

# Reference Card material with additional explanations

## *Contents*

[Fundamental concepts](#)

[Basic syntax](#)

[Characters used](#)

[Names](#)

[Module names](#)

[Type reference names](#)

[Identifiers](#)

[Value reference names](#)

[Multi-word names](#)

[Comments](#)

[Module boiler-plate](#)

[Type and value assignments](#)

[More type definition examples](#)

[And yet more type definitions .....](#)

[Export and import clauses](#)

[Example Object Identifier Values](#)

[Constraining types](#)

[ASN.1 as an XML schema definition](#)

[Extensibility](#)

[Other syntax](#)

[Definitions for information objects](#)

[Parameterisation](#)

[Encodings](#)

[Example of a complete specification](#)

## *Fundamental concepts*

ASN.1 is primarily concerned with the definition of **types** (data structures) whose values are encoded into binary or XML format to form the messages used in computer communication.

As with most languages for defining data structures, there are a number of primitive data-types defined, plus construction mechanisms to produce more complex data structures.

These more complex data structures are named with **type references** in a <type assignment> statement, and can then be used as if they were primitive data-types in the definition of more complex structures, and ultimately in the definition of the data structures forming the messages used in a specific application.

ASN.1 also provides a **value notation** for specifying the values of any data-type that can be defined in ASN.1. This notation is mainly used to specify that if certain elements of a data structure are omitted in an instance of communication, a default value is assumed for them.

These defined values can be assigned a **value reference name** in a <value assignment statement>.

ASN.1 type and value assignment statements are grouped together into **ASN.1 modules**. (In rough terms, an ASN.1 module corresponds to an XML namespace.) In the simplest case, any type (or value) referenced in a module has to be defined in that module, but EXPORT/IMPORT mechanisms are available to allow types defined in one module to be used in a different module.

A complete ASN.1 specification could in theory consist of a single module, but in practice use of several modules is common.

## ***Basic syntax***

### ***Characters used***

Elements of the basic syntax are composed of characters from the ASCII character set. Characters in comment and in character string values can in theory include any character in the world, but most ASN.1 tools require input to the tool to be ASCII-encoded, and hence the character set for ASN.1 specifications is effectively ASCII.

NOTE: The ASN.1 specification does permit national language characters in reference names and identifiers in national variants of the ASN.1 specifications, but this is rarely activated.

### ***Names***

Names consist of arbitrarily many letters (case-sensitive), digits, and hyphen. Two names are different no matter how long, and no matter how far into the name they first differ. Apart from names of primitive types used to form XML tags, names do not and cannot contain underscore.

ASN.1 tools exist that map ASN.1 type definitions into C, C++ and Java datastructures and classes. These tools commonly replace hyphen with underscore, and will disambiguate names within the first 32 characters.

ASN.1 restricts the case (upper-case or lower-case) of the initial letter of all names used in the syntax. For example, all type reference names are required to start with an upper-case letter and all value reference names are required to start with a lower-case letter.

Historically, these restrictions were necessary to avoid ambiguity, but that is less true today. However, the rules remain, and a writer of ASN.1 must strictly adhere to the rules concerning the case of the first letter of a name.

Subsequent letters can in almost all cases be either upper-case or lower-case, but two names that are identical apart from the use of case are still considered to be distinct names - in that sense, case is significant throughout the name.

### ***Module names***

ASN.1 specifications consist of one or more modules. A module is typically published as a contiguous piece of specification (and tools normally require this).

Modules can reference and use types from other modules.

Module names have to start with an upper case letter.

### *Type reference names*

Type reference names have to start with an upper case letter.

### *Identifiers*

The names of the fields in a record structure (ASN.1 calls them the **identifiers in a sequence type**), or the names of alternatives in a choice type are called identifiers.

It is also possible to

- Name the bits in a bit-string (to make value notation more human- readable).
- Name integer values in an integer type.
- Name the enumerations in an enumerated type.

These names are also called **identifiers**.

All identifiers have to start with a lower-case letter.

### *Value reference names*

Value reference names have to start with a lower-case letter.

### *Multi-word names*

It is common to use either or both of hyphens or upper case within names to separate parts of the name. (e.g. My-type or MyType or My-Type).

The choice of these approaches is entirely a matter of style, and all styles occur, often even within the same specification.

### *Comments*

There are two forms of comment in ASN.1, single line comments and block comments.

Single-line comments start with -- (a pair of hyphens) and end with -- or new line.

Block comments start with /\* and end with \*/. Block comments can contain other block comments and/or double-hyphen comments.

Block comments were only added at the end of the 1990s, so many specifications use multiple

single-line comments instead of block comments.

It is important to note that if single line comments are used in this way, a pair of hyphens must appear at the start of every line of comment, as a new line (as well as a pair of hyphens) terminates single line comment.

## *Module boiler-plate*

An ASN.1 module has the following form:

```
MY-MODULE { <oid> }
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN
    EXPORTS <exports clause>;
    IMPORTS <import clause>;
    <Type and value assignments>
END
```

The items enclosed in angle brackets are covered by later clauses. Note that the angle brackets are **not** part of the notation! Note also that the EXPORTS and IMPORTS clauses must be in that order (if both are present), and are both terminated by a semi-colon. (These are the only ASN.1 statements that need a semi-colon terminator.)

The object identifier value (<oid>) and its surrounding curly braces can be omitted, but it is normal today for it to be included in any published specification.

The inclusion of "AUTOMATIC TAGS" is a piece of magic. Specifications that do not include this will normally contain "IMPLICIT TAGS" (or perhaps even nothing at all), and will then contain quite a lot of annotations of type definition looking like "[0]". These are "tags", but an explanation of them is outside the scope of even this explanatory material.

There is an example below of a [complete specification](#) consisting of four complete module definitions, each with an object identifier value. These modules use mainly the illustrations used in the reference card, with some additional definitions in order to produce complete modules.

## *Type and value assignments*

The general form of a type assignment statement is:

```
<TypeReferenceName> ::= <TypeDefinition>
```

For example:

```
Age           ::= INTEGER
Country-name  ::= UTF8String
Greeting      ::= UTF8String
Hex-string    ::= OCTET STRING
```

There are many different character string types in ASN.1, but the best to use today is UTF8String, which enables all Unicode (ISO 10646) characters, whether in the Basic Multilingual Plane or not, to be present, with an encoding which represents ASCII characters as ASCII codes, and other characters with a progressively more verbose encoding.

The general form of a value reference statement is:

```
<valueReferenceName> <TypeOfValue> ::= <ValueNotation>
```

For example:

```
twenty-one-today Age ::= 21
spain Country-name   ::= "ESPAÑA"
when GeneralizedTime ::= "200208192349.57894Z"
large-prime INTEGER  ::= 1999999973
```

These are normally read as, for example, "twenty-one-today of type Age is 21".

Note the presence of the N-tilde character in the Country-name, illustrating the ability to include any Unicode (ISO 10646) character.

The GeneralizedTime value is for the 19th of August 2002, at 11.49 and 57.894 seconds, with the Z indicating GMT. This is in fact the form used in encodings as well.

It is also possible to define a value using an XML format for the value notation. Value reference names defined in this way are interchangeable with value reference names defined with the older notation.

The general form of an XML value assignment is:

```
<XMLvalueReferenceName> ::= <XMLValueNotation>
```

and examples are:

```
sixty-five-today ::= <Age>65</Age>
my-octets        ::= <Hex-string>89aef764AEF</Hex-string>
plain-greeting   ::= <Greeting>Hello World!</Greeting>
bells-and-whistles-greeting ::=
  <Greeting>
    <bel/><stx/>Hello World!<etx/>
  </Greeting>
```

Note that both upper and lower case letters can be used to represent hex digits in XML value notation (for normal value notation, only upper- case is allowed).

Note also that the XML specification prohibits the inclusion in a document of Unicode (ISO

10646) characters such as BEL, ETX, and STX, even using the & escape mechanism. ASN.1 therefore provides empty content tags to allow these characters to be represented in XML value notation, but more importantly as the encoding of the value of a UTF8String in an XML document.

## *More type definition examples*

This illustrates some of the primitive types, and the use of simple value notation in DEFAULTS:

```
My-sequence ::= SEQUENCE {
    first    BOOLEAN,
    second   INTEGER OPTIONAL,
    third    INTEGER DEFAULT 129,
    fourth   BOOLEAN DEFAULT TRUE,
    fifth    REAL    DEFAULT 0.629,
            -- Or DEFAULT 62.9E-2
    sixth    UTF8String DEFAULT

            -- Unicode characters. (If your browser does not display
            -- the above value as Unicode characters, check your browser
            -- is configured for Unicode (UTF8).
    seventh IA5String  DEFAULT "James Morrison",
            -- ASCII characters
    eighth   BIT STRING  DEFAULT '101100011'B,
    ninth    OCTET STRING DEFAULT '89AEF764'H,
    tenth    Alternatives }
```

The use of OPTIONAL means that in an encoding, the value can be omitted. The use of DEFAULT means that it can be omitted, but if it is then the DEFAULT value is being represented.

There are many other notations for REAL values, but these are not much used, and are not discussed further.

The UTF8String value illustrates the inclusion of non-ASCII characters. The IA5String type (International Alphabet No 5 - effectively ASCII) is often used (particularly in older specifications) for a string which is restricted to ASCII characters.

Note the use of the single quotation marks for binary and hexadecimal notation.

The tenth field illustrates the use of a forward type reference, with Alternatives being defined below:

```
Alternatives ::= CHOICE {
    first-alternative    TypeA,
    second-alternative   TypeB,
    third-alternative    NULL }
```

and finally we illustrate use of a sequence-of construction, with the number of repetitions restricted to being at least 28 and at most 31:

```
DailyMaxTemperaturesForMonth ::=
```

```
SEQUENCE (SIZE(28..31)) OF temperature INTEGER
```

Notice the presence of the optional identifier "temperature". This is frequently omitted, and is mainly of use when XML encodings are intended to be used, as it gives an easy means of controlling the XML tag used for the components.

## *And yet more type definitions .....*

Here we have some slightly more complex definitions:

```
VersionsSupported ::= BIT STRING {  
    version1 (0),  
    version2 (1),  
    version3 (2) }
```

This is a bit-string with named bits, commonly used to transmit a bit- map indicating which versions are supported, with bit 0 (the left-most in the value notation) representing version 1, etc. The names are commonly used in value notation such as "{version2, version3}", meaning that the version 2 and the version3 bits are set (to 1), and all other bits are unset.

It might be used like this:

```
Message ::= SEQUENCE { ..... ,  
    version-bit-map VersionsSupported  
    DEFAULT {version1} }
```

Here is an example of an integer with some distinguished values and a range constraint:

```
Color ::= INTEGER {  
    red(10), orange(20), yellow(30), green(40),  
    blue(50), indigo(60), violet(70) } (0..80)
```

Here the type Color can take any value in the range zero to 80, but names have been assigned for the special values 10, 20, etc. This is really just a short-hand for a series of value assignments for these names. The names can then be used as DEFAULT values.

Finally, we show an ENUMERATED type.

```
Codes ::= ENUMERATED {code1(0), code2(1), code3(2)}
```

The actual numbers are the ones that will be used to represent the enumerations in an encoding. They were required in earlier versions of the ASN.1 standards, but can (and probably should) now be omitted in new specifications.

## *Export and import clauses*

This is an export clause:

```
EXPORTS TypeA, TypeB, valueC ;  
    -- Note the semi-colon
```

Here MODULE-A defines and exports TypeA, TypeB, and valueC. We assume there is also a MODULE-B that defines and exports TypeD and TypeE. These five types can then be imported into MODULE-C:

```
IMPORTS TypeA, TypeB, valueC FROM
    MODULE-A { ..... }
TypeD, TypeE FROM
    MODULE-B { ..... } ;
```

The two ..... represent (solely for the purposes of the Reference Card) object identifier values which unambiguously identify MODULE-A and MODULE-B. Export and import clauses are illustrated in the [complete specification](#), but the snippets above have not been used in that example.

## *Example Object Identifier Values*

There are many forms for object identifiers. Allocations of object identifier values are made on a hierarchical basis. All components have associated numbers, which are all that are transmitted in an encoding. The names indicate the use of a particular component, and are just for human readability. If a number alone is used in the value notation, the round brackets around the number are also omitted.

The first example is commonly used for the object identifier of an ASN.1 module:

```
oid1 OBJECT IDENTIFIER ::=
    {iso standard 2345 modules (0) basic-types (1)}
```

The numbers corresponding to "iso" and "standard" (1 and zero respectively) are specified in the ASN.1 standard, and are almost always omitted. Thus "iso standard 2345" provides object identifier namespace for that standard, maintained by the writers of the standard. In this case they have used the next component to identify the set of all their modules, with the following component being unique to a particular module.

Another very common first two components would be:

```
oid2 OBJECT IDENTIFIER ::= {joint-iso-itu-t ds(5)}
```

Finally, we can use a defined value for an object identifier as the first part of another object identifier. This is very commonly done in published specifications to avoid repeating the early parts of object identifier values. It is also possible to omit the names completely. These points are illustrated below:

```
oid3 OBJECT IDENTIFIER ::= { oid2 modules(0) }
oid4 OBJECT IDENTIFIER ::= { oid3 basic-types(1) }
oid5 OBJECT IDENTIFIER ::= { 2 5 0 1 } -- equals oid4
```

## *Constraining types*

We have already seen examples of a range constraint on an integer, and a size constraint on the number of iterations of a sequence-of.

ASN.1 has a very powerful set of mechanisms for constraining the values of a type (defining a new sub-type). Some of these are illustrated below:

```
INTEGER (0..MAX)           -- only non-negative values
INTEGER (-6..3 | 10..30)   -- only -6 to 3 or 10-30
INTEGER (ALL EXCEPT 0)  -- 0 not allowed
SEQUENCE (SIZE (0..10)) OF INTEGER
IA5String (SIZE (1..25)) (FROM ("A" .. "Z"))
    -- Only sizes 1-25 and characters "A"- "Z" allowed
```

The above have applied constraints to primitive types and constructors, but they can equally be applied to use of a type reference, either as a component of a sequence or in a new type assignment statement.

You can specify just about any desired subset of the values of any ASN.1 type using the constraint notation.

The following four examples are more complex constraints, and are included here simply so that they can be recognised if they are present in a published specification:

```
OCTET STRING (CONTAINING My-Type ENCODED BY perBasicAligned)
    -- perBasicAligned is imported from the ASN.1 standards
```

This constraint enables the contents of an OCTET STRING to be specified as the values of some ASN.1 type, encoded by a specific encoding rule (which need not be the same as that used for the encoding of the carrier protocol. This is one mechanism (but only one) that supports the traditional protocol layering architecture. In this case, the type used to provide the values of the octet string, and the encoding of its values, is statically specified. Other mechanisms allow either or both of these to be determined in an instance of communication.

The following constraint allows a UTF8String (or any other character string type) to be constrained to have a particular pattern - in this case, four digits, a hyphen, two digits, a hyphen, and a further two digits, presumably intended to represent a date as yyyy-mm-dd:

```
UTF8String (PATTERN "\d#4-\d#2-\d#2")
```

The string following the word PATTERN is called a "regular expression", following Unix terminology. A detailed treatment of this is outside the scope of the Reference Card and this Web page.

The next illustration is of a "user-defined constraint". In this case the "....." will contain a comment saying what the constraint is. Typically this will be an encryption or hash or digital signature of the value of some type. This construction is almost always associated with parameterisation, so that the actual type being encrypted can be supplied as an actual

parameter (see later). The syntax is:

```
BIT STRING (CONSTRAINED BY ..... )
```

Finally, we illustrate a sequence in which constraints are applied to individual components. This has two main uses. The first is to express constraints such as "There are three optional elements, but at least one of them must be present." (See the [complete specification](#).) The second is to define a "basic class" for a complete protocol. The syntax is:

```
SEQUENCE {.....} (WITH COMPONENTS .....)
```

## *ASN.1 as an XML schema definition*

ASN.1 is supported by XML Encoding Rules, which provide a well-formed XML document as a way of transferring values of an ASN.1 type. This makes ASN.1 a schema notation for XML.

The Reference Card illustrates only the default XML which is produced to encode the ASN.1 values, but further notation is available to (for example) specify that some components are to be carried as XML attributes, or that the values of a sequence-of are to be carried as a space-separated list. ASN.1 has similar power to the more common (but more obscure!) use of XSD for the specification of XML documents.

Here is a simple ASN.1-defined message:

```
Message ::= SEQUENCE {
    sender-id    OBJECT IDENTIFIER,
    urgency      ENUMERATED { high, normal, low },
    actions      SEQUENCE OF CHOICE {
        insert   InsertionDetails,
        remove   RemovalDetails,
        update   UpdateDetails},
    names        SEQUENCE OF name UTF8String,
    confirm      BOOLEAN }

```

This can be used to validate the following XML document:

```
xml-document ::=
<Message>
  <sender-id>2.39.6.45</sender-id>
  <urgency><normal/></urgency>
  <actions>
    <remove>.....</remove>
    <insert>.....</insert>
    <insert>.....</insert>
  </actions>
  <names>
    <name>.....</name>
    <name>.....</name>
  </names>
  <confirm><true/></confirm>
</Message>

```

As before, the "....." is used in the Reference Card to represent material that is not important

for the illustration. In the [complete specification](#), we use a simple UTF8String, but in a real case the "....." would typically be arbitrarily complex XML elements.

Note the dot-separated list that is used for an object identifier value in the XML encoding (and in XML value notation).

## *Extensibility*

You will frequently encounter the ellipsis ("...") in published specifications. This signifies that, at that point in the specification, extensions are likely to be added in a future version of the message.

This does two things: It affects the encoding (particularly the efficient binary encodings), in such a way that version 1 systems are easily able to skip added version 2 material if they are sent a version 2 message; secondly, it tells a version 1 system that it should not diagnose an error if there is added material at this point. (The precise action can also be indicated with an exception marker - an exclamation mark - but this is not much used).

The example below shows this version 1 message enhanced in versions 2, 4, and 6 of the complete specification. (It is assumed that other parts of the specification - perhaps the detailed definition of "TypeA" - were enhanced in versions 3 and 5).

```
Message ::= SEQUENCE {
    first   TypeA,
    second  TypeB,
    ... /
    [[2: version2      TypeC ]],
    [[4: version4-first TypeD,
      version4-second TypeE ]],
    [[6: version6      TypeF ]]
```

(In the [complete specification](#), the TypeA etc are replaced by INTEGER.)

The use of the double square brackets (no space allowed between them) enables version additions to be grouped together, and optionally given a version number. Omission of the version number (and indeed of the square brackets also) is permitted, and is common as these features were not present in the early ASN.1 support for extensibility.

The next three examples show an extension marker added in a version 1 CHOICE type (with a version2 addition without version brackets), an enumeration with two extensions, and the relaxation in a later version of a constraint on an integer:

```
Alternatives ::= CHOICE {
    first   TypeA,
    second  TypeB,
    ... /
    version2 TypeC }

Codes ::= ENUMERATED {code1, code2, ...,
                      v2-code, another-code}

NameSizes ::= INTEGER (1..64, ..., 65..MAX)
```

NOTE: The use of version2 without version brackets as an addition to Alternatives is not allowed in the same module with the additions in Message. In the [complete specification](#), version brackets are added to this.

The NameSizes in the last example is probably intended to constrain the size of a character string.

## *Other syntax*

Obsolete, not commonly used or deprecated syntax was grayed out in the Reference Card, with other syntax for advanced use being listed. These constructions are out of the scope of the Reference Card. They are included so that they can be recognised if seen in a published specification.

They are not included in the [complete specification](#), but some comments are included in the following (space did not permit the inclusion of these comments in the Reference Card):

```
MODULE-NAME.TypeName  -- A historical alternative to IMPORTS.

ABSTRACT-SYNTAX       -- Rarely used.

IMPLICIT TAGS        -- Largely historical, but often seen.
                    -- It relates to tags, and is out of scope.

EXPLICIT TAGS        -- Again historical, but rarely used.

EXPORTS ALL          -- Used in the complete specification.

EXTENSIBILITY IMPLIED -- Almost never used.

selection < ChoiceType -- Occasionally used,
                    -- but not important.

COMPONENTS OF SequenceType -- Used to insert common components

SEQUENCE {
    first    [0] INTEGER OPTIONAL,
    second   [1] EXPLICIT INTEGER,
    last     [99] IMPLICIT UserData }
                    -- An illustration of tags

SEQUENCE { ....., ... !29 }    -- An exception marker

[APPLICATION 29], [PRIVATE 6]  -- More tag illustrations

SET { ..... } -- An alternative to SEQUENCE.

SET OF                -- An alternative to SEQUENCE OF.

RELATIVE-OID          -- A type which carries the tail-end of an
                    -- object identifier value, with the root
                    -- statically determined. Sometimes misused to
                    -- provide an efficient encoding in BER of
                    -- SEQUENCE OF INTEGER.
```

```

EMBEDDED PDV      -- Used to embed messages from other
                  -- specifications, with both the message and
                  -- the encoding identified at communication time.

EXTERNAL          -- Historical (earlier version of EMBEDDED PDV).

INSTANCE OF      -- Another mechanism for embedding material.
                  -- Out of scope.

My-values INTEGER ::=
  {Set1 INTERSECTION (Set2 UNION Set3) EXCEPT Set4}
  -- An illustration of complex constraints being
  -- applied, and also of a Value Set Assignment.
  -- My-values INTEGER ::= { ..... }
  -- is exactly equivalent to:
  -- My-values ::= INTEGER ( ..... ).

PrintableString (SIZE (NameSizes) )
  --where-- NameSizes ::= INTEGER (0..64)
  -- Another example of a complex constraint.

CHARACTER STRING -- A character string type in which both the
                  -- character repertoire and the character
                  -- encoding are identified at communication
                  -- time. Not much used.

ObjectDescriptor -- Almost never used.

UTCTime          -- Historical relic, GeneralizedTime with a
                  -- two-digit year.

```

The following additional character string types also exist. Many are historical relics, but will be seen in published specifications. Those not grayed out in the card have some advantages over UTF8String for some applications:

```

BMPString        -- If unconstrained, produces 16-bit Unicode.
GeneralString
GraphicString
ISO646String
NumericString    -- Useful for numeric strings
                  -- (0 to 9 and space).

PrintableString
T61String
TeletexString
UniversalString
VideotexString
VisibleString    -- Printing ASCII characters.

```

## ***Definitions for information objects***

The information object class, information object, and information object set concepts were added to ASN.1 in the 1994 version, and replaced an earlier "Macro Notation", which had many problems.

These constructions are often used in published specifications, and their syntax is illustrated in the Reference Card. However, a detailed explanation of the concepts and functionality goes

beyond the scope of this Web page - a few short paragraphs have to suffice!

Broadly, an information object class definition assigns a name to a class of information (think of it as defining the form of a table, with the names of columns defined, and the nature of the information (an ASN.1 type, an ASN.1 value, etc) required for each column.

An information object definition (for a given class) provides the information needed to fill in one complete row of that table. Typically there will be many objects of a given class defined.

An information object set names a collection of information objects (of the same class).

The importance of this concept is mainly (but not exclusively) as a way of specifying what are the allowed types for inclusion as embedded types in an **open type**, how those types are to be identified at communication time (typically assignment of an object identifier value), and sometimes how they should be processed (for example, "matching rules").

When a class is defined, a WITH SYNTAX clause can be added which allows the definer of the class to determine the exact syntax to be used (frequently in some other standard) to define an object of that class. The WITH SYNTAX clause effectively allows a definition of an object to specify a row of the table, with ordinary words interspersed between the definition of the cell contents, in order to improve readability.

Information object class names and words in WITH SYNTAX clauses are required to be all upper case.

Information object names start with a lower-case letter, information object set names start with an upper case letter.

One very common information object class is a table with just two columns, one containing a type definition, and the second an object identifier value to be used to identify that type in an instance of communication.

NOTE: Because the object identifier namespace is hierarchically structured, widely available, and provides globally unique identification, it is common for different organizations to provide their own rows for such a table - to define their own information objects of the class.

This information object class is pre-defined (built-in) in the ASN.1 specification (the TYPE-IDENTIFIER class), and can be used by reference:

```
MY-SIMPLE-CLASS ::= TYPE-IDENTIFIER
```

A more typical class definition, illustrating most of the sorts of column that can be specified, would be:

```
MY-CLASS ::= CLASS {
  -- Note use of upper/lower case after &.
  -- This is semantically significant.
  &id                OBJECT IDENTIFIER UNIQUE,
  &simple-value       ENUMERATED
                    {high, medium, low} DEFAULT medium,
  &Set-of-values     INTEGER OPTIONAL,
```

```

&Any-type,
&an-inform-object    SOME-CLASS,
&A-set-of-objects    SOME-OTHER-CLASS }
WITH SYNTAX
{ KEY &id
  [ URGENCY &simple-value ] -- Optional
  [ VALUE-RANGE &Set-of-values ]
  PARAMETERS &Any-type
  SYNTAX &an-inform-object
  MATCHING-RULES &A-set-of-objects
  -- WORDS are optional and commas can be used
  -- as separators --}

```

The &id column holds values of type OBJECT IDENTIFIER. Values have to be unique within any complete table (information object set).

The DEFAULT on &simple-value says that if it is not specified when defining an information object, the cell in that row takes the default value.

The OPTIONAL on &Set-of-values indicates that if the definition is omitted, the cell in that row remains empty. If present, a set of INTEGER values are needed. (The type could be any ASN.1 type definition or type reference).

The &Any-type specifies that the nature of that column is to contain an ASN.1 type definition. When an object is defined, an in-line type definition or a type reference is needed.

The last two fields in the class definition say that the corresponding columns hold, respectively, the definition of (or reference to) a single information object or to a set of information objects of a specified information object class. (In the [complete specification](#), SOME-CLASS and SOME-OTHER-CLASS are given simple - not realistic - definitions.)

NOTE: There are two other forms for specifying the nature of the information that has to go into a column of the table, the so-called variable type value fields and the variable type value set fields. These are out of scope.

The following WITH SYNTAX clause is optional, and determines the syntax to be used for specifying objects of that class. Its use is illustrated below:

```

my-object MY-CLASS ::= {
  KEY      { ..... }
  URGENCY  high
  VALUE-RANGE { 1..10 | 20..30 }
  PARAMETERS My-type
  SYNTAX  defined-syntax  MATCHING-RULES { at-start | at-end | exact }
}

```

With a number of such definition, we can define an information object set:

```

My-object-set MY-CLASS ::= { object1|object2|object3,
  ...,
  version2-object }

```

And finally, we can apply it to constrain an open type in our message:

```
Message ::= SEQUENCE {
    key      MY-CLASS.&id ({My-object-set}),
    -- Has to be an OBJECT-ID from the set
    parms    MY-CLASS.&Any-type ({My-object-set} {@key})
    -- Has to be the PARAMETERS for the
    -- object with KEY. -- }
```

The type of the "key" component is simply an OBJECT IDENTIFIER (the type listed for the "&id" field of the class), but it is constrained by My-object-set (a so-called table constraint) to contain a value from one of the rows of the table represented by the information object set.

The type of the "parms" component is an "open type". It is just a "hole" for embedded material, but is required to contain a value of the type specified in the &Any-type column of the My-object-set table in the row that contains the &id value that is carried in the "key" field. This is called a component relation constraint.

The [complete specification](#) includes a possible value of Message, using the XML encoding rules.

## *Parameterisation*

ASN.1 provides a very powerful parameterisation mechanism. All assignments defining reference names (type references, value references, class definitions, object definitions, object set definitions) can be given a dummy parameter list. In the assignment statement, the dummy parameter is used on the right, and the parameterised reference name can be instantiated with actual parameters when needed.

For example:

```
Invoke-message {INTEGER:normal-priority, Parameter} ::=
    SEQUENCE {
        component1    INTEGER DEFAULT normal-priority,
        component2    Parameter }
```

Here we have two dummy parameters, an integer called normal-priority that is used to provide a DEFAULT value, and a type parameter that is used to provide the type forming the body of the Invoke-message.

Now we define our messages as a choice of two possibilities, that differ only in the default priority and the Type that is to be used:

```
Messages ::= CHOICE {
    first Invoke-message { low-priority, Type1 },
    second Invoke-message { high-priority, Type2 },
    ... }
```

## *Encodings*

ASN.1 provides a number of encoding rules, which the normal user really does not need to know much about - tools provide library routines to encode and decode C, C++ and Java values for implementors. The general characteristics of the different encoding rules are, however, of interest.

NOTE: ASN.1 is independent of the encoding rules. Any ASN.1 specification can be encoded using any of the encoding rules.

The current encoding rules are:

**PER:** A compact binary encoding transferring the minimum information needed to identify a value.

**XER:** Encoding ASN.1 values as XML syntax.

**BER:** A tag-length-value style of binary encoding very popular in the 1980s.

**DER:** An encoding with only one way to encode a given value (no encoder's options) used in security work.

**CER:** Another security-related encoding, not much used.

An encoding control notation (ECN) is available to completely determine the encoding of ASN.1 values. This is intended to enable legacy protocols, whose encoding were designed by hand, to be re-defined using ASN.1, with no change to the bits-on-the-line.

There are also Encoding Instructions that can vary XER and other encodings, for example, to determine which components of a sequence are to be encoded as XML attributes. These are not in the scope of this Reference Card or Web Page.

## *Example of a complete specification*

The following contains the examples from the Reference Card, but with added material to produce a syntactically correct (but rather artificial!) ASN.1 specification.

```
MY-MODULE1 { iso standard 2345 modules (0) basic-types (1) }
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

EXPORTS ALL;
IMPORTS perBasicAligned FROM ASN1-Object-Identifier-Module
    { joint-iso-itu-t asn1(1) specification(0) modules(0)
object-identifiers(1) };

Age ::= INTEGER
Country-name ::= UTF8String
Greeting ::= UTF8String
Hex-string ::= OCTET STRING

twenty-one-today Age ::= 21
spain Country-name ::= "ESPAÑA"
when GeneralizedTime ::= "200208192349.57894Z"
large-prime INTEGER ::= 1999999973
```

```

My-sequence ::= SEQUENCE {
    first    BOOLEAN,
    second   INTEGER OPTIONAL,
    third    INTEGER DEFAULT 129,
    fourth   BOOLEAN DEFAULT TRUE,
    fifth    REAL    DEFAULT 0.629,
    -- Or DEFAULT 62.9E-2
    sixth    UTF8String DEFAU "گَعِهَاجِ لَهِ ٱ",
    -- Unicode characters
    seventh  IA5String  DEFAULT "James Morrison",
    -- ASCII characters
    eighth   BIT STRING  DEFAULT '101100011'B,
    ninth    OCTET STRING DEFAULT '89AEF764'H,
    tenth    Alternatives }

Alternatives ::= CHOICE {
    first-alternative    TypeA,
    second-alternative   TypeB,
    third-alternative     NULL  }

DailyMaxTemperaturesForMonth ::=
    SEQUENCE (SIZE(28..31)) OF temperature INTEGER

VersionsSupported ::= BIT STRING {
    version1 (0),
    version2 (1),
    version3 (2) }

Message ::= SEQUENCE {
    message-id          INTEGER,
    version-bit-map     VersionsSupported
        DEFAULT {version1} }

Color ::= INTEGER {
    red(10), orange(20), yellow(30), green(40),
    blue(50), indigo(60), violet(70) } (0..80)

Codes ::= ENUMERATED {code1(0),code2(1),code3(2)}

oid1 OBJECT IDENTIFIER ::=
    {iso standard 2345 modules (0) basic-types (1)}

oid2 OBJECT IDENTIFIER ::= {joint-iso-itu-t ds(5)}

oid3 OBJECT IDENTIFIER ::= { oid2 modules(0) }

oid4 OBJECT IDENTIFIER ::= { oid3 basic-types(1) }

oid5 OBJECT IDENTIFIER ::= { 2 5 0 1 } -- equals oid4

TypeA ::= INTEGER (0..MAX)          -- only non-negative values

TypeB ::= INTEGER (-6..3 | 10..30)  -- only -6 to 3 or 10-30

TypeC ::= INTEGER (ALL EXCEPT 0)  -- 0 not allowed

TypeD ::= SEQUENCE (SIZE (0..10)) OF INTEGER

TypeE ::= IA5String (SIZE (1..25))(FROM ("A" .. "Z"))
    -- Only sizes 1-25 and characters "A"- "Z" allowed

```

```

TypeF ::= OCTET STRING (CONTAINING TypeI ENCODED BY perBasicAligned)
      -- perBasicAligned is imported from the ASN.1 standards

TypeG ::= UTF8String (PATTERN "\d#4-\d#2-\d#2")

TypeH ::= BIT STRING (CONSTRAINED BY { -- Results from encryption of --
      -- Message: See clause 59.4 -- } )

TypeI ::= SEQUENCE {a INTEGER OPTIONAL,
      b INTEGER OPTIONAL,
      c INTEGER OPTIONAL }
      (WITH COMPONENTS {... , a PRESENT } |
      WITH COMPONENTS {... , b PRESENT } |
      WITH COMPONENTS {... , c PRESENT } )
      -- Note the use of ellipsis in the WITH COMPONENTS
      -- syntax. This is, however, NOT the extension
      -- ellipsis, but is a use of the same three dots
      -- in a way that is specific to WITH COMPONENTS.

Message2 ::= SEQUENCE {
      first INTEGER,
      second INTEGER,
      ... /
      [[2: version2 INTEGER ]],
      [[4: version4-first INTEGER,
      version4-second INTEGER ]],
      [[6: version6 INTEGER ]] }

Alternatives2 ::= CHOICE {
      first INTEGER,
      second INTEGER,
      ... /
      [[3: version2 INTEGER ]] }

Codes2 ::= ENUMERATED {code1, code2, ...,
      v2-code, another-code}

NameSizes ::= INTEGER (1..64, ..., 65..MAX)

END

MY-MODULE2 { iso standard 2345 modules (0) xml-examples (2) }
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

IMPORTS Age, Hex-string, Greeting FROM MY-MODULE1
      { iso standard 2345 modules (0) basic-types (1) } ;

sixty-five-today ::= <Age>65</Age>
my-octets ::= <Hex-string>89aef764AEF</Hex-string>
plain-greeting ::= <Greeting>Hello World!</Greeting>
bells-and-whistles-greeting ::=
      <Greeting>
      <bel/><stx/>Hello World!<etx/>
      </Greeting>

Message ::= SEQUENCE {
      sender-id OBJECT IDENTIFIER,
      urgency ENUMERATED { high, normal, low },

```

```

actions      SEQUENCE OF CHOICE {
    insert    InsertionDetails,
    remove    RemovalDetails,
    update    UpdateDetails},
names        SEQUENCE OF name UTF8String,
confirm      BOOLEAN }

InsertionDetails ::= UTF8String

RemovalDetails ::= UTF8String

UpdateDetails ::= UTF8String

xml-document ::=
  <Message>
    <sender-id>2.39.6.45</sender-id>
    <urgency><normal/></urgency>
    <actions>
      <remove>Person X</remove>
      <insert>Person Y</insert>
      <insert>Person Z</insert>
    </actions>
    <names>
      <name>Confirmation name1</name>
      <name>Confirmation name2</name>
    </names>
    <confirm><true/></confirm>
  </Message>

END

MY-MODULE3 { iso standard 2345 modules (0) information-objects (3) }
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

MY-SIMPLE-CLASS ::= TYPE-IDENTIFIER

MY-CLASS ::= CLASS {
  -- Note use of upper/lower case after &.
  -- This is semantically significant.
  &id                OBJECT IDENTIFIER UNIQUE,
  &simple-value       ENUMERATED
                    {high, medium, low} DEFAULT medium,
  &Set-of-values     INTEGER OPTIONAL,
  &Any-type,
  &an-inform-object  SOME-CLASS,
  &A-set-of-objects  SOME-OTHER-CLASS }
WITH SYNTAX
{ KEY &id
  [ URGENCY &simple-value ] -- Optional
  [ VALUE-RANGE &Set-of-values ]
  PARAMETERS &Any-type
  SYNTAX &an-inform-object
  MATCHING-RULES &A-set-of-objects
  -- WORDS are optional and commas can be used
  -- as separators -- }

SOME-CLASS ::= CLASS { &syntax ENUMERATED {strict-XML, flexible-XML } }

defined-syntax SOME-CLASS ::= { &syntax strict-XML }

```

```

SOME-OTHER-CLASS ::= CLASS { &myString IA5String }
                    WITH SYNTAX { &myString }

at-start SOME-OTHER-CLASS ::= { "Hello" }
at-end   SOME-OTHER-CLASS ::= { "World" }
exact    SOME-OTHER-CLASS ::= { "!" }

my-object MY-CLASS ::= {
    KEY      { 1 2 3 4 5 6 }
    URGENCY  high
    VALUE-RANGE { 1..10 | 20..30 }
    PARAMETERS My-Type
    SYNTAX    defined-syntax
    MATCHING-RULES { at-start | at-end | exact } }

My-Type ::= INTEGER

My-object-set MY-CLASS ::= { object1|object2|object3,
    ...,
    version2-object }

object1 MY-CLASS ::= my-object

object2 MY-CLASS ::= { KEY { 1 2 3 4 5 7 }
    PARAMETERS My-Type
    SYNTAX    defined-syntax
    MATCHING-RULES { at-start | exact } }

object3 MY-CLASS ::= { KEY { 1 2 3 4 5 8 }
    PARAMETERS My-Type
    SYNTAX    defined-syntax
    MATCHING-RULES { at-end | exact } }

version2-object MY-CLASS ::= { KEY { 1 2 3 4 5 9 }
    PARAMETERS My-Type
    SYNTAX    defined-syntax
    MATCHING-RULES { at-start | at-end } }

Message ::= SEQUENCE {
    key      MY-CLASS.&id ({My-object-set}),
    -- Has to be an OBJECT-ID from the set
    parms    MY-CLASS.&Any-type ({My-object-set} {@key})
    -- Has to be the PARAMETERS for the
    -- object with KEY. -- }

myMessage ::= <Message>
            <key>1.2.3.4.5.6</key>

<parms><My-Type>7654321</My-Type></parms>
</Message>
    -- Note that in a binary encoding the tag
    -- <My-Type> will
    -- be absent, and only the relational constraint
    -- determines the decoding and meaning of
    -- the "parms" field.

END

MY-MODULE4 { iso standard 2345 modules (0) parameterization (4) }

```

```
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

Invoke-message {INTEGER:normal-priority, Parameter} ::=
  SEQUENCE {
    component1    INTEGER DEFAULT normal-priority,
    component2    Parameter }

Messages ::= CHOICE {
  first Invoke-message { low-priority, Type1 },
  second Invoke-message { high-priority, Type2 },
  ... }

low-priority INTEGER ::= 1
high-priority INTEGER ::= 10

Type1 ::= INTEGER
Type2 ::= IA5String

END
```

Copyright © 2005 OSS Nokalva, Inc. [All Rights Reserved.](#)