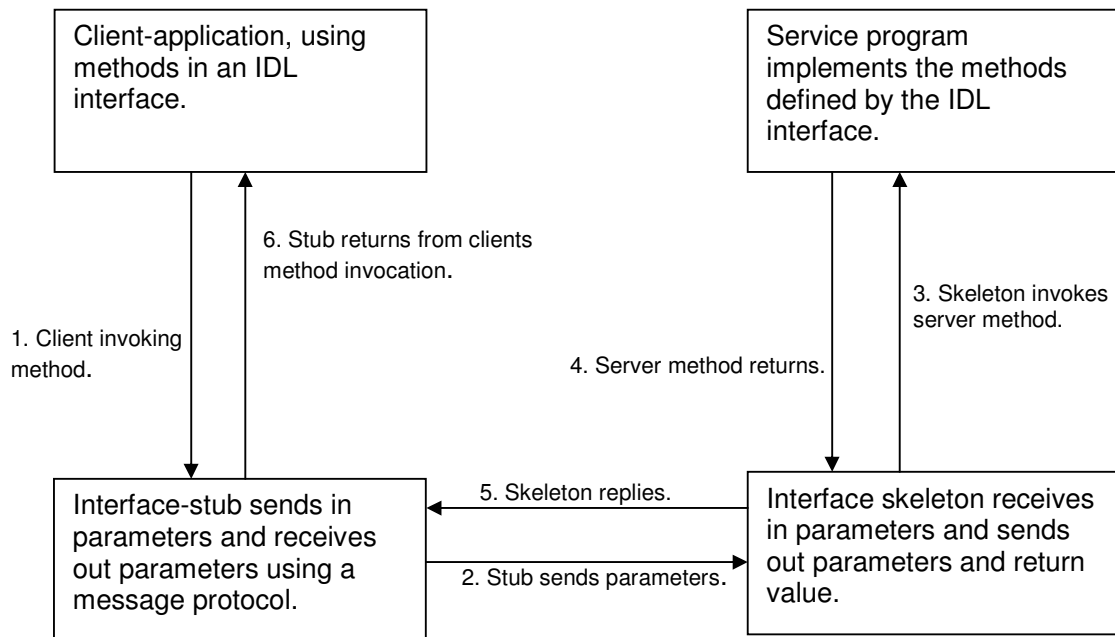
The above definition includes the definition from the IDL file named `idl-file` added the `.idl` extension located in the `$AGETOR_HOME/app/idl` directory. The types declared in this file will also be available in the current IDL. When compiling the IDL and generating code all types and interfaces in the referenced IDL files are processed.

1.4 Code generation

Code for both the server and client side is generated from the IDL definition.

For the client, a **stub** is generated, which marshal in parameters to a message, sends this message, receives a reply message from the server and demarshals out parameters from this message.

For the server, a **skeleton** is generated, which receives a request message demarshals in parameters from a client, invokes the local method, marshals out parameters to a reply message and sends this to the client.



1.5 Java mapping rules

This section describes how to access IDL defined types, structures and operations defined in Java.

1.5.1 Mapping of types to Java

Java does not support pass-by-variable parameters, so out and inout parameters to IDL generated functions must be passed through a holder object. Otherwise the server programs changes to these variables cannot be passed back to the client.

IDL-type	Java-type (in parameter or struct member)	Java-type (out/inout parameter)
boolean	boolean	BooleanHolder
char (8 bit)	char	CharHolder
octet (8 bit)	byte	ByteHolder
string (af char)	String	StringHolder
short (16 bit)	int	IntHolder
long (32 bit)	int	IntHolder
float (32 bit)	float	FloatHolder
double (64 bit)	double	DoubleHolder

All holder objects contains a `type get ()` method, returning the contents of the holder object as the appropriate type. As well a `set (type)` method is provided for changing the contents of the holder object. These objects belong to the `dk.bordings.inside.orb` package.

The IDL type sequence is (default) mapped to the standard java Vector type (in the `java.util` package). Setting the general property `agetor.tools.idl2java.typedsequences` to true, will cause the `idl2java` tool to generate a separate typed sequence class for each IDL sequence².

The predefined type Date is mapped to the standard java Date type (`java.util` package). Any type `type` is mapped to a java class named `type` with a constructor and all members declared public, i.e. directly accessible.

² This feature is available from ADK 1.2