

3.0

IBM Record Generator for Java V3.0.1



Note

Before using this information and the product it supports, read the information in [Chapter 10, “Notices,”](#) on page 37.

IBM Record Generator for Java V3.0.1

This edition applies to the IBM® Record Generator for Java™ V3.0, (product number 5655-CI2) and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright International Business Machines Corporation 2017-2020.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book.....	V
Chapter 1. What is IBM Record Generator for Java?.....	1
Chapter 2. What's new in V3.0.1.....	5
Chapter 3. Getting started with IBM Record Generator for Java.....	7
Chapter 4. Using IBM Record Generator for Java with COBOL.....	9
Example COBOL copybook.....	9
Creating ADATA files with the IBM Enterprise COBOL for z/OS compiler.....	9
Syntax of COBOL RecordClassGenerator.....	10
Running the COBOL RecordClassGenerator.....	12
Using the generated Java helper class in an application.....	13
Support for COBOL data division.....	13
Support for COBOL data types.....	13
Name mapping.....	14
JavaNameGenerator.....	14
OCCURS clause.....	16
REDEFINES clause.....	18
VALUE clause.....	18
Condition-name: level-88 statements.....	19
Chapter 5. Using IBM Record Generator for Java with assembler.....	21
Example DSECT	21
Creating ADATA files with IBM High Level Assembler.....	21
Syntax of assembler RecordClassGenerator	22
Running the assembler RecordClassGenerator.....	24
Using the generated Java helper class in an application.....	25
Support for assembler data types	25
Multi-operand statements	26
Chapter 6. Buffer offset feature.....	27
Chapter 7. XML support.....	29
Syntax for RecordXMLGenerator.....	31
Running the RecordXMLGenerator.....	32
Generating XML from an ADATA file	33
Generating Java source from an XML file.....	33
Chapter 8. Troubleshooting and support.....	35
Collecting troubleshooting data (MustGather) for IBM Support.....	35
Notices.....	43

About this book

This PDF is the product documentation for IBM Record Generator for Java V3.0. This information is also provided in IBM Knowledge Center, where you can also find the associated Javadoc.

Find the online version of this information in [IBM Knowledge Center](#).

Last updated

This PDF was created in September 2020.

Chapter 1. What is IBM Record Generator for Java?

IBM Record Generator for Java V3.0 is a stand-alone Java utility that generates Java helper classes to describe language-specific record structures. These helper classes can then be used in a Java application to marshal data to and from the byte-oriented record structures that are commonly used in z/OS applications, such as CICS COMMAREAs or VSAM files.

IBM Record Generator for Java V3.0 supersedes the alphaWorks version of the JZOS Record Generator V2.4.6, and provides new capabilities. For a summary of the new capabilities, see [Chapter 2, “What's new in V3.0.1,”](#) on page 5.

There are two implementations of IBM Record Generator for Java:

- For COBOL copybooks, implemented by the `RecordClassGenerator` class in the supplied `com.ibm.recordgen.cobol` Java package.
- For assembler-language DSECTs, implemented by the `RecordClassGenerator` class in the supplied `com.ibm.recordgen.asm` Java package.

[Figure 1 on page 2](#) shows the flow from COBOL and assembler input into ADATA files, which are used by IBM Record Generator for Java, along with the API package from the IBM JZOS Toolkit, to generate the Java helper classes.

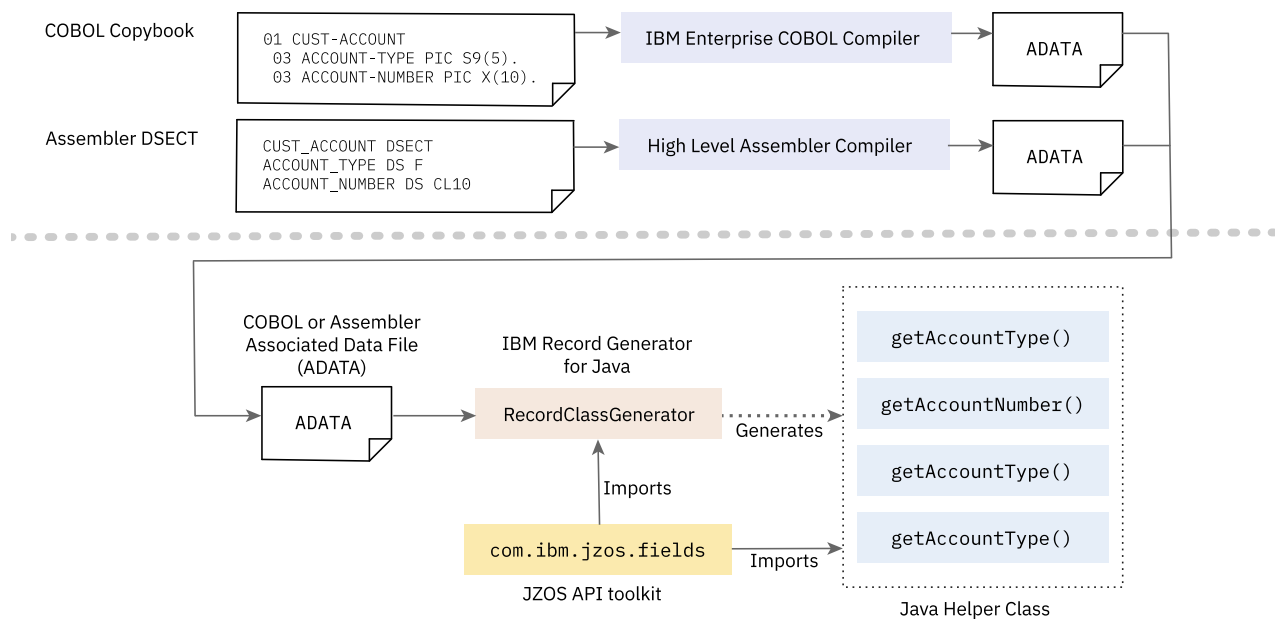


Figure 1. How IBM Record Generator works

The `RecordClassGenerator` class reads as input the ADATA output of the IBM Enterprise COBOL for z/OS and IBM High Level Assembler compilers and generates Java source code to map a selected language-specific record structure.

The source that is generated by the `RecordClassGenerator` uses the `com.ibm.jzos.fields` API package from the IBM JZOS Toolkit, which contains data type converters for the elemental COBOL and assembler data types. An additional class, `RecordXMLGenerator`, can generate an XML representation of the imported language structures that can be modified in third-party tools, such as XSLT, then used as input to the `RecordClassGenerator` to generate the Java helper classes.

After it is generated, the source code of the Java helper classes can be compiled with the `com.ibm.jzos.fields` package to produce a Java class that has accessors for each individual field in the record. Each field in the record can be written to using the appropriate setter method and also read through a corresponding getter method. The entire byte array that represents the record can also be referenced by using the supplied `getByteBuffer()` method. This greatly simplifies the development of Java applications that need to interact with structured enterprise data.

Chapter 2. What's new in V3.0.1

IBM Record Generator for Java V3.0 supersedes the alphaWorks version of the JZOS Record Generator V2.4.6, and provides new capabilities.

New in Version 3.0.1

New and changed functions	Description
Support for COBOL group conditions	Level 88 condition entries are now supported for group items as well as for elementary items, where permitted by COBOL syntax.
Support for the COBOL ALL figurative constant	The ALL figurative constant is now supported in the VALUE clause, wherever the VALUE clause is itself supported.

New in Version 3.0.0

New and changed functions	Description
New options for <code>RecordClassGenerator</code>	The <code>RecordClassGenerator</code> class offers the following new options: <code>genprotectedfields</code> can be set to false to cause static field variables to be generated with public access, instead of protected access. This is useful for dynamic modification of field behavior, or testing. <code>ignoreoccurs1</code> can be used to ignore OCCURS 1 on COBOL single element arrays so that fields are generated as an array of size 1. <code>preinitialize</code> can be set to true to generate code in the <code>setInitialValues()</code> method to initialize fields with a fixed location and length that are not arrays to blanks or zero. <code>stringencoding</code> can be set to an alternative single-byte EBCDIC code page that is used for String fields. <code>stringtrim</code> can be set to true to generate code that trims spaces from the end of String fields as they are accessed.
New class: <code>JavaNameGenerator</code>	The option for <code>JavaNameGenerator</code> is used to implement an alternative form of Java name generation for variables and accessor methods. This class is responsible for converting COBOL names into Java names of different types. You can subclass this class and override the behavior to affect the names that are used in Java accessors, static fields, instance variables, and parameter variables. For more information, see “JavaNameGenerator” on page 14
New function: reading ADATA without a record descriptor word (RDW)	The <code>RecordClassGenerator</code> class can now read ADATA without having to prefix each record with a record descriptor word (RDW). This makes it easier to transfer the ADATA files from z/OS for use on distributed platforms.
New package names for IBM Record Generator for Java V3.0	The new package names for IBM Record Generator for Java V3.0 are <code>com.ibm.recordgen.cobol</code> and <code>com.ibm.recordgen.asm</code> . These replace the package names that were used previously, which were <code>com.ibm.jzos.recordgen.cobol</code> and <code>com.ibm.jzos.recordgen.asm</code> .

Chapter 3. Getting started with IBM Record Generator for Java

To get started with IBM Record Generator for Java, check that you have the prerequisite environment, then download the product. After you extract the `ibm-recgen.jar` file from the download and transfer it to the system where you want to run IBM Record Generator for Java, you can start to work with it.

Supported environments

To use IBM Record Generator for Java, you need:

- Java 7-compatible Java Runtime Environment (JRE), either 32-bit (31-bit on z/OS®) or 64-bit
- To work with COBOL copybooks, IBM Enterprise COBOL for z/OS V4.2 or later
- To work with assembler DSECTs, IBM High Level Assembler for MVS™ and VM and VSE 1.6 and future fix packs
- IBM JZOS Toolkit API package `com.ibm.jzos.fields` at V2.4.8 or later, provided by IBM SDK for z/OS Java Technology Edition.

Installing IBM Record Generator for Java

1. Download IBM Record Generator for Java from the [IBM Record Generator for Java product page](#).
2. Extract the `ibm-recgen.jar` file from the downloaded zip file.
3. Transfer the `ibm-recgen.jar` file in binary format to the system where you want to run IBM Record Generator for Java.
4. If you are running on a non-z/OS® platform, the IBM JZOS Toolkit API must be available on the non-z/OS platform. Do this by transferring the `ibmjzos.jar` from the IBM SDK for z/OS Java Technology Edition.

Using IBM Record Generator for Java

1. Create the ADATA file from the COBOL copybook or assembler DSECT by running the z/OS Enterprise COBOL or High Level Assembler compiler with the ADATA option. For details, see, [“Creating ADATA files with the IBM Enterprise COBOL for z/OS compiler” on page 9](#), or [“Creating ADATA files with IBM High Level Assembler” on page 21](#).
2. Run either the COBOL or assembler version of the `RecordClassGenerator` class, supplying the ADATA file as input. This produces Java source code that represents the record structure. If you are running on a non-z/OS platform, make sure that the IBM JZOS Toolkit API is also available on the non-z/OS platform. For details, see [“Running the COBOL RecordClassGenerator” on page 12](#), or [“Running the assembler RecordClassGenerator” on page 24](#).
3. Transfer the generated Java class to your Java integrated development environment (IDE) and include it with your Java application that needs to access the structured record data.
4. Within your Java application, use the accessor methods on the Java class to get and set the field values for the record data.
5. Compile the Java application and deploy to the target system such as a CICS® JVM server, WebSphere® Application Server for z/OS, or a Java batch environment.

Optionally, you can use the `RecordXMLGenerator` class to create an intermediary description of the COBOL copybook or assembler DSECT. For details, see [“Running the RecordXMLGenerator” on page 32](#).

Chapter 4. Using IBM Record Generator for Java with COBOL

Use IBM Record Generator for Java to generate Java helper classes that can construct and parse byte arrays, which are used to interact with COBOL programs. Before you run IBM Record Generator for Java, you need an ADATA file that is produced by the IBM z/OS Enterprise COBOL compiler. This ADATA file must be based on the language structure in the COBOL source code for which you want a generated Java helper class. The ADATA file is used by IBM Record Generator for Java to generate the source code for the Java helper classes.

Example COBOL copybook

Here is an example of a COBOL data structure. This example is referenced in this documentation to explain how IBM Record Generator for Java processes COBOL data structures.

This data structure contains the same data as that used in the [“Example DSECT ”](#) on page 21.

```
01 MY-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 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).
```

Creating ADATA files with the IBM Enterprise COBOL for z/OS compiler

The ADATA file that is required as input to IBM Record Generator for Java is a binary file that contains specific record information about the program that is collected during compilation. The file can be a traditional MVS data set or a z/OS UNIX file.

To create the ADATA file that can be used as input to IBM Record Generator for Java, add the ADATA option to the list of options in the PARM parameter on the EXEC statement of the COBOL compiler job. For example:

```
//COBOL EXEC PGM=IGYCRCTL,REGION=200M,
//          PARM=(NODYNAM,RENT,LIST,MAP,XREF,CICS,ADATA)
```

The compiler produces the ADATA file that contains additional program data.

You control the location of this file by using the SYSADATA DD statement on the COBOL compiler job. For example:

```
//SYSADATA DD DSNAME=dsname
```

If you want to use the ADATA file on your local workstation as input for IBM Record Generator for Java, transfer it in binary mode to prevent data corruption.

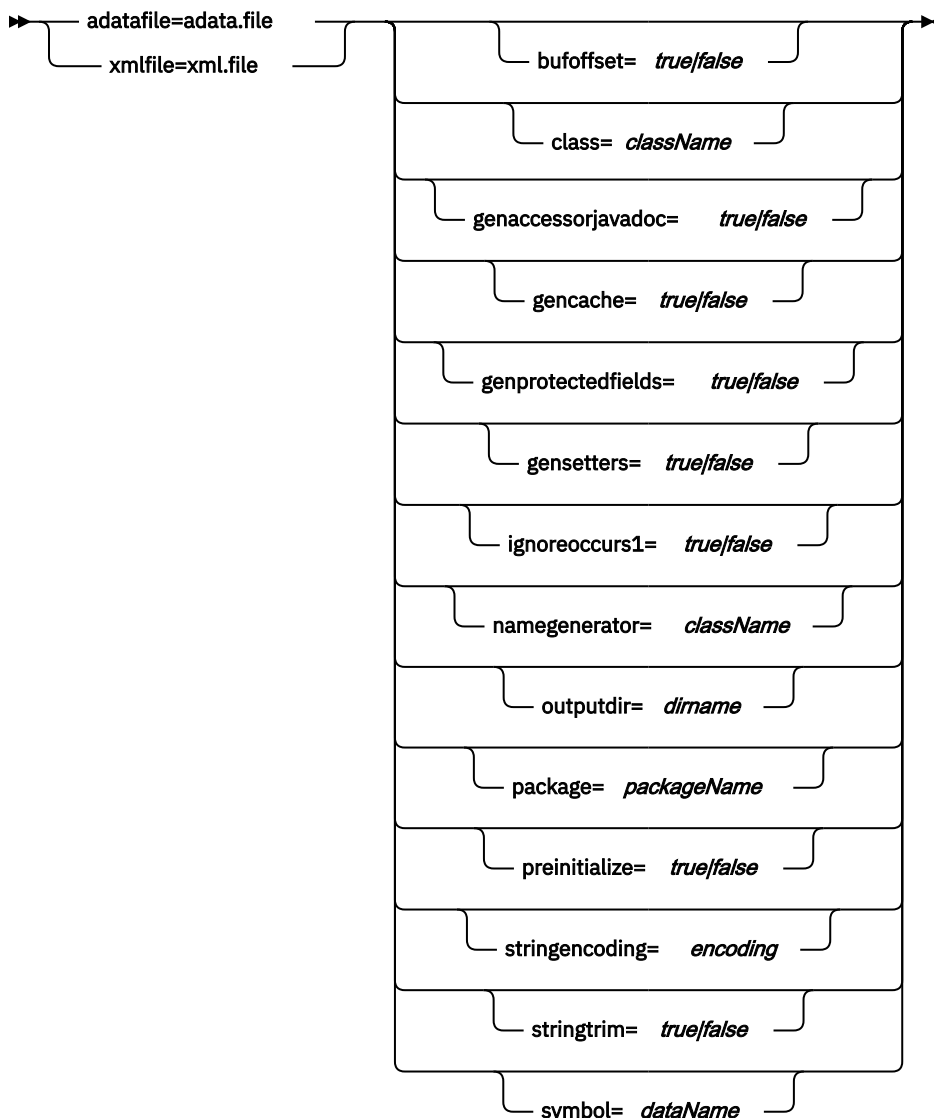
For details, see [COBOL SYSADATA file contents](#) in the IBM Enterprise COBOL for z/OS documentation.

Syntax of COBOL RecordClassGenerator

When you have created the ADATA file, it is used as input to IBM Record Generator for Java. The COBOL implementation of IBM Record Generator for Java is through the `RecordClassGenerator` class in the `com.ibm.recordgen.cobol` package. This generates the source code for a Java class that maps a byte array matching the description that is given by a COBOL copybook. You can set options to control the processing.

Syntax

The `RecordClassGenerator` takes as input the following options, using either `adatafile` or `xmlfile` as the only required option.



Options

adatafile (default://DD:SYSADATA; mutually exclusive with xmlfile)

Used to specify the input ADATA file that is generated by the IBM Enterprise COBOL for z/OS compiler from the COBOL copybook. You must specify either this option or `xmlfile`.

If the adatafile starts with `//`, it names a z/OS data set which might include MVS data set names (`//x.y.z`), member names (`//x.y.z(m)`), ddnames (`//DD:ddname`), and DD member names (`//DD:ddname(member)`).

Data set names that are not enclosed in single quotation marks are automatically prefixed with the current user ID.

If the adatafile is a PDS name without a member or a directory name, all members of the PDS (or files in the directory) are processed. For processing to work properly, you must specify the `outputdir` option but not the `class` and `symbol` options. This allows the copybook top-level record name to be used to generate unique classnames for each class. One invocation of the `RecordClassGenerator` can generate Java classes for an entire PDS library of ADATA files.

If the adatafile is a path name that does not start with `//`, it is assumed to be the name of a file in the z/OS UNIX (or workstation) file system that contains the ADATA records output from the compiler. The file might have each record prefixed by an IBM record descriptor name (RDW), but this is not required.

bufoffset (default:false)

Set to true to generate code that allows the Java record to be mapped to a non-zero offset in a byte array. For more information, see [Chapter 6, “Buffer offset feature,” on page 27](#).

class (default:top level group name)

Used to specify the Java class name for the generated output source.

genaccessorjavadoc (default:false)

Set to true to cause Javadoc comments to be generated for field getter and setter methods.

gencache (default:true)

Set to false to prevent generation of instance variables and code to cache the value of fields.

genprotectedfields (default:true)

Set to false to cause static field variables to be generated with public access, rather than protected access. This is useful for dynamic modification of field behavior or testing.

gensetters (default:true)

Set to false to prevent generation of Java setter methods.

ignoreoccurs1 (default:false)

Set to true to ignore OCCURS 1 clauses so that indexed accessors are not generated for these degenerate cases.

namegenerator (default:com.ibm.recordgen.cobol.JavaNameGenerator)

Set to a fully-qualified Java class that is a subclass of `com.ibm.recordgen.cobol.JavaNameGenerator`. This can be used to implement an alternative form of name generation for variables and accessor methods.

outputdir (default: writes file to System.out)

Specifies the output z/OS UNIX (or workstation) directory path where the generated Java class is created. The output file is created in the package-qualified subdirectory. For example, if `outputdir=/u/myid` and `package=com.ibm.cobrecs` and `class=MyRecord` (or implicitly determined from the copybook record MY-RECORD), then the output file that is created is: `/u/myid/com/ibm/cobrecs/MyRecord.java`.

package (default: mypackage)

Used to specify the Java package name that is used in the generated output source.

preinitialize (default: false)

Set to true to generate code in the `setInitialValues()` method to initialize fields with a fixed location and length that are not arrays to blanks or zero.

stringencoding (default: IBM-1047)

Set to an alternative single-byte EBCDIC code page that is used for String fields. The encoding that is used for individual String fields is (in order of preference, the first non-null value):

1. The encoding set in the individual field `StringField.setEncoding(String)`.
2. The encoding set by this option, or by the `setStringEncoding(String)` method of this class.
3. IBM-1047.

stringtrim (default: false)

Set to true to generate code that trims spaces from the end of String fields as they are accessed.

symbol (default: first level 01 found)

Used to specify the name of the first COBOL level 01 that is selected for generation. If not specified, the default is the first level 01 name found.

xmlfile (default: none; mutually exclusive with adatafile)

If the name of an xmlfile is given, it is either the `//dataset.name` or `/path/name` of the file that contains XML that was generated by, or compatible with, the output of the `RecordXMLGenerator` class. You must specify either this option or the `adatafile`.

Running the COBOL RecordClassGenerator

The COBOL `RecordClassGenerator` requires the `adatafile` option to specify the location of the ADATA file. All of the other options use default values. When you run IBM Record Generator for Java, make sure that your Java CLASSPATH points to the `ibm-recgen.jar`. If you run on a non-z/OS platform, you must also include in the CLASSPATH the `ibmjzos.jar` file from the IBM SDK for z/OS Java Technology Edition.

For descriptions of all the available options, see [“Syntax of COBOL RecordClassGenerator”](#) on page 10.

Examples

This example uses the language structure MY-RECORD within the USER.COBOL.ADATA(MYRECORD) PDS member and generates the source for a Java helper class file that is called `MyRecord.java` within the Java package `cobol.records`. This is written to the directory `generatedClasses/cobol/records`, based on the concatenation of the values on the `outdir` and `package` options.

```
java com.ibm.recordgen.cobol.RecordClassGenerator \
  adataFile="//'USER.COBOL.ADATA(MYRECORD)'" \
  package=cobol.records \
  class=MyRecord \
  symbol=MY-RECORD \
  outputdir=generatedClasses
```

This example uses, by default, the first 01 level data structure that is found within the `myRecord.adata` file, and generates the source for a Java helper class file that is called `MyRecord.java` within the Java package `cobol.records`. This is written to the directory `generatedClasses/cobol/records`.

```
java com.ibm.recordgen.cobol.RecordClassGenerator \
  adatafile=myRecord.adata \
  package=cobol.records \
  class=MyRecord \
  outputdir=generatedClasses
```

Using the generated Java helper class in an application

The Java class created by the `RecordClassGenerator` can be used as part of an existing Java application to construct or parse a byte array passed to or from a COBOL application.

Each generated Java helper class contains two constructors that can be used to create an instance of the generated class. Also, individual public accessor methods are generated for each symbol in the COBOL source code. These constructors and accessor methods enable you to either:

- Create a new instance of the generated class then use the accessor methods to set the fields in the class and obtain a byte array that represents the fields you have set, ready to be used by the COBOL program
- Create an instance of the generated class based on an existing byte array that was passed to the Java program from the COBOL program. The accessor methods can then be used to obtain the values from fields in the byte array.

For an example of using generated Java records to interact with a CICS COBOL application from a Java application, see [this article](#).

Constructing a new record

This example creates a new instance of the `MyRecord` class by using the zero argument constructor and uses the `setClaimNumber()` method to set a value for the `CLAIM_NUMBER` field. Other accessor methods might then be used to set the other fields in the record. When all the fields are set, then the `getByteBuffer()` method is used to obtain a byte array that is in the correct format to be used as input to the COBOL program.

```
MyRecord rec = new MyRecord();
String claimnum = "000000000123456789";
rec.setClaimNumber(claimnum);
byte[] record = rec.getByteBuffer();
```

Creating a record from an existing byte array

If the Java program already has access to a byte array that contains data from the COBOL program, then it can be used within the `MyRecord` class constructor to create an instance of the `MyRecord` class, which represents the data within the array. The Java program can use the `getClaimNumber()` method to obtain the value of the field within the byte array and manipulate it as necessary.

```
byte[] record;
MyRecord rec = new MyRecord(record);
String claimnum = rec.getClaimNumber();
```

Support for COBOL data division

The COBOL language uses the `PICTURE` clause to specify the characteristics of elementary data items. This section describes how the `RecordClassGenerator` processes data items to generate Java helper classes that correctly map to the individual fields. It describes the supported constructs by supplying the smallest COBOL sample that shows the construct, and shows the Java code that would be used to interact with the generated helper class.

Support for COBOL data types

Elemental COBOL data types are mapped to field converter classes in the `com.ibm.jzos.fields` package. The IBM JZOS Toolkit provides field converter classes for mapping byte array fields into Java data types.

- Alpha, alpha-numeric, and alpha-numeric edited items are mapped to `StringField`. The `stringtrim` option to the `RecordClassGenerator` class 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`.
- Only selected CONSTANT values are supported (others are ignored).
 - Literal strings
 - Figurative literals : SPACE(S), ZERO(S), LOW-VALUE(S), HIGH-VALUE(S), ALL
 - Numeric literals (integers and floating point).
- Constant values for array fields are ignored.
- RENAMEs are not supported and are ignored.
- Top-level groups with OCCURS are not supported.

Name mapping

The `RecordClassGenerator` class maps COBOL variable names from the COBOL copybook to equivalent field names in Java.

Each of the fields has an accessor method that can be used to access that field. Here is how that mapping occurs.

For example, a field called CLAIM-NUMBER in COBOL becomes the field CLAIM_NUMBER in Java and the accessors `getClaimNumber()` and `setClaimNumber()` are generated.

If a field is not uniquely named, it is generated as `getItemName_In_GroupName()`. In `_GroupName`, suffixing continues until the name is unique.

JavaNameGenerator

The `JavaNameGenerator` class is responsible for converting COBOL field names into formatted Java names.

This class can be subclassed by a user written class to provide an alternative naming convention to affect the names that are used within the generated helper classes.

You can allow a user class to override the naming conventions for Java class names by using the following methods:

- Java class names
- variable instance names
- static names

- accessor names
- level separators

Two utility methods are also provided which can be used to convert a COBOL symbol name into either a camel case or "_" separated string, for example, the COBOL symbol TEST-RECORD can be converted into either TestRecord or TEST_RECORD. All of the available methods are documented in the Javadoc.

For more information, see [JavaNameGenerator](#).

For more information, see [Javadoc for IBM Record Generator for Java](#) in the IBM Knowledge Center.

Examples

This first example class prefixes the class name with the String "Generated" to denote this as a generated class.

```
package com.mycompany.naming;
import com.ibm.recordgen.cobol.JavaNameGenerator;
public class ClassNamingVariation extends JavaNameGenerator {
    @Override
    public String getJavaClassName(String cobolSymbol) {
        return "Generated" + getCamelCasedName(cobolSymbol, true);
    }
}
```

Compiling and exporting this class to the file namingConventions.jar and using it when you are running the RecordClassGenerator by using the command

```
java -cp ibm-recgen.jar:ibmjzos.jar:namingConventions.jar \
com.ibm.recordgen.cobol.RecordClassGenerator adatafile=myrecord.adata \
nameGenerator=com.mycompany.naming.ClassNamingVariation
```

against an ADATA file generated from the BasicRecord COBOL example, will generate a class that has the following declaration:

```
public class GeneratedBasicRecord {
```

This naming convention can be overridden by the class parameter when running RecordClassGenerator.

This next sample class overrides the default accessor method name generation

```
package com.mycompany.naming;
import com.ibm.recordgen.cobol.JavaNameGenerator;
public class AccessorNamingVariation extends JavaNameGenerator {
    @Override
    public String getJavaAccessorName(String cobolSymbol) {
        return getUppercasedUnderscoreDelimitedName(cobolSymbol);
    }
}
```

By default the Java accessor method for a COBOL field CLAIM-NUMBER would appear in the generated class as:

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

However, by using the `AccessorNamingVariation` class when you are running the `RecordClassGenerator` they appear as:

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

OCCURS clause

The OCCURS clause states that a data division element repeats multiple times, and can be applied to a single element, a group of elements and define either a variable or static number of occurrences. An elemental data item within an OCCURS clause is generated with indexed accessors.

OCCURS with a single element

At the most basic, the following COBOL code defines that the field `FIRST-STRING` repeats five times.

```
01 BASIC-TEST-RECORD.
02 FIRST-STRING OCCURS 5 TIMES PIC X(10).
```

The generated Java helper class creates accessor methods that enable you to directly index the five instances as shown in this example (note that the index starts at zero, not 1):

```
public static void main(String [] args){
    Occurs1 occurs1 = new Occurs1();
    occurs1.setFirstString(0, "ONE");
    occurs1.setFirstString(1, "TWO");
    occurs1.setFirstString(2, "THREE");
    occurs1.setFirstString(3, "FOUR");
    occurs1.setFirstString(4, "FIVE");

    String thirdString = occurs1.getFirstString(2);
    // thirdString has the value of THREE
}
```

OCCURS in a group

OCCURS can also appear in a group, for example, the following code snippet shows the declaration of `GROUPA` that contains the fields `FIRST-STRING` and `FIRST-NUMBER`. The entire group occurs three times.

```
01 BASIC-TEST-RECORD.
02 GROUPA OCCURS 3 TIMES.
03 FIRST-STRING          PIC X(10) VALUE SPACES.
03 FIRST-NUMBER          PIC S9(8).
```

In this case, two helper classes are generated within the same Java source file:

- One class represents the `BASIC-TEST-RECORD`
- Another class represents the repeating `GROUPA` structure.

Although two classes are generated they are within the same Java source file. The outer class again maintains an array to hold the repeating elements.

The next snippet of code shows these objects that are being used:

```
public static void main(String [] args){
    //Create a new record and populate the fields
    Occurs2 groupOccurs = new Occurs2();

    Groupa firstGroup = groupOccurs.getGroupa(0);
}
```

```

firstGroup.setFirstNumber(20);
firstGroup.setFirstString("TWENTY");

Groupa secondGroup = groupOccurs.getGroupa(1);
secondGroup.setFirstNumber(30);
secondGroup.setFirstString("THIRTY");

//Byte buffer containing COBOL record data
byte[] buffer;
//This will have been initialised by our COBOL program.

//create a new record based upon the generated data
groupOccurs = new Occurs2(buffer);

int firstNumberInGroup3 = groupOccurs.getGroupa(3).getFirstNumber();
}

```

Variable OCCURS

The OCCURS construct can also define elements that variably repeat depending on another element. For example:

```

01 BASIC-TEST-RECORD.
02 NUMBER-OF-REPEATS          PIC 9(1).
02 GROUPA OCCURS 1 TO 9 DEPENDING ON NUMBER-OF-REPEATS.
03 FIRST-STRING              PIC X(10) VALUE SPACES.
03 FIRST-NUMBER              PIC S9(8).

```

This time, GROUPA repeats at least once but up to nine times. However, it isn't known how many times it repeats, so the generator is unable to create a zero argument constructor. Instead, you must calculate the size of the array that is required and generate a blank array. This blank array must be passed to the generated class within the constructor alongside a boolean value that describes if the byte array is blank or not. When this is done, the generated class is ready to update the fields.

```

public static void main(String [] args){
    //define the size of the repeating structure
    //and the number of times it will repeat
    int sizeOfGroupA = 18;
    int numberOfOccurs = 3;

    //generate a new blank buffer of the correct size
    //use the buffer to construct the class
    byte[] buffer = new byte[sizeOfGroupA * numberOfOccurs];
    Occurs3 occursDependingOn = new Occurs3(buffer, false);
    //false as the array is blank

    //now we can fill in the fields values
    occursDependingOn.setNumberOfRepeats(numberOfOccurs);
    occursDependingOn.getGroupa(0).setFirstNumber(10);
    occursDependingOn.getGroupa(0).setFirstString("TEN");
}

```

If a single character element repeats multiple times, then instead of creating an array, the RecordClassGenerator class can create a single accessor method to retrieve the elements as a single string.

```

01 BASIC-TEST-RECORD.
02 COUNTER                  PIC 9(3).
02 VAR-STRING OCCURS 1 TO 256 TIMES DEPENDING ON COUNTER PIC X.

```

Again, a default zero argument constructor cannot be defined because the size of the record can vary. However, instead of interacting with the VAR-STRING object as a 256 element array, it can be handled as if it were just a string element:

```

public static void main(String [] args){
    //define the size of the repeating structure
    int sizeOfRecord = 35;
}

```

```

byte[] buffer = new byte[numberOfRecord];

Occurs4 variableStringOccurs = new Occurs4(buffer,false);
variableStringOccurs.setCounter(32);
variableStringOccurs.setVarString("This is a variable length String");
}

```

REDEFINES clause

COBOL allows a section of data to be mapped by two separate structures. Usefully, it allows multiple mutually exclusive records to share the same logical space.

This record contains two structures, FIRST-SECTION and SECOND-SECTION. However, instead of these two structures taking up sequential areas of memory, the second structure simply remaps the first. In this instance, it splits the 10-byte string in the first section into two smaller areas.

```

01 BASIC-TEST-RECORD.
02 FIRST-SECTION.
03 FIRST-STRING          PIC X(10).
03 FIRST-NUMBER          PIC S9(8).
02 SECOND-SECTION REDEFINES FIRST-SECTION.
03 SECOND-STRING-A      PIC X(5).
03 SECOND-STRING-B      PIC X(5).
03 FIRST-NUMBER          PIC S9(8).

```

IBM Record Generator for Java supports REDEFINES structures. However it is important that the Java programmer understands the nature of the REDEFINES statement, as the generated code contains accessor methods to all fields and allows them to be used in any order, as the following Java code highlights:

```

public static void main(String [] args){
    Redefines redefinesRecord = new Redefines();

    redefinesRecord.setSecondStringA("Hello");
    redefinesRecord.setSecondStringB("World");

    System.out.println(redefinesRecord.getFirstString()); //prints "elloWorld"
    redefinesRecord.setFirstString("123456789");
    System.out.println(redefinesRecord.getSecondStringA()); //prints 12345
    System.out.println(redefinesRecord.getSecondStringB()); //prints 6789
}

```

VALUE clause

COBOL can assign variables default values as they are being declared.

In this example, the variable FIRST-STRING is set to be blank characters, FIRST-NUMBER is given the value 5 and SECOND-STRING the value 'HELLO'. It can be important that these values are preserved because the target program might expect that these values are in place.

```

01 VALUES-TEST.
02 FIRST-STRING          PIC X(10) VALUE SPACES.
02 FIRST-NUMBER          PIC S9(8) VALUE 5.
02 SECOND-STRING         PIC X(10) VALUE 'HELLO'.

```

IBM Record Generator for Java supplies the method `setInitialValues()` that sets the default values. This method is automatically called when the default zero arguments constructor is called.

```

public static void main(String [] args){
    ValuesTest values = new ValuesTest();

    int firstNumber = values.getFirstNumber(); //Will be set to the value 5
    String secondString = values.getSecondString(); //Will be set to HELLO
}

```


Condition-name: level-88 statements

COBOL can define a set of level-88 elements, which list potential values for the preceding element. The level-88 statements can be used to test the value of the preceding element.

For example, if the element FLAG1 has a value of 'Y', then FLAG1-ON resolves to true when used to test the value of FLAG1.

```
01 BASIC-TEST-RECORD.  
  02 FLAG1                PIC X.  
    88 FLAG1-ON           VALUE 'Y'.  
    88 FLAG1-OFF          VALUE 'N'.
```

When a structure of this type is processed by IBM Record Generator for Java, static fields are generated for each of the level-88 statements, which can be used to set the value of the preceding element. Also, methods are generated to test if the preceding element contains a specific value. For example:

```
public static void main(String[] args) {  
    BooleanExample boolExample = new BooleanExample();  
  
    //Set Flag 1 to have the value of 'N' or FLAG1-OFF  
    boolExample.setFlag1(boolExample.FLAG1_OFF);  
  
    //Test the value of flag 1  
    if(boolExample.isFlag1On()){  
        System.out.println("Flag is on");  
    }else{  
        System.out.println("Flag is off"); //This will print as the flag is off  
    }  
}
```

Level-88 elements that list multiple potential values or ranges, as illustrated in the following example, are *not* supported by the IBM Record Generator for Java.

```
02 FLAG2                PIC X(2).  
  88 FLAG-ALPHA          VALUES 'AA' 'AB' 'AC'.  
  88 FLAG-NUMERIC        VALUE 11 THRU 99.
```

If multiple potential values are listed as part of a COBOL level-88 element, the IBM Record Generator for Java generates a Java method that only checks for the first value listed, and not the entire range.

Chapter 5. Using IBM Record Generator for Java with assembler

Use IBM Record Generator for Java to generate Java helper classes that can construct and parse byte arrays, which are used to interact with assembler-language programs. Before you run IBM Record Generator for Java, you need an ADATA file that is produced by the IBM High-Level Assembler. This ADATA file must be based on the DSECT in the assembler source code for which you want a generated Java helper class. The ADATA file is used by IBM Record Generator for Java to generate the source code for the Java helper classes.

Example DSECT

Here is an example of an assembler DSECT. This example is referenced in this documentation to explain how IBM Record Generator for Java processes assembler data structures.

This data structure contains the same data as that used in the [“ Example COBOL copybook”](#) on page 9.

```
MY_RECORD      DSECT
CLAIM_NUMBER   DS    CL19
ADMISSION_DATE DS    PL4
FROM_DATE      DS    PL4
THRU_DATE      DS    PL4
DISCHARGE_DATE DS    PL4
FULL_DAYS      DS    PL3
COINSURANCE_DAYS DS   HL2
LIFETIME_RES_DAYS DS   FL4
INTERMEDIARY_NUM DS  FDL8
PROVIDER       DS    CL13
INPATIENT_DED  DS    PL4
BLOOD_DED      DS    PL4
TOTAL_CHARGES  DS    ZL9
PATIENT_STATUS DS    CL2
BLOOD_PINTS_FURNISHED DS FL4
BLOOD_PINTS_REPLACED DS HL2
SEQUENCE_COUNTER DS   HL2
TRANSACTION_IND DS    ZL1
BILL_SOURCE    DS    ZL1
BENEFITS_EXHAUST_IND DS  ZL1
BENEFITS_PAY_IND DS    ZL1
AUTO_ADJUSTMENT_IND DS  CL1
INTERMEDIARY_CTRL_NUM DS CL23
```

Creating ADATA files with IBM High Level Assembler

The ADATA file that is required as input to IBM Record Generator for Java is a file that contains specific record information about the program that is collected during assembly. The file can be a traditional MVS™ data set or a z/OS® UNIX file.

To create an ADATA file that can be used as input to IBM Record Generator for Java, add the ADATA option to the list of options in the PARM parameter on the EXEC statement of the assembler job. For example:

```
//ASSEMBLE EXEC ASMAC,PARM='ADATA,LIST,NOTERM,NODECK,NNOBJECT'
//C.SYSLIB DD
//      DD DSN=SYS1.MODGEN,DISP=SHR
```

The assembler produces the ADATA file that contains additional program data.

You control the location of this file by using the SYSADATA DD statement on the assembler job. For example:

```
//SYSADATA DD DSNAME=DSNAME,DISP=(OLD,DELETE)
```

If you want to use the ADATA file on your local workstation as input for the IIBM Record Generator for Java, transfer it in binary mode to prevent data corruption.

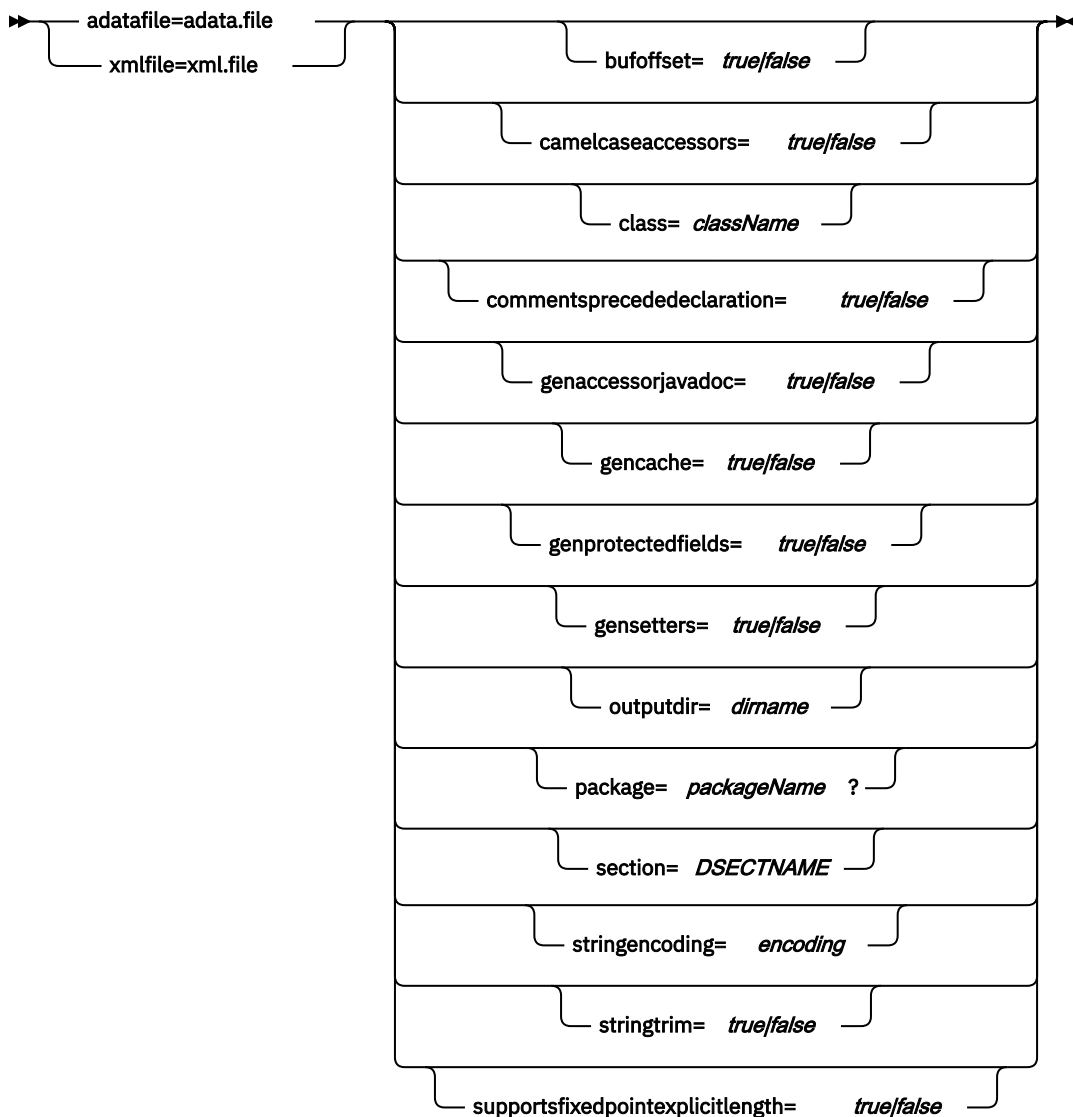
For details, see [Associated data file output](#) in the HLASM documentation.

Syntax of assembler RecordClassGenerator

When you have created the ADATA file, it is used as input to IBM Record Generator for Java. The assembler implementation of IBM Record Generator for Java is through the `RecordClassGenerator` class in the `com.ibm.recordgen.asm` package. This generates the source code for a Java class that maps a byte array matching the description that is given by an assembler DSECT. You can set options to control the processing.

Syntax

The `RecordClassGenerator` takes as input the following options, using either `adatafile` or `xmlfile` as the only required option.



adatafile (default: `//DD:SYSADATA`; mutually exclusive with `xmlfile`)

Used to specify the input ADATA file that is generated by the IBM High Level Assembler from the assembler DSECT. You must specify either this option or `xmlfile`.

If the adatafile starts with `//`, it names a z/OS data set which might include MVS data set names (`//x.y.z`), member names (`//x.y.z(m)`), ddnames (`//DD:ddname`), and DD member names (`//DD:ddname(member)`).

Data set names that are not enclosed in single quotation marks are automatically prefixed with the current user ID.

If the adatafile is a PDS name without a member or a directory name, all members of the PDS (or files in the directory) are processed. For processing to work properly, you must specify the `outputdir` option but not the `class` and `section` options. This allows the first DSECT name in each ADATA file to be used to generate unique classnames for each class. One invocation of the `RecordClassGenerator` can generate Java classes for an entire PDS library of ADATA files.

If the adatafile is a path name that does not start with `//`, it is assumed to be the name of a file in the z/OS UNIX (or workstation) file system that contains the ADATA records output from the compiler. The file might have each record prefixed by an IBM record descriptor name (RDW), but this is not required.

bufoffset (default: false)

Set to true to generate code that allows the Java record to be mapped to a non-zero offset in a byte array. For more information, see [Chapter 6, “Buffer offset feature,”](#) on page 27.

camelcaseaccessors (default: true)

Set to false to prevent getter and setter methods from being camel cased. The default is true, so each “_” - separated segment of the assembler symbol name is converted to a Java name that starts with an uppercase letter, followed by lowercase letters.

class (default: top-level group name)

Used to specify the Java class name for the generated output source. . The default is to use the name of the ADATA section (mapped into a valid Java class name).

commentsprecededdeclaration (default: false)

Configures how the Javadoc for a given declaration is generated when using multi-line comments. The default is false. For example, by convention, many assembler language data declarations have comment lines that follow the declaration. Other conventions have the comments preceding the declaration.

Consider the following example with comments preceding the declaration.

```
MY_RECORD      DSECT      COPYBOOK FOR JZOS RECORD SAMPLE
*              DS CL19     CHARACTER STRING LENGTH 19 BYTES
CLAIM_NUMBER    CLAIM NUMBER IS PRIMARY KEY
```

With `commentsprecededdeclaration=true`, the field generated for `CLAIM_NUMBER` will appear as follows:

```
/** <pre>
*              DS CL19     CHARACTER STRING LENGTH 19 BYTES
CLAIM_NUMBER    CLAIM NUMBER IS PRIMARY KEY </pre>
*/
protected static final StringField CLAIM_NUMBER = factory.getStringField(19);
```

genaccessorjavadoc (default: false)

Set to true to cause Javadoc comments to be generated for field getter and setter methods.

gencache (default: true)

Set to false to prevent generation of instance variables and code to cache the value of fields.

genprotectedfields (default: true)

Set to false to cause static field variables to be generated with public access, rather than protected access. This is useful for dynamic modification of field behavior or testing.

gensetters (default: true)

Set to false to prevent generation of Java setter methods.

outputdir (default: writes file to System.out)

The output file is created in the package-qualified subdirectory with a file name `Classname.java`.

package (default: mypackage)

Used to specify the Java package name that is used in the generated output source.

section (default: process the first CSECT or DSECT found)

Selects which CSECT or DSECT in the ADATA file is processed. The default is to select the first section in the ADATA file.

stringencoding (default: IBM-1047)

Set to an alternative single-byte EBCDIC code page that is used for String fields. The encoding that is used for individual String fields is (in order of preference, the first non-null value):

1. The encoding set in the individual field `StringField.setEncoding(String)`.
2. The encoding set by this option, or by the `setStringEncoding(String)` method of this class.
3. IBM-1047.

stringtrim (default: false)

Set to true to generate code that trims spaces from the end of String fields as they are accessed.

supportsfixedpointexplicitlength (default: true)

Set to false to revert to compatibility with older versions of assembler record generator. Previously, a DC/DS type FLx would generate a byte array field. Now, the default is to generate a Binary field (int or long) for lengths less than or equal to eight.

xmlfile (default: none; mutually exclusive with adatafile)

If the name of an xmlfile is given, it is either the `//dataset.name` or `/path/name` of the file that contains XML that was generated by, or compatible with, the output of the `RecordXMLGenerator` class. You must specify either this option or the `adatafile`.

Running the assembler RecordClassGenerator

The assembler `RecordClassGenerator` requires the `adatafile` option to specify the location of the ADATA file. All of the other options use default values. When you run IBM Record Generator for Java, make sure that your Java CLASSPATH points to the `ibm-recgen.jar`. If you run on a non-z/OS platform, you must also include in the CLASSPATH the `ibmjzos.jar` file from the IBM SDK for z/OS Java Technology Edition.

For descriptions of all the available options, see [“Syntax of COBOL RecordClassGenerator”](#) on page 10.

Examples

This example uses the DSECT MY_RECORD within the USER.ASSEM.ADATA(MYRECORD) PDS member and generates the source for a Java helper class file that is called `MyRecord.java` within the Java package `assem.records`. This is written to the directory `generatedClasses/assem/records`, based on the concatenation of the values on the `outdir` and `package` options.

```
java com.ibm.recordgen.asm.RecordClassGenerator \
  adataFile="//'USER.ASSEM.ADATA(MYRECORD)'" \
  package=assem.records \
  class=MyRecord \
  section=MY_RECORD \
  outputdir=generatedClasses
```

This example uses, by default, the first DSECT that is found in the `myRecord.adata` file and generates the source for a Java helper class file called `MyRecord.java` within the Java package `assem.records`. This will be written to the directory `generatedClasses/assem/records`.

```
java com.ibm.recordgen.asm.RecordClassGenerator \
  adatafile=myRecord.adata \
  package=assem.records \
  class=MyRecord \
  outputdir=generatedClasses
```

Using the generated Java helper class in an application

The Java class created by the `RecordClassGenerator` class can be used as part of an existing Java application to construct or parse a byte array passed to or from an assembler application.

Each generated Java helper class contains two constructors that can be used to create an instance of the generated class. Also, individual public accessor methods are generated for each symbol in the assembler source code. These constructors and accessor methods enable you to either:

- Create a new instance of the generated class then use the accessor methods to set the fields in the class and obtain a byte array that represents the fields you have set, ready to be used by the assembler program.
- Create an instance of the generated class based on an existing byte array that was passed to the Java program from the assembler program. The accessor methods can then be used to obtain the values from fields in the byte array.

Constructing a new record

This example creates a new instance of the `MyRecord` class by using the zero argument constructor and uses the `setClaimNumber()` method to set a value for the `CLAIM_NUMBER` field. Other accessor methods might then be used to set the other fields in the record. When all the fields are set, then the `getByteBuffer()` method is used to obtain a byte array that is in the correct format to be used as input to the assembler program.

```
MyRecord rec = new MyRecord();
String claimnum = "000000000123456789";
rec.setClaimNumber(claimnum);
byte[] record = rec.getByteBuffer();
```

Creating a record from an existing byte array

If the Java program already has access to a byte array that contains data from the assembler program, then it can be used within the `MyRecord` class constructor to create an instance of the `MyRecord` class, which represents the data within the array. The Java program can use the `getClaimNumber()` method to obtain the value of the field within the byte array and manipulate it as necessary.

```
byte[] record;
MyRecord rec = new MyRecord(record);
String claimnum = rec.getClaimNumber();
```

Support for assembler data types

Elemental assembler data types are mapped to field converter classes in the `com.ibm.jzos.fields` package. The IBM JZOS Toolkit provides field converter classes for mapping byte array fields into Java data types.

- DS C items are mapped to `StringField`. The `stringtrim` argument to the `RecordClassGenerator` class determines trimming behavior.
- DS A, DS F, DS H, DS R, DS S, DS V, DS Y, DS X items (length less than or equal to four bytes) are mapped to `BinaryAsIntField`.
- DS B, DS X items (length greater than or equal to four bytes) are mapped to `ByteArrayField`.
- DS E, DS D, DS L items are mapped to `IbmFloatField` or `IbmDoubleField`, based on length.
- Binary floating point numbers (DS EB, DS DB, DS LB) are not supported and are mapped to `ByteArrayField`.
- DS P items are mapped to `PackedDecimalAsIntField` or `PackedDecimalAsLongField`, depending on length.

- DS Z items are mapped to `ExternalDecimalAsIntField` or `ExternalDecimalAsLongField`, depending on length.

Current limitations

Assembler labels can contain symbols that are not valid in Java names. The following conversions are performed during code generation:

- @ is mapped to the string `_at_`
- # is mapped to the string `_hash_`

Multi-operand statements

The assembler `RecordClassGenerator` allows for multiple operands to be associated with a single symbol.

In this case, the generated code gives the first operand the symbol name. Each additional operand is given the symbol name with a numeric suffix so that they can be distinguished. For example, the DSECT statement:

```
CLAIM_NUMBER          DS H,CL19    CLAIM NUMBER IS PRIMARY KEY
```

Results in the following generated code:

```
/** <pre>
CLAIM_NUMBER          DS H,CL19    CLAIM NUMBER IS PRIMARY KEY </pre>
*/
protected static final BinaryAsIntField CLAIM_NUMBER = \
factory.getBinaryAsIntField(2, true); \
protected static final StringField CLAIM_NUMBER_1 = \
factory.getStringField(19);
```


Chapter 6. Buffer offset feature

The buffer offset feature of the `RecordClassGenerator` class (specified by the `bufoffset` option) results in a generated class that provides an additional constructor to access a record at a non-zero offset in a byte array. The `bufoffset` option can be used by both the COBOL `RecordClassGenerator` and the assembler `RecordClassGenerator`.

With large COBOL record structures, it can be easier to split the record into smaller subrecords and generate Java helper classes for each subrecord. In this case, the buffer offset feature is useful because it allows each helper class to parse their own section from a relative offset in the byte array. Take the following example:

```
01 BASIC-TEST-RECORD.
  02 SECTION-A.
  03 FIRST-STRING          PIC X(10) VALUE SPACES.
  02 SECTION-B.
  03 SECOND-STRING         PIC X(10) VALUE SPACES.
  02 SECTION-C.
  03 THIRD-STRING          PIC X(10) VALUE SPACES.
```

The BASIC-TEST-RECORD structure consists of three sections each containing a 10-byte string. This is a simple example; in reality, each section might be much longer and contain varying variables. To generate each section as an individual record, you need to break the copybook into three smaller sections and individually generate Java helper classes for each 01 level section. Use the `symbol` option of `RecordClassGenerator` to dictate the section for which you want to generate a helper class. You also need to set the `bufoffset` option to true. This causes the additional constructor with the buffer offset to be added to the generated class.

```
01 SECTION-A.
  03 FIRST-STRING          PIC X(10) VALUE SPACES.
01 SECTION-B.
  03 SECOND-STRING         PIC X(10) VALUE SPACES.
01 SECTION-C.
  03 THIRD-STRING          PIC X(10) VALUE SPACES.
```

The example below shows the class for just one of the sections, `SectionC`. It would be repeated for each of the other sections.

```
java com.ibm.recordgen.cobol.RecordClassGenerator \
  adatafile=buffer.adt \
  bufoffset=true \
  symbol=SECTION-C \
  outputdir=output \
  package=coboldata \
  class=SectionC
```

These three classes can then be used to build each segment of the structure or to parse a subsection of the structure.

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

    SectionA sectionA = new SectionA();
    SectionB sectionB = new SectionB();
    SectionC sectionC = new SectionC();

    sectionA.setFirstString("Hello");
    sectionB.setSecondString("World");
    sectionC.setThirdString("Again!");

    //Create a record that contains all three sections
    //use the ByteBuffer class to concatenate each section
    ByteBuffer bb = ByteBuffer.allocate(30);
    bb.put(sectionA.getBytes(), 0, 10);
```

```
bb.put(sectionB.getByteBuffer(), 0, 10);
bb.put(sectionC.getByteBuffer(), 0, 10);
byte[] record = bb.array();

//reconstruct the three sections from the record
//use the buffoffset option in the constructor
//to define the offset
sectionA = new SectionA(record, 0);
sectionB = new SectionB(record, 10);
sectionC = new SectionC(record, 20);

//finally print out the data to prove that it has
//all worked
System.out.println(sectionA.getFirstString());
System.out.println(sectionB.getSecondString());
System.out.println(sectionC.getThirdString());
}
```

Chapter 7. XML support

IBM Record Generator for Java can also generate XML files that describe the fields within an assembler DSECT or COBOL language structure. This XML file can then be processed by XML stylesheets or similar technology, then used as input into the `RecordClassGenerator` class to generate the Java helper classes.

Before you run IBM Record Generator for Java, you need an ADATA file that is produced by either the IBM Enterprise COBOL for z/OS compiler or the IBM High Level Assembler. This ADATA file must be based on the language structure in the COBOL or assembler source code for which you want to generate an XML file.

A Java class that is called `RecordXMLGenerator` is provided for XML support. There are two implementations of `RecordXMLGenerator`:

- Implements the XML support for COBOL and is in the `com.ibm.recordgen.cobol` package.
- Implements the XML support for assembler and is in the `com.ibm.recordgen.asm` package.

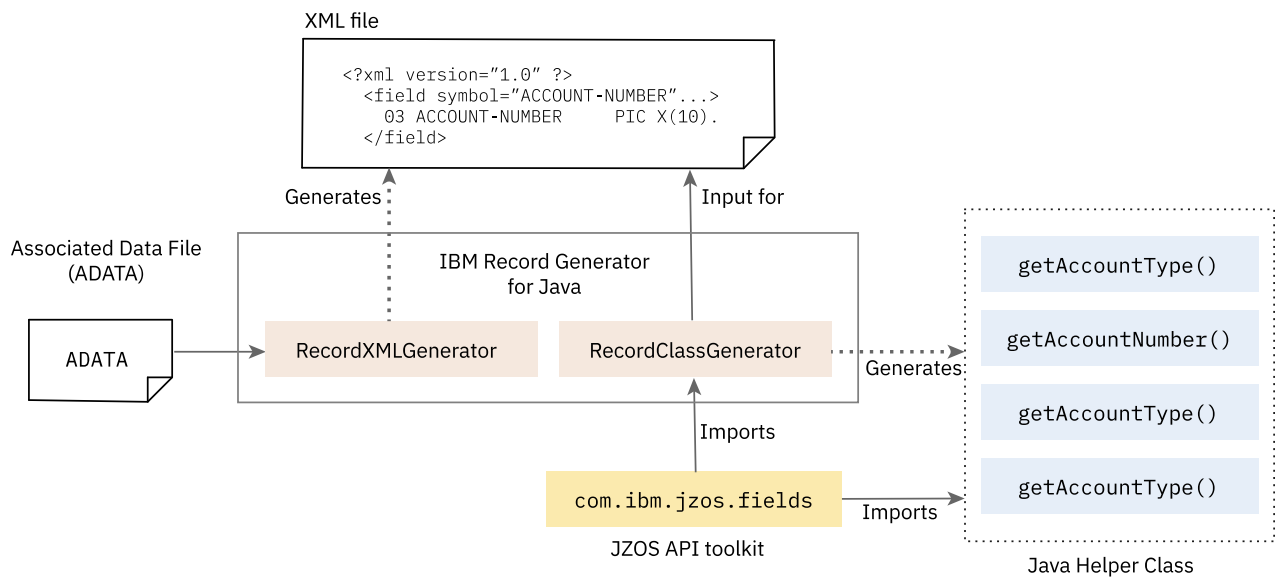


Figure 2. How the XML generation works

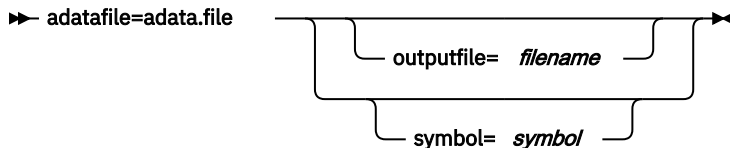
This flow chart is an example of how using the ADATA file, RecordXMLGenerator can generate XML files and then uses the RecordClassGenerator to generate Java helper classes.

Syntax for RecordXMLGenerator

The RecordXMLGenerator class is invoked in a similar way to the RecordClassGenerator class, but it has fewer options to control processing.

For COBOL

The COBOL implementation of the RecordXMLGenerator class takes as input the following options, where `adatafile` is the only required option.



adatafile

Used to specify the input ADATA file that is generated by the IBM Enterprise COBOL for z/OS compiler from the COBOL copybook. You must specify this option.

If the `adatafile` starts with `//`, it names a z/OS data set by using the same syntax as the `com.ibm.jzos.ZFile()` constructor, which includes MVS data set names (`//x.y.z`), member names (`//x.y.z(m)`), ddnames (`//DD:ddname`), and DD member names (`//DD:ddname(member)`). Data set names that are not enclosed in single quotation marks are automatically prefixed with the current user ID.

If the `adatafile` is a path name that does not start with `//`, it is assumed to be the name of a file in the z/OS UNIX (or workstation) file system that contains the ADATA records output from the compiler. The file might have each record prefixed by an IBM record descriptor name (RDW), but this is not required.

outputfile (default: writes file to System.out)

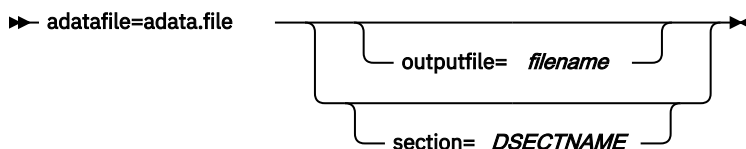
If omitted, the XML is written to `System.out`. Otherwise, it should represent the file to which the RecordXMLGenerator class should write the output.

symbol (default: first level 01 found)

Used to specify the name of the first COBOL level 01 that is selected for generation. If not specified, the default is the first level 01 name found.

For assembler

The assembler implementation of the RecordXMLGenerator class takes as input the following options, where `adatafile` is the only required option.



adatafile

Used to specify the input ADATA file that is generated by the IBM High Level Assembler from the assembler DSECT. You must specify this parameter.

If the `adatafile` starts with `//`, it names a z/OS data set which might include MVS data set names (`//x.y.z`), member names (`//x.y.z(m)`), ddnames (`//DD:ddname`), and DD member names (`//DD:ddname(member)`).

If the adatafile is a path name that does not start with //, it is assumed to be the name of a file in the z/OS UNIX (or workstation) file system that contains the ADATA records output from the compiler. The file might have each record prefixed by an IBM record descriptor name (RDW), but this is not required.

outputfile (default: writes file to System.out)

If omitted, the XML is written to System.out. Otherwise, it should represent the file to which the RecordXMLGenerator class should write the output.

section (default: process the first CSECT or DSECT found)

Selects which CSECT or DSECT in the ADATA file is processed. The default is to select the first section in the ADATA file.

Running the RecordXMLGenerator

Before the RecordXMLGenerator class can be used you must have generated an ADATA file from the COBOL or assembler language structure that you wish to use. The RecordClassGenerator requires the adatafile option to specify the location of the ADATA file. All of the other options use default values. When you run IBM Record Generator for Java, make sure that your Java CLASSPATH points to the ibm-recgen.jar. If you run on a non-z/OS platform, you must also include in the CLASSPATH the ibmjzos.jar file from the IBM SDK for z/OS Java Technology Edition.

For descriptions of all the available options, see [“Syntax for RecordXMLGenerator” on page 31](#).

COBOL examples

This example uses the language structure MY-RECORD within the USER.COBOL.ADATA(MYRECORD) PDS member and generates an XML file called MyRecord.xml.

```
java com.ibm.recordgen.cobol.RecordXMLGenerator \  
  adatafile="//'USER.COBOL.ADATA(MYRECORD)'" \  
  symbol=MY-RECORD \  
  outputfile=MyRecord.xml
```

This example uses the first 01 level data structure found within the myRecord.adata file and generates the source for an XML file called MyRecord.xml.

```
java com.ibm.recordgen.cobol.RecordXMLGenerator \  
  adatafile=myRecord.adata \  
  outputfile=MyRecord.xml
```

Assembler examples

This example uses the DSECT MY_RECORD within the USER.ASSEM.ADATA(MYRECORD) PDS member and generates an XML file called MyRecord.xml.

```
java com.ibm.recordgen.asm.RecordXMLGenerator \  
  adatafile="//'USER.ASSEM.ADATA(MYRECORD)'" \  
  section=MY_RECORD \  
  outputfile=MyRecord.xml
```

This example uses the first DSECT found within the myRecord.adata file and generates an XML file called MyRecord.xml.

```
java com.ibm.recordgen.asm.RecordXMLGenerator \  
  adatafile=myRecord.adata \  
  outputfile=MyRecord.xml
```

Generating XML from an ADATA file

The COBOL or assembler `RecordXMLGenerator` class can be used to generate an XML file from ADATA files. The XML file represents the ADATA. Subsequently, this XML file can be used as input to the `RecordClassGenerator` class, by specifying the `xmlfile` option.

You can generate the XML file by entering a command, similar to the example below. This example, using the COBOL `RecordXMLGenerator`, looks for the 01 level symbol `BASIC-TEST-RECORD` in the `basic.adt` file, generates an XML file that represents the content of the ADATA file and stores it in `basicTestRecord.xml`.

```
java ibm-recgen.jar:ibmjzos.jar \  
  com.ibm.recordgen.cobol.RecordXMLGenerator \  
  adatafile=basic.adt \  
  symbol=BASIC-TEST-RECORD \  
  outputfile=basicTestRecord.xml
```

After the command completes, the `basicTestRecord.xml` file contains the following content:

```
<?xml version="1.0" ?>  
<group symbol="BASIC-TEST-RECORD" level="1" symbolId="3149" size="18" sourceStmt="11" offset="0">  
  01 BASIC-TEST-RECORD.  
    <field symbol="FIRST-STRING" level="2" symbolId="3233" size="10" sourceStmt="12" offset="0"  
value="SPACES" fieldType="String" length="10" trim="true" padLeft="false">  
      02 FIRST-STRING          PIC X(10) VALUE SPACES.  
    </field>  
    <field symbol="FIRST-NUMBER" level="2" symbolId="3317" size="8" sourceStmt="13" offset="10"  
fieldType="ExternalDecimal" precision="8" scale="0" signed="true" signTrailing="true"  
signExternal="false" blankWhenZero="false">  
      02 FIRST-NUMBER          PIC S9(8).  
    </field>
```

Generating Java source from an XML file

XML that is generated from the `RecordXMLGenerator` Class can be used to generate a Java helper class by using the `xmlfile` option of the `RecordClassGenerator` class.

The following example, for COBOL, takes the `basicTestRecord.xml` file that was created in previous examples and generates the source file `BasicTestRecord.java` in the output directory `output/xml/test`. The `package` option is used to create sub-directories in the output folder.

```
java com.ibm.recordgen.cobol.RecordClassGenerator \  
  xmlfile=basicTestRecord.xml \  
  outputfile=output \  
  package=xml.test
```

Chapter 8. Troubleshooting and support

To isolate and resolve IBM Record Generator for Java V3.0 problems, use the troubleshooting information here.

You can also find information in the [Q&A forum](#) for IBM Record Generator for Java V3.0. If you require additional support, contact IBM through the [IBM Support Portal](#).

java.lang.IllegalStateException specTree root is not a StorageSpec

Check that the file specified on the RecordClassGenerator class with the `adatafile` option is a valid ADATA file and not COBOL or assembler source code. If you are running on a distributed platform, make sure that the ADATA file was downloaded in binary format.

java.lang.ClassDefNotFoundException

Check that the `ibmjzos.jar` (from the IBM SDK for z/OS Java Technology Edition) and `ibm-recgen.jar` (from IBM Record Generator for Java V3.0) are specified on the Java CLASSPATH.

java.lang.UnsupportedClassVersionError JVMCFRE003 bad major version; class=com/ibm/jzos/fields/Field, offset=6

The version of the `ibmjzos.jar` that is on your Java CLASSPATH is from a later version of Java than the one you are using. For example, the `ibmjzos.jar` was taken from Version 8 of IBM SDK for z/OS Java Technology Edition but on a distributed platform, you are using Java 7.

Error while processing an XML file

If you encounter the message `javax.xml.stream.XMLStreamException: ParseError at [row,col]:[2,98], with the explanation "Unexpected element "group" encountered parsing XML. Input file is invalid and cannot be processed."`, you are using an XML file as input to the assembler version of RecordClassGenerator that was generated by using the COBOL version of the RecordXMLGenerator.

java.lang.IllegalArgumentException

If you encounter a `java.lang.IllegalArgumentException` running the RecordClassGenerator, check that you are using the correct language version of the RecordClassGenerator. For example, you might be using an ADATA file from a COBOL copybook as input to the assembler version of the RecordClassGenerator.

Collecting troubleshooting data (MustGather) for IBM Support

If you have to contact IBM Support about a problem in IBM Record Generator for Java, ensure that you collect troubleshooting data, also known as MustGather.

Component-specific information

Gather the following information that is specific to IBM Record Generator for Java:

- The ADATA file that is used as input to the IBM Record Generator for Java, downloaded in binary format.
- Preferably, the original source code file that is used to produce that ADATA.
- Any console output from the IBM Record Generator for Java. This is any output to `System.out` or `System.err` (`sysout` or `syserr`), or any files that are generated especially if the generated code has a problem.
- Any options used on the command line that is used to invoke the IBM Record Generator for Java.
- The content of the CLASSPATH environment variable.

General information

Gather the following general information about the problem and your environment:

- A complete description of the problem that includes the following questions:

- When did the problem first occur?
- Is the problem a one-time failure or recurring?
- Was software or hardware maintenance applied?
- Did the failure occur while you were doing a specific task?
- Product version, release, and maintenance level. (For IBM Record Generator for Java, this is written out to the terminal.)
- Operating system version, release, and maintenance level
- The version and release levels of related products.

Chapter 10. Notices

This information was developed for products and services offered in the U.S.A. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property rights may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com[®] are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml) at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Chapter 11. IBM Record Generator for Java V3.0

IBM Record Generator for Java V3.0 provides a stand-alone utility that generates Java helper classes based on the associated-data (ADATA) files produced from compiling COBOL copybooks or assembler DSECTs. These Java helper classes can then be used in a Java applications to marshal data to and from the COBOL or assembler language-specific record structures.

Learn more

[“What is IBM Record Generator for Java?” on page 1](#)

IBM Record Generator for Java V3.0 is a stand-alone Java utility that generates Java helper classes to describe language-specific record structures. These helper classes can then be used in a Java application to marshal data to and from the byte-oriented record structures that are commonly used in z/OS applications, such as CICS COMMAREAs or VSAM files.

[Product Legal Notices](#)

[“What's new in V3.0.1” on page 5](#)

IBM Record Generator for Java V3.0 supersedes the alphaWorks version of the JZOS Record Generator V2.4.6, and provides new capabilities.

[Articles from the experts](#)

[Announcement](#)

[Get this doc in PDF](#)

Try it

[Download](#)

Get support

[Ask a question](#)

[Request a feature](#)

[IBM Z on Twitter](#)

[IBM Support Portal](#)

Notices

This information was developed for products and services offered in the U.S.A. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property rights may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at [Copyright and trademark information at www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

