CODEC Tutorial

Alberto Sierra

22nd July 2003

Contents

Chapter 1

Introduction to ASN.1

The Internet implies the interconnection of applications running on remote machines that may represent internally the same data (e.g. a character string) in different ways. The hardware as well as the the software may affect this representation, e.g.:

- x86 Intel chips transmit first the least significant byte of a word (so called "little-endian" system) whereas Motorola does just the opposite ("big-endian" system);
- the end of a line is stored in UNIX systems only with the line feed ASCII symbol while DOS systems use the carriage return and the line feed symbols;
- in C language the \0 character is added at the end of a character string.

To solve this problem the applications will have to agree on a common encoding of the data. This implies that they will have to define on the one hand, a common set of data types (similar to those defined in programming languages, e.g. integers, booleans, arrays, etc.) and on the other hand, a common set of encoding rules to translate the defined data types into a bit stream for transmission across the network.

Matching these aspects ASN.1 was developed and internationally standardized by the ITU in 1985. It is a language with which any data transmitted between communicating applications can be defined. The next example shows how a data structure may be defined in ASN.1:

```
EmailMessage ::= SEQUENCE {
    from            IA5String,
    to              IA5String,
    subject         IA5String,
    message         IA5String,
    attachment      OCTET STRING,
    date            GeneralizedTime,
    urgent          BOOLEAN }
```

This declaration defines a data structure consisting of several components each denoted by an identifier (at the left) and its corresponding ASN.1 type.

ASN.1 provides a number of pre-defined types to describe basic data such as:

- integers (`INTEGER`),
- booleans (`BOOLEAN`),

- character strings (`IA5String`, `UniversalString`...),
- octet (8-bit bytes) strings (`OCTET STRING`),
- etc.

as well as constructed data types such as:

- structures, containing elements of different types (`SEQUENCE`),
- lists of elements of the same type (`SEQUENCE OF`),
- alternative types (`CHOICE`),
- etc.

You can find a complete list of the ASN.1 types and their corresponding Java classes within the CODEC package in the next chapter.

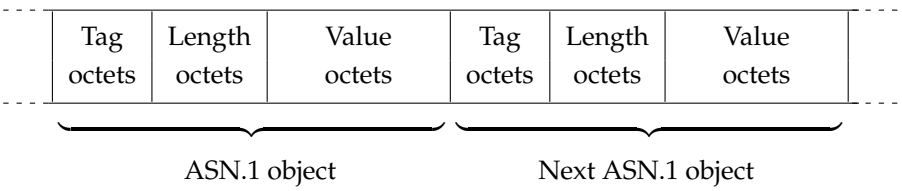
ASN.1 provides some encoding rules which determine how the data described by a certain ASN.1 declaration has to be translated into a stream of bits. E.g. if we want to transmit the following data packed in a structure as it was declared before

```
from            "somebody@somewhere.org"
to              "somebody_else@somewhere_else.org"
subject         "Hello"
message         "How do you do ..."
attachment      pic.jpg
date            10:30 01.01.03
urgent          false
```

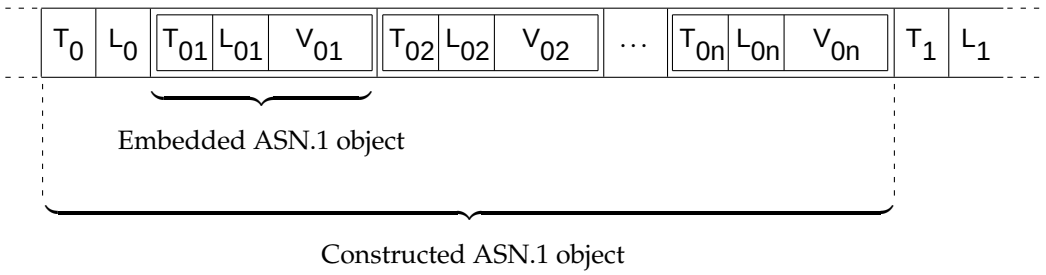
then ASN.1 will provide a precise bit pattern (directly deduced from the ASN.1 declaration) for the transmission of this data.

It has to be said that ASN.1 is not a programming language since there are no operators to handle the values or to make calculations with. It just offers some rules that must be taken into account while implementing communicating applications that may be running on different platforms and may be programmed in different languages.

Let us now take a detailed look at coding rules. The original encoding rules of ASN.1 are the Basic Encoding Rules (BER). But with time new encoding rules have been designed (DER, PER, etc.) to satisfy application dependent aspects. The principle of BER is that each ASN.1 object is encoded following the so called "TLV" pattern: tag, length and value. This means that every object receives a header that is usually two bytes long. The first byte indicates the ASN.1 type of the contents, the so called tag, and the second byte will indicate its length. The following bytes represent the value of the object in an encoded form. The encoding of the value depends on its ASN.1 type. Each ASN.1 type has its own encoding rules. For decoding the transmitted bit stream the receiving party will read the first byte (the tag) and the length of the follwing value, so that it will know how to decode the following bytes and how many have to be decoded that way.



The next figure shows the encoding of a constructed ASN.1 object, like the `EmailMessage` type declared above. The header of the constructed object is followed by the components, each one introduced by its own data type identifier (tag) and length:



Chapter 2

Simple Coding/Decoding Example

As it has been mentioned in the previous chapter ASN.1 was designed to facilitate transactions between heterogeneous systems. In this chapter we want to show some Java code pretending to simulate such a transaction in a very simple way. We will demonstrate the use of some of the basic classes of the CODEC package by performing the follwing tasks:

- first we will create a Java object that will represent some data modeled by an ASN.1 type;
- after that we will encode this object according to the encoding rules tied to the ASN.1 standard, creating an array of bytes (ones and zeros);
- and finally we will decode the byte array obtaining an identical Java object to the one created at the beginning.

So let us start with the first step:

```
ASN1IA5String asn1Object = new ASN1IA5String("Hello World");
```

`ASN1IA5String` is a Java class within the `codec.asn1` package that represents the ASN.1 `IA5String` type. In the `codec.asn1` package you can find all the Java classes that represent the corresponding standard ASN.1 types as shown in table 2.1

Encoding

ASN.1 supports several encoding rules. Those implemented in the CODEC package are the Distinguished Encoding Rules (DER). The reason for this is that these rules were designed to meet the needs of secure data transmission and the CODEC package was developed in this context. The class within the CODEC package that implements these rules is the `DEREncoder` class.

The next lines of code show how to encode the Java object we just created:

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
DEREncoder encoder = new DEREncoder(os);
asn1Object.encode(encoder);
```

First an empty output stream has to be instantiated. This stream is passed to the `DEREncoder` instance created in the next line. The encoder will need this stream for the output of the encoded data. Finally the `encode(Encoder)` method of the `ASN1IA5String` class is called. Within this

ASN.1 Type	Java Class in the CODEC package
NULL	ASN1Null
BOOLEAN	ASN1Boolean
INTEGER	ASN1Integer
ENUMERATED	ASN1Enumerated
IA5String	ASN1IA5String
UTF8String	ASN1UTF8String
T61String	ASN1T61String
BMPString	ASN1BMPString
UniversalString	ASN1UniversalString
VisibleString	ASN1VisibleString
PrintableString	ASN1PrintableString
GeneralizedTime	ASN1GeneralizedTime
UTCTime	ASN1UTCTime
OBJECT IDENTIFIER	ASN1ObjectIdentifier
BIT STRING	ASN1BitString
OCTET STRING	ASN1OctetString
SEQUENCE	ASN1Sequence
SEQUENCE OF	ASN1SequenceOf
SET	ASN1Set
SET OF	ASN1SetOf
CHOICE	ASN1Choice

Table 2.1: ASN.1 types and their respective Java classes in the CODEC package

call the encoder will read the `java.lang.String` object stored within `asn1Object` and will write the bytes representing the encoded `IA5String` object to the output stream. The parameter passed to the method (`Encoder`) is an interface of the CODEC package and denotes any class that may perform the encoding task. Till now the CODEC package only offers for this purpose the `DEREncoder` class but in the future further classes may be included in the package implementing other existing encoding rules.

With the following statement

```
byte[] encodedAsn1Object = out.toByteArray();
```

we can store the bytes in the output stream in a byte array. We will need the byte array later for demonstrating the decoding process. Printing its contents in hexadecimal representation will show the following 13 bytes:

```
0x16 0x0b 0x48 0x65 0x6c 0x6c 0x6f 0x20 0x57 0x6f 0x72 0x6c 0x64
 Tag  Len  H    e    l    l    o         W    o    r    l    d
```

According to the TLV pattern presented in the first chapter the first byte (`0x16`) represents the data type (tag) of the ASN.1 object (`IA5String`). In an adequate ASN.1 documentation you can find the correspondence between the ASN.1 standard types and the values of their respective tags. The second byte (`0x0b`, i.e. 11) means the length of the value. And each of the remaining 11 bytes represents one character of the "Hello World" string.

Decoding

In this section it will be shown how to decode the byte array just created, i.e. how a new `ASN1IA5String` instance storing the string "Hello World" can be created by reading these bytes. For it we will first create a new empty `ASN1IA5String` instance:

```
ASN1IA5String newAsn1Object = new ASN1IA5String();
```

It might surprise that for decoding, we should know its type beforehand. But actually when applications communicate, they follow some kind of dialog or protocol. This protocol will determine the different types of data each party will have to expect to receive in each state of the communication. If the communication is based on ASN.1 each party will own the ASN.1 declarations that describe the transmitted data structures. Otherwise they will not be able to interpret them.

There is to differentiate between decoding and interpreting data. Actually for just decoding ASN.1 encoded data, its type does not need to be known in advance, as we will show in the next section. But usually communicating applications will exchange data structures, in which each element has a concrete meaning, which has to be interpreted by the decoding party. This will be treated in more detail in the next chapter, when we deal with data structures.

So let us go on with the implementation of Java code for decoding the byte array created before. After initializing an empty `ASN1IA5String` object an input stream containing the encoded data has to be created. As you remember this data was stored in a byte array called `encodedAsn1Object`. The input stream will be then passed to a `DERDecoder` class instance, which will decode it contents.

```
ByteArrayInputStream in = new ByteArrayInputStream(encodedAsn1Object);
DERDecoder decoder = new DERDecoder(in);
```

Finally the encoded data, now located within the `DERDecoder` instance, can be decoded calling the `decode(Decoder)` method of the `ASN1IA5String` class:

```
newAsn1Object.decode(decoder);
```

This statement orders the decoder to read the first octet of the data to be decoded, which represents its ASN.1 type. This type has to match the ASN.1 type represented by the class of the Java object from which the `decode(Decoder)` method has been called. Otherwise an exception will be thrown. Then the next byte is read, which indicates the number of bytes that represent the encoded value. After this the decoder will be able to decode the number of bytes just read following the decoding rules for the ASN.1 type recognized. The decoded data will be stored as a `java.lang.String` object within `newAsn1Object`.

You can encode and decode any other class of the CODEC package representing an ASN.1 type, as well as any subclass of them, like it has been shown in this chapter since they all implement the `encode(Encoder)` and the `decode(Decoder)` methods.

Alternative decoding procedure

The `DERDecoder` class provides the `readType()` method with which a byte array can be decoded without having to know its ASN.1 type beforehand.

```
ByteArrayInputStream in = new ByteArrayInputStream(encodedAsn1Object);
DERDecoder decoder = new DERDecoder(in);
ASN1Type asn1Type = decoder.readType();
```

This method returns an `ASN1Type` instance. This is an interface implemented by all the classes in the CODEC package that represent an ASN.1 type, like the `ASN1Integer` class, the `ASN1Sequence` class, etc. You will be able to print to the standard output the contents of the `ASN1Type` instance. However if you want to determine the concrete class of the Java value stored within this instance, you will have to query all the possible classes that an `ASN1Type` can store, e. g. the `java.lang.String` class, the `java.math.BigInteger` class, etc.

```
ASN1Type asn1Type = decoder.readType();
Object value = asn1Type.getValue();

if (value instanceof String)
{
    ...
}
else if (value instanceof BigInteger)
{
    ...
}
else if (value instanceof ArrayList)
{
    ...
}
else ...
```

As you can see we have been able to decode the data this way, but a further processing will be quite difficult, since we do not know its type. Another characteristic of this way of communication is that any value of the same type will be treated the same way, i. e. it will have the same meaning for the receiver.

Decoding this way can be useful for testing and debugging purposes when you are not sure what kind of data your application is receiving.

You can find a complete Java application for encoding and decoding as it has been shown in this chapter in the appendix A (Coding/Decoding Demo).

Chapter 3

How to implement SEQUENCE types

Handling sets of data with a certain structure is something very common in the time of electronic data management. E.g. let us imagine a warehouse in which following data about the products sold may be stored:

```
product name          (stored as a string of characters),
product category      (stored as a string of characters),
price                 (stored as a number),
available quantity    (stored as a number)
```

Such kind of data could be described in ASN.1 with a `SEQUENCE` type, a type for defining data structures that consist of several components of different types, like the following sample ASN.1 declaration shows:

```
Product ::= SEQUENCE {
    product-name            IA5String,
    product-category        IA5String,
    price                   INTEGER,
    available-quantity      INTEGER }
```

We see that the data structure defined offers a fixed list of components denoted by an identifier and its corresponding ASN.1 type. These identifiers are not encoded and transmitted; they only serve as a description of the the meaning of each component. However they are indispensable since an ASN.1 type will be usually implemented in some programming language and the implementation will have to declare some variables, usually with similar names, to store and handle the data of these components. You will understand this better taking a look at the following Java implementation of a `SEQUENCE` type. For this purpose let us define a shorter ASN.1 type with only two components to avoid the resulting Java code to be too extensive:

```
Order ::= SEQUENCE {
    product-name         IA5String,
    needed-quantity      INTEGER }
```

To implement a Java class that will represent a `SEQUENCE` type it has to extend the `ASN1Sequence` class of the `codec.asn1` package. So the first lines of code would be:

```
import codec.asn1.*;

public class Order extends ASN1Sequence
{
...
```

The next step would be to declare member variables that will represent the components of the `Order` type:

```
    private ASN1IA5String productName_ = null;
    private ASN1Integer neededQuantity_ = null;
```

Obviously these have to match the Java class that represents the corresponding ASN.1 type indicated in the ASN.1 declaration.

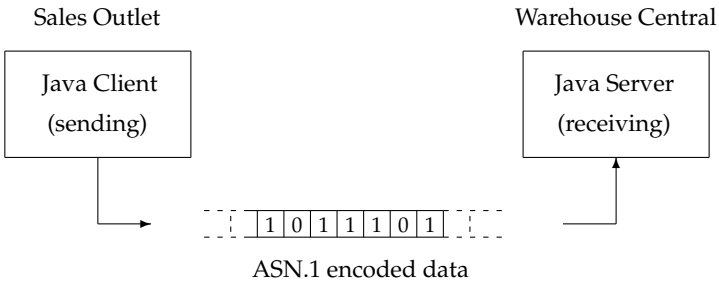
The next step would be to implement the constructors. Basically there are two different constructors you will have to implement:

- one with parameters for encoding the data to be sent.
- one without parameters for creating an empty object ready to be filled with Java values obtained by decoding the bit stream received.

It may be the case that for your concrete application you will only need to implement one of the constructors for a certain ASN.1 type.

The constructor with parameters

Before we go on let us describe the following scenario in which the implementation of ASN.1 may make sense.



Let us assume we have two applications running in different machines and that perform following tasks:

- the client application runs on a sales outlet and sends orders of the needed quantity of each product.
- the server application runs at the central warehouse and collects the orders of the clients (suppose there are other clients programmed in other languages and running on different platforms; otherwise there would be no necessity to use ASN.1 since Java already solves the problem of the platform diversity).

When two applications communicate exchanging ASN.1 coded coded data a set of ASN.1 types defining all the possible data structures that may be transmitted must have been previously declared. It is up to each party to implement its own code, may be in a different programming language, in which these ASN.1 types may be represented (e. g. as Java classes containing basic Java values such as character strings, integers, etc.) and in which mechanisms may be implemented to encode and decode the data following the ASN.1 coding rules.

So let us take the `Order` type defined above as the ASN.1 type to describe the transmitted data between the client and the server and the `Order` class we are demonstrating as the Java class that represents this type. Next you can see some lines of code that may belong to the implementation of the client application:

```
...
// Variables that represent the data to be transmitted.
String productName1;
int neededQuantity1;
String productName2;
int neededQuantity2;
...
(Set the variables declared above e. g. by reading a database.)
...
// Create instances of the Order class with the data read
// from the database.
Order order1 = new Order(productName1, neededQuantity1);
Order order2 = new Order(productName2, neededQuantity2);
...
// Encode the Order instances.
byte[] encodedOrder1 = order1.getEncoded();
byte[] encodedOrder2 = order2.getEncoded();
...
(Transmit the byte arrays containing the encoded data.)
...
```

The `byte[] getEncoded()` method will be explained in detail later.

As you can see the `Order` class is instantiated receiving some Java values through the parameters. The constructor called here is the one needed to encode the data to be transmitted. Its implementation would look like:

```
public Order(String productName, int neededQuantity)
{
    // Allocate memory for the member variables.
    super(2);

    // Create member variables with the use of the parameters.
    productName_ = new ASN1IA5String(productName);
    neededQuantity_ = new ASN1Integer(neededQuantity);

    // Add the member variables to the class.
    add(productName_);
    add(neededQuantity_);
}
```

First of all lets take a look at the superclasses of the `Order` class:

```
java.util.ArrayList
        |
    codec.asn1.ASN1AbstractCollection
            |
        codec.asn1.ASN1Sequence
                |
              Order
```

As you can see the `ASN1Sequence` class is a `java.util.ArrayList` subclass, i.e. it represents a list of Java objects.

The first call of the constructor shown above (`super(2)`) is not necessary; it just optimizes the initial memory allocation for this class. The parameter passed to the superclass call (two) indicates the number of member variables this class has, i. e. the number of components the corresponding ASN.1 type has. According to this value the adequate initial memory will be allocated, actually for two `java.lang.Object` instances. The lines of code that follow initialize the member variables with the values of the parameter list and add them to the `Order` class calling the `add(Object)` method inherited from the `ArrayList` class.

Note: We would like to point out that while defining a constructor for encoding data refering to Java classes of the CODEC package in the parameter list should be avoided, like in the next example:

```
public Order(ASN1IA5String productName, ASN1Integer neededQuantity)
{
...
```

The disadvantage of such a constructor declaration becomes visible if we have to call it, like in the piece of code we showed before:

```
import codec.asn1.*;
...
(Read database and set variables with the data read.)
...
// Create instances of the Order class with the data obtained
// from database.
Order product1 = new Order(
    new ASN1IA5String(productName1),
    new ASN1Integer(neededQuantity1));
Order product2 = new Order(
    new ASN1IA5String(productName2),
    new ASN1Integer(neededQuantity2));
...
(Encode the Order instances.)
...
(Transmit the byte arrays containing the encoded data.)
...
```

As you can see we would then be forced to import the CODEC packages while implementing the client application. The CODEC package, as any other package, should only be imported whenever it is necessary, e.g. in the implementation of the classes that represent the custom ASN.1 types.

The constructor without parameters

Now lets concentrate on the server. It expects to receive data defined by the `Order` type in encoded form. The next lines of code show very roughly how such a server may be implemented:

```
...
(Listen to input stream for incoming data.)
...
// Byte array to store incoming data.
byte[] encodedAsn1Object;
...
(Store incoming data in byte array.)
...
// Create empty instance of the Order class ready to
// decode the data received.
Order order = new Order();

// Decode received data.
order.decode(encodedAsn1Object);

// Extract the interesting data.
String productName = order.getProductName();
int neededQuantity = order.getNeededQuantity();
...
(Process/Forward the order.)
...
```

The `decode(byte[])` method will be explained in detail in the next section.

For transforming the data received in usable Java values an empty `Order` object is created. With it the data can be decoded and properly stored and accessed. The next constructor shows how `ASN1Sequence` subclasses will have be initialized to perform this task:

```
public Order()
{
    super(2);

    // Initialize the member variables.
    productName_ = new ASN1IA5String();
    neededQuantity_ = new ASN1Integer();

    // Add the member variables to the class.
    add(productName_);
    add(neededQuantity_);
}
```

It is almost the same as the constructor for encoding data, only that the member variables are initialized calling their respective constructors without parameters.

Note: An important thing to keep in mind is that the member variables have to be added to the class in both constructors in the same order as the components are listed in the corresponding ASN.1 type declaration. In the encoding constructor this will determine the order in which the member variables are encoded. In the decoding constructor it will determine the order in which the encoded components are expected.

Coding/Decoding

Now let us take a detailed view on the encoding/decoding steps. As you saw in the previous pieces of code belonging to the client and the server implementations the `byte[] getEncoded()` and `decode(byte[])` methods were called. The first method was called at the client and it returns a byte array representing an encoded `Order` object by reading the data stored in its member variables:

```
public byte[] getEncoded()
{
    // Create an output stream to which the encoder will write the
    // bytes representing the encoded Order type object.
    ByteArrayOutputStream out = new ByteArrayOutputStream();

    // Create encoder object.
    DEREncoder encoder = new DEREncoder(out);

    // Byte array to store the bytes representing the encoded Order
    // type value and that will be returned by this method.
    byte[] encodedAsn1Object = null;

    try
    {
        // Order the encoder to read the member variables of this
        // class and to write the bytes representing an Order type
        // value in encoded form to the output stream.
        this.encode(encoder);

        // Store the bytes within the output stream in the byte
        // array to be returned by this method.
        encodedAsn1Object = out.toByteArray();

        // Close the stream.
        encoder.close();
    }
    catch (ASN1Exception e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    // Return the byte array containing the encoded Order type
    // value.
    return encodedAsn1Object;
}
```

The implementation of the `decode(byte[])` method would receive a byte array representing an encoded `Order` type value and set the member variables representing its components:

```
public void decode(byte[] encodedData)
```

```
{
    // Create an input stream initialized with the bytes
    // representing an encoded Order type value and a decoder to
    // decode them.
    ByteArrayInputStream in = new ByteArrayInputStream(encodedData);
    DERDecoder decoder = new DERDecoder(in);

    try
    {
        // Decoder reads the bytes in the input stream and sets the
        // member variables of this class.
        this.decode(decoder);
        decoder.close();
    }
    catch (ASN1Exception e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

As you can see both methods follow the same steps that were in the previous chapter. The curious thing of them is that you can copy and paste them in any class you may implement that subclasses any of the CODEC classes that represents a standard ASN.1 type. They will perform the same task correctly and you will not need to adapt them to your class.

Now let us instantiate the `Order` class as follows:

```
    Order asn1Object = new Order("Soap", 152);
```

and take a look to the bytes of the encoded instance:

```
 0x30 0x0a 0x16 0x04 0x53 0x6f 0x61 0x70 0x02 0x02 0x00 0x98
  T0   L0   T01  L01  S    o    a    p    T02  L02    152
```

The first byte (0x30) represents the standard ASN.1 type of the `Order` type: the `SEQUENCE` type. The second byte (0x0a, i.e. 10) indicates that 10 bytes follow representing the contents of the `Order` object. The third byte (0x16) represents the ASN.1 type of the first component: the `IA5String` type. The fourth byte (0x04, i.e. 4) represents the length of the value of the first component, according to the 4 characters of the string "Soap". The nineth and tenth bytes (0x02, 0x02) represent the ASN.1 type (`INTEGER`) and the length of the second component since two bytes represent the number 152.

Set and get methods

The set and get methods would be implemented as follows:

```
    public void setProductName(String productName)
    {
        productName_ = new ASN1IA5String(productName);
```

```
        set(0, productName_);
    }

    public String getProductName()
    {
        return productName_.getString();
    }

    public void setNeededQuantity(int neededQuantity)
    {
        neededQuantity_ = new ASN1Integer(neededQuantity);
        set(1, neededQuantity_);
    }

    public int getNeededQuantity()
    {
        return neededQuantity_.getBigInteger().intValue();
    }
```

Here it should be also taken into account to represent the parameters of the set methods as well as the return values of the get methods by standard Java values since these methods will be called by other classes.

Additionally you should take care in the set methods to set the member variables at the correct index. As you remember they were added to the class in the constructors in a certain order, namely in the order that the components were given in the corresponding ASN.1 declaration. In the set methods the member variables receive new values and these have to replace the old ones that had been previously included to the list represented by the `Order` class. This is done by calling the `set(int, Object)` method.

Set methods are optional. We have introduced them for the sake of completeness. On the other hand get methods are very useful since they make accessible the data corresponding to each component of the `SEQUENCE` type.

You can find the complete sources to this chapter in the appendix B (`SEQUENCE` Demo).

Chapter 4

How to implement SEQUENCE OF types

The ASN.1 `SEQUENCE OF` type represents a list of an indefinite number of elements of the same type. To illustrate this type let us use the `Order` type defined in the last chapter:

```
Order ::= SEQUENCE {
    product-name         IA5String,
    needed-quantity      INTEGER }
```

It may happen that not only an order for a single product may be necessary but rather for several. For this case the definition of a `SEQUENCE OF` type would be very helpful since it would represent a list of orders for all the quantities needed. A declaration of such a type would be as simple as:

```
OrderList ::= SEQUENCE OF Order
```

To implement a Java class that will represent a `SEQUENCE OF` type it will have to extend the `ASN1SequenceOf` class:

```
import codec.asn1.*;

public class OrderList extends ASN1SequenceOf
{
...
```

The implementation of such a class will basically require only the implementation of one constructor (one without parameters):

```
public OrderList()
{
    super(Order.class);
}
```

It will declare the class of the elements it will store, here the `Order` class. This class should be the Java implementation of the corresponding ASN.1 type, the one that defines the elements stored by the `OrderList` type. It is obvious that to be able to compile and run the `OrderList` class we will also have to implement and compile the `Order` class.

For demonstration purposes we have introduced in the source code of this class (see appendix C - `SEQUENCE OF` Demo) an additional constructor with an array of `Order` class instances as a parameter.

The implementation of the `byte[] getEncoded()` and `decode(byte[])` methods are the same as in the previous chapter.

The next lines of code show how an `ASN1SequenceOf` subclass object may be initialized and filled with data for encoding:

```
OrderList orderList = new OrderList();
Order order0 = new Order("Soap", 152);
Order order1 = new Order("Shampoo", 346);
orderList.add(order0);
orderList.add(order1);
```

The `ASN1SequenceOf` class is also a subclass of the `java.util.ArrayList` class and inherits consequently its `add(Object)` method, with which elements can be added to it. Once we have filled the `ASN1SequenceOf` instance with data we could then encode and decode the object the same way as shown in the previous chapter.

If you want to replace an element with a new value or access a single element you can use the `set(int, Object)` and the `Object get(int)` methods inherited from the `ArrayList` class:

```
Order newOrder1 = new Order("Shampoo", 256);
orderList.set(1, newOrder1);

Order someOrder = (Order)orderList.get(0);
```

Chapter 5

Note about SET and SET OF types

We do not recommend to define and use `SET` and `SET OF` types since they do not offer a significant functional gain. These types are per definition equal to the `SEQUENCE` and `SEQUENCE OF` types except that the elements they include do not necessarily have to be transmitted in the order established by the corresponding ASN.1 declaration. It is up to the implementing party of that types to establish an ordering criteria in which the components will be encoded (e.g. alphanumeric order). The decoding party will decode the elements obviously in the order they are received but the decoded data may be provided to the application on top according to a different ordering criteria than the sending party.

In the CODEC package the implementation of the corresponding classes `ASN1Set` and `ASN1SetOf` is the same as the implementation of the `ASN1Sequence` and `ASN1SequenceOf` classes, i.e. the elements that will be stored within these classes are encoded and decoded in the same order as they will be added to the class. It is then up to the programmer (implementing its own ASN.1 type) to etablish the order in which the elements will be added to the class to determine in which order they will be encoded. Only the question arises, in which case such a decission should be necessary.

Chapter 6

How to implement CHOICE types

A `CHOICE` type provides a list of different possible data types it may stand for. This type is used when a certain information can be modeled in different ways. The next ASN.1 declaration shows such a case:

```
Payment-method ::= CHOICE {
    check          IA5String,
    credit-card    Credit-card,
    cash           NULL }

Credit-card ::= SEQUENCE {
    ...
    }
```

For demonstrating a sample implementation of a Java class that will represent a `CHOICE` type let us define a shorter ASN.1 type:

```
Response ::= CHOICE {
    acknowledgment       NULL,
    error-code           INTEGER }
```

The corresponding class will have to extend the `ASN1Choice` class so the first lines of code would be:

```
import codec.asn1.*;

public class Response extends ASN1Choice
{
...
```

For each different data type the `CHOICE` type may represent a member variable of the corresponding class will have to be declared:

```
private ASN1Null acknowledgment_ = null;
private ASN1Integer errorCode_ = null;
```

The implementation of an constructor for encoding data is usually not recommended. The reason for this is that the `CHOICE` type does not have an own tag in encoded form. This means that to encode a value of a this type just an value of one of the possible types has to be encoded. This has to be then taken into account when the `CHOICE` type is enclosed in a structured type and the corresponding Java class has to be implemented. We will see an example below.

FRAGE: Implementierung von einer SEQUENCE mit CHOICE als componente ?

The constructor for decoding would be implemented as follows:

```
public Response()
{
    // Allocate memory for the member variables.
    super(2);

    // Initialize the member variables.
    acknowledgment_ = new ASN1Null();
    errorCode_ = new ASN1Integer();

    // Add the member variables to a list accessible when
    // accessing this class as a generic ASN1Choice instance.
    addType(acknowledgment_);
    addType(errorCode_);
}
```

Like in the implementation of a `SEQUENCE` type the superconstructor call (`super(2)`) can be omitted since it just optimizes the initial memory allocation for this class. The number passed to the superconstructor call (here two) represents the number of choices of the corresponding `CHOICE` type.

The following calls initialize the member variables and add them to this class calling the `addType(ASN1Type)` method inherited from the `ASN1Choice` class. The member variables are in fact stored within a `java.util.ArrayList` object which is a private member variable of the `ASN1Choice` class. This way the member variables of the `Response` class are accessible, when an instance of this class is treated as a generic `ASN1Choice` instance.

The next lines of code show how encoding and decoding is done for `ASN1Choice` subclasses. As it has been said before `ASN1Choice` subclasses should not be instantiated. Instead of this just an object of one of the possible types should be created. In the following sample code a `NULL` object is created and encoded:

```
ASN1Null ack = new ASN1Null();
ByteArrayOutputStream os = new ByteArrayOutputStream();
DEREncoder encoder = new DEREncoder(os);
ack.encode(encoder);
byte[] encodedData = out.toByteArray();
```

Printing the bytes representing the encoded acknowledgment object to the standard output will show the following bytes:

```
0x05 0x00
```

Here you can see the encoded representation a `NULL` object which consists exceptionally of only two bytes, tag value 5 and length 0. Since we expect a `Response` object, either a `NULL` or a `INTEGERT` object, we can decode these bytes with:

```
Response newAsn1Object = new Response();
ByteArrayInputStream in = new ByteArrayInputStream(encodedData);
DERDecoder decoder = new DERDecoder(in);
newAsn1Object.decode(decoder);
```

The implementation of the `decode(Decoder)` method of the `ASN1Choice` class causes the decoder instance to query the ASN.1 types represented by the member variables of the `Response` class. If one of them matches the ASN.1 type of the byte array to be decoded then the value will be decoded and stored in the corresponding member variable.

You can find the whole sources from which the pieces of code shown in this chapter have been extracted in appendix D (CHOICE Demo).

Chapter 7

How to implement ENUMERATED types

`ENUMERATED` types are used for declaring a list of possible values (not ASN.1 types!) an object may store. Let us take a look at the following example:

```
Response ::= ENUMERATED {
    successful            (0),
    try-later             (1),
    corrupted             (2),
    unauthorized          (3) }
```

Each possible value is associated with an integer value. Only this value will be encoded so that we can say that the `ENUMERATED` type is in principle equal to the `INTEGER` type at least in the way the values are encoded, just the tag is different. At the receiver the integer value is decoded and interpreted according to the given ASN.1 type declaration.

The `ENUMERATED` type is commonly used within communicating applications for describing the state of a system or for error codes.

Now let us look at the Java implementation of the`Response` type. Therefore the `ASN1Enumerated` class has to be subclassed:

```
import codec.asn1.*;

public class Response extends ASN1Enumerated
{
...
```

For each possible value the `ENUMERATED` type may store a constant may be defined:

```
    public static final int SUCCESSFUL = 0;
    public static final int TRY_LATER = 1;
    public static final int CORRUPTED = 2;
    public static final int UNAUTHORIZED = 3;
```

A constructor for encoding may be implemented as follows:

```
public Response(int value) throws IllegalArgumentException
{
    super(value);

    if (   (value != SUCCESSFUL)
        && (value != TRY_LATER)
        && (value != CORRUPTED)
        && (value != UNAUTHORIZED))
    {
        throw new IllegalArgumentException();
    }
}
```

It just should filter out the inappropriate arguments (integers) according to the ASN.1 type declaration.

A constructor for decoding would be implemented as simple as:

```
public Response()
{
    super();
}
```

Next the set and get methods:

```
public void setInt(int value)
{
    if (   (value != SUCCESSFUL)
        && (value != TRY_LATER)
        && (value != CORRUPTED)
        && (value != UNAUTHORIZED))
    {
        throw new IllegalArgumentException();
    }
    else
    {
        try
        {
            setBigInteger(BigInteger.valueOf(value));
        }
        catch (ConstraintException e)
        {
            e.printStackTrace();
        }
    }
}

public int getInt()
{
    return getBigInteger().intValue();
}
```

Notice that the `ASN1Enumerated` class as well as the `ASN1Integer` class store their integer value as a `java.math.BigInteger` object.

You can find the whole source code in appendix E (ENUMERATED Demo).

Chapter 8

How to implement optional fields

ASN.1 offers the possibility to set components within a SEQUENCE type as optional. This is done when it is expected not to have always data to be encoded for these components.

The next ASN.1 declaration describes a web form for registration in an online shop or something similar, in which some fields must be filled out and others, denoted with the OPTIONAL clause, can be left empty by the user:

```
Form ::= SEQUENCE {
    title      [0]   IA5String    OPTIONAL,
    name       [1]   IA5String,
    address    [2]   IA5String,
    phone      [3]   IA5String,
    fax        [4]   IA5String    OPTIONAL,
    email      [5]   IA5String }
```

The meaning of the numbers within brackets will be explained in the next chapter.

The use of the OPTIONAL clause permits not to encode the corresponding components, if there was no data provided for them. This feature avoids unnecessary overhead, specially when the optional components represent complex SEQUENCE types.

For the demonstration of a Java implementation of a SEQUENCE type with optional fields let us define a shorter ASN.1 type:

```
Person ::= SEQUENCE {
    age      INTEGER OPTIONAL,
    name     IA5String }
```

As it had been shown in chapter 3 "How to implement a SEQUENCE type" the first lines of code would be:

```
import codec.asn1.*;

public class Person extends ASN1Sequence
{
    private ASN1Integer age_ = null;
    private ASN1IA5String name_ = null;
    ...
```

Since it may occur that there is no data to be transmitted for the optional components, it may make sense to implement two constructors for the encoding of data: one setting values for all the fields (also the optional ones) and another setting values only for the non-optional fields.

```
public Person(int age, String name)
{
    super(2);
    age_ = new ASN1Integer(age);
    name_ = new ASN1IA5String(name);
}

public Person(String name)
{
    super(1);
    name_ = new ASN1IA5String(name);
}
```

As you can see, here the member variables are not added to the class as it had been shown in chapter 3. We recommend to implement a separate function, like the one shown next, in which this is done.

```
protected void reinit()
{
    clear();
    if (age_ != null)
    {
        add(age_);
    }
    add(name_);
}
```

The `clear()` function called here is inherited from the `ArrayList` class and removes all the objects that may have been previously added to the class. This method ensures that the optional member variables are added to the class, and hence encoded, only if they are not empty.

The `reinit()` method should be called always before encoding, so you should overwrite the inherited `encode(Encoder)` method:

```
public void encode(Encoder enc)
{
    reinit();
    super.encode(enc);
}
```

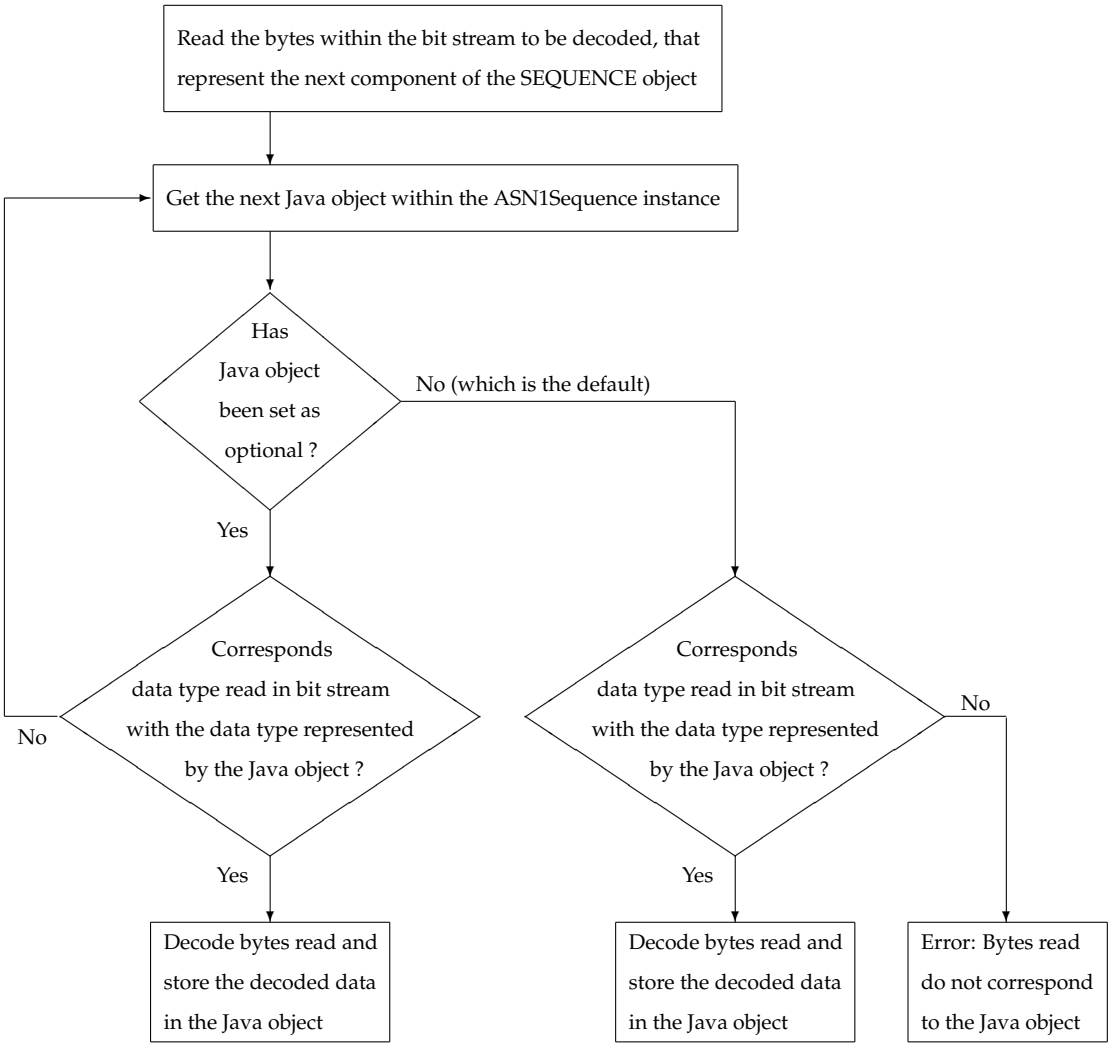
The constructor for decoding would be implemented as follows:

```
public Person()
{
    super(2);

    age_ = new ASN1Integer();
    age_.setOptional(true);
    name_ = new ASN1IA5String();
```

```
        add(age_);
        add(name_);
    }
```

Here all the member variables, also the optional ones, are initialized and added to the class. The member variables representing the optional components have to call `setOptional(true)`. This way the decoder can check whether the byte array to be decoded has necessarily to contain data for that component. In the next flow chart we can see in more detail the steps the decoder goes to decode the bytes corresponding to a component of a `SEQUENCE` type.



This procedure shown here has to be reproduced for decoding each component of the `SEQUENCE` object. Now let us follow this procedure in a concrete example. For it let us create an object of the class we are demonstrating without providing any data for the optional component:

```
Person asn1Object = new Person("Volker");

ByteArrayOutputStream out = new ByteArrayOutputStream();
DEREncoder encoder = new DEREncoder(out);
byte[] encodedAsn1Object = null;

asn1Object.encode(encoder);
encodedAsn1Object = out.toByteArray();
encoder.close();
```

The bytes representing the created object in encoded form would have the following values:

```
0x30 0x08 0x16 0x06 0x56 0x6f 0x6c 0x6b 0x65 0x72
 T0   L0   T01  L01  V    o    l    k    e    r
```

The first two bytes represent the header of the object: the tag for `SEQUENCE` types (0x30) and the length of this object (8 bytes). The third byte would represent the ASN.1 type of the second component (`IA5String`) since the first component had not been encoded.

Now for decoding the byte array just created we could write the following code:

```
Person newAsn1Object = new Person();
ByteArrayInputStream in = new ByteArrayInputStream(encodedData);
DERDecoder decoder = new DERDecoder(in);
newAsn1Object.decode(decoder);
```

The decoder will go through the following steps to decode the bytes:

1. It will check if the first member variable of `newAsn1Object` (`age_`) had been set as optional, which is the case. `age_` is the first member variable of `newAsn1Object`, because it had been added first to the class in the constructor without paramenters, which had been called for creating `newAsn1Object`.
2. The decoder will compare the ASN.1 type represented by this member variable (`INTEGER`) with the ASN.1 type of the first component within the byte array to be decoded. The ASN.1 type of this component is the `IA5String` type, denoted by its tag value (0x16).
3. Since these types do not match and the member variable is optional the decoder will disregard this member variable assuming that no data to be decoded was provided for it.
4. Then the next member variable will be checked if it had been set as optional, which is not the case.
5. As the ASN.1 type of the second member variable (`IA5String`) and the ASN.1 type of the next component to be decoded are the same, the bytes will be decoded and stored in this member variable.

FRAGE: Sollen wir nicht diese ganze Prozedur nicht mit einem oder zwei Stze zusammenfassen ?

It just remains to show the set and get methods:

```
protected void setAge(int age)
{
    age_ = new ASN1Integer(age);
```

```
}

protected int getAge()
{
    return age_.getBigInteger().intValue();
}

protected void setName(String name)
{
    name_ = new ASN1IA5String(name);
}

protected String getName()
{
    return name_.getString();
}
```

As you can see, in the set methods the new values of the member variables do not replace the old ones that might have been added previously to the class, as it had been shown in chapter 3. This is done in the `reinit()` method shown before.

The implementation of a function like the following may make sense if you want to ensure an optional component not to be encoded:

```
public void removeAge()
{
    age_ = null;
}
```

You can find the whole sources to this chapter in appendix F (OPTIONAL Fields Demo).

Chapter 9

How to implement tagging

Lets us consider the following ASN.1 type declaration:

```
Person ::= SEQUENCE {
    title         IA5String,
    name          IA5String }
```

And now let us create an object of this type (let it call `person0`) assigning some values to its components.

```
person0 Person ::= {
    title "Sir",
    name "Peter" }
```

The bytes representing this object in encoded form would have the following values:

```
0x30 0x0c 0x16 0x03 0x53 0x69 0x72 0x16 0x05 0x50 0x65 0x74 0x65 0x72
 T0   L0   T01  L01  S    i    r    T02  L02  P    e    t    e    r
```

The first two bytes (0x30, 0xa2) would represent the header of the whole `SEQUENCE` object, i. e. its type and its length. The subsequent bytes would represent the encoded data for the two components. As long as the decoding party also owns the same ASN.1 type declaration, it will be able to assign the data to be decoded to the corresponding components, since these were encoded in the order given by the type declaration.

Let us now consider the same type declaration but with the first component set as optional:

```
Person ::= SEQUENCE {
    title         IA5String    OPTIONAL,
    name          IA5String }
```

In this case we will have the problem that the decoder will not be able to assign unequivocally the arriving data to the corresponding components. To understand better the reason for this let us create two data structures of the last type (with optional elements) by specifying some values for the components:

```
form1 Form ::= {
    title "",
    name "Peter",
    etc. }

form1 Form ::= {
    title "Sir",
    name "Peter",
    etc. }
```

The values of the bytes representing these two objects in encoded form would be the following:

```
form1 :
 0x30 0x07 0x16 0x05 0x50 0x65 0x74 0x65 0x72
  T0   L0   T01  L01  P    e    t    e    r

form2 :
 0x30 0x0c 0x16 0x03 0x53 0x69 0x72 0x16 0x05 0x50 0x65 0x74 0x65 0x72
  T0   L0   T01  L01  S    i    r    T02  L02  P    e    t    e    r
```

The bytes that follow the main header in the `form1` object would correspond to the `IA5String` value representing the character string "Peter". This data should be assigned to the `name` component since it was produced encoding this component.

On the other hand the bytes that follow the `SEQUENCE` header in the `form2` object would represent the character string "Sir" and should be assigned to the `title` component.

So the question is: How does the decoder know to which component it should assign the bytes right after the `SEQUENCE` header ?

For these cases the tagging mechanism should be applied. It consists in the following: The components whose encoded data may be wrongly assigned to other components at decoding time should be marked in the type declaration with a number notated within brackets. E. g.

```
Person ::= SEQUENCE {
    title      [0] IA5String    OPTIONAL,
    name           IA5String }
```

This way the data of these components receive a new special header in which the number given in the declaration is contained. The decoder, also owning this declaration, is then able to assign the encoded data to its corresponding components since these have been unambiguously identified with this number.

There are two ways of tagging components: explicitly and implicitly. Roughly they differ in that the encoding for implicit tagging is more compact although it cannot always be applied.

There are two main cases in which tagging has to be used:

- in a `SEQUENCE` type in which there are optional and not optional components or only optional components of the same standard ASN.1 type, e.g.:

```
MySequence1 ::= SEQUENCE {
    component1       [0] IA5String    OPTIONAL,
    component2           IA5String,
    etc. }

MySequence2 ::= SEQUENCE {
```

```
    component1      [0] IA5String    OPTIONAL,
    component2      [1] IA5String    OPTIONAL,
    etc. }
```

- in a `CHOICE` type in which the choices are of the same standard ASN.1 type, e. g.

```
MyChoice ::= CHOICE {
    choice1         [0] OCTET STRING,
    choice2         [1] OCTET STRING }
```

Explicit tagging

This way of tagging is the default, i. e. to indicate that this kind of tagging will be applied just a number between brackets has to be notated for each component that has to be tagged, as it has been shown in the previous examples. Let us take a look at the values of the bytes representing the previously defined `form2` object in encoded form while implementing explicit tagging:

```
form2:
 0x30 0x0e 0xa0 0x05 0x16 0x03 0x53 0x69 0x72 0x16 0x05 0x50 0x65 0x74 0x65 0x72
  T0   L0   T01  L01 T011 L011  S    i    r    T02  L02  P    e    t    e    r
```

The first two bytes represent the header of the whole `SEQUENCE` object. The following two bytes (0xa0 0x05) represent a new header (tag and length) that is inserted when components are tagged explicitly. The tag of this header has a value of 0xa0, i. e. 160. The values of the tags of explicitly tagged components is computed as follows:

```
value of tag = 160 + number within brackets in type declaration
```

The values of these tags cannot be confounded with the values of the tags of not tagged components. The possible values of the tags of not tagged components go from 0 to 30 and each value represents a different ASN.1 type.

The second byte of the new header (0x05) represents the length of the tagged component, including its own header. That is the next 5 bytes represent the tagged component as it would have been transmitted without having it tagged:

```
... 0x16 0x03 0x53 0x69 0x72 ...
... T011 L011  S    i    r   ...
```

Implicit tagging

This way of tagging offers a more compact encoding but it cannot always be applied, e.g. to tag a component which is a `CHOICE` type. We will explain the reason for this later. To declare that this kind of tagging is to be applied, the keyword `"IMPLICIT"` is to be inserted in the ASN.1 type declaration after the numbers within brackets:

```
Person ::= SEQUENCE {
    title      [0] IMPLICIT IA5String    OPTIONAL,
    name                    IA5String }
```

Now let us see how the `form2` object defined before would be encoded using this kind of tagging:

```
form2 :
 0x30 0x0c 0x80 0x03 0x53 0x69 0x72 0x16 0x05 0x50 0x65 0x74 0x65 0x72
  T0   L0   T01  L01  S    i    r    T02  L02  P    e    t    e    r
```

As you can see, in this case a new header is not introduced but the original header of the tagged component is replaced with a new one (see bytes 3 and 4). The decoder will get the information about the ASN.1 type of this component from the type declaration. The values of the tags of implicitly tagged components is computed as follows:

```
value of tag = 124 + number within brackets in type declaration
```

This way of tagging offers a more compact encoding but it cannot be applied for tagging `CHOICE` types. Let us see the following example:

```
MySequence ::= SEQUENCE {
    component1    [0] IMPLICIT MyChoice OPTIONAL,
    component2                 IA5String }

MyChoice ::= CHOICE {
    choice1    IA5String,
    choice2    UTF8String }
```

Since the implicit tagging mechanism overwrites the header of the tagged component, we will not be able to know which choice has been set in the `CHOICE` object at decoding time.

Java implementation

Now let us see how the following ASN.1 type would be implemented in Java:

```
Person ::= SEQUENCE {
    title      [0] IA5String    OPTIONAL,
    name           IA5String }
```

In this declaration the first component is explicitly tagged. The first lines would be the same as in a common `ASN1Sequence` implementation.

```
import codec.asn1.*;

class Person extends ASN1Sequence
{
    ASN1IA5String title_ = null;
    ASN1IA5String name_ = null;
    ...
```

Constants defining the tag values, i. e. the numbers given in the ASN.1 declaration between brackets, may be declared:

```
    final int TITLE_TAG = 0;
```

The constructors for encoding data would be the same as shown in the last chapter since here we have also an optional component:

```
public Person(String title, String name)
{
    super(2);
    title_ = new ASN1IA5String(title);
    name_ = new ASN1IA5String(name);
}

public Person(String name)
{
    super(1);
    name_ = new ASN1IA5String(name);
}
```

The member variables are then added in the `reinit()` function, which is always to be called before an encoding is performed:

```
protected void reinit()
{
    clear();
    if (title_ != null)
    {
        add(new ASN1TaggedType(TITLE_TAG, title_, true));
    }
    add(name_);
}

public void encode(Encoder enc)
{
    reinit();
    super.encode(enc);
}
```

As you can see the member variable representing the tagged component is not added to the class but an instance of the `ASN1TaggedType` class. The first argument (`TITLE`) represents the nmber between brackets given in the ASN.1 declaration for that component. The second argument (`title_`) is the member variable itself, which contains the data to be transmitted and its ASN.1 type. And the third argument denotes the tagging mechanism: explicit or implicit. The argument given here (`true`) indicates that explicit tagging has to be applied. If in the ASN.1 declaration was implicit tagging requested just the `false` value should be passed.

The constructor for decoding would be coded as follows:

```
public Person()
{
    super(2);

    title_ = new ASN1IA5String();
    name_ = new ASN1IA5String();

    add(new ASN1TaggedType(TITLE_TAG, title_, true, true));
    add(name_);
}
```

Here you can see that the `ASN1TaggedType` instance added to the class is created calling a different constructor than the one called in the `reinit()` function. This one requires a fourth parameter, a boolean value, that indicates whether the corresponding tagged component is optional, which is the case in our example.

In the last chapter we showed that in the constructor for decoding, the member variables representing optional components should be set as optional calling the `setOptional(true)` method. Here this optional flag is set through the constructor of the `ASN1TaggedType` class just described.

While encoding, this flag does not need to be set. In this and the last chapter we implemented the classes in a way that optional components were simply not added to the class and hence not encoded if they were empty (see `reinit()` function).

You can find the whole sources to this chapter in appendix G (Tagging Demo).

Chapter 10

How to implement default values

ASN.1 offers the possibility to define default values for components of a `SEQUENCE` type.

```
Client ::= SEQUENCE {
    name             IA5String,
    address          IA5String,
    country      [0] IA5String    DEFAULT Germany }
```

Such components should be encoded only if their value is different than the default one. This implies that the decoder will automatically assume that the default value has to be set for this component if no data was sent for it.

Let us see how this feature can be implemented in Java. For this we could define a Java constant storing the default value:

```
String DEFAULT_COUNTRY = "Germany";
```

Since the `country` component has to be explicitly tagged we can define a constant value for its new tag:

```
final int COUNTRY_TAG = 0;
```

Two constructors for encoding data may be implemented: one setting a value different than the default and another one assuming the default value for that component.

```
public Client(String name, String address, String country)
{
    super(3);
    name_ = new ASN1IA5String(s1);
    address_ = new ASN1IA5String(s3);
    if (!country.equals(DEFAULT_COUNTRY))
    {
        country_ = new ASN1IA5String(s2);
    }
}

public Client(String name, String address)
{
```

```
        super(2);
        name_ = new ASN1IA5String(name);
        address_ = new ASN1IA5String(address);
    }
```

The `reinit()` and the `encode(Encoder)` functions:

```
    protected void reinit()
    {
        clear();
        add(name_);
        add(address_);
        if (country_ != null)
        {
            add(new ASN1TaggedType(COUNTRY_TAG, country_, true));
        }
    }

    public void encode(Encoder enc)
    {
        reinit();
        super.encode(enc);
    }
```

The constructor for decoding:

```
    public Client()
    {
        super(3);

        name_ = new ASN1IA5String();
        address_ = new ASN1IA5String();
        country_ = new ASN1IA5String();

        add(name_);
        add(address_);
        add(new ASN1TaggedType(COUNTRY_TAG, country_, true, true));
    }
```

Finally the set and get methods for the `country` component could be implemented as follows:

```
    public setCountry(String country)
    {
        if (!s2.equals(DEFAULT_COUNTRY))
        {
            country_ = new ASN1IA5String(country);
        }
    }

    public String getCountry()
    {
        if (country_ != null)
            return country_.getString();
```

```
        else
            return DEFAULT_STRING;

    }
```

You can find the whole sources to this chapter in appendix H (Default Values Demo).

Chapter 11

How to implement ANY DEFINED BY types

This type has disappeared from the ASN.1 standard and its use is strongly inadvisable. Nevertheless the CODEC package contains classes to deal with it. Lets take a look at the following example:

```
ErrorMessage ::= SEQUENCE {
    code          OBJECT IDENTIFIER,
    parameter     ANY DEFINED BY code

      --   code | ASN.1 type of parameter
      -- =======|=======================
      --    1.0 | NULL
      --    1.1 | INTEGER
      --    1.3 | ErrorParameterType1 }

ErrorParameterType1 := SEQUENCE {
    p1      BOOLEAN,
    p2      IA5String }
```

The `ANY DEFINED BY` clause indicates that the ASN.1 type of the component `parameter` depends on the value of the component `code`. A component referenced after the `ANY DEFINED BY` clause can only be of type `OBJECT IDENTIFIER`, `INTEGER` or `CHOICE` between these two types.

The ASN.1 `OBJECT IDENTIFIER` type represents a list of integers. It provides the capabillity to classify any kind of objects in a hierarchical way, e. g. like the chapters and subchapters of a book: 1.0, 1.1.0, 1.1.1, 1.2, 2.0, etc.

As you can see, a table has been included in the `ErrorMessage` declaration as a comment. In ASN.1 comments can be added to a declaration after a double dash "–". The table shows the different values the component `code` may have and the corresponding ASN.1 types of the component `parameter`. This table has to be provided to the sender and to the receiver, but not necessarily this way.

The component referenced after the `ANY DEFINED BY` clause should always be declared before the `ANY DEFINED BY` component. This way its value will be encoded and decoded first. This way the specific ASN.1 type of the `ANY DEFINED BY` component can be determined before its decoding, what is necessary for it.

The ANY DEFINED BY type was originally meant to be used during the specification design phase provided that the sender and the receiver had agreed on a few types they could exchange.

Next we will show how to implement the ErrorMessage type. As it represents an ASN.1 SEQUENCE type it will have to extend the ASN1Sequence class.

The member variable representing the ANY DEFINED BY component (the component parameter) should be declared as an ASN1OpenType instance.

```
private ASN1ObjectIdentifier code_ = null;
private ASN1OpenType parameter_ = null;
```

The ASN1OpenType class has the capability to decode any class representing an ASN.1 type and is therefore appropriate for modeling ANY DEFINED BY types.

Next you can see the implmentation of the constructor for encoding data:

```
public ErrorMessage(ASN1ObjectIdentifier code, ASN1Type parameter)
{
    super(2);

    code_ = code;
    parameter_ = parameter;

    add(code_);
    add(parameter_);
}
```

FRAGE: Hier die Parameter sind CODEC Klassen.

The argument parameter has to be declared as an ASN1Type instance since it can represent different ASN.1 types. Remember that the ASN1Type interface is implemented by all the classes in the CODEC package that represent a standard ASN.1 type, and hence by all their subclasses.

Next you can see the implementation of the constructor for decoding:

```
public ErrorMessage()
{
    super(2);

    code_ = new ASN1ObjectIdentifier();
    parameter_ = new ASN1OpenType(new ErrorResolver(code_));

    add(code_);
    add(parameter_);
}
```

As you can see the member variable parameter_ is initialized receiving as an argument an instance of the ErrorResolver class.

When modeling ANY DEFINED BY types the ASN1OpenType class requires as an argument an instance of a class that has to implement the Resolver interface. Such a class will actually implement the mapping between the values of the component referenced after the ANY DEFINED BY clause, in our example the component code, and the possible ASN.1 types of the ANY DEFINED BY type, in the ErrorMesssage example, the component parameter.

As you can see the ErrorResolver instance passed to the ASN1OpenType class receives itself as an argument a reference to the member variable code_, i. e. the member variable whose value

will determine the ASN.1 type of the component `parameter`. This reference will be needed later for decoding the component.

Let us take a look at the implementation of the `ErrorResolver` class to understand how the `ANY DEFINED BY` type will be decoded.

The class declaration will just have to indicate the implementation of the `Resolver` interface:

```
class ErrorResolver implements Resolver
{
...
```

A member variable that will store a reference to the member variable `code_` of the `ErrorMessage` has to be declared. The value of this variable will determine which Java class the `ErrorResolver` class will have to provide for decoding the component `parameter`.

```
    private ASN1ObjectIdentifier code_ = null;
```

The reference to the member variable `code_` of the `ErrorMessage` class will be obtained through the constructor. Remember that this constructor is called in the decoding constructor of the `ErrorMessage` class.

```
    public ErrorResolver(ASN1ObjectIdentifier code)
    {
        code_ = code;
    }
```

The `resolve(ASN1Type)` method is the one that have to implement the calsses that implement the `Resolver` interface. It has to return an instance of a class with which the `ANY DEFINED BY` component, in our case the component `parameter`, can be decoded, depending on the value of the component `code_`. This method will be called from the `decode(Decoder)` method of the `ASN1OpenType` class.

```
    public ASN1Type resolve(ASN1Type caller) throws ResolverException
    {
        if (code_.toString().equals("1.0"))
        {
            return new ASN1Null();
        }
        else if (code_.toString().equals("1.1"))
        {
            return new ASN1Integer();
        }
        else if (code_.toString().equals("1.2"))
        {
            return new ErrorParameterType1();
        }
        else throw new ResolverException();
    }
```

It has to be noticed that, when the `ErrorResolver` class is constructed, the member variable `code_` of the `ErrorMessage` does not have a value. However the `resolve(ASN1Type)` method will be called right before the bytes corresponding to the `parameter` component are to be decoded, i. e. after the component `code` has been decoded. At this point the member variable

`code_` will have a value and the ASN.1 type of the component `parameter` can be determined so that it can be decoded.

And that would be everything we need for implementing the `ErrorResolver` class.

Regarding to the implementation of the `ErrorMessage` it just remains to show the implementation of the get methods:

```
public ASN1ObjectIdentifier getCode() {
    return code_;
}

public ASN1Type getParameter() {
    return parameter_;
}
```

FRAGE: Set Methoden sinnvoll oder nicht ?

You can find the whole sources to this chapter in the appendix I (ANY DEFINED BY Demo).

Chapter 12

The OID Registry

The `OIDRegistry` class from the CODEC package permits to build a mapping of `OBJECT IDENTIFIER` values with ASN.1 types for resolving `ANY DEFINED BY` values. This class permits to add or remove other `OIDRegistry` instances, so that a mapping can be extended or constrained.

The first lines of code of your own registry class would be:

```
import java.util.*;

public class SampleOIDRegistry extends AbstractOIDRegistry {
...
```

The identifiers and the respective ASN.1 types would be then specified as follows:

```
    // Store each OID as an array of integers.
    static final private int[][] oids_ =
    {
        {1,0},
        {1,1},
        {1,2},
        {1,3}
    };

    // The ASN.1 types registered under the OIDs.
    static final private Object[] types_ =
    {
        ASN1Null.class,
        ASN1Integer.class,
        "ErrorParameterType1"
    };
```

This way the `new ASN1ObjectIdentifier("1.0")` object would deliver the `ASN1Null` class. As you can see there is also the possibility to add ASN.1 types as strings (see "ErrorParameterType1"). You can declare then a prefix string indicating the package of the classes given as strings.

PENDING: Wozu ist der prefix da ? Ist es notwendig ? Sol es erwhnt werden ?

```
static final private String prefix_ = "oid_registry_demo";
```

You should also declare a variable of the `java.util.Map` class to store the ids and the corresponding Java classes representing the ASN.1 types.

```
static private Map map_ = new HashMap();
```

Then a constructor should be implemented:

```
public SampleOIDRegistry(OIDRegistry parent)
   {
       super(parent);

       synchronized(map_)
        {
            if (map_.size() == 0)
             {
                 int i;

                 for (i=0; i<types_.length; i++)
                     map_.put(
                         new ASN1ObjectIdentifier(oids_[i]),
                         types_[i]);
             }
        }
   }
```

In it the `OBJECT IDENTIFIER` values and the corresponding ASN.1 types are mapped. And that would be everything we need for the implementation of our own registry. The ASN.1 types are accessible through the following method inherited from the `OIDRegistry` class:

```
ASN1Type getASN1Type(ASN1ObjectIdentifier)
```

Now let us see how a `SEQEUNCE` type with an `ANY DEFINED BY` field may be implemented using a registry. Threfore let us base on the `ErrorMessage` type defined in the last chapter.

The only difference will appear in the implementation of the constructor for decoding:

```
public ErrorMessage()
{
    super(2);

    code_ = new ASN1ObjectIdentifier();
    parameter_ = new ASN1OpenType(new SampleOIDRegistry(), code_);

    add(code_);
    add(parameter_);
}
```

Here the `ASN1OpenType` class is instantiated with a `SampleOIDRegistry` object. Next you can see the implementation of this constructor:

```
public ASN1OpenType(OIDRegistry registry, ASN1ObjectIdentifier oid)
{
    resolver_ = new DefinedByResolver(registry, oid);
}
```

As we explained in the last chapter an `ASN1OpenType` instance needs a resolver that provides the appropriate class for decoding the `ANY DEFINED BY` component that it represents. In this case the resolver is created instantianting the `DefinedByResolver` class. This CODEC class implements the `Resolver` interface and is able to provide an appropriate Java object for decoding the `ANY DEFINED BY` component, by quering the registry with which it is initialized.

If you want to take a look at the whole sources to this chapter you will find them in the appendix J (OIDRegistry Demo).

Appendix A

Coding/Decoding Demo

```
package simple_codec_demo;

import java.io.*;
import codec.asn1.*;

/**
 * This class shows how a simple ASN.1 object is created, encoded and decoded.
 */

public class CodingDecodingDemo
{

    public static void main(String[] args)
    {
        // Coding process.

        // Create ASN.1 object.
        ASN1IA5String asn1Object = new ASN1IA5String("Hello World");

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object.toString());
        System.out.println();

        // Create an output stream to which an encoder will write the bytes
        // representing the encoded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the encoded ASN.1 object
        // before closing the output steam.
        byte[] encodedAsn1Object = null;

        try
        {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the encoded ASN.1 object to the output stream.
            asn1Object.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            encodedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e)
```

```
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}

// Print the bytes representing the encoded ASN.1 object to the
// standard output.
StringBuffer buf = new StringBuffer();
String octet;
int i;

for (i=0; i<encodedAsn1Object.length; i++)
{
    octet = Integer.toHexString(encodedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1)
    {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the encoded ASN.1 object:");
System.out.println(buf.toString());
System.out.println();


// Decoding process.

// Create new empty ASN.1 object.
ASN1IA5String newAsn1Object = new ASN1IA5String();

// Create an input stream initialized with the bytes representing the
// encoded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(encodedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try
{
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    newAsn1Object.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e)
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(newAsn1Object.toString());
System.out.println();


// Show alternative decoding process.

// Coding process.
```

```
    asn1Object = new ASN1IA5String("How are you ?");

    System.out.println("ASN.1 object: ");
    System.out.println(asn1Object.toString());
    System.out.println();

    out = new ByteArrayOutputStream();
    encoder = new DEREncoder(out);
    encodedAsn1Object = null;

    try
    {
        asn1Object.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();
    }
    catch (ASN1Exception e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    buf = new StringBuffer();
    for (i=0; i<encodedAsn1Object.length; i++)
    {
        octet = Integer.toHexString(encodedAsn1Object[i] & 0xff);
        buf.append(" 0x");
        if (octet.length() == 1)
        {
            buf.append('0');
        }
        buf.append(octet);
    }
    System.out.println("Bytes representing the encoded ASN.1 object:");
    System.out.println(buf.toString());
    System.out.println();


    // Alternative decoding procedure.

    // Create new empty ASN.1 object.
    ASN1Type asn1Type = null;
    in = new ByteArrayInputStream(encodedAsn1Object);
    decoder = new DERDecoder(in);

    try
    {
        asn1Type = decoder.readType();
        decoder.close();
    }
    catch (ASN1Exception e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    System.out.println("New ASN.1 object got by decoding the bytes above:");
    System.out.println(asn1Type.toString());
    System.out.println();
}
```

```
}
```

Appendix B

SEQUENCE Demo

```
package sequence_demo;

import java.io.*;
import codec.asn1.*;

/**
 * This class shows a sample implementation of the ASN.1 SEQUENCE type:
 *
 * Order := SEQUENCE {
 *     product-name             IA5String,
 *     available-quantity       INTEGER }
 */

public class Order extends ASN1Sequence
{

    /**
     * Member variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1IA5String productName_;
    private ASN1Integer neededQuantity_;


    /**
     * Constructor with parameters for creating an object to be encoded.
     */
    public Order(String productName, int neededQuantity)
    {
        // Allocate memory for the member variables.
        super(2);

        // Create ASN.1 objects from the parameters.
        productName_ = new ASN1IA5String(productName);
        neededQuantity_ = new ASN1Integer(neededQuantity);

        // Add the member variables to the SEQUENCE.
        add(productName_);
        add(neededQuantity_);
    }


    /**
     * Constructor without parameters to create an object ready to decode.
     */
    public Order()
{
        super(2);
```

```
    // Create empty member variables.
    productName_ = new ASN1IA5String();
    neededQuantity_ = new ASN1Integer();

    // Add the member variables to the SEQUENCE.
    add(productName_);
    add(neededQuantity_);
}



/**
 * Returns an instance of this class in encoded form (as a byte array).
 */
public byte[] getEncoded()
{
    // Create an output stream to which an encoder will write the bytes
    // representing the encoded ASN.1 object.
    ByteArrayOutputStream out = new ByteArrayOutputStream();

    // Create encoder object.
    DEREncoder encoder = new DEREncoder(out);

    // Byte array to store the bytes representing the encoded ASN.1 object
    // before closing the output steam.
    byte[] encodedAsn1Object = null;

    try
    {
        // Encoder reads the ASN.1 object and writes the bytes representing
        // the encoded ASN.1 object to the output stream.
        this.encode(encoder);

        // Store the bytes in the output stream in a byte array.
        encodedAsn1Object = out.toByteArray();

        // Close the stream.
        encoder.close();
    }
    catch (ASN1Exception e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    return encodedAsn1Object;
}


/**
 * Decodes the byte array passed as argument (representing a encoded
 * OrderList object).
 */
public void decode(byte[] encodedData)
{
    // Create an input stream initialized with the bytes representing the
    // encoded ASN.1 object from before and a decoder to read it.
    ByteArrayInputStream in = new ByteArrayInputStream(encodedData);
    DERDecoder decoder = new DERDecoder(in);

    try
    {
        // Decoder reads the bytes in the input stream and sets the value
        // of the new created ASN.1 object
```

```
            this.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }


    /**
     * Set and get methods.
     */
    public void setProductName(String productName)
{
        productName_ = new ASN1IA5String(productName);
        set(0, productName_);
    }

    public String getProductName()
{
        return productName_.getString();
    }

    public void setNeededQuantity(int neededQuantity)
{
        neededQuantity_ = new ASN1Integer(neededQuantity);
        set(1, neededQuantity_);
    }

    public int getNeededQuantity()
    {
        return neededQuantity_.getBigInteger().intValue();
    }

}
```

Appendix C

SEQUENCE OF Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE OF type:
 *
 * Students ::= SEQUENCE OF Student
 */

public class Students extends ASN1SequenceOf {

    /**
     * Constructor.
     */
    public Students() {

        super(Student.class);
    }


    /**
     * Constructor with an array as a parameter.
     */
    public Students(Student[] pStudents) {

        super(Student.class, pStudents.length);

        // Add the elements of the array to this class.
        for (int i=0; i<pStudents.length; i++) {
            add(i, pStudents[i]);
        }
    }

}
```

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SET OF type:
 *
 * SetOfStudents ::= SET OF Student
 */

public class SetOfStudents extends ASN1SetOf {

    /**
     * Constructor.
     */
    public SetOfStudents() {

        super(Student.class);
    }


    /**
     * Constructor with an array as a parameter.
     */
    public SetOfStudents(Student[] pStudents) {

        super(Student.class, pStudents.length);

        // Add the elements of the array to this class.
        for (int i=0; i<pStudents.length; i++) {
            add(pStudents[i]);
        }
    }

}
```

```
import java.io.*;

import codec.*;
import codec.asn1.*;

/**
 * This class shows how a SEQUENCE OF object is created, coded and decoded.
 */

public class CodingDecodingSequenceOf {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Students asn1Object = new Students();

        Student student1 = new Student(
            new ASN1IA5String("Student1"),
            new ASN1Integer("12345"));

        Student student2 = new Student(
            new ASN1IA5String("Student2"),
            new ASN1Integer("67890"));

        asn1Object.add(student1);
        asn1Object.add(student2);

        // Print the ASN.1 object to the standard output.
        System.out.print("ASN.1 object: ");
        System.out.println(asn1Object.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");


        // Decoding process.

        // Create new empty ASN.1 object.
        Students newAsn1Object = new Students();

        // Create an input stream initialized with the bytes representing the
        // coded ASN.1 object from before and a decoder to read it.
        ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
        DERDecoder decoder = new DERDecoder(in);

        try {
            // Decoder reads the bytes in the input stream and sets the value
            // of the new created ASN.1 object
            newAsn1Object.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(newAsn1Object.toString() + "\n");
    }

}
```

Appendix D

CHOICE Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 CHOICE type:
 *
 * Number ::= CHOICE {
 *     integer    INTEGER,
 *     string     IA5String }
 */

public class Number extends ASN1Choice {

    /**
     * Memeber variables representing each possible choice.
     */
    private ASN1Integer integer;
    private ASN1IA5String string;

    /**
     * Constructors.
     */
    public Number() {

        super(2);

        integer = new ASN1Integer();
        string = new ASN1IA5String();

        addType(integer);
        addType(string);
    }

    public Number(ASN1Integer pInteger) {

        super(2);

        integer = pInteger;
        string = new ASN1IA5String();

        addType(integer);
        addType(string);

        setInnerType(integer);
    }
```

```
    public Number(ASN1IA5String pString) {

        super(2);

        integer = new ASN1Integer();
        string = pString;

        addType(integer);
        addType(string);

        setInnerType(string);
    }


    /**
     * Set and get methods.
     */
    public void setInteger(ASN1Integer pInteger) {

        integer = pInteger;
        setInnerType(integer);
    }

    public ASN1Integer getInteger() {
        if (getInnerType() instanceof ASN1Integer) {
            return integer;
        }
        else {
            throw new IllegalStateException();
        }
    }

    public void setString(ASN1IA5String pString) {

        string = pString;
        setInnerType(string);
    }

    public ASN1IA5String getString() {
        if (getInnerType() instanceof ASN1IA5String) {
            return string;
        }
        else {
            throw new IllegalStateException();
        }
    }

}
```

```
import java.io.*;
import codec.*;
import codec.asn1.*;

/**
 * This class shows how a CHOICE object is created, coded and decoded.
 */
public class CodingDecodingChoice {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Number asn1Object1 = new Number(new ASN1Integer(1));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }
```

```
System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Number asn1Object2 = new Number();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");

// Setting choice to a string object.
asn1Object2.setString(new ASN1IA5String("One"));

// Print the ASN.1 object to the standard output.
System.out.println("Setting choice to a string object: ");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}
```

```
        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Number asn1Object3 = new Number();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object3.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object3.toString() + "\n");

    }

}
```

Appendix E

ENUMERATED Demo

```
import codec.asn1.*;
import java.math.BigInteger;

/**
 * This class shows a sample implementation of an ASN.1 ENUMERATED type:
 *
 * ResponseStatus ::= ENUMERATED {
 *     successful          (0),  -- understood request
 *     malformedRequest    (1)   -- malformed request }
 */

public class ResponseStatus extends ASN1Enumerated {

    /**
     * Constants representing each possible value.
     */
    public static final int SUCCESSFUL = 0;
    public static final int MALFORMED_REQUEST = 1;


    /**
     * Constructor.
     */
    public ResponseStatus(int value) {

        super(value);

        if ((value != SUCCESSFUL) &&
            (value != MALFORMED_REQUEST)) {
            throw new IllegalArgumentException();
        }
    }
```

```
    /**
     * Set and get methods.
     */
    public void setInt(int value) {

        if ((value != SUCCESSFUL) &&
            (value != MALFORMED_REQUEST)) {
            throw new IllegalArgumentException();
        }
        else {
            try {
                setBigInteger(BigInteger.valueOf(value));
            }
            catch (ConstraintException e) {
                e.printStackTrace();
            }
        }
    }

    public int getInt() {
        return getBigInteger().intValue();
    }
}
```

Appendix F

OPTIONAL Fields Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type
 * with an OPTIONAL component:
 *
 * Response := SEQUENCE {
 *     yes        BOOLEAN,
 *     reason     IA5String    OPTIONAL }
 */
public class Response extends ASN1Sequence {

    /**
     * Memeber variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1Boolean yes;
    private ASN1IA5String reason;

    /**
     * Constructor without parameters.
     */
    public Response() {

        super(2);

        yes = new ASN1Boolean();
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Constructor for setting a value for the optional component.
     */
    public Response(ASN1Boolean pYes, ASN1IA5String pReason) {

        super(2);

        yes = pYes;
        reason = pReason;

        add(yes);
        add(reason);
    }
```

```
    /**
     * Constructor for leaving the optional component empty.
     */
    public Response(ASN1Boolean pYes) {

        super(2);

        yes = pYes;
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Set and get methods.
     */
    public void setYes(ASN1Boolean pYes) {
        yes = pYes;
        set(0, yes);
    }

    public ASN1Boolean getYes() {
        return yes;
    }

    public void setReason(ASN1IA5String pReason) {
        reason = pReason;
        set(1, reason);
    }

    public ASN1IA5String getReason() {
        return reason;
    }

    /**
     * Remove the optional field.
     */
    public void removeReason() {
        reason = new ASN1IA5String();
        reason.setOptional(true);
        set(1, reason);
    }

}
```

```
import codec.*;
import codec.asn1.*;

import java.io.*;
import java.math.BigInteger;

/**
 * This class shows how a SEQUENCE object with OPTIONAL fields is created,
 * coded and decoded.
 */
public class CodingDecodingOptional {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Response asn1Object1 = new Response(
            new ASN1Boolean(false), new ASN1IA5String("I don't want."));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Response asn1Object2 = new Response();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");


// Removing optional field.
asn1Object2.removeReason();

// Print the ASN.1 object to the standard output.
System.out.println("Removing optional field. ASN.1 object:");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding.
Response asn1Object3 = new Response();

in = new ByteArrayInputStream(codedAsn1Object);
decoder = new DERDecoder(in);

try {
    asn1Object3.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object3.toString() + "\n");

// Setting optional field.
asn1Object3.setReason("I don't like.");

// Print the ASN.1 object to the standard output.
System.out.println("Setting optional field. ASN.1 object:");
System.out.println(asn1Object3.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object3.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
        // Print the coded ASN.1 object to the standard output.
        buf = new StringBuffer();

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Response asn1Object4 = new Response();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object4.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object4.toString() + "\n");

    }

}
```

Appendix G

OPTIONAL Fields Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type
 * with an OPTIONAL component:
 *
 * Response := SEQUENCE {
 *     yes        BOOLEAN,
 *     reason     IA5String    OPTIONAL }
 */
public class Response extends ASN1Sequence {

    /**
     * Memeber variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1Boolean yes;
    private ASN1IA5String reason;

    /**
     * Constructor without parameters.
     */
    public Response() {

        super(2);

        yes = new ASN1Boolean();
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Constructor for setting a value for the optional component.
     */
    public Response(ASN1Boolean pYes, ASN1IA5String pReason) {

        super(2);

        yes = pYes;
        reason = pReason;

        add(yes);
        add(reason);
    }
```

```
    /**
     * Constructor for leaving the optional component empty.
     */
    public Response(ASN1Boolean pYes) {

        super(2);

        yes = pYes;
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Set and get methods.
     */
    public void setYes(ASN1Boolean pYes) {
        yes = pYes;
        set(0, yes);
    }

    public ASN1Boolean getYes() {
        return yes;
    }

    public void setReason(ASN1IA5String pReason) {
        reason = pReason;
        set(1, reason);
    }

    public ASN1IA5String getReason() {
        return reason;
    }

    /**
     * Remove the optional field.
     */
    public void removeReason() {
        reason = new ASN1IA5String();
        reason.setOptional(true);
        set(1, reason);
    }

}
```

```
import codec.*;
import codec.asn1.*;

import java.io.*;
import java.math.BigInteger;

/**
 * This class shows how a SEQUENCE object with OPTIONAL fields is created,
 * coded and decoded.
 */
public class CodingDecodingOptional {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Response asn1Object1 = new Response(
            new ASN1Boolean(false), new ASN1IA5String("I don't want."));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Response asn1Object2 = new Response();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");


// Removing optional field.
asn1Object2.removeReason();

// Print the ASN.1 object to the standard output.
System.out.println("Removing optional field. ASN.1 object:");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding.
Response asn1Object3 = new Response();

in = new ByteArrayInputStream(codedAsn1Object);
decoder = new DERDecoder(in);

try {
    asn1Object3.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object3.toString() + "\n");

// Setting optional field.
asn1Object3.setReason("I don't like.");

// Print the ASN.1 object to the standard output.
System.out.println("Setting optional field. ASN.1 object:");
System.out.println(asn1Object3.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object3.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
        // Print the coded ASN.1 object to the standard output.
        buf = new StringBuffer();

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Response asn1Object4 = new Response();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object4.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object4.toString() + "\n");

    }

}
```

Appendix H

OPTIONAL Fields Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type
 * with an OPTIONAL component:
 *
 * Response := SEQUENCE {
 *     yes        BOOLEAN,
 *     reason     IA5String    OPTIONAL }
 */
public class Response extends ASN1Sequence {

    /**
     * Memeber variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1Boolean yes;
    private ASN1IA5String reason;

    /**
     * Constructor without parameters.
     */
    public Response() {

        super(2);

        yes = new ASN1Boolean();
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Constructor for setting a value for the optional component.
     */
    public Response(ASN1Boolean pYes, ASN1IA5String pReason) {

        super(2);

        yes = pYes;
        reason = pReason;

        add(yes);
        add(reason);
    }
```

```
    /**
     * Constructor for leaving the optional component empty.
     */
    public Response(ASN1Boolean pYes) {

        super(2);

        yes = pYes;
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Set and get methods.
     */
    public void setYes(ASN1Boolean pYes) {
        yes = pYes;
        set(0, yes);
    }

    public ASN1Boolean getYes() {
        return yes;
    }

    public void setReason(ASN1IA5String pReason) {
        reason = pReason;
        set(1, reason);
    }

    public ASN1IA5String getReason() {
        return reason;
    }

    /**
     * Remove the optional field.
     */
    public void removeReason() {
        reason = new ASN1IA5String();
        reason.setOptional(true);
        set(1, reason);
    }

}
```

```
import codec.*;
import codec.asn1.*;

import java.io.*;
import java.math.BigInteger;

/**
 * This class shows how a SEQUENCE object with OPTIONAL fields is created,
 * coded and decoded.
 */
public class CodingDecodingOptional {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Response asn1Object1 = new Response(
            new ASN1Boolean(false), new ASN1IA5String("I don't want."));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Response asn1Object2 = new Response();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");


// Removing optional field.
asn1Object2.removeReason();

// Print the ASN.1 object to the standard output.
System.out.println("Removing optional field. ASN.1 object:");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding.
Response asn1Object3 = new Response();

in = new ByteArrayInputStream(codedAsn1Object);
decoder = new DERDecoder(in);

try {
    asn1Object3.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object3.toString() + "\n");

// Setting optional field.
asn1Object3.setReason("I don't like.");

// Print the ASN.1 object to the standard output.
System.out.println("Setting optional field. ASN.1 object:");
System.out.println(asn1Object3.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object3.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
        // Print the coded ASN.1 object to the standard output.
        buf = new StringBuffer();

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Response asn1Object4 = new Response();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object4.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object4.toString() + "\n");

    }

}
```

Appendix I

OPTIONAL Fields Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type
 * with an OPTIONAL component:
 *
 * Response := SEQUENCE {
 *     yes        BOOLEAN,
 *     reason     IA5String    OPTIONAL }
 */
public class Response extends ASN1Sequence {

    /**
     * Memeber variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1Boolean yes;
    private ASN1IA5String reason;

    /**
     * Constructor without parameters.
     */
    public Response() {

        super(2);

        yes = new ASN1Boolean();
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Constructor for setting a value for the optional component.
     */
    public Response(ASN1Boolean pYes, ASN1IA5String pReason) {

        super(2);

        yes = pYes;
        reason = pReason;

        add(yes);
        add(reason);
    }
```

```
    /**
     * Constructor for leaving the optional component empty.
     */
    public Response(ASN1Boolean pYes) {

        super(2);

        yes = pYes;
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Set and get methods.
     */
    public void setYes(ASN1Boolean pYes) {
        yes = pYes;
        set(0, yes);
    }

    public ASN1Boolean getYes() {
        return yes;
    }

    public void setReason(ASN1IA5String pReason) {
        reason = pReason;
        set(1, reason);
    }

    public ASN1IA5String getReason() {
        return reason;
    }

    /**
     * Remove the optional field.
     */
    public void removeReason() {
        reason = new ASN1IA5String();
        reason.setOptional(true);
        set(1, reason);
    }

}
```

```
import codec.*;
import codec.asn1.*;

import java.io.*;
import java.math.BigInteger;

/**
 * This class shows how a SEQUENCE object with OPTIONAL fields is created,
 * coded and decoded.
 */
public class CodingDecodingOptional {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Response asn1Object1 = new Response(
            new ASN1Boolean(false), new ASN1IA5String("I don't want."));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Response asn1Object2 = new Response();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");


// Removing optional field.
asn1Object2.removeReason();

// Print the ASN.1 object to the standard output.
System.out.println("Removing optional field. ASN.1 object:");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding.
Response asn1Object3 = new Response();

in = new ByteArrayInputStream(codedAsn1Object);
decoder = new DERDecoder(in);

try {
    asn1Object3.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object3.toString() + "\n");

// Setting optional field.
asn1Object3.setReason("I don't like.");

// Print the ASN.1 object to the standard output.
System.out.println("Setting optional field. ASN.1 object:");
System.out.println(asn1Object3.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object3.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
        // Print the coded ASN.1 object to the standard output.
        buf = new StringBuffer();

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Response asn1Object4 = new Response();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object4.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object4.toString() + "\n");

    }

}
```

Appendix J

OPTIONAL Fields Demo

```
import codec.asn1.*;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type
 * with an OPTIONAL component:
 *
 * Response := SEQUENCE {
 *     yes        BOOLEAN,
 *     reason     IA5String    OPTIONAL }
 */
public class Response extends ASN1Sequence {

    /**
     * Memeber variables representing the fields of the ASN.1 SEQUENCE.
     */
    private ASN1Boolean yes;
    private ASN1IA5String reason;

    /**
     * Constructor without parameters.
     */
    public Response() {

        super(2);

        yes = new ASN1Boolean();
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Constructor for setting a value for the optional component.
     */
    public Response(ASN1Boolean pYes, ASN1IA5String pReason) {

        super(2);

        yes = pYes;
        reason = pReason;

        add(yes);
        add(reason);
    }
```

```
    /**
     * Constructor for leaving the optional component empty.
     */
    public Response(ASN1Boolean pYes) {

        super(2);

        yes = pYes;
        reason = new ASN1IA5String();
        reason.setOptional(true);

        add(yes);
        add(reason);
    }


    /**
     * Set and get methods.
     */
    public void setYes(ASN1Boolean pYes) {
        yes = pYes;
        set(0, yes);
    }

    public ASN1Boolean getYes() {
        return yes;
    }

    public void setReason(ASN1IA5String pReason) {
        reason = pReason;
        set(1, reason);
    }

    public ASN1IA5String getReason() {
        return reason;
    }

    /**
     * Remove the optional field.
     */
    public void removeReason() {
        reason = new ASN1IA5String();
        reason.setOptional(true);
        set(1, reason);
    }

}
```

```
import codec.*;
import codec.asn1.*;

import java.io.*;
import java.math.BigInteger;

/**
 * This class shows how a SEQUENCE object with OPTIONAL fields is created,
 * coded and decoded.
 */
public class CodingDecodingOptional {

    public static void main(String[] args) {

        // Coding process.

        // Create ASN.1 object.
        Response asn1Object1 = new Response(
            new ASN1Boolean(false), new ASN1IA5String("I don't want."));

        // Print the ASN.1 object to the standard output.
        System.out.println("ASN.1 object: ");
        System.out.println(asn1Object1.toString() + "\n");

        // Create an output stream to which an encoder will write the bytes
        // representing the coded ASN.1 object.
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        // Create encoder.
        DEREncoder encoder = new DEREncoder(out);

        // Byte array to store the bytes representing the coded ASN.1 object
        // before closing the output steam.
        byte[] codedAsn1Object = null;

        try {
            // Order the encoder to read the ASN.1 object and to write the
            // bytes representing the coded ASN.1 object to the output stream.
            asn1Object1.encode(encoder);

            // Store the bytes in the output stream in a byte array.
            codedAsn1Object = out.toByteArray();

            // Close the stream.
            encoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the bytes representing the coded ASN.1 object to the standard
        // output.
        StringBuffer buf = new StringBuffer();
        String octet;
        int i;
```

```
for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding process.

// Create new empty ASN.1 object.
Response asn1Object2 = new Response();

// Create an input stream initialized with the bytes representing the
// coded ASN.1 object from before and a decoder to read it.
ByteArrayInputStream in = new ByteArrayInputStream(codedAsn1Object);
DERDecoder decoder = new DERDecoder(in);

try {
    // Decoder reads the bytes in the input stream and sets the value
    // of the new created ASN.1 object
    asn1Object2.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object2.toString() + "\n");


// Removing optional field.
asn1Object2.removeReason();

// Print the ASN.1 object to the standard output.
System.out.println("Removing optional field. ASN.1 object:");
System.out.println(asn1Object2.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object2.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
// Print the coded ASN.1 object to the standard output.
buf = new StringBuffer();

for (i=0; i<codedAsn1Object.length; i++) {
    octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

    buf.append(" 0x");

    if (octet.length() == 1) {
        buf.append('0');
    }
    buf.append(octet);
}

System.out.println("Bytes representing the coded ASN.1 object:");
System.out.println(buf.toString() + "\n");

// Decoding.
Response asn1Object3 = new Response();

in = new ByteArrayInputStream(codedAsn1Object);
decoder = new DERDecoder(in);

try {
    asn1Object3.decode(decoder);
    decoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}

// Print the new ASN.1 object to the standard output.
System.out.println("New ASN.1 object got by decoding the bytes above:");
System.out.println(asn1Object3.toString() + "\n");

// Setting optional field.
asn1Object3.setReason("I don't like.");

// Print the ASN.1 object to the standard output.
System.out.println("Setting optional field. ASN.1 object:");
System.out.println(asn1Object3.toString() + "\n");

// Coding.
out = new ByteArrayOutputStream();
encoder = new DEREncoder(out);

try {
    asn1Object3.encode(encoder);
    codedAsn1Object = out.toByteArray();
    encoder.close();
}
catch (ASN1Exception e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

```
        // Print the coded ASN.1 object to the standard output.
        buf = new StringBuffer();

        for (i=0; i<codedAsn1Object.length; i++) {
            octet = Integer.toHexString(codedAsn1Object[i] & 0xff);

            buf.append(" 0x");

            if (octet.length() == 1) {
                buf.append('0');
            }
            buf.append(octet);
        }

        System.out.println("Bytes representing the coded ASN.1 object:");
        System.out.println(buf.toString() + "\n");

        // Decoding.
        Response asn1Object4 = new Response();

        in = new ByteArrayInputStream(codedAsn1Object);
        decoder = new DERDecoder(in);

        try {
            asn1Object4.decode(decoder);
            decoder.close();
        }
        catch (ASN1Exception e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }

        // Print the new ASN.1 object to the standard output.
        System.out.println("New ASN.1 object got by decoding the bytes above:");
        System.out.println(asn1Object4.toString() + "\n");

    }

}
```