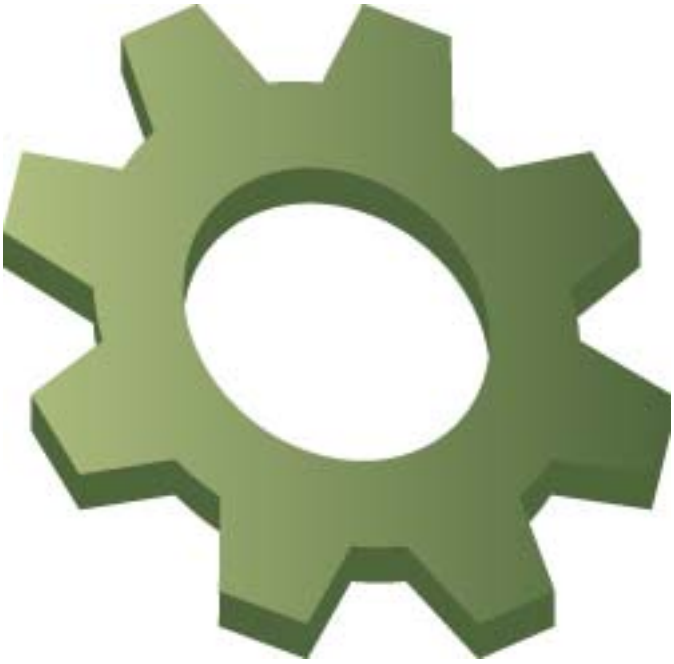
ASN Sower

for Java



Developer guide

by Nathanaël COTTIN

Version 1.2.4

Table of content

Introduction

PowerASN for Java is a high performance ASN.1 codec. It takes advantage of the latest Java technology (generics, etc.).

This guide refers to PowerASN for Java version 1.2.1 (and newer).

Features

PowerASN is developed with respect to flexibility, performance, memory management and security considerations. Thus, encoding and decoding processes are fully streamed.

PowerASN defines two security levels:

- Implementation level: use of multiple interfaces and restrictions on available operations (protected inheritance, final parameters, etc.).
- Run-time level: ASN.1 types constraints are checked before encoding and during decoding. Furthermore, provided (default) decoders can make sure that no extra (i.e. non-decoded) datum is added within parsed source streams (see appendix A for further information). They also prevent from indefinite waiting thanks to user-defined timeouts.

This package includes:

- All ASN.1 types but REAL and EMBEDDED PDV.
- An ASN.1 to Java compiler (will soon be available).
- BER, CER and DER codecs, in conformance with UIT-T Rec. X.690 (07/2002)
- Basic and canonical XER codec, according to UIT-T Rec. X.693 (12/2001).

Extra tools, such as a BER and DER viewer, are also available to provide developers with an extensive ASN.1 environment. These tools are made freely available at *http://www.powerasn.com*.

Requirements

PowerASN for Java is a standalone package which only requires a 1.5 (or newer) JDK. PowerASN for Java compiler makes use of "`compiler.jar`" and "`parser.jar`" to operate. These packages are available at *http://www.powerasn.com*.

Running the given examples

Examples are provided without imports (but the first "getting started" example). "main" operations may be partially shown: given pieces of code must be surrounded with "try – catch" blocks to handle PowerASN exceptions.

When not specified, default DER encoding and decoding rules do apply.

Getting started

Example 1: encoding an decoding a string

This first example shows how to DER-encode and decode an ASN.1 VisibleString containing « Hello ASN.1 world! ».

Encoding process

The encoding process requires to create an ASN.1 object and assign a value to it. The appropriate ASN.1 output stream is then used to produce the corresponding encoding.

```
ASN1VisibleString asn = new ASN1VisibleString();
asn.setValue("Hello ASN.1 world!");

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);
aos.encode(asn);
aos.close();
baos.close();
```

Decoding process

Decoding is performed in two steps :

1. Decode the received stream into an ASN1DecodedObject element. This decodig procedure parses the encoding independently from the destinatory object.
2. The newly created decoded object is used to initialize an (empty) ASN.1 object.

```
byte[] data = baos.toByteArray();

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

ASN1VisibleString decodedAsn = new ASN1VisibleString();
dasn.decodeInto(decodedAsn);
```

Complete source code

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

import pasn.ASN1VisibleString;
import pasn.encoding.ASN1DecodedObject;
import pasn.encoding.ASN1InputStream;
import pasn.encoding.ASN1OutputStream;
```

```
import pasn.encoding.der.DERInputStream;
import pasn.encoding.der.DEROutputStream;
import pasn.misc.ASN1Printer;

public final class Example1 {

  public static void main(String[] args) {
    try {
      // Encoding process

      ASN1VisibleString asn = new ASN1VisibleString();
      asn.setValue("Hello ASN.1 world!");

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ASN1OutputStream aos = new DEROutputStream(baos);
      aos.encode(asn);
      aos.close();
      baos.close();

      // Encoding printing

      byte[] data = baos.toByteArray();
      System.out.println(ASN1Printer.getHexValue(data, ' '));

      // Decoding process

      ByteArrayInputStream bais = new ByteArrayInputStream(data);
      ASN1InputStream ais = new DERInputStream(bais);
      ASN1DecodedObject dasn = ais.decode();
      ais.close();

      ASN1VisibleString decodedAsn = new ASN1VisibleString();
      dasn.decodeInto(decodedAsn);

      System.out.println(decodedAsn);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Note that instead of creating another ASN.1 VisibleString (“decodedAsn”) to received the decoded value, a single object could be used as “decodeInto(…)” calls the destinatory ASN.1 object “reset()” operation which removes its internal value.

Result

Executing this example should print the following on screen:

```
1A 12 48 65 6C 6C 6F 20 41 53 4E 2E 31 20 77 6F 72 6C 64 21
Hello ASN.1 world!
```

Example 2: tagged ASN.1 objects

Description

PowerASN provides means for encoding and decoding ASN.1 implicitly or explicitly tagged objects. Indeed, *ASN1taggedObject* objects are used instead of directly using ASN.1 objects.

Tagging consists in replacing the `UNIVERSAL` object type with a tag (positive) number. The associated class (`CONTEXT`, `APPLICATION`, `PRIVATE`) and explicitness (`IMPLICIT`, `EXPLICIT`) are also specified.

Tagged ASN.1 objects are declared IMPLICIT by default as many specifications make use of “IMPLICIT TAGS” clause.

This example describes DER-encoding and decoding of an application explicitly tagged INTEGER. When no class is specified, `CONTEXT` is used by default (as specified by UIT-T ASN.1 recommendations).

First, an ASN1TaggedObject object is used. Second source code example shows this tagged object wrapping into a *component*. The main difference between a tagged object and a component relies in components optionality (which has no interest in this particular example) and untagged types holding. Indeed, components should be reserved for structured types SEQUENCE and SET, and required components for SEQUENCE OF and SET OF types.

Tagged object version

Source code

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
asn.setValue(99);

ASN1TaggedObject tagged = new ASN1TaggedObject(asn, 0,
  ASN1Class.APPLICATION, true);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);
```

```
aos.encode(tagged);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

tagged.reset(); // Optional operation - performed by "decodeInto()"
dasn.decodeInto(tagged);

System.out.println(tagged);
```

Result

The on-screen printing is as follows:

```
60 03 02 01 63
99
```

Component version

Source code

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
asn.setValue(99);

ASN1TaggedObject tagged = new ASN1TaggedObject(asn, 0,
  ASN1Class.APPLICATION, true);
ASN1Component comp = new ASN1Component(tagged);
// ASN1RequiredComponent could also be used

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(comp);

aos.close();
baos.close();
```

```
// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

comp.reset(); // Optional operation - performed by "decodeInto()"
dasn.decodeInto(comp);

System.out.println(comp);
```

Result

The on-screen printing is as follows:

```
60 03 02 01 63
99
```

This result is identical to the *tagged object* previous example. As no component optionality is taken into account, an ASN1MandatoryComponent could have been used instead of an ASN1Component.

Non-constrained encoding and decoding

This section gives examples of non-constrained types encoding and decoding. Please refer to next section for constrained encoding and decoding examples.

Exampe 3: SEQUENCE (SET)

Description

Consider an ASN.1 specification which identifies an individual using firstname and lastname strings:

```
Person ::= SEQUENCE {
  first  VisibleString,
  last   VisibleString
}
```

The instance manipulted is declared by:

```
Me ::= Person {
  first "Nathanael",
  last  "COTTIN"
}
```

Source code

```
public class Person extends ASN1Sequence {

  public final ASN1VisibleStringfirst  = new ASN1VisibleString();
  public final ASN1VisibleStringlast   = new ASN1VisibleString();

  public Person() {
    super();
  }

  public final List<ASN1Component> getComponents() {
    List<ASN1Component> components = new Vector<ASN1Component>(2);
    components.add(new ASN1Component("first", first));
    components.add(new ASN1Component("last", last));

    return components;
  }

  public static void main(String[] args) {
    try {
      // Encoding process

      Person asn = new Person();

      asn.first.setValue("Nathanael");
```

```
        asn.last.setValue("COTTIN");

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ASN1OutputStream aos = new DEROutputStream(baos);

        aos.encode(asn);

        aos.close();
        baos.close();

        // Encoding printing

        byte[] data = baos.toByteArray();
        System.out.println(ASN1Printer.getHexValue(data, ' '));

        // Decoding process

        ByteArrayInputStream bais = new ByteArrayInputStream(data);
        ASN1InputStream ais = new DERInputStream(bais);
        ASN1DecodedObject dasn = ais.decode();
        ais.close();

        asn = new Person();
        dasn.decodeInto(asn);

        System.out.println(asn.first);
        System.out.println(asn.last);
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
}
```

Result

The on-screen printing is as follows:

```
30 13 1A 09 4E 61 74 68 61 6E 61 65 6C 1A 06 43 4F 54 54 49 4E
Nathanael
COTTIN
```

Example 4: SEQUENCE OF (SET OF)

Description

This example aims at creating a list of individuals following this ASN.1 specification:

```
Persons ::= SEQUENCE OF [0] IMPLICIT Person
-- Reference to 'Person' is given in example #3
```

A list of 5 indiviuals is created, encoded and decoded. Each individual holds an identical first name and last name, except an incremental index suffix.

Source code

```
public class Persons extends ASN1SequenceOf<Person> {

  public Persons() {
    super();
  }

  public final ASN1MandatoryComponent createComponent(Person value)
  {
    // [0] IMPLICIT Person
    return new ASN1MandatoryComponent(
      new ASN1TaggedObject(value, 0));
  }

  public final Person createObject() {
    return new Person();
  }

  public static void main(String[] args) {
    try {
      // Encoding process

      Persons asn = new Persons();
      Person person;

      for (int i = 0; i < 5; i++) {
        person = new Person();
        person.first.setValue("Firstname " + i);
        person.last.setValue("Lastname  " + i);
        asn.add(person);
      }

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ASN1OutputStream aos = new DEROutputStream(baos);

      aos.encode(asn);

      aos.close();
      baos.close();

      // Encoding printing

      byte[] data = baos.toByteArray();
      System.out.println(ASN1Printer.getHexValue(data, ' '));

      // Decoding process
```

```
      ByteArrayInputStream bais = new ByteArrayInputStream(data);
      ASN1InputStream ais = new DERInputStream(bais);
      ASN1DecodedObject dasn = ais.decode();
      ais.close();

      asn = new Persons();
      dasn.decodeInto(asn);

      System.out.println();
      for (ASN1MandatoryComponent comp : asn) {
        person = (Person) comp.getInnerObject();
        System.out.println(person.first);
        System.out.println(person.last);
        System.out.println();
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result

The on-screen printing is as follows:

```
30 81 8C A0 1A 1A 0B 46 69 72 73 74 6E 61 6D 65 20 30 1A 0B 4C 61 73
74 6E 61 6D 65 20 20 30 A0 1A 1A 0B 46 69 72 73 74 6E 61 6D 65 20 31
1A 0B 4C 61 73 74 6E 61 6D 65 20 20 31 A0 1A 1A 0B 46 69 72 73 74 6E
61 6D 65 20 32 1A 0B 4C 61 73 74 6E 61 6D 65 20 20 32 A0 1A 1A 0B 46
69 72 73 74 6E 61 6D 65 20 33 1A 0B 4C 61 73 74 6E 61 6D 65 20 20 33
A0 1A 1A 0B 46 69 72 73 74 6E 61 6D 65 20 34 1A 0B 4C 61 73 74 6E 61
6D 65 20 20 34

Firstname 0
Lastname  0

Firstname 1
Lastname  1

Firstname 2
Lastname  2

Firstname 3
Lastname  3

Firstname 4
Lastname  4
```

Example 5: OpenType (ANY)

Description

This type is a transparent type, which means that it does not appear in the encoding. Many OpenType examples show different ways to encode and decode untagged and tagged OpenType objects.

In most cases, OpenType decodings are differed until a matching component is used. Indeed, full decoding is achieved in two steps:

1. Source input stream decoding into an ASN1DecodedObject temporary object stored within the OpenType
2. Component assignment using the OpenType stored decoded object.

Untagged OpenType

Description

The following source codes all encode and decode an OpenType which contains an INTEGER initialized with `99`. Different variants are suggested.

Source code (first version)

This source code does not remove the inner INTEGER from the OpenType object. Thus, the decoding phase takes this INTEGER from the OpenType.

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
ASN1OpenType openType = new ASN1OpenType();
openType.setInnerComponent(asn);

asn.setValue(99);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(openType);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
```

```
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

openType.reset();

if (dasn.decodeInto(openType)) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("OpenType inner component not initialized");
}
```

Source code (second version)

This source code keeps the inner INTEGER from the OpenType object before decoding (using “reset()” operation). First decoding replaces the OpenType inner INTEGER with a BOOLEAN. As the assignment process fails, this BOOLEAN is replaced with an INTEGER and the assignment succeeds.

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
ASN1OpenType openType = new ASN1OpenType();
openType.setInnerComponent(asn);

asn.setValue(99);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(openType);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

openType.reset();
openType.setInnerComponent(new ASN1Boolean());
```

```
dasn.decodeInto(openType);
// Returns "true" but inner BOOLEAN is not initialized as it
// does not match the decoded value. Calling "openType.decode()"
// returns "false"

if (!openType.decode()) {
  System.out.println("OpenType inner element could not be decoded");
}

openType.setInnerComponent(new ASN1Integer());

// A matching element is found: decoding succeeeds
if (openType.decode()) {
  System.out.println(openType.getInnerComponent());
}
```

Source code (third version)

This source code removes the inner INTEGER from the OpenType object. Thus, the decoding phase does not know in advanced the destinatory of the decoded object.

First, the OpenType inner decoded object is assigned to a BOOLEAN. As a result, this BOOLEAN is not initialized (and `null` is displayed). Then an INTEGER is used to receive the decoded value. Finally, the OpenType is reset. Indeed, any decoding attempt fails, even if an INTEGER is indicated as the destination object.

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
ASN1OpenType openType = new ASN1OpenType();
openType.setInnerComponent(asn);

asn.setValue(99);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(openType);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process
```

```
ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

openType = new ASN1OpenType();

dasn.decodeInto(openType);

if (openType.decodeInto(new ASN1Boolean())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Boolean");
}

if (openType.decodeInto(new ASN1Integer())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Integer");
}

openType.reset();
if (openType.decodeInto(new ASN1Integer())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Integer");
}
```

Results

First version result:

```
02 01 63
99
```

Second version result:

```
02 01 63
OpenType inner element could not be decoded
99
```

Third version result:

```
02 01 63
Unable to decode into ASN1Boolean
99
Unable to decode into ASN1Integer
```

Tagged OpenType

Description

As for all ASN1ExplicitObject objects, OpenType objects tagging is necessarily `EXPLICIT`, i.e. there is no need to call `setExplicit(true)` and `isExplicit()` always returns `true` (contrary to most of tagged ASN.1 objects implicit default declaration).

The following source code relies on the third version of the untagged OpenType previous examples to tag the OpenType `[APPLICATION 0]`.

Source code

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
ASN1OpenType openType = new ASN1OpenType();
openType.setInnerComponent(asn);

ASN1TaggedObject obj = new ASN1TaggedObject(openType, 0,
  ASN1Class.APPLICATION);
// obj is forced to be explicitly tagged: isExplicit() returns true

asn.setValue(99);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(obj);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

openType = new ASN1OpenType();
obj = new ASN1TaggedObject(openType, 0, ASN1Class.APPLICATION);

dasn.decodeInto(obj);
```

```
if (openType.decodeInto(new ASN1Boolean())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Boolean");
}

if (openType.decodeInto(new ASN1Integer())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Integer");
}

openType.reset();
if (openType.decodeInto(new ASN1Integer())) {
  System.out.println(openType.getInnerComponent());
}
else {
  System.out.println("Unable to decode into ASN1Integer");
}
```

Result

The printing corresponds to an EXPLICIT DER encoding of an INTEGER:

```
60 03 02 01 63
Unable to decode into ASN1Boolean
99
Unable to decode into ASN1Integer
```

Example 6: CHOICE

Description

This example indicates the following ASN.1 specification DER-encoding and decoding:

```
Example6 ::= CHOICE {
  asnInt   [1] EXPLICIT   INTEGER,
  asnBool                 BOOLEAN
}
```

The inner element “asnInt” is initialized with `123456789`.

Source code

```
public final class Example6 extends ASN1Choice {

  public final ASN1Integer  asnInt = new ASN1Integer();
```

```
public final ASN1Boolean  asnBool  = new ASN1Boolean();

public Example6() {
  super();
}

public final Set<ASN1MandatoryComponent> getComponents() {
  Set<ASN1MandatoryComponent> components =
    new HashSet<ASN1MandatoryComponent>(2);
  components.add(new ASN1MandatoryComponent(
    "int", new ASN1TaggedObject(asnInt, 1, true)));
  components.add(new ASN1MandatoryComponent("bool", asnBool));

  return components;
}

public static void main(String[] args) {
  try {
    // Encoding process

    Example6 asn = new Example6();

    // Select one ASN.1 object to initialize
    // Initializing both objects leads to choose only
    // one to produce encoding

    asn.asnInt.setValue(123456789);
    // asn.asnBool.setValue(true);

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ASN1OutputStream aos = new DEROutputStream(baos);

    aos.encode(asn);

    aos.close();
    baos.close();

    // Encoding printing

    byte[] data = baos.toByteArray();
    System.out.println(ASN1Printer.getHexValue(data, ' '));

    // Decoding process

    ByteArrayInputStream bais = new ByteArrayInputStream(data);
    ASN1InputStream ais = new DERInputStream(bais);
    ASN1DecodedObject dasn = ais.decode();
    ais.close();
```

```
      asn = new Example6();
      dasn.decodeInto(asn);

      System.out.println(asn.getInnerComponent().getInnerObject());
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Results

asnInt initialization

```
A1 06 02 04 07 5B CD 15
123456789
```

asnBool initialization

```
01 01 FF
true
```

Example 7: ENUMERATED

Although an ENUMERATED is similar to an INTEGER, PowerASN describes ENUMERATED as an abstract class. Indeed, the set of possible values must be declared.

Description

This examples exposes the creation of an ENUMERATED described by the following ASN.1 specification:

```
MyASN1Enum ::= ENUMERATED {
  SUCCESS  (0),
  ERROR    (-1)
}
```

This ENUMERATED is set to SUCCESS and DER-encoded and decoded.

Source code

```
public final class MyASN1Enum extends ASN1Enumerated {

  public static final BigInteger  SUCCESS  = BigInteger.valueOf(0);
  public static final BigInteger  ERROR    = BigInteger.valueOf(-1);

  public MyASN1Enum() {
    super();
  }
```

```
public MyASN1Enum(String name) {
  super(name);
}

@Override
public Set<BigInteger> getValues() {
  Set<BigInteger> set = new HashSet<BigInteger>(2);
  set.add(SUCCESS);
  set.add(ERROR);

  return set;
}

public static void main(String[] args) {
  try {
    // Encoding process

    MyASN1Enum asn = new MyASN1Enum();
    asn.setValue(MyASN1Enum.ERROR);

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ASN1OutputStream aos = new DEROutputStream(baos);

    aos.encode(asn);

    aos.close();
    baos.close();

    // Encoding printing

    byte[] data = baos.toByteArray();
    System.out.println(ASN1Printer.getHexValue(data, ' '));

    // Decoding process

    ByteArrayInputStream bais = new ByteArrayInputStream(data);
    ASN1InputStream ais = new DERInputStream(bais);
    ASN1DecodedObject dasn = ais.decode();
    ais.close();

    MyASN1Enum decodedAsn = new MyASN1Enum();
    dasn.decodeInto(decodedAsn);

    System.out.println(decodedAsn.toString());
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
}
```

Result

```
0A 01 FF
-1
```

Example 8: time management

Description

This example describes the DER encoding and decoding processes of a UTCTime ASN.1 object initialized one second before december 25th, 2006. This time value is expressed in Zulu time (ends with a "Z"). Setting a non-existing time value would raise a constraint exception (this default behavior can be disabled with `asn.setTimeValidation(false)`). Last screen-printed line demonstrates that the appropriate time format is deduced from the encoded value.

Source code

```
// Encoding process

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

ASN1Time asn = new ASN1UTCTime();
asn.setValue("061224235959Z");
// Time format automatically switches to FORMAT_1
// Use "asn.setValue()" to initialize with current time

aos.encode(asn);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

asn = new ASN1UTCTime();
dasn.decodeInto(asn);
```

```
System.out.println(asn + " (" + asn.getTimeFormat().name() + ")");
```

Result

```
17 0D 30 36 31 32 32 34 32 33 35 39 35 39 5A
0612242359592 (FORMAT_1)
```

Using default values

Default values are used by ASN.1 encoding and decoding rules to determine whether a given ASN.1 object produces an encoding or is considered as optional (even if not specifically set optional) and therefore does not produce an encoding.

Most of ASN.1 encoding rules do not produce encodings for default valued object with either no value or similar value and default value.

Example 9: encoding a default valued ASN.1 INTEGER

Description

Let us manipulate an ASN.1 INTEGER with an internal value which equals its default value of `10`. Considering DER, no encoding is produced. A similar result could be obtained when assigning no value but a default value to this INTEGER.

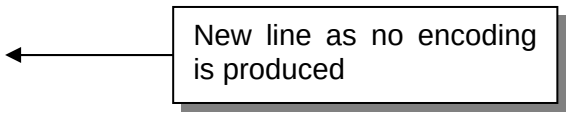
The decoding process takes an empty input stream to assign a value. Thus, no value is set to this INTEGER but its default value is kept.

Source code

```
// Encoding process

ASN1Integer asn = new ASN1Integer();

asn.setValue(10);
asn.setDefaultValue(10);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);

aos.close();
baos.close();

// Encoding printing: no encoding is produced

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process
```

```
ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

asn.reset(); // Removes the value (but not the default value)
dasn.decodeInto(asn);

// No value is assigned by the decoding process...
System.out.println(asn.getValue());

// ...default value must be used
System.out.println(asn.getValueOrDefault());
```

Result




```
null
10
```

Example 10: encoding a default valued ASN.1 SEQUENCE

Description

Consider the given SEQUENCE Person (based on previous declaration):

```
DefaultPerson ::= SEQUENCE {
  first      VisibleString DEFAULT "Nathanael",
  last   [0] VisibleString DEFAULT "COTTIN"
}
```

Note that “last” component must be tagged to avoid ambiguous decoding. Please refer to PowerASN tutorials for further information.

This example indicates how to assign a default value and a value which equals this default value. As a result, no DER encoding is produced and the decoded SEQUENCE holds only its default value.

Both value and default value take the following values:

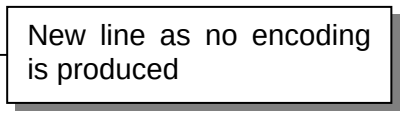
```
myself ::= DefaultPerson {
  first "Nathanael",
  last  "COTTIN"
}
```

Source code

```
public class DefaultPerson extends ASN1Sequence {

  public final ASN1VisibleStringfirst  = new ASN1VisibleString();
  public final ASN1VisibleStringlast= new ASN1VisibleString();

  public DefaultPerson() throws ASN1ConstraintException {
    super();
    first.setDefaultValue("Nathanael");
    last.setDefaultValue("COTTIN");
  }

  public final List<ASN1Component> getComponents() {
    List<ASN1Component> components = new Vector<ASN1Component>(2);
    components.add(new ASN1Component("first", first));
    components.add(new ASN1Component("last",
      new ASN1TaggedObject(last, 0)));

    return components;
  }

  public static void main(String[] args) {
    try {
      // Encoding process

      DefaultPerson asn = new DefaultPerson();

      asn.first.setValue("Nathanael");
      asn.last.setValue("COTTIN");

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ASN1OutputStream aos = new DEROutputStream(baos);

      aos.encode(asn);

      aos.close();
      baos.close();

      // Encoding printing

      byte[] data = baos.toByteArray();
      System.out.println(ASN1Printer.getHexValue(data, ' '));

      // Decoding process

      ByteArrayInputStream bais = new ByteArrayInputStream(data);
      ASN1InputStream ais = new DERInputStream(bais);
      ASN1DecodedObject dasn = ais.decode();
      ais.close();
```

```
      asn = new DefaultPerson();
      dasn.decodeInto(asn);

      // Default values are used for printing
      System.out.println(asn.first);
      System.out.println(asn.last);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result




```
Nathanael
COTTIN
```

Example 11: encoding an empty default valued SEQUENCE OF

Description

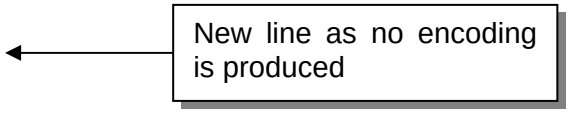
A particular case concerns empty SEQUENCE OF or SET OF objects.

Source code

```
// Encoding process

Persons defaultAsn = new Persons();
// defaultAsn is an empty SEQUENCE OF Person

Persons asn = new Persons();
asn.setDefaultValue(defaultAsn);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);

aos.close();
baos.close();

// Encoding printing: no encoding produced

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process
```

```
ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

asn.reset(); // Keep default value only
dasn.decodeInto(asn);

System.out.println(asn.isEmpty());
if (asn.hasDefaultValue()) {
  System.out.println(asn.isDefaultEmpty());
}
```

Result




```
true
true
```

Without assigning “asn” with a default value, the result would have been:

```
30 00
true
```

Input streams security checks

Description

ASN.1 input streams used to decode ASN.1 objects can be constrained with a maximum length (number of octets) allowed for decoding as well as an end-of-stream check as well as a maximum waiting delay (in milliseconds). These constraints may be specified when constructing the ASN.1 input stream or using the corresponding setters.

Maximum length constraint

Mentioning a positive maximum length tells the ASN.1 input stream to process decoding until either end of decoding or if this upper size limit is reached. This constraint is disabled using `ASN1Constants.NO_VALUE`.

Description

The following piece of code DER-encodes a VisibleString and sets a maximum length lower than the encoding length. As a result, an `ASN1SecurityException` is raised.

Source code

```
// Encoding process

ASN1VisibleString asn = new ASN1VisibleString();
asn.setValue("Constrained stream test");

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais, 24);
ASN1DecodedObject dasn = ais.decode(); // Fails here
ais.close();
```

Result

```
pasn.error.ASN1SecurityException: Attempting to decode more bytes than allowed
```

End of stream verification

ASN.1 input streams can make sure that the decoded information is not followed by extra octets (see Appendix A). When not specified, PowerASN provided ASN.1 input streams assume that this security check must be performed. This check is bypassed setting `extraAllowed` constructors argument to `true` or specifying `setExtraAllowed(true)` before starting decoding.

Description

The following piece of code DER-encodes a VisibleString and adds an extra byte to the produced encoding. As a result, an `ASN1SecurityException` is raised.

Source code

```
// Encoding process

ASN1VisibleString asn = new ASN1VisibleString();
asn.setValue("Constrained stream test");

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);

aos.close();
baos.close();

// Encoding printing

byte[] data = baos.toByteArray();

byte[] data2 = new byte[data.length + 1];
System.arraycopy(data, 0, data2, 0, data.length);
data2[data.length] = (byte) 0x00;
data = null;

System.out.println(ASN1Printer.getHexValue(data2, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data2);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();   // Fails here
ais.close();
```

Result

```
pasn.error.ASN1SecurityException: End of stream expected
```

Maximum waiting delay

When reading from an input stream, non-secured decoders may indefinitely wait if the issuer does not send expected data. As a result, the whole application way be frozen until either the stream closes or any datum is received.

To avoid this blocking state, default PowerASN decoders (which inherit from the abstract class `ASN1InputStream`) make use of a maximum waiting delay before generating a `TimerException` error.

This delay is specified either during ASN.1 input streams construction or by means of `setDelay(long, long)`.

As generating this behaviour is not straightforward (required a client-server communication with only part of the information transmitted by the server), no example is provided here.

Constrained encoding and decoding

PowerASN also allows constraining ASN.1 simple and structured types (all but OpenType). Contrary to stream constraints, the error occurs when setting the decoded value to the receiver rather than when calling the `decode()` operation.

String types constraints make a difference between length constraints (minimum and maximum number of characters allowed) and characters values (restrictions on alphabets: NumericString objects must reject non numeric characters for example).

Default character values constraints are hidden to the developer and thus cannot be modified.

All structured collections (SEQUENCE OF and SET OF) make use of customized size constraints (minimum and maximum number of elements to decode). SEQUENCE and SET size constraints are automatically handled.

The examples given hereafter indicate the way to constrain a VisibleString, an INTEGER and a SEQUENCE OF.

Constrained string

Description

This example sets a VisibleString minimum length of `30` to decode to. As the encoded string value length (`23`) is less than the required minimum length, an `ASN1ConstraintException` is raised.

Default behavior accepts authorized characters only. Complementary checks can be performed by means of `validate(char, int)` redefinition and regular expression matching.

Source code

```
// Encoding process

ASN1VisibleString asn = new ASN1VisibleString();
asn.setValue("Constrained string test");

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);
```

```
aos.close();
baos.close();


// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

ASN1VisibleString decodedAsn = new ASN1VisibleString();
decodedAsn.setMinimumLength(30);
dasn.decodeInto(decodedAsn);  // Fails here
```

Result

```
1A 17 43 6F 6E 73 74 72 61 69 6E 65 64 20 73 74 72 69 6E 67 20 74 65
73 74
pasn.error.ASN1ConstraintException: String value length is less than
minimum required
```

Constrained INTEGER

Description

An INTEGER is created and initialized with value `50`. The corresponding DER-encoding is generated. The INTEGER used to assign the decoded object with is constrained with a maximum value `40`. Indeed, the assignment process raises a constraint error.

Source code

```
// Encoding process

ASN1Integer asn = new ASN1Integer();
asn.setValue(50);

ByteArrayOutputStream baos = new ByteArrayOutputStream();
ASN1OutputStream aos = new DEROutputStream(baos);

aos.encode(asn);

aos.close();
baos.close();
```

```
// Encoding printing

byte[] data = baos.toByteArray();
System.out.println(ASN1Printer.getHexValue(data, ' '));

// Decoding process

ByteArrayInputStream bais = new ByteArrayInputStream(data);
ASN1InputStream ais = new DERInputStream(bais);
ASN1DecodedObject dasn = ais.decode();
ais.close();

ASN1Integer decodedAsn = new ASN1Integer();
decodedAsn.setMaximumValue(40);
dasn.decodeInto(decodedAsn);  // Fails here
```

Result

```
02 01 32
pasn.error.ASN1ConstraintException: INTEGER value is greater than maximum value allowed
```

Constrained SEQUENCE OF

Description

This example shows how to set a SEQUENCE OF maximum number of inner components. First, a SEQUENCE OF BOOLEAN holding `5` inner elements is DER-encoded. The produced encoding is then submitted to another SEQUENCE OF BOOLEAN constrained with a maximum of `4` inner elements. Inner BOOLEAN elements alternatively take `TRUE` and `FALSE` values. As a result, a constraint exception is raised.

Source code

```
public class ConstrainedSequenceOf extends
  ASN1SequenceOf<ASN1Boolean> {

  public final ASN1MandatoryComponent createComponent(
    ASN1Boolean asn) {
    return new ASN1MandatoryComponent(asn);
  }

  public final ASN1Boolean createObject() {
    return new ASN1Boolean();
  }

  public static void main(String[] args) {
    try {
```

```
      // Encoding process

      ConstrainedSequenceOf asn = new ConstrainedSequenceOf();
      ASN1Boolean bool;
      boolean val = true;
      for (int i = 0; i < 5; i++) {
        bool = new ASN1Boolean();
        bool.setValue(val);
        asn.add(bool);
        val = !val;
      }

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ASN1OutputStream aos = new DEROutputStream(baos);

      aos.encode(asn);

      aos.close();
      baos.close();

      // Encoding printing

      byte[] data = baos.toByteArray();
      System.out.println(ASN1Printer.getHexValue(data, ' '));

      // Decoding process

      ByteArrayInputStream bais = new ByteArrayInputStream(data);
      ASN1InputStream ais = new DERInputStream(bais);
      ASN1DecodedObject dasn = ais.decode();
      ais.close();

      ConstrainedSequenceOf decodedAsn =
        new ConstrainedSequenceOf();
      decodedAsn.setMaximumSize(4);
      dasn.decodeInto(decodedAsn);  // Fails here
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result

```
30 0F 01 01 FF 01 01 00 01 01 FF 01 01 00 01 01 FF
pasn.error.ASN1ConstraintException: Structured object length is greater
than maximum allowed
```

Setting specific encoding rules

Some specifications require multiple encoding rules to properly encode ASN.1 types. For example, RFC 2630 (“Cryptographic Message Syntax”) specifies DER encoding of signed attributes within BER encodings.

A specific set of encoding rules “A” called during an encoding process using “B” encoding rules if called “A-over-B”.

PowerASN for Java handles this particular issue with specific encodings. Many sub-encodings can be defined depending on the parent rules (i.e. the “caller”) as DER-over-BER is not necessarily applicable with other calling rules (XER for example should not use DER if not specified).

Description

The following example shows a DER-over-BER encoding of a simple SEQUENCE defined as follows:

```
DER-over-BER ::= SEQUENCE {
  berEncoded INTEGER,
  derEncoded INTEGER
  -- Encoded using DER
}
```

An indefinite-length encoding is performed to demonstrate that DER apply to the second component of this SEQUENCE, as DER uses definite-length encodings only.

Source code

```
public class DER_over_BER extends ASN1Sequence {

  public final ASN1Integer  berEncoded = new ASN1Integer();
  public final ASN1Integer  derEncoded = new ASN1Integer();

  public DER_over_BER() {
    super();

    // Specifies that DER must be applied when BER is currently used
    derEncoded.setSpecificEncoding(BEROutputStream.id,
      new DEROutputStream());
  }

  public final List<ASN1Component> getComponents() {
    List<ASN1Component> components = new Vector<ASN1Component>(2);
    components.add(new ASN1Component("berEncoded", berEncoded));
    components.add(new ASN1Component("derEncoded", derEncoded));
```

```
    return components;
  }

  public static void main(String[] args) {
    try {
      // Encoding process

      DER_over_BER asn = new DER_over_BER();

      asn.berEncoded.setValue(1);
      asn.derEncoded.setValue(2);

      ByteArrayOutputStream baos = new ByteArrayOutputStream();
      ASN1OutputStream aos = new BEROutputStream(baos, true);

      aos.encode(asn);

      aos.close();
      baos.close();

      // Encoding printing

      byte[] data = baos.toByteArray();
      System.out.println(ASN1Printer.getHexValue(data, ' '));

      // Decoding process

      ByteArrayInputStream bais = new ByteArrayInputStream(data);
      ASN1InputStream ais = new BERInputStream(bais);
      ASN1DecodedObject dasn = ais.decode();
      ais.close();

      asn = new DER_over_BER();
      dasn.decodeInto(asn);

      System.out.println(asn.berEncoded);
      System.out.println(asn.derEncoded);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Result

```
30 80 02 80 01 00 00 02 01 02 00 00
1
2
```

Creating your own material

As an open environment, PowerASN for Java is made to facilitate the integration of new types and ASN.1 codecs. New types (or codecs) may either replace existing types (resp. codecs) or complete existing ones. Following this vision, PowerASN provided (default) codecs only refer to Java interfaces and not directly to implemented types.

ASN.1 types creation

ASN.1 specification define simple, structured and transparent types as follows:

- Simple types refer to INTEGER, BOOLEAN, ENUMERATED, Strings, etc.: these types enclose a single, well-known value
- Structured types refer to SEQUENCE, SET and collections (SEQUENCE OF, SET OF): they enclose many inner ASN.1 types. Inner elements ordering may be required (SEQUENCE, SEQUENCE OF) or not taken into account (SET, SET OF)
- Transparent types are placeholders for other types. They basically refer to OpenType (single) and CHOICE (alternative) types.

Integration requirements

Next table indicates the required interfaces implementations to ensure existing codecs ability to encode and decode new types implementations:

Types	Forms	Interfaces
Simple	PRIMITIVE	`ASN1PrimitiveObject`
	UNDEFINED	`ASN1PrimitiveOrConstructedObject`
Structured (but collection)	CONSTRUCTED	`ASN1StructuredObject` `ASN1OrderedObject` or `ASN1UnorderedObject`
Collection	CONSTRUCTED	`ASN1StructuredMandatoryObject` `ASN1OrderedObject` or `ASN1UnorderedObject`
Transparent (single)	UNDEFINED	`ASN1TransparentObject`
Transparent (alternative)	UNDEFINED	`ASN1AlternativeObject`

Examples

Creating an INTEGER (in place of the existing PowerASN `ASN1Integer` class) requires inheriting from the `ASN1PrimitiveObject` interface.

An OpenType needs to derive the `ASN1TransparentObject` interface.

A CHOICE is represented as an `ASN1AlternativeObject` object.

A SET OF must inherit from both `ASN1StructuredMandatoryObject` and `ASN1UnorderedObject` to respectively indicate that OPTIONAL inner elements and ordering are not supported.

ASN.1 codecs creation

Developers can create their own ASN.1 codecs independently from existing codecs and relying on existing ASN.1 types or ad-hoc types.

PowerASN for Java provided codecs are decomposed into encoders and decoders.

Coders creation

`ASN1OutputStream` abstract class is the root class for default encoders. It provides functionalities for destination streams and files management as well as non-optional ASN.1 objects and components encodings.

This output stream is linked with a value encoder. This value encoder returns PRIMITIVE (simple) types encoded values. Thus, a DER encoder makes use of a DER value encoder. This value encoder should refer to ASN.1 types (as defined by ASN.1 specifications and accessible with "`ASN1GenericObject.getType()`") and not their Java mappings to ensure user-defined types encoding ability.

ASN.1 objects encoding is decomposed depending on their form (PRIMITIVE, CONSTRUCTED, UNDEFINED), transparency (OpenType and CHOICE) and tagging.

Decoders creation

Although PowerASN for Java allows building ASN.1 decoders from scratch, `ASN1InputStream` abstract class is provided as a root class for decoders. A value decoder is registered within this input stream. This value decoder decodes PRIMITIVE (simple) types encoded values. Similarly to values encoders, values decoders should refer to ASN.1 types and avoid PowerASN (or third parties) implementations.

Security being a key feature of PowerASN, any decoder which derives from this input stream can be automatically constrained with a maximum length (maximum number of decoded octets) as well as remaining trailing octets check and a timeout, as far as "readByte(…)" operations are used to read data from the encoded input stream.

As a result, an `ASN1DecodedObject` is generated. Currently, PowerASN for Java includes three decoded objects adapted to supported decoders:

- `ASN1UnnamedDecodedObject`: generated by TLV-based decoders, such as BER and DER decoders
- `ASN1NamedDecodedObject`: originally returned by XER decoders (named XML elements)
- `ASN1EmptyDecodedObject`: this particular object should be returned by any decoder when no data is available from the input stream. An empty decoded object can be successfully decoded to an ASN.1 (tagged) type depending on encoding rules. Taking DER, default valued types must not be encoded and thus produce no encoding. Decoding is indeed performed on an empty input stream.

ASN.1 decoded objects creation

PowerASN decoding is split in two complementary phases:

1. Encoded input stream decoding to produce an ASN.1 decoded object
2. ASN.1 decoded object assignment to an ASN.1 (tagged) type.

Previous section demonstrated how input streams decoding can be defined when creating decoders.

PowerASN for Java manipulates types, tagged objects and (possibly mandatory) components:

- An ASN.1 type inherits from the root interface ASN1GenericObject
- A tagged object refers to a tagged type or a recursively tagged type (type tagged more than once)
- A mandatory component is a non-optional component, basically used within collections (SEQUENCE OF, SET OF), alternative (CHOICE) and OpenType
- A component holds either a type or a tagged object. It can be declared OPTIONAL. It is primarily defined to be used within structured objects (SEQUENCE, SET) where inner types can be declared OPTIONAL and tagged.

ASN.1 decoded objects require implementing the following operations:

- `refersTo(…)`: makes sure that this decoded object hold a value compatible with the given ASN.1 type or tagged object
- `decodeInto(…)`: performs the value assignment. This operation should first call “`refersTo(…)`” to ensure that decoded value and its final destination do match.

Compiling ASN.1 specifications

Advantages

ASN.1 specifications (generally described in `.asn` files) must be transformed into programmable objects and linked with encoding and decoding processes.

On the one hand, developers can create these objects relying on existing PowerASN types and transposing specifications into self-made source code. This process can be automated using a (language-specific) compiler. Please refer to appendix C.

PowerASN includes a compiler to create Java classes from ASN.1 specifications. This compiler is not ready at this time and thus not included in the current PowerASN for Java distribution.

Requirements

PowerASN for Java ASN.1 specifications compiler is an extra package which can be downloaded for free at *http://www.powerasn.com*. It is not integrated within PowerASN for Java global distribution for two main reasons:

- It can be modified (evolve) independently from ASN.1 types and codecs
- There is no need to integrate this compiler into applications which rely on PowerASN for Java as ASN.1 specifications compilation can be considered as an external process.

PowerASN for Java compiler projects ASN.1 specifications into default types implementations. Therefore, user-defined types are not handled.

Conclusion

Reaching the best

PowerASN or Java is in constant evolution to improve its security and performances and provide developers with an extensive environment (including documentation efforts and complementary tools).

BER particularities

BER allows indefinite-length encodings. Developers can specify whether definite-length or indefinite-length applies.

Moreover, BIT STRING and OCTET STRING encoding form (PRIMITIVE or CONSTRUCTED) can be specified along with the maximum size (in octets) of each constructed part.

XER particularities

The provided examples make use of DER to encode and decode ASN.1 objects. Applying other rules is straightforward, although XER requires named ASN.1 objects. This modification refers to ASN.1 structured types components.

Structured objects (SEQUENCE OF and SET OF collections) inner elements (components) must be named to be correctly XER-encoded and decoded as PowerASN takes their ASN.1 name by default (INTEGER, BOOLEAN , etc.).

Collections mandatory components may not be named as ITU-T X.693 requires to make use of:

- ASN.1 primitive types ASN.1 names (INTEGER, BOOLEAN, ENUMERATED, NumericString, GeneralizedTime, etc.)
- ASN.1 structured types objects names, which corresponds to PowerASN structured types derived objects Java classes names (example: `Person`, `Persons`).

XER encoding needs a XEROutputStream instance. This instance can be created as follows:

```
ASN1OutputStream aos = new XEROutputStream(out, false, true);
```

where:

- `out` refers to any output stream
- `false` indicates that basic XER is used (as opposed to canonical XER)
- `true` specifies that XER XML prolog must be written prior to ASN.1 types encodings.

Contact

For any comment, suggestion or bug fix related to current PowerASN for Java implementation, please refer to PowerASN website (http://www.powerasn.com) to get the appropriate email address.

Appendix

Appendix A: input streams validity checking

PowerASN detects malformed ASN.1 BER, CER and DER encoded streams and raises exceptions during decoding. Here is a straightforward example.

Consider this ASN.1 specification:

```
Person ::= SEQUENCE {
  first  VisibleString,
  last   VisibleString
}
```

and the following instance :

```
Me ::= Person {
  first "Nathanael", last "COTTIN"
}
```

A valid DER encoding produces

30 13 1A 09 4E 61 74 68 61 6E 61 65 6C 1A 06 43 4F 54 54 49 4E

Append extra information

Extra information can be added at the end of the encoded data. Taking the previous example, appending two octets (01_{16} and 02_{16}) may be as follows:

30 13 1A 09 4E 61 74 68 61 6E 61 65 6C 1A 06 43 4F 54 54 49 4E 01 02

Insert extra information

Slight modifications allow to integrate an extra letter, say ‘A’, underlined (single line):

30 14 1A 09 4E 61 74 68 61 6E 61 65 6C 61 1A 06 43 4F 54 54 49 4E

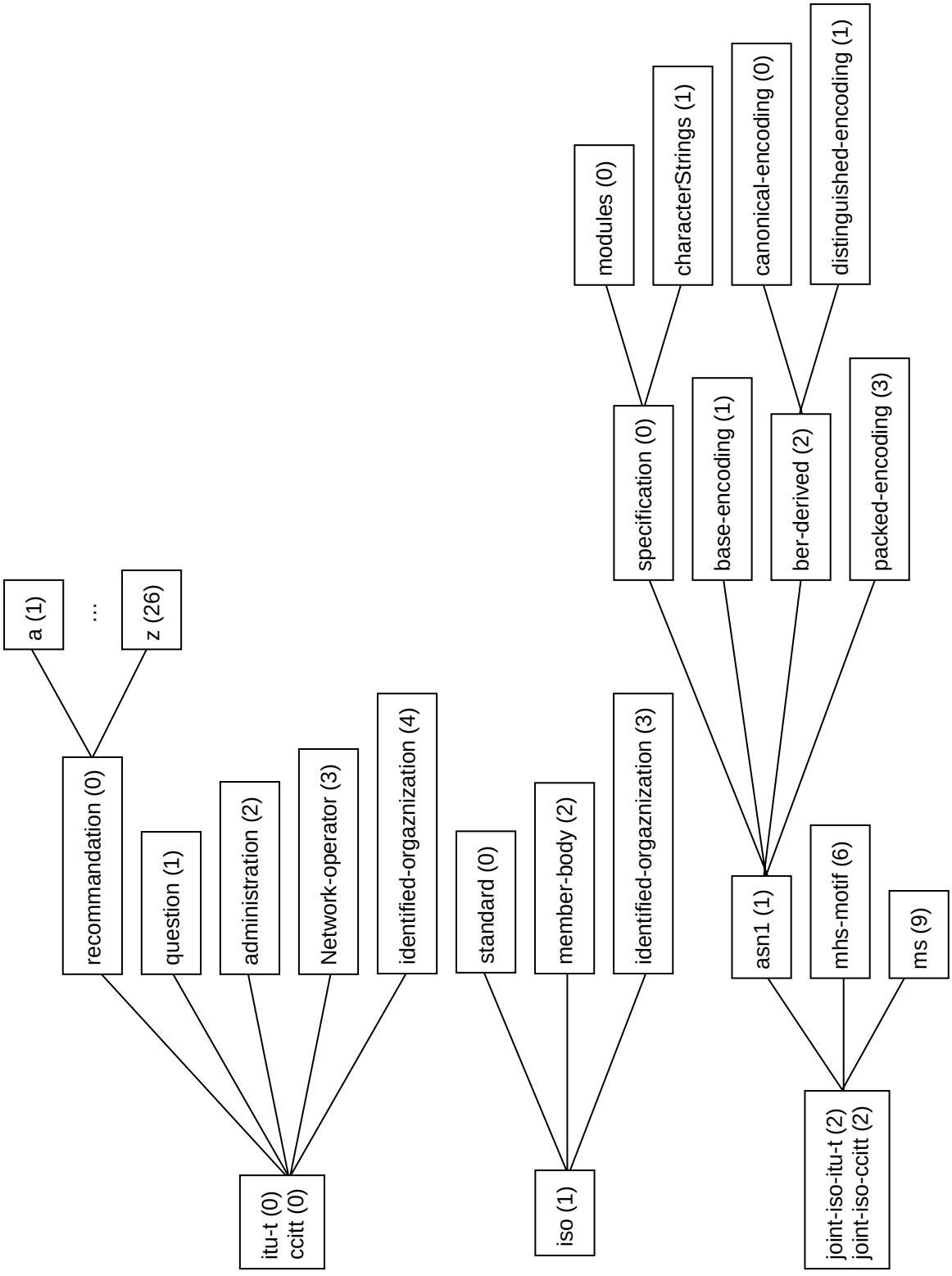
To keep the overall validity of the encoding, length of the `SEQUENCE` type has been changed from 13_{16} to 14_{16} (double underlined).

Contrary to many decoders, PowerASN checks the inner component decoded length (independently from definite or indefinite length encoding) against its container expected length. Any difference between the effective decoded length and the expected length throws an exception.

Thus, the sum of ‘Person’ inner components lengths equals $0B_{16} + 08_{16} = 13_{16}$, which does not correspond to ‘Person’ expected length of 14_{16}.

Please refer to ASN.1 security issues presentation available on-line at www.powerasn.com for further attacks and details.

Appendix B: ISO object identifiers registration tree



Appendix C: PowerASN processes