CODEC Tutorial

http://www.semoa.org/misc/CodecTutorial.pdf

Alberto Sierra
Peter Ebinger
Volker Roth

November 6, 2007

Contents

Chapter 1

Introduction to CODEC

What is CODEC ?

CODEC is a Java package for encoding and decoding data structures defined by the Abstract Syntax Notation One (ASN.1), the standard notation of the Open System Interconnection (OSI) to specify any kind of messages or data structures. The name of the package is an abbreviation for COder/DECoder, which represents the main part of CODEC . Besides there are Java classes representing some complex data structures according to the family of Public Key Cryptography Standards (PKCS).

CODEC is written in Java to make it easier to run the same code on different platforms and is distributed as open source to allow as many developers as possible to use it without having to buy expensive APIs.

Why CODEC has been developed ?

CODEC was developed as part of the SeMoA project (Secure Mobile Agents, http://www.semoa.org) at the Fraunhofer Institute for Computer Graphics. In this project was a mobile agent platform developed with a focus on security that resulted in the need for strong cryptographic functions.

The starting point for CODEC were our problems with inconsistencies of different crypto providers and the low efficiency of existing ASN.1 implementations. No freely distributed Java package was available that met our requirements for the SeMoA project. Therefore, we decided to develop our own package.

The design and the main part of the implementation of CODEC were done by Volker Roth of Fraunhofer-IGD who founded the SeMoA project. The first version of CODEC was ready at the end of 1999 as a well modularized and structured package. The CDC department of the Technical University of Darmstadt contributed to CODEC by implementing CODEC packages for X.509 and PKCS#12 and there is an ongoing collaboration for maintenance, enhancement and extension of CODEC .

CODEC is distributed as open source software to make it available to as many users as possible. You can obtain the package from the SeMoA website (http://www.semoa.org/download/step1.ht downloading the SeMoA bundle in which it is included. The CODEC package is also availabale at http://www.informatik.tu-darmstadt.de/TI/Forschung/FlexiProvider/download.html, the site of a Java crypto-provider developed at the TU Darmstadt using the package.

The Architecture of CODEC

CODEC provides the means to encode and decode basic ASN.1 structures and a library of objects and functions concerning the PKCS family and the X.501/X.509 recommendations. These standards of the RSA Data Security, Inc and the International Telecommunication Union (ITU) are written down in ASN.1.

In the CODEC bundle you can find the following packages:

- `codec.asn1`: Classes that represent the standard ASN.1 types as well as methods for encoding and decoding these data structures.

Besides, there are the following packages corresponding to the PKCS:

- `codec.pkcs1`: Contains the class `DigestInfo` for supporting the encoding within the RSA Cryptography Standard (PKCS #1).
- `codec.pkcs7`: Contains classes representing data types of the Cryptographic Message Syntax Standard (PKCS #7), which describes a general syntax for data that may have cryptography applied to it. PKCS #7 is used for digital signatures and encryption in SMIME email messages.
- `codec.pkcs8`: Contains classes modeling data types of the Private-Key Information Syntax Standard (PKCS #8).
- `codec.pkcs9`: Contains a classes modeling the Selected Attribute Types, described in the PKCS #9 standard.
- `codec.pkcs10`: Contains a class modeling a certification request following the Certification Request Syntax Standard (PKCS #10).
- `codec.pkcs12`: Contains classes modeling a portable format for storing or transporting a user's private keys, certificates, miscellaneous secrets, etc. defined in the Personal Information Exchange Syntax Standard (PKCS #12).

In addition, CODEC contains two packages according to the X.500 recommendation series of the ITU. This series deals with an international and distributed database (directory) to store any kind of information about persons, organizations, communicating application entities, terminals, mailing lists, etc.

- codec.x501: Classes that represent some concepts of the X.501 specification, like X.501 names and attributes
- codec.x509: Classes representing the X.509 Directory-Authentication Framework, based on digital certificates.

Chapter 2

Introduction to ASN.1

It is a known fact that programs running on different platforms may represent internally the same data in different ways. That is, the internal representation (as a string of bits) of e. g. a certain character string, is not the same in each computer. The hardware as well as the software may affect this. For example:

- x86 Intel chips transmit the least significant byte of a word first. This is the so called ”little-endian” system, whereas Motorola chips do just the opposite (”big-endian” system);
- In UNIX systems the end of a line is represented by the line feed ASCII symbol; in DOS systems this is represented with the carriage return and the line feed ASCII symbols;
- in C language the '\0' character is added at the end of a character string.

Since the Internet involves the communication of applications running on diverse platforms it has been necessary to develop and standardize protocols for such applications. These protocols define among other things a set of possible messages that may be sent by the application and in particular how they have to be encoded in a bit string for their transmission.

Since messages may comprise complex data structures a notation had to be developed that defines the structure of the messages and the building blocks of which they are composed. It comprises some general, abstract data types, simple as well as structured ones. To accomplish the task of a protocol (i. e. to define the encoding scheme of the messages) the notation also establishes some encoding rules for the data types declared.

In the early 1980's a notation was developed for defining the messages of the protocols of the Message Handling Systems (MHS), a standard proposed by the International Telegraph and Telephone Consultative Committee (CCITT) for the exchange of electronic mail. The notation and its encoding scheme were machine-independent and could convey complex data structures. In 1984, CCITT standardized the notation under the reference X.409. The notation was totally independent of the MHS system, what lead many groups that were working on standardization of Open System Interconnection (OSI) applications to realize that it could also prove useful to them. As a result, the Abstract Syntax Notation One (ASN.1) was born and standardized in 1987 [1].

Since then ASN.1 has widened its scope out of OSI and benefited from numerous improvements, in which substantial functionalities related to technological changes in telecommunication (high rate data transfer, multimedia environment, frequent service protocol updating, multiple alphabets, e.g. Chinese, etc) were added.

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:

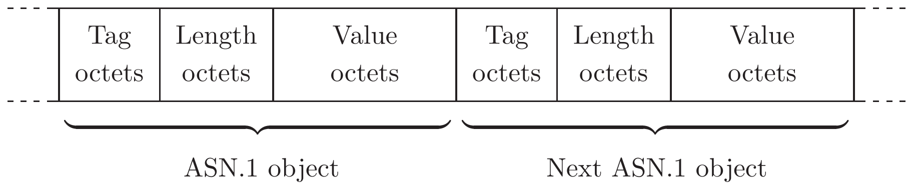
- structures, containing elements of different types (`SEQUENCE`),
- lists of elements of the same type (`SEQUENCE OF`),
- alternative types (`CHOICE`),
- etc.

ASN.1 also provides a notation to declare user defined data types like in the following example:

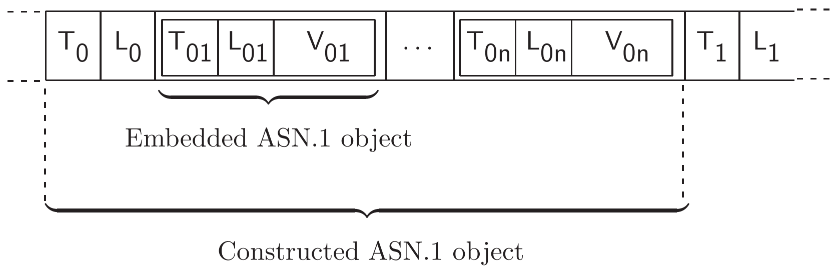
```
EmailMessage ::= SEQUENCE {
    from            IA5String,
    to              IA5String,
    subject         IA5String,
    message         IA5String,
    attachment      OCTET STRING,
    date            GeneralizedTime,
    urgent          BOOLEAN }
```

ASN.1 provides sets of encoding rules which determine how each ASN.1 data type is translated into a bit stream. The original encoding rules of ASN.1 are the Basic Encoding Rules (BER). But with time new encoding rules have been designed, the Distinguished Encoding Rules (DER), the Packed Encoding Rules (PER) etc. to satisfy application dependent aspects [2]. The principle of BER is that each ASN.1 value is encoded following the so called "TLV" pattern: tag, length and value. This means that every value receives a header that is usually two bytes long. The first byte indicates the ASN.1 type of the value, the so called tag, while the second indicates the length in bytes of the encoded value (without the header). The following bytes represent then the encoded value.

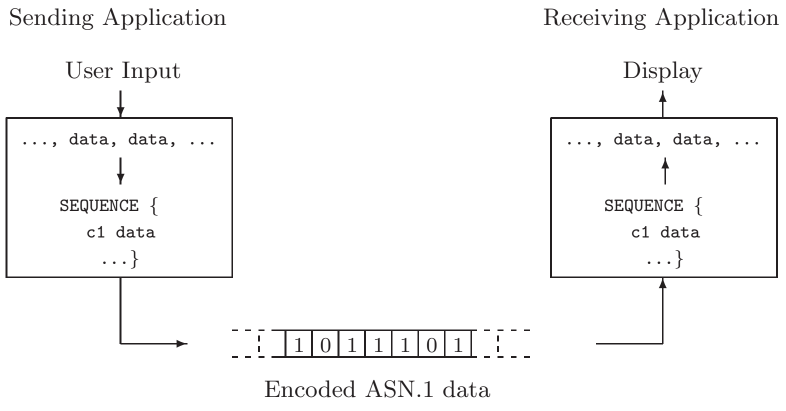
Its encoding depends on its type. Each ASN.1 type has its own encoding rules. For decoding the transmitted bit stream the receiving party first reads the the tag and the length of the encoded value, so that it knows which bytes represent the encoded value and how to decode them.



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:



Finally a figure is presented that shows a general scenario in which ASN.1 is used for data exchange and explain in a few words what happens in each stage of the transmission:



Lets assume the data that has to be transmitted is entered by a user. The data is packed in a message which as described by the respective protocol in ASN.1 and then it is encoded. The receiving application reconstructs the ASN.1 message from the bit stream. Then the data is extracted from the ASN.1 message and can be then processed further (e. g. displayed).

For additional documentation regarding ASN.1 we recommend to visit http://www.oss.com/asn1/booksintro.html, where you can find the following online-books:

- Olivier (D.). *ASN.1 Communication between Heterogeneous Systems.*
- Larmouth (J.). *ASN.1 Complete.*

Chapter 3

Simple Coding/Decoding Example

As mentioned in the previous chapter ASN.1 was designed to facilitate transactions between heterogeneous systems. In this chapter some Java code is shown that simulates such a transaction in a very simple way. The use of some of the basic classes of the CODEC package is demonstrated by performing the following tasks:

1. first a Java object is created that represents some data modeled by an ASN.1 type;
2. then the ASN.1 encoding rules are applied to that Java object;
3. and finally the encoded data is decoded obtaining a Java object representing the identical data as the one created before encoding it.

When using CODEC it is important to distinguished between the **two modes of operation**. The Java objects that are created in step one and step three have two different purposes, the first one is for encoding and the second one is for decoding.

- In step one an **object for encoding** is created, i.e. data stored in standard Java objects that is to be transformed into ASN.1 objects for encoding. Therefore a constructor is used that takes a standard Java object as argument.
- In step three the data is stored in an ASN.1 encoded byte array and a **Java object for decoding** is created. Therefore the default constructor for decoding is used, the constructor that does not take any arguments.

So let us start with the first step. Imagine some basic data has to be modeled, e. g. a character string, with an ASN.1 type. There are several standard ASN.1 types that represent character strings. They differ mainly in that they comprehend different sets of characters. The `IA5String` type is one of them. The next ASN.1 expression declares a value of this type representing the character string "Hello World":

```
asn1Object IA5String ::= "Hello World"
```

With the use of the CODEC classes such a statement would be implemented in Java as follows:

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

As you can see a **constructor for encoding** is used, i.e. which takes a standard Java object as argument (`java.lang.String`). `ASN1IA5String` is the Java class in the `codec.asn1` package that represents the `IA5String` type. It provides the methods for encoding and decoding this ASN.1 type following the ASN.1 coding rules.

In the `codec.asn1` package you can find the Java classes that represent the standard ASN.1 types as shown in table 3.1.

ASN.1 Type	**Java Class in 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 3.1: ASN.1 types and their respective Java classes in the CODEC package

Encoding

ASN.1 supports several coding 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 as we mentioned in the first chapter. 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 instance created before:

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

First an output stream has to be instantiated. The `DEREncoder` instance (created in the next line) writes the encoded data to this stream. Finally the `encode(Encoder)` method of the Java instance to be encoded is called. During this call the encoder reads the `java.lang.String` instance stored within `asn1Object` and writes the bytes representing the encoded `IA5String` value to the output stream. The parameter passed to the method has to implement the `Encoder` interface. This is an interface of the CODEC package and denotes any class that may perform the encoding task. Till now the CODEC package only offers the `DEREncoder` class for this purpose but in the future further classes may be included in the package implementing other existing encoding rules.

With the following statement

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

the bytes in the output stream can be stored in a byte array. The byte array will be needed later for demonstrating the decoding process. Printing its contents in hexadecimal representation shows 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 previous 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 encoded value. And each of the remaining 11 bytes represents one character of the "Hello World" string.

Decoding

In this section it is shown how to decode the byte array just produced, i. e. how the "Hello World" character string can be reconstructed by creating a new `ASN1IA5String` instance and reading the ASN.1 encoded bytes array. For this purpose first a new empty `ASN1IA5String` instance is created:

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

As you can see a **constructor for decoding** is used, i.e. which does not take any arguments. As next an input stream containing the encoded data has to be created. As you remember this data was stored in the byte array `encodedAsn1Object`. The input stream has then to be passed to a `DERDecoder` class instance, which decodes its contents.

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

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

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

Processing this statement the decoder reads 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 CODEC instance from which the `decode(Decoder)` method has been called. Otherwise an exception is be thrown. Then the next byte is read, indicating the number of bytes that follow which represent the encoded data value. Then the decoder decodes the data bytes representing the encoded character string following the decoding rules for the `IA5String` type. The decoded data is stored as a `java.lang.String` object in `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.

Note: If you remember, before decoding a Java instance of a certain class is initialized assuming to know the ASN.1 type of the arriving data. This might surprise, but it reflects the normal behavior of communicating applications, since they use to follow some kind of dialog or protocol for performing their data transactions. That is, they know which kind of data should be arriving at each state of the communication.

Alternative decoding procedure

The `DERDecoder` class provides also a `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 instance of `ASN1Type`. This is an interface that is implemented by all classes of the CODEC package that represent an ASN.1 type, e. g. the `ASN1IA5String` class. You can store the data of an `ASN1Type` instance in a file or print it to the standard output, however if you want to determine the concrete class of the Java value stored within this instance, you have to check all 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, it is possible to decode the data this way, but further processing is quite complicated, since its type is not known.

On the other hand, this decoding procedure can be very useful for testing and debugging purposes when it is not exactly known in advance what kind of an application receives.

A complete Java application implementing encoding and decoding as it has been shown in this chapter can be found in Appendix A (Coding/Decoding Example).

Chapter 4

How to implement SEQUENCE types

The `SEQUENCE` type models data structures which describe some information stored in several components that may be of different data types. Such structures, very common in programming languages, can be used for describing some object or instance, storing the data about its attributes in the components of the data structure.

As next you can see an example of an ASN.1 declaration of a `SEQUENCE` type:

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

As you can see, the data structure defined offers a fixed list of components, each one denoted by an identifier (a name describing the component) and its corresponding ASN.1 type.

As next it is shows how to implement such a `SEQUENCE` type in Java using the CODEC package. Therefore a separate Java class is written that represents the ASN.1 type. Let us first define a simpler `SEQUENCE` type with only two components to avoid the resulting Java code to be too extensive:

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

In general, every Java class that represents a `SEQUENCE` type has to extend the `ASN1Sequence` class of the `codec.asn1` package. The first lines of code are the following:

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

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

The next step is to declare member variables that represent the components of the `SEQUENCE` type:

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

The member variables have to match the Java class that represents the ASN.1 type of the corresponding component in the `SEQUENCE` declaration.

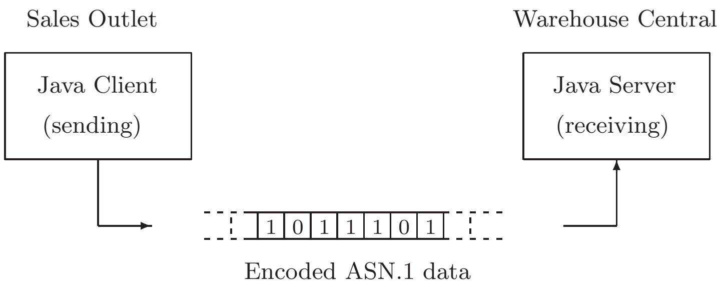
As next it is shown how the constructorso are implemented . Basically two different constructorswe have to be written for a class representing an ASN.1 type:

- one with parameters for encoding the data received from the application.
- one without parameters for creating an empty object ready to be filled with Java values obtained by decoding an ASN.1 encoded bit stream.

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

The constructor for encoding

Before we go on let us look a scenario in which the implementation of ASN.1 may be used.



Let us assume there are two applications running on different machines that perform the following tasks:

- The client application runs at a sales outlet and sends orders to the warehouse with the type and quantity of each product that is needed.
- The server application runs at a central warehouse and collects the orders of the clients which could be programmed in different languages and run on different operating system.

Before two applications can exchange data a set of ASN.1 types has to be declared that defines all possible data structures that may be transmitted. It is up to each party to choose a programming language and to implement the corresponding code to represent these ASN.1 types (e. g. as Java classes containing basic Java values such as character strings, integers, etc.) and to provide the necessary mechanisms for data encoding and decoding according to the ASN.1 coding rules.

The ASN.1 `Order` type defined above is implemented in a Java class to demonstrate how data can be transmitted between a client and the warehouse server. The following lines of code belong to the implementation of the client application:

```
...
// Variables that represent the data to be transmitted.
String productName;
int neededQuantity;
Order order;
...
(Set the variables e. g. by reading a database.)
...
// Create instances of the Order class with the data read
// from the database.
order = new Order(productName, neededQuantity);
...
(Encode the instances just created.)
...
(Transmit the encoded data.)
...
```

As you can see a `Order` object is instantiated with two Java values as parameters. The constructor that is used is the one for data encoding which is implemented as follows:

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

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

        /* Add the member variables to this 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 instruction of the constructor shown above `super(2)` calls the super class constructor and passes the number of member variables as argument, i. e. the number of components the corresponding ASN.1 type has. According to this value the adequate initial memory is allocated, in this case for two objects. The instructions in the following lines initialize the member variables with the parameter values and add them to the `Order` object calling the `add(Object)` method inherited from the `ArrayList` class.

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

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

The disadvantage of such a constructor declaration becomes evident when we look at a piece of code as shown 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.
product = new Order(
    new ASN1IA5String(productName1),
    new ASN1Integer(neededQuantity1));
...
(Encode the Order instances.)
...
(Transmit the byte arrays containing the encoded data.)
...
```

As you can see, the CODEC packages have to be imported in the implementation of the client application and also all application data has to be wrapped in CODEC classes before they are passed to the constructor. This contradicts the basic rule that the CODEC packages, 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, but they should stay hidden from the application developers otherwise.

The constructor for decoding

Now let us look at the server which expects to receive ASN.1 encoded data as defined by the `Order` type. 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 and store decoded values in the
 empty Order instance just created. From now on the decoded
 data is accessible as basic Java values from the Order
 instance.)
```

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 the subclass of `ASN1Sequence` has to be initialized to perform this task:

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

        /* Initialize the member variables ready for decoding
         * each component of the SEQUENCE.
         */
        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 for decoding, i.e. without parameters.

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

Coding/Decoding

Now let us take a detailed look at the encoding/decoding steps. In the previous pieces of code the actual implementation of the encoding and the decoding process were ommitted. In the following two methods for this purpose are shown, `getEncoded()` and `decode(byte[])`.

The first method is called at the client side. It reads and encodes the data stored in the member variables of the object and returns a byte array representing an encoded `Order` type value..

```
public byte[] getEncoded() throws ASN1Exception, IOException
{
    ByteArrayOutputStream out;
    DEREncoder encoder;
    byte[] encodedAsn1Object;

    /* Initialize an output stream to which the encoded data will
```

```
     * be written.
     */
    out     = new ByteArrayOutputStream();

    /* Initialize encoder instance with this output stream.
     */
    encoder = new DEREncoder(out);


    /* Encoder reads the data stored in the member variables of
     * this class and encodes it writing the output to the output
     * stream.
     */
    this.encode(encoder);

    /* Store the data in the output stream in a byte array. This
     * array will be returned by this method.
     */
    encodedAsn1Object = out.toByteArray();

    /* Close the stream.
     */
    encoder.close();

    /* Return the encoded data.
     */
    return encodedAsn1Object;
}
```

The `decode(byte[])` method takes as argument a byte array representing an encoded `Order` type value. The bytes of each component are decoded and stored in the corresponding member variables of the `Order` class.

```
public void decode(byte[] encodedData)
throws ASN1Exception, IOException
{
    ByteArrayInputStream in;
    DERDecoder decoder;

    /* Initialize input stream containing the encoded data.
     */
    in      = new ByteArrayInputStream(encodedData);

    /* Initialize decoder instance with this input stream.
     */
    decoder = new DERDecoder(in);

    /* Decode the data in the input stream and stored it in the
     * member variables of this class that represent the
     * components of the corresponding ASN.1 type.
```

```
         */
        this.decode(decoder);

        /* Close the stream.
         */
        decoder.close();
    }
```

As you can see both methods follow the same steps that were shown in the previous chapter. The code shown here is quite useful because it works in all class that implements `codec.asn1.ASN1Type`, as e.g. any class that extends one of the ASN.1 standard types shown in table 3.1 in the previous chapter. You can copy and paste them and they perform the same task correctly, you do 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` structure. 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 the last two bytes represent the number 152.

Set and get methods

The set and get methods can 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();
    }
```

It has to be taken 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 (cf. chapter 9).

In the set methods the member variables receive new values that replace the old values that have been previously included to the list represented by the `Order` class. This is done by calling the `set(int, Object)` method of the super class `java.util.ArrayList`.

Set methods are optional. They were 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.

Note: The parameters of the set methods as well as the return values of the get methods should be represented by standard Java values and not by CODEC types since these methods are used by other classes.

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

Chapter 5

How to implement SEQUENCE OF types

The ASN.1 `SEQUENCE OF` type represents a list 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 is demanded, but rather a list of several orders. For this case the definition of a `SEQUENCE OF` type is very helpful since it can represent a list of orders with all the quantities needed. A declaration of such a type is as simple as the following:

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

To implement a Java class that represents a `SEQUENCE OF` type the `ASN1SequenceOf` class is extended:

```
import codec.asn1.*;

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

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

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

It declares what type of elements are stored in this object, here instances of the `Order` class. This class refers to the Java implementation of the ASN.1 type of the elements of the `ASN1SequenceOf` structure. In the source code of this class (see Appendix C - `SEQUENCE OF` Example) you can find an additional constructor which takes an array of `Order` objects as a parameter.

Methods for encoding and decoding (`byte[] getEncoded()` and `decode(byte[])`) can be implemented exactly 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 the `add(Object)` method, with which elements can be added to it. Once the `ASN1SequenceOf` instance is filled with data it can be encoded and decoded as shown before.

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 6

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 equal to the `SEQUENCE` and `SEQUENCE OF` types except that the elements they contain do not necessarily have to be encoded in the order established by the corresponding ASN.1 declaration.

In the CODEC package the implementation of the DER encoding of the corresponding classes `ASN1Set` and `ASN1SetOf` is the same as the one of the `ASN1Sequence` and `ASN1SequenceOf` classes, which is not standard compliant. The encoding of the elements of a `SET` type object has to be in ascending lexicographic order (i. e. comparing the DER encodings of the elements bit by bit).

This encoding procedure was not implemented because it requires much more processing time and storage space because it contradicts the slim and efficient design of CODEC which encodes all ASN.1 structure sequentially. Sorting would require storage and comparison of data seqments that can become arbitrary large.

Chapter 7

How to implement CHOICE types

A `CHOICE` type provides a list of different alternative data types it may stand for. It is used for example when a certain information can be modeled in different ways. The next ASN.1 declaration describes usual payment methods in online transactions:

```
    Payment-method ::= CHOICE {
        debit           [0] Debit,
        credit-card     [1] Credit-card }

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

    Debit ::= SEQUENCE {
        ... }
```

The meaning of the numbers within brackets will be explained in chapter 10 ”How to implement tagging”.

For demonstrating a sample implementation of a Java class that represents a `CHOICE` type let us define a simple ASN.1 type:

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

The corresponding class has to extend the `ASN1Choice` class, so the first lines of code are:

```
import codec.asn1.*;

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

For each possible data type of the CHOICE has to be declared a member variable of the corresponding class:

```
    private ASN1Null acknowledgment_;
    private ASN1Integer errorCode_;
```

A Java class representing a CHOICE type should only be used for decoding, never for encoding. When ASN.1 data has to be encoded the data type of the structure that is encoded is obviously known and we can simply choose this alternative from the possible data types and use the corresponding Java class. There is no point in wrapping the data in a CHOICE object since there is not a separate encoding tag for a CHOICE type. This has also to be taken into account when the CHOICE type is enclosed in a structured type. In the following an example is shown.

The constructor for decoding the CHOICE type is implemented as follows:

```
    public Response()
    {
        /* Allocate memory for storing two objects (for each
         * choice).
         */
        super(2);

        /* Initialize the member variables ready for decoding
         * each of the possible choices.
         */
        acknowledgment_ = new Acknowledgment();
        errorCode_     = new ErrorCode();

        /* Store the possible choices.
         */
        addType(acknowledgment_);
        addType(errorCode_);
    }
```

As in the implementation of a SEQUENCE type the call of the super constructor (super(2)) just optimizes the initial memory allocation. The number that is passed represents the number of alternative choices the corresponding CHOICE type offers. The following calls initialize the member variables and adds them using the addType(ASN1Type) method inherited from ASN1Choice.

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 used for encoding, but an object of one of the possible types should be created instead. 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 shows the following bytes which are the encoded representation a `NULL` object:

```
    0x05 0x00
```

Note: An ASN.1 encoded `NULL` object consists of only two bytes, tag value 5 and length 0. This is an exception of the general TLV principle (cf. chapter 2).

Since it is expected that a `Response` object is either a `NULL` or a `INTEGER` data structure, these bytes can be decoded with:

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

The `decode(Decoder)` method of the `ASN1Choice` class checks the different types of the alternative choices. If one of them matches the ASN.1 type of the byte array to be decoded, the value is decoded and stored in the corresponding object.

You can find the complete sources from which the pieces of code shown in this chapter have been extracted in Appendix D (`CHOICE` Example).

Chapter 8

How to implement ENUMERATED types

ENUMERATED types are used for declaring a data type which comprise only a limited number of different values. Let us take a look at the following example:

```
    ResponseStatus ::= ENUMERATED {
        successful          (0),  -- understood request
        malformedRequest    (1)   -- malformed request }
```

Each possible value is associated with an integer value. Only this value is encoded, therefore 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 side the integer value is decoded and interpreted according to the given ASN.1 type declaration. Within communicating applications the ENUMERATED type is commonly used for describing the state of a system or for error codes.

Now let us look at the Java implementation of the ResponseStatus type. For this purpose the ASN1Enumerated class is subclassed:

```
import codec.asn1.*;

public class ResponseStatus extends ASN1Enumerated
{
...
```

For each possible value that the ENUMERATED type may represent is a constant defined:

```
    public static final int SUCCESSFUL        = 0;
    public static final int MALFORMED_REQUEST = 1;
```

These values are declared as public and static so that they can not be altered and not be accessed from any other class.

A constructor for encoding may be implemented as follows:

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

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

It filters out inappropriate arguments (integers) according to the ASN.1 type declaration.

The constructor for decoding is implemented as simple as:

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

And finally the get method:

```
    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` objects.

You can find the whole source code in Appendix E (`ENUMERATED` Example).

Chapter 9

How to implement optional fields

ASN.1 offers the possibility to set components within a `SEQUENCE` type as optional. Such components are not encoded, if they do not contain any data. This way the overhead of unnecessary data encoding and transmission can be avoided, especially when the optional component of the `SEQUENCE` is itself a complex ASN.1 type.

The next ASN.1 declaration describes the content of a web form for registration at an online shop or something similar, in which some fields must be filled out and others, denoted with the `OPTIONAL` clause, may 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.

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

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

As in chapter 4 ”How to implement a `SEQUENCE` type” the first lines of code are:

```
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, two constructors for encoding can be implemented: one setting values for all fields (including optional ones) and another setting values only for 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, the member variables are not added to the class as it has been shown in chapter 4. If an ASN.1 `SEQUENCE` contains `OPTIONAL` elements we recommend to implement a separate function, like the one shown next, in which the structure is re-initialized and the components are added right before encoding.

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

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

The `map()` method has to be called always before encoding. Therefore the inherited `encode(Encoder)` method has to be overwritten:

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

The constructor for decoding is implemented as follows:

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

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

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

Here all elements of the `SEQUENCE` type, also the optional ones, are initialized and added to the class. The member variables representing the optional components have to be marked as optional using the `setOptional(true)` method. This method sets or removes the optional flag. It belongs to `ASN1Type`, an interface of the CODEC package that is implemented by all classes that represent an ASN.1 type.

The decoder always checks this flag before decoding a component of a `SEQUENCE` type. If set to true the compontent is optional and the decoder does not throw an exception if no data was delivered for this component. If set to false the decoder requires data of the same type as the component to be decoded, otherwise an exception is thrown. In the flow chart in figure 9.1 you can see in more detail the steps for the decoding of a component of a `SEQUENCE` type.

Now let us follow the procedure in a concrete example. Therefore an object of `Person` class is created 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();
```

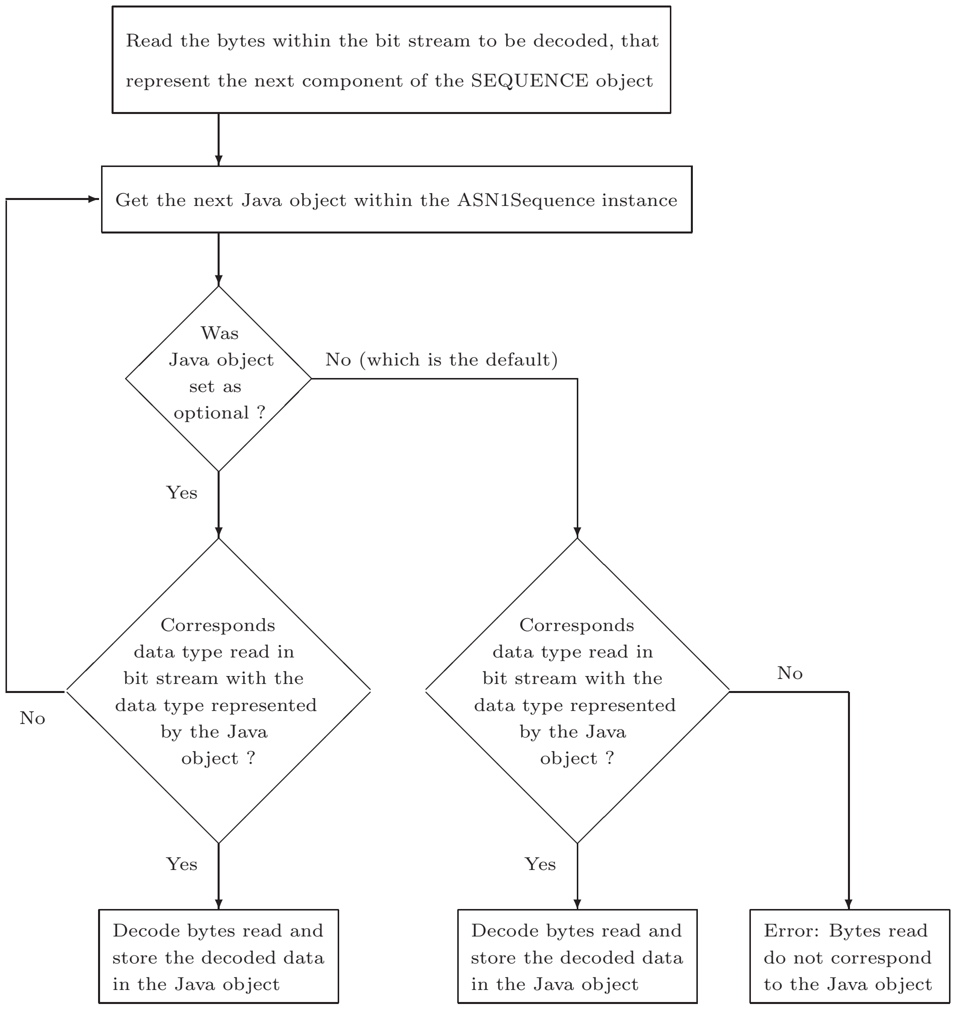

Read the bytes within the bit stream to be decoded, that represent the next component of the SEQUENCE object
Get the next Java object within the ASN1Sequence instance
Was Java object set as optional ?
No (which is the default)
Yes
Corresponds data type read in bit stream with the data type represented by the Java object ?
No
Corresponds data type read in bit stream with the data type represented by the Java object ?
No
Yes
Yes
Decode bytes read and store the decoded data in the Java object
Decode bytes read and store the decoded data in the Java object
Error: Bytes read do not correspond to the Java object

Figure 9.1: Flow chart representing the decoding of a component within a `SEQUENCE`

The encoded bytes of the object just created has 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 represents the ASN.1 type of the second component (`IA5String`) since the first component had not been encoded.

For decoding the byte array the following code is used:

```
Person newAsn1Object = new Person();
ByteArrayInputStream in = new ByteArrayInputStream(encodedData);
```

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

The decoder goes through the following steps to decode the bytes:

1. It checks 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 parameters, which had been called for creating `newAsn1Object`.
2. The decoder compares 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 ignores this member variable assuming that no data to be decoded was provided for it.
4. Then the next member variable is 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 are decoded and stored in this member variable.

It just remains to show the set and get methods for ASN.1 structures with optional elements:

```
    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();
    }
```

In this case the set methods `set(int, Object)` of the upper class `ArrayList` is not called for replacing the old values with the new one as in chapter 4. In contrast, the new values are added just before encoding using the `map()` method as shown before.

The implementation of a function like the following may make sense to remove a previously set optional component before encoding:

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

You can find the whole sources to this chapter in Appendix F (`OPTIONAL` Fields Example).

Chapter 10

How to implement tagging

Consider the following ASN.1 type declaration:

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

and two possible values for the components of the ASN.1 `SEQUENCE`:

```
    person1 Person ::= {
        title    "",
        name     "Peter" }

    person2 Person ::= {
        title    "Sir",
        name     "Peter" }
```

The values of the bytes of the encoded data structures would the following:

```
person1:
 0x30 0x07 0x16 0x05 0x50 0x65 0x74 0x65 0x72
  T0   L0   T01  L01  P    e    t    e    r

person2:
 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
```

This would result in a conflict during decoding. As we saw in the last chapter the decoder decodes the components of a `SEQUENCE` one after the other. If a component is optional and the ASN.1 type of the bytes to be decoded next do not match the ASN.1 type of the optional component, the decoder assumes that no data was encoded for it. But if the ASN.1 type of the bytes to be decoded next match the ASN.1 type of the optional component, then the bytes are decoded and assigned to the optional component.

Following this procedure, when decoding the bytes corresponding to `person1`, the decoder would wrongly assign the bytes representing the character string ”Peter” to the `title` component, since both are of the same ASN.1 type.

This is an example for a case where the tagging procedure has to be applied. It works as follows: the components whose encoded data may be wrongly assigned to other components at decoding time, have to be marked in the ASN.1 type declaration with a tag value in square brackets. For example,

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

This way the encoder gives these components a new header with a special tag value, which is computed using the number in square brackets. The resulting header is unique within this `SEQUENCE`. The decoding party, also knowing the ASN.1 declaration, will be able to assign the encoded data to the corresponding components, since these can be unambiguously identified with the new headers.

There are two different ways of tagging a component: explicitly and implicitly. Roughly they differ in that the implicit tagging mode offers a more compact encoding by simply overwriting the original header while explicit tagging wraps the original data structure in an additional tagging structure. Explicit tagging can always be applied whereas we will see later some cases in which implicit tagging can not be used.

Basically there are two different situations in which tagging has to be used:

- in `SEQUENCE` types in which there are optional and not optional components or only optional components of the same ASN.1 type, for example:

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

    MySequence2 ::= SEQUENCE {
        component1       [0] MySequence3   OPTIONAL,
        component2       [1] MySequence4   OPTIONAL }

    MySequence3 ::= SEQUENCE {
        ... }

    MySequence4 ::= SEQUENCE {
        ... }
```

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

```
    MyChoice ::= CHOICE {
        choice1         [0] MySequence5,
        choice2         [1] MySequence6,
        choice3             IA5String }
```

Explicit tagging

Explicit tagging is generally the default, i. e. to indicate that this kind of tagging should be applied just a number between square brackets has to be written before the ASN.1 type of the component 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 `person2` object in encoded form while implementing explicit tagging:

```
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 the 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. Generally the value of an explicit tag can be computed as follows:

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

Since the values of the usual ASN.1 tags range from 0 to 30 (each representing a different standard ASN.1 type), they do not conflict with the values of tagged components.

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. for tagging a component which is a `CHOICE` type. The reasons for this will be explained later. To declare that this kind of tagging should be applied, the keyword `"IMPLICIT"` is to be inserted in the ASN.1 type declaration after the numbers within square brackets:

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

Now let us see how the `person2` object defined before would be encoded using this kind of tagging:

```
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 no new header is added, but the original header of the tagged component is overwritten with a new one (see bytes 3 and 4). The decoder has to get the information about the ASN.1 type of this component from the type declaration. The values of implicit tags are computed as follows:

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

As said before, this way of tagging offers a more compact encoding but it cannot be applied for tagging `CHOICE` types. Look at 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 can not tell which choice has been set in the `CHOICE` object at decoding time. In cases like this explicit tagging has to be used.

Java implementation

Now let us see how the ASN.1 type presented before 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 are 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 tag values given in the ASN.1 declaration, are declared:

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

The constructors for encoding data is 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 added in the `map()` function, which always has to be called before an encoding is performed:

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

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

As you can see the member variable representing the tagged component is not added to the class directly, but enclosed in an instance of the `ASN1TaggedType` class. The first argument (`TITLE_TAG`) represents the tag value 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 is applied. If the ASN.1 declaration indicated implicit tagging, a `false` value would have to be passed here instead.

The constructor for decoding is implemented 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 `map()` 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 explained.

During the encoding, this flag does not need to be set. As in the last chapter, optional components are simply not added to the class and hence not encoded if they are empty (see `map()` function).

You can find the whole sources to this chapter in Appendix G (Tagging Example).

Chapter 11

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 automatically assumes the default value if no data was sent for the component.

In general, `DEFAULT` elements work exactly as `OPTIONAL` for which a default value is assumed if they are not there.

Let us see how this feature can be implemented in Java. The first step could be to define a Java constant for the default value:

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

Since the `country` component is explicitly tagged we define a constant 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 `map()` and the `encode(Encoder)` functions work exactly as with `OPTIONAL` components:

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

    public void encode(Encoder enc)
    {
        map();
        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 DEFAULT_COUNTRY;
    }
    else
    {
        return country_.getString();
    }
}
```

You can find the whole sources to this chapter in Appendix H (`DEFAULT` Values Example).

Chapter 12

How to implement ANY types

The `ANY` type serves to model a variable whose type is unspecified. It has disappeared since 1994 from the ASN.1 standard and its use is strongly inadvisable. It has been replaced by the introduction of the concepts of parameterization and the information object class, particularly the `TYPE-IDENTIFIER` class.

The `ANY` 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. It has often been the type of a component within a `SEQUENCE` type conditioned by the value of some other component of the same `SEQUENCE`. This semantic link could be indicated by the `DEFINED BY` clause. The next example shows such a case:

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

The component referenced after the `DEFINED BY` clause had to be of type `INTEGER` or `OBJECT IDENTIFIER`.

Unfortunately ASN.1 provided no way of formally indicating what the possible values of this semantic link could be. The standard recommended defining an association list by means of comments within the type declaration. For the type `ErrorMessage` above, one could have written

```
--          oid        | ASN.1 type of parameter
-- ===================|========================
-- 2.1.0.2.0           | NULL
-- 2.1.0.2.2           | INTEGER
-- 1.2.276.1000.8.5.89 | ErrorParameterType1
```

with `ErrorParameterType1`

```
ErrorParameterType1 := SEQUENCE {
    critical      BOOLEAN,
    reason        IA5String }
```

The ASN.1 `OBJECT IDENTIFIER` type represents a list of numbers and it may classify any kind of objects. It is intended to be used for giving each object which may be of universal interest a unique value, e. g. for international standards. In the CODEC package the `ASN1ObjectIdentifier` class stores its value as a character string consisting of numbers separated by dots.

The component referenced after the `DEFINED BY` clause has to always be declared before the `ANY` component. This way its value is encoded and decoded first. This way the specific ASN.1 type of the `ANY` component can be determined (as necessary) before it is decoded.

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

The member variable representing the `ANY` component has to be declared as `ASN1Type` since it can represent an arbitrary ASN.1 type.

```
private ASN1ObjectIdentifier oid_ = null;
private ASN1Type parameter_ = null;
```

Remember that the `ASN1Type` interface is implemented by all classes of the CODEC package that represent an ASN.1 type, and hence by all their subclasses.

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

```
public ErrorMessage(String oid, Object obj)
{
    super(2);

    oid_ = new ASN1ObjectIdentifier(oid);

    if (oid.equals("2.1.0.2.0"))
    {
        parameter_ = new ASN1Null();
    }
    else if (oid.equals("2.1.0.2.2"))
    {
        parameter_ = new ASN1Integer((BigInteger)obj);
    }
    else if (oid.equals("1.2.276.1000.8.5.89"))
    {
        parameter_ = obj;
    }

    add(oid_);
    add(parameter_);
}
```

Here the implementation of the constructor for decoding:

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

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

        add(oid_);
        add(parameter_);
    }
```

As you can see the member variable `parameter_` is initialized calling an `ASN1OpenType` constructor. This class has the capability to decode any class representing an ASN.1 type. In this case the constructor receives an instance of the `ErrorResolver` class as argument.

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

As you can see the `ErrorResolver` instance passed to the `ASN1OpenType` class receives itself as argument a reference to the member variable `oid_`. This reference will be needed later for decoding the component `parameter`.

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

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

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

This class needs a reference to the member variable `oid_` of the `ErrorMessage` class. The value of this variable determines which Java class the `ErrorResolver` class has to provide for decoding the component `parameter`.

```
private ASN1ObjectIdentifier oid_ = null;
```

The reference to this member variable `oid_` of the `ErrorMessage` class is passed to the `ErrorResolver` in its constructor which is called in the decoding constructor of the `ErrorMessage` class.

```
public ErrorResolver(ASN1ObjectIdentifier oid)
{
    oid_ = oid;
}
```

`Resolver` classes have to implement the `resolve(ASN1Type)` method. It returns an instance of a class with which the `ANY DEFINED BY` component, in our case `parameter`, should be decoded, depending on the `DEFINED BY` value, in our example `oid`. This method is called from the `decode(Decoder)` method of the `ASN1OpenType` class.

```
public ASN1Type resolve(ASN1Type caller) throws ResolverException
{
    if (oid_.toString().equals("2.1.0.2.0"))
    {
        return new ASN1Null();
    }
    else if (oid_.toString().equals("2.1.0.2.2"))
    {
        return new ASN1Integer();
    }
    else if (oid_.toString().equals("1.2.276.1000.8.5.89"))
    {
        return new ErrorParameterType1();
    }
    else throw new ResolverException();
}
```

It has to be noticed that, when the `ErrorResolver` class is constructed, the member variable `oid_` of the `ErrorMessage` has just been initialized with an empty default value. However the `resolve(ASN1Type)` method is called right before the bytes corresponding to the `parameter` component are decoded, i. e. after the component `oid` has been decoded. At this point the member variable `oid_` has the correct value and the ASN.1 type of the component `parameter` can be determined, and therefore it can be decoded.

You can find the whole sources to this chapter in Appendix I (`ANY DEFINED BY` Example).

Chapter 13

The OID Registry

A registry serves as a mapping of `OBJECT IDENTIFIER` values to ASN.1 types for resolving `ANY DEFINED BY` values.

On one hand, there is a `OIDRegistry` class In the CODEC package, which permits to add or remove other `OIDRegistry` instances, so that a tree of registries can be build. This class serves to represent a top level registry. On the other hand there is the `AbstractOIDRegistry` class which provides mechanisms to look up for a specific ASN.1 type registered in an instance of this class. This class represents a local registry. If you want to implement your own local registry class you have to extend this class:

```
import java.util.Map;

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

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

```
// Store each OID as an array of integers.
static final private int[][] oids_ =
{
    {2,1,0,2,0},
    {2,1,0,2,2},
    {1,2,276,1000,8,5,89}
};

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

This way the `new ASN1ObjectIdentifier("2.1.0.2.0")` object delivers the `ASN1Null` class, etc. As you can see there is also the possibility to add ASN.1 types as character strings (see ”ErrorParameterType1”). You can declare a prefix indicating the package of the classes given as strings.

```
static final private String prefix_ = "codec.examples.oidRegistry.";
```

You also have to 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();
```

The constructor for this class should map the `OBJECT IDENTIFIER` values and the corresponding ASN.1 types:

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

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

Two more methods have to be implemented by your own registry class:

```
protected Map getOIDMap()
{
    return map_;
}

protected String getPrefix()
{
    return prefix_;
}
```

These methods are called when an ASN.1 type is looked up in this registry, during the decoding of an `ANY DEFINED BY` value.

Now let us see how `ErrorMessage`, the `ANY DEFINED BY` type defined in the last chapter, may be implemented using a registry. There is only one change which is 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_);
}
```

This time a `SampleOIDRegistry` object is passed to the `ASN1OpenType` constructor instead of a `Resolver` as in the last chapter. The implementation of this constructor is as follows:

```
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 can deliver the appropriate class for decoding the `ANY DEFINED BY` component. In this case the resolver is created instantiating the `DefinedByResolver` class with an `OIDRegistry`. This class belongs to the CODEC package and implements the `Resolver` interface. It provides an appropriate Java object for decoding the `ANY DEFINED BY` component, by querying the registry with which it is initialized.

You can find the whole sources to this chapter in Appendix J (OIDRegistry Example).

Appendix A

Coding/Decoding Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.simpleExample;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Type;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows how a simple ASN.1 value is created, encoded and decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:05 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
```

```
{
    ASN1IA5String asn1Object;
    ByteArrayOutputStream out;
    DEREncoder encoder;
    byte[] encodedAsn1Object;
    StringBuffer buf;
    String octet;
    ASN1IA5String newAsn1Object;
    ByteArrayInputStream in;
    DERDecoder decoder;
    ASN1Type asn1Type;


    /* Create ASN.1 object.
     */
    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();

    /* Encoding process.
     */
    /* Initialize an output stream to which the encoded data will be
     * written.
     */
    out     = new ByteArrayOutputStream();

    /* Initialize encoder instance with this output stream.
     */
    encoder     = new DEREncoder(out);

    /* Byte array to store the encoded data.
     */
    encodedAsn1Object = null;

    try
    {
        /* Encoder reads the data stored in the variable asn1Object and
         * encodes it writing the output to the output stream.
         */
        asn1Object.encode(encoder);

        /* Store the data in the output stream in a byte array. This arra
         * will be decoded later.
         */
        encodedAsn1Object = out.toByteArray();
        /* Close the stream.
         */
        encoder.close();
    }
    catch (ASN1Exception e)
    {
        System.out.println("Error␣during␣encoding.");
        e.printStackTrace();
    }
    catch (IOException e)
    {
        System.out.println("Error␣during␣encoding.");
        e.printStackTrace();
    }


    /* Print the encoded data to the standard output in hexadecimal
     * representation.
     */
    buf = new StringBuffer();

    for (int 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("Encoded␣ASN.1␣object:");
    System.out.println(buf.toString());
    System.out.println();


    /* Decoding process.
     */
    /* Create new empty object of the expected class. This object will
     * store the decoded values.
     */
    newAsn1Object     = new ASN1IA5String();

    /* Initialize input stream containing the encoded data.
     */
    in     = new ByteArrayInputStream(encodedAsn1Object);

    /* Initialize decoder instance with this input stream.
     */
```

```
decoder = new DERDecoder(in);

try
{
    /* Decode the data in the input stream and stored it in
     * newAsn1Object.
     */
    newAsn1Object.decode(decoder);

    /* Close the stream.
     */
    decoder.close();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}

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


/* Alternative decoding procedure without assuming to know the ASN.1
 * type to be decoded.
 */
System.out.println("Alternative␣decoding␣procedure:");

/* Variable to store the data that will be decoded. Its type is not
 * determined.
 */
asn1Type     = null;

/* Input stream containing the encoded data.
 */
in     = new ByteArrayInputStream(encodedAsn1Object);

/* Initialize decoder instance with this input stream.
 */
decoder = new DERDecoder(in);
```

```
        try
        {
            /* Decoder returns a Java object of the corresponding CODEC class
             * and already containing the decoded data.
             */
            asn1Type = decoder.readType();
            decoder.close();
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(asn1Type.toString());
        System.out.println();
    }
}
```

Appendix B

SEQUENCE Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.sequence;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Sequence;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type.
 * <PRE>Order := SEQUENCE {
 *     product-name              IA5String,
 *     available-quantity        INTEGER }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Order.java,v 1.2 2004/08/05 13:21:05 pebinger Exp $"
 */
public class Order extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */
```

```
/**
 * DOCUMENT ME!
 */
private ASN1IA5String productName_ = null;

/**
 * DOCUMENT ME!
 */
private ASN1Integer neededQuantity_ = null;

/**
 * Constructor to create an object to be encoded.
 *
 * @param productName DOCUMENT ME!
 * @param neededQuantity DOCUMENT ME!
 */
public Order(String productName, int neededQuantity)
{
    /* Allocate memory for the member variables.
     */
    super(2);

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

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

/**
 * Constructor to create an object ready to decode data.
 */
public Order()
{
    super(2);

    /* Initialize the member variables ready for decoding each component
     * of the SEQUENCE.
     */
    productName_        = new ASN1IA5String();
    neededQuantity_     = new ASN1Integer();

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

/**
 * Returns a byte array representing an encoded instance of this class.
 *
 * @return DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
public byte[] getEncoded() throws ASN1Exception, IOException
{
    ByteArrayOutputStream out;
    DEREncoder encoder;
    byte[] encodedAsn1Object;

    /* Initialize an output stream to which the encoded data will be
     * written.
     */
    out     = new ByteArrayOutputStream();

    /* Initialize encoder instance with this output stream.
     */
    encoder = new DEREncoder(out);

    /* Encoder reads the data stored in the member variables of this class
     * and encodes it writing the output to the output stream.
     */
    this.encode(encoder);

    /* Store the data in the output stream in a byte array. This array wil
     * be returned by this method.
     */
    encodedAsn1Object = out.toByteArray();

    /* Close the stream.
     */
    encoder.close();

    /* Return the encoded data.
     */
    return encodedAsn1Object;
}

/**
 * Decodes the byte array passed as argument. The decoded values are
 * stored in the member variables of this class that represent the
 * components of the corresponding ASN.1 type.
 *
 * @param encodedData DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
public void decode(byte[] encodedData) throws ASN1Exception, IOException
{
    ByteArrayInputStream in;
    DERDecoder decoder;

    /* Initialize input stream containing the encoded data.
     */
    in      = new ByteArrayInputStream(encodedData);

    /* Initialize decoder instance with this input stream.
     */
    decoder = new DERDecoder(in);

    /* Decode the data in the input stream and stored it in the member
     * variables of this class that represent the components of the
     * corresponding ASN.1 type.
     */
    this.decode(decoder);

    /* Close the stream.
     */
    decoder.close();
}

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

/**
 * DOCUMENT ME!
 *
 * @return DOCUMENT ME!
 */
public String getProductName()
{
    return productName_.getString();
}
```

```
    /**
     * DOCUMENT ME!
     *
     * @param neededQuantity DOCUMENT ME!
     */
    public void setNeededQuantity(int neededQuantity)
    {
        neededQuantity_ = new ASN1Integer(neededQuantity);
        set(1, neededQuantity_);
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public int getNeededQuantity()
    {
        return neededQuantity_.getBigInteger().intValue();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.sequence;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 SEQUENCE value is created, encoded and
 * decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:05 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        Order asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        Order newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object = new Order("Soap", 152);

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
```

```
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}


/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Order();

/* Decode the data stored in the byte array encodedAsn1Object and stor
 * it in newAsn1Object.
 */
try
{
```

```
            newAsn1Object.decode(encodedAsn1Object);
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix C

SEQUENCE OF Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.sequenceOf;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1SequenceOf;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of the ASN.1 SEQUENCE OF type.
 * <PRE>OrderList := SEQUENCE OF Order</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: OrderList.java,v 1.2 2004/08/05 13:21:05 pebinger Exp $"
 */
public class OrderList extends ASN1SequenceOf
{
    /**
     * Constructor to create an object either be encoded or to decode data.
     * If used for encoding, elements containing the data have to be added
     * to the object created by calling this constructor.
     */
    public OrderList()
```

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

/**
 * Constructor to create an object to be encoded. The data is provided by
 * the array given as a parameter.
 *
 * @param orderArray DOCUMENT ME!
 */
public OrderList(Order[] orderArray)
{
    super(Order.class, orderArray.length);

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

/**
 * Returns a byte array representing an encoded instance of this class.
 *
 * @return DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
public byte[] getEncoded() throws ASN1Exception, IOException
{
    ByteArrayOutputStream out;
    DEREncoder encoder;
    byte[] encodedAsn1Object;

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

    this.encode(encoder);
    encodedAsn1Object = out.toByteArray();
    encoder.close();

    return encodedAsn1Object;
}

/**
 * Decodes the byte array passed as argument. The decoded values are
 * stored in the member variables of this class that represent the
 * components of the corresponding ASN.1 type.
```

```
         *
         * @param encodedData DOCUMENT ME!
         *
         * @throws ASN1Exception DOCUMENT ME!
         * @throws IOException DOCUMENT ME!
         */
        public void decode(byte[] encodedData) throws ASN1Exception, IOException
        {
            ByteArrayInputStream in;
            DERDecoder decoder;

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

            this.decode(decoder);
            decoder.close();
        }
    }
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.sequenceOf;

import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Sequence;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type.
 * <PRE>Order := SEQUENCE {
 *     product-name              IA5String,
 *     available-quantity        INTEGER }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Order.java,v 1.2 2004/08/05 13:21:05 pebinger Exp $"
 */
public class Order extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1IA5String productName_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1Integer neededQuantity_ = null;

    /**
     * Constructor for creating an object to be encoded.
     *
     * @param productName DOCUMENT ME!
     * @param neededQuantity DOCUMENT ME!
     */
    public Order(String productName, int neededQuantity)
    {
        super(2);

        productName_        = new ASN1IA5String(productName);
        neededQuantity_     = new ASN1Integer(neededQuantity);
        add(productName_);
        add(neededQuantity_);
    }

    /**
     * Constructor to create an object ready to decode data.
     */
    public Order()
    {
        super(2);

        productName_        = new ASN1IA5String();
        neededQuantity_     = new ASN1Integer();

        add(productName_);
        add(neededQuantity_);
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.sequenceOf;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 SEQUENCE OF value is created, encoded and
 * decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:04 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        OrderList asn1Object;
        Order order0;
        Order order1;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        OrderList newAsn1Object;


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

        order0         = new Order("Soap", 152);
        order1         = new Order("Shampoo", 298);

        asn1Object.add(order0);
        asn1Object.add(order1);

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
        encodedAsn1Object = null;
        try
        {
            encodedAsn1Object = asn1Object.getEncoded();
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣encoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣encoding.");
            e.printStackTrace();
        }

        /* Print the encoded data to the standard output in hexadecimal
         * representation.
         */
        buf = new StringBuffer();

        for (int 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("Encoded␣ASN.1␣object:");
        System.out.println(buf.toString());
        System.out.println();


        /* Decoding process.
         */
        /* Create new empty object of the expected class. This object will
         * store the decoded values.
         */
```

```
        newAsn1Object = new OrderList();

        /* Decode the data stored in encodedAsn1Object and store it in
         * newAsn1Object.
         */
        try
        {
            newAsn1Object.decode(encodedAsn1Object);
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix D

CHOICE Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.choice;

import codec.asn1.ASN1Choice;
import codec.asn1.ASN1Exception;
import codec.asn1.DERDecoder;

import java.io.ByteArrayInputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 CHOICE type.
 * <PRE>Response ::= CHOICE {
 *     acknoledgment_        Acknoledgement,
 *     errorCode_            ErrorCode }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Response.java,v 1.2 2004/08/05 13:07:32 pebinger Exp $"
 */
public class Response extends ASN1Choice
{
    /* Member variables representing each possible choice.
     */

    /**
     * DOCUMENT ME!
     */
    private Acknowledgment acknoledgment_ = null;

    /**
     * DOCUMENT ME!
     */
    private ErrorCode errorCode_ = null;

    /**
     * Constructor for decoding.
     */
    public Response()
    {
        /* Allocate memory for storing two objects (for each choice).
         */
        super(2);

        /* Initialize the member variables ready for decoding each of the
         * possible choices.
         */
        acknoledgment_      = new Acknowledgment();
        errorCode_          = new ErrorCode();

        /* Store the possible choices.
         */
        addType(acknoledgment_);
        addType(errorCode_);
    }

    /**
     * Decodes the byte array passed as argument. The decoded values are
     * stored in the member variables of this class that represent the
     * components of the corresponding ASN.1 type.
     *
     * @param encodedData DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void decode(byte[] encodedData) throws ASN1Exception, IOException
    {
        ByteArrayInputStream in;
        DERDecoder decoder;

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

        this.decode(decoder);
        decoder.close();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.choice;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1Null;
import codec.asn1.DEREncoder;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class represents an acknowledgment modelled by an ASN.1 NULL object.
 * <PRE>Acknowledgment ::= NULL</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Acknowledgment.java,v 1.16 2004/08/05 13:07:32 pebinger Exp
 */
public class Acknowledgment extends ASN1Null
{
    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

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

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.choice;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1Integer;
import codec.asn1.DEREncoder;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class represents an error code value modelled by an ASN.1 INTEGER
 * object.
 * <PRE>ErrorCode ::= INTEGER</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorCode.java,v 1.2 2004/08/05 13:07:32 pebinger Exp $"
 */
public class ErrorCode extends ASN1Integer
{
    /**
     * Constructor for encoding.
     *
     * @param value DOCUMENT ME!
     */
    public ErrorCode(int value)
    {
        super(value);
    }

    /**
     * Constructor for decoding.
     */
    public ErrorCode()
    {
        super();
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

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

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with the terms of the licens
 * agreement you entered into with Fraunhofer Gesellschaft.
 */
package codec.examples.choice;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 CHOICE value is created, encoded and decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:07:32 pebinge
 */
public class CodingDecodingExample
{

    /**
     * DOCUMENT ME!
     */
    public static void main(String[] args)
    {

        Acknowledgment asn1Object1;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        Response newAsn1Object;
        ErrorCode asn1Object2;


        /* Create ASN.1 object.
         */
        asn1Object1 = new Acknowledgment();

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
        encodedAsn1Object = null;
```

```
try
{
    encodedAsn1Object = asn1Object1.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */

/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Response();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
```

```
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}

/* Print the new decoded ASN.1 object to the standard output.
 */
System.out.println("New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣byte
System.out.println(newAsn1Object.toString());
System.out.println();


/*********************************************************************
 Repeat process with the other possible data type valid for the CHOICE
/*********************************************************************

/* Create ASN.1 object.
 */
asn1Object2 = new ErrorCode(1);

/* Print the ASN.1 object to the standard output.
 */
System.out.println("ASN.1␣object:␣");
System.out.println(asn1Object2.toString());
System.out.println();

/* Encode the ASN.1 object and store the encoded data in
 * encodedAsn1Object.
 */
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object2.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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 object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Response();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}

/* Print the new decoded ASN.1 object to the standard output.
```

```
         */
        System.out.println("New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣data
        System.out.println(newAsn1Object.toString());
        System.out.println();

    }

}
```

Appendix E

ENUMERATED Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.enumerated;

import codec.asn1.ASN1Enumerated;
import codec.asn1.ASN1Exception;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 ENUMERATED type.
 * <PRE>ResponseStatus ::= ENUMERATED {
 *      successful          (0),  -- understood request
 *      malformedRequest    (1)   -- malformed request }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ResponseStatus.java,v 1.2 2004/08/05 13:21:04 pebinger Exp $
 */
public class ResponseStatus extends ASN1Enumerated
{
    /* Constants representing each possible value.
     */

    /**
     * DOCUMENT ME!
     */
    public static final int SUCCESSFUL = 0;

    /**
     * DOCUMENT ME!
     */
    public static final int MALFORMED_REQUEST = 1;

    /**
     * Constructor for encoding.
     *
     * @param value DOCUMENT ME!
     *
     * @throws IllegalArgumentException DOCUMENT ME!
     */
    public ResponseStatus(int value) throws IllegalArgumentException
    {
        super(value);

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

    /**
     * Constructor for decoding.
     */
    public ResponseStatus()
    {
        super();
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

        out         = new ByteArrayOutputStream();
        encoder     = new DEREncoder(out);
```

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

            return encodedAsn1Object;
        }

        /**
         * Decodes the byte array passed as argument. The decoded values are
         * stored in the member variables of this class that represent the
         * components of the corresponding ASN.1 type.
         *
         * @param encodedData DOCUMENT ME!
         *
         * @throws ASN1Exception DOCUMENT ME!
         * @throws IOException DOCUMENT ME!
         */
        public void decode(byte[] encodedData) throws ASN1Exception, IOException
        {
            ByteArrayInputStream in;
            DERDecoder decoder;

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

            this.decode(decoder);
            decoder.close();
        }

        /**
         * Returns the value stored.
         *
         * @return DOCUMENT ME!
         */
        public int getInt()
        {
            return getBigInteger().intValue();
        }
    }
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.enumerated;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 ENUMERATED value is created, encoded and
 * decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:04 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        ResponseStatus asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        ResponseStatus newAsn1Object;


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

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
```

```
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new ResponseStatus();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
```

```
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix F

OPTIONAL Fields Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.optionalFields;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Sequence;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;
import codec.asn1.Encoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type with an
 * OPTIONAL component.
 * <PRE>Person := SEQUENCE {
 *      age    INTEGER OPTIONAL,
 *     name    IA5String }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Person.java,v 1.3 2004/08/05 13:21:04 pebinger Exp $"
 */
public class Person extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1Integer age_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1IA5String name_ = null;

    /**
     * Constructor for encoding setting a value for the optional field.
     *
     * @param age DOCUMENT ME!
     * @param name DOCUMENT ME!
     */
    public Person(int age, String name)
    {
        super(2);

        age_      = new ASN1Integer(age);
        name_     = new ASN1IA5String(name);
    }

    /**
     * Constructor for encoding leaving optional field empty.
     *
     * @param name DOCUMENT ME!
     */
    public Person(String name)
    {
        super(1);

        name_ = new ASN1IA5String(name);
    }

    /**
     * Constructor for decoding.
     */
    public Person()
    {
        super(2);

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

        add(age_);
        add(name_);
    }

    /**
     * Add only the components that will be encoded to class.
     */
    protected void map()
    {
        clear();
        if (age_ != null)
        {
            add(age_);
        }
        add(name_);
    }

    /**
     * Override the encode(Encoder) method so that map() is called before
     * each call of this method.
     *
     * @param enc DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void encode(Encoder enc) throws ASN1Exception, IOException
    {
        map();
        super.encode(enc);
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

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

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }

    /**
     * Decodes the byte array passed as argument. The decoded values are
     * stored in the member variables of this class that represent the
     * components of the corresponding ASN.1 type.
     *
     * @param encodedData DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void decode(byte[] encodedData) throws ASN1Exception, IOException
    {
        ByteArrayInputStream in;
        DERDecoder decoder;

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

        this.decode(decoder);
        decoder.close();
    }

    /**
     * Set and get methods.
     *
     * @param age DOCUMENT ME!
     */
    public void setAge(int age)
    {
        age_ = new ASN1Integer(age);
        age_.setOptional(false);
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public int getAge()
    {
        return age_.getBigInteger().intValue();
    }

    /**
     * DOCUMENT ME!
     *
     * @param name DOCUMENT ME!
     */
    public void setName(String name)
    {
        name_ = new ASN1IA5String(name);
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getName()
    {
        return name_.getString();
    }


    /* Remove the value of the optional field.
     */
    public void removeAge()
    {
        age_ = null;
    }

    /**
     * Override the toString() method so that map() is called before each
     * call of this method.
     *
     * @return DOCUMENT ME!
     */
    public String toString()
    {
        map();
        return super.toString();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with the terms of the licens
 * agreement you entered into with Fraunhofer Gesellschaft.
 */

package codec.examples.optionalFields;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 SEQUENCE value with optional fields is
 * created, encoded and decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:04 pebinge
 */

public class CodingDecodingExample
{

    /**
     * DOCUMENT ME!
     *
     * @param args
     */
    public static void main(String[] args)
    {

        Person asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        Person newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object = new Person(30, "Volker");

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

```
/* Encode the ASN.1 object and store the encoded data in
 * encodedAsn1Object.
 */
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */

/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Person();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}

/* Print the new decoded ASN.1 object to the standard output.
 */
System.out.println("New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣byte
System.out.println(newAsn1Object.toString());
System.out.println();

/*******************************************************************
          Repeat process without encoding optional component.
/*******************************************************************

/* Create ASN.1 object.
 */
asn1Object = new Person("Volker");

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

/* Encode the ASN.1 object and store the encoded data in
 * encodedAsn1Object.
 */
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
```

```
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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 object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Person();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣decoding.");
    e.printStackTrace();
```

```
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println("New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣data
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }

}
```

Appendix G

Tagging Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.tagging;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Sequence;
import codec.asn1.ASN1TaggedType;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;
import codec.asn1.Encoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type with a
 * tagged component.
 * <PRE>Person := SEQUENCE {
 *     title     [0] IA5string OPTIONAL,
 *     name          IA5string }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Person.java,v 1.3 2004/08/05 13:21:04 pebinger Exp $"
 */
public class Person extends ASN1Sequence
{
```

```
/* Member variables representing the components of the SEQUENCE.
 */

/**
 * DOCUMENT ME!
 */
private ASN1IA5String title_ = null;

/**
 * DOCUMENT ME!
 */
private ASN1IA5String name_ = null;

/**
 * Constant representing the tag value of title.
 */
final int TITLE_TAG = 0;

/**
 * Constructor for encoding setting a value for the optional field.
 *
 * @param title DOCUMENT ME!
 * @param name DOCUMENT ME!
 */
public Person(String title, String name)
{
    super(2);

    title_     = new ASN1IA5String(title);
    name_      = new ASN1IA5String(name);
}

/**
 * Constructor for encoding leaving optional field empty.
 *
 * @param name DOCUMENT ME!
 */
public Person(String name)
{
    super(1);

    name_ = new ASN1IA5String(name);
}

/**
 * Constructor for decoding.
 */
public Person()
{
    super(2);
```

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

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

    /**
     * Add only the components that will be encoded to class.
     */
    protected void map()
    {
        clear();
        if (title_ != null)
        {
            add(new ASN1TaggedType(TITLE_TAG, title_, true));
        }
        add(name_);
    }

    /**
     * Override the encode(Encoder) method so that map() is called before
     * each call of this method.
     *
     * @param enc DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void encode(Encoder enc) throws ASN1Exception, IOException
    {
        map();
        super.encode(enc);
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;
        out          = new ByteArrayOutputStream();
        encoder      = new DEREncoder(out);

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }

    /**
     * Decodes the byte array passed as argument. The decoded values are
     * stored in the member variables of this class that represent the
     * components of the corresponding ASN.1 type.
     *
     * @param encodedData DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void decode(byte[] encodedData) throws ASN1Exception, IOException
    {
        ByteArrayInputStream in;
        DERDecoder decoder;

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

        this.decode(decoder);
        decoder.close();
    }

    /**
     * Set and get methods.
     *
     * @param title DOCUMENT ME!
     */
    public void setTitle(String title)
    {
        title_ = new ASN1IA5String(title);
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getTitle()
    {
        return title_.getString();
    }

    /**
     * DOCUMENT ME!
     *
     * @param name DOCUMENT ME!
     */
    public void setName(String name)
    {
        name_ = new ASN1IA5String(name);
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getName()
    {
        return name_.getString();
    }

    /**
     * Remove the value of the optional field.
     */
    public void removeTitle()
    {
        title_ = null;
    }

    /**
     * Override the toString() method so that map() is called before each
     * call of this method.
     *
     * @return DOCUMENT ME!
     */
    public String toString()
    {
        map();
        return super.toString();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.tagging;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how a SEQUENCE value with tagged fields is created,
 * encoded and decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:04 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        Person asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        Person newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object = new Person("Sir", "Peter");

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
```

```
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Person();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
```

```
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix H

DEFAULT Values Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.defaultValues;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Sequence;
import codec.asn1.ASN1TaggedType;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;
import codec.asn1.Encoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type with a
 * DEFAULT value for a component.
 * <PRE>Payment ::= SEQUENCE {
 *      quantity   INTEGER,
 *      currency   IA5String  DEFAULT Euro }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: Payment.java,v 1.3 2004/08/05 13:19:27 pebinger Exp $"
 */
public class Payment extends ASN1Sequence
```

```
{
    /**
     * Member variable representing the quantity field of the ASN.1 SEQUENCE.
     */
    private ASN1Integer quantity_ = null;

    /**
     * Member variable representing the currency field of the ASN.1 SEQUENCE.
     */
    private ASN1IA5String currency_ = null;

    /**
     * Constant for the default currency.
     */
    private final String DEFAULT_CURRENCY = "Euro";

    /**
     * Constant for the currency tag.
     */
    private final int CURRENCY_TAG = 0;

    /**
     * Constructor for encoding, setting a value different than the default.
     *
     * @param quantity DOCUMENT ME!
     * @param currency DOCUMENT ME!
     */
    public Payment(int quantity, String currency)
    {
        super(2);

        quantity_ = new ASN1Integer(quantity);
        if (!currency.equals(DEFAULT_CURRENCY))
        {
            currency_ = new ASN1IA5String(currency);
        }
    }

    /**
     * Constructor for encoding, assuming the default value.
     *
     * @param quantity DOCUMENT ME!
     */
    public Payment(int quantity)
    {
        super(1);

        quantity_ = new ASN1Integer(quantity);
    }
```

```
/**
 * Constructor for decoding.
 */
public Payment()
{
    super(3);

    quantity_     = new ASN1Integer();
    currency_     = new ASN1IA5String();

    add(quantity_);
    add(new ASN1TaggedType(CURRENCY_TAG, currency_, true, true));
}

/**
 * Add only the components that will be encoded to class.
 */
protected void map()
{
    clear();
    add(quantity_);
    if (currency_ != null)
    {
        add(new ASN1TaggedType(CURRENCY_TAG, currency_, true));
    }
}

/**
 * Override the encode(Encoder) method so that map() is called before
 * each call of this method.
 *
 * @param enc DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
public void encode(Encoder enc) throws ASN1Exception, IOException
{
    map();
    super.encode(enc);
}

/**
 * Returns a byte array representing an encoded instance of this class.
 *
 * @return DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
```

```
public byte[] getEncoded() throws ASN1Exception, IOException
{
    ByteArrayOutputStream out;
    DEREncoder encoder;
    byte[] encodedAsn1Object;

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

    this.encode(encoder);
    encodedAsn1Object = out.toByteArray();
    encoder.close();

    return encodedAsn1Object;
}

/**
 * Decodes the byte array passed as argument. The decoded values are
 * stored in the member variables of this class that represent the
 * components of the corresponding ASN.1 type.
 *
 * @param encodedData DOCUMENT ME!
 *
 * @throws ASN1Exception DOCUMENT ME!
 * @throws IOException DOCUMENT ME!
 */
public void decode(byte[] encodedData) throws ASN1Exception, IOException
{
    ByteArrayInputStream in;
    DERDecoder decoder;

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

    this.decode(decoder);
    decoder.close();
}

/**
 * Set and get methods.
 *
 * @param quantity DOCUMENT ME!
 */
public void setQuantity(int quantity)
{
    quantity_ = new ASN1Integer(quantity);
}

/**
 * DOCUMENT ME!
```

```
     *
     * @return DOCUMENT ME!
     */
    public int getQuantity()
    {
        return quantity_.getBigInteger().intValue();
    }

    /**
     * DOCUMENT ME!
     *
     * @param currency DOCUMENT ME!
     */
    public void setCurrency(String currency)
    {
        if (!currency.equals(DEFAULT_CURRENCY))
        {
            currency_ = new ASN1IA5String(currency);
        }
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getCurrency()
    {
        if (currency_ == null)
        {
            return DEFAULT_CURRENCY;
        }
        else
        {
            return currency_.getString();
        }
    }

    /**
     * Override the toString() method so that the default value is printed.
     *
     * @return DOCUMENT ME!
     */
    public String toString()
    {
        if (currency_ == null)
        {
            currency_ = new ASN1IA5String(DEFAULT_CURRENCY);
            map();
            currency_ = null;
            return super.toString();
        }
        else
        {
            if (currency_.getString() == "")
            {
                currency_ = new ASN1IA5String(DEFAULT_CURRENCY);
                map();
                currency_ = new ASN1IA5String();
                return super.toString();
            }
            else
            {
                map();
                return super.toString();
            }
        }
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.defaultValues;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 SEQUENCE value with default values is
 * created, encoded and decoded.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:09:09 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        Payment asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        Payment newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object = new Payment(100);

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

        /* Encode the ASN.1 object and store the encoded data in
         * encodedAsn1Object.
         */
```

```
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new Payment();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
try
{
    newAsn1Object.decode(encodedAsn1Object);
```

```
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix I

ANY DEFINED BY Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.anyDefinedBy;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Null;
import codec.asn1.ASN1ObjectIdentifier;
import codec.asn1.ASN1OpenType;
import codec.asn1.ASN1Sequence;
import codec.asn1.ASN1Type;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

import java.math.BigInteger;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type with an
 * ANY DEFINED BY field.
 * <PRE>
 * ErrorMessage := SEQUENCE {
```

```
 *   oid             OBJECT IDENTIFIER,
 *   parameter       ANY DEFINED BY oid
 *
 *     --    oid    | ASN.1 type of parameter
 *     -- =========|=======================
 *     -- 2.1.0.2.0 | NULL
 *     -- 2.1.0.2.2 | INTEGER
 *     -- 1.2.3.4.5 | ErrorParameterType1 }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorMessage.java,v 1.2 2004/08/05 13:07:18 pebinger Exp $"
 */
public class ErrorMessage extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1ObjectIdentifier oid_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1Type parameter_ = null;

    /**
     * Constructor for encoding.
     *
     * @param oid DOCUMENT ME!
     * @param obj DOCUMENT ME!
     */
    public ErrorMessage(String oid, Object obj)
    {
        super(2);

        oid_ = new ASN1ObjectIdentifier(oid);

        if (oid.equals("2.1.0.2.0"))
        {
            parameter_ = new ASN1Null();
        }
        else if (oid.equals("2.1.0.2.2"))
        {
            parameter_ = new ASN1Integer((BigInteger)obj);
        }
        else if (oid.equals("1.2.3.4.5"))
        {
            parameter_ = (ASN1Type)obj;
        }

        add(oid_);
        add(parameter_);
    }

    /**
     * Constructor for decoding.
     */
    public ErrorMessage()
    {
        super(2);

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

        add(oid_);
        add(parameter_);
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

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

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }

    /**
     * Decodes the byte array passed as argument. The decoded values are
     * stored in the member variables of this class that represent the
     * components of the corresponding ASN.1 type.
     *
     * @param encodedData DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public void decode(byte[] encodedData) throws ASN1Exception, IOException
    {
        ByteArrayInputStream in;
        DERDecoder decoder;

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

        this.decode(decoder);
        decoder.close();
    }

    /**
     * Get methods.
     *
     * @return DOCUMENT ME!
     */
    public String getOID()
    {
        return oid_.toString();
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public ASN1Type getParameter()
    {
        return parameter_;
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.anyDefinedBy;

import codec.asn1.ASN1Boolean;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Sequence;

/**
 * This class represents an error parameter type modelled by an ASN.1 SEQUENCE
 * <PRE>ErrorParameterType1 := SEQUENCE {
 *   critical       BOOLEAN,
 *   info           IA5String }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorParameterType1.java,v 1.2 2004/08/05 13:07:18 pebinger
 */
public class ErrorParameterType1 extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1Boolean critical_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1IA5String info_ = null;

    /**
     * Constructor for encoding.
     *
     * @param critical DOCUMENT ME!
     * @param info DOCUMENT ME!
     */
    public ErrorParameterType1(boolean critical, String info)
    {
        super(2);

        critical_     = new ASN1Boolean(critical);
        info_         = new ASN1IA5String(info);
        add(critical_);
        add(info_);
    }

    /**
     * Constructor for decoding.
     */
    public ErrorParameterType1()
    {
        super(2);

        critical_     = new ASN1Boolean();
        info_         = new ASN1IA5String();

        add(critical_);
        add(info_);
    }

    /**
     * Get methods.
     *
     * @return DOCUMENT ME!
     */
    public boolean getCritical()
    {
        return critical_.isTrue();
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getInfo()
    {
        return info_.getString();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.anyDefinedBy;

import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Null;
import codec.asn1.ASN1ObjectIdentifier;
import codec.asn1.ASN1Type;
import codec.asn1.Resolver;
import codec.asn1.ResolverException;

/**
 * This class shows a sample implementation of a Resolver class. The
 * resolve(ASN1Type) method of this class will return an ASN1Type object for
 * decoding an ANY DEFINED BY value. The ASN.1 type retuned will depend on
 * the value of an OBJECT IDENTIFIER object according to the following
 * table:
 * <PRE>
 *      --    oid    | ASN.1 type of parameter
 *      -- ==========|=========================
 *      -- 2.1.0.2.0 | NULL
 *      -- 2.1.0.2.2 | INTEGER
 *      -- 1.2.3.4.5 | ErrorParameterType1</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorResolver.java,v 1.2 2004/08/05 13:07:18 pebinger Exp $'
 */
public class ErrorResolver implements Resolver
{
    /* Reference to the object identifier instance whose value determines whic
     * ASN.1 type has to be returned by the resolve(ASN1Type) method of this
     * class.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1ObjectIdentifier oid_;

    /**
     * Constructor.
     *
     * @param oid The reference to the object identifier whose value will
     *        determine the ASN.1 type of the ANY DEFINED BY value. At this
```

```
         *         time the object identifier is empty.
         */
        public ErrorResolver(ASN1ObjectIdentifier oid)
        {
            oid_ = oid;
        }

        /**
         * This method returns an empty instance of the appropriate ASN.1 type
         * for decoding the ANY DEFINED BY value. When this method is called the
         * OBJECT IDENTIFIER pointed by oid_ has been decoded in the meantime,
         * so that its value can be queried.
         *
         * @param caller The instance from which this method will be called,
         *         usually an ASN1OpenType instance (see decode(Decoder) in the
         *         source of ASN1OpenType).
         *
         * @return DOCUMENT ME!
         *
         * @throws ResolverException DOCUMENT ME!
         */
        public ASN1Type resolve(ASN1Type caller) throws ResolverException
        {
            if (oid_.toString().equals("2.1.0.2.0"))
            {
                return new ASN1Null();
            }
            else if (oid_.toString().equals("2.1.0.2.2"))
            {
                return new ASN1Integer();
            }
            else if (oid_.toString().equals("1.2.3.4.5"))
            {
                return new ErrorParameterType1();
            }
            else
            {
                throw new ResolverException();
            }
        }
    }
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.anyDefinedBy;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 object implmenting an ANY DEFINED BY type is
 * created, encoded and decoded with the help of a Resolver.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.5 2004/08/19 16:41:29 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        ErrorMessage asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        ErrorMessage newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object =
            new ErrorMessage(
                "1.2.3.4.5",
                new ErrorParameterType1(true, "Error"));

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

```
/* Encode the ASN.1 object and store the encoded data in
 * encodedAsn1Object.
 */
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new ErrorMessage();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
```

```
        try
        {
            newAsn1Object.decode(encodedAsn1Object);
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Appendix J

OIDRegistry Example

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.oidRegistry;

import codec.asn1.ASN1Exception;
import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Null;
import codec.asn1.ASN1ObjectIdentifier;
import codec.asn1.ASN1OpenType;
import codec.asn1.ASN1Sequence;
import codec.asn1.ASN1Type;
import codec.asn1.DERDecoder;
import codec.asn1.DEREncoder;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

import java.math.BigInteger;

/**
 * This class shows a sample implementation of an ASN.1 SEQUENCE type with an
 * ANY DEFINED BY field.
 * <PRE>ErrorMessage := SEQUENCE {
 *   oid             OBJECT IDENTIFIER,
 *   parameter       ANY DEFINED BY oid
 *
 *      --    oid    | ASN.1 type of parameter
 *      -- ==========|=========================
 *      -- 2.1.0.2.0 | NULL
 *      -- 2.1.0.2.2 | INTEGER
 *      -- 1.2.3.4.5 | ErrorParameterType1 }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorMessage.java,v 1.2 2004/08/05 13:21:04 pebinger Exp $"
 */
public class ErrorMessage extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1ObjectIdentifier oid_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1Type parameter_ = null;

    /**
     * Constructor for encoding.
     *
     * @param oid DOCUMENT ME!
     * @param obj DOCUMENT ME!
     */
    public ErrorMessage(String oid, Object obj)
    {
        super(2);

        oid_ = new ASN1ObjectIdentifier(oid);

        if (oid.equals("2.1.0.2.0"))
        {
            parameter_ = new ASN1Null();
        }
        else if (oid.equals("2.1.0.2.2"))
        {
            parameter_ = new ASN1Integer((BigInteger)obj);
        }
        else if (oid.equals("1.2.3.4.5"))
        {
            parameter_ = (ASN1Type)obj;
        }

        add(oid_);
        add(parameter_);
    }

    /**
     * Constructor for decoding.
     */
    public ErrorMessage()
    {
        super(2);

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

        add(oid_);
        add(parameter_);
    }

    /**
     * Returns a byte array representing an encoded instance of this class.
     *
     * @return DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
    public byte[] getEncoded() throws ASN1Exception, IOException
    {
        ByteArrayOutputStream out;
        DEREncoder encoder;
        byte[] encodedAsn1Object;

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

        this.encode(encoder);
        encodedAsn1Object = out.toByteArray();
        encoder.close();

        return encodedAsn1Object;
    }

    /**
     * Decodes the byte array passed as argument. The decoded values are
     * stored in the member variables of this class that represent the
     * components of the corresponding ASN.1 type.
     *
     * @param encodedData DOCUMENT ME!
     *
     * @throws ASN1Exception DOCUMENT ME!
     * @throws IOException DOCUMENT ME!
     */
```

```
    public void decode(byte[] encodedData) throws ASN1Exception, IOException
    {
        ByteArrayInputStream in;
        DERDecoder decoder;

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

        this.decode(decoder);
        decoder.close();
    }

    /**
     * Get methods.
     *
     * @return DOCUMENT ME!
     */
    public String getOID()
    {
        return oid_.toString();
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public ASN1Type getParameter()
    {
        return parameter_;
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.oidRegistry;

import codec.asn1.ASN1Boolean;
import codec.asn1.ASN1IA5String;
import codec.asn1.ASN1Sequence;

/**
 * This class represents an error parameter type modelled by an ASN.1 SEQUENCE
 * <PRE>ErrorParameterType1 := SEQUENCE {
 *   critical      BOOLEAN,
 *   info          IA5String }</PRE>
 *
 * @author Alberto Sierra
 * @version "$Id: ErrorParameterType1.java,v 1.2 2004/08/05 13:21:04 pebinger
 */
public class ErrorParameterType1 extends ASN1Sequence
{
    /* Member variables representing the components of the SEQUENCE.
     */

    /**
     * DOCUMENT ME!
     */
    private ASN1Boolean critical_ = null;

    /**
     * DOCUMENT ME!
     */
    private ASN1IA5String info_ = null;

    /**
     * Constructor for encoding.
     *
     * @param critical DOCUMENT ME!
     * @param info DOCUMENT ME!
     */
    public ErrorParameterType1(boolean critical, String info)
    {
        super(2);

        critical_     = new ASN1Boolean(critical);
        info_         = new ASN1IA5String(info);
        add(critical_);
        add(info_);
    }

    /**
     * Constructor for decoding.
     */
    public ErrorParameterType1()
    {
        super(2);

        critical_     = new ASN1Boolean();
        info_         = new ASN1IA5String();

        add(critical_);
        add(info_);
    }

    /**
     * Get methods.
     *
     * @return DOCUMENT ME!
     */
    public boolean getCritical()
    {
        return critical_.isTrue();
    }

    /**
     * DOCUMENT ME!
     *
     * @return DOCUMENT ME!
     */
    public String getInfo()
    {
        return info_.getString();
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.oidRegistry;

import codec.asn1.ASN1Integer;
import codec.asn1.ASN1Null;
import codec.asn1.ASN1ObjectIdentifier;
import codec.asn1.AbstractOIDRegistry;
import codec.asn1.OIDRegistry;

import java.util.HashMap;
import java.util.Map;

/**
 * This class maps ASN.1 object identifiers onto ASN.1 types suitable for
 * decoding the structure defined by the given OID.
 *
 * @author Alberto Sierra
 * @version "$Id: SampleOIDRegistry.java,v 1.2 2004/08/05 13:21:04 pebinger Ex
 */
public class SampleOIDRegistry extends AbstractOIDRegistry
{
    /* The OIDs specified as arrays of integers.
     */

    /**
     * DOCUMENT ME!
     */
    private static final int[][] OIDS_ =
        {
            { 2, 1, 0, 2, 0 },
            { 2, 1, 0, 2, 2 },
            { 1, 2, 3, 4, 5 }
        };


    /* The ASN.1 types registered under the OIDs. You can indicate the classes
     * as Class objects as well as character strings.
     */

    /**
     * DOCUMENT ME!
     */
    private static final Object[] TYPES_ =
        { ASN1Null.class, ASN1Integer.class, "ErrorParameterType1" };

    /**
     * If you have many classes within the previous array "types_", which
     * were given as character strings and which are located in the same
     * package, you can save yourself to mention the name of the package for
     * each class name and give it here.
     */
    /**
     * DOCUMENT ME!
     */
    private static final String PREFIX_ = "codec.examples.oidRegistry.";

    /**
     * The mapping from OID to ASN.1 types implementing encoding and decoding
     * of the ASN.1 structure registered under the given OID.
     *
     * <p>
     * This field is initialised on the first call to method {@link
     * #getOIDMap getOIDMap()}.
     * </p>
     */
    /**
     * DOCUMENT ME!
     */
    private static Map map_ = new HashMap();

    /**
     * The default registry. This instance calls the global registry if a
     * requested OID could not be found locally.
     */
    private static SampleOIDRegistry default_ =
        new SampleOIDRegistry(OIDRegistry.getGlobalOIDRegistry());

    /**
     * Creates an instance of this class with no parent.
     */
    public SampleOIDRegistry()
    {
        this(null);
    }

    /**
     * Creates an instance with the given parent.
     *
     * @param parent the parent OID registry.
     */
    public SampleOIDRegistry(OIDRegistry parent)
    {
        super(parent);
```

```
        synchronized (map_)
        {
            if (map_.size() == 0)
            {
                for (int i = 0; i < TYPES_.length; i++)
                {
                    map_.put(new ASN1ObjectIdentifier(OIDS_[i]), TYPES_[i]);
                }
            }
        }
    }

    /**
     * This method returns the default OID registry. The default registry
     * delegates to the global OID registry if a requested OID could not be
     * found locally.
     *
     * @return The default OID registry.
     */
    public static OIDRegistry getDefaultRegistry()
    {
        return default_;
    }

    /**
     * Returns the mapping from OID to ASN.1 types. Types are given initially
     * as a string representing the postfix of the class name implementing
     * the type. This string is replaced by the super class by the
     * corresponding class object when that type is first referenced.
     *
     * <p>
     * There must be at least as many OID in the {@link #oids_ oids_} array
     * as elements in the {@link #types_ types_} array.
     * </p>
     *
     * @return The OID mapping for registered PKCS#7 classes.
     */
    protected Map getOIDMap()
    {
        return map_;
    }

    /**
     * Returns the prefix that is prepended to strings in the mapping
     * returned by {@link #getOIDMap getOIDMap()} in order to form the fully
     * qualified class name.
     *
     * @return The prefix of class names in the mapping.
     */
    protected String getPrefix()
    {
        return PREFIX_;
    }
}
```

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec.examples.oidRegistry;

import codec.asn1.ASN1Exception;

import java.io.IOException;

/**
 * This class shows how an ASN.1 object implmenting an ANY DEFINED BY type is
 * created, encoded and decoded with the help of an OIDRegistry.
 *
 * @author Alberto Sierra
 * @version "$Id: CodingDecodingExample.java,v 1.2 2004/08/05 13:21:04 pebinge
 */
public class CodingDecodingExample
{
    /**
     * DOCUMENT ME!
     *
     * @param args DOCUMENT ME!
     */
    public static void main(String[] args)
    {
        ErrorMessage asn1Object;
        byte[] encodedAsn1Object;
        StringBuffer buf;
        String octet;
        ErrorMessage newAsn1Object;


        /* Create ASN.1 object.
         */
        asn1Object =
            new ErrorMessage(
                "1.2.3.4.5",
                new ErrorParameterType1(true, "BUSSY"));

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

```
/* Encode the ASN.1 object and store the encoded data in
 * encodedAsn1Object.
 */
encodedAsn1Object = null;
try
{
    encodedAsn1Object = asn1Object.getEncoded();
}
catch (ASN1Exception e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}
catch (IOException e)
{
    System.out.println("Error␣during␣encoding.");
    e.printStackTrace();
}

/* Print the encoded data to the standard output in hexadecimal
 * representation.
 */
buf = new StringBuffer();

for (int 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("Encoded␣ASN.1␣object:");
System.out.println(buf.toString());
System.out.println();


/* Decoding process.
 */
/* Create new empty object of the expected class. This object will
 * store the decoded values.
 */
newAsn1Object = new ErrorMessage();

/* Decode the data stored in encodedAsn1Object and store it in
 * newAsn1Object.
 */
```

```
        try
        {
            newAsn1Object.decode(encodedAsn1Object);
        }
        catch (ASN1Exception e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }
        catch (IOException e)
        {
            System.out.println("Error␣during␣decoding.");
            e.printStackTrace();
        }

        /* Print the new decoded ASN.1 object to the standard output.
         */
        System.out.println(
            "New␣ASN.1␣object␣got␣by␣decoding␣the␣previous␣bytes:");
        System.out.println(newAsn1Object.toString());
        System.out.println();
    }
}
```

Bibliography

[1] ITU-T Recommendation X.680 - Abstract Syntax Notation One (ASN.1): Specification of Basic Notation, 1997. http://www.itu.int/itudoc/itu-t/rec/x/x500up/x680.html

[2] ITU-T Recommendation X.690 - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distingushed Encoding Rules (DER), 1997. http://www.itu.int/itudoc/itu-t/rec/x/x500up/x690.html