JZOS COBOL Record Generator

User's Guide
Version 2.2.1

Title: JZOS COBOL Record Generator
Version: 2.2.1
First edition: 18 October 2007

Contents

Overview

The JZOS COBOL Record Generator is a Java class library that generates Java classes to describe COBOL records.

Features

- Reads a “SYSADATA” file created by the z/OS Enterprise COBOL compiler and processes a selected record (group item). By the default, the first group item encountered is processed. Typically, a tiny stub program is compiled which simply compiles a COBOL “copy book” containing the record description.
- Generates a “.java” source file for a Java class which can be used to map a Java byte array containing a record matching the COBOL data descriptions.
- Uses the ***com.ibm.jzos.fields*** package, which contains data-type converters for *most* elemental COBOL data types. The ***com.ibm.jzos.fields*** package is part of the IBM JZOS Tookit and is required to both generate and execute the Java classes. The generation class library itself is not required to run the generated classes.
- The generated Java “record” classes may be used in conjunction with the IBM JZOS batch toolkit to read and write z/OS datasets which are also processed by COBOL applications described by the source COBOL “copy books”.

Getting Started

Installation

The JZOS COBOL RecordClassGenerator is included in the alphaWorks distribution of the IBM JZOS Toolkit. Refer to the "JZOS alphaWorks Installation" manual for installation instructions.

Running the COBOL RecordClassGenerator

- Follow the instructions in "JZOS alphaWorks Installation" for installing the package, and setting up the necessary CLASSPATH. Note that the JZOS alphaWorks JAR files must be configured as "extension" JARs since they have newer versions of some classes that appear in the JZOS jar in the z/OS Java SDK. The only JAR file that is actually needed to run the RecordClassGenerator is "jzos_recgen.jar".
- Run the z/OS Enterprise COBOL compiler with the "ADATA" option to create a "SYSADATA" file.
- Run the "com.ibm.jzos.recgen.cobol.RecordClassGenerator" class specifying the following arguments:
  - adataFile=*adataFileName* (default: //DD:SYSADATA)
  - outputDir=*dirname* (default: write .java file to System.out)
  - symbol=*RECORD-NAME* (default: first group data item found)
  - packageName=*fully.qfied.pkg.name* (default: "mypackage")
  - class=*className* (default: the top-level record name)
  - bufoffset=*true*|*false* (default: false. See "bufOffset feature")
  - genSetters=*true*|*false* (default: true)
  - genCache=*true*|*false* (default: true)
  - stringTrim=*true*|*false* (default: false)
  - genAccessorJavadoc=*true*|*false* (default: false)

  If outputDir is given, the output file is created in the package-qualified subdirectory with a file name "*className.java*". For example, if outputDir=*/u/myid* and package=*com.myco.cobrecs* and class=*MyClass* or implicitly determined from the copybook record MY-RECORD, then the output file created will be: */u/myid/com/myco/cobrecs/MyClass.java.*

  If the given adatafile is a pathname not starting with "//", then it is assumed to be the name of a binary file in the USS (or workstation) filesystem containing the binary SYSADATA file. In order to process SYSADATA files on the workstation, they

must be downloaded with IBM "RDWs" included. This can be accomplished using FTP along with the "bin" and "quote site rdw" commands".

If the given adatafile is a PDS name without a member or a directory name, then all members of the PDS (or files in the directory) are processed. For this to work properly, the outputDir option must be specified without the class and symbol options so that the copybook top-level record name is used to generate unique classnames for each class. This feature allows for one invocation of the RecordClassGenerator to generate Java classes for an entire PDS library of SYSADATA files. (See the example job below)

The RecordClassGenerator will write the generated Java source file to System.out, so this file should be redirected to the file where you want to store your Java source. This would be something like: `src/com/myco/records/MyClass.java`, assuming that `packageName=com.myco.records` and `class=MyClass`.

The `genCache` parameter can be used to control whether code is generated to use instance variables to cache field values. Regardless of this setting, fields that are the objects (sizes) of OCCURS DEPENDING ON clauses are always cached.

The `stringTrim` parameter controls whether the generated code configures its DataTypeFactory to build trimmed StringFields.

The `genAccessorJavadoc` parameter can be used to cause comment Javadoc to be generated for field getter and setter methods to include the same COBOL source lines that are used to tag the static Field declarations. If set to false, no comments are generated for Field getter and setter methods.

RecordClassGenerator Example Job

The following JCL can be used to run the z/OS Enterprise COBOL Compiler to create a "SYSADATA" files and translate them into Java source files. This JCL can be found in the "COBGEN" sample JCL which is shipped with the alphaWorks version of JZOS.

```
//XXXXXXXX JOB (),NAME
//*
//*********************************************************************
//*
//* Batch job to generate a Java class from a COBOL Copy Book
//*
//* Tailor the proc and job for your installation:
//* 1.) Modify the Job card per your installation's requirements
//* 2.) Customize the JCL SET variables
//* 3.) edit JAVA_HOME to point to the location of the Java SDK
//* 4.) edit JZOSAW_HOME to point to the JZOS alphaWorks directory
//* 5.) Modify MAINARGS DD arguments to RecordClassGenerator
//*
//*********************************************************************
//*
// SET COBPRFX='DDESHR.COB340'
// SET CPYBKLIB='JZOSAW.SAMPLE.JCL'
//*
//* Inline proc to compile a COBOL copy book to a temp ADATA PDS
//COBADATA PROC CPYBOOK=
//COBCC EXEC PROC=IGYWC,LNGPRFX=&COBPRFX,PARM='LIB,ADATA,MAP'
//COBOL.SYSLIB DD DISP=SHR,DSN=&CPYBKLIB
//COBOL.SYSADATA DD DSN=&&ADATA(&CPYBOOK),
//             DISP=(MOD,PASS),SPACE=(CYL,(3,3,10))
//COBOL.SYSIN  DD DSN=&CPYBKLIB(COBPREFX),DISP=SHR   << COBOL PGM BEGIN
//             DD DSN=&CPYBKLIB(&CPYBOOK),DISP=SHR   << COPYBOOK
//             DD DSN=&CPYBKLIB(COBSUFIX),DISP=SHR   << COBOL PGM END
//  PEND
//*
//* Compile one or many copy books to temp &ADATA PDS
// EXEC COBADATA,CPYBOOK=MEDICARE
// EXEC COBADATA,CPYBOOK=FIELDS
//*
//* Generate a .java file for each copy book
//JAVA EXEC PROC=EXJZOSVM,VERSION='50'
//MAINARGS DD *
com.ibm.jzos.recordgen.cobol.RecordClassGenerator
  bufoffset=false
  package=com.myco.cobol.records
  outputDir=~/cobgen

//SYSADATA DD DSN=&&ADATA,DISP=(OLD,DELETE)
```

(continued on next page)

```
//STDENV DD *
# This is a shell script which configures
# any environment variables for the Java JVM.
# Variables must be exported to be seen by the launcher.

. /etc/profile

#############################################################
# Customize below to match your installation:
# JAVA_HOME - The location of the SDK
# JZOSAW_HOME - The location of the JZOS alphaworks preview
#############################################################
export JAVA_HOME=<JAVA_HOME>
JZOSAW_HOME=<JZOSAW_HOME>

export PATH=/bin:"${JAVA_HOME}"/bin:

LIBPATH=/lib:/usr/lib:"${JAVA_HOME}"/bin
LIBPATH="$LIBPATH":"${JAVA_HOME}"/bin/classic
LIBPATH="$LIBPATH":"${JZOSAW_HOME}"
export LIBPATH="$LIBPATH":

# Force the alphaWorks version of JZOS to run overriding the SDK
OPTS="-Djava.ext.dirs=${JZOSAW_HOME}:${JAVA_HOME}/lib/ext"
export JZOS_JVM_OPTIONS="$OPTS"
# Add the JZOS alphaWorks jars to the classpath
for i in "${JZOSAW_HOME}"/*.jar; do
   CLASSPATH="$CLASSPATH":"$i"
    done
export CLASSPATH="$CLASSPATH":

# Set JZOS specific options
# Use this variable to specify encoding for DD STDOUT and STDERR
#export JZOS_OUTPUT_ENCODING=Cp1047
# Use this variable to prevent JZOS from handling MVS operator commands
#export JZOS_ENABLE_MVS_COMMANDS=false
# Use this variable to supply additional arguments to main
#export JZOS_MAIN_ARGS=""

# Configure JVM options
IJO="-Xms16m -Xmx128m"
//
```

Sample output:

```
Generating //DD:SYSADATA(FIELDS) to:
    /u/myid/cobgen/com/myco/cobol/records/FieldsRecord.java
Generating //DD:SYSADATA(MEDICARE) to:
    /u/myid/cobgen/com/myco/cobol/records/MedicareRecord.java
```

Example Copy Book

Using the preceding sample JCL, the following COBOL copybook is processed:

```
       01  MEDICARE-RECORD.
           05 CLAIM-NUMBER              PIC X(19).
           05 ADMISSION-DATE            PACKED-DECIMAL PIC S9(7).
           05 FROM-DATE                 PACKED-DECIMAL PIC S9(7).
           05 THRU-DATE                 PACKED-DECIMAL PIC S9(7).
           05 DISCHARGE-DATE            PACKED-DECIMAL PIC S9(7).
           05 FULL-DAYS                 PACKED-DECIMAL PIC S9(5).
           05 COINSURANCE-DAYS          BINARY         PIC 9(4).
           05 LIFETIME-RES-DAYS         BINARY         PIC 9(6).
           05 INTERMEDIARY-NUM          BINARY         PIC 9(10).
           05 MEDICARE-PROVIDER                        PIC X(13).
           05 INPATIENT-DED             PACKED-DECIMAL PIC S9(4)V99.
           05 BLOOD-DED                 PACKED-DECIMAL PIC S9(4)V99.
           05 TOTAL-CHARGES      PIC S9(7)V99 DISPLAY SIGN LEADING.
           05 PATIENT-STATUS                           PIC X(2).
           05 BLOOD-PINTS-FURNISHED     BINARY         PIC 9(5).
           05 BLOOD-PINTS-REPLACED      BINARY         PIC 9(4).
           05 SEQUENCE-COUNTER          BINARY         PIC 9(3).
           05 TRANSACTION-IND                          PIC 9.
           05 BILL-SOURCE                              PIC 9.
           05 BENEFITS-EXHAUST-IND                     PIC 9.
           05 BENEFITS-PAY-IND                         PIC 9.
           05 AUTO-ADJUSTMENT-IND                      PIC X.
           05 INTERMEDIARY-CTRL-NUM                    PIC X(23).
```

The JZOS COBOL RecordClassGenerator generates the source code for a Java class: ***com.myco.cobol.records.MedicareRecord.java***. The head of this file is:

```
package com.myco.cobol.records;
import com.ibm.jzos.fields.*;
import java.math.*;

// Generated by com.ibm.jzos.recordgen.cobol.RecordClassGenerator on ...

public class MedicareRecord {
```

Next, the generated source code defines ***static*** variables for each data item in the COBOL copy book. Each of these is an instance of a field converter object from the ***com.ibm.jzos.fields*** package and is created by using a ***CobolDatatypeFactory,*** which also assigns a running offset to each field. Note that since these fields objects are defined as Java ***static*** variables, they are created once during class initialization and are shared between potentially many instances.

```
protected static CobolDatatypeFactory factory = new CobolDatatypeFactory();

/** <pre>
 01  MEDICARE-RECORD. </pre> */
 protected static final int MEDICARE_RECORD_len = 120;

/** <pre>
 05 CLAIM-NUMBER             PIC X(19). </pre> */
 protected static StringField CLAIM_NUMBER = factory.getStringField(19, true);

/** <pre>
 05 ADMISSION-DATE           PACKED-DECIMAL PIC S9(7). </pre> */
 protected static PackedDecimalAsIntField ADMISSION_DATE =
                                    factory.getPackedDecimalAsIntField(7,
                                                                       true);
...
```

After the static fields are defined, the non-static instance variables are declared. These include:

- A "_byteBuffer" variable to hold the actual byte array for each record instance.
- Instance variables to cache access to each record field.

```
protected byte[] _byteBuffer;
// Instance variables used to cache field values
private String claimNumber;
private Integer admissionDate;
...
```

And then two constructors are generated:

```
public MedicareRecord (byte[] buffer) {
    this._byteBuffer = buffer;
}

public MedicareRecord () {
    this._byteBuffer = new byte[MEDICARE_RECORD_len];
}
```

Finally, “getter” and “setter” methods are generated for each field:

```
public String getClaimNumber() {
    if (claimNumber == null) {
        claimNumber = CLAIM_NUMBER.getString(_byteBuffer);
    }
    return claimNumber;
}

public void setClaimNumber(String claimNumber) {
    if (CLAIM_NUMBER.equals(this.claimNumber, claimNumber)) {
        return;
    }
    CLAIM_NUMBER.putString(claimNumber, _byteBuffer);
    this.claimNumber = claimNumber;
}
```

The following example code fragment demonstrates reading records from a MVS dataset using the JZOS ZFile class:

```
import com.ibm.jzos.ZFile;
...
ZFile zfile = new ZFile("//DD:INPUT","rb,type=record,noseek");
byte[] bytes = new byte[zfile.getLrecl()];
while (zfile.read(bytes) > 0) {
    MedicareRecord rec = new MedicareRecord(bytes);
    String s = rec.getClaimNumber();
    int d = rec.getAdmissionDate();
    ...
}
...
```

COBOL Datatypes Support

Elemental COBOL Data types are mapped to field converter classes in the ***com.ibm.jzos.fields*** package as follows:

- Alpha, alpha-numeric, and alpha-numeric edited items are mapped to StringField. The StringTrim argument to the RecordClassGenerator determines trimming behavior.
- Unscaled numeric items with DISPLAY usage are mapped to ExternalDecimalAsIntField or ExternalDecimalAsLongField, depending on the length.
- Scaled numeric items with DISPLAY usage are mapped to ExternalDecimalAsBigDecimalField.
- Numeric items with implicit scaling ('P' picture codes) are mapped to ExternalDecimalAsBigIntegerField with a negative scale amount.
- SIGN EXTERNAL , SIGN LEADING , and SIGN TRAILING clauses are supported as properties on ExternalDecimalAs<>Field.
- Numeric items with USAGE BINARY or USAGE COMP-5 are mapped to BinaryAsIntField or BinaryAsLongField depending on length.
- COMP-1 items are mapped to IbmFloatField.
- COMP-2 items are mapped to IbmDoubleField.
- External Float (USAGE DISPLAY) items are mapped to ExternalFloatField.

Current limitations:

- Numeric-edited items are currently mapped to an untrimmed StringField.
- DBCS (USAGE DISPLAY-1) items are currently only supported as ByteArrayFields.
- NATIONAL items are currently only supported as ByteArrayFields.

Support for COBOL Data Division

Name mapping

- ITEM-NAME is mapped to getItemName() and setItemName().
- If ITEM-NAME is not unique, then generated as getItemName_In_GroupName(). In_GroupName suffixing will continue until the name is unique.

```
/** <pre>
    05 GRP-A  </pre> */
 public static final int GRP_A_len = 4;

/** <pre>
       10 FIELD-A                          PIC X(4). </pre> */
 protected static StringField FIELD_A_In_GRP_A =
                                             factory.getStringField(4, true);

/** <pre>
    05 GRP-B  </pre> */
 public static final int GRP_B_len = 2;

/** <pre>
       10 FIELD-A                          PIC X(2). </pre> */
 protected static StringField FIELD_A_In_GRP_B =
                                             factory.getStringField(2, true);
...

 public String getFieldA_In_GrpA() { ... }

 public String getFieldA_In_GrpB() { ... }

...
```

- FILLER elements are mapped to StringFields named Filler_nn .

```
/** <pre>
     02  FILLER PIC X. </pre> */
public static StringField FILLER_1 = factory.getStringField(1, false);

/** <pre>
     02  FILLER PIC X(3). </pre> */
public static StringField FILLER_2 = factory.getStringField(3, true);
```

OCCURS clauses

An elemental data item with an OCCURS clause is generated with indexed accessors.

```
/** <pre>
     05 INT-05      OCCURS 10 TIMES         PIC 9(4). </pre> */
public static ExternalDecimalAsIntField INT_05 =
        factory.getExternalDecimalAsIntField(4, false, false, false, false);
public static final int INT_05_num = 10;
static { factory.incrementOffset((INT_05_num - 1) * INT_05.getByteLength()); }
...
protected Integer[] int05 = new Integer[INT_05_num];
...
public int getInt05(int _arrayIndex) {
    if (int05[_arrayIndex] == null) {
        int _arrayOffset = _arrayIndex * INT_05.getByteLength();
        int05[_arrayIndex] =
            new Integer(INT_05.getInt(_byteBuffer, _arrayOffset));
    }
    return int05[_arrayIndex].intValue();
}

public void setInt05(int _arrayIndex, int int05) {
    if (INT_05.equals(this.int05[_arrayIndex], int05)) {
        return;
    }
    int _arrayOffset = _arrayIndex * INT_05.getByteLength();
    INT_05.putInt(int05, _byteBuffer, _arrayOffset);
    this.int05[_arrayIndex] = new Integer(int05);
}
```

A group data item with an OCCURS clause will generate an inner class for the group and an indexed getter for elements of the group.

```
/** <pre>
   05 GROUP-05                 OCCURS 10 TIMES. </pre> */
public static final int GROUP_05_len = 94;
public static final int GROUP_05_offset = factory.getOffset();
public static final int GROUP_05_num = 10;
static { factory.incrementOffset(GROUP_05_num * Group05.GROUP_05_len); }
...
private Group05[] group05 = new Group05[GROUP_05_num];
...
public Group05 getGroup05(int _arrayIndex) {
    if (group05[_arrayIndex] == null) {
        int _arrayOffset = _arrayIndex * GROUP_05_len;
        group05[_arrayIndex] = new Group05(_byteBuffer, GROUP_05_offset +
                                           _arrayOffset);
    }
    return group05[_arrayIndex];
}
...
/** <pre>
   05 GROUP-05                 OCCURS 10 TIMES. </pre> */
public static class Group05 {
...
}
```

The OCCURS DEPENDING ON clause is supported, but the object (size) must be set before any fields following the field are accessed. Once set, the object size may not be changed on a given record instance. An OCCURS DEPENDING ON clause may not be nested within another OCCURS clause.

```
/** <pre>
     05 COUNT2                                PIC 9(9). </pre> */
public static ExternalDecimalAsIntField COUNT2 =
          factory.getExternalDecimalAsIntField(9, false, false, false, false);

/** <pre>
     05 GROUP-05 OCCURS 0 TO 99 TIMES
                   DEPENDING ON COUNT2. </pre> */
public static final int GROUP_05_len = 32;
public static final int GROUP_05_offset = factory.getOffset();
public static final int GROUP_05_num = 99;
static { factory.setOffset(0); } // storage following is variably located

/** <pre>
     05 VARLOC-CHARS2                         PIC X(5). </pre> */
public static StringField VARLOC_CHARS2 = factory.getStringField(5, true);
...
public int getCount2() {
    if (count2 == null) {
        throw new IllegalStateException("ODO object not initialized");
    }
    return count2.intValue();
}
...
protected int getGroup05_EndOffset() {
    int start = getVarString_EndOffset() + GROUP_05_offset;
    int curLen = getCount2() * GROUP_05_len;
    return start + curLen;
}

public String getVarlocChars2() {
    if (varlocChars2 == null) {
        int _variableOffset = getGroup05_EndOffset();
        varlocChars2 = VARLOC_CHARS2.getString(_byteBuffer, _variableOffset);
    }
    return varlocChars2;
}

...
```

When a generated class contains an OCCURS DEPENDING ON clause, a special constructor is generated which allows a record to be built on existing data or a new byte array. A zero-argument constructor is not generated, since the size of the underlying byte array is variable and cannot be pre-determined.

```
public OdomultiRecord (byte[] buffer, boolean bufferContainsData) {
    this._byteBuffer = buffer;
    if (bufferContainsData) {
        cacheOdoSizes();
    }
}

private void cacheOdoSizes() {
    count1 = new Integer(COUNT1.getInt(_byteBuffer));
    count2 = new Integer(COUNT2.getInt(_byteBuffer));
}
```

If an OCCURS DEPENDING ON clause appears on a single-character element, code is generated to simulate a variable-length String rather than an array of one byte Strings.

```
/** <pre>
     05 VAR-STRING                                    PIC X(1)
                       OCCURS 0 TO 256 TIMES
                       DEPENDING ON COUNT1. </pre> */
public static StringField VAR_STRING = factory.getStringField(1, false);
public static final int VAR_STRING_num = 256;
static { factory.setOffset(0); } // storage following is variably located
...
protected String varString;
...
public String getVarString() {
    if (varString == null) {
        StringField _field = new StringField(VAR_STRING.getOffset(),
                                             getCount1(), false);
        varString = _field.getString(_byteBuffer);
    }
    return varString;
}

public void setVarString(String varString) {
    if (VAR_STRING.equals(this.varString, varString)) {
        return;
    }
    StringField _field = new StringField(VAR_STRING.getOffset(), getCount1(),
                                         false);
    _field.putString(varString, _byteBuffer);
    this.varString = varString;
}
```

REDEFINES clause

REDEFINES clauses are supported, but involved fields will not have index-variable caches for the getter and setter methods.

```
/** <pre>
    05 COUNT1                                  PIC 9(3). </pre> */
public static ExternalDecimalAsIntField COUNT1 =
          factory.getExternalDecimalAsIntField(3, false, false, false, false);

/** <pre>
    05 RCOUNT1 REDEFINES COUNT1                PIC 9(3). </pre> */
static { factory.pushOffset(); }
static { factory.setOffset(COUNT1.getOffset()); }
public static ExternalDecimalAsIntField RCOUNT1 =
          factory.getExternalDecimalAsIntField(3, false, false, false, false);
static { factory.popOffset(); }
...
public int getCount1() {
    return COUNT1.getInt(_byteBuffer);
}

public void setCount1(int count1) {
    COUNT1.putInt(count1, _byteBuffer);
}

public int getRcount1() {
    return RCOUNT1.getInt(_byteBuffer);
}

public void setRcount1(int rcount1) {
    RCOUNT1.putInt(rcount1, _byteBuffer);
}
```

This feature - the disabling of redefined/redefining field caches - also applies to more complicated REDEFINE cases, such as GROUP redefines. This is necessary so that the fields that are changed will be reflected in their overlaid counterparts, but may have performance implications. Users may wish to eliminate copy book redefinitions where not needed, especially in applications where fields are repeatedly accessed in the same record instance (and caching would significantly improve performance).

VALUE clause

The VALUE clause is mapped to a Java static constant and a setInitialValues() method will be generated which sets all fields with initial values. Supported literal values include literal strings, fixed and floating point values and the figurative constants: BLANK(S), ZERO(S), LOW-VALUE(S), HIGH-VALUE(S). Multiple value array initialization is not supported.

```
/** <pre>
     05 CHAR-FIELD1              PIC X(10) VALUE SPACES. </pre> */
public static StringField CHAR_FIELD1 = factory.getStringField(10, true);
public static final String CHAR_FIELD1_ival = StringField.makeString(10, ' ');

/** <pre>
     05 CHAR-FIELD2              PIC X(5)  VALUE 'ABCDE'. </pre> */
public static StringField CHAR_FIELD2 = factory.getStringField(5, true);
public static final String CHAR_FIELD2_ival = "ABCDEF";

/** <pre>
     05 BIN-FIELD                BINARY PIC S9(4) VALUE -1. </pre> */
public static BinaryAsIntField BIN_FIELD = factory.getBinaryAsIntField(4,
                                                          true);
public static final int BIN_FIELD_ival = -1;

/** <pre>
     05 DOUBLE-FIELD             COMP-2          VALUE -88.7E-3. </pre> */
public static IbmDoubleField DOUBLE_FIELD = factory.getIbmDoubleField();
public static final double DOUBLE_FIELD_ival = -88.7E-3;

/** <pre>
     05 DECIMAL-FIELD                   PIC S9(7)V99 VALUE -2.3. </pre> */
public static ExternalDecimalAsBigDecimalField DECIMAL_FIELD =
  factory.getExternalDecimalAsBigDecimalField(9, 2, true, true, false, false);
public static final BigDecimal DECIMAL_FIELD_ival = new BigDecimal("-2.3");

...
public ConstntsRecord (byte[] buffer) {
    this._byteBuffer = buffer;
}

public ConstntsRecord () {
    this._byteBuffer = new byte[CONSTNTS_RECORD_len];
    setInitialValues();
}

public void setInitialValues() {
    setCharField1(CHAR_FIELD1_ival);
    setCharField2(CHAR_FIELD2_ival);
    setBinField(BIN_FIELD_ival);
    setDoubleField(DOUBLE_FIELD_ival);
    setDecimalField(DECIMAL_FIELD_ival);
}
```

Condition (Level 88) Elements

Conditions (88-Level items) cause Java static constants to be generated for the VALUE clause, and and isValueName() boolean method to be generated.

```
/** <pre>
     05 FLAG1                           PIC X VALUE 'Y'. </pre> */
public static StringField FLAG1 = factory.getStringField(1, false);
public static final String FLAG1_ival = "Y";

/** <pre>
        88 FLAG1-ON                             VALUE 'Y'. </pre> */
public static final String FLAG1_ON = "Y";

/** <pre>
        88 FLAG1-OFF                            VALUE 'N'. </pre> */
public static final String FLAG1_OFF = "N";

...
public boolean isFlag1On() {
    return getFlag1().equals(FLAG1_ON);
}

public boolean isFlag1Off() {
    return getFlag1().equals(FLAG1_OFF);
}
```

RENAMES clause

- Data items with the RENAMES clause are currently ignored.

RecordClassGenerator bufOffset feature

The bufOffset feature of the RecordClassGenerator will result in a generated class that can access a record at a non-zero offset in a byte array. This feature may be used to factor a large copy book into several nested Java records.

```
...
public MyRecord (byte[] buffer, int bufOffset) {
    this._byteBuffer = buffer;
    this._byteBufferOffset = bufOffset;
}

public MyRecord () {
    this._byteBuffer = new byte[MY_RECORD_len];
}
...
public String getField1() {
    if (field1 == null) {
        field1 = FIELD_1.getString(_byteBuffer, _byteBufferOffset);
    }
    return field1;
}

public void setField1(String field1) {
    if (FIELD_1.equals(this.field1, field1)) {
        return;
    }
    FIELD_1.putString(field1, _byteBuffer, _byteBufferOffset);
    this.field1 = field1;
}
```