English

Fujitsu Software BS2000

JENV
Development and Runtime Environment

User Guide

Valid for:
JENV V17.0A

June 2025

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com.

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Table of Contents

1 User Guide

Introduction

This documentation for the BS2000 Environment For Java™(JENV) explains the main points of calling Java commands insofar as they differ from Oracle's original description. It also describes the special features which arise from the conversion from ASCII to EBCDIC, and from working with the Java Native Interface (JNI) within the context of JENV V17.0A. JENV V17.0A is an implementation of the "Java Platform, Standard Edition" (Java SE™ based on OpenJDK 17 for FUJITSU Software BS2000/OSD-BC with the full name "BS2000 Environment for Java™ V17.0A".

The product includes a runtime environment (JRE) that complies with the relevant specifications:

- The Java™ Language Specification, Java SE 17 Edition"
  https://docs.oracle.com/javase/specs/jls/se17/html/index.html ,
- The Java™ Virtual Machine Specification, Java SE 17,
- die versionsspezifische API Spezifikation
  „Java™ Platform, Standard Edition 17 API Specification"
  https://docs.oracle.com/en/java/javase/17/docs/api/index.html .

The product also includes a software development kit (JDK) with a range of development tools. These can be used to develop applications or applets that comply with the above API specification.

JENV V17.0A supports all features of OpenJDK with the following exceptions:

- Audio-Features
- JDGA (Java Direct Graphic Access)
- Oracle Java Mission Control
- Java Flight Recorder
- Advanced Management Console
- Usage Tracker
- Java Web Start
- the Java-plugin for Web Browser
- the modules jdk.hotspot.agent and jdk.internal.vm.ci.

JENV V17.0A also includes font files from the DejaVu Fonts Package.

The only VM technology used is the HotSpot client VM.

The OpenJDK demo programs are not contained in the product.

By default, SerialGC is used as the garbage collector instead of the G1 garbage collector.

Further differences to OpenJDK (e.g. command line options of the tools, available commands) are shown in the section "Commands for BS2000".

Module concept

The packages com.fujitsu.ts.java.bs2000 and com.fujitsu.ts.java.io are included in the module java.base.
The com.fujitsu.ts.jrio package is included in the jdk.jrio module

Optimized variant for SE systems

Exactly one optimized variant is provided for all SE /390 and x86 systems. Depending on the architecture, the corresponding operating mode of JENV is used automatically.

Objectives and target groups of this manual

The documentation is intended for all those who wish to use Java™ for development work and/or in their system environment.

Summary of contents

Only the special BS2000 features and the special BS2000 parts are described in this manual. Knowledge of the original description of Oracle is a requirement.

Conversion from ASCII to EBCDIC

Java is a product which was developed in an ASCII world (Unix systems and Windows systems). In an operating system based on EBCDIC code, therefore, you will notice a number of peculiarities when working with code sets, localized streams, print streams, and standard streams, for example. These peculiarities are described in this documentation.

JNI under BS2000

This documentation also describes a number of special features that you as a user of Java Native Interfaces (JNIs) in BS2000 must take into consideration, such as the use of Java data types in C and the dynamic loading of native methods.

Contents of the documentation

This manual has the following contents:

- The chapter "Environment variables" contains a description of the file structure, how to use the classpath and the environment variables.
- The chapter "Conversion from ASCII to EBCDIC" describes the special issues that need to be taken into account as a result of the different code set used by BS2000 (EBCDIC).
- The chapter "The Java package JRIO" describes the interfaces and the implementations of JRIO.
- The chapter "Invoking the VM from the BS2000 command interface" describes the procedures *INITIALIZE*, *DELETE* and *START*.
- The chapter "JNI under BS2000" explains the special issues that users of Java Native Interfaces (JNI) must take into account in BS2000.
- The chapter "JCI - Invocation API for COBOL" describes the particularities, that a user of the Java-COBOL-Interface (JCI) in BS2000 must observe.
- The chapter "Commands for BS2000" describes the *mk_shobj* and *pr_shobj* commands that have been additionally implemented in JENV and the commands whose description deviates from that in “JDK Tools and Utilities" [11].
- The chapter "Appendix: Compatibility with earlier versions and migration" describes incompatibilities between JENV V17.0A and predecessor versions and it describes how to migrate from earlier versions to JENV V17.0A.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at *https://bs2manuals.ts.fujitsu.com*.

Notational conventions

This documentation uses the following notational conventions.

- Description of commands
- Names of files, commands and programs
- Description of execution sequences

Description of commands

The description of the commands keeps - where possible - to a fixed framework:

Syntax

Shows the command syntax.

Description

Meaning, function, and mode of operation of the command. Where necessary, an explanation of the prerequisites or conditions to be adhered to is provided.

Options

Description of the relevant command line options.

See also

Further sources of information relating to the command described.

Syntax representation

The metasyntax used has the following meaning:

Bold characters

Constants. Bold characters must be entered exactly as shown.

Normal characters

Variables. These strings represent real specifications that you enter or select.

Italics

Variables in options, which you have to replace with real specifications.

[ ]

Options. Arguments in square brackets are optional. The square brackets themselves must not be entered.

...

The previous expression can be repeated.

{ | }

Selection option. Chose precisely one of the expressions separated by vertical lines. The braces themselves must not be entered.

Names of files, commands and programs

Names of files, commands, and programs etc. are shown in the text in *italics*. If variables occur, they are placed in <*angle*> brackets.

Description of execution sequences

Activities are subdivided into individual steps:

- Step which is part of the overall operating sequence. This is where you enter a command or perform an action.

Further information and sources

You will find further information about Java™

- in the chapter "Related publications"
- under the Web page with the URL *https://www.fujitsu.com/emeia/products/computing/servers/mainframe/bs2000/software/programming/javabs2000.html*

License regulations

JENV V17.0A is Open Source Software.
JENV is based on a port of OpenJDK 17.
All relevant license information can be found in

`SYSDOC.JENV.170.OSS`

oder on the internet at

JENV170 license information.

Environment variables

This section describes the following environment variables:

CLASSPATH

The syntactical structure of the *CLASSPATH* environment variable corresponds to that of the *PATH* environment variable and describes the directories and JAR and ZIP archives in which the user classes are searched for.

When using the *java* commands and tools, users must only define this environment variable so that their own classes are found. If the environment variable is not set, the search path for user classes is set to the current directory.

Alternatively, the -classpath option can also be used for the JAVA interpreters to define the path to the user classes.

JAVA_HOME

The environment variable *JAVA_HOME* describes the installation location of the JAVA runtime environment. It is only needed for application programs which access JAVA using the invocation API.

For a standard installation JAVA_HOME is to be set to /opt/java/jdk-17.0.5. Refer to the Release Notice for the currently valid name

The Java tools use their own mechanisms to determine their installation location. This environment variable should thus no be set if the Java Interpreter and the other Java tools are to be used.

JENV_VMTYPE

For user programs which utilize the invocation API no interface exists to select the VM for processing. This environment variable can be used to request a special VM for such programs. The following values are permitted:

client

Selection of the HotSpot™ client VM

If the variable is not set, the default applies (see subsection "Options for selecting the HotSpot™ VM type" in section "java"). However, because only one VM implementation is currently available, this variable is not needed.

The Java tools do not use this environment variable but evaluate the corresponding command line options.

JENV_SYSHSI

The environment variable *JENV_SYSHSI* specifies the HSI variant to be used for the VM when calling the *java* command (see section "java"). The following values are possible:

s390

The S390 variant of JENV is used (if available).

x86

The X86 variant of JENV is used (if available).

If you don’t specify the variable, the default value is used, as desribed in section "Options for selecting the HSI variant". In case you explicitly specify the variant in the *java* command, this value precedes the environment variable.

LD_LIBRARY_PATH

The environment variable *LD_LIBRARY_PATH* describes the directories in which a search will be made for “Shared Objects” with the user’s native methods. In its syntactical structure it corresponds to the environment variable *PATH*.

Other mechanisms are used for the search for native methods of Java implementation. With applications that use the invocation API, they are found using *JAVA_HOME* for example.

Conversion from ASCII to EBCDIC

The Java SE JDK was developed in an ASCII environment (Unix systems and Windows systems). Since the BS2000 code set is quite different (EBCDIC), therefore, you will notice a number of peculiarities which are described below.

Code sets

In ASCII-based operating systems, the partial identity between ASCII and Unicode means that it is not always necessary to distinguish between text and binary input/output. However, in BS2000 (and other non-ASCII-based operating systems, such as OS/390), this distinction is extremely important. If this is not taken into consideration in Java programs, not only will they not be portable, but they will have to be modified if they are to function correctly on BS2000.

Java works internally in Unicode. For communication with the outside world Java can use any code. For the input /output of text data, the new classes
InputStreamReader and *OutputStreamWriter*, which perform the appropriate code conversions, have been introduced in JDK 1.1. The standard code conversion which is used here is determined by the value of the system property *file.encoding*. By default this is set to OSD_EBCDIC_DF04_1. When Java is called, this setting can be changed either globally via *-Dfile.encoding=XXX* or else locally through specification of an appropriate code set during instantiation of the classes *InputStreamReader* and *OutputStreamWriter*.

Supported code sets

The following code sets are additionally supported in BS2000 and accordingly are **not** available in other Java implementations:

OSD_EBCDIC_DF04_1

> Default code set in BS2000. It is the same as the EBCDIC.DF.04-1 character set, except that the EBCDIC characters x'15' and x'25' are swapped, so that x'15' is interpreted as the character for newline. This is in keeping with current practice in POSIX and in C programming in BS2000.
>
> This character set is compatible with the ISO 8859-1 character set, the default character set used in Unix systems. “Compatible” here means that it contains the same character set and can therefore be mapped 1:1, it is just that encoding is different.

OSD_EBCDIC_DF03_IRV

> EBCDIC.DF.03.IRV (international reference version) character set, in which, once again, x'15' is the character for newline.

OSD_EBCDIC_DF04_15

> This is the same as the EBCDIC.DF.04_15 character set, except that the EBCDIC characters x'15' and x'25' are swapped, so that x'15' is interpreted as the character for newline. This is in keeping with current practice in POSIX and in C programming in BS2000.
> This character set is fully compatible with the ISO 8859-15 character set. “Compatible” here means that it contains the same character set and can therefore be mapped 1:1, it is just that encoding is different.

Specification of the code set

The commands *javac*, *javadoc* and *native2ascii* support the *-encoding* option, which allows you to specify the character set for the files to be accessed by the command.

Localized streams

For JENV, as for OS/390, various new classes and methods have been implemented for localized streams, with the result that a number of ASCII/EBCDIC problems have been resolved. As an applications programmer, however, you are advised to restrict yourself to the *InputStreamReader* and *OutputStreamWriter* classes defined by Oracle America Inc. for inputting and outputting text.

The new classes implemented for this purpose are as follows

- *com.fujitsu.ts.java.io.LocalizedInputStream*
- *com.fujitsu.ts.java.io.LocalizedOutputStream*
- *com.fujitsu.ts.java.io.LocalizedPrintStream*

These classes cannot be instantiated, but they do offer a static method *localize()*, which converts a specified stream into a “Localized Stream” if the specified stream is based on a file.

These methods are:

- *com.fujitsu.ts.java.io.LocalizedInputStream.localize(InputStream)*
- *com.fujitsu.ts.java.io.LocalizedOutputStream.localize(OutputStream)*
- *com.fujitsu.ts.java.io.LocalizedPrintStream.localize(OutputStream)*

These methods now actually return an *InputStream* or *OutputStream* in BS2000 for which the behavior is modified in relation to the original stream in such a way that the entire I/O via this steam is subject to code set conversion from or into the implemented standard code set (value of system property *file.encoding*). However this only occurs for streams which are based on files. These methods have no effect on other streams (e.g. *ByteArray*).

These streams modified in this way thus behave in a similar way to the objects of the new classes *InputStreamReader* and *OutputStreamWriter*, but in contrast to them, remain of data type *InputStream* or *OutputStream*, and can thus be used wherever only objects of this type are permitted.

There are in-built precautionary features against double conversions. Thus, a stream cannot be “Localized” twice. If a *getLocalized...* method is called for a stream which has already been localized, that stream is simply returned. An instance of *InputStreamReader* or *OutputStreamWriter* can also be formed from a “Localized Stream” without any danger of this causing double conversions.

This JENV-specific extension can be deactivated by setting the system property *java.localized.streams* to the value *False*. This can be achieved if Java is called via *-Djava.localized.streams=False*.

Property files

Property files can be written and read with the methods *store()* and *load()* in the class *java.util.Properties*. If the specified streams are file streams, it is assumed in BS2000 that these files are read or created in the default code set (value of system property *file.encoding*).

This does not happen if this JENV extension for the “Localized Streams” has been deactivated (see section "Localized streams"). Property files are then always written or expected in the ISO8859-1 encoding (i.e. ASCII encoding).

This behavior is compatible with that on IBM systems.

Policy files

Policy files used by the default policy implementation must be coded in UTF-8 code set. Because the first 127 characters of the UTF-8 code set are identical to those of the ASCII code set, users can generate a file in this code set by first creating the file with the editor in the normal native code set (*OSD_EBCDIC_DF04_1*) and then using the *native2ascii* tool to convert the file to the ASCII code set.

> **CAUTION!**
>
> *When the new file is generated, native2ascii* does not transfer the access rights of the old file. If necessary, these must be changed using *chmod*.

As of version JENV V1.4B the system property *sun.security.policy.utf8* is provided which you can use for policy files with native codeset. *sun.security.policy.utf8* can have the values *true* or *false*. You therefore can use policy files in native encoding with the following call:

```
java -Dsun.security.policy.utf8=false...
```

We however recommend to use UTF-8 encoded policy files.

PrintStream

The output streams of type *java.io.PrintStream* are not modified as the default option in the BS2000 port, but are mentioned here because they can cause special difficulties.

Methods of the java.io.PrintStream class

In accordance with the Java API specification some methods in the class *java.io.PrintStream* convert their outputs into the default code set (value of the system property *file.encoding*), whereas others do not. With this class it is therefore extremely easy to write programs which apparently function in an ASCII world but do not deliver the expected results in BS2000. The following simple example will illustrate this point:

Example

```
...
PrintStream out = new PrintStream(new
                                  FileOutputStream("test"));
...
out.print("This is a text.");
out.write('\n');
...
```

In an ASCII-based system the content of file *test* will then be a line ending with newline and containing the above text. In BS2000 the file would contain an EBCDIC-encoded version of the text, however the line would not end with newline but would contain a “smudge” as the last character.

This example shows clearly how important it is for the input/output of text in a new implementation of Java code to use the new read and write classes (i.e. *InputStreamReader* and *OutputStreamWriter*).

In BS2000 an additional option is available which changes the behavior of *PrintStream* so that no conversion is performed by any method any more. This can be achieved if Java is called via *-Djava.localized.print=False*. With this setting, the class *PrintStream* no longer behaves in accordance with the specification; however, this can actually be useful for existing applications.

For the sake of completeness, mention should be made of the fact that the use of “Localized Streams” as the basis for PrintStreams or the localization of a PrintStream does not result in multiple conversions. However, for PrintStreams handled in this way it is of course then the case that **all** methods convert.

Interaction between the readLine() and println() methods

It is often assumed that data written with *println()* to a *PrintStream* could be reread by the *readLine()* methods of some *InputStream* classes. In BS2000, however, this assumption may result in an error. This is due to the fact that although data will be converted to the native code set (OSD_EBCDIC_DF04_1 in BS2000) during output to a *PrintStream*, this is not carried out by any of the *readLine()* methods of the *InputStream* classes during a read operation. Instead, you should use the new *Reader* and *Writer* classes or use “Localized Streams” for input.

Standard streams

The class *java.lang.System* provides three standard streams *in*, *out*, and *err*. By analogy to the solution in OS/390, these standard streams are “Localized Streams” in JENV. This means that normal text input and output is possible in BS2000 via these streams.

This can be set selectively for each of the three streams if the following system properties are defined when the program is started:

```
System.in     -Djava.localized.in=...
System.out    -Djava.localized.out=...
System.err    -Djava.localized.err=...
```

These streams are not modified if the extension for “Localized Streams” is deactivated (*-Djava.localized.streams=False*). Setting or amending these system properties later on has no effect on the currently defined standard streams either.

The following values can be specified:

Default

The original streams (which are set when the program is started) are localized. This is the default value.

Full

Both the original streams and also the standard streams which are set later on using *setIn()* etc. are localized.

None

The standard streams are not modified.

If an application uses the methods *setIn()*, *setOut()*, or *setErr()* in order to assign its own streams, there are two options for guaranteeing correct operation: either you must ensure that all standard streams are “Localized Streams” (i.e. text streams), or see to it that a clear distinction is made between text and binary input/output when using standard streams. The following example shows both options.

The second option is the preferred solution, and should be applied as a matter of principle when working with standard streams. However, the first option may be necessary if you are working with existing Java classes which have not been implemented in a portable fashion.

Example

The following code (similar examples of which can be found in the JavaSoft demo programs) would lead to a binary input/output via these streams, with the result that the output files would be unreadable or the input might be misinterpreted if text input/output was really intended.

```
...
public static String read_write() {
     StringBuffer buf = new StringBuffer(80);
     int c;
     try {
          while ((c = System.in.read()) != -1) {
               char ch = (char) c;
               System.out.write(c);
               if (ch == '\n')
                   break;
               buf.append(ch);
    }
   } catch (IOException e) {
           System.err.println(e);
           }
     return (buf.toString());
  }
...
System.setIn(new FileInputStream("myinputfile"));
System.setOut(new PrintStream(new FileOutputStream("myoutputfile")));
...
line = read_write();
...
```

The following program fragment shows the first option, where all standard streams are “Localized Streams” (i.e. text streams). This solution would have to be implemented by the calling program.

```
...
System.setIn(com.fujitsu.ts.java.io.LocalizedInputStream.
localize (new FileInputStream("myinputfile")));
System.setOut(com.fujitsu.ts.java.io.LocalizedPrintStream.
localize (new FileOutputStream("myoutputfile")));
...
line = read_write();
...
```

The code fragment for the second solution could look like this and would have to be implemented by the user of the standard streams. It involves making a clear distinction between text and binary input/output when using standard streams.

```
...
public static String read_write() {
   StringBuffer buf = new StringBuffer(80);
   int c;
   InputStreamReader in = new InputStreamReader(System.in);
   OutputStreamWriter out = new OutputStreamWriter(System.out);
   try {
        while ((c = in.read()) != -1) {
          char ch = (char) c;
          out.write(c);
          if (ch == '\n')
              break;
          buf.append(ch);
          }
   } catch (IOException e) {
         System.err.println(e);
        }
   return (buf.toString());
}
...
```

JAR archives

In the context of the problems associated with ASCII/EBCDIC conversion, JAR archives can create special difficulties because they also constitute an exchange format between different environments (systems). You can pack applets including all their resources into JAR archives and load them over the network by a browser. Java offers corresponding methods for accessing the resources packed in this way (see *java.util.ResourceBundle*).

The typical resources here often also include property files (e.g. with error messages). To ensure interchangeability, property files which are stored in JAR archives must therefore always be in `ISO8859-1` encoding i.e. they must previously be converted into this code set by the creator of such a JAR archive in BS2000.

If the user introduces a manifest file of his/her own into the JAR archive (option *-m*):

- The manifest file is generated by the *jar* command itself, this occurs automatically using ISO8859-1 encoding.
- If the user creates the manifest file himself, it must first be converted into the ISO8859-1 code set.

The methods for accessing these resources in JAR archives are designed so that they also expect ASCII input in BS2000.

To support the code conversion of files, the command *native2ascii* is provided.

Program arguments

The call arguments which are transferred to the method *main()* of a Java program are automatically converted from EBCDIC to Unicode.

The Java package JRIO

The JRIO package is a collection of Java classes to permit direct handling of files with a record or block structure and for record- or block-oriented input/output to such files. Naturally these files include above all the BS2000 files of DMS/DVS.

In contrast to normal Java I/O (*java.io package*), these interfaces also allow operations which cannot be expressed with the given Java IO classes (which we cannot extend).

The interfaces and implementations of JRIO are contained in the proprietary package *com.fujitsu.ts.jrio* and further subpackages. These will not be available in other Java implementations. However, as far as it makes sense technically, they are very similar to the corresponding IBM package *com.ibm.recordio*.

Concepts

The implementation of JRIO is geared to extensibility. The sections below describe these concepts and their realization.

File systems

Unlike with the normal Java IO classes, various file systems are supported by concept under JRIO (see section "RecordFile"). The following file systems will supported in future versions:

- the BS2000 file system (referred to as DMS in the following)
- the hierarchical file system POSIX (referred to as UFS in the following)
- the BS2000 library file system (referred to as LMS in the following)

In this version only DMS will be supported initially.
Each of the file systems has its own syntax for specifying file names. When a *RecordFile* object is created this is associated uniquely and permanently with one file system. This allocation can be specified either implicitly or explicitly by the user and cannot be modified later. The allocation then determines the semantics of most of the methods of the *RecordFile* object.

File names in the DMS file system

File names in the DMS file system are formed in accordance with this file system's rules (see manual "Introductory Guide to DMS" [8]. Partially qualified file names and wildcard specifications are not supported at any of the JRIO interfaces with the exception of the specifications permissible as a directory and the file identifier in policy files (see also section "Security").

Only the specification of a lone catalog ID (Catid) which is enclosed in colons, a user ID (Userid) with a leading dollar character and a closing period or a combination of the two are regarded as directories in the DMS file system. The customary special way of specifying the system standard ID is also permitted. Consequently only the following specifications are possible for directories in DMS:

```
:catid:
$userid.
:catid:$userid.
$.
:catid:$.
```

As DMS is a flat file system and actually has no directory concept, directories cannot be set up or deleted with the interfaces provided here. Neither do they have attributes such as modification date or a size. Only the methods for listing directory contents are practical for the above-mentioned artificial directories of DMS.

As in the DMS interfaces, the so-called logical system files (SYSFILE environment) are not supported. In addition, JRIO does not support EAM files, either.

Normalized path names

When generating a *RecordFile* object and at locations where the user can specify a file or path name at the JRIO interfaces, not only the syntax and semantics check is performed, but also what is known as normalization. For DMS files this normalization of the name means that any lower-case letters contained in the name are converted to upper-case letters. In addition, file names which contain no periods but begin with a dollar character ($) are converted into names with a leading system standard ID in accordance with the DMS conventions:

Example

```
$EDT => $.EDT
```

Absolute path names

A path name in DMS is regarded as absolute if it begins with a catalog ID. Thus when an absolute path name is generated this means that if a catalog ID is not already contained in the name, the default catalog ID of any user ID specified or of the calling program is added (see section "RecordFile").

Canonical path names

A path name in DMS is canonical if it consists only of a catalog ID or contains both a catalog ID and a user ID. Thus when a canonical path name is generated, this means that (if it is not already included) the default catalog ID of any specified user ID or that of the calling program (see section "RecordFile").

File names in the UFS file system

The same rules apply for the syntactical structure and the semantic definition as for the *java.io.File class*. The terms „“absolute path name”, “canonical path name” and “normalized path name” are also used in the same way at these interfaces.

File types

The following file types are currently supported in the DMS file system:

- SAM files with fixed or variable record length.
- ISAM files with fixed or variable record length.
- PAM files.

The UFS file system does not distinguish between file types. In particular, there are no defined file types with record /block structure. Only the content of a file and the processing programs determine what can happen to it or what it is intended for (see section "Access methods").

As with *java.io.File*, only regular files and directories are supported under JRIO.

Access methods

The term access method is normally used to refer to a set of interfaces which permit access to data from files and thus offer a particular logical view of this data. Generally this logical view will differ to a greater or lesser extent from the physical storage in the file (the data repository). Here access methods which enable record-oriented processing of the data in a file are of interest; the logical view is thus restricted. The access methods considered here thus implement the following:

- Definition of a record and mapping of this logical view of the data onto a physical storage form (file type).
- Definition of the order of the records from the viewpoint of the user or program; this order need not necessarily have anything to do with the physical order of the data in the file.
- Interfaces to read and write records in their entirety.

Elementary access methods are a special type of access method. These are distinguished by the fact that the file system (in which they are effective) knows of them and, for example, a file and access method can already be assigned to each other via file types. Such access methods are provided in DMS, albeit not reversibly unique. Other file systems (for example UFS) know only a single elementary access method which generally offers the raw physical view of the data and logically also has no content-oriented file types.

However, there can also be any further access method desired, these generally being implemented using one of the elementary access methods and offering further logical views of the data. These access methods have one problem in common. As the file system has no knowledge of them, it is not possible to tell from a file whether and with which of these access methods it can be successfully processed. Interpretation errors will thus only be recognized during processing, if at all.

In UFS there is no elementary access method which offers record-oriented processing. However, the following access methods, for example, are conceivable:

- **TEXT** - Access method which regards text files as record-structured files with records of variable and unlimited length. The physical record separator would be the new line character, which would be masked out in the logical view.
- **CISAM** - ISAM-type access method for Unix file systems.

In DMS there are several elementary access methods, of which the following are supported directly in JRIO:

- **SAM** - sequential access method
- **ISAM** - indexed-sequential access method
- **UPAM** - block-oriented access method

In DMS, too, there are access methods which are based on one of the elementary access methods and supply a different logical view. A prominent example of these is an access method based on ISAM which is used by various tools (editors, compilers, ...) to render ISAM files usable for normal texts. For this purpose, ISAM files with standard keys are used. In the logical view these keys (which in ISAM are a part of the record) are masked out, and are generated by the access method when records are written.

In the JRIO interfaces you always will encounter the access methods when you must take a decision as to how access to a file is to be implemented.

Currently JRIO only supports the DMS file system, and in this only the access methods SAM, ISAM and UPAM. However, the JRIO architecture will in future permit extension by the addition of further file systems without the user interfaces needing to be modified.

Access types

The starting point for designing the JRIO interfaces is an abstract view of the type and manner of data access (of the sort that is also taken as a basis in the IBM implementation) which is independent of file systems and access methods.

From the viewpoint of the application, the type and manner of data access can then be classified in the following access types:

- Sequential access

  Read access to records/pages takes place sequentially. Write access extends the file at the end.
- Random access

  In a file processed using this access type any individual records be positioned to before reading or writing.
- Keyed access

  In a file processed using this access type, individual records can be selected for reading and/or writing by specifying keys.

Shared update processing

By means of locks, JRIO permits the simultaneous, synchronized processing of a file by multiple applications (*shared update processing*) if this is supported by the particular file system and access method.

This type of processing must be explicitly set by the application when the file is opened. It ensures that the processing steps (e.g. write, delete or a combination of read and write) are protected by locks and cannot be interfered with by competing applications. *Shared update processing* may be subject to file system-specific restrictions. For example, it may not be permitted for certain file types or open modes, or it may not support certain actions such as increasing or reducing the size of files.

The lock mechanism employed by JRIO in *shared update processing* is:

- record-oriented,
- implicit,
- deadlock secure.

Record-oriented means that *logically* an application locks or releases records only within a file. However, certain file systems or access methods can *physically* implement a larger lock granularity. This is not visible to your own applications but competing applications may encounter a lock when they attempt to access a record within the larger lock granularity although the requested record itself is *logically* locked.

Implicit means that records are implicitly locked when they are read, written or deleted, and that the lock is implicitly released after the write or delete operation has been completed. Methods are also offered for the explicit release of records that are locked for reading but are not to be written.

Deadlock security is achieved by ensuring that an application can only ever logically lock one record per file. Setting a lock for an operation leads implicitly to the release of any other lock for another record. Some file systems and access methods are also able to implement deadlock security beyond file boundaries; in other words, only one lock per application is permitted - regardless of in which file.

In *shared update processing* JRIO allows the behavior of the application to be controlled in the event of access conflicts. The application can demand immediate transfer of control (*NO_WAIT parameter*). In the event of access conflicts, a corresponding exception (*RecordLockedException*) is then generated or the application can wait for granting of the lock as a thread (*THREAD_WAIT* parameter) or at the system interface (*APPLICATION_WAIT parameter*). The wait time is unlimited in both cases, i.e. the application waits until a lock is received or until the application itself is terminated. Waiting as a thread has the advantage that other threads of the application are not blocked. However, in extreme situations it can happen that the lock is received by a competing application at the very moment that the waiting application makes a renewed attempt although the lock was available in the meantime. Not all file systems offer all wait variants. If, however, a variant is offered, the semantics described then apply.

Options and restrictions relating to access types in DMS

Not all access types are possible with all access methods/file types. The following table provides an overview of the relationship between access types and access methods/file types:

Access type	SAM access method	ISAM access method	UPAM access method
Sequential	Reading/writing for SAM files. Physical record sequence.Shared update processing is not possible.	Reading/writing for ISAM files. Record sequence determined by primary key.Shared update processing is possible for reading or adding to opened files.	Reading/writing for PAM, SAM and ISAM files. Physical block sequence. Shared update processing is possible only for PAM files opened for reading.
Random	Reading/writing for SAM files. When records of variable length are overwritten, the record to be written must be of the same length as the record to be overwritten. Shared update processing is not possible.	Not possible.	Reading/writing for PAM, SAM and ISAM files. Shared update processing is possible only for PAM files opened as “INPUT” or “INOUT”.
Keyed	Not possible.	Reading/writing for ISAM files. Shared update processing is possible for files opened as “INPUT” or “INOUT”. Only the first opening application may open “OUTIN”.	Not possible.

Table 1: Overview of the relationship between access types and access methods/file types inthe DMS file system

Drivers

JRIO has a dynamic driver concept that separates both the file system implementations and the implementations of the various access methods from the JRIO user interfaces. New file system drivers or access method drivers can be added without user interfaces needing to be modified.

When an application is started, it is determined dynamically which drivers are available for file systems and access methods. These are then loaded dynamically as required. However, the associated interfaces (in particular the driver API) and the configuration mechanisms are currently not to be made accessible to users and are consequently not described here.

Security

Applications that use JRIO and run under a *Security Manager* are started with, for example, the following command:

```
java -Djava.security.manager <application-name>
```

All accesses to files and directories of the supported file systems are initially rejected by the *Security Manager*. Access is granted only to files in the UFS directory that contains the loaded class.

When handled by the *Security Manager*, UFS files are subject to the same mechanism in JRIO as offered by *java.io* . The special features of the DMS file system are therefore described below.

To allow an application to access certain files and directories of the DMS file system under the *Security Manager*, appropriate *permissions* must first be granted in a policy file. The mechanism for selecting the valid policy file is no different from the usual method in Java; in particular, the policy file can also be specified directly:

```
java -Djava.security.manager
     -Djava.security.policy= <policy-file> <application>
```

JRIO features two new *permissions* that can be granted in the policy file:

`com.fujitsu.ts.jrio.DMS.FilePermission`

`com.fujitsu.ts.java.bs2000.SystemInfoPermission`

> You can make entries in the policy file using any normal editor.
>
> Note that the policy file must be available in UTF8 coding.

File permission

com.fujitsu.ts.jrio.DMS.FilePermission controls access to files and directories. The syntax of an entry in the policy file is as follows:

```
grant [codeBase ... | signedBy ...] {
     permission com.fujitsu.ts.jrio.DMS.FilePermission
     "file-identifier" , "action-list";
};
```

The *file identifier* is either a valid BS2000 directory name or a valid BS2000 file name with or without catalog ID and /or user ID , i.e. a catalog ID (in the format ":catid:"), a user ID (in the format "$userid.") or a combination of the two (in the format ":catid:$userid."). The last character of the file name may be “ * ”. Access permission then relates to all files whose name begins with the string preceding “ * ”. In this case, it need only be possible to complete the name part preceding the “ * ” to form a valid file name. For catalog and user IDs you can also use “:*:” or “$*.” to grant access for all catalog IDs or all user IDs. The abbreviation “$.” for the default system ID is permitted but not the abbreviation “$file” for “$.file”. Refer to the section "File names in the DMS file system".

If no user ID is explicitly specified, permission relates to files under the user ID of the caller (who need not be known by name to the application). If no catalog ID is specified, permission relates to files of the default catalog ID of the corresponding user ID. The string <<*ALL FILES*>> permits access to all files and directories. Further details are provided in the shipped JAVADOC documentation for the *com.fujitsu.ts.jrio.DMS.FilePermission* class.

The *action-list* is a comma-separated list of the permitted *read*, *write* and *delete actions for the file*. If permission to perform the action is not granted in this file or directory, any access attempt is rejected with a *SecurityException*. This also applies to information functions such as *list()* or *listFiles()* that require read permission for the underlying directory.

SystemInfo permission

Within JRIO, *com.fujitsu.ts.java.bs2000.SystemInfoPermission* is used to control which information on the DMS file system the application is allowed to obtain. The syntax is:

```
grant [codeBase ... | signedBy ...] {
     permission com.fujitsu.ts.java.bs2000.SystemInfoPermission
     "Name";
};
```

Name is a value formed from *HomePubset*, *UserName*, *UserPubset*, *DefaultUserName*, *DefaultUserPubset* and *ForeignUserPubset* or the string *<<ALL INFO>>* with which permission is granted for all named data. If the *permission* is granted, the application is allowed to determine the corresponding catalog and user IDs via the *getCanonicalPath()*, *getCanonicalFile()*, *getAbsolutePath()* and *getAbsoluteFile()* interfaces of the *RecordFile* class. Otherwise, any attempt is rejected with a *SecurityException*. The names beginning with *User...* relate to the ID of the caller, the names beginning with *Default...* to the default system ID, and the names beginning with *Foreign...* to all foreign user IDs. Permission relates only to the interfaces that provide access to the corresponding file names when completed, but not to actual access to the files under these catalog or user IDs.

Example

An application is granted access to the file named *HUGO* under the ID of the caller although the application does not have permission to determine the ID of the caller:

```
grant [codeBase ... | signedBy ...] {
     permission com.fujitsu.ts.jrio.DMS.FilePermission
     "Hugo", "read, write";
};
```

This setting allows the file to be opened, read and written. However, completing the file name with, for example *, getCanonicalPath(...)* is not permitted.

API overview

The public classes which constitute the JRIO interfaces are shown below:

Class	Use
Record	Represents a record/block
BufferOverflowException	This exception is triggered when records are being read whenever the record object provided by the user is too small to incorporate the data.
RecordLockedException	This exception is triggered in shared update processing when a record that is locked by another application is accessed and the user has specified that the application should not wait for the lock to be granted.
RecordNotLockedException	This exception is triggered in shared update processing when an attempt is made to access a record using a method that requires the record to be locked first but the lock does not yet exist or no longer exists.
RecordFile	Represents a file with record/block structure (see *java.io.File*).
RecordFileFilter	Interface for implementing user-specific classes which can be used as filters in the *listFiles()* method of the *RecordFile* class (see *java.io.FileFilter*).
RecordFilenameFilter	Interface for implementing user-specific classes which can be used as filters in the *list()* method of the *RecordFile* class (see *java.io.FilenameFilter*).
AccessParameter	Represents the general parameters which are required for access to a file with record/block structure when using a particular access method.
DMS/AccessParameterSAM	Represents a selection of parameters which are required for access to a file (in particular generation) using the SAM access method in DMS.
DMS/AccessParameterISAM	Represents a selection of parameters which are required for access to a file (in particular generation) using the ISAM access method in DMS.
DMS/AccessParameterUPAM	Represents a selection of parameters which are required for access to a file (in particular generation) using the UPAM access method in DMS.
DMS/FilePermission	Permits the fine-grained granting of access permissions for files and directories in the DMS file system. This class is normally used only in the context of entries in the policy file.
InputRecordStream	Abstract base class for *FileInputRecordStream* and *ArrayInputRecord*Stream and user-implemented input classes (see *java.io.InputStream*).
ArrayInputRecordStream	Class for sequential reading of records from an array of records (see *java.io. ByteArrayInputStream*).
FileInputRecordStream	Represents a file with record/block structure that is open for sequential reading (see *java.io.FileInputStream*).

OutputRecordStream	Abstract base class for *FileOutputRecordStream* and *ArrayOutputRecordStream* and user-implemented output classes (see *java.io.OutputStream*).
ArrayOutputRecordStream	Class for sequential writing of records to an array of records (see *java.io. ByteArrayOutputStream*).
FileOutputRecordStream	Represents a file with record/block structure that is open for sequential writing (see j*ava.io.FileOutputStream*).
RandomAccessRecordFile	Represents a file with record/block structure that is open for random access (see *java.io.RandomAccessFile*).
KeyedAccessRecordFile	Represents a file with record/block structure that is open for keyed access.
KeyDescriptor	Describes a record key of an indexed-sequential file.
DMS /PrimaryKeyDescriptorISAM	Describes the primary key of an ISAM file.
DMS /SecondaryKeyDescriptorISAM	Describes a secondary key of an ISAM file.
KeyValue	Represents the concrete value of a record key.

Table 2: Public classes which constitute the JRIO interfaces

The sections below describe the most important of the classes from the JRIO package which are mentioned above, together with their principal and most common methods and fields. A complete description of the interfaces is contained in the javadoc documentation provided (please refer to the Installation directory under *doc/jrio*).

Record

A *Record* object represents a logical record of a file and consists of a record buffer which contains the actual data record and the separately administered length of the data within the record buffer.

The *Record* class provides methods to access the data in the record buffer and their length, and to set or modify these. No methods are provided for accessing numerical data fields; users can implement these themselves on the basis of the methods provided.

A *Record* object is typically used to store or transfer the data of the record-by-record or pageby-page access operations to files. It is serializable and can therefore be used for Remote Method Interfaces (RMI). The *Cloneable* interface is also implemented.

Positions within a record are counted starting with position 0 (the first data byte of a record thus has position 0 and so on). A logical data record of a file contains only the user data, while the data record stored physically in the file can contain additional meta information (for example record length). Consequently the numbering of record positions for example at the DMS macro interfaces of BS2000 (these supply the physical record) can differ from that at the JRIO interfaces (these supply the logical record).

Constructors

When a *Record* object is generated, either an empty record buffer of a required size can be created or a buffer provided by the user can be used. If this buffer already contains data, the length of the data can also be transferred.

Typically you should select the size of the buffer so that there will be space in it for the longest expected record. The *Record* object can then always be reused for input/output if the old content is no longer required instead of repeatedly generating new instances.

A *Copy* constructor is also available which generates a new *Record* object from the data of another record. A one-to-one copy of a *Record* object can be generated with the *clone()* method.

General methods

The *getBuffer()* method returns the record's record buffer. You can use this to process or provide the content with the help of other classes and methods. Note that manipulations on this record buffer modify the object from which the record buffer originates because this is not a copy of the data. The current length of the data within the record buffer can be determined using the *getDataLength()* method.

A record buffer can be replaced using the *setBuffer() methods*. If the user's buffer which is transferred already contains data, the data length can also be transferred.

With the *setDataLength()* method users can themselves define the occupancy level of the record buffer. No check is made to see whether the data in the record buffer is useful.

Methods for extracting the data of a record

The *getByteData()* method enables all the data of a record to be returned in binary format (as bytes).

The methods of the *getStringData()* family return all the data of a record interpreted as text (string).

If no encoding for converting the data to text was specified by the user, the systemdependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

Methods for extracting the data fields of a record

The various methods of the *getByteField()* family enable the data of a specified data field (defined by position and length within the record) to be returned in binary format (as bytes).

The methods of the *getStringField()* family return the data of a specified data field interpreted as text (string).

If no encoding for converting the data to text was specified by the user, the systemdependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

The *getKeyField()* method returns, on the basis of a key description, the content of a key field as key value.

Methods for filling a record with data

The *setByteData()* methods fill a record completely with binary data (bytes); the old content is lost in the process. The data length of the record subsequently corresponds exactly to the length of the data entered.

The methods of the *setStringData()* family fill a record completely with text data (string). The data length of the record subsequently corresponds exactly to the length of the data entered.

If no encoding for converting text to data was specified by the user, the system-dependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

Methods for filling data fields of a record

The various methods of the *setByteField()* family fill binary data (bytes) into a specified data field (defined by position and length) of a record.

These methods update the data length if the record was lengthened when the record's data fields were filled. If the data is shorter than the selected data field, the rest can optionally be filled with a filler byte. If the data is longer than the selected data field, the length of the data transferred into the record buffer is limited to the length of the data field.

The methods of the *setStringField()* family fill text data (string) into a specified data field. If no encoding for converting text to data was specified by the user, the system-dependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

These methods update the data length if the record was lengthened when the record's data fields were filled. If the data is shorter than the selected data field, the rest can optionally be filled with blanks. If the data is longer than the selected data field, the length of the data transferred into the record buffer is limited to the length of the data field.

The *setKeyField()* method fills a record's key field with a concrete key value.

RecordFile

For record-oriented input/output, the *RecordFile* class plays approximately the same role as the *java.io.File* class for normal Java I/O. It defines the objects of the basic file system(s), in other words normally files and directories.

Unlike with the *java.io.File* class, different file systems are actually supported by the *RecordFile* class and not just one. Consequently a *RecordFile* object always consists of a path name (file or directory name) and an associated file system (DMS, UFS, ...).

Thus there can be objects with the same name, especially in different file systems. In BS2000 it is perfectly conceivable that a file named *HALLO* can exist both in UFS (Posix file system), in DMS (BS2000 file system) and also in LMS (as a member of a PLAM library). This approach consequently reflects the actual situation in BS2000 better than the monolithic approach of *java.io.File* (see section "File systems").

The *RecordFile* class (like *java.io.File*, too) provides methods and fields for analyzing and transforming the path name. These may be defined differently for each supported file system. For these operations it is normally unimportant whether the file system actually contains a file or directory with the name in question, because recourse is generally not made to the basic file system.

In addition, the *RecordFile* class also provides methods for accessing, and possibly modifying, the attributes of existing files and directories.

Furthermore, with the *RecordFile* class methods are provided for performing typical file system operations. These include renaming and deleting existing files and directories, creating files and directories which do not yet exist, and listing directory contents.

All methods which actually access the file system should be subject to the restrictions of the active *Security Manager* and trigger corresponding exceptions when access to the file system is restricted (see section "Security").

In the sections below the particular features relating to the UFS file system are generally not referred to. In these cases what applies for *java.io.File* for the Unix file system also applies for the UFS file system.

Basic structure of a file name

Generally a path name in all of the file systems supported comprises a file system prefix (if present) and no name part, or a sequence of one or more name parts which may be separated by separator characters. Each name part in a path name, except the last one, designates a directory. The last name part can designate either a directory or a file. The empty path name has no prefix and an empty sequence of name parts. Whether an empty path name is permitted and what the semantics of the path name is depends on the file system

The file system prefix or prefixes are defined on a file-system-specific basis. It is guaranteed that all root directories returned by *listRoots()* are permissible file system prefixes. In the DMS file system each catalog ID is interpreted as a file system prefix in this sense, regardless of whether this catalog ID exists in the file system, and in the UFS file system the root “/” is the only file system prefix.

If multi-part names are permitted in a file system, a separator is generally (but not always) defined with which the name parts are separated (for example “/” in UFS). However, the file system involved ultimately defines whether and how many name parts are permitted and how they are separated.

There is no defined separator for path names in the DMS file system. In addition to the file system prefix (the catalog ID with colons ':' at the beginning and end), the path name can also contain up to two name parts: a user ID (with dollar '$' at the beginning and period '.' at the end) and/or a file name. Even if both parts are contained in the path name, there is no additional separator between them.

The same naming rules are used for path names in the UFS file system as for *java.io.File*.

Constructors

A *RecordFile* object is formed from a given path name and a file system specification. Here a check is made to ensure that the given path name satisfies the syntactical rules of the specified file system, and what is termed normalization of the path name takes place. What this means specifically is defined separately for each file system supported (see section "File systems"). This normalized path name is then the name of this object and the basis of all operations on it.

There are constructor variants which permit path name specification in a different form, either simply as a single string, or separately as two strings which constitute the directory part and the file name part of the path name, or as a *RecordFile* object for the directory part and a string for the file name part. In the latter case the file system specification is omitted because the *RecordFile* object already includes this implicitly for the directory part.

Fields

The separator *separatorChar* or *separator* (in string form) is defined on a file-system-specific basis. Normally this separator is used to separate different name parts within a path name.

Special features of the DMS file system

The DMS file system knows no separators in this sense. Consequently the null character is used for *separatorChar* and an empty string for *separator*. However, this calls for care when it is used because the null character is a defined character within character strings.

The *separator pathSeparatorChar* or *pathSeparator* (in string form) is also defined on a filesystem-specific basis. This separator is used in order to separate the individual path names from one another when path name lists are specified.

Special features of the DMS file system

The separator for path name lists is the comma “ , ”.

Unlike in *java.io.File*, the separators are not static fields as several file systems are supported here. During instantiation of a RecordFile object the separators are initialized by the underlying file system.

General methods

The *getPath()* method returns the path name of this RecordFile object. The *getFileSystem()* method returns the name of the file system associated with the path name. The string “DMS” is returned for the DMS file system, and the string “UFS” for the UFS file system (currently not supported by JRIO).

Methods for analyzing and transforming path names

The *getName()* method returns the last name part of the path name of this RecordFile object. The result is formed by dropping any file system prefix there may be and every name part except the last. If the path name consists only of a name part and this is not a file system prefix, the object's path name is returned. If the path name is empty or consists only of the file system prefix, an empty string is returned.

Special features of the DMS file system

The file system prefix is the catalog ID.

Example

Name	Result
:JAVA:$USER.HALLO.JAVA	HALLO.JAVA
:JAVA:HALLO.JAVA	HALLO.JAVA
$USER.HALLO.JAVA	HALLO.JAVA
HALLO.JAVA	HALLO.JAVA
:JAVA:$.HALLO.JAVA	HALLO.JAVA
:JAVA:$USER.	$USER.
:JAVA:	empty string ""
$USER.	$USER.

The *getParent()* method returns the parent of this path name as a string, or the return value null if the path name has no parent. The parent of a path name consists of the file system prefix (if present) and of every name part, except the last, in the name sequence of the path name. If the name sequence is empty, then the path name has no parent.

Special features of the DMS file system

The file system prefix is the catalog ID.

Example

Name	Result
:JAVA:$USER.HALLO.JAVA	:JAVA:$USER.
:JAVA:HALLO.JAVA	:JAVA:

$USER.HALLO.JAVA	$USER.
HALLO.JAVA	null
:JAVA:$.HALLO.JAVA	:JAVA:$.
:JAVA:$USER.	:JAVA:
:JAVA:	null
$USER.	null

The *getParentFile()* method, like the *getParent()* method, returns the parent of this path name, but as a *RecordFile* object. If the path name has no parent, null is returned.

The *isAbsolute()* method returns true if the path name of this *RecordFile* object is an absolute path name. What an absolute path name is defined on a file-system-specific basis (see section "File systems").

Example

Name	**Result**
:catid:$userid.	true
:catid:$.	true
$userid.	false
$.	false
:catid:	true
`$.HALLO`	false
$USER.HALLO	false
:JAVA:$.HALLO.JAVA	true

The *getAbsolutePath()* method returns the absolute form of this RecordFile object's path name as a string. If the RecordFile object was constructed with the aid of an absolute path name, this name is returned. If this is not the case, the name is supplemented on a filesystem-specific basis (see section "File systems").

Special features of the DMS file system

In DMS it may not be possible to form the absolute path name for a syntactically correct path name. For example, a file name consisting of 42 characters with a 5-character user ID is syntactically correct. However, if it is complemented by a catalog ID comprising 4 characters, a syntactically incorrect (too long) path name results.

The *getAbsoluteFile()* method returns the absolute form of this RecordFile object's path name as a RecordFile object.

The *getCanonicalPath()* method returns the canonical form of this RecordFile object's path name as a string. A canonical path name is both absolute and unique. The precise definition of the canonical form depends on the file system (see section "File systems").

Special features of the DMS file system

In DMS it may not be possible to form the canonical path name for a syntactically correct path name. For example, a file name consisting of 42 characters with a 4-character catalog ID is syntactically correct. However, if it is complemented by user ID comprising 5 or more characters, a syntactically incorrect (too long) path name results.

The *getCanonicalFile()* method returns the canonical form of this RecordFile object's path name as a RecordFile object.

File name completion using the above methods provides the application with an insight into the structure of the file system and must therefore be monitored by an active *Security Manager*. In certain circumstances, file name completion is rejected with a corresponding *exception* (see section "Security").

The *compareTo()* method compares two path names lexicographically. If the two path names belong to different file systems, first of all the file system names are compared.

The *equals()* method compares two path names. It returns true only if the specified object is a *RecordFile* object which is assigned to the same file system and if the path names of both objects (in the sense of *compareTo()*) are equal. Equality is checked on the basis of the path names and not on the basis of the file or directory in the underlying file system, in other words if different names designate the same existing file, false is still returned.

The *hashcode()* method calculates a hash code from the characters of the path name and the file system name. Two *RecordFile* objects with the same path names and the same file system name also have the same hash code. However, two *RecordFile* objects with the same hash codes do not necessarily have the same path name.

Methods for inquiring file and directory attributes

The *exists()* method checks whether the file or the directory exists in the file system. Many of the methods offered can only be used effectively if the file or directory exists and is visible (for example files are not always visible for the calling program in foreign user IDs).

Special features of the DMS file system

A file is regarded as existing if it has already been opened once. This means that a catalog entry for the existence of a file is not sufficient. A directory exists if the specified catalog ID and/or user ID is accessible in the file system.

The *canRead()* method checks whether the file or the directory for this object exists (in the sense of *exists()*) and is readable for the calling program.

Special features of the DMS file system

true is always returned for existing directories.

The *canWrite()* method checks whether the file or the directory for this object exists (in the sense of *exists()*) and is writable for the calling program.

Special features of the DMS file system

Always returns false for a directory consisting only of the catalog ID if the calling program is not privileged, also for foreign user IDs.

The *isDirectory()* method returns true if an existing directory in the associated file system is involved.

The *isFile()* method returns true if a normal file in the associated file system is involved. A file is normal if it is not a directory and also meets file-system-specific criteria (for example special files in UFS are not normal files). Every file generated by a Java application which is not a directory is guaranteed to be a normal file.

The *isHidden()* method can be used to determine whether the file or directory is hidden in the file system. What precisely hidden means is defined on a file-system-specific basis.

Special features of the DMS file system

Temporary files in the DMS sense are always regarded as hidden.

The *lastModified()* method returns the time of the last modification to the file or directory if the file system supports this.

Special features of the DMS file system

Directories do not have their own modification date. Consequently 0 is always returned.

The *length()* method returns the size of a file or directory. How the size of a directory is defined is file-system-specific.

Special features of the DMS file system

For files the number of used (not reserved) PAM pages in the file multiplied by 2048 is returned, and for directories always the value 0.

The *getAccessParameter()* method returns the parameters for accessing this file with the specified access method. The *AccessParameter* object returned can, for example, be used to generate a new file with the same parameters and is used internally for file access.

The static method *getDefaultAccessParameter()* returns the default parameters for the given access method in the given file system. The user can then, if required, modify these parameters and generate new files with them.

The *getPreferredAccessMethod()* method returns the name of the preferred access method for an existing file in the associated file system. It is not guaranteed that the file was generated with this access method, especially if the access methods in this file system are only a logical view of the file contents and not inquirable file attributes.

The *getAllowedAccessMethods()* method returns a list of the names of the permitted access methods for an existing file in the associated file system. It is not guaranteed that the file was generated with one of these access methods, especially if the access methods in this file system are only a logical view of the file contents and not inquirable file attributes.

The static method *getAllAccessMethods()* returns a list of the names of all the access methods supported in the specified file system.

Methods for modifying file and directory attributes

The *setLastModified()* method sets the modification date to the specified value if the file system supports this.

Special features of the DMS file system

The modification date cannot be set.

The *setReadOnly() method* modifies the file attributes so that only read operations are permitted.

Special features of the DMS file system

For files the files attribute *ACCESS* is set to the value *READ*. This attribute cannot be set for directories and temporary files in the DMS sense.

Methods for generating files and directories

The *createNewFile()* methods generate a new file with the path name of the *RecordFile* object using the specified access parameters or the default parameters of the specified access method.

Special features of the DMS file system

When a file is generated, it is opened exclusively and then closed. Any *shared update option* set in the access parameters is ignored. The *shared update option* does not
takes effect until the file is opened for processing.

The static methods *createTempFile()* provide the option of creating temporary files using the specified access parameters. A temporary file has a generated name which definitely does not already exist in the file system. The file is either created in the specified directory or, if no directory was specified, in a file-system-specific directory. The file is not automatically deleted; the user must call the *deleteOnExit()* method if the file is to be deleted when the application is terminated. The name is formed from the user's prefix and suffix specifications and a string generated on a file-system-specific basis.

Special features of the DMS file system

No temporary files in the usual DMS sense are generated, but always permanent files which users must delete themselves. The file-system-specific directory always refers to the default catalog ID of the calling user. The directory parameter can be used in the DMS file system to generate temporary files on a different catalog ID from the default catalog ID.

When a temporary file is generated, it is opened exclusively and then closed. Any *shared update option* set in the access parameters is ignored. The *shared update option* does not take effect until the file is opened for processing.

The *mkdir()* method creates a directory with the name of this object in the associated file system.

Special features of the DMS file system

No directories can be created.

The *mkdirs()* method creates a directory including all the necessary parent directories in the associated file system.

Special features of the DMS file system

No directories can be created.

Methods for deleting and renaming files and directories

The *renameTo()* method renames this file or directory in the specified path names. The target file may not already exist. Renaming is only possible within a file system.

> The Record-File object itself, stays assigned to the old name, thus it may not represent an existing file afterwards.

Special features of the DMS file system

It is not possible to rename directories. Files can only be renamed if the same catalog ID and user ID are used.

The *delete()* method deletes this file or directory in the associated file system. If the path name designates a directory, this can only be deleted if it is empty.

Special features of the DMS file system

Directories cannot be deleted.

The *deleteOnExit()* method ensures that this file or directory is deleted in the associated file system when the application is terminated. Deletions are only performed if the application terminates normally.

Special features of the DMS file system

Directories cannot be deleted.

Methods for listing directories

The *list()* methods return a list of all or selected names of files and directories in the directory which this *RecordFile* object represents. The file/directory names are returned without parent directories. The order of the names returned is not defined. The files and directories returned can be selected using a filter (see *RecordfilenameFilter* in chapter "API overview").

Name	Results
:JAVA:	Only the home user ID, for example $USER., for non-privileged users or all user IDs in the pubset *JAVA* for a privileged user
:JAVA:HALLO.JAVA	Empty as no directory was specified
$USER.	Visible files of the user ID *USER* in their default catalog ID
$.	Visible files of the standard system ID
:JAVA:$.	Visible files of the standard system ID on the pubset *JAVA*

Table 3: Sample DMS file system

The *listFiles()* methods return an array of *RecordFile* objects with path names of files or directories in the directory which is represented by the path name of this *RecordFile* object. The path names created are created from the directory itself and the file and directory names that are ascertained. The order of the names is not defined. The files and directories returned can be selected using filters (see *RecordFilenameFilter* and *RecordFileFilter* in chapter "API overview").

The *list()* and *listFiles()* methods can provide an application with information on file names that do not belong to their actual area. These methods are therefore monitored by an active *Security Manager*. Consequently, they are allowed only if the application has read permission for the corresponding directory (see section "Security").

The static method *listRoots()* returns a list of all file system prefixes (“roots”) for the specified file system. It is guaranteed that the canonical path name of a file that actually exists physically begins with one of the prefixes returned by *listRoots()*.

The *listRoots()* method provides an application with information on the structure of the file system. If a *Security Manager* is active, only the roots for which read permission has been granted are shown. If you want the application to be able to determine all roots, you must grant the application read permission for <<*ALL FILES*>> (see section "Security").

Special features of the DMS file system

A list of all accessible catalog IDs is returned.

AccessParameter

The *AccessParameter* class and the classes derived from it define all parameters required to access record-oriented files (and which are supported) and contain at least the access method used and the associated file system, plus the default parameters record format and record length.

The access-method-specific implementations of this class can define additional parameters and then offer methods for setting and inquiring the values of these parameters. The subsections below describe the general methods which every implementation must provide, as well as the specific methods for the access methods currently supported.

Objects of the access-method-specific implementations of the *AccessParameter* class can be used to create new files in the corresponding file system using the relevant access method. Internally such objects are also used for other accesses to files (for example to open files). They can only be used to generate files in the file system from which they originate.

Objects of this abstract class cannot be generated by the user. However, the *recordFile* class provides the *getAccessParameter()* and *getDefaultAccessParameter()* methods which you can use to have objects of the access-method-specific implementations of this class returned.

Inadmissible values in the individual parameters are generally not discovered when the values are entered in the *Parameter* object, but only when this object is used.

General parameter methods

The *getFileSystem()* method returns the name of the associated file system.

The *getAccessMethod()* method returns the name of the access method to which this *Parameter* object belongs.

In addition, each implementation must provide the *getRecordFormat()*, *setRecordFormat()*, *getRecordLength()* and *setRecordLength()* methods. These are not dealt with here because the specific details are described in the following sections.

The constants *RECORD_FORMAT_UNKNOWN*, *RECORD_FORMAT_FIXED* and *RECORD_FORMAT_VARIABLE* are used as arguments when calling the *setRecordFormat()* method. Their meanings as used in their specific access methods are explained below.

The constants *NO_WAIT, THREAD_WAIT* and *APPLICATION_WAIT* are used as arguments in *shared update processing* when calling the method *setWaitMode()*. Their meanings as used in their specific access methods are explained below.

Parameters for SAM in DMS

The *AccessParameterSAM* class in the *com.fujitsu.ts.jrio.DMS* package provides a raft of additional methods for setting and inquiring further parameters which are specific to this access method.

Objects of this abstract class cannot be generated by the user. However, the *RecordFile* class provides the *getAccessParameter()* and *getDefaultAccessParameter()* methods via which the user can receive objects of this class's implementation.

Inadmissible values in the individual parameters are generally not discovered when the values are entered in the *Parameter* object, but only when this object is used.

The *getRecordFormat()* method returns the record format stored in this parameter object. The *setRecordFormat()* method sets the record format in this *Parameter* object. *RECORD_FORMAT_FIXED* and *RECORD_FORMAT_VARIABLE* can be specified when SAM is used. This parameter corresponds to the *RECFORM* specification in DMS.

The *getRecordLength()* method returns the record length stored in this parameter object. The *setRecordLength()* method sets the record length in this *Parameter* object. This parameter corresponds to the *RECSIZE* specification in DMS. In conjunction with fixed record format, this parameter defines the exact length of each record in a file. With variable record format it defines the maximum length of each record. The length specification 0 is then permitted and means unlimited record length. The DMS/SAM-specific restrictions and dependencies on other parameters (record format, block length) naturally apply as much for JRIO as at other DMS interfaces.

The *getBlockSize()* method returns the block length stored in this parameter. The *setBlockSize()* method sets the logical block length (as a number of PAM blocks) in this parameter object. This parameter corresponds to the `BLKSIZE=(STD,n)` specification in DMS. The dependencies on the record length naturally apply as much for JRIO as at other DMS interfaces.

The *getBlockControl()* method returns the block format stored in this parameter. This parameter corresponds to the *BLKCTRL* specification in DMS. The *setBlockControl()* method sets the block format in this parameter object. *BLOCK_CONTROL_BY_PUBSET*, *BLOCK_CONTROL_DATA*, *BLOCK_CONTROL_NO*, *BLOCK_CONTROL_PAMKEY*, *BLOCK_CONTROL_DATA_2K* and *BLOCK_CONTROL_DATA_4K* can be specified. This parameter is only of significance when new files are generated.

The *getPrimarySpaceAllocation()* method returns the value stored in this parameter for the primary space allocation in a file. The *setPrimarySpaceAllocation()* method sets the value for the primary space allocation of a file in this parameter object. This parameter corresponds to the first part of the *SPACE* specification in DMS.

The *getSecondarySpaceAllocation()* method returns the value stored in this parameter for the secondary space allocation in a file. The *setSecondarySpaceAllocation()* method sets the value for the secondary space allocation in a file in this *Parameter* object. This parameter corresponds to the second part of the *SPACE* specification in DMS.

The SAM access method enables a file to be opened simultaneous for read-only access by multiple applications. For this reason, s*hared update processing* is not possible for SAM files.

Parameter method for ISAM in DMS

The *AccessParameterISAM* class in the *com.fujitsu.ts.jrio.DMS* package provides a raft of additional methods for setting and inquiring further parameters which are specific to this access method.

Objects of this abstract class cannot be generated by the user. However, the *RecordFile* class provides the *getAccessParameter()* and *getDefaultAccessParameter()* via which the user can receive objects of this class's implementation.

Inadmissible values in the individual parameters are generally not discovered when the values are entered in the parameter object, but only when this object is used.

The *getRecordFormat()* method returns the record format stored in this parameter object. The *setRecordFormat()* method sets the record format in this *Parameter* object. *RECORD_FORMAT_FIXED* and *RECORD_FORMAT_VARIABLE* can be specified when ISAM is used. This parameter corresponds to the *RECFORM* specification in DMS.

The *getRecordLength()* method returns the record length stored in this parameter object. The *setRecordLength()* method sets the record length in this parameter object. This parameter corresponds to the *RECSIZE* specification in DMS. In conjunction with fixed record format, this parameter defines the exact length of each record in a file. With variable record format it defines the maximum length of each record. The length specification 0 is then permitted and means unlimited record length. The DMS/SAM-specific restrictions and dependencies on other parameters (record format, block length) naturally apply as much for JRIO as at other DMS interfaces.

The *getBlockSize()* method returns the block length stored in this parameter object. The *setBlockSize()* method sets the logical block length (as a number of PAM blocks) in this parameter object. This parameter corresponds to the BLKSIZE=(STD,n) specification in DMS. The dependencies on the record length naturally apply as much for JRIO as at other DMS interfaces.

The *getBlockControl()* method returns the block format stored in this parameter object. This parameter corresponds to the BLKCTRL specification in DMS. The *setBlockControl()* method sets the block format in this parameter object. *BLOCK_CONTROL_BY_PUBSET, BLOCK_CONTROL_DATA, BLOCK_CONTROL_NO, BLOCK_CONTROL_PAMKEY, BLOCK_CONTROL_DATA_2K* and *BLOCK_CONTROL_DATA_4K* can be specified. This parameter is only of significance when new files are generated.

The *getSharedUpdate()* method returns *true* or *false* depending on whether simultaneous processing of a file by multiple applications (s*hared update processing*) is permitted (or is to be permitted) or is prohibited (or is to be prohibited) with the parameter object. The *setSharedUpdate()* method specifies whether *shared update processing* for a file is to be allowed (*setSharedUpdate(true)*) or not (*setSharedUpdate(false)*) with the parameter object. The parameter is relevant only when a file is opened. It corresponds to the *SHARUPD* specification in DMS.

The *getWaitMode()* method returns the setting stored in the parameter object to control the behavior of the application in the event of conflicts during *shared update processing* for a file opened with the parameter object.

The *setWaitMode()* method controls the behavior of the application in the event of conflicts during *shared update processing* for a file. The specifications *NO_WAIT*, *THREAD_WAIT* and *APPLICATION_WAIT* are possible. *NO_WAIT* causes the application not to wait for granting of the lock and causes a *RecordLockedException* to be triggered in the event of a lock.

THREAD_WAIT causes the thread to be placed in a wait state. After expiry of a (brief, internally specified) wait time, repeated attempts are made to receive a lock until this succeeds or the application is terminated. *APPLICATION_WAIT* causes the entire application to wait at the system interface for the granting of the lock. The wait time at the interface is limited by the operating system to approx. 12 hr. After this period and after expiry of a (brief, internally specified) wait time, the system call is repeatedly issued until the lock is received or the application is terminated. This parameter has no direct equivalent in DMS because the wait behavior with *ISAM shared update* can only be controlled by means of the *EXLST* mechanism.

The *getPrimarySpaceAllocation()* method returns the value stored in this parameter object for the primary space allocation in a file. The *setPrimarySpaceAllocation()* method sets the value for the primary space allocation of a file in this parameter object. This parameter corresponds to the first part of the *SPACE* specification in DMS.

The *getSecondarySpaceAllocation()* method returns the value stored in this parameter object for the secondary space allocation in a file. The *setSecondarySpaceAllocation()* method sets the value for the secondary space allocation of a file in this parameter object. This parameter corresponds to the second part of the *SPACE* specification in DMS.

The *getPrimaryKeyPosition()* method returns the value stored in this parameter object for the key position of an ISAM file. The *setPrimaryKeyPosition()* method sets the value for the key position of an ISAM file in this parameter object. This parameter corresponds to the *KEYPOS* specification in DMS but with the difference that the numbering of the positions in JRIO deviates from that of other DMS interfaces (see section "Record").

The *getPrimaryKeyLength()* method returns the value stored in this parameter for the key length of an ISAM file. The *setPrimaryKeyLength()* method sets the value for the key length of an ISAM file in this parameter object. This parameter corresponds to the *KEYLEN* specification in DMS.

The *getDuplicateKeyIndicator()* method returns the value stored in this parameter for permitting duplication of the same key values in an ISAM file. The *setDuplicateKeyIndicator()* method sets the value for permitting duplication of the same key values in an ISAM file in this parameter object. This parameter corresponds to the *DUPEKY* specification in DMS.

Parameter methods for UPAM in DMS

The *AccessParameterUPAM* class in the *com.fujitsu.ts.jrio.DMS* package provides a raft of additional methods for setting and inquiring further parameters which are specific to this access method.

Objects of this abstract class cannot be generated by the user. However, the *RecordFile* class provides the *getAccessParameter()* and *getDefaultAccessParameter()* via which the user can receive objects of this class's implementation.

Inadmissible values in the individual parameters are generally not discovered when the values are entered in the parameter object, but only when this object is used.

The *getRecordFormat()* method returns the record format stored in this parameter object. The *setRecordFormat()* method sets the record format in this *parameter* object. Only *RECORD_FORMAT_FIXED* is permitted in UPAM.

The *getRecordLength()* method returns the record length stored in this parameter object. The *setRecordLength()* method sets the record length in this parameter object. For UPAM, the record length is always identical to the logical block length in bytes. Thus only values which are multiples of 2048 are permitted.

The *getBlockControl()* method returns the block format stored in this parameter object. This parameter corresponds to the BLKCTRL specification in DMS. The *setBlockControl()* method sets the block format in this parameter object. *BLOCK_CONTROL_BY_PUBSET, BLOCK_CONTROL_DATA, BLOCK_CONTROL_NO, BLOCK_CONTROL_PAMKEY, BLOCK_CONTROL_DATA_2K* and *BLOCK_CONTROL_DATA_4K* can be specified. This parameter is only of significance when new files are generated.

The *getPrimarySpaceAllocation()* method returns the value stored in this parameter object for the primary space allocation in a file. The *setPrimarySpaceAllocation()* method sets the value for the primary space allocation of a file in this parameter object. This parameter corresponds to the first part of the *SPACE* specification in DMS.

The *getSecondarySpaceAllocation()* method returns the value stored in this parameter object for the secondary space allocation in a file. The *setSecondarySpaceAllocation()* method sets the value for the secondary space allocation of a file in this parameter object. This parameter corresponds to the second part of the *SPACE* specification in DMS.

The *getSharedUpdate()* method returns *true* or *false* depending on whether simultaneous processing of a file by multiple applications (s*hared update processing*) is permitted (or is to be permitted) or is prohibited (or is to be prohibited) with the parameter object. The *setSharedUpdate()* method specifies whether *shared update processing* for a file is to be allowed (*setSharedUpdate(true)*) or not (*setSharedUpdate(false)*) with the parameter object. The parameter is relevant only when a file is opened. It corresponds to the *SHARUPD* specification in DMS.

The *getWaitMode()* method returns the setting stored in the parameter object to control the behavior of the application in the event of conflicts during *shared update processing* for a file opened with the parameter object. The *setWaitMode()* method controls the behavior of the application in the event of conflicts during *shared update processing* for a file. The specifications *NO_WAIT*, *THREAD_WAIT* and *APPLICATION_WAIT* are possible. *NO_WAIT* causes the application not to wait for granting of the lock and causes a *RecordLockedException* to be triggered in the event of a lock.

THREAD_WAIT causes the thread to be placed in a wait state. After expiry of a (brief, internally specified) wait time, repeated attempts are made to receive a lock until this succeeds or the application is terminated. *APPLICATION_WAIT* causes the entire application to wait at the system interface for the granting of the lock. The wait time at the interface is limited by the operating system to approx. 12 hr. After this period and after expiry of a (brief, internally specified) wait time, the system call is repeatedly issued until the lock is received or the application is terminated. This parameter has no direct equivalent in DMS because the wait behavior with *UPAM shared update* can only be controlled by means of the *PAMTOUT* value.

Sequential data processing

Separate interface groups for input and output are available for the sequential processing of files or other media which contain data records. The structure, designation and functionality of these interfaces is based on the classes known from the normal package *java.io* for sequential input/output familiar from normal Java I/O.

InputRecordStream

The abstract class *InputRecordStream* is the base class for all implementations of classes which permit sequential reading of records. The JRIO API provides two implementations of this abstract class, the *FileInputRecordStream* class for sequential reading from a file, and the *ArrayInputRecordStream* class for sequential reading from an array of *Record* objects.

The abstract class specifies the implementation of methods for sequential reading and skipping of records and for closing the file, as well as a method group for elementary repositioning (mark/reset), but which need not necessarily be supported by implementations.

The methods of the abstract class are not described in more detail here, but explained with the individual implementations. The API documentation contains this description for users who wish to define their own implementations.

FileInputRecordStream

A *FileInputRecordStream* object represents a file that has been opened for sequential read access. The file is opened implicitly when the object is created (see section "Opening and closing a file").

The *FileInputRecordStream* class offers methods for reading and skipping records and for closing the file. The method group for positioning is present, but provides no functionality.

The file that is to be opened must already exist in the underlying file system. The *createNewFile()* method of the *RecordFile* class must be used to generate a file.

For a file opened for sequential read access a current file position is always defined at which the next read operation is performed. The current file position is defined by the number of the record in accordance with the order of the records in this file, the records of a file being numbered starting with 0. After the file has been opened the current file position is the start of the file.

Opening and closing a file

When a *FileInputRecordStream* object is constructed, the file specified as *RecordFile* object is opened in read mode with the specified access method or with the specified access parameters. The file must exist in the underlying file system and the access method must belong to this file system and must be permissible for this file. The user must possess the access rights which permit the file to be read. If a *Security Manager* is active and its restrictions do not allow the file to be read, an *exception* is triggered (see section "Security").

If access parameters are specified for opening the file, these are taken into account when the file is opened provided the file parameters do not have priority. After the file has been opened, the parameters are updated with the corresponding values of the opened file.

The *close()* method closes the file. Subsequently no I/O operations can be performed via this *FileInputRecordStream* object.

Special features of the DMS file system

> *Shared update processing* (see section "Shared update processing" and section "AccessParameter") of a *FileInputRecordStream* is possible with the ISAM and UPAM access methods. However, with UPAM only PAM files can be opened in *shared update processing*. Because the file is opened for reading only, all accesses are made without locks. As a result, no access conflicts can arise. However, it must be expected that another application changes the contents of the record in the meantime.

Methods for reading records

The *read* method is offered in two variants, one in which the record read is provided as a result in a newly generated *Record* object, and a second in which a *Record* object transferred by the calling program as an argument is filled with the data of the record read.

When a record buffer is created, it has precisely the same size as the data read. If the calling program provides the *Record* object, it must ensure that the record buffer is large enough to contain the data of the record to be read. If the specified record buffer is too small to contain all the data, an exception is triggered and no data is transferred.

The *read()* methods read the record at the current file position. The current file position is subsequently incremented by one, in other words the next record is automatically positioned on.

The *skip()* method enables the specified number of records in the file to be skipped. It may be the case that it is not possible to skip exactly the number of records specified (for example if there are no longer enough records in the file). The return value of *skip()* specifies the actual number of records that are skipped.

The *available()* method returns the minimum number of records that can be read without blocking. But even the result null, which is often returned if it is impossible or difficult to determine whether a read attempt leads to a wait state (of the thread), does not justify the assumption that the next call of *read()* or *skip()* will actually lead to such a wait state.

Methods for positioning

The *markSupported()* method provides information on whether marking or repositioning is supported for this file. As with the *java.io.FileInputStream* class, positioning is currently not supported for objects of this class, in other words this method always returns false.

The *mark()* method is present, but has no function.
Calling *reset()* results in an exception as this functionality is currently not supported.

ArrayInputRecordStream

An *ArrayInputRecordStream* object represents an array of *Record* objects opened for sequential read access. Opening takes place implicitly when an object is generated (see "Opening and closing"), but has no further meaning here as it would with files.

The *ArrayInputRecordStream* class offers methods for reading and skipping records. The method group for positioning is also supported in full.

Within the array from which is read a current read position is always defined at which the next read operation is performed. The current read position is defined by the number of the record in the array, the numbering of the records starting with zero. After opening, the current read position is zero.

Opening and closing

When an *ArrayInputRecordStream* object is constructed, the calling program provides the array with data records which are to be read later. This array is used directly and not copied, in other words any manipulations on this array or the records contained in it have a direct affect on the *ArrayInputRecordStream* object. With a second variant of the constructor the user can make part of an array with records (defined by offset and length) available for input.

The *close()* methods is present, but has no function for this class.

Methods for reading records

The *read* method is offered in two variants, one in which the record read is provided as a result in a newly generated *Record* object, and a second in which a *Record* object transferred by the calling program as an argument is filled with the data of the record read.

When a record buffer is created, it has precisely the same size as the data read. If the calling program provides the *Record* object, it must ensure that the record buffer is large enough to contain the data of the record to be read. If the specified record buffer is too small to contain all the data, an exception is triggered and no data is transferred.

The *read()* methods read the record at the current read position. The current read position is subsequently incremented by one, in other words the next record in the array is automatically positioned on.

The *skip()* method enables the specified number of records in the array to be skipped. It may be the case that it is not possible to skip exactly the number of records specified because the array no longer contains enough records. The return value of *skip()* specifies the actual number of records that are skipped.

The *available()* method returns the number of records which can still be read before the end of the array is reached. Reading from an array of records never leads to wait states.

Methods for positioning

The *markSupported()* method provides information on whether marking or repositioning is supported for this data stream. In this class this method always returns true.

The *mark()* method notes the current read position so as to be able to reposition to it later. The argument envisaged for *mark()* is ignored in this implementation and should always be specified as 0.

Calling *reset()* repositions the pointer to a read position previously noted with *mark()*.

OutputRecordStream

The abstract class *OutputRecordStream* is the base class for all implementations of classes which permit sequential writing of records. The JRIO API provides two implementations of this abstract class, the *FileOutputRecordStream* class for sequential writing to a file, and the *ArrayOutputRecordStream* class for sequential writing to an array of *Record* objects.

This abstract class specifies the implementation of methods for sequential writing of records and for closing the file.

The methods of the abstract class are not described in more detail here, but explained with the individual implementations. The API documentation contains this description for users who wish to define their own implementations.

FileOutputRecordStream

A *FileOutputRecordStream* object represents a file that has been opened for sequential write access. The file is opened implicitly when the object is created (see section “Opening and closing a file” below).

The *FileOutputRecordStream* class offers methods for writing records and for closing the file. In a file opened for sequential write access, records are always added at the end.

The file that is to be opened must already exist in the underlying file system. The *createNewFile()* method of the *RecordFile* class must be used to generate a file.

Opening and closing a file

When a *FileOutputRecordStream* object is constructed, the file specified as *RecordFile* object is opened in write mode with the specified access method or with the specified access parameters. Users can decide whether or not any content the file may have should be deleted when the file is opened.

The file must already exist in the underlying file system and the access method must belong to this file system and must be permissible for this file. The user must possess the access rights which permit writing. If a *Security Manager* is active and its restrictions mean that writing is not permitted for the file, an *exception* is triggered (see section "Security").

If access parameters are specified for opening the file, these are taken into account when the file is opened provided the file parameters do not have priority. After the file has been opened, the parameters are updated with the corresponding values of the opened file.

The *close()* method closes the file. Subsequently no I/O operations can be performed via this *FileOutputRecordStream* object.

Special features of the DMS file system

> *Shared update processing* (see section "Shared update processing" and section "AccessParameter") for a *FileOutputRecordStream* is only possible with the ISAM access method if an existing file is opened in order to add to it. Other applications cannot then also open the file as *FileOutputRecordStream*. You are generally advised not to use *shared update processing* in conjunction with *FileOutputRecordStream*. In exceptional cases, simultaneous opening as *FileInputRecordStream* may be useful.

Methods for writing records

The *write()* method writes a record after the last record in the file. A lock is implicitly requested when *shared update processing* is used. This can trigger a *RecordLockedException* or, depending on the option set with *setWaitMode()*, can cause the thread or the entire application to wait in the event of access conflicts. The lock is released after completion of the write operation.

The *flush()* method ensures that all the records written with *write()* are actually output into the file, even if the basic access method envisages buffering the outputs. An existing lock for this file is released if *shared update processing* is used.

ArrayOutputRecordStream

An *ArrayOutputRecordStream* object represents an array of *Record* objects opened for sequential write access. The array is created implicitly when it is opened (see "Opening and closing") and expands with the data written into it.

The *ArrayOutputRecordStream* class offers methods for reading records. When writing, records are always added at the end of the array. In addition, this class also offers methods to fetch the entire contents of the data stream, to delete the contents, or to inquire the size.

Opening and closing

When an *ArrayOutputRecordStream* object is constructed, an array is provided internally into which records are later to be written. When doing this, the calling program can specify how many records the array should initially receive. If it does not do this, a default size is assumed. However, if this size is not sufficient to accommodate the records, the array is automatically enlarged internally.

The *close()* method is present, but has no function.

Methods for writing records

The *write()* method adds a record after the last record in the array.

The *flush()* method is present, but has no function for this class.

Methods for access to the content of a data stream

The *size()* method returns the number of records in the array.

The *reset()* method enables the entire contents of the array to be deleted. The array itself is retained unchanged in size and is refilled when further *write()* calls are made.

The *toRecordArray()* method returns the entire current contents of the data stream as an array of *Record objects*. The array returned is, in contrast to the one used internally, of exactly the size required to contain the data. The individual records are not copied here, which means that manipulation of the record contents has an effect on the content of the data stream.

The *writeTo()* method writes the entire current contents of the data stream into another specified data stream. Every data stream whose implementation is derived from the abstract class *OutputRecordStream* is suitable for this.

RandomAccessRecordFile

A *RandomAccessRecordFile* object represents a file opened for random access. The file is opened implicitly when the object is generated (see section "Opening and closing a file" below).

The *RandomAccessRecordFile* class offers methods for reading and writing records and for shortening and extending this file. There are also methods for positioning and for closing the file.

The file that is to be opened must already exist in the underlying file system. The *createNewFile()* method of the *RecordFile* class must be used to generate a file.

For a file that has been opened for random access a current file position is always defined at which the next read or write operation takes place. The current file position is defined by the number of the record in accordance with the sequence of records in this file, the records being numbered starting with zero. The current file position after the file is opened is the start of file.

When a file is opened for random access, the specific access direction can be restricted and deletion of the contents of an existing file can be requested.

The following open modes are permitted with this class:

- **INPUT**
  After the file has been opened only read operations are permitted.
- **OUTIN**
  After the file has been opened both write and read operations are permitted. The entire file contents are deleted when the file is opened.
- **INOUT**
  After the file has been opened both read and write operations are permitted. The file contents remain unchanged when the file is opened.

After the file has been closed the *RandomAccessRecordFile* object should no longer be used.

Opening and closing a file

When a *RandomAccessRecordFile* object is constructed, the file specified as *RecordFile* object is opened in the specified mode with the specified access method or with the specified access parameters.

The file must already exist in the underlying file system and the access method must belong to this file system and must be permissible for this file. The user must possess the necessary access rights to the file for the specified open mode. If a *Security Manager* is active and its restrictions for this file conflict with the specified open mode, an *exception* is triggered (see section "Security").

If access parameters are specified for opening the file, these are taken into account when the file is opened provided the file parameters do not have priority. After the file has been opened, the parameters are updated with the corresponding values of the opened file.

The *close()* method closes the file. Subsequently no I/O operations can be performed via this *RandomAccessRecordFile* object.

Special features of the DMS file system

Shared update processing (see section "Shared update processing" and section "AccessParameter") for a *RandomAccessRecordFile* is possible with the UPAM access method only for PAM files in the *INPUT* and *INOUT* open modes. If the file was opened in *INPUT* open mode, all accesses are made without locks. Consequently, no access conflicts can arise. However, it must be expected that another application changes the contents of the record in the meantime. In the *INOUT* open mode, read and write accesses are made with an implicit lock. In the event of access conflicts the option set using *setWaitMode()* can trigger a *RecordLockedException* or cause the thread or the entire application to wait. Locks are implicitly released when the locked record is written but can also be explicitly released using *flush()*. Details are provided in the appropriate interface description in the shipped JAVADOC documentation.

Methods for reading records

The *read* method is offered in two variants, one in which the record read is provided as a result in a newly generated *Record* object, and a second in which a *Record* object transferred by the calling program as an argument is filled with the data of the record read.

When a record buffer is created, it has precisely the same size as the data read. If the calling program provides the *Record* object, it must ensure that the record buffer is large enough to contain the data of the record to be read. If the specified record buffer is too small to contain all the data, an exception is triggered and no data is transferred.

The *read()* methods read the record at the current file position. The current file position is subsequently incremented by one, in other words the next record in the array is automatically positioned on.

Methods for writing records

The *write()* method writes a record into the file at the current file position. Any existing record is overwritten, but only if the restrictions applicable for the access method (for example same record length) are complied with. If the current file position is the end of file (or after this), the file is extended. After writing, the current file position is the record after the written record or the end of file.

In *shared update processing*, an existing record can only be changed safely (when there are competing applications) if the record lock implicitly set when reading is not released between reading and writing - in particular, no other record must be read or written in the meantime. You should therefore follow a corresponding sequence of actions in *shared update processing*; however, no check of this sequence is made.

The *flush()* method ensures that all records written with *write()* are output to the file even if the underlying access method provides buffering. *Shared update processing* also ensures that a lock received for a file by the application is released.

Special features of the DMS file system

When *shared update processing* is used in the DMS file system, information on the current end-of-file cannot be synchronized between participating applications. Simultaneous extension of *RandomAccessRecordFiles* by multiple applications is not therefore recommended.

Methods for positioning and changing size

The *getCurrentRecordNumber()* method returns the current file position as a record number.

The *setCurrentRecordNumber()* method sets the current position of the file to the record with the specified number. The special constants *POS_FIRST* and *POS_LAST* can be used to position to the start or end of file.

The *getRecordCount()* method returns the number of records in the file. That is simultaneously the position of the end of file.

The *setRecordCount()* method modifies the size of the file to the number of records specified. If the specified number of records is less than the current number of records in the file, the file is shortened so that it only contains as many records as specified. If in this case the current file position was greater than the new file size, the current file position is set to the new end of file. If the specified number of records is greater than the current number of records in the file, the file can be extended. An access method can reject such a file extension, for example for files with variable record format. When the operation has been executed, the content of the newly added records is undefined.

Special features of the DMS file system

When *shared update processing* is used in the DMS file system, it is not possible to reduce the size of a file. This would trigger an *IOException*.

When *shared update processing* is used in the DMS file system, information on the current end-of-file cannot be synchronized between participating applications. Simultaneous extension of *RandomAccessRecordFiles* by several applications is not therefore recommended. Nevertheless, the locks are set as if the file were being extended by writing individual records on after the other.

Indexed-sequential data processing

Keys play a very central role in indexed-sequential data processing.

Keys define the order of the records within an indexed-sequential file. A key is always part of a record and is defined by the key field (position and length) within each record of an indexed-sequential file. The content of a key field is the key value. In addition, a key can have a name if, for example, this is necessary to distinguish different keys in an implementation.

A distinction is made between primary and secondary keys. Each indexed-sequential file always has precisely one primary key and can have one or more secondary keys. Secondary keys must always have a unique name. If a file has several keys, each of these keys may define a different order.

Identical key values in different records are permitted for a key.

Special features of the BS2000 access method ISAM

For ISAM files, an unnamed primary key is always defined. Only for NK-ISAM files can several secondary keys (up to 30) be defined in addition. The secondary keys must always have a unique name (up to 8 characters). The restrictions which apply for ISAM (for example regarding key length) must naturally also be taken into account when the JRIO interfaces are used. Identical secondary keys in different records are only permissible if no identical key values are permitted in different records for the primary key and if and if identical key values in different records have already been permitted for all other secondary keys.

The marking options (value flag and logical marking) which ISAM offers are not supported by JRIO.

Note that at the ISAM DMS interfaces, positions within a record, in particular key positions, can be numbered differently than at the JRIO interfaces (see section "Record").

KeyDescriptor

The *KeyDescriptor* class defines the position, length and other attributes of a particular key field within a record of an indexed-sequential file (key definition). It provides methods for accessing these key attributes of an indexed-sequential file.

A *KeyDescriptor* object is used for generating or extracting a concrete key value. Appropriate implementations of this abstract class are provided for ISAM. You can thus generate such *KeyDescriptor* objects themselves or have them provided via the methods of the *KeyedAccessRecordFile* class.

If you are working on an ISAM file with key definitions they have generated themselves, you must naturally ensure that these fit the keys defined in the file.

A *KeyDescriptor* object is serializable and can thus be used for Remote Method Interfaces (RMIs).

Methods

The *getPosition()* method returns the position of the key field in a record.

The *getLength()* method returns the length of the key field.
The *getName()* method returns the name of a named key, or null for unnamed keys. Thus with secondary keys the unique name is always returned. In the case of the primary key, whether or not a name is returned depends on the implementation.

The *hasDuplicates()* method is used to check whether identical key values are permitted in different records for the key concerned.

Whether the key is a primary or secondary key is checked using the *isPrimary()* or *isSecondary()* method.

PrimaryKeyDescriptorISAM

The *PrimaryKeyDescriptorISAM* class in the *com.fujitsu.ts.jrio.DMS* package is an implementation of the abstract class *KeyDescriptor* and represents the primary key of an ISAM file. The class offers only those methods which the abstract class specifies, as well as constructors for generating the key definitions. The following particular features apply for ISAM:

- The key position must be a value between 0 and 32767. However, this does not mean that these values always make sense. The values actually used for I/O depend on other factors (block size, record format, key length), but these cannot be checked by the constructor.
- The length of the key must be a value between 1 and 255.
- The primary ISAM key does not have a name, and the *getName()* method therefore always returns null.

SecondaryKeyDescriptorISAM

The *SecondaryKeyDescriptorISAM* class in the *com.fujitsu.ts.jrio.DMS* package is an implementation of the abstract class *KeyDescriptor* and represents a secondary key of an ISAM file. The class offers only those methods which the abstract class specifies, as well as constructors for generating the key definitions. The following particular features apply for ISAM:

- The key position must be a value between 0 and 32767. However, this does not mean that these values always make sense. The values actually used for I/O depend on other factors (block size, record format, key length), but these cannot be checked by the constructor.
- The length of the key must be a value between 1 and 127.

- A secondary ISAM key must have a unique name up to 8 characters in length which complies with the DMS rules. Upper/lower case is ignored in these names, and a name is always returned in upper case by *getName()*.

KeyValue

The *KeyValue* class defines an actual key value. Every key value has a key definition associated with it. This class provides methods for manipulating the key value and for inquiring the attributes of the associated key description.

A *KeyValue* object can be used to select a record in an indexed-sequential file using this key.

A *KeyValue* object is serializable can thus be used for Remote Method Interfaces (RMIs).

Constructors

When a *KeyValue* object is generated, the key value is filled with the user's data. This data can be specified as a byte array or string. If the user specifies no data or the data specified is shorter than the key, the complete key value is padded with null bytes or blanks. If the data is longer than the key, only as much data is transferred as will fit in the key.

If the user specifies the data as a string but specifies no encoding for converting text to data, the system-dependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

Methods for manipulating the key value

The *setValue()* methods fill the key with the specified user data. If the user specifies no data, the entire key value is filled with null bytes. If the data is shorter than the key, the rest is filled with a filler byte. The filler byte can be supplied by the user, otherwise a null byte is used. If the data is longer than the key, only as much data is transferred as will fit in the key.

The *setStringValue()* methods fill the key with the converted data of the specified string. If the user specifies no data, the entire key value is filled with blanks. If the data is shorter than the key, the rest is filled with blanks. If the data is longer than the key, only as much data is transferred as will fit in the key.

If no encoding for converting text to data was specified by the user, the system-dependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

The *getValue()* methods are used to return the key value of a key. The key value is either transferred to a buffer provided by the user or returned as a copy of the value. As the key value is therefore always copied, this means that manipulations on the result returned have no influence on the object from which the value originates. If the value in the object is to be modified, the setValue() method must subsequently be used.

With the *getStringValue()* methods, the key value of a key is returned converted into a string. If no encoding for converting text to data was specified by the user, the system-dependent standard encoding (in BS2000 the default value is *OSD_EBCDIC_DF04_1*) is used.

Methods for determining the key attributes

The *getPosition()* method returns the position of the key field in a record.

The *getLength()* method returns the length of the key field.
The *getKeyDescriptor()* method returns the key definition associated with the key value.

KeyedAccessRecordFile

A *KeyedAccessRecordFile* object represents a file opened for keyed access. The file is opened implicitly when the object is generated (see section "Opening and closing a file").

The *KeyedAccessRecordFile* class offers methods for reading, writing and deleting records in this file. There are also methods for handling keys and for closing the file.

The file that is to be opened must already exist in the underlying file system. The *createNewFile()* method of the *RecordFile* class must be used to generate a file.

When a file is opened for keyed access, the specific access direction can be restricted and deletion of the contents of an existing file can be requested.

The following open modes are permitted with this class:

- **INPUT**
  After the file has been opened only read operations are permitted.
- **OUTIN**
  After the file has been opened both write and read operations are permitted. The entire file contents are deleted when the file is opened.
- **INOUT**
  After the file has been opened both read and write operations are permitted. The file contents remain unchanged when the file is opened.

After the file has been closed the *KeyedAccessRecordFile* object should no longer be used.

Opening and closing a file

When a *KeyedAccessRecordFile* object is constructed, the file specified as *RecordFile* object is opened in the specified mode with the specified access method or with the specified access parameters.

The file must already exist in the underlying file system and the access method must belong to this file system and must be permissible for this file. The user must possess the necessary access rights to the file for the specified open mode. gegebenen Open-Modus erforderlichen Zugriffsrechte auf die Datei besitzen. If a *Security Manager* is active and its restrictions for this file conflict with the specified open mode, an *exception* is triggered (see section "Security").

If access parameters are specified for opening the file, these are taken into account when the file is opened provided the file parameters do not have priority. After the file has been opened, the parameters are updated with the corresponding values of the opened file.

The *close()* method closes the indexed-sequential file. Subsequently no I/O operations can be performed via this *KeyedAccessRecordFile* object.

Special features of the DMS file system

Shared update processing (see section "Shared update processing" and section "AccessParameter") of a *KeyedAccessRecordFile* is possible for all open modes (*INPUT*, *INOUT*, *OUTIN*). However, *OUTIN* open mode is permitted only for the application that opened the file first. If the file was opened in *INPUT* open mode, all accesses are made without locks. As a result, no access conflicts can arise. However, it must be expected that another application changes the contents of the record in the meantime. With the other open modes, read and write accesses are made with implicit locks. In the event of access conflicts, this can trigger a RecordLockedException or cause the thread or the entire application to wait, depending on the option set using *setWaitMode()*. Locks are released after writing or deleting the locked record. They can also be released explicitly using *unlock()*. When a record is read, an existing lock for another record is also released. Details are provided in the appropriate interface description in the shipped JAVADOC documentation.

Methods for reading records

All *read* methods are offered in two variants, one in which the record read is provided as a result in a newly generated *Record* object, and a second in which a *Record* object transferred by the calling program as an argument is filled with the data of the record read.

When a record buffer is created, it has precisely the same size as the data read. If the calling program provides the *Record* object, it must ensure that the record buffer is large enough to contain the data of the record to be read. If the specified record buffer is too small to contain all the data, an exception is triggered and no data is transferred.

With the *read* methods in which a *KeyValue* or *KeyDescriptor* object can be specified, such arguments are only accepted if they are suitable for the file (see the *getPrimaryKeydescriptor()* and *getSecondaryKeydescriptor()* methods).

The *read()* methods read the record which is selected by the specified key value. If there is more than one record with the same key value in the file, the first one is returned. Both a value of the primary key and a value of the secondary key can be specified as the key value.

The *readNext()* methods read the next record in the order determined by the given argument. There are three variants of these methods:

- If no order argument is specified, the next record defined by the order of the primary key is read. If this method is called as the first operation after a file has been opened, the record with the lowest available primary key value is read. In all other cases this operation reads the record following the last record read, provided the last record read was also read via the primary key (in other cases the behavior is access-method-specific). This is a method to permit sequential reading of records which contain the same key value.
- If a key definition is specified as an order argument, the next record defined by the order of the primary or secondary key of the given key definition is read. If this method is called immediately after a file has been opened, the record with the lowest available key value as defined in the given key definition is read. In all other cases this operation reads the record following the last record read, provided the last record read was read via the same key definition (in other cases the behavior is access-method-specific). This is a method to permit sequential reading of records which contain the same key value.
- If a key value is specified as an order argument, the record is read with the next highest key in accordance with the order of the associated key definition.

The *readPrevious()* methods read like the *readNext() methods*, but they read the preceding record rather than the following record.

Methods for writing and deleting records

When records are written to an indexed-sequential file, the position of a written record is determined by the key fields contained in the record.

The *write()* method writes a record to a file. If a file with the same primary key value already exists and no duplicate keys are permitted for the primary key, the existing record is replaced. If duplicate keys are permitted and the record already exists, the record is added after the last record with the same primary key value.

The *writeNew()* method writes a record to the file, but only if no record with the same primary key exists in the file.

The *writeBack()* method overwrites a record in the file that was read directly beforehand. Between the read and write operations, no modification may be made to the record's primary key field. In *shared update processing* an existing record is not overwritten unless the lock set in order to read the record still applies. Otherwise, a *RecordNotLockedException* is triggered.

The *delete()* method deletes the record selected by the specified key value. If there are several records with the same key value in the file, the first one is deleted. Either a value of the primary key or a value of a secondary key can be specified as the key value.

Methods for unconditional lock release

The *unlock()* method is used to explicitly release a lock set implicitly by a read operation in *shared update processing*.

Methods for determining key definitions

The *getPrimaryKeyDescriptor()* method returns the key definition for the primary key of this file.

The *getSecondaryKeyDescriptor()* method returns the key definition for the secondary key with the specified name.

The *getKeyDescriptorNames()* method returns a list of the names of all of this file's secondary keys.

Methods for generating and deleting secondary keys

The *createSecondaryKey()* methods generate a new secondary key for this indexedsequential file with the specified parameters. There are two parameter variants. One variant is that all fields of the *KeyDescriptor* object (name, key position, key length and the specification as to whether identical key values are permitted in different records for this key) are specified individually, and the second is that a *KeyDescriptor* object is specified for a secondary key. The second variant enables, for example, the attributes of another file's secondary key to be used in order to generate a corresponding secondary key in this file.

The *deleteSecondaryKey()* method deletes the specified secondary key of this indexedsequential file.

The *createSecondaryKey()* and *deleteSecondaryKey()* methods require exclusive access to the file and are therefore not permitted in *shared update processing*. They would trigger an *IOException*.

Implementation details

The attributes marked as implementation-specific in the API descriptions are defined in this section.

File-system-specific definitions

Both in the API descriptions in this document and in the actual API specifications it is specified in various places that a file system implementation can specify particular definitions.

These definitions are shown in the table below for the file systems supported in this version. The UFS file system is included merely to complete the picture, although it is currently not supported.

Detail	**DMS file system**	**UFS file system**
Name to be used at the JRIO interfaces	“DMS”	“UFS”
Access methods	ISAM, SAM, UPAM	Currently none
File system prefixes	Catalog IDs ("*:catid:*")	Root directory '/'
Normalization	Lower-case letters are converted to upper-case letters and path names *$<name>* to *$.<name>*	. and .. directories are cancelled and double slashes '//' are converted into single slashes; a '/' at the end of the path name is deleted
Absolute path name	Supplementing the path name with the catalog ID	Supplementing the current directory for relative path names
Canonical path name	Either only catalog ID or the file name supplemented by catalog ID and user ID, if required with cancellation of the standard system ID	Conversion like absolute path name and resolution of all symbolic links
Empty path name	Standard catalog ID of the user	Root directory '/'
Normal file	All files are normal files	Regular files (for example no special files)
Hidden files and directories	Temporary files in the DMS sense	All files and directories whose name begins with a period '.'
Size of a file with the *length()* method	Number of PAM pages used * 2048 (last page pointer)	Size in bytes
Size of a directory with the *length() method*	Always 0	Size in bytes
File name	See the manual “Introductory Guide to DMS” [8]	See manual “POSIX, Basics for Users and Systems Administrators” [1]
Separator between path name parts *separatorChar* and *separator*	Not defined	Slash '/' or "/"

Separator between path names *pathSeparatorChar* and *pathSeparator*	Comma ',' or ","	Colon ':' or ":"
Default directory when creating a temporary file with the *createTempFile()* method	Default catalog ID of the calling program	Default directory which is assigned to the system property *java.io.tmpdir*
Generated name part of a temporary file (between suffix and prefix specifications)	String with the length 7	String with the length 7
Shared update processing	Supported (with restrictions)	Not supported

Table 4: File-system-specific definitions

Access-method-specific definitions

Both in the API descriptions in this document and in the actual API specifications it is specified in various places that an access method implementation can specify particular definitions.

These definitions are shown in the table below for the DMS access methods supported in this version.

Details	**SAM access method**	**ISAM access method**	**UPAM access method**
Name to be used at the JRIO interfaces	“SAM”	“ISAM”	“UPAM”
Permissible record formats	Record format with variable and fixed record length	Record format with variable and fixed record length	Record format with fixed record length
Maximum record length (depending on the record format (fixed, variable), logical block format (NO, KEY, DATA) and block size (1 <= BS <= 16) - the *setRecordLength* method of the *AccessParameter*... classes also accepts greater values because block size or record format can be modified later)	fixed, KEY BS * 2048 fixed, NO, DATA BS * 2048 - 16 variable, KEY: BS * 2048 -4 variable, NO, DATA BS * 2048 - 20	fixed: BS * 2048 variable: BS * 2048 - 4 In the event of full utilization overflow blocks may occur	fixed, NO, KEY BS * 2048 fixed, DATABS * 2048 (the first 12 bytes contain metadata!)
Permissible values for *setRecordLength*() of the *AccessParameter*... classes	0 through 32768 (0 means: variable, restricted only by block size)	0 through 32768 (0 means: variable, restricted only by block size)	0 through 32768 (0 means: pubset standard) Values != n*2048 (n=0,, 16) are not permitted
Permissible values for the *setBlocksize*, *setPrimarySpaceAllocation*, *setSecondarySpaceAllocation*, *setPrimary-KeyPosition*, *setSecondaryKeyPosition* methods of the *AccessParameter* ... classes	See API documentation on the *AccessParameter SAM interface*	See API documentation on the *AccessParameter ISAM interface*	See API documentation on the *AccessParameter U PAM interface*
markSupported() methodFor marking and repositioning in the event of sequential reading of the *FileInputRecordStream* class	Always false	Always false	Always false

Writing buffered output to the output stream with the *flush()* method of the *FileOutputRecordStream* class	The write buffer is emptied	The write buffer is emptied	No function
Permissible values when generating secondary keys with the *createSecondaryKey()* method of the *KeyedAccessRecordFile* class	Not supported	Yes Max. 30 secondary keys, each max. 127 bytes long. keyPos <= 32495	Not supported
Name of secondary keys (*createSecondaryKey()* method of the *KeyedAccessRecordFile* class)	Not supported	8-character, as per DMS rules, lower-case letters may be converted to upper-case letters	Not supported
Setting the file position with the *setCurrentRecordNumber()* method of the *RandomAccessRecordFile* class after the last record (value of *getRecordCount()*) - or writing at such a position	Yes - empty records (with variable record format) or records with undefined contents (with fixed record format) may be added	No	Yes - records with undefined contents may be added
Overwriting records with the *write()* method	Same record length at records with variable lenght		Possible without restrictions
Overwriting records with the *writeBack()* method	-	The primary key may not be modified	
Sequence in the event of sequential reading with the *KeyedAccessRecordFile* class	-	Write or delete operations modify the file position and should therefore not be used between sequential read operations. With regard to different keys, ISAM behavior applies for the sequence in sequential read operations (see themanual “Introductory Guide to DMS” [8])	
Shared update processing: general	Not possible	Possible as: *FileInputStream*, *FileOutputStream* *KeyedAccessRecordFile*	Only possible for PAM files as *FileInputStream* or as *RandomAccessRecordFile*

Shared update processing: open modes	-	*INPUT*, *INOUT* or *OUTIN* permitted, *FileOutputStream* only to add to a file, *OUTIN* only for the first application that opens the file	*INPUT* or *INOUT* permitted
Shared update processing: lock granularity	-	With *NK-ISAM*, the lock is on key level (primary key), with*K-ISAM* the lock is on block level	Lock is on block level
Shared update processing:other special features	-	Locks apply for the entire application (not only for a file)	It is not possible to increase or decrease the size of a file

Table 5: Access-method-specific definitions

Default values of the DMS access methods

The table below provides an overview of the default values for the access methods in the DMS file system of an *AccessParameter* object that was generated with the *getDefaultAccessParameter()* method. The overview is structured according to the methods used for reading.

Method	**SAM access method**	**ISAM access method**	**UPAM access method**
getAccessMethod()	“SAM”	“ISAM”	“UPAM”
getFileSystem()	“DMS”	“DMS”	“DMS”
getRecordFormat()	RECORD_ FORMAT_ VARIABLE	RECORD_ FORMAT_ VARIABLE	RECORD_ FORMAT_ FIXED
getRecordLength()	0	0	0
getBlockSize()	0	0	-
getDuplicateKeyIndicator()	-	false	-
getPrimaryKeyLength()	-	8	-
getPrimaryKeyPosition()	-	0	-
getPrimarySpaceAllocation()	0	0	0
getSecondarySpaceAllocation()	-1	-1	-1
getBlockControl()	BLOCK-CONTROL_ BY_PUBSET	BLOCK-CONTROL_ BY_PUBSET	BLOCK-CONTROL_ BY_PUBSET
getSharedUpdate()	-	false	false
getWaitMode()	-	THREAD_WAIT	THREAD_WAIT

Table 6: Default values of the DMS access methods

The value 0 with *getRecordLength()* designates the value “variable - only limited by block size” for the SAM and ISAM access methods and the “pubset-specific default” for UPAM.

The values 0 with *getBlockSize()*, 0 with *getPrimarySpaceAllocation()* and -1 with *getSecondarySpaceAllocation()* designate the “pubset-specific default”. If the file has been created, the current values are entered here.

Restrictions

The following explicit restrictions are defined for DMS under JRIO:

- Tape files and private disks are not supported.
- EAM and logical system files are not supported.
- Not all file parameters can be manipulated or set via JRIO. Only the parameters explicitly named in the API descriptions are taken into account. Especially when new files are created this results in restrictions when particularly special attributes are to be used. However, the most common parameters of the various access methods are already supported with the *AccessParameter* class and the associated implementations of the access methods.
- Only the access methods shown and the files related to them are supported.
- Shared update and locking are not supported in this version.
- Reverse reading is supported only for keyed access, not for sequential or random access.
- In ISAM the logical value flag is not supported. ISAM files which contain such a flag cannot be processed. ISAM pools are also not supported.
- The undefined record format is not supported. Files with undefined record format cannot be processed.
- File generation groups are not supported.

Examples

The examples in the sections below are designed to show the various access types and general use of the JRIO interfaces on the basis of one or two (more or less typical) problems.

All the examples given here consist of complete programs which can be executed. The source texts of all sample programs are supplied with the product and are contained in the subdirectory *demo/jrio* of the installation directory. In conjunction with your inline documentation, the programs should be largely self-explanatory.

Sequential data processing

A simple copy program for SAM files is used to demonstrate sequential data processing. The program requires two parameters: the name of the file which is to be copied and the file name of the copy. If the target file already exists, it is deleted (take care!) and created again.

The example is so designed that no knowledge of the file attributes, such as record format or record length, of the file to be copied is required. The error handling in the example is not particularly convenient, simply to prevent the comprehensive code which would be required for this distracting the reader from the way the interface is actually used.

The program is contained in the *CopySAM.java file*:

```
import java.io.*;
import com.fujitsu.ts.jrio.*;
/**
 * This sample program demonstrates the use of the
 * JRIO interfaces for file handling and sequential
 * input and output.
 *
 * The program creates a copy of a DMS file of type SAM
 * by sequentially copying each record of the file.
 *
 * The interesting part of this program is the method
 * doCopySAM(), all other methods are added to make it
 * a complete executable program.
 */
public class CopySAM
{
    /**
     * The main method, which analyses the program arguments,
     * calls the work method and provides global error
     * handling.
    */
   public static void main(String args[])
   {
       String source = null;
       String target = null;
       for (int i = 0; i < args.length; i++)
       {
           if (source == null)
               source = args[i];
           else if (target == null)
               target = args[i];
           else
               usage();
       }
       if (source == null || target == null)
           usage();
       try {
           doCopySAM(source,target);
       } catch (Exception e) {
           error(e.toString());
       }
   }
   /**
    * Print a usage message and exit with error
```

```
 */
private static void usage()
{
    error("Usage: CopySAM source target");
}
/**
 * Print the given error message and exit
 */
private static void error(String msg)
{
    System.err.println(msg);
    System.exit(1);
}
/**
* The work method.
* This method demonstrates, how the JRIO interfaces
* may be used to copy a complete SAM file by
* sequential read and write operations.
*
* @param   source
*          The name of the file to be copied
* @param   target
*          The name of the copied file
*/
public static void doCopySAM(String source,String target)
    throws IOException
{
    Record rec;
    RecordFile sourceFile;
    RecordFile targetFile;
    FileInputRecordStream input;
    FileOutputRecordStream output;
    /**
     * check file names and create RecordFile objects
     */
    sourceFile = new RecordFile(source,"DMS");
    targetFile = new RecordFile(target,"DMS");
    /**
     * check source file existence
     */
    if (!sourceFile.exists())
      error("Source file " + source + " does not exist");
    /**
     * check target file existence
     */
    if (targetFile.exists())
    {
        /**
        * delete the existing file
        */
        if (!targetFile.delete())
        error("Target file " + target
             + " could not be deleted");
}
/**
 * create an empty output file with same attributes
 */
if (!targetFile.createNewFile(
         sourceFile.getAccessParameter("SAM")))
```

```
    error("Target file " + target + " still exists");
   /**
    * open source for input
    */
   input = new FileInputRecordStream(sourceFile,"SAM");
   /**
    * open target for output
    */
   output =new FileOutputRecordStream(targetFile,"SAM");
   /**
    * read and write all records
    */
   while ((rec = input.read()) != null)
         output.write(rec);
    /**
     * close all files
    */
    input.close();
    output.close();
   }
}
```

The program can be compiled with the Java compiler javac and then be run. No particular specifications are required to make the JRIO interfaces available.

Random data processing

Two examples are used to demonstrate random data processing. The first program solves the problem of deleting one or more records from a SAM file. For this purpose the program expects as parameters the name of an existing file and the number or the number range of the records to be deleted. Note that here, too, the records are numbered consecutively starting with zero.

The example is so designed that no knowledge of the file attributes, such as record format or record length, of the file to be processed is required. The error handling in the example is not particularly convenient, simply to prevent the comprehensive code which would be required for this distracting the reader from the way the interface is actually used.

The program is contained in the *DeleteRecordsSAM.java* file:

```
import java.io.*;
import com.fujitsu.ts.jrio.*;
/**
 * This sample program demonstrates the use of the
 * JRIO interfaces for file handling and random
 * access to a file.
 *
 * This program deletes a sequence of specified records
 * from a DMS file of type SAM.
 *
 * The interesting part of this program is the method
 * doDeleteRecordsSAM(), all other methods are added
 * to make it a complete executable program.
 */
public class DeleteRecordsSAM
{
    /**
     * The main method, which analyses the program arguments,
     * calls the work method and provides global error
     * handling.
     */
    public static void main(String args[])
    {
        String file = null;
        String first = null;
        String last = null;
        int firstNum, lastNum;
        for (int i = 0; i < args.length; i++)
        {
            if (file == null)
                file = args[i];
            else if (first == null)
                first = args[i];
            else if (last == null)
                last = args[i];
            else
                usage();
        }
        if (file == null || first == null)
            usage();
        try {
            firstNum = Integer.parseInt(first);
            if (firstNum < 0)
```

```
                error("Illegal record number " + firstNum);
            if (last != null)
        {
            lastNum = Integer.parseInt(last);
            if (lastNum < 0 || lastNum < firstNum)
                error("Illegal record number " + lastNum);
        }
        else
            lastNum = firstNum;
        doDeleteRecordsSAM(file,firstNum,lastNum);
        } catch (Exception e) {
            error(e.toString());
        }
    }
    /**
     * Print a usage message and exit with error
     */
    private static void usage()
    {
        error("Usage: DeleteRecordsSAM file first [last]");
    }
    /**
     * Print the given error message and exit
     */
    private static void error(String msg)
    {
        System.err.println(msg);
        System.exit(1);
    }
    /**
     * The work method.
     * Delete all records between the given record
     * numbers in a SAM accessible file using
     * the random access classes of JRIO.
    *
    * @param   file
    *          The file to modify
    * @param   first
    *          The first record to delete
    * @param   last
    *          The last record to delete
    */
    public static void doDeleteRecordsSAM(
            String file,int first,int last)
        throws IOException
    {
        Record rec;
        RecordFile sourceFile;
        RandomAccessRecordFile update;
        ArrayOutputRecordStream buffer;
        Record[] remaining;
        /**
         * check file name and create RecordFile object
         */
        sourceFile = new RecordFile(file,"DMS");
        /**
         * check source file existence and write rights
         */
        if (!sourceFile.exists() || !sourceFile.canWrite())
```

```
            error("Source file " + file + " does not exist"
            + " or is not writeable");
    /**
     * open file for update
     */
    update = new RandomAccessRecordFile(sourceFile,"SAM",
            RandomAccessRecordFile.INOUT);
    /**
     * check record numbers
     */
    if (first >= update.getRecordCount())
    {
        /**
         * nothing todo
         */
        update.close();
        return;
    }
    /**
     * position to first record after delete area
     */
    update.setCurrentRecordNumber(last + 1);
    /**
     * read all remaining records into an array
     */
    buffer = new ArrayOutputRecordStream();
    while ((rec = update.read()) != null)
        buffer.write(rec);
    remaining = buffer.toRecordArray();
    /**
     * truncate file
     */
    update.setRecordCount(first);
    /**
     * append the buffered records to the truncated file
     */
    for (int i = 0; i < remaining.length; i++)
         update.write(remaining[i]);
    /**
     * close the file
     */
    update.close();
  }
}
```

The program can be compiled with the Java compiler javac and then be run. No particular specifications are required to make the JRIO interfaces available.

The second program outputs a randomly selected “slogan of the day” from a file (SAM) with slogans. For this purpose the program expects as a parameter the name of the file with the slogans. If this does not yet exist, it is created with a basic stock of slogans.

The example is also so designed that no knowledge of the file attributes, such as record format or record length, of the file to be processed is required. The error handling in the example is not particularly convenient, simply to prevent the comprehensive code which would be required for this distracting the reader from the way the interface is actually used.

The program is contained in the *SloganOfTheDay.java file*:

```
import com.fujitsu.ts.jrio.*;
import java.io.*;
import java.util.Random;
/**
 * This example demonstrates a random access to a SAM file.
 *
 * A randomly selected record (the slogan of the day) is read
 * from the file and written to the standard output stream.
 * If the file with the slogans does not yet exist, it is
 * created and filled with some standard slogans.
 *
 * The interesting part of this program is the method
 * doSloganOfTheDay(), all other methods are added to make it
 * a complete executable program.
 */
public class SloganOfTheDay
{
    /**
     * The main method, which analyses the program arguments,
     * calls the work method and provides global error
     * handling.
     */
    public static void main(String[] args)
    {
        if (args.length != 1)
            usage();
        try {
            doSloganOfTheDay(args[0]);
        } catch (Exception e) {
            error(e.toString());
        }
    }
    /**
     * Print a usage message and exit with error
     */
    private static void usage()
    {
        error("Usage: SloganOfTheDay file");
    }
    /**
     * Print the given error message and exit
     */
    private static void error(String msg)
    {
        System.err.println(msg);
        System.exit(1);
    }
    /**
     * The work method.
     * It demonstrates how the JRIO interfaces may be used
     * for random access to a file.
     *
     * @param filename
    *        the file containing the slogans
    */
    public static void doSloganOfTheDay(String filename)
```

```
    throws IOException
{
    /**
     * the random number generator, used to select the
     * slogan
     */
    Random generator = new Random();
/**
 * some slogans to be written to the slogan file
 * in case it is still empty.
 */
String[] data = {
    "Schuster bleib bei deinen Leisten.",
    "Es fuehren viele Wege nach Rom.",
    "In ungezaehlten Muehen waechst das Schoene.",
    "It's better to burn out, than to fade away.",
    "Make your ideas work!",
    "Erlaubt ist, was gefaellt.",
    "Der schoenste Morgen bringt uns das Gestern "
        + "nicht zurueck.",
    "Sage nicht immer, was du weisst, aber wisse "
        + "immer, was du sagst.",
    "Alles muss man selber machen - sogar das Lachen."
};
/**
 * Definition of the slogan file
 */
RecordFile rf = new RecordFile(filename, "DMS");
RandomAccessRecordFile slogfile = null;
/**
 * The record object used for accessing
 * the slogan file
 */
Record record = null;
/**
 * the number of records in the slogan file
 */
long numOfRecs = 0;
/**
 * Check if the slogan file is already existing
 */
if (!rf.exists())
{
    rf.createNewFile("SAM");
    slogfile = new RandomAccessRecordFile(rf, "SAM",
        RandomAccessRecordFile.OUTIN);
    for (int i = 0; i < data.length; i++)
    {
        record = new Record(data[i].length());
        record.setStringData(data[i]);
        slogfile.write(record);
    }
}
else
{
    slogfile = new RandomAccessRecordFile(rf, "SAM",
        RandomAccessRecordFile.INPUT);
}
/**
```

```
     * check if there is at least 1 record in the
     * slogan file
     * if not, the modulo function would fail
     */
    if ((numOfRecs = slogfile.getRecordCount()) == 0)
    {
        slogfile.close();
        error("Slogan file is empty!");
    }
    /**
     * Position to a randomly selected record within
     * the file.
     * Thanks to the modulo function (%) we are sure
     * that the position will always be inside the file
     */
     slogfile.setCurrentRecordNumber(
        Math.abs(generator.nextInt() % numOfRecs));
    /**
     * read the record and show the slogan
     */
     record = slogfile.read();
     System.out.println("Slogan of the day: "
          + record.getStringData());
     /**
      * close the slogan file
      */
     slogfile.close();
    }
}
```

The program can be compiled with the Java compiler javac and then be run. No particular specifications are required to make the JRIO interfaces available.

Indexed-sequential data processing

To demonstrate indexed-sequential data processing a program is used which monitors the lifetime of files on an ID. The program expects two parameters: the user ID to be monitored and the name of the file in which the program can store data. When first called the file should not yet exist. It is then created with the correct attributes for the program.

The example generates an ISAM file with fixed record length as a database. The error handling in the example is not particularly convenient, simply to prevent the comprehensive code which would be required for this distracting the reader from the way the interface is actually used.

The program is contained in the *FileHistory.java* file:

```
import java.io.*;
import com.fujitsu.ts.jrio.*;
import com.fujitsu.ts.jrio.DMS.AccessParameterISAM;
import java.util.Date;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
/**
 * The demo program FileHistory provides a
 * simple mechanism to log changes in the files belonging
 * to a given BS2000 userid. In fact, only two dates are
 * logged for each file: date first seen and date last seen.
 *
 * Every time the program is started it synchronizes the
 * current list of filenames with the list of filenames
 * given by the logfile:
 *
 * New filenames are added to the logfile with date first seen
 * and date last seen set to the current date.
 *
 * For filenames of the current list which are already logged
 * the date last seen is updated.
 *
 * Filenames in the logfile which are no more in the current
 * list remain untouched.
 *
 * The program should run once a day, to create a complete
 * history.
 *
 * The interesting part of this program is the method
 * doFileHistory(), all other methods are added to make it
 * a complete executable program.
 */
public class FileHistory
{
    /**
     * The main method, which analyses the program arguments,
     * calls the work method and provides global error
     * handling.
     */
    public static void main(String args[])
    {
        String userid = null;
        String logfilename = null;
        for (int i = 0; i < args.length; i++)
```

```
        {
            if (userid == null)
                userid = "$" + args[i] + ".";
            else if (logfilename == null)
                logfilename = args[i];
            else
                usage();
        }
        if (userid == null || logfilename == null)
            usage();
        try {
            doFileHistory(userid,logfilename);
        } catch (Exception e) {
            error(e.toString());
        }
    }
    /**
     * Print a usage message and exit with error
     */
    private static void usage()
    {
        error("Usage: FileHistory userid logfile\n"
            + " - userid without '$' and '.'");
    }
    /**
     * Print the given error message and exit
     */
    private static void error(String msg)
    {
        System.err.println(msg);
        System.exit(1);
    }
    /**
     * The work method.
     * This method demonstrates, how the JRIO interfaces
     * may be used to update records in an ISAM file
     *
     * @param userid
     *        the userid (with '$' and '.') to be scanned
     * @param logfilename
     *        the file containing the log records
     */
     public static void doFileHistory(String userid,
             String logfilename)
         throws IOException
     {
         /**
          * The current Date as string,
          * to be written to the log record
          */
         DateFormat df = new SimpleDateFormat("yyyy.MM.dd");
         String toDay = df.format(new Date());
         /**
          * The directory to be scanned for additional or
          * deleted files in its canonical form
          */
         RecordFile root =
             new RecordFile(userid,"DMS").getCanonicalFile();
         /**
```

```
 * List of filenames within the scanned directory
 */
String[] rfList = root.list();
/**
 * Definition of file for logging
 */
RecordFile logfile =
    new RecordFile(logfilename,"DMS");
KeyedAccessRecordFile log = null;
/**
 * key descriptor of the log file
 * and dummy key value (will be filled later
 * and used for reading)
 */
KeyDescriptor keyDesc = null;
KeyValue keyVal = null;
/**
 * Records from the logfile are read into this buffer.
 * It has fixed length: filename (54),
 * date fist seen (10), date last seen (10)
 */
Record logrec = new Record(54 + 10 + 10);
/**
 * check if the logfile already exists
 * and prepare access parameter
 */
AccessParameterISAM accesspar;
if (!logfile.exists())
{
    /* No, create it */
    accesspar = (AccessParameterISAM)
        logfile.getDefaultAccessParameter("ISAM");
    accesspar.setPrimaryKeyPosition(0);
    accesspar.setPrimaryKeyLength(54);
    accesspar.setRecordFormat(
        AccessParameter.RECORD_FORMAT_FIXED);
    accesspar.setRecordLength(54 + 10 + 10);
    if (logfile.createNewFile(accesspar) == false)
        error("Cannot create file " + logfilename);
}
else
{
    accesspar = (AccessParameterISAM)
        logfile.getAccessParameter("ISAM");
}
/**
 * Open the log file
 */
log = new KeyedAccessRecordFile(
    logfile,accesspar,KeyedAccessRecordFile.INOUT);
/**
 * Get the key descriptor of the log file
 */
keyDesc = log.getPrimaryKeyDescriptor();
/**
 * Consistency check
 */
if (keyDesc.getPosition() != 0
    || keyDesc.getLength() != 54)
```

```
        {
            log.close();
            error("File " + logfile
                + " is no valid logfile.");
        }
        /**
         * create key value connected with key descriptor
         * proper values will be inserted later
         */
        keyVal = new KeyValue(keyDesc);
        /**
         * loop through the list of filenames
         */
        for (int i = 0; i < rfList.length; i++)
        {
           /**
            * prepare key vaue for reading the
            * log record for this filename
            */
           keyVal.setStringValue(rfList[i]);
           /**
            * check if the filename is already in the log
            */
           if (log.read(keyVal,logrec) > -1)
           {
               /**
                * yes, filename did exist at last run,
                * update 'date last seen' field
                */
               logrec.setStringField(toDay,64,10);
               /**
                * write updated record back to logfile
                */
               log.writeBack(logrec);
           }
           else
           {
               /**
                * filename is new:  build a new record
                */
               logrec.setKeyField(keyVal);
               logrec.setStringField(toDay,54,10);
               logrec.setStringField(toDay,64,10);
               /**
                * write new record to log file
                */
               log.write(logrec);
           }
       }
       /**
        * close the logfile
        */
       log.close();
   }
}
```

The program can be compiled with the Java compiler javac and then be run. No particular specifications are required to make the JRIO interfaces available.

Invoking the VM from the BS2000 command interface

The *INITIALIZE*, *DELETE* and *START* procedures are available in the PLAM library *SYSPRC.JENV.170*.

- *INITIALIZE* is used to set the environment variables needed to execute the VM.
- *START* is used to start the VM. If *INITIALIZE* is not invoked before *START*, the default values are used.
- *DELETE* is used to delete all environment variables set by *INITIALIZE*.

The procedures are also delivered in a compiled variant so that the user can execute them without the product SDF-P.

A prerequisite for execution is that the user has permission to run POSIX programs and is authorized to access the POSIX file system on which the POSIX part of JENV is installed.

INITIALIZE procedure

The *INITIALIZE* procedure sets the environment variables that are evaluated by the Java VM. This is done by setting the corresponding structure elements of the structure variable *SYSPOSIX*. Other existing structure elements of this structure remain unchanged. If the structure variable *SYSPOSIX* does not already exist, it will be created.

Parameters

JAVA-HOME

Determines the value of the environment variable *JAVA_HOME* (see chapter "Environment variables"). If the parameter is not specified or is set to *'*STD'*, the variable is not assigned. Any existing assignment is cleared.

CLASSPATH

Determines the value of the environment variable *CLASSPATH*. If the parameter is not specified or is set to *'*STD'*, the variable is not assigned. Any existing assignment is cleared.

LD-LIBRARY-PATH

Determines the value of the environment variable *LD_LIBRARY_PATH*. If the parameter is not specified or is set to *'*STD'*, the variable is not assigned. Any existing assignment is cleared.

PWD

Sets the value of environment variable *PWD* and thereby determines the *current working directory* . If the parameter is not specified or is set to *'*STD'* , the directory set for the user id with the command */MODIFY-POSIX-USER-ATTRIBUTES DIRECTORY*= ... is used.

DISPLAY

Determines the value of the environment variable *DISPLAY*. This specifies the address of the screen in which the X-Windows are displayed. If the application operates without X-Windows, the value of this variable is irrelevant.If the parameter is not specified or is set to *'*STD'*, the variable is not changed.

SCOPE

Specifies the scope of the structure variable *SYSPOSIX*. The default value is *'*TASK'*. The parameter is passed directly to the *SCOPE* operand of the *DECLARE-VARIABLE* command (see manual "SDF-P (BS2000) " [7]). Only the *'*TASK'* and *'*PROCEDURE'* procedures with their sub-operands are meaningful, and '*PROCEDURE' is only meaningful if the procedure is called with *INCLUDE-PROCEDURE*.

Since the system is case-sensitive, all parameter values must be entered enclosed in single quotes.

In addition to this, the following environment variables are always set implicitly:

PROGRAM_ENVIRONMENT = 'shell'

as the Java VM can only be run in this mode.

HOME

to the home directory which was set for the user id with the command */MODIFY-POSIX-USER-ATTRIBUTES DIRECTORY*= ... .

START procedure

The *START* function starts the VM with the command *START-PROGRAM* and passes the parameters which have been set. If the structure variable *SYSPOSIX* does not already exist, the *INITIALIZE* procedure is first invoked using the default values. If the structure variable *SYSPOSIX* does already exist, *INITIALIZE* will not be called. The internal environment variables necessary for calling the tool will however be set.

Parameters

CMD

Must be assigned one of the following values:

'jar'
'jarsigner'
'java'
'javac'
'javadoc'
'javap'
'jconsole'
'jdb'
'jdeps'
'jimage'
'jlink'
'jmod'
'keytool'
'native2ascii'
'rmiregistry'
'serialver'

The values correspond to the shell commands.

Other values:

'?'

'help' outputs a help text in English.

'hilfe' outputs a help text in German.

ARGS

The arguments for the command above are to be enclosed in single quotes.

The wildcard substitution function, which is usually available under the shell, is not supported.

REDIRECT

This parameter must be used if input/output is to be redirected. This is done in the same way as for the corresponding option under the shell. For example: *REDIRECT='2>MyFile'* redirects the output of *stderr* to *MyFile*.

See the section "Redirection of default streams".

SYSHSI

This parameter must be assigned to one of the following values:

'*STD'
'X86'
'S390'

This parameter specifies, whether the s390 variant of the Java VM or the X86 variant is used.

Default value: '*STD'

The variant corresponding to the system is used.

INSTALLATION-ID

User ID of the JENV installation. This parameter must only be specified if the object to be started under VM is not stored under the same user ID as the procedure library in which the *START* procedure is located.

Redirection of default streams

If *PROGRAM_ENVIRONMENT='shell'* is set, the file names into which the default streams are redirected refer to files in the POSIX file system.

It is possible to redirect the streams to BS2000 files using the usual prefix */BS2/*. To redirect to *SYSDTA*, *SYSOUT* or *SYSLST* you must also use this prefix, i.e. */BS2/(SYSDTA)*, */BS2/(SYSOUT)* or */BS2/(SYSLST)*. If the prefix is not used, redirecting to *(SYSOUT)* will result in a POSIX file being written with the name *(SYSOUT)*.

The same applies to redirections which indicate special treatment under the shell. Outside the shell everything to the right of < or > is interpreted as a file name. So, for example, a redirection of *2>&1* creates a file called *&1*.

The redirection of *stdout* and *stderr* to the same BS2000 file is not possible, and if these streams are redirected to the same POSIX file, output data may be lost.

Example

If an applet is to be started via the file */MyDir/MyTest/Plasma.class* and the terminal has the symbolic address *ABCD1234*, this could be achieved as follows:

```
/CALL-PROCEDURE *LIB($TSOS.SYSPRC.JENV.170,INITIALIZE),
                (PWD='/MyDir/MyTest ',DISPLAY='ABCD1234:0.0')
/CALL-PROCEDURE *LIB($TSOS.SYSPRC.JENV.170,START),
                 (CMD='java',ARGS='Plasma')
```

DELETE procedure

The *DELETE* procedure deletes all elements of the structure variable *SYSPOSIX* which are set by the *INITIALIZE* procedure. If the *SYSPOSIX* structure subsequently contains no elements, it is itself deleted.

Parameters

SCOPE

Specifies the scope of the structure variable SYSPOSIX. The default value is *'*TASK'.* The *'*PROCEDURE'* value need only be specified if it was specified in the *INITIALIZE* procedure (see section "INITIALIZE procedure").

Invoking the VM using the invocation API

If a C or C++ program which invokes the VM via the invocation API is started using *START-PROGRAM*, the environment variables must be set using the *INITIALIZE* procedure. The following operands must be set in the *START-PROGRAM* command:

```
PROGRAM-MODE=*ANY,RUN-MODE=*ADVANCED,SHARE-SCOPE=*NONE.
```

> A C/C++ program must be linked with the Java Runtime Adapter and not with the normal CRTE-, C++ or socket libraries (see section "Invocation API").

Special considerations

When invoking a BS2000 program using *START-PROGRAM* neither the */etc/profile* nor the *.profile* file of the user is executed. The result of this is that a program may, in some cases, behave differently than if it had been started under the shell. If the file access rights of newly created files are restricted in the profiles using *umask*, this does not apply to programs started using *START-PROGRAM*. The result of this is that these programs then create files with more extensive access rights than intended. There is currently no solution available to remedy this. The tools are also affected because they are called with the *START-PROGRAM* command in the *START* procedure.

The environment variable *PATH* is not set after *START-PROGRAM*. The consequence of this is that creating a new process with *fork/exec* is not possible under some circumstances when the program to be started cannot be found. It is possible to resolve this problem by setting the SDF-P variable *SYSPOSIX.PATH* to the value used in the shell before calling *START-PROGRAM*, or by specifying a complete path name in the program for *exec()*. In Java this problem effects the method *Runtime.exec()*.

Example

The following instruction can only be excuted if the environment variable *PATH* was set correctly:

```
Process child = Runtime.getRuntime().exec("java Myclass");
```

The following instruction rectifies the problem:

```
Process child =
Runtime.getRuntime().exec(System.getProperty("java.home") +
"/bin/java Myclass");
```

Even if the VM is started using *START-PROGRAM,* the input/output is, by default, directed to files in the POSIX file system. BS2000 files can be opened using the package JRIO. The class files must be located in the POSIX file system.

JNI under BS2000

This chapter describes the special features which a user of Java native interfaces (JNI) needs to look out for in BS2000. The chapter will not go in any depth into the general use of the native interfaces (i.e. independent of the operating system)

Specifications and tutorials on this are available in the internet and on the book market.

The use of the JNI for real applications is not simple, since komplex interaction between the Java and C environments is possible. Before making the decision to use the JNI, you should discuss the alternatives carefully.

The different variants of JNI

Only Version 1.2 of JNI is still supported.

Java data types in C

A mapping, which essentially also applies to BS2000, has been defined between the primitive Java data types and the native C representation. The following table provides a summary of the data types and any special features:

Java type	C type	Compatible C type	Remarks
boolean	jboolean	unsigned char	JNI_FALSE, JNI_TRUE
byte	jbyte	signed char	
char	jchar	unsigned short	Unicode
short	jshort	signed short	
int	jint	signed int	
long	jlong	signed longlong	from C/C++ V3.0B
float	jfloat	float	IEEE
double	jdouble	double	IEEE
void	void	void	

Table 7: Java data types in C

For complex data types, JNI defines corresponding access and conversion functions which can be used in BS2000 analogously to other operating systems. A special role is played here by strings as the UTF-8 encoding of Unicode strings which is used by Java, although closely related to ASCII, is quite unlike EBCDIC encoding. A C programmer in an ASCII environment (Unix systems, Windows systems) will therefore easily succumb to the temptation to use this similarity, with a result that it will not be possible to use such C programs in BS2000 (i.e. in the EBCDIC environment) without taking some further measures.

When C code and Java are linked up via the JNI, there will inevitably be instances in BS2000 where different forms of data encoding coincide. Users must decide for themselves where they want to make corresponding conversion points between the data representations. The essential and critical conversion points are shown in the following table:

Data	Representation in Java	Normal representation in BS2000	Alternative representation in BS2000
Whole numbers	32 and 64 bit	32 bit	32 and 64 bit
Floating point numbers	IEEE format	/390 format	IEEE format
Strings, characters	Unicode, UTF-8, ASCII	EBCDIC	ASCII

Table 8: C code in Java and BS2000

In order that the user can make a free choice of conversion point, appropriate help on the various topics is provided through the compiler and runtime systems.

Typically, a JNI interface user will implement this conversion point either directly at the JNI interface and have all his C code run in the normal BS2000 environment or else he will have parts of his C code (or even all of it) run in the alternative representation which is more closely oriented to Java (and Unix systems) and, for example, only carry out the relevant conversions in the context of legacy applications (use of well-tried software).

The sections below describe the support available for the various data types.

Whole numbers

The Java data type *long* is a 64-bit data type which is represented in the JNI by the C data types *jlong*

The C/C++ compiler (as of version 3.0B) supports the data type *longlong* or *int64_t*, which is compatible with the above mentioned data types (i.e. *jlong*). This means that this data can be used in C without any further precautionary measures being required. The scope of the support available through C runtime system functions as of CRTE V2.1B is explained in the appropriate CRTE documentation.

Floating point numbers

The Java data types *float* and *double* are floating point data types which are represented in the JNI through the C data types *jfloat* and *jdouble*.

These data types are formally compatible with the C data types *float* and *double*. However, as they are represented in IEEE format (instead of /390 format) they cannot be used in C without taking precautionary measures.

As well as explicit conversion options, appropriate compiler and runtime system extensions are provided to support the IEEE format. These allow you to work directly with this number format in C.

Explicit conversion

A number of functions are available for explicit conversion between floating-point numbers in IEEE format and in /390 format. These are declared in the header file *ieee_390.h*, which is part of the CRTE distribution. These conversion functions are described in the manual “CRTE” [3].

Example

The following example shows the use of a conversion function in a native method which performs arithmetic manipulations on a floating point number. On the Java side the method will be declared as:

```
public native double manipulate(double arg);
```

The associated C program could look like this:

```
#include <jni.h>
#include ".....h" // javah generated Header
#include <ieee_390.h>
JNIEXPORT jdouble JNICALL
Java_..._manipulate(JNIEnv *env, jobject jthis, jdouble num);
{
  double result, arg;
  arg = ieee2double(num);
  result = (arg < 1.7)? arg * 3.4 : arg - 1.0;
  return double2ieee(result);
}
```

The above code example does not contain any error handling for possible conversion errors.

IEEE floating point numbers in the C code

As of version V3.0B, the C/C++ compiler allows you to generate code for IEEE format as an alternative to /390 format for floating point numbers. The setting, which is controlled via the compiler option *-Kieee_floats*, applies to the entire compilation unit (source file).

This option only has an effect on floating point constants in the source code, and on arithmetic, type conversion or comparison of floating point numbers. It has no effect on the passing of such data to other functions or simple assignments

Setting this option also has the effect of implicitly permitting the use of C library functions with floating point arguments and/or floating point result in a variant for IEEE arithmetic.

All the arithmetic is processed using corresponding emulation routines. This applies to SQ systems too, as long as generation of native code for the corresponding commands via Asstran is not possible. Naturally this has a negative effect on performance. C programs which make intensive use of floating point arithmetic should therefore not be run in this mode.

Example

The example shown above could then be implemented as follows:

```
#include <jni.h>
#include ".....h" // javah generated Header
JNIEXPORT jdouble JNICALL
Java_..._manipulate(JNIEnv *env, jobject jthis, jdouble num)
{
    return (num < 1.7)? num * 3.4 : num - 1.0;
}
```

The compilation must be carried out using the C compiler option *-Kieee_floats*.

IEEE floating point numbers in the C runtime system

The C runtime system contains, in addition to the conversion routines which are declared in the *ieee_390.h*, all the essential XPG4 functions which work with floating point numbers in a variant for IEEE arithmetic. When the aforementioned compiler option for using IEEE is selected, the corresponding library functions are normally used automatically without the user needing to do anything. You can also modify this behavior for mixed mode (see the manual “CRTE” [3]).

Example

The next example illustrates the use of the IEEE version of the C function *tanh* in a native method for calculating the hyperbolic tangent in a Java class. On the Java side the method will be declared as:

```
public native double tanhyp(double arg);
```

The associated C program could look like this:

```
#include <math.h>
#include <jni.h
#include ".....h" // javah generated Header
JNIEXPORT jdouble JNICALL
Java_..._tanhyp(JNIEnv *env, jobject jthis, jdouble num)
{
    //printf("tan_hyp called with: %e\n",num);
    return tanh(num);
}
```

To work correctly it must naturally be compiled in this form using the C compiler option *-Kieee_floats*.

Strings

The Java data type *string* is provided in JNI as data type *jstring*. This type cannot be used directly in C; in particular, it has no commonality with the C data type *char* *. In order to convert the string to a form which can be processed in C, the corresponding JNI interfaces must be used for the conversion (see JNI documentation).

The Java data type*Char* is available at the JNI interface as data type *jchar*. This is compatible with the C data type *unsigned short* and constitutes one character in Unicode representation. The first 256 characters in Unicode are identical to the ISO8859-1 encoding. Unicode characters outside this range are not supported in C/C + + in BS2000. Processing of these characters must therefore be undertaken by users themselves.

The UTF-8 representation of Unicode, which is partially used by Java in the JNI, plays a special role. In UTF-8 representation, Unicode characters are encoded into one, two or three bytes. Under this encoding, Unicode characters with codes 1 to 127 are represented with this value in a single byte, corresponding once again exactly to the ASCII encoding of these characters.

Moreover, UTF-8 byte sequences are always terminated in Java with a NULL byte, which enables them to be processed as C strings. Here, the Unicode NULL character is encoded into two bytes so as to avoid confusion with the string delimiter in C, since, unlike in C, it is perfectly acceptable in Java for strings to contain NULL characters.

The following simple rules apply to the processing of UTF-8 byte sequences in C:

- The NULL byte marks the end of the byte sequence, and is absolutely essential.
- Bytes for which the function *isascii_ascii()* returns the value “`true`” (1-127) are also in fact ASCII characters as per ISO8859-1
- To represent Unicode characters outside the range 1 to 127, all the other bytes are treated as if they were part of a multibyte sequence. These have to be interpreted by the user.

As nearly all these conversion functions constitute character sequences at least in a form which is upwardly compatible with ASCII, code conversion from ASCII to EBCDIC and vice versa does not play a special role in BS2000. Naturally, this applies not only to strings but also, for example to byte arrays or characters (*jchar*).

References to “ASCII” in this manual always refer to the ISO8859-1 character set (ISO Latin 1) or its 7 bit offshoot (ISO 646). “EBCDIC” refers to the character set DF04-1 (international reference version) with swapped 0x15 and 0x25 or its 7 bit offshoot DF03-1.

As well as explicit conversion facilities, to support ASCII strings, appropriate compiler and runtime system extensions are available which allow you to work directly with ASCII strings and characters in C.

Explicit conversion

The JNI conversion functions (see „JavaTM Native Interface” [13]) work in BS2000 exactly as specified. They always return or else expect Unicode or UTF-8.

Some functions are available in CRTE for explicit conversion between ASCII (8859-1) and EBCDIC (DF04-1). These are declared in the header file <*ascii_ebcdic.h*>, which is part of the CRTE distribution. These conversion functions are described in the manual “CRTE” [3].

Example

The next example illustrates usage in a native method which ascertains the value of an environment variable and removes the prefix *JAVA_* from this. On the Java side the method will be declared as:

```
public native String get_jenviron(String name);
```

The associated C program could look like this:

```
#include <jni.h>
#include ".....h"         // Header generated by javah
#include <stdlib.h>
#include <ascii_ebcdic.h>
JNIEXPORT jstring JNICALL
Java_..._get_jenviron(JNIEnv *env, jobject jthis,
                      jstring name)
{
    const char *utf_name;
    char *ebcdic_name, *ebcdic_value, *utf_value;
    jstring value;
    utf_name = (env*)->GetStringUTFChars(env,name,NULL),
    ebcdic_name = _a2e_dup(utf_name);
    (*env)->ReleaseStringUTFChars(env,name,utf_name);
    ebcdic_value = getenv(ebcdic_name);
    free(ebcdic_name);
    if (ebcdic_value == NULL)
      return NULL;
    if (strncmp(ebcdic_value,"JAVA_",5) == 0)
      utf_value = _e2a_dup(ebcdic_value+5);
    else
       utf_value = _e2a_dup(ebcdic_value);
    value = (*env)->NewStringUTF(env,utf_value);

    free(utf_value);
    return value;
}
```

The above sample code does not contain any error handling. It is implicitly assumed that in all strings only characters from the 7 bit ASCII character set will occur. Moreover, this code is naturally very much BS2000-specific.

ASCII strings in the C code

As of version V3.0B, the C/C++ compiler allows you to generate an equivalent ASCII code as an alternative to the normal EBCDIC encoding for string and character literals. This setting must apply to a complete compilation unit (source file) and is controlled via the compiler options *-Kliteral_encoding_ascii* and *-Kliteral_encoding_ascii_full*. The difference between the two options lies in the treatment of octal and hexadecimal sequences in such literals. With *-Kliteral_encoding_ascii* such literal parts are not converted.

ASCII strings in the C runtime system

In addition to the above conversion routines, the C runtime system provides further support for the use of ASCII strings and characters. All key XPG4 functions that work with or return strings or characters are available in a variant for ASCII coding. When one of the compiler options for ASCII use described in the section "ASCII strings in the C code" is set, the corresponding library functions are generally used automatically without the need for user intervention. You can change this behavior for mixed operation (see the manual “CRTE” [3]).

If the compiler option *-Kieee_floats* is set at the same time, the combined ASCII/IEEE variants are used (e.g. with *printf*).

As of C Compiler V3.1A and CRTE V2.4C, the arguments of the vector *argv[]* are passed as ASCII strings when compiling the main program with one of the compiler options described in the section "ASCII strings in the C code". The global variables of the C runtime system *tzname* and the strings of *environ* are saved as ASCII strings. Explicit conversion of *argv[]* is therefore unnecessary.

If explicit access is made to the strings of the global variables *tzname* or *environ*, it should be noted that as of JENV V1.4B these are stored as ASCII strings (formerly EBCDIC strings). However, the Technical Standard “the Single UNIX Specification” warns against explicit access to the *environ* variable (see “X/Open System Interface (XSI) Specification” [16]). Implicit access using *getenv()* and *putenv()* functions as in the past and is compatible with previous versions.

Example

If you use these options, the above C program could look like this:

```
#include <jni.h>
#include ".....h" //        javah generated Header
#include <stdlib.h>

JNIEXPORT jstring JNICALL
Java_..._get_jenviron(JNIEnv *env, jobject jthis,
                      jstring name)
{
   const char *utf_name;
   char *utf_value;
   utf_name = (*env)->GetStringUTFChars(env,name,NULL);
   utf_value = getenv(utf_name);
   (*env)->ReleaseStringUTFChars(env,name,utf_name);
   if (utf_value == NULL)
       return NULL;
   if (strncmp(utf_value,"JAVA_",5) == 0)
     return (*env)->NewStringUTF(env,utf_value+5);
   else
     return (*env)->NewStringUTF(env,utf_value);
}
```

This implementation is exactly the same as one which could also be used on Unix systems This form is therefore the one most highly recommended for ported code.

Dynamic loading of native methods

Native methods for Java must be dynamically loadable. The procedure here is very similar to the established methods in Unix systems (shared libraries). The Unix concepts and the BS2000 implementation will now be compared. The BS2000 solution and the associated requirements for the user will then be described in detail.

Java applications on Unix platforms require that native methods are produced as shared libraries. The native methods can then be dynamically loaded and called. The C system functions *dlopen()* and *dlsym()* are used for this purpose.

Although in OSD-POSIX there is now a shared libraries implementation, the analogous mechanism familiar from the preceding version has been retained. However, not all the functionality of the shared libraries is offered here but only those functions which are needed in the Java environment.

Shared libraries in Unix systems

Shared libraries contain an object (i.e. a module which can be loaded and executed by the system loader) with a special structure (a “shared object”). One of the characteristics of shared objects is that they can be dynamically loaded during program execution.

List of required objects

A shared object can specify other objects which are necessary in order for it to be executed. These objects are loaded at the same time as a shared object is loaded and are considered during resolution of unresolved external references. Here again, each of these objects can specify other required objects, so that chains are formed.

Name spaces

When a shared object is loaded, other dynamically loaded shared objects are not accessed unless they are included in the list of required objects.

An exception here is the context in which the program was loaded on startup (and all the objects which were dynamically loaded at that time).

This causes the name spaces to be partitioned.

Search sequence

The search for shared objects during program execution is controlled through the environment variable *LD_LIBRARY_PATH*, in which different directories can be specified which the system will search through in the specified sequence, looking for the shared objects which are to be loaded.

Resolution of external references

When a shared object is loaded, any unresolved external references are initially resolved from the primary load context. The current shared object is then included and finally the objects which were loaded as required objects. (This is a simplified version. Full details are provided in the interfaces descriptions of *dlopen()* and *dlsym()* in the corresponding Unix manuals).

As the external references within a shared object are not resolved, a function which exists in a shared object can be overwritten by a function of the primary load context (this is not possible in LLMs!).

Naming convention

Shared libraries always begin with the prefix *lib* and end with the suffix *.so*, for example, *libhello.so*. Often a name also has a version suffix for the co-existence and unambiguous assignment of different interface versions, for example, *libXm.so.1.2*.

Shared libraries in BS2000

As already mentioned, in BS2000 there is no exact correspondence to the familiar shared objects from Unix systems. The characteristics essential to Java such as dynamic loading, the partitioning of name spaces and the dynamic determination of function addresses are mapped during the Java port. On the other hand, the naming property of multiple usage, the implicit loading of shared objects at program start and the subtleties of resolution cannot be mapped. This would require extension of the linking loader.

As the BS2000 linking loader cannot dynamically load any module from the POSIX file system, native methods must be created as LLMs and stored in PLAM libraries.

In the LLM there is no means of specifying a “list of required objects”, yet this functionality is necessary for Java and a search method analogous to Unix systems would appear to be useful in the POSIX file system. Hence, an additional description file has been implemented. This file contains what amounts to a description of a shared object. It is stored in the POSIX file system, observes the same naming conventions as shared libraries in Unix systems and contains all the information needed by Java in order to dynamically load and call the native methods.

This information comprises above all the PLAM library in which the LLM is stored, the name of the module (or modules) and, if appropriate, the list of required objects.

List of required objects

A list of required objects can be entered in the description file. These objects are dynamically loaded before the current object. Objects which already exist are not dynamically loaded again. Objects are identified by their POSIX file names.

These objects are included during loading of the current object to resolve external references.

It is perfectly possible for different shared objects to contain the same objects in their lists of required objects. The first reference to such an object then leads to dynamic loading

Name spaces (link contexts)

Each object is loaded in a separate link context. Objects are therefore partitioned in their name space.

The BS2000 linking loader now allows 200 link contexts. If more objects are loaded the application is aborted.

Search sequence

The search for shared objects (or rather, for the description files) operates in exactly the same way as in Unix systems, i.e. it is controlled through the environment variable *LD_LIBRARY_PATH*.

Resolution of external references

The contexts into which the required objects have been loaded are specified as reference contexts. The default context is used as reference context with the highest priority.

Searching through the share scope is explicitly prevented as it is not possible at the present time to see to it that this does not happen until after the reference contexts have been handled.

To resolve any unresolved external references, the system therefore initially searches through the default context and then through the required objects. All other objects are ignored.

> This continues to be different from Unix systems. In particular, all external references in an LLM are shorted, so that no function in an LLM can be overwritten.

Creation of shared objects

The next few sections explain the procedure for creating a shared object with native methods which can later be dynamically loaded by the Java VM.

Compilation of source code

To compile the C source code of Java native methods, the C/C++ compiler as of V3.0B must be used for the parts of the source code which work with the JNI.

C++ sources have to be compiled with a C/C++ compiler with version 4.0A or higher.
For language mode, only the option -X 2017 or -X 2020 is allowed, the library version must be 1 (-K library_version=1 or unspecified).

When compiling the C or C++ parts, it is essential that the following compiler options are used:

-I <Installation path>**/include**

This option is necessary in order that the Java distribution header files are found. For <*Installation path*> the path in which JENV has been installed must be substituted. For a standard installation, this is */opt/java/jdk-17.0.5*. Refer to the Release Notice for the currently valid name.

-K workspace_stack

This is necessary in order that the Garbage Collector can also find the Java objects used in the C parts and that the objects can be thread-safe.

-K c_names_unlimited

This is necessary in order that the name mangling correctly functions for native interface functions.

-K llm_keep

This is necessary in order that the name mangling correctly functions for native interface functions and that the runtime system functions are found.

-K llm_case_lower

This is necessary in order that the name mangling correctly functions for native interface functions and that the runtime system functions are found.

-D __SNI_THREAD_SUPPORT

This option is mandatory for C++ compilations.

The following compiler options can be useful:

-K ieee_floats

Used when you want the IEEE format for floating point numbers to also be used in the C code.

-K literal_encoding_ascii
-K literal_encoding_ascii_full

Used when you want to use ASCII strings in the C code.

-K enum_long

Should always be set, as the default setting does not conform to the ANSI standard.

Furthermore, it is essential that C-Sources are compiled in ANSI mode (*-Xa* or *-Xc*).

Linking a main module

If the implementation of a shared object is to consist of several modules, then these should be linked together into a main module. This is done using the command *cc* or *c89*, where the following options must be specified

-r

This option has the effect of linking a main module without adding any standard libraries like (CRTE). Under no circumstances should these be explicitly linked to it with *-lc* or *-lsocket*.

-B llm4

This option cause the linker to create a main module in LLM4 format which is necessary for the long name of the Java native methods.

Creating an LMS library

The main module created (and held in the POSIX file system) must be stored in a PLAM library as an element with the element type L. The best way to do this is with the POSIX command *bs2cp*, which also creates the library if it does not yet exist.

It is quite in order for several shared objects to be stored in such a PLAM library

> The element name of the module in the library must not exceed 32 characters.

Creating the object description

To create the necessary description file for a shared object, the command *mk_shobj* is available. The command *pr_shobj* is used to view the content of such a description file. Both commands are part of the Java distribution and are described in detail in chapter "Commands for BS2000".

C++ objects must be labeled as such (see subsection "Options" of section "mk_shobj").

Use of shared objects from Java

To dynamically load the user's native methods, it is necessary in Java to call, for example, the method *System. loadLibrary()*.

A new name is formed from the name specified in *loadLibrary()*, under which the library is then searched for. This name is *lib<name>.so*

The library is searched for using the environment variable *LD_LIBRARY_PATH*. The first description file found using this method is then used to dynamically load the appropriate module or modules.

The JVM and the native methods of Java are held in separate shared libraries and loaded into separate contexts in each case. Thus, if JVM interfaces which extend beyond the JNI interface are used (which they should not be), the corresponding dependencies to the shared Libraries should be entered in the user libraries.

When the first C++ method is started a C++ runtime system is loaded dynamically (including the tools and standard library) and C++ is initialized, if this has not already been done.

During dynamic loading of shared libraries (BIND macro), at present the system does not search through the "share scope" to resolve any open external references as this would mean it could no longer be guaranteed that the Java private CRTE or sockets will be used.

This must likewise be done if the user himself dynamically loads via BIND code, at least when references to the C runtime system and the sockets exist.

Java native methods and main modules containing them cannot be pre-loaded with the current linking loader.

Invocation API

The invocation API is a part of the JNI for invoking Java from C/C++ applications. Only Version 1.2 is still supported.

Changes to the invocation API

The invocation API provides no interface which allows you to select which variant of the HotSpot™ VM (client, server, etc.) is to be used by the program.

In BS2000 the client VM is normally used by default. However, in this implementation it is also possible to use the environment variable *JENV_VMTYPE* to select another VM variant (if available) (see chapter "Environment variables").

Compiling the C and C++ sources

C++ sources have to be compiled with a C/C++ compiler with version 4.0A or higher.
For language mode, only the option -X 2017 or -X 2020 is allowed, the library version must be 1 (-K library_version=1 or unspecified).

The compiler options described in section "Implementation of the Java code" of chapter "Implementation of a native method in C" must be used when compiling the C/C++ parts.

The HotSpot™ VM handles overflow events itself. To prevent interrupts occurring you must also specify:

-K no_integer_overflow

This option must be set for the main program.

The C/C++ Compiler as of Version 3.1A20 has been changed (and is therefore incompatible) so that if a main program is compiled with this compiler the argument strings are automatically passed as ASCII strings if the *-K literal_encoding_ascii* or *-K literal_encoding_ascii_full* option was set. Explicit conversion with, for example, _e2a() is not needed. If an existing main program already performs this conversion and you do not want to change it, the following option must be specified for reasons of compatibility:

-K environment_encoding_ebcdic

The argument strings continue to be passed as EBCDIC strings.

Linking C and C++ applications with Java and Green Threads

When linking C/C++ applications the link options described in section "Implementation of a native method in C++" must be used.

With JENV a runtime adapter is provided which has to be linked with C applications which need to call Java via the invocation API (part of the JNI). This adapter contains the functions of the Invocation API as well as the adapter to the thread-safe C and C++ runtime system and to the thread-safe socket library.

The runtime adapter is available in an optimized variant, as is used in *java*. The runtime systems are located in PLAM libraries which are part of the scope of delivery of JENV:

For S390:

SYSLNK.JENV.170.GREEN-JAVA.

For X86:

SKULNK.JENV.170.GREEN-JAVA

When linking an application with this runtime adapter, it must also be borne in mind that, due to the long names which occur in Java, this runtime system is of LLM type 4. It is therefore essential that the compiler option *-B llm4* is used during linking. It should also be noted that the C compiler normally automatically links a CRTE during linking and in the case of C++ a standard library. This must be prevented to avoid any conflict with the thread-safe runtime system already contained in the runtime adapter. This is achieved with the *Kno_link_stdlibs* option. For the same reason, no socket library can be explicitly linked and nor can any tools library. During linking, the options *-lc, -lsocket* or *-ltools* should therefore never be used.

Under POSIX a C application can be linked as follows with JENV:

```
export BLSLIB00='$.SYSLNK.JENV.170.GREEN-JAVA'
cc -Kno_link_stdlibs -B llm4 -o <program> \
    <objects> -l BLSLIB
```

The linked program can then be run without further precautions although naturally it needs a completely installed JENV under the standard installation path. If a Java installed elsewhere is to be used, the environment variable *JAVA_HOME* must be set to the installation path of the Java runtime environment (see chapter "Environment variables").

The *cc* command implicitly links the POSIX linkage option. If linkage is not carried out under the shell using the *cc* command, but under the BS2000 command line interface using *BINDER*, this option must be linked from *$.SYSLNK.CRTE.POSIX*.

In order to use the BINDER to obtain the required LLM4 format when using a OSD V3 for production you must specify the operand
*FOR-BS2000-VERSIONS=*FROM-OSD-V4* in *SAVE-LLM* . The objects can also run on OSD V3.

This procedure applies equally for C++ applications, in which case the command *CC* with the options specified above is to be used for linkage.

An application that explicitly calls the C interfaces of the POSIX sockets may not link the modules of the socket library but must link the *LIBSOCKET* module from *SYSLNK.JENV.170.GREEN* (or *SKULNK.JENV.170.GREEN*).

Examples

Four examples will now illustrate the complete process of creating a Java application using the JNI.

Implementation of a native method in C

Our sample application will consist of two Java classes *Hello* and *Work*, each of which contains a native method. One of them issues a greeting message, while the other performs a calculation. This example is highly artificial as normally no user would have this performed using native methods.

The native methods in both classes are to be stored in a common library called *example1*.

Implementation of the Java code

In a file called *Hello.java* the following Java class is defined:

```
class Hello {
   public native void greetings(String text);
   static {
     System.loadLibrary("example1");
   }
   public static void main(String[] args) {
       new Hello().greetings("Hello");
       new Work().compute();
   }
}
```

In the file *Work.java* the second Java class is defined:

```
import java.io.*;
class Work {
   public native double docompute(double arg);
   public void compute() {
     System.out.println("Resultat 1: " + docompute(1.0));
     System.out.println("Resultat 2: " + docompute(7.0));
     System.out.println("Resultat 3: " + docompute(3.11));
   }
}
```

If you had wanted to store the native methods in different libraries, each class would have to load its own library during initialization.

Compiling the Java code

The two Java classes can now be simply compiled using the command

```
javac Hello.java
```

The dependent class *Work* is created during this compilation.

Creation of header files

The header files which are needed in order to implement the native methods can be generated from the class files using the tool *javah*:

```
javah -jni Hello
javah -jni Work
```

Once this step is complete, the header files *Hello.h* and *Work.h* will be available with the prototypes of the native functions.

Implementation of the C code

The native methods are now typically implemented in corresponding source files. In our example these will be the files *Hello.c* and *Work.c*. Both files include the header which is provided with JENV *jni.h* and in each case the associated header previously generated, *Hello.h* or *Work.h*. The function definition must match the prototype which has been generated. The further coding depends on the desired implementation.

The program *Hello.c* will now be implemented in the example as follows:

```
#include <jni.h>
#include "Hello.h"
#include <stdio.h>
#include <stdlib.h>
#include <ascii_ebcdic.h>
JNIEXPORT void JNICALL
Java_Hello_greetings(JNIEnv *env, jobject jthis, jstring text)
{
  char *ebcdic_text;
  const char *utf_text;
  utf_text = (*env)->GetStringUTFChars(env,text,NULL);
  ebcdic_text = _a2e_dup(utf_text);
  (*env)->ReleaseStringUTFChars(env,text,utf_text);
  printf("The program responds here %s\n",ebcdic_text); free(ebcdic_text);
}
```

The file *Work.c* contains the following code:

```
#include <jni.h>
#include "Work.h"
JNIEXPORT jdouble JNICALL
Java_Work_docompute(JNIEnv *env, jobject jthis, jdouble num)
{
  return (num < 1.7) ? num * 3.4 : num - 1.0;
}
```

In the file *Work.c* use has been made of the option of transparent usage of IEEE functions, described in more detail above. In file *Hello.c* explicit ASCII-EBCDI conversions are carried out.

To make the examples clear and at the same time keep them short, detailed error handling has been omitted.

Compiling the C source

The C source code implemented in the section above must now be compiled using the correct compiler options. For *Hello.c* these are the standard options which are described in more detail above:

```
cc -c -I/opt/java/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited Hello.c
```

For *Work.c* the IEEE arithmetic must also be considered:

```
cc -c -I/opt/java/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited \
      -Kieee_floats Work.c
```

This results in the object files being made available

Creation of the shared object

The previously created objects can be linked to a main module with the following command:

```
cc -r -B llm4 -o example1.o Hello.o Work.o
```

The main module created is then stored in a BS2000 library.

```
bs2cp example1.o bs2:'syslnk.example1(example1,L)'
```

Finally, a description file which complies with the naming convention is created. This must naturally contain the correct references.

```
mk_shobj -l syslnk.example1 -m example1 libexample1.so
```

Processing of the program

To run the program all that remains now is to set the environment variable *LD_LIBRARY_PATH* so that the created shared object is also found. In our example this can be done using

```
export LD_LIBRARY_PATH=.
```

The application can now be started with

```
java Hello
```

Implementation of a native method in C++

Implementation in C++ is largely identical to the procedure used for implementation in C. The differences from the example above are as follows:

Program *Hello.cpp* is now implemented as follows:

```
#include <jni.h>
#include "Hello.h"
#include <iostream.h>
#include <ascii_ebcdic.h>
#include <stdlib.h>
JNIEXPORT void JNICALL
Java_Hello_greetings(JNIEnv *env, jobject jthis, jstring text)
{
  char *ebcdic_text;
  const char *utf_text;
  utf_text = env->GetStringUTFChars(text,NULL);
  ebcdic_text = _a2e_dup(utf_text);
  env->ReleaseStringUTFChars(text,utf_text);
  cout <<  "The program responds here" << ebcdic_text << endl;
  free(ebcdic_text);
}
```

For compilation, instead of using command *cc*, use command *CC*. For creation of the shared object, the flag *cpp* must be set:

```
mk_shobj -f cpp -l syslnk.example1 -m example1 libexample1.so
```

Use of Java from a C application

The next example illustrates the use of the Java invocation API (part of the JNI) for calling Java programs from C. The example we have chosen is consciously kept simple.

A Java echo program will output all its arguments to standard output. This Java program will then be called from a C program.

Implementation of the Java code

In the file *Echo.java* we define the following class:

```
class Echo {
  public static void main(String[] args)
  {
    for (int i = 0; i < args.length; i++)
    {
      if (i > 0)
        System.out.print(" ");
      System.out.print(args[i]);
    }
    System.out.println("");
  }
}
```

Compiling the Java code

The above defined Java class can now be simply compiled using the command

```
javac Echo.java
```

By calling

```
java Echo This is a test
```

you can prove to yourself that the program is working.

Implementation of the C code

In our example, the following C program will call the above Java program and at the same time transfer its call arguments to it. Once again you should note that all strings transferred to JNI functions must be coded in ASCII. This example will therefore be implemented and produced completely in ASCII mode.

The file *Echo.c* is implemented as follows:

```
#include <jni.h>
int
main(int argc, char *argv[])
{
   JavaVMInitArgs vm_args;
   JavaVMOption options[1];
   JavaVM *jvm;
   JNIEnv *env;
```

```
jint res;
jclass cls;
jmethodID mid;
jobjectArray args;
int i;
/*
** Prepare VM Options
*/
options[0].optionString = "-Djava.class.path=.";
/*
** Prepare VM configuration
*/
vm_args.version = JNI_VERSION_1_4;
vm_args.nOptions = 1;
vm_args.options = options;
vm_args.ignoreUnrecognized = JNI_FALSE;
/*
** Create the Java VM
*/
res = JNI_CreateJavaVM(&jvm,(void **)&env,&vm_args);
if (res < 0)
{
  fprintf(stderr,"Can't create Java VM\n");
  exit(1);
}
/*
** Get class Echo
*/
cls = (*env)->FindClass(env,"Echo");
if (cls == NULL)
{
  fprintf(stderr,"Can't find Echo class\n");
  exit(1);
}
/*
** Get main method

*/
mid = (*env)->GetStaticMethodID(env,cls,"main",
            "([Ljava/lang/String;)V");
if (mid == 0)
{
  fprintf(stderr,"Can't find main in Echo\n");
  exit(1);
}
/*
** Allocate argument array
*/
args = (*env)->NewObjectArray(env,argc-1,
          (*env)->FindClass(env,"java/lang/String"),NULL);
if (args == 0)
{
  fprintf(stderr,"Out of memory\n");
  exit(1);
}
/*
** Prepare arguments
*/
for (i=1; i<argc; i++)
```

```
   {
     jstring jstr;
     jstr = (*env)->NewStringUTF(env,argv[i]);
     if (jstr == NULL)
   {
      fprintf(stderr,"Out of memory\n");
      exit(1);
   }
     (*env)->SetObjectArrayElement(env,args,i-1,jstr);
   }
   /*
   ** Call Java method
   */
   (*env)->CallStaticVoidMethod(env,cls,mid,args);
   /*
   ** Destroy Java VM
   */
   (*jvm)->DestroyJavaVM(jvm);
   return 0;
}
```

The program functions in this form only with a standard JENV installation. If you want to run the program with a private installation, you must set the *JAVA_HOME* environment variable accordingly (see chapter "Environment variables").

Compilation of the C source code

The C source code implemented in the section above must now be compiled using the correct compiler options. For *Echo.c* , in addition to the standard options which have been described in detail above, the ASCII mode must also be considered:

```
cc -c -I/opt/java/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited \
      -Kliteral_encoding_ascii
      -Kno_integer_overflow Echo.c
```

This results in an object file being made available.

Linking and executing the application

When linking the application, it must be remembered that the Java runtime adapter is linked and not the “normal“ runtime systems.

The application can be statically linked with the following commands:

```
export BLSLIB00='$.SYSLNK.JENV.170.GREEN-JAVA'
cc -Kno_link_stdlibs -B llm4 -o Echo \
      Echo.o -l BLSLIB
```

The program can be called like any other POSIX program. However, for JENV to execute, it must be installed under the default installation path. To use a JENV installed elsewhere, you must set the *JAVA_HOME* environment variable accordingly (see chapter "Environment variables").

The call using

```
Echo This is a Java echo
```

produces the expected output:

```
This is a Java echo
```

The program is run using the default VM described in section "Options for selecting the HotSpot™ VM type" in section "java". By selecting the environment variable *JENV_VMTYPE* beforehand you can determine the VM type explicitly. For example:

```
export JENV_VMTYPE=client
```

This results in the HotSpot™ client VM being used.

Use of Java from a C++ application

The differences as compared to using Java from a C application (see section "Use of Java from a C application") are listed below.

Implementation of the C++ code

Let us assume that the *Echo.cpp* file is implemented as follows:

```
#include <jni.h>
int
main(int argc, char *argv[])
{
   JavaVMInitArgs vm_args;
   JavaVMOption options[1];
   JavaVM *jvm;
   JNIEnv *env;
   jint res;
   jclass cls;
   jmethodID mid;
   jobjectArray args;
   int i;
   /*
   ** Prepare VM Options
   */
   options[0].optionString = "-Djava.class.path=.";
   /*
   ** Prepare VM configuration
   */
   vm_args.version = JNI_VERSION_1_4;
   vm_args.nOptions = 2;
   vm_args.options = options;
   vm_args.ignoreUnrecognized = JNI_FALSE;
   /*
   ** Create the Java VM
   */
   res = JNI_CreateJavaVM(&jvm,(void **)&env,&vm_args);
   if (res < 0)
   {
     fprintf(stderr,"Can't create Java VM\n");
     exit(1);
   }
   /*

   ** Get class Echo
   */
   cls = env->FindClass("Echo");
   if (cls == NULL)
   {
     fprintf(stderr,"Can't find Echo class\n");
     exit(1);
   }
   /*
   ** Get main method
   */
   mid = env->GetStaticMethodID(cls,"main",
               "([Ljava/lang/String;)V");
```

```
   if (mid == 0)
   {
     fprintf(stderr,"Can't find main in Echo\n");
     exit(1);
   }
   /*
   ** Allocate argument array
   */
   args = env->NewObjectArray(argc-1,
            env->FindClass("java/lang/String"),NULL);
   if (args == 0)
   {
     fprintf(stderr,"Out of memory\n");
     exit(1);
   }
   /*
   ** Prepare arguments
   */
   for (i=1; i<argc; i++)
   {
     jstring jstr;
     jstr = env->NewStringUTF(argv[i]);
     if (jstr == NULL)
   {
      fprintf(stderr,"Out of memory\n");
      exit(1);
   }
   env->SetObjectArrayElement(args,i-1,jstr);
   }
   /*
   ** Call Java method
   */
   env->CallStaticVoidMethod(cls,mid,args);
   /*
   ** Destroy Java VM
   */
   (*jvm)->DestroyJavaVM(jvm);
   return 0;
}
```

Compiling the C++ source

You must now compile the above C++ source using the CC command and the correct compiler options. For *Echo.cpp*, you must also take ASCII mode into account in addition to the default options described above. This example generates an application that can be executed with the X86 variant of JENV on SQ systems:

```
CC -c -I<installation-path>/include \
      -Kllm_keep,llm_case_lower \
      -Kworkspace_stack,c_names_unlimited \
      -Kliteral_encoding_ascii -Kno_integer_overflow
      -D_SNI_THREAD_SUPPORT Echo.cpp
```

Linking and executing the application

When you link the application, you must remember that the X86 runtime adapter of Java is linked and not the “normal” runtime systems.

You can link the application with the following commands:

```
export BLSLIB00='$.SKULNK.JENV.170.GREEN-JAVA'
CC -Kno_link_stdlibs -B llm4 -o Echo \
      Echo.o -l BLSLIB
```

The program can be called like any other POSIX program on an SQ system. However, for it to run, a X86 variant of JENV must be installed under the default installation path. To use a JENV installed elsewhere, you must set the JAVA_HOME environment variable accordingly (see chapter "Environment variables").

The call using

```
Echo This is a Java echo
```

produces the expected output:

```
This is a Java echo
```

The program is executed with the default VM described in "Options for selecting the HotSpot™ VM type" in section "java". The VM type can be specified explicitly by setting the *JENV_VMTYPE* environment variable beforehand. For example:

```
export JENV_VMTYPE=client
Echo This is a Java echo
```

This causes the HotSpot™ Client-VM to be used for execution.

JCI - Invocation API for COBOL

The Java-COBOL Interface (JCI) is a collection of functions and COBOL-COPY elements to permit simpler operation of the interfaces of the *Java Invocation API* from COBOL programs.

The *Java Invocation API* is part of the *Java Native Interface* (JNI). As it is designed for th language C/C++, its interfaces are inconvenient to operate directly from COBOL programs.

The JCI supports the following functions:

- Starting a Java VM
- Loading classes
- Calling methods
- Generating and editing Java objects
- Checking whether an exception has been generated
- Terminating a Java VM

The option of creating and calling native COBOL methods is not supported.

Compiling the COBOL source codes

A COBOL2000 compiler Version V1.4A or higher is required to compile a COBOL source code which uses JCI interfaces.

Assigning the JCI-COPY library

The JCI-COPY elements are contained in the POSIX directory <*installationpath*>`/include`. Here the path under which JENV was installed must be used for <*installation-path*>. For standard installation this is */opt/java/jdk-17.0.5*. The currently valid name can be found in the Release Notice.

This path must be made known to the compiler under the BS2000 command line interface by means of the S variable `SYSIOL-<libname>` or `SYSIOL-COBLIB`:

`DECL-VAR SYSIOL-COBLIB,INIT='*POSIX( <`*Installations-Pfad*`>/include)',SCOPE=*TASK`

For details, see “COBOL2000 (BS2000) User Manual” [5].

Under POSIX, the environment variable `<libname>` or `COBLIB` must be set:

`export COBLIB=.:` <*Installations-Pfad*>`/include`

For details, see “COBOL2000 (BS2000) User Manual” [5].

Required options/directives

As data structures which contain pointers are used at the interface to JCI functions, the option below is required when the COBOL program is compiled:

```
SOURCE-PROPERTIES=*PAR(STANDARD-DEVIATION=*YES,...)
```

Under POSIX, this corresponds to the option:

```
-C PERMIT-STANDARD-DEVIATION=YES
```

All JCI functions return an integral return value according to ILCS conventions (i.e. in general register R1). To enable this value to be used in the COBOL program, it must be made available in the COBOL special register RETURN-CODE after it has been returned. You can do this as follows:

- Specification of the option

  ```
  SOURCE-PROPERTIES=*PAR(RETURN-CODE=*FROM-ALL-SUBPROGRAMS,...)
  ```

- or under POSIX

  ```
  -C ACTIVATE-XPG4-RETURNCODE=YES
  ```

- or on a targeted basis in the source program by specifying the directive

  ```
  >>CALL-CONVENTION ILCS-SET-RETURN-CODE
  ```

The options apply for the entire source program, the directive only until a `>>CALL-CONVENTION` directive with a different value is specified, see “COBOL2000 (BS2000) Reference Manual” [6]).

The module generated must be available in LLM format. When compilation takes place under the BS2000 command line interface, the option below is required for this purpose:

```
COMPILER-ACTION=*MODULE-GENERATION(MODULE-FORMAT=*LLM,...)
```

When compilation takes place under POSIX, no corresponding options exists as an LLM is always generated there.

Linking COBOL applications with Java

The JCI functions are provided in two PLAM libraries:

SYSLNK.JENV.170.GREEN-JAVA (for the *S390* variant),

SKULNK.JENV.170.GREEN-JAVA (for the *X86* variant)

In addition, these libraries contain the JNI functions called by the JCI, the thread-safe C/C++ runtime system, and the complete COBOL runtime system, the latter always in *S390* format.

External references from applications which call JCI functions must be resolved with priority from one of these libraries.

Under POSIX, the environment variable `BLSLIB00` must be assigned to do this:

```
export BLSLIB00='$.SYSLNK.JENV.170.GREEN-JAVA'
cobol -g -M <PROG-ID> -o <program> <objekte> -l BLSLIB
```

The `cobol` command implicitly links the POSIX linkage option. If linkage is not carried out under the shell using the `cobol` command, but under the BS2000 command line interface using BINDER, this option must be linked from *$.SYSLNK.CRTE.POSIX*.

Processing COBOL applications with Java

Before an application which calls JCI functions is started from the BS2000 command line interface, the POSIX environment must be initialized for the run with the *INITIALIZE* procedures (see "INITIALIZE procedure").

The COBOL runtime system then mainly behaves as if it had been started under the POSIX shell (see “ COBOL2000 (BS2000) User Manual” [5] and section "Special considerations").

After the application has terminated, the POSIX environment must on all accounts be reset by calling the *DELETE* procedure. Otherwise the environment is set incorrectly for further compilations runs.

Characters and strings

Alphanumeric and national strings are transferred to JCI functions in structures which contain a length field in addition to the data area.

Example:

```
01 ANUM.
05 ANUM-LEN  PIC S9(9) COMP-5 VALUE 10.
05 ANUM-TEXT PIC X(10) VALUE "ANUM-TEXT".
01 NAT.
05 NAT-LEN  PIC S9(9) COMP-5 VALUE 10.
05 NAT-TEXT PIC N(10) VALUE N"NAT-TEXT".
```

In this chapter, such structures in the formats are referred to as `Cobvar` or `CobNvar`.

Whether blanks at the end of the text area are ignored or retained depends on the function called. In some functions this behavior can be controlled by means of an additional parameter.

Alphanumeric characters and COBOL strings have an EBCDIC representation, while the Java VM expects or supplies a UTF representation (depending on the interface, UTF8 or UTF16). Necessary conversions are performed automatically in the JCI functions. For this purpose, it must be possible to represent all characters in EDF03IRV. National characters (strings) (UTF16 representation) must be used for characters (strings) for which no such representation exists, otherwise the result is undefined. National strings must also be used for strings which contain binary zeros. Only convertible characters may be used for interfaces for which no national strings are defined (e.g. class and method names).

Java strings are available as objects. Conversion between Java strings and COBOL strings takes place automatically in the JCI functions.

Conversion consists of two partial steps:

- Conversion between EBCDIC strings and UTF8 strings (for alphanumeric strings only).
- Conversion between UTF8 and UTF16 strings and objects.

If an error occurs in any of these conversions (e.g. lack of memory), the condition variable `ResErrCode` (COPY element *JCI-METHODRES*) is set to the value `RES-ERR-NOMEM` (error in the first step) or `RES-ERR-OBJECT` (error in the second step).

If the length field of the COBOL structure is equal to 0 before the conversion, the text area remains unchanged when a Java string is converted to a COBOL string. In the case of conversion in the other direction, an object is created for a null string. If the length field is less than 0 before the conversion, the condition variable `ResErrCode` is set to `RES-ERR-LENGTH`.

Floating point numbers

The Java floating point types *float* and *double* are represented in IEEE format, while the COBOL floating point types *COMP-1* and *COMP-2* are represented in /390 format. The conversion is performed automatically in the JCI functions.

During conversion, the following exceptional situations can occur, which are displayed to the caller as a condition variable in the field `ResErrCode` (COPY element *JCI-METHODRES*) when returning from the JCI function:

- COMP-1 ---> IEEE:

  `RES-ERR-FLOAT-UNDERFLOW`

  The /390 floating point number is lower than the smallest representable IEEE floating point number.

  `RES-ERR-FLOAT-OVERFLOW`

  The /390 floating point number is greater than the largest representable IEEE floating point number.

- COMP-2 ---> IEEE:

  (none)

- IEEE ---> COMP-1:

  `RES-ERR-FLOAT-INVALID`

  The IEEE floating point number equals *NaN* or *infinity*.

- IEEE ---> COMP-2:

  `RES-ERR-FLOAT-UNDERFLOW`

  The IEEE floating point number.is less than the smallest representable /390 floating point number.

  `RES-ERR-FLOAT-OVERFLOW`

  The IEEE floating point number.is greater than the largest representable /390 floating point number.

  `RES-ERR-FLOAT-INVALID`

  The IEEE floating point number equals *NaN* or *infinity*.

If bit positions are lost in the conversion, this does not lead to an exceptional situation.

Object references

Java objects are transferred to the COBOL program as local object references.

To prevent the *Garbage Collector* from removing the referenced objects, the VM registers all the transferred references.

The references are valid until a native method returns to Java. However, this is never the case with a COBOL main program. To release the resources required for the registration and to enable the *Garbage Collector* to remove the objects referenced by the object references, the references must therefore be released explicitly by the COBOL program (see section "Object references").

For object references, the TYPEDEF `JCI-object` is defined in the COPY element *JCI-TYPEDEF*.

Java handle

Some JCI functions use parameters with an opaque data type. These are referred to as Java handles in the formats.

In the COPY element *JCI-TYPEDEF*, the TYPEDEF `JCI-handle` is defined for Java handles.

Return code in special register RETURN-CODE

All JCI functions are `int` functions which return either a truth value or an error indicator in the special register RETURN-CODE.

A separate parameter is used to return other values. Unless described differently, the content of the result field referenced by this parameter is undefined in the event of an error.

Arguments and event values of Java methods

Structures are used to transfer arguments and result values between the COBOL program and JCI functions which call Java methods or edit Java data fields. These must contain all the necessary information. The structures are defined in the COPY elements *JCI-METHODARGS* and *JCI-METHODRES* (see sections "JCI-METHODARGS - Function arguments" and "JCI-METHODRES - Function result"). In this chapter they are referred to as `MethodArg` or `MethodRes`.

If nothing else is defined in the function descriptions, the following prerequisites apply for calling functions which reference an argument of the type `MethodArg` or `MethodRes`:

- Arguments

  Before the function is called, the `CallArgNum` field must contain the number of arguments to be transferred.

  For each argument, an element of the `CallArg` table must be supplied with values in the structure.

  In the `ArgType` field, the condition name `ARG-...` which corresponds to the COBOL data type must be set.The address of a `Cobvar` or `CobNvar` structure must be specified for strings. If trailing spaces are to be ignored, `IGNORE-TRAILING-SPACES` must also be set. Other arguments must be transferred directly into the structure.

- Result values

  For result values, the condition name which corresponds to the COBOL data type `RES-...` must be set in the `ResType` field, or `RES-VOID` if no return value is expected. If a string is expected as the return value, the address of a `Cobvar` or `CobNvar` structure must be specified with a maximum length for the data area in the `ResValAddr` field. If the length <=0, the result value is not transferred.

  After returning from the function, a `Cobvar` or `CobNvar` structure referenced as a return value contains the number of transferred characters (maximum entry value) in the length field, and the transferred characters in the data area. For other data types, the return value is transferred directly into the structure.

If an exceptional situation occurred during the conversion of a floating point data field or string object, the `ResErrCode` field contains the corresponding error code after returning from the function. This can be inquired by means of the condition variable `RES-ERR-<condition>` (see sections "Characters and strings" and "Floating point numbers"). If an argument was incorrect (RETURN-CODE RET-ERR-EARGUMENT), the `ResErrIndex` field contains the number of the argument.

The table below provides an overview of the definitions and the corresponding COBOL and Java types. For COBOL types whose name begins with '`JCI-`', a type definition exists in the COPY element *JCI-TYPEDEF.* A '*' in the last column specifies that automatic conversion of the argument or result value will take place in the JCI functions.

Java type	**COBOL type or TYPEDEF**	**Variable** `ResVal` **...,** `ArgVal` **...**	**Condition name** `RES-` **...,** `ARG-` **...**	
String	Structure `CobVar` Structure `CobNVar`	Addr	ANUM-STRING NAT-STRING	*
byte	JCI-byte	Byte	BYTE	
char	PIC X	Achar	ANUM-CHAR	*
char	PIC N	Nchar	NAT-CHAR	

boolean	JCI-boolean	Boolean	BOOLEAN	
short	JCI-short	Short	SHORT	
int	JCI-int	Int	INT	
long	JCI-long	Long	LONG	
float	USAGE COMP-1	Float	FLOAT	*
double	USAGE COMP-2	Double	DOUBLE	*
Java object (also string object)	JCI-object	Object	OBJECT	

Exceptions

Exceptions can be triggered both by the JCI functions and explicitly by a Java method. This can generally not be recognized from the function's return value.

JCI functions are available to inquire the existence of an exception, have information output, and to remove the exception (see section "Exceptions").

When an exception has been triggered, it must first be removed by calling `JCI_ExceptionClear` before the error-free execution of further JCI functions is guaranteed.

COPY elements

COPY elements are made available for general definitions and structures.

These are contained in the POSIX directory `include` beneath the path under which JENV was installed.

JCI-CONST - Definition of constants

This element defines the COBOL partial structure `JCI-Const` which contains all constants which are relevant to the JCI as data fields:

```
*> Copyright (c) 2016 Fujitsu Technology Solutions GmbH
*>               All Rights Reserved
>>SOURCE FORMAT IS FREE
41 JCI-Const.
42 JCI-Versions.
43  JCI-INTERFACE-VERSION   PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
43  JCI-VERSION-1           PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
42 JCI-ReturnValues.
*> success
43  JCI-RET-OK         PIC S9(009) USAGE COMP-5 SYNC VALUE 000.
*> truth-value false (from test-functions)
43  JCI-RET-FALSE      PIC S9(009) USAGE COMP-5 SYNC VALUE 000.
*> truth-value true (from test-functions)
43  JCI-RET-TRUE       PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
*> unspecific error
43  JCI-RET-ERR        PIC S9(009) USAGE COMP-5 SYNC VALUE 010.
*> VM not created
43  JCI-RET-ENOVM      PIC S9(009) USAGE COMP-5 SYNC VALUE 011.
*> class/method/... not found
43  JCI-RET-ENOTFOUND  PIC S9(009) USAGE COMP-5 SYNC VALUE 012.
*> JCI-NULL object not allowed
43  JCI-RET-ENULLOBJ   PIC S9(009) USAGE COMP-5 SYNC VALUE 013.
*> JCI-NULL method-id/field-id not allowed
43  JCI-RET-ENULLID    PIC S9(009) USAGE COMP-5 SYNC VALUE 014.
*> array-index out of bounds
43  JCI-RET-EINDAOB    PIC S9(009) USAGE COMP-5 SYNC VALUE 015.
*> invalid argument
43  JCI-RET-EARGUMENT  PIC S9(009) USAGE COMP-5 SYNC VALUE 016.
*> not enough memory
43  JCI-RET-ENOMEM     PIC S9(009) USAGE COMP-5 SYNC VALUE 017.
*> VM already created
43  JCI-RET-EEXIST     PIC S9(009) USAGE COMP-5 SYNC VALUE 020.
*> invalid version in option structure
43  JCI-RET-EOPTVERS   PIC S9(009) USAGE COMP-5 SYNC VALUE 021.
*> invalid option number
43  JCI-RET-OPTNUM     PIC S9(009) USAGE COMP-5 SYNC VALUE 022.
*> invalid version in argument structure
43  JCI-RET-EARGVERS   PIC S9(009) USAGE COMP-5 SYNC VALUE 101.
*> invalid version in result structure
43  JCI-RET-ERESVERS   PIC S9(009) USAGE COMP-5 SYNC VALUE 102.
*> invalid argument number
43  JCI-RET-EARGNUM    PIC S9(009) USAGE COMP-5 SYNC VALUE 103.
*> invalid argument-type
43  JCI-RET-EARGTYPE   PIC S9(009) USAGE COMP-5 SYNC VALUE 110.
*> invalid result-type
43  JCI-RET-ERESTYPE   PIC S9(009) USAGE COMP-5 SYNC VALUE 111.
*> argument conversion error
43  JCI-RET-EARGCONV   PIC S9(009) USAGE COMP-5 SYNC VALUE 112.
*> result conversion error
43  JCI-RET-ERESCONV   PIC S9(009) USAGE COMP-5 SYNC VALUE 113.
*> pending exception after method-call
43  JCI-RET-EEXCEPT    PIC S9(009) USAGE COMP-5 SYNC VALUE 120.
42 JCI-Values.
43  JCI-NULL PIC S9(009) USAGE COMP-5 SYNC VALUE 000.
42 JCI-BooleanValues.
43  JCI-FALSE PIC X(001) VALUE X'00'.
43  JCI-TRUE  PIC X(001) VALUE X'01'.
```

JCI-TYPEDEFS - Type definitions

This element contains all elementary type definitions which are relevant to the JCI:

```
*> Copyright (c) 2016 Fujitsu Technology Solutions GmbH
*>               All Rights Reserved
>>SOURCE FORMAT IS FREE
01 JCI-short    TYPEDEF PIC S9(004) USAGE COMP-5.
01 JCI-int      TYPEDEF PIC S9(009) USAGE COMP-5.
01 JCI-long     TYPEDEF PIC S9(018) USAGE COMP-5.
01 JCI-size     TYPEDEF TYPE JCI-int.
01 JCI-object   TYPEDEF PIC S9(009) USAGE COMP-5.
01 JCI-handle   TYPEDEF PIC S9(009) USAGE COMP-5.
01 JCI-byte     TYPEDEF PIC X(001).
01 JCI-boolean  TYPEDEF PIC X(001).
```

JCI-VMOPT - Structure for transferring options

This element contains the partial structure `JCI-VMopt` which is required to transfer options when the VM is started:

```
*> Copyright (c) 2016 Fujitsu Technology Solutions GmbH
*>               All Rights Reserved
>>SOURCE FORMAT IS FREE
41 JCI-VMopt.
42            PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
42            PIC S9(004) USAGE COMP-5 SYNC

VALUE <max-options>.
42 VMOptNum   PIC S9(004) USAGE COMP-5 SYNC VALUE 000.
42 VMOptFlag  PIC X(001) VALUE X'00'.
88 IGNORE-UNRECOGNIZED VALUE X'01'
WHEN SET TO FALSE X'00'.
42            PIC X(003) VALUE X'00'.
42 VMOpt OCCURS <max-options>.
43 VMOptVstring USAGE POINTER.
43 VMExtrainf   USAGE PROGRAM-POINTER.
```

This structure is referred to as `OptArg` below.

Then umber of elements with which the options table is to be expanded (maximum number of arguments) must be set by the REPLACING entry in the COPY statement:

`COPY JCI-VMOPT REPLACING == <max-options> == BY` *num* `.`

The following statement is required for dynamic initialization of the structure as a whole in order to ensure the correct values for fields which are reserved internally:

```
INITIALIZE JCI-VMopt WITH FILLER ALL TO VALUE
```

JCI-METHODARGS - Function arguments

This element contains the partial structure `JCI-MethodArgs` required for transferring arguments:

```
*> Copyright (c) 2016 Fujitsu Technology Solutions GmbH
*>            All Rights Reserved
>>SOURCE FORMAT IS FREE
41 JCI-MethodArgs.
42                        USAGE COMP-2 SYNC VALUE 000.
42            PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
42            PIC S9(004) USAGE COMP-5 SYNC
VALUE <max-arguments>.
42 CallArgNum PIC S9(004) USAGE COMP-5 SYNC VALUE 000.
42 CallArg OCCURS <max-arguments>.
43 ArgType     PIC X(001) VALUE X'00'.
88 ARG-BYTE        VALUE X'01'.
88 ARG-ANUM-CHAR   VALUE X'02'.
88 ARG-NAT-CHAR    VALUE X'03'.
88 ARG-DOUBLE      VALUE X'04'.
88 ARG-FLOAT       VALUE X'05'.
88 ARG-LONG        VALUE X'06'.
88 ARG-INT         VALUE X'07'.
88 ARG-SHORT       VALUE X'08'.
88 ARG-BOOLEAN     VALUE X'09'.
88 ARG-ANUM-STRING VALUE X'0A'.
88 ARG-NAT-STRING  VALUE X'0B'.
88 ARG-OBJECT      VALUE X'0C'.
43 ArgInd      PIC X(001) VALUE X'00'.
*> Indicator for Strings
88 IGNORE-TRAILING-SPACES VALUE X'01'
WHEN SET TO FALSE X'00'.
43             PIC X(002) VALUE ALL X'00'.
43 ArgValAddr    USAGE POINTER.
43 ArgValDouble  USAGE COMP-2 SYNC VALUE 0.
43 ArgValFloat   REDEFINES ArgValDouble             USAGE COMP-1.
43 ArgValLong    REDEFINES ArgValDouble PIC S9(018) USAGE COMP-5.
43 ArgValInt     REDEFINES ArgValDouble PIC S9(009) USAGE COMP-5.
43 ArgValShort   REDEFINES ArgValDouble PIC S9(004) USAGE COMP-5.
43 ArgValObject  REDEFINES ArgValDouble PIC S9(009) USAGE COMP-5.
43 ArgValBoolean REDEFINES ArgValDouble PIC X(001).
43 ArgValByte    REDEFINES ArgValDouble PIC X(001).
43 ArgValAchar   REDEFINES ArgValDouble PIC X(001).
43 ArgValNchar   REDEFINES ArgValDouble PIC N(001).
```

The number of elements to be used to expand the argument table (maximum number of arguments) must be set by means of the REPLACING entry in the COPY statement:

`COPY JCI-METHODARGS REPLACING == <max-arguments> == BY` *num* `.`

The following statement is required for dynamic initialization of the structure as a whole in order to ensure the correct values for both reserved fields and for the table elements:

```
INITIALIZE JCI-MethodArgs WITH FILLER ALL TO VALUE
    THEN REPLACING ALPHANUMERIC BY ALL X'00'
    THEN TO DEFAULT
```

JCI-METHODRES - Function result

This element contains the partial structure `JCI-MethodRes` required for transferring result values and error information:

```
*> Copyright (c) 2016 Fujitsu Technology Solutions GmbH
*>              All Rights Reserved
>>SOURCE FORMAT IS FREE
41 JCI-MethodRes.
42                          USAGE COMP-2 SYNC VALUE 000.
42             PIC S9(009) USAGE COMP-5 SYNC VALUE 001.
*> index to argument/table-element that caused a conversion-error
42 ResErrIndex PIC S9(009) USAGE COMP-5 SYNC VALUE 000.
*> additional information for function return-code
*> JCI-RET-EARGCONV and JCI-RET-ERESCONV
42 ResErrCode  PIC S9(004) USAGE COMP-5 SYNC VALUE 000.
*> no error
88 RES-NO-ERROR            VALUE 000.
*> not enough memory to create/convert data
88 RES-ERR-NOMEM           VALUE 001.
*> object conversion error (object <-> string)
88 RES-ERR-OBJECT          VALUE 010.
*> floating-point conversion-errors (S390 <-> IEEE)
88 RES-ERR-FLOAT-UNDERFLOW VALUE 020.
88 RES-ERR-FLOAT-OVERFLOW  VALUE 021.
88 RES-ERR-FLOAT-INVALID   VALUE 022.
42               PIC X(006) VALUE ALL X'00'.
42 ResultValue.
43 ResType     PIC X(001) VALUE X'00'.
88 RES-VOID        VALUE X'00'.
88 RES-BYTE        VALUE X'01'.
88 RES-ANUM-CHAR   VALUE X'02'.
88 RES-NAT-CHAR    VALUE X'03'.
88 RES-DOUBLE      VALUE X'04'.
88 RES-FLOAT       VALUE X'05'.
88 RES-LONG        VALUE X'06'.
88 RES-INT         VALUE X'07'.
88 RES-SHORT       VALUE X'08'.
88 RES-BOOLEAN     VALUE X'09'.

88 RES-ANUM-STRING VALUE X'0A'.
88 RES-NAT-STRING  VALUE X'0B'.
88 RES-OBJECT      VALUE X'0C'.
43             PIC X(003) VALUE ALL X'00'.
43 ResValAddr    USAGE POINTER.
43 ResValDouble  USAGE COMP-2 SYNC VALUE 0.
43 ResValFloat   REDEFINES ResValDouble            USAGE COMP-1.
43 ResValLong    REDEFINES ResValDouble PIC S9(018) USAGE COMP-5.
43 ResValInt     REDEFINES ResValDouble PIC S9(009) USAGE COMP-5.
43 ResValShort   REDEFINES ResValDouble PIC S9(004) USAGE COMP-5.
43 ResValObject  REDEFINES ResValDouble PIC S9(009) USAGE COMP-5.
43 ResValBoolean REDEFINES ResValDouble PIC X(001).
43 ResValByte    REDEFINES ResValDouble PIC X(001).
43 ResValAchar   REDEFINES ResValDouble PIC X(001).
43 ResValNchar   REDEFINES ResValDouble PIC N(001).
```

The following statement is required for dynamic initialization of the structure as a whole in order to ensure the correct values for fields which are reserved internally:

```
INITIALIZE JCI-MethodRes WITH FILLER ALL TO VALUE
```

Functions

The interfaces of the JCI functions are described according to aspects relating to content in this section.

For simplicity’s sake, object references are mainly referred to as objects in the formats.

Class object refers to a reference to an object of the class *java.lang.Class*.

Starting and terminating the Java VM

This section describes the JCI functions which are required to start and terminate the Java VM.

- JCI_CreateJavaVM
- JCI_DestroyJavaVM

JCI_CreateJavaVM

This function generates, i.e. loads and initializes, the Java VM.
It is equivalent to the JNI function `JNI_CreateJavaVM`.

Call

```
CALL 'JCI_CreateJavaVM' USING opt
```

opt Options for the Java VM

Arguments

opt A structure in the form `OptArg` with the following elements:

`VMOptNum`

The number of VM options; the value may not exceed the value specified for `<max-options>` (see section "JCI-VMOPT - Structure for transferring options").

`VMOptFlag`

Displays whether unknown options are to be ignored (condition name `IGNORE-UNRECOGNIZED`).

`VMOptVstring`

For each option, the address of a `Cobvar` structure.Trailing spaces at the end of the text are ignored

`VMExtrainf`

Depending on the option, the address of an external function.

All options can be specified which are also permissible in the JNI function `JNI_CreateJavaVM`.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-EVERSION`

The statically generated version number in *opt* is invalid (possibly overwritten).

`JCI-RET-EOPTNUM`

The number of options transferred (`VMOptNum`) is less than 0 or greater than the value specified for `<max-options>` (see section "JCI-VMOPT - Structure for transferring options").

`JCI-RET-EEXIST`

A Java VM has already been generated.

`JCI-RET-ENOMEM`

Not enough memory is available to generate the Java VM.

`JCI-RET-ERR`

An error which is not specified in more detail has occurred (e.g. invalid option and `IGNORE-UNRECOGNIZED` not set).

Notes

Only one *JavaVM* can be generated in a program run.

No new Java VM can be generated after terminating the VM with `JCI_DestroyJavaVM`, either.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 OptCP.
05 PIC S9(9) COMP-5 VALUE 30.
05 PIC X(30) VALUE '-Djava.class.path=.'.
01 OptEnc.
05 PIC S9(9) COMP-5 VALUE 40.
05 PIC X(40) VALUE '-Dfile.encoding=OSD_EBCDIC_DF04_15'.
01 JVMOptions.
COPY JCI-VMOPT REPLACING == <max-options> == BY 2.
...
PROCEDURE DIVISION.
*>
*> Prepare VM options
*>
MOVE 2 TO VMOptNum.
SET IGNORE-UNRECOGNIZED TO FALSE.
SET VMOptVstring(1) TO ADDRESS OF OptCP
SET VMOptVstring(2) TO ADDRESS OF OptEnc
*>
*> Create the Java VM
*>
CALL 'JCI_CreateJavaVM' USING JVMOptions
IF RETURN-CODE NOT = JCI-RET-OK
...
```

JCI_DestroyJavaVM

This function releases resources of the Java VM.
It is equivalent to the JNI function `JNI_DestroyJavaVM`.

Call

```
CALL 'JCI_DestroyJavaVM'
```

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-ERR`

An error which is not specified in more detail has occurred.

Notes

- The function should not be called if the call of the function `JCI_CreateJavaVM` was not successful.
- After the function has been executed correctly, no further JCI functions can be called.
- The Java VM is not unloaded.
- It is not possible to reboot the Java VM with `JCI_CreateJavaVM`.

Classes and methods

This section describes the JCI functions which are required to load classes and call methods.

- JCI_FindClass
- JCI_GetStaticMethodID
- JCI_CallStaticMethod
- JCI_GetMethodID
- JCI_CallMethod
- JCI_CallNonvirtualMethod

JCI_FindClass

This function localizes and loads a class.
It is equivalent to the JNI function `FindClass`.

Call

`CALL 'JCI_FindClass' USING` *cName cObj*

cName Name of the class

cObj Class object returned by the function

Arguments

cName Structure of the type `Cobvar`
Fully qualified name of the class (i.e. a package-name separated by "/" followed by the name and class) which is to be searched for. If the name begins with "[" (array signature character), an array class is returned.
Trailing spaces at the end of the text are ignored.

cObj Data field of the type `JCI-object`
After the function has been successfully executed, the field contains a class object of the class being searched for.
In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENOTFOUND`

The class could not be loaded.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `FindClass`.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 className.
02 PIC S9(9) USAGE COMP-5 VALUE 30.
02 PIC X(30) VALUE 'Hello'.
...
01 classObj TYPE JCI-object.
...
PROCEDURE DIVISION.
...
CALL 'JCI_FindClass' USING className classObj
IF RETURN-CODE NOT = JCI-RET-OK
...
```

JCI_GetStaticMethodID

This function returns the method ID (Java handle) for a static method of a class.It is equivalent to the JNI function `GetStaticMethodID`.

Call

`CALL 'JCI_GetStaticMethodID' USING` *cObj mName mSig mID*

cObj	Class object
mName	Name of the method
mSig	Signature of the method
mID	Method ID returned by the function

Arguments

cObj	Data field of the type `JCI-object` Object of the class in which the method is to be searched for.
mName	Structure of the type `Cobvar` Name of the method which is to be searched for. Trailing spaces at the end of the text are ignored.
mSig	Structure of the type `Cobvar` Signature of the method which is to be searched for. Trailing spaces at the end of the text are ignored.
mID	Data field of the type `JCI-handle` After the function has been successfully executed, the field contains the method ID of the method being searched for. In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-ENOTFOUND`

The method could not be found.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `GetStaticMethodID`.

Notes

The method is identified by the name and the signature. The signature can be received by the statement,

```
javap -s <class-name>
```

the <`class-name`> being the name of the class identified by *cObj*.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 methodName.
05 PIC S9(9) COMP-5 VALUE 30.
05 PIC X(30) VALUE 'hello'.
01 methodSig.
05 PIC S9(9) COMP-5 VALUE 80.
05 PIC X(80) VALUE '(Ljava/lang/String;)V'.
...
01 classObj  TYPE JCI-object.
01 methodID  TYPE JCI-handle.
...
PROCEDURE DIVISION.
...
CALL 'JCI_GetStaticMethodID' USING classObj methodName
methodSig methodID
IF RETURN-CODE NOT = JCI-RET-OK
...
```

JCI_CallStaticMethod

This function calls a static method.
It is equivalent to the JNI functions `CallStatic<type>Method`. However, it also offers the option of transferring or receiving strings directly.

Call

`CALL 'JCI_CallStaticMethod' USING` *cObj mID arg res*

cObj	Class object
mID	Method ID
arg	Method arguments
res	Method result

Arguments

cObj	Data field of the type `JCI-object` Class object whose method is to be called.
mID	Data field of the type `JCI-handle` ID of the method which is to be called. The method ID must be obtained by calling the function `JCI_GetStaticMethodID` for the *cObj* class.
arg	A structure of the form `MethodArg` Description of the arguments for the method call (see section "Arguments and event values of Java methods").
res	A structure of the form `MethodRes` Description of the return value for the method call and error information (see section "Arguments and event values of Java methods"). If the return value of the method is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-ENULLID`

mId is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-EARGVERS`

The statically generated version number in *arg* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGNUM`

The number of arguments transferred (`CallArgNum`) is less than 0 or greater than the value used for `<max-arguments>` (see section "JCI-METHODARGS - Function arguments").

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid. The `ResErrIndex` field contains the number of the faulty argument.

`JCI-RET-ERESTYPE`

The value of the `ResType` field is invalid.

`JCI-RET-EARGCONV`

An error occurred while an argument was being converted.
The `ResErrIndex` field contains the number of the argument, the `ResErrCode` field a more precise error code.

`JCI-RET-ERESCONV`

An error occurred while the result was being converted.
The `ResErrCode` field contains a more precise error code.

`JCI-RET-EEXCEPT`

An exception exists after the method was called. No distinction is made between whether the exception was generated by this or an earlier function call.
The field corresponding to the method result in the *res* structure is unchanged.

Exceptions

All exceptions which were generated by the called method.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
...
01 MethodArgs.
COPY JCI-METHODARGS REPLACING ==<max-arguments>== BY 2.
01 MethodRes.
COPY JCI-METHODRES.
...
01 myName.
05 len PIC S9(9) COMP-5 VALUE 30.
05 txt PIC X(30).
01 classObj  TYPE JCI-object.
01 methodID  TYPE JCI-handle.
PROCEDURE DIVISION.
...
MOVE 1 TO CallArgNum
SET RES-VOID TO TRUE
SET ARG-ANUM-STRING(1) IGNORE-TRAILING-SPACES(1) TO TRUE
SET ArgValAddr(1) TO ADDRESS OF myName
CALL 'JCI_CallStaticMethod' USING classObj methodId
MethodArgs MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
...
```

JCI_GetMethodID

This function returns the method ID (Java handle) for an instance method of a class or interface.
It is equivalent to the JNI function `GetMethodID`.

Call

`CALL 'JCI_GetMethodID' USING` *cObj mName mSig mID*

cObj	Class object
mName	Name of the method
mSig	Signature of the method
mID	Method ID returned by the function

Arguments

See function `JCI_GetStaticMethodID`.

Return value (RETURN-CODE)

See function `JCI_GetStaticMethodID`.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `GetMethodID`.

Notes

The method can be defined in an upper class of the class referenced by *cObj* and be inherited by the latter.

The method is identified by the name and the signature. The signature can be received by the statement,

```
javap -s <class-name>
```

the <`class-name`> being the name of the class identified by *cObj*.

JCI_CallMethod

This function calls an instance method.
It is equivalent to the JNI functions `Call<type>Method`. However, it also offers the option of transferring or receiving strings directly.

Call

`CALL 'JCI_CallMethod' USING` *obj mID arg res*

obj Instance object

mID Method ID

arg Method arguments

res Method result

Arguments

obj Data field of the type `JCI-object`
Instance object for which the method is to be called.

mID Data field of the type `JCI-handle`
ID of the method which is to be called. The method ID must be obtained by calling the `JCI_GetStaticMethodID` function for the class of the object *obj* or one of its upper classes.

arg A structure of the form `MethodArg`
Description of the arguments for the method call (see section "Arguments and event values of Java methods").

res A structure of the form `MethodRes`
Description of the return value for the method call and error information (see section "Arguments and event values of Java methods"). If the return value of the method is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj is `JCI-NULL`.

All other values as in `JCI_CallStaticMethod`.

Exceptions

All exceptions which were generated by the called method.

JCI_CallNonvirtualMethod

This function calls an instance method of a predefined class.
It is equivalent to the JNI functions `CallNonvirtual<type>Method`. However, it also offers the option of transferring or receiving strings directly.

Call

`CALL 'JCI_CallNonvirtualMethod' USING` *obj* `cObj` *mID arg res*

obj Instance object

cObj Object of the class in which the method is defined.

mID Method ID

arg Method arguments

res Method result

Arguments

obj Data field of the type `JCI-object`
Instance object for which the method is to be called.

cObj Data field of the type `JCI-object`
Object of the class whose method is to be called.

mID Data field of the type `JCI-handle`
ID of the method which is to be called.
The method ID must be obtained by calling the function `JCI_GetMethodID` for the *cObj* class. This class must match the class of the *obj* object or of one of its upper classes.

arg A structure of the form `MethodArg`
Description of the arguments for the method call (see section "Arguments and event values of Java methods").

res A structure of the form `MethodRes`
Description of the return value for the method call and error information (see section "Arguments and event values of Java methods"). If the return value of the method is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj or *cObj* is `JCI-NULL`.

All other values as in `JCI_CallStaticMethod`.

Exceptions

All exceptions which were generated by the called method.

Object references

This section describes the JCI functions required to manage local object references.

- JCI_DeleteLocalRef
- JCI_NewLocalRef

JCI_DeleteLocalRef

This function deletes a local object reference.
It is equivalent to the JNI function `DeleteLocalRef`.

Call

```
CALL 'JCI_DeleteLocalRef' USING obj
```

obj Object reference

Arguments

obj Data field of the type `JCI-object`
Object reference which is to be deleted.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

Notes

After the `JCI_DeleteLocalRef` function has been called, the object reference *obj* may no longer be used.

JCI_NewLocalRef

This function generates a new local reference to an object.
It is equivalent to the JNI function `NewLocalRef`.

Call

```
CALL 'JCI_NewLocalRef' USING obj newObj
```

obj	Object reference
newObj	Object reference returned by the function

Arguments

obj	Data field of the type `JCI-object` Object reference to the object to which a new reference is to be generated.
newObj	Data field of the type `JCI-object` New object reference to the object referenced by *obj*.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

Objects

This section describes the JCI functions required to generate and edit Java objects.

- JCI_NewObject
- JCI_GetObjectClass
- JCI_IsInstanceOf
- JCI_IsSameObject

JCI_NewObject

This function generates a new Java object.
It is equivalent to the JNI function `NewObject`. However, it also offers the option of transferring strings directly.

Call

`CALL 'JCI_NewObject' USING` *cObj mID arg res*

cObj	Class object
mID	Method ID
arg	Constructor arguments
res	Result

Arguments

cObj Data field of the type `JCI-object`

Class object for which an object is to be generated. It may not refer to an array class.

mID Data field of the type `JCI-handle`
ID of the constructor method. The method ID must be obtained by calling the function `JCI_GetMethodID` with the name `<init>` and signature `(...)V` for the *cObj* class.

arg A structure of the form `MethodArg`
Description of the arguments for the constructor call (see section "Arguments and event values of Java methods").

res A structure of the form `MethodRes`
Return value (new object) and error information (output only, result in `ResValObject`). In the event of an error, JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-ENULLID`

mId is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object or refers to an array.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-EARGVERS`

The statically generated version number in *arg* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGNUM`

The number of arguments transferred (`CallArgNum`) is less than 0 or greater than the value used for `<max-arguments>` (see section "JCI-METHODARGS - Function arguments").

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid. The `ResErrIndex` field contains the number of the faulty argument.

`JCI-RET-EARGCONV`

An error occurred while an argument was being converted.
The `ResErrIndex` field contains the number of the argument, the `ResErrCode` field a more precise error code.

`JCI-RET-ERR`

The object could not be generated.

Exceptions

All exceptions generated by the constructor.

The other exceptions generated by the function correspond to those of the JNI function `NewObject`.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 className.
02 PIC S9(9) COMP-5 VALUE 30.
02 PIC X(30) VALUE 'myClass'.
01 methodName.
05 PIC S9(9) COMP-5 VALUE 9.
05 PIC X(10) VALUE '<init>'.
01 methodSig.
05 PIC S9(9) COMP-5 VALUE 80.
05 PIC X(80) VALUE
'(Ljava/lang/String;Ljava/lang/String;)V'.
01 nText.
05 PIC S9(9) COMP-5 VALUE 8.
05 PIC N(20) VALUE N'COBOL'.
01 aText.
05 PIC S9(9) COMP-5 VALUE 8.
05 PIC X(20) VALUE 'Java'.
01 classObj  TYPE JCI-object.
01 instanceObj TYPE JCI-object.
01 methodID  TYPE JCI-handle.
01 MethodArgs.
COPY JCI-METHODARGS REPLACING ==<max-arguments>== BY 2.
01 MethodRes.
COPY JCI-METHODRES.
...
PROCEDURE DIVISION.
...
CALL 'JCI_FindClass' USING className classObject
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF
CALL 'JCI_GetMethodID' USING classObj methodName
methodSig methodID
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF
MOVE 2 TO CallArgNum
SET ARG-NAT-STRING(1) IGNORE-TRAILING-SPACES(1) TO TRUE
SET ArgValAddr(1) TO ADDRESS OF nText
SET ARG-ANUM-STRING(2) IGNORE-TRAILING-SPACES(2) TO TRUE
SET ArgValAddr(2) TO ADDRESS OF aText

CALL 'JCI_NewObject' USING classObj methodId
MethodArgs MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF
MOVE ResValObject TO instanceObject
...
```

JCI_GetObjectClass

This function returns the class object of an object.
It is equivalent to the JNI function `GetObjectClass`.

Call

```
CALL 'JCI_GetObjectClass' USING obj cObj
```

obj Instance object

cObj Class object returned by the function

Arguments

obj Data field of the type `JCI-object`

Instance object whose class object is to be returned. The object may not be 0.

cObj Data field of the type `JCI-object`
After the function has been successfully executed, the field contains the class object of the class being searched for.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

obj is `JCI-NULL`.

JCI_IsInstanceOf

This function checks whether an object is an instance of a class.
It is equivalent to the JNI function `IsInstanceOf`.

Call

`CALL 'JCI_IsInstanceOf' USING` *obj cObj*

obj Instance object

cObj Class object

Arguments

obj Data field of the type `JCI-object`
Object which is to be checked. If *obj* is `JCI-NULL`, it is an instance of every class.

cObj Data field of the type `JCI-object`
Class which is to be checked for.

Return value (RETURN-CODE)

`JCI-RET-TRUE`

`obj` is an instance of `cObj`.

`JCI-RET-FALSE`

`obj` is not an instance of `cObj`.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

JCI_IsSameObject

This function checks whether two object references refer to the same object.It is equivalent to the JNI function `IsSameObject`.

Call

`CALL 'JCI_IsSameObject' USING` *obj1 obj2*

obj1 Object

obj2 Object

Arguments

obj1, obj2

Data fields of the type `JCI-object`
Objects which are to be compared.

Return value (RETURN-CODE)

`JCI-RET-TRUE`

Both object references refer to the same object or are both `JCI-NULL`.

`JCI-RET-FALSE`

The object references refer to different objects.

`JCI-RET-ENOVM`

No Java VM has been started.

Fields

This section describes the JCI functions which enable fields in Java objects to be edited.

- JCI_GetStaticFieldID
- JCI_GetStaticField
- JCI_SetStaticField
- JCI_GetFieldID
- JCI_GetField
- JCI_SetField

JCI_GetStaticFieldID

This function returns the field ID (Java handle) for a static field of a class.It is equivalent to the JNI function `GetStaticFieldID`.

Call

`CALL 'JCI_GetStaticFieldID' USING` *cObj fName fSig fID*

cObj	Class object
fName	Name of the field
fSig	Signature of the field
fID	Field ID returned by the function

Arguments

cObj	Data field of the type `JCI-object` Object of the class in which the method is to be searched for.
fName	Structure of the type `Cobvar` Name of the field which is to be searched for. Trailing spaces at the end of the text are ignored.
fSig	Structure of the type `Cobvar` Signature of the field which is to be searched for. Trailing spaces at the end of the text are ignored.
fID	Data field of the type `JCI-handle` After the function has been successfully executed, the field contains the field ID of the field being searched for. In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-ENOTFOUND`

The field could not be found.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `GetStaticFieldID`.

Notes

The field is identified by the name and the signature. The signature can be received by the statement,

```
javap -s <class-name>
```

the <`class-name`> being the name of the class identified by *cObj*.

JCI_GetStaticField

This function returns the value of a static field of a class.
It is equivalent to the JNI functions `GetStatic<type>Field`. However, it also offers the option of obtaining strings directly.

Call

`CALL 'JCI_GetStaticField' USING` *cObj fID res*

cObj Class object

fID Field ID

res Result

Arguments

cObj Data field of the type `JCI-object`
Class object whose field content is to be returned.

fID Data field of the type `JCI-handle`
ID of the field whose content is to be returned. The field ID must be obtained by calling the function `JCI_GetStaticFieldID` for the *cObj* class.

res A structure of the form `MethodRes`
Description of the return value for the field content and error information (see section "Arguments and event values of Java methods"). If the content of the field is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-ENULLID`

fID is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-ERESTYPE`

The value of the `ResType` field is invalid.

`JCI-RET-ERESCONV`

An error occurred while the result was being converted.
The `ResErrCode` field contains a more precise error code.

JCI_SetStaticField

This function sets the value of a static field of a class.
It is equivalent to the JNI functions `SetStatic<type>Field`. However, it also offers the option of transferring strings directly.

Call

`CALL 'JCI_SetStaticField' USING` *cObj fID arg res*

cObj	Class object
fID	Field ID
arg	New value
res	Result

Arguments

cObj	Data field of the type `JCI-object` Class object whose field content is to be set.
fID	Data field of the type `JCI-handle` ID of the field whose content is to be set. The field ID must be obtained by calling the function `JCI_GetStaticFieldID` for the *cObj* class.
arg	A structure of the form `MethodArg` Description of the new value for the field content (see section "Arguments and event values of Java methods"). Only the partial structure `CallArg(1)` is required.
res	A structure of the form `MethodRes`Error information (output only).

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLID`

fID is `JCI-NULL`.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-EARGVERS`

The statically generated version number in *arg* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid.

`JCI-RET-EARGCONV`

An error occurred while the argument was being converted.
The `ResErrCode` field contains a more precise error code.

JCI_GetFieldID

This function returns the field ID (Java handle) for an instance field of a class.It is equivalent to the JNI function `GetFieldID`.

Call

`CALL 'JCI_GetFieldID' USING` *cObj fName fSig fID*

cObj	Class object
fName	Name of the field
fSig	Signature of the field
fID	Field ID returned by the function

Arguments

See function JCI_GetStaticFieldID.

Return value (RETURN-CODE)

See function JCI_GetStaticFieldID.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `GetFieldID`.

Notes

See function JCI_GetStaticField.

JCI_GetField

This function returns the value of an instance field of an object.
It is equivalent to the JNI functions `Get<type>Field`. However, it also offers the option of obtaining strings directly.

Call

`CALL 'JCI_GetField' USING` *obj fID res*

obj Instance object

fID Field ID

res Result

Arguments

obj Data field of the type `JCI-object`
Instance object whose field content is to be returned.

fID Data field of the type `JCI-handle`
ID of the field whose content is to be returned. The field ID must be obtained by calling the function `JCI_GetFieldID`.

res A structure of the form `MethodRes`
Description of the return value for the field content and error information (see section "Arguments and event values of Java methods"). If the content of the field is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj is `JCI-NULL`.

All other values as in JCI_GetStaticField.

JCI_SetField

This function sets the value of a static field of a class.
It is equivalent to the JNI functions `Set<type>Field`. However, it also offers the option of transferring strings directly.

Call

`CALL 'JCI_SetField' USING` *obj fID arg res*

obj Instance object

fID Field ID

arg New value

res Result

Arguments

obj Data field of the type `JCI-object`
Instance object whose field content is to be modified.

fID Data field of the type `JCI-handle`
ID of the field whose content is to be set. The field ID must be obtained by calling the function `JCI_GetFieldID`.

arg A structure of the form `MethodArg`
Description of the new value for the field content (see section "Arguments and event values of Java methods").
Only the partial structure `CallArg(1)` is required.

res A structure of the form `MethodRes`
Error information (output only).

Return value (RETURN-CODE)

`JCI-RET-ENULLOBJ`

obj is `JCI-NULL`.

All other values as in JCI_GetStaticField.

Strings

This section describes the JCI functions which enable Java strings to be generated and edited.

- JCI_NewString
- JCI_GetStringLength
- JCI_GetString

JCI_NewString

This function generates a new Java string object from a COBOL string.It is equivalent to the JNI function `NewString`. However, it also offers the option of transferring alphanumeric (EBCDIC) strings directly.

Call

`CALL 'JCI_NewString' USING` *arg res*

arg Argument description

res Result description

Arguments

arg A structure of the form `MethodArg`

Description of the string from which the string object is to be generated (see section "Arguments and event values of Java methods").
Only the partial structure `CallArg(1)` is required.
The only permissible value for `ArgType(1)` is `ARG-ANUM-STRING` or `ARG-NAT-STRING`.

res A structure of the form `MethodRes`
Return value and error information (output only, result in `ResValObject`). In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-EARGVERS`

The statically generated version number in *arg* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid.

`JCI-RET-EARGCONV`

An error occurred while the argument was being converted.
The `ResErrCode` field contains a more precise error code.

`JCI-RET-ERR`

The object could not be generated.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `NewString`.

JCI_GetStringLength

This function returns the length (number of Unicode characters) of a Java string.
It is equivalent to the JNI function `GetStringLength`.

Call

```
CALL 'JCI_GetStringLength' USING sObj len
```

sObj String object

len Length

Arguments

sObj Data field of the type `JCI-object`
String object whose length is to be returned.

len Data field of the type `JCI-size`
After the function has been successfully executed, the field contains the number of Unicode characters in the string object referenced by *sObj*.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

sObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

sObj is not a string object.

JCI_GetString

This function copies part of a Java string to a data area provided.
It is equivalent to the JNI function `GetStringRegion`. However, it also offers the option of obtaining alphanumeric (EBCDIC) strings directly.

Call

```
CALL 'JCI_GetString' USING sObj start res
```

sObj String object

start Start position

res Result description

Arguments

sObj Data field of the type `JCI-object`
String object whose content is to be copied.

start Data field of the type `JCI-size`
Position of the first character which is to be returned (beginning with 1).

res A structure of the form `MethodRes`
Return value and error information (see section "Arguments and event values of Java methods").
The only permissible value for `ResType` is `RES-ANUM-STRING` or `RES-NAT-STRING`.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

sObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

sObj is not a string object.

`JCI-RET-EINDAOB`

start is less than 1 or greater than the number of characters in the Java string.

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-ERESTYPE`

The value of the `ResType` field is invalid.

`JCI-RET-ERESCONV`

An error occurred while the string was being converted.
The `ResErrCode` field contains a more precise error code.

Notes

The maximum length of the transfer (length of Java string – *start* + 1) or of the value *len* equals that of the output structure.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 sObj TYPE JCI-object.
01 sPos PIC S9(9) COMP-5 VALUE 0.
01 sLen PIC S9(9) COMP-5 VALUE 0.
01 aText.
05 alen PIC S9(9) COMP-5 VALUE 80.
05 atxt PIC X(80) VALUE SPACE.
...
01 MethodRes.
COPY JCI-METHODRES.
...
PROCEDURE DIVISION.
...
CALL 'JCI_GetStringLength' USING sObj sLen
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF.
SET RES-ANUM-STRING TO TRUE
SET ResValAddr TO ADDRESS OF aText
MOVE LENGTH OF atxt TO alen
*> loop to output the complete java-string
PERFORM VARYING sPos FROM 1 BY alen UNTIL sPos > sLen
CALL 'JCI_GetString' USING sobj sPos MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF
DISPLAY aTxt(1:aLen) UPON T
END-PERFORM
...
```

Arrays

This section describes the JCI functions which enable Java arrays to be generated and processed.

- JCI_GetArrayLength
- JCI_NewObjectArray
- JCI_GetObjectArrayElement
- JCI_SetObjectArrayElement
- JCI_NewArray
- JCI_GetArray
- JCI_SetArray

JCI_GetArrayLength

This function is equivalent to the JNI function `GetArrayLength`.

Call

```
CALL 'JCI_GetArrayLength' USING aObj num
```

aObj Array object

num Number of elements

Arguments

aObj Data field of the type `JCI-object`
Array object whose number of elements is to be returned.

num Data field of the type `JCI-size`
After the function has been successfully executed, the field contains the number of elements of the array object referenced by *aObj*.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

aObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj is not an array object.

JCI_NewObjectArray

This function generates an array object for object elements.
It is equivalent to the JNI function `NewObjectArray`.

Call

`CALL 'JCI_NewObjectArray' USING` *num cObj eObj res*

num Number of elements

cObj Element class

eObj Element initil value

res Result description

Arguments

num Data field of the type `JCI-size`
Number of elements in the array.

cObj Data field of the type `JCI-object`
Class object for the class of the array elements.

eObj Data field of the type `JCI-object`
Initial value for the array elements (may also be JCI-NULL).

res A structure of the form `MethodRes`
Return value (new object reference) in `ResValObject`. In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENULLOBJ`

cObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

cObj is not a class object.

`JCI-RET-EINDAOB`

num is less than 0.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-ERR`

The array object could not be generated.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `NewObjectArray`.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 className.
05 len PIC S9(9) COMP-5 VALUE 40.
05 txt PIC X(40) VALUE SPACE.
01 classObj TYPE JCI-object.
01 initObj  TYPE JCI-object.
01 arrayObj TYPE JCI-object.
01 numElements PIC S9(9) COMP-5.
...
...
01 MethodRes.
COPY JCI-METHODRES.
...
PROCEDURE DIVISION.
*>
*> Create array of 10 String-elements
*>
MOVE 'java/lang/String' TO txt IN className
CALL 'JCI_FindClass' USING className classId
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF.
MOVE 10 TO numElements
MOVE JCI-NULL TO initObj
CALL 'JCI_NewObjectArray' USING numElements classId
initObj MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF.
MOVE ResValObject TO arrayObj
...
```

JCI_GetObjectArrayElement

This function returns an element of an object array.
It is equivalent to the JNI function `GetObjectArrayElement`. However, it also offers the option of obtaining strings instead of string objects.

Call

`CALL 'JCI_GetObjectArrayElement' USING` *aObj index res*

aObj	Array object
index	Array index
res	Result description

Arguments

aObj	Data field of the type `JCI-object` Array object whose element is to be returned.
index	Data field of the type `JCI-size` Position of the element in the array which is to be returned (beginning with 1).
res	A structure of the form `MethodRes` Return value and error information (see section "Arguments and event values of Java methods"). The only permissible values for `ResType` are `RES-OBJECT`, `RES-ANUM-STRING`, and `RES-NAT-STRING`. If the array element is a NULL object, the length field of the target structure is set to 0 for the types `RES-ANUM-STRING` and `RES-NAT-STRING`, and the text area remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

aObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj is not an array object.

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-ERESTYPE`

The value of the `ResType` field is invalid.

`JCI-RET-ERESCONV`

An error occurred while the element was being converted.
The `ResErrCode` field contains a more precise error code.

`JCI-RET-EINDAOB`

index is less than 1 or greater than the number of elements in the array.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 arrayObj TYPE JCI-object.
01 arrayIndex PIC S9(9) COMP-5.
01 natText.
05 nlen PIC S9(9) COMP-5 VALUE 80.
05 ntxt PIC N(80) VALUE SPACE.
...
01 MethodRes.
COPY JCI-METHODRES.
...
PROCEDURE DIVISION.
MOVE 7 TO arrayIndex
SET RES-NAT-STRING TO TRUE
SET ResValAddr TO ADDRESS OF natText
CALL 'JCI_GetObjectArrayElement' USING
arrayObj arrayIndex MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
...
END-IF.
DISPLAY FUNCTION DISPLAY-OF(ntxt(1:nlen)) UPON T
...
```

JCI_SetObjectArrayElement

This function sets an element of an object array.
It is equivalent to the JNI function `SetObjectArrayElement`. However, it also offers the option of transferring strings instead of string objects.

Call

`CALL 'JCI_SetObjectArrayElement' USING` *aObj index arg res*

aObj	Array object
index	Array index
arg	Argument description
res	Result description

Arguments

aObj	Data field of the type `JCI-object` Array object which is to be modified.
index	Data field of the type `JCI-size` Position of the element in the array which is to be set (beginning with 1).
arg	A structure of the form `MethodArg` Description of the new value for the array element (see section "Arguments and event values of Java methods"). Only the partial structure `CallArg(1)` is required. The only permissible values for `ArgType(1)` are `ARG-OBJECT`, `ARG-ANUM-STRING`, and `ARG-NAT-STRING`.
res	A structure of the form `MethodRes` Error information (output only).

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

aObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj is not an array object.

`JCI-RET-EARGVERS`

The statically generated version number in *elem* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid.

`JCI-RET-EARGCONV`

An error occurred while the argument was being converted.
The `ResErrCode` field contains a more precise error code.

`JCI-RET-EINDAOB`

index is less than 1 or greater than the number of elements in the array.

Exceptions

The exceptions generated by the function correspond to those of the JNI function `SetObjectArrayElement`.

JCI_NewArray

This function generates an array object for non-object elements which is initialized with binary zeros.
It is equivalent to the JNI functions `New<PrimitiveType>Array`.

Call

`CALL 'JCI_NewArray' USING` *num arg res*

num Number of elements

arg Element description

res Result description

Arguments

num Data field of the type `JCI-size`

Number of elements in the array.

arg A structure of the form `MethodArg`
Type description of the array elements.
Only the `ArgType(1)` field is required.
`ArgType(1)` may not be `ARG-OBJECT`, `ARG-ANUM-STRING`, or `ARG-NAT-STRING`.

r A structure of the form `MethodRes`
Return value (new object reference) in `ResValObject`. In the event of an error, the value JCI-NULL is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-EINDAOB`

num is less than 0.

`JCI-RET-EARGVERS`

The statically generated version number in *elem* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid.

`JCI-RET-ERR`

The array object could not be generated.

Exceptions

The exceptions generated by the function correspond to those of the JNI functions `New<PrimitiveType>Array`.

JCI_GetArray

This function copies elements of a Java array to a COBOL table provided.It is equivalent to the JNI functions `Get<PrimitiveType>ArrayRegion`.

Call

`CALL 'JCI_GetArray' USING` *aObj start num res*

aObj Array object

start Start position

num Number

r Result description

Arguments

aObj Data field of the type `JCI-object`
Array object whose elements are to be copied.

start Data field of the type `JCI-size`
Position of the first element in the Java array which is to be transferred (beginning with 1).

num Data field of the type `JCI-size`
Maximum number of elements which are to be transferred.
After the call, *num* contains the number of elements which were actually transferred.

r A structure of the form `MethodRes`
Return value and error information (see section "Arguments and event values of Java methods").
`ResType` must be set in accordance with the COBOL data type of the table elements. Neither `RES-OBJECT` nor `RES-ANUM-STRING` nor `RES-NAT-STRING` is permissible.
The address of the COBOL table to which the elements are to be copied is always transferred in the `ResValAddr` field, regardless of the data type.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

aObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj is not an array object.

`JCI-RET-EINDAOB`

num is less than 0 or *start* is less than 1 or greater than the number of elements in the array.

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-ERESTYPE`

The value of the `ResType` field is invalid.

`JCI-RET-ERESCONV`

An error occurred while the table elements were being converted.
The `ResErrIndex` field contains the number of the COBOL table element (beginning with 1), the `ResErrCode` field a more precise error code.
All elements up to the faulty element are transferred; all subsequent fields of the COBOL table remain unchanged.

Notes

A maximum of (number of array elements - *start* + 1) or *num* elements are transferred.

JCI_SetArray

This function copies a COBOL table to the elements of a Java array.It is equivalent to the JNI functions `Set<PrimitiveType>ArrayRegion`.

Call

`CALL 'JCI_SetArray' USING` *aObj start num arg res*

aObj Array object

start Start position

num Number

arg Argument description

r Result description

Arguments

aObj Data field of the type `JCI-object`
Array object whose elements are to be set.

start Data field of the type `JCI-size`
Position of the first element in the Java array which is to be overwritten (beginning with 1).

num Data field of the type `JCI-size`
Maximum number of elements which are to be transferred.
After the call, *num* contains the number of elements which were actually transferred, and in the case of an error 0.

arg A structure of the form `MethodArg`
Description of the array elements.
Only the `ArgType(1)` and `ArgValAddr(1)` fields are required.
`ArgType(1)` must be set in accordance with the COBOL data type of the table elements. Neither `ARG-OBJECT` nor `ARG-ANUM-STRING` nor `ARG-NAT-STRING` is permissible.
The address of the COBOL table from which the elements are to be copied is always transferred in the `ArgValAddr(1)` field, regardless of the data type.

r A structure of the form `MethodRes`
Error information (output only).

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

`JCI-RET-ENOVM`

No Java VM has been started.

`JCI-RET-ENULLOBJ`

aObj is `JCI-NULL`.

`JCI-RET-EARGUMENT`

aObj is not an array object.

`JCI-RET-EARGVERS`

The statically generated version number in *elem* is invalid (possibly overwritten).

`JCI-RET-ERESVERS`

The statically generated version number in *res* is invalid (possibly overwritten).

`JCI-RET-EARGTYPE`

The value of the `ArgType` field is invalid.

`JCI-RET-EARGCONV`

An error occurred while the table elements were being converted.
The `ResErrIndex` field contains the number of the COBOL table element (beginning with 1), the `ResErrCode` field a more precise error code.
If a conversion error occurs in an element, no transfer takes place, i.e. all fields of the Java array remain unchanged.

`JCI-RET-EINDAOB`

num is less than 0 or *start* is less than 1 or greater than the number of elements in the array.

Notes

A maximum of (number of array elements - *start* + 1) or *num* elements are transferred.

Exceptions

This section describes the JCI functions required to process Java exceptions.

- JCI_ExceptionCheck
- JCI_ExceptionOccurred
- JCI_ExceptionDescribe
- JCI_ExceptionClear

JCI_ExceptionCheck

This function checks whether a pending exception exists.
It is equivalent to the JNI function `ExceptionCheck`.

Call

```
CALL 'JCI_ExceptionCheck'
```

Return value (RETURN-CODE)

`JCI-RET-TRUE`

An exception is pending.

`JCI-RET-FALSE`

No exception is pending.

Notes

If the function is called without the Java VM being started, `JCI-RET-FALSE` is returned.

JCI_ExceptionOccurred

This function checks whether a pending exception exists, and returns the associated exception object.
It is equivalent to the JNI function `ExceptionOccurred`.

Call

```
CALL 'JCI_ExceptionOccurred' USING eObj
```

eObj Exception object

Arguments

eObj Data field of the type `JCI-object`
Reference to the pending exception object.
If no object was created, `JCI-NULL` is returned.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

Notes

If the function is called without the Java VM being started, `JCI-NULL` and `JCI-RET-OK` are returned.

JCI_ExceptionDescribe

This function outputs information in English about a pending exception to `stderr`.It is equivalent to the JNI function `ExceptionDescribe`.

Call

```
CALL 'JCI_ExceptionDescribe'
```

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

Notes

This function may also be called when no Java VM has been started.

If the VM has not been started or no exception is pending, the output does not take place.If the program was started from the BS2000 command line interface, the output is directed to SYSOUT.

JCI_ExceptionClear

This function removes any pending exception.
It is equivalent to the JNI function `ExceptionClear`.

Call

```
CALL 'JCI_ExceptionClear'
```

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

Notes

This function may also be called when no Java VM has been started or no exception is pending.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
...
01 className.
02 PIC S9(9) USAGE COMP-5 VALUE 30.
02 PIC X(30) VALUE 'hello'.
...
01 classObj  TYPE JCI-object.
...
PROCEDURE DIVISION.
...
CALL 'JCI_FindClass' USING className classObj
IF RETURN-CODE NOT = JCI-RET-OK
CALL 'JCI_ExceptionCheck'
IF RETURN-CODE = JCI-RET-TRUE
CALL 'JCI_ExceptionDescribe'
CALL 'JCI_ExceptionClear'
END-IF
ELSE
...
END-IF.
...
```

If the *hello* class does not exist, the output looks roughly as follows:

```
Exception in thread "main" java.lang.NoClassDefFoundError: hello
Caused by: java.lang.ClassNotFoundException: hello
at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:332)
at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
```

Other functions

This section described all JCI functions for which there are no equivalent JNI functions.

- JCI_GetVersion
- JCI_GetErrorInformation

JCI_GetVersion

This function returns the version of the Java COBOL interface module.

Call

```
CALL 'JCI_GetVersion' USING vers
```

vers Version

Arguments

vers Data field of the type `JCI-int`
Data field to which the version number of the JCI is to be transferred.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

Notes

The version of the COPY elements used in the COBOL application is defined in *JCI-CONST* as `JCI-interface-version`. This may not be greater than the version returned by `JCI_GetVersion`.

JCI_GetErrorInformation

This function returns more precise error information.

Call

```
CALL 'JCI_GetErrorInformation' USING eInf
```

eInf Error information

Arguments

eInf A structure of the type `Cobvar`
Structure to which the at most 256-character-long error information of the JCI is to be transferred.
The transfer occupies at most the length of the length field. If this is less than or equal to 0 or no error information exists, it is set to 0, and no transfer takes place.
If no error information is available, the length field in *eInf* is set to 0, and the text area
remains unchanged.

Return value (RETURN-CODE)

`JCI-RET-OK`

The call was successful.

Notes

This function can always be called, even when the call of JCI functions is described as invalid, as in, for instance, section "Exceptions".

In the event of an error, more precise information on this error is stored in a joint field by all JCI functions which can supply an error return code. If no error occurs, the field is deleted. As a result, only the information of the function called most recently is ever available. The text is in English and only intended to be displayed to the user.

After the `JCI_GetErrorInformation` function has been called, the error information is no longer available for further calls.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 eInf.
02 len PIC S9(9) USAGE COMP-5 SYNC VALUE 256.
02 txt PIC X(256).
...
PROCEDURE DIVISION.
...
CALL 'JCI_GetErrorInformation ' USING eInf
IF len IN eInf > 0
DISPLAY txt IN eInf(1:len IN eInf) UPON T
END-IF
...
```

Examples

In this example, all Java sources are available in the `/myhome/jcitest` directory.

JENV is installed in the `/myjava` directory under the `$MYJAVA` ID.

Java class

The following class is defined in the `Hello.java` file:

```
class Hello {
public static void hello(String arg)
{
System.out.println(">> Hello " + arg + "!");
}
}
```

Compiling the Java code

The Java class defined above can now be simply compiled using the command

```
javac /myhome/jcitest/Hello.java
```

The call generates the `Hello.class` file in the `/myhome/jcitest` directory.

Calling

```
javap -s -cp /myhome/jcitest Hello
```

returns, among other things, the signature of the `hello` method:

```
public static void hello(java.lang.String);
descriptor: (Ljava/lang/String;)V
```

COBOL program

The COBOL program *HELLO* is implemented in the `Hello.cob` file as follows:

```
>>SOURCE FREE
>>IMP LISTING-OPTIONS MERGE-DIAGNOSTICS
ID DIVISION.
PROGRAM-ID. HELLO.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
ARGUMENT-NUMBER IS ARGNUM
ARGUMENT-VALUE IS ARGVAL
TERMINAL IS T.
DATA DIVISION.
WORKING-STORAGE SECTION.
*> Types and constants
COPY JCI-TYPEDEFS.
01 JCIConstants.
COPY JCI-CONST.
*> Constant strings
01 optCP.
05 PIC S9(9) COMP-5 VALUE 80.
05 PIC X(80) VALUE '-Djava.class.path=.:/myhome/jcitest'.
01 OptEnc.
05 PIC S9(9) COMP-5 VALUE 40.
05 PIC X(40) VALUE '-Dfile.encoding=OSD_EBCDIC_DF04_15'.
01 className.
05 PIC S9(9) COMP-5 VALUE 30.
05 PIC X(30) VALUE 'Hello'.
01 methodName.
05 PIC S9(9) COMP-5 VALUE 30.
05 PIC X(30) VALUE 'hello'.
01 methodSig.
05 PIC S9(9) COMP-5 VALUE 80.
05 PIC X(80) VALUE '(Ljava/lang/String;)V'.
LOCAL-STORAGE SECTION.
*> JCI structures
01 JVMOptions.
COPY JCI-VMOPT REPLACING == <max-options> == BY 2.
01 MethodArgs.
COPY JCI-METHODARGS REPLACING == <max-arguments> == BY 4.
01 MethodRes.
COPY JCI-METHODRES.
*> String structures
01 myName.
05 len PIC S9(9) COMP-5 VALUE 30.
05 txt PIC X(30).
*> Objects and handles
01 classObj TYPE JCI-object.
01 methodId TYPE JCI-handle.
*> Error handling
01 ErrIdent    PIC X(10) VALUE SPACE.
01 RetcodeSave PIC S9(9) COMP-5 VALUE 0.
01 errorInf.
05 len PIC S9(9) COMP-5 VALUE 300.
05 txt PIC X(300).
PROCEDURE DIVISION.
```

```
>>CALL-CONVENTION ILCS-SET-RETURN-CODE
*>
*> get name from terminal
*>
DISPLAY ">> Please enter name" UPON T
ACCEPT txt IN myName FROM T
*>
*> Prepare VM options
*>
MOVE 2 TO VMOptnum.
SET IGNORE-UNRECOGNIZED TO FALSE.
SET VMOptVstring(1) TO ADDRESS OF optCP
SET VMOptVstring(2) TO ADDRESS OF optEnc
*>
*> Create the Java VM
*>
CALL 'JCI_CreateJavaVM' USING JVMOptions
IF RETURN-CODE NOT = JCI-RET-OK
MOVE 'CreateVM' TO ErrIdent
GO TO ERROR-EXIT
END-IF.
*>
*> Get class Hello
*>
CALL 'JCI_FindClass' USING className classObj
IF RETURN-CODE NOT = JCI-RET-OK
MOVE 'FindClass' TO ErrIdent
GO TO ERROR-EXIT
END-IF.
*>
*> Get method hello
*>
CALL 'JCI_GetStaticMethodID' USING classObj methodName
methodSig methodId
IF RETURN-CODE NOT = JCI-RET-OK
MOVE 'GetMethod' TO ErrIdent
GO TO ERROR-EXIT
END-IF.
*>
*> Call Java method
*>
MOVE 1 TO CallArgNum
SET RES-VOID TO TRUE
SET ARG-ANUM-STRING(1) IGNORE-TRAILING-SPACES(1) TO TRUE
SET ArgValAddr(1) TO ADDRESS OF myName
CALL 'JCI_CallStaticMethod' USING classObj methodId MethodArgs MethodRes
IF RETURN-CODE NOT = JCI-RET-OK
MOVE 'CallMeth' TO ErrIdent
GO TO ERROR-EXIT
END-IF.
*>
*> Destroy Java VM
*>
CALL 'JCI_DestroyJavaVM'
IF RETURN-CODE NOT = JCI-RET-OK
MOVE 'DestroyVM' TO ErrIdent
GO TO ERROR-EXIT
END-IF.
GOBACK.
```

```
*>
*> Error exit
*>
ERROR-EXIT.
MOVE RETURN-CODE TO RetcodeSave
CALL 'JCI_GetErrorInformation' USING errorInf
IF len IN errorInf > 0
DISPLAY 'Message from ' ErrIdent ': "' txt IN errorInf(1:len IN errorInf) '"'
UPON T
END-IF
CALL 'JCI_ExceptionCheck'
IF RETURN-CODE = JCI-RET-TRUE
CALL 'JCI_ExceptionDescribe'
CALL 'JCI_ExceptionClear'
END-IF
CALL 'JCI_DestroyJavaVM'
MOVE RetcodeSave  TO RETURN-CODE
GOBACK.
END PROGRAM HELLO.
```

Compiling the COBOL program in POSIX

In this example, the COBOL source program resides in the POSIX directory `/myhome/jcitest`.

The following commands are needed to compile the COBOL program *HELLO*:

```
export COBLIB='/myjava/include'
cobol -c -C PERMIT-STANDARD-DEVIATION=YES \
/myhome/jcitest/Hello.cob
```

The object file `Hello.o` is available as the result.

Linking the COBOL program in POSIX

When linking the application, it must be remembered that the runtime routines for the languages C/C++ and COBOL are linked from the Java runtime library and not from the CRTE.

The application can be linked with the following commands:

```
export BLSLIB00='$MYJAVA.SYSLNK.JENV.170.GREEN-JAVA'
cobol -M HELLO -o Hello Hello.o -l BLSLIB
```

Processing of the COBOL program in POSIX

As the standard installation path of JENV is not to be used for this example, the environment variable `JAVA_HOME` must be set before calling the program.

The call and processing are then as follows:

```
export JAVA_HOME=/myjava
Hello
>> Please enter name
Susanne
>> Hello Susanne!
```

Compiling the COBOL program under the BS2000 command line interface

In this example, the COBOL source program resides in the LMS library `SRC.LIB`, but the JCI-COPY elements in the POSIX directory `/myjava/include`.

Consequently the following commands are required for compilation:

```
/DECL-VAR SYSIOL-COBLIB,INIT='*POSIX(/myjava/include)',
SCOPE=*TASK
/START-COBOL2-COMP SO=*LIB(SRC.LIB,HELLO.COB),
SOURCE-PROPERTIES=*PAR(ST-DEV=*YES),
COMPILER-ACTION=*MOD-GEN(MOD-FORM=*LLM),
MODULE-OUTPUT=*LIB(MOD.LIB,HELLO),
RUNTIME-OPTIONS=*PARAMETERS(ENABLE-UFS-ACCESS=*YES)
```

Linking the COBOL program under the BS2000 command line interface

In addition to functions and CRTE from the Java runtime library, the POSIX options must also be linked:

```
/START-BINDER
//START-LLM-CREATION HELLO
//INCLUDE LIB=MOD.LIB,ELEM=HELLO
//INCLUDE LIB=$.SYSLNK.CRTE.POSIX
//RESOLVE LIB=$MYJAVA.SYSLNK.JENV.170.GREEN-JAVA
//SAVE-LLM LIB=LLM.LIB,ELEM=HELLO
//END
```

Processing of the COBOL program under the BS2000 command line interface

Before the application is started, the POSIX environment must be initialized for processing. The COBOL runtime system then behaves as if it had been started under the POSIX shell (see „COBOL2000 (BS2000) User Manual“ [5]).

After the application has terminated, the POSIX environment must on all accounts be reset by calling the *DELETE* procedure. Otherwise the environment is set incorrectly for further compilations runs.

The call and processing are then as follows under the ID $MYHOME:

```
/CALL-PROCEDURE *LIB($MYJAVA.SYSPRC.JENV.170,INITIALIZE),
     (PWD='myhome/work',JAVA-HOME='/myjava')
/START-PROGRAM *MODULE(LIBRARY=LLM.LIB,ELEMENT=HELLO,
PROGRAM-MODE=ANY,RUN-MODE=*ADVANCED(SHARE-SCOPE=*NONE))
%  BLS0523 ELEMENT 'HELLO', VERSION '@', TYPE 'L' FROM LIBRARY
':LUNB:$MYHOME.LLM.LIB' IN PROCESS
%  BLS0524 LLM 'HELLO', VERSION ' ' OF '2024-02-13 15:17:10' LOADED
>> Please enter name
Susanne
>> Hello Susanne!
/CALL-PROCEDURE *LIB($MYJAVA.SYSPRC.JENV.170,DELETE)
```

Commands for BS2000

The tools belonging to the JDK are described in “JDK Tools and Utilities" [11]. JENV supports all the tools listed there for Solaris with the following exceptions:

- Monitoring und Management Tools *jps*, *jstat*, *jstatd*
- Troubleshooting Tools *jcmd*, *jinfo*, *jhat*, *jmap*, *jsadebugd*, *jstack*
- Scripting Tool *jrunscript*.

This chapter only includes the commands which differ from the description in “JDK Tools and Utilities" [11], namely:

- The mk_shobj and pr_shobj commands
  JENV offers these is in addition to supporting the shared object description files.
- The java command
  Its options differ from those described for Solaris.
- The native2ascii command
  This is described in more detail because of its greater importance in the EBCDIC environment.
- The jconsole, jdb commands

mk_shobj

The *mk_shobj* command creates and processes descriptive files for shared objects.

Syntax

mk_shobj [Options ...] Filename

Options ...

One or more command line options, separated by spaces.

Filename

Description file for shared objects in the POSIX file system which *mk_shobj* is to create.

Description

The *mk_shobj* command creates and processes descriptive files for shared objects in the POSIX file system. These descriptive files are evaluated by the Java interpreter if native methods are loaded (methods *loadLibrary()* or *load()* of the classes *runtime* and *system*).

The names of the descriptive files must be put together in such a way that they can be found by the VM using the search procedure described under the use of shared objects from Java, or in other words, beginning with the prefix *lib* and ending with the suffix *.so*.

Options

-?	Outputs help information for the command.
-l *lib*	Specifies the PLAM library (in BS2000) in which the LLM to be loaded is located.
-o *userid*	BS2000 user ID, under which the PLAM library *lib* is installed. Where "**.**" stands for the current user ID and "*$*" stands for the system ID and the form *%name* indicates, that the user ID to be used at runtime can be taken from the environment variable *name* and any other specification stands for the names of user IDs. Default: current user ID
-m *modulename*	Specification of the module which is to be loaded. This option can be specified several times, and then all specified modules can be loaded dynamically. The module name may not be longer than 32 characters.
-n *filename*	Specifies the required shared objects (descriptive file). The shared object specified here is loaded before the primary shared object. This option can be specified several times, and all the required shared objects are loaded before the current shared object.
-u	The specified descriptive file must exist and is updated using the specified information. This can be used, for example, to subsequently modify the user ID. If the *-u* option is not specified, the descriptive file is generated again.

-f cpp If the shared object has been implemented in C++, this flag must be set to ensure that the required runtime libraries can be loaded and initialized.

-d If this flag is set, the module is loaded in the default context *LOCAL#DEFAULT*.

-c ctxt The module is loaded in the specified context.

Example

The command

```
mk_shobj -l syslnk.hello -m helloworld libhello.so
```

creates the file *libhello.so* in the current file directory of the POSIX file system, and specifies that when *hello* is loaded the module *helloworld* is to be dynamically loaded from the PLAM library *syslnk.hello* of the current user ID. When *hello* is loaded the Java interpreter expands *loadLibrary(hello)* to read *libhello.so*.

pr_shobj

The command *pr_shobj* outputs the contents of a shared object descriptive file.

Syntax

pr_shobj Filename

Filename

Descriptive file for which the contents is to be output.

Description

The command *pr_shobj* outputs the contents of a shared object descriptive file to *stdout*.

Example

```
pr_shobj libhello.so
```

Output:

```
Library: syslnk.hello
UserID : .
Module : helloworld
```

java

Options for selecting the HotSpot™ VM type

-client

The HotSpot™ client VM is used. This VM optimizes the generated object code for short-running programs (default).

-server

The option is not supported.

-d32
-d64

The options are not supported.

Options for selecting the HSI variant

-s390 The S390 variant of JENV is used (if available). This option is useful only if both the S390 variant and the X86 variant of JENV are installed on one system and you want to explicitly select one of them for execution.
This option overrides any specification in the environment variable *JENV_SYSHSI* (see the chapter "Environment variables").

The variant that matches your system is used by default, i.e. if no value has been assigned to the environment variable *JENV_SYSHSI* either.

-x86 The X86 variant of JENV is used (if available). This option is useful only if both the S390 variant and the X86 variant of JENV are installed on one SQ system and you want to explicitly select one of them for execution.
This option overrides any specification in the environment variable *JENV_SYSHSI* (see the chapter "Environment variables").

The variant that matches your system is used by default, i.e. if no value has been assigned to the environment variable *JENV_SYSHSI* either.

Non-standard options

-Xmaxjitcodesize *size*

In contrast to the original description, the cache size is specified without an equals sign, e.g.:

```
-Xmaxjitcodesize48m
```

Controlling the Java heap memory

The following options allow the user to control heap expansion or reduction. Since the standard settings for heap expansion are suitable for most applications, it is not necessary to use these options in most situations. You should only use them if you understand the effects of the options on the applications concerned. Deliberately setting these options can just as easily adversely affect system performance as improve it.

In BS2000 the maximum size of heap memory is always requested by the system right from the start and always remains reserved in this size. Option *-Xms* merely controls how much of the heap memory is to be used currently. The smaller this area is, the faster garbage collection proceeds since only the area currently being used must be searched. On the other hand it can be that garbage collection has to be called unnecessarily frequently if there is only a small amount of space for new objects in the currently used area.

Minimum and default values which differ from the original description are defined for these options:

-Xsssize

Minimum value: 512K
Default value: 1M

-Xmssize

Minimum value: 1M
Default value: 3.5M

-Xmxsize

Minimum value: 1M
Default value: 64M

> The specified value is rounded off the next multiple of 2M.

native2ascii

This command converts a file from any code set into the US-ASCII (7 bit ASCII) code set.

Syntax

native2ascii [Options ...] [input file[output file]]

Options ...

One or more command line options, separated by blanks.

Input file

File which is to be converted. If *input file* is not specified, the input is expected on *stdin*.

Output file

Destination file for the conversion. If *output file* is not specified, output is on *stdout*.

output file and *input file* may also be the same.

Description

The *native2ascii* command converts text available in any code set (e.g OSD_EBCDIC_DF04_1) into US-ASCII (7-bit ASCII); non-printable characters in ASCII are printed in portable Unicode (\uxxxx). Conversion in the reverse direction is also possible. Portable Unicode is interpreted, for example, when property files are loaded.

If property files are stored in JAR archives, they must be present in code set ISO8859-1. The same applies to manifest files or other texts. This command makes it possible to prepare the corresponding files for this because the full US-ASCII code set is included in ISO8859-1.

As of JENV V1.4A policy files, which are used by the standard policy implementation, must be encoded in the UTF-8 codeset. native2ascii can be used for the conversion, as the UTF-8 codeset concurs with the first 127 characters of the US-ASCII codeset.

Options

-encoding *character set*

Specifies the character set from which or into which the command converts. If the option is not specified, the value set via the system property *file.encoding* is used. Since JENV V1.2A the default value for this system property is OSD_EBCDIC_DF04_1. Permitted values can be found in the Specification entitled “Supported Encodings” [14]. The character sets additionally supported since JENV V1.2A are described in section "Code sets".

-reverse

The conversion is performed in the reverse direction: A text which is present in character set US-ASCII is converted into the character set specified by *-encoding*. Any portable Unicode representations in the input (\uxxxx) are interpreted when this is done. Characters which cannot be shown in the output character set are output there in portable Unicode representation.

-J *javaoption*

Passes *javaoption* to the JVM, where *javaoption* is one of the options described for *java*.

jconsole

In BS2000, the use of a process ID (pid) is not supported when setting up a connection with a Java application.

jdb

jdb does not work when the default input is connected with a BS2000 block terminal, in which case *jdb* is terminated with an error message.

Appendix: Compatibility with earlier versions and migration

JENV V17.0A is an implementation of the “Java Platform, Standard Edition” (Java SE™) for BS2000 based on OpenJDK 17.

With OpenJDK 9, a module concept was introduced in Java. Due to this module concept, neither source nor binary compatibility with versions < V9.0A can be guaranteed.

In addition, a distinction must be made between incompatibilities in the OpenJDK implementation and those in the BS2000 portation.

OpenJDK-specific Incompatibilities

A major change in OpenJDK 17 is the deletion of tools, modules and APIs such as <??? pack200, unpack200, rmic, rmid>.

A detailed description of the incompatibilities between OpenJDK 17 and earlier versions as well as a detailed migration guide is available at the link https://docs.oracle.com/en/java/javase/17/migrate.

BS2000-specific Incompatibilities

The packages com.fsc.java.bs2000, com.fsc.java.io and com.fsc.jrio are no longer supported in JENV V11.0A.
Since JENV V7.0A the functionality has been replaced by the corresponding com.fujitsu.ts. packages.
The affected Java sources have been marked in JENV V9.0 as "deprecated and marked for removal" by a tightening of the @Deprecated annotation.

Related publications

You will find the manuals on the internet at https://bs2manuals.ts.fujitsu.com.

[1] **POSIX**
POSIX, Basics for Users and Systems Administrators
User manual

[2] **CRTE**
C Library functions for POSIX applications
Reference Manual

[3] **CRTE**
Common RunTime Environment
User Manual

[4] **C/C++**
C/C++-Compiler
User Manual

[5] **COBOL2000**
COBOL-Compiler
User Manual

[6] **COBOL2000**
COBOL-Compiler
Reference Manual

[7] **SDF-P**
Programming in the Command Language
User Guide

[8] **BS2000 OSD/BC**
Introductory Guide to DMS
User Guide

Texts for Java

You will find the following texts in the internet, mainly on the Web pages of Oracle America Inc.:

All links given below were valid when going to press. However, no guarantee can be given for their future validity. The information in this manual always takes precedence over information in the internet.

[9] Java Platform Standard Edition 17 Documentation
https://docs.oracle.com/en/java/javase/17/

[10] The Java™ Language and Virtual Machine Specifications
https://docs.oracle.com/javase/specs/

[11] JDK Tools and Utilities
https://docs.oracle.com/en/java/javase/17/tools/

[12] The Java™ Platform, Standard Edition 17 API Specification
https://docs.oracle.com/en/java/javase/17/docs/api/

[13] Java™ Native Interface
https://docs.oracle.com/en/java/javase/17/docs/specs/jni/intro.html

[14] Supported Encodings
https://docs.oracle.com/en/java/javase/17/intl/supported-encodings.html

Further literature

[15] Erich Gamma
Richard Helm

Ralph E. Johnson

John Vlissides

Design Pattern
Addison Wesley 1994

[16] Technical Standard
X/Open System Interface (XSI) Specification
System Interfaces and Headers, Issue 4, Version 2